Reimagining the Programming Experience

By

MEHRDAD AFSHARI

B.S. (University of Tehran) 2010
M.S. (University of California, Davis) 2012

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Zhendong Su, Chair

Premkumar Devanbu

Todd Green

Committee in Charge

2016

Contents

Mehrdad Afshari
June 2016
Computer Science

Reimagining the Programming Experience

Abstract

A key mission of computer science is to enable people realize their creative ideas as naturally and painlessly as possible. Software engineering is at the center of this mission — software technologies enable reification of ideas into working systems. As computers become ubiquitous, both in availability and the aspects of human lives they touch, the quantity and diversity of ideas also rapidly grow. Our programming systems and technologies need to evolve to make this reification process — *transforming ideas into software* — as quick and accessible as possible.

This dissertation advocates and highlights the "transforming ideas to software" mission as a moonshot for software engineering research. This is a long-term direction for the community, and there is no silver bullet that can get us there. To make this mission a reality, as a community, we need to improve the status quo across many dimensions. Thus, the second goal is to outline a number of directions to modernize our contemporary programming technologies for decades to come, describe work that has been undertaken along those vectors, and pinpoint critical challenges.

A key contribution towards that direction is introducing the *prorogued programming paradigm*, a new paradigm more closely aligned with a programmer's thought process. A prorogued programming language (PPL) supports three basic principles: 1) *proroguing concerns*[1]: the ability to defer a concern, to focus on and finish the current concern; 2) *hybrid computation*: the ability to involve the programmer as an integral part of computation; and 3) *executable refinement*: the ability to execute any intermediate program refinements. Working in a PPL, the programmer can run and experiment with an *incomplete* program, and gradually and iteratively reify the missing parts while catching design and implementation mistakes early. We describe the prorogued programming paradigm, our design and realization of the paradigm using Prorogued C#, our extension to C#, and demonstrate its utility through a few use cases.

Furthermore, we discuss *Live Symbols*, a technique to invigorate an integrated development environment by making identifiers not only meaningful to the compiler, but also direct the programming environment to interact with the user in a context-specific way. We illustrate leveraging identifiers as hooks not just for gluing pieces of a program together, but as terminals to interact with the programmer, program documentation, knowledge external to the program, the programming environment, and serve as tools to manipulate the program itself in specialized ways.

Finally, we explore a powerful mechanism to create *white-box abstractions* through program refinement, to encourage flattening of abstraction hierarchies, and to make it easier to manipulate and customize software components when necessary.

[1]A *concern* is any piece of interest or focus in a program [**Dijkstra, 1974**].

To Mom & Dad.

Acknowledgments

First, I thank my parents for their unconditional support and belief in me and letting me follow my heart.

Most directly influential in this work, I thank my PhD adviser, Prof. Zhendong Su, for taking a gamble on me when I was nothing but a research "noob," and giving me the opportunity to join his group, and for his continued support and guidance since. I was lucky to do a PhD with him and I genuinely cannot be more thankful. Thank you, thank you, thank you, Zhendong! I am going to miss our weekly discussions nerding out about technology, philosophy, programming languages, the future, and the world at large.

I also thank my amazing dissertation committee members, Prof. Prem Devanbu, and Prof. Todd Green, for their inspiration, thoughts, discussion, feedback, and comments on this work and throughout my PhD years.

I thank my colleagues, Prof. Earl Barr, and Dr. Vu Le for their insights, discussions, great work, and co-authoring papers with me. In particular, Earl has been instrumental in shaping the prorogued programming paradigm, discussed in detail in chapter 2. I also thank Prof. Phillip Rogaway for suggesting the name "prorogue" to us and for being an all-around inspiration in the Department of Computer Science.

Last, but not least, I thank the people of the beautiful State of California for supporting our beloved University of California.

CHAPTER 1

Introduction

A key mission of computer science is to enable people realize their creative ideas as naturally and painlessly as possible. Software engineering is at the center of this mission — software technologies enable reification of ideas into working systems. As computers become ubiquitous, both in availability and the aspects of human lives they touch, the quantity and diversity of ideas also rapidly grow. Our programming systems and technologies need to evolve to make this reification process — *transforming ideas into software* — as quick and accessible as possible.

The goal of this chapter is twofold. First, it advocates and highlights the "transforming ideas to software" mission as a moonshot for software engineering research. This is a long-term direction for the community, and there is no silver bullet that can get us there. To make this mission a reality, as a community, we need to improve the status quo across many dimensions. Thus, the second goal is to outline a number of directions to modernize our contemporary programming technologies for decades to come, describe work that has been undertaken along those vectors, and pinpoint critical challenges.

1.1. Toward Rapid Transformation of Ideas Into Software

There has been no shortage of creative technological ideas, but few have been realized — it is a daunting task to transform an idea into a working prototype. Indeed, software engineering — the process of expressing and refining ideas in a programming language — has been regarded one of the most challenging human endeavors. Programming innovations, such as procedural abstraction and object orientation, have helped increase programmer productivity. However, we still build software essentially the same way as we did decades ago. As a community, we should rethink and redesign methodologies and techniques for programming to make software development more natural and painless to help people realize their creative ideas.

We believe that *transforming ideas into software (TIIS)* should be identified as a long-term, catalytic mission for the software engineering community. Decades of research and development have led to better languages, methodologies, tools, environments, and processes. However, it is fair to say that most have been incremental improvements and do not promise significant advances demanded for the mission. Identifying and highlighting the TIIS mission can help unite the community and clarify important research focuses to achieve significant innovations.

The TIIS mission requires a multi-faceted approach, which we organize around several key principles:

- *Quick experimentation*: to provide developers with immediate feedback on their code modifications and allow them to experiment with incomplete systems;
- *Programming knowledge reuse*: to allow developers quick access to the vast amount of accumulated programming knowledge and wisdom;
- *Proactive programming assistant*: to monitor the developers' actions and proactively feed them relevant information about the program; and
- *Intelligent, conversational interfaces*: to provide alternative interfaces that allow developers to express their intentions and conduct interactive exchanges with the system.

The two core questions in programming are "What" and "How": (1) "What" specifies the intention, and (2) "How" concerns the solution. The first three principles center around the "How" question, while the last principle the "What". Next, we discuss the above principles, and pinpoint specific research problems and challenges.

1.2. Directions and Challenges

The vision for quick transformation of ideas into software is broad, and advances in a number of directions are necessary and can move the state of affairs forward. We discuss several directions that we have identified that can be influential toward our goal. We have done early work along some of these directions and hope the community as a whole can help accelerate the progress toward improving programming and in particular, the pace of concretizing ideas.

1.2.1. Quick Experimentation. Live programming has gained momentum following Bret Victor's presentation [**Victor, 2012**], in which he highlighted the importance of immediate connection between the idea and observing its effect, not just as a catalyst, but as an enabler, in an effective creative process. Since then, several live programming environments, *e.g.* Xcode [**Apple Inc., 2014**] (via its Playground feature), LightTable [**Kodowa, Inc., 2012**], and LogicBlox [**Green et al., 2015**] have been influenced by this principle.

Prorogued programming [**Afshari et al., 2012**] is a programming paradigm that explicitly deals with the issue of quick experimentation. It is focused on liberating the programmer from having to deal with programming concerns that are necessary to get a partial, incomplete, program running and meaningfully experiment with it and observe its behavior. It does so by providing the ability to annotate function calls or type instantiations with a special keyword, `prorogue`. The prorogue keyword acts as a hint for the compiler to let it know that the implementation for the particular method being called is unavailable. At runtime, after a prorogued call is executed, a lazy *future* object is returned in lieu of the return value and the program execution continues. Later, if the value of that object is consulted during the program execution, the user will be asked to provide a concrete return value for the call interactively, while presenting him the actual arguments in that specific invocation. The user interaction will then be recorded and persisted for the rest of the program execution and for subsequent runs, so that the program can be run and experimented with in spite of the unimplemented method body.

In effect, prorogued programming aids quick experimentation and top-down design by letting the programmer freely rearrange his workflow as he sees fit, rather than having to follow an order imposed by the toolchain they are using.

More interestingly, through *hybrid computation*, prorogued calls can act as hooks to glue a program written in an imperative textual programming language into more domain-specific programming systems that would capture the human intent much better and in a more concise fashion for particular purposes. The other end of hybrid computation does not even have to be an imperative program. It can be a machine learning model that is trained to provide the desired function that would be hard to express the host language. Alternatively, it can be an interactive system that computes the desired output through some user interaction. It is possible to have a hybrid computation engine that is mostly similar to mainstream textual programming languages, except it is much *softer* when it comes to interpreting programmer intent, leaving room for the compiler to make educated guesses and at the same time be more lenient to programmer mistakes, at the expense of precision.

1.2.2. Programming Knowledge Reuse. Software is rarely written from scratch. Rather, programs are generally composed of smaller pieces. That makes software engineering activity largely a system integration process. Software engineers build more complex abstractions out of simpler ones and that lets them build increasingly sophisticated systems. While seeing the effects of a program live helps, the question remains that given that there are vast amounts of source code available on the Internet, should we move from writing new code to casting programming as a search problem?

The programming knowledge publicly available today comes in various forms, such as questions and answers on Stack Overflow [**Stack Overflow Inc., 2008**], sometimes including code snippets as well as answers, or through publicly accessible code repositories such as the ones hosted on GitHub [**GitHub Inc., 2008**].

Commercial software development endeavors also collect internal data about their development process, including the version history of the code base, data about bugs and defects, and free-form knowledge in form of comments written on the code review tool, wikis, and sometimes in other forms, like tracking the time the programmers spend on various tasks, storing the search queries they perform [**Sadowski et al., 2015**], or looking at their behavior within the development environment.

In software engineering practice, major effort is expended to integrate various systems and assemble a program from building blocks. Given the large amount of code available, it is conceivable that what a programmer plans to write is already written and available in some shape or form. Effective code search can help the programmer discover the existing functionality from existing code bases import it in the code being written [**Microsoft Research, 2015**].

With a mechanism to locate pieces of functionality through existing APIs or code snippets mined from the Internet, we need to be able to run the resulting *mashup* consisting of the different pieces and quickly experiment with them. A programming paradigm like prorogued programming is well-suited for this task. Proroguing programming concerns not only helps in piecing together the building blocks of functionality discovered in the existing code bases, but also provides a way to effectively insert *holes* in the program, which can be filled later. Filling these holes can be done through traditional implementation, *i.e.*

writing a body for the unimplemented method, or it can be done through more innovative means, like acting as a signal in addition to the search query and helping the search engine know the context in which the code snippet being searched for is going to be live in. In addition to providing that context, the input/output examples persisted during runtime invocation of prorogued calls are a great source of input for an I/O-based code search engine and act as a final filter for validation of code found by a simple keyword based code search engine.

Collecting data about the programmer's actions is helpful in other ways as well. By looking at the actions the programmer performs within their development environment, for example, it is possible to predict what they intend to accomplish and propose shortcuts to achieve what they are aiming for more efficiently [**Gu et al., 2014, Murphy-Hill et al., 2012**], thereby educating the programmer and making them more effective in the future. Obviously, this can help the IDE designer improve the development environment and simplify its user interface as well.

Reusing programming knowledge is also beneficial in activities beyond writing code. For instance, we are able to leverage debugging knowledge accumulated over the previous debugging sessions to automatically help the programmer fix the new, similar, issues [**Gu et al., 2012**]. One way that has been accomplished is by collecting and matching the program traces that exhibit buggy behavior and pattern matching new traces against the ones in the bug database, revealing information about the nature of the bug and how it was previously fixed, potentially helping the programmer understand and fix the new issue.

1.2.3. Proactive Programming Assistant. Many programming analysis tools have been developed. In practice, program analyses are primarily left to compile time and later. We believe that we should surface as much relevant information as possible to the programmer as soon as possible. Programming tools should capture runtime data and run background static or dynamic analysis while the code is being written, and guide the programmer throughout the coding process. With the popularity of compiler-as-a-library solutions like libclang [**The Clang Team, 2007**] or Roslyn [**.NET Foundation, 2015**], we are already seeing this shift accelerating. Our editors are indeed becoming more proactive in issuing compiler warnings and providing safe refactoring tools.

That said, in particular, the potential for capturing dynamic information and surfacing it in useful ways while coding remains largely untapped. Among other things, the captured data can feed into the live programming aspects of the system, providing the user with a concrete view of the program, instead of a purely abstract one relying solely on static analysis. We believe what information is useful to the programmer and how to best surface it will be an exciting and impactful avenue for further research.

Speculative analysis [**Brun et al., 2010**] and its follow up work can perhaps be viewed as a specific instance of this direction, where the focus is on using speculative analysis in the background to help developers make certain decisions.

1.2.4. Human Interface Innovation. Textual code is a precise and expressive medium for communicating intent. Looking back at the past half century of programming history, it is hard to see it going away anytime soon. However, most of the computing devices shipped today are phones that do not have a physical keyboard and mouse. While it may seem unlikely that most professional programming will ever be done on such devices—at least without some external accessories—, it is almost certain that end-users would want to use them to accomplish custom computational goals or control systems by defining actions that would happen in response to specific events.

Accomplishing this requires innovation both on the human interface front and on the backend engine. It is likely that many of the functionalities will be exposed via artificial intelligence-based assistants, and will be expressed as interactive voice conversations. On the backend, we need to build more interactive programming systems that can make educated guesses and synthesize programs with incomplete specification, and interactively adapt them as specifications perfected by gradually asking for and capturing additional user input.

Moreover, even on more traditional computers, *e.g.* desktops and laptops, we need fundamental interface innovations to support alternative programmers [**Schachman, 2012**], *i.e.* people who are not professional programmers and write programs that do computation and produces a result, which is the object of interest to them, as opposed to the program itself. An important class of people who would benefit from such interface innovations are people doing analysis on various data sets. Already, tools like IPython [**Pérez and Granger, 2007**] that have more interactive characteristics and suit domain-specific use-cases well have gained widespread adoption in that community. We believe that there is enormous potential to carry out research that would substantially impact the life of alternative programmers in a positive way in this area.

1.3. Outline

In this chapter, we have advocated the TIIS mission for quick realization of ideas as working systems and the modernization of our techniques and tools to better support programming in the coming decades. Given the ubiquity of connected computer systems — mostly in the form of smartphones — we are just at the beginning of an explosion of ideas and applications that wait to be realized by professional developers or end users. Consequently, it is even more important that we do our best as a community to improve our programming practice to adapt it for the future challenges we will likely face. Achieving the TIIS mission will require significant efforts spanning many directions. We have identified, as a first step, several directions centered around four principles. We hope that the community unite to move the state-of-the-art forward toward the TIIS vision along these and other important pertinent directions.

In the chapters that follow, we deep dive into a few of these directions and further discuss how we can leverage some of the new techniques that we developed to move towards acheiving this mission.

CHAPTER 2

Liberating the Programmer with Prorogued Programming

Programming is the process of expressing and refining ideas in a programming language. Ideally, we want our programming language to flexibly fit our natural thought process. Language innovations, such as procedural abstraction, object and aspect orientation, have helped increase programming agility. However, they still lack important features that a programmer could exploit to quickly experiment with design and implementation choices.

We propose *prorogued programming*, a new paradigm more closely aligned with a programmer's thought process. A prorogued programming language (PPL) supports three basic principles: 1) *proroguing concerns*[1]: the ability to defer a concern, to focus on and finish the current concern; 2) *hybrid computation*: the ability to involve the programmer as an integral part of computation;

[1]A *concern* is any piece of interest or focus in a program [**Dijkstra, 1974**].

and 3) *executable refinement*: the ability to execute any intermediate program refinements. Working in a PPL, the programmer can run and experiment with an *incomplete* program, and gradually and iteratively reify the missing parts while catching design and implementation mistakes early. We describe the prorogued programming paradigm, our design and realization of the paradigm using Prorogued C#, our extension to C#, and demonstrate its utility through a few use cases.

2.1. Introduction

Programming is one of the most challenging human endeavors, as it forces a programmer to simultaneously manage many, at times even conflicting, concerns. It is a gradual, iterative process of expressing, experimenting with, and refining ideas in a programming language. Advances in programming language design have helped programmers focus on important programming-related concerns rather than less critical ones [**Brooks, 1995**]. Nonetheless, opportunities for improvement remain, especially during program construction.

For example, a programmer may need to invoke a function that has not yet been written to test and experiment with other parts of a system. In mainstream languages, the compiler will not compile code that depends on a nonexistent function. To satisfy the compiler, a programmer must either implement the function or write a stub for it. This may break the programmer's train of thought and force a distracting abstraction shift upon her [**Czerwinski et al., 2004, Iqbal and Horvitz, 2007**]. Writing a stub requires typing a function declaration, and, in safer languages like Java, convincing the compiler that the stub returns correctly. Even when a refactoring tool helps generate a stub, that stub is code that is likely to change and whose maintenance can be distracting.

This phenomenon may explain the increasing industrial adoption of dynamic languages[2] like Python and JavaScript. However, the status quo is little better in dynamic languages. While dynamic languages dispense with ahead-of-time compilation and can start running an incomplete program, the interpreter halts execution when it fails to dispatch a call to an unimplemented method. Moreover, the execution of incomplete programs comes at the expense of compile-time guarantees that a statically-typed language provides.

2.1.1. Prorogued Programming. We propose a new programming paradigm, *prorogued programming*, to lift these restrictions: it allows programmers to compile and run incomplete programs so they can test and refine work-in-progress. It supports the following three basic principles.

Proroguing Concerns. Prorogued programming allows a programmer to *prorogue* a concern so that she can continue her train of thought and quickly experiment with high-level design and implementation decisions. A prorogued concern can be a yet-to-be-implemented function whose implementation would derail the current task, is being implemented by another developer, or whose need is unclear pending a high-level design decision. To achieve this, we let the programmer designate function invocations as *prorogued*, explicitly making the compiler or interpreter aware that the callee is not yet implemented. When it encounters a prorogued call, the compiler continues translating and statically checking the program. During execution, the *prorogue dispatcher* intercepts a

[2]In this chapter, we refer to mainstream interpreted implementations of dynamically-typed languages as *dynamic languages*.

prorogued call and supplies a placeholder instance, a *prorogued value*, as its return value.

Hybrid Computation. A prorogued program continues its execution under the semantics of the host language until it reaches a prorogued call. At that point, it brings a human into the process: it displays the arguments of the most recent call to a prorogued function and asks the programmer to supply a return value. The prorogue dispatcher saves the programmer's response. Subsequent uses of the prorogued value, or prorogued invocations with identical arguments, do not need interactive resolution: normal execution continues with the previously user-supplied value. By bringing humans into the process, we can enable a more meaningful execution for partial implementations than mechanically generated stubs that do not capture programmer intent. Prorogued programming is therefore ideal for problems for which a general solution is hard to implement but for which it is easy for humans to generate examples [**von Ahn et al., 2008, von Ahn and Dabbish, 2004, von Ahn et al., 2003**]. section 2.5 discusses a few such examples using Prorogued C#, our realization of prorogued programming for C#.

Executable Refinement. As programming is the iterative process of expressing and evolving programs, prorogued programming lets the programmer compile, statically analyze, execute, and observe the behavior of program refinements, or incomplete programs. This allows the compiler to typecheck and catch errors throughout program construction. Prorogued programming is about maintaining incomplete, but readily testable programs. With little effort, these partial programs are compilable and executable, so that a developer can seamlessly transition to experimenting with her code at any time. To this end, a prorogued language interacts with a user to enable the execution of an incomplete program. Integrating human and traditional machine computation, this hybrid model opens up opportunities for more productive program construction, including crowdsourcing (subsection 2.5.1).

In short, prorogued programming helps programmers hew to their natural workflow, evolving the program by focusing on the top-down design and implementation, and filling in the details as needed [**Wirth, 1971**], thus avoiding housekeeping merely to satisfy a language's implementation. As a result, the development team can iterate more quickly, recognize fundamental and high-level design and implementation errors throughout program construction. Furthermore, a program's modules can run independently by proroguing their dependencies.

Prorogued programming targets functionality for which the developer has a few input-output pairs in mind. In this case, a prorogued call enables the developer to explore the caller's logic. When the developer does not have a small set of input-output pairs in mind or when execution generates inputs outside of that set, excessive interaction can ensue. Most often the solution is to wrap the prorogued call in logic that suppresses unwanted interactions. Further, we acknowledge that, most of the time the programmer must eventually write the code that realizes a prorogued method. The fact that we do not eliminate this work is orthogonal to what prorogued programming does provide — *viz*, new and better workflows. First, prorogued programming gives programmers the power to defer that work until they can, and wish to, concentrate on it. Second, prorogued programming allows parallel development on prorogued methods: while one developer continues to develop using a prorogued call, another developer can examine its IO store — the collected input/output values — and begin its implementation. The existence of the IO store also may facilitate the implementation of the module it approximates by allowing the implementor to study the IO store and gain insight. The IO store may also provide useful input to synthesis tools that learn from examples [**Harris and Gulwani, 2011**, **Gulwani, 2011**, **Witten and Mo, 1993**, **Lau et al., 2003a**, **Lau et al., 2000**, **Lau et al., 2003b**] and test-based code search tools [**Reiss, 2009**, **Lemos et al., 2007**, **Lemos et al., 2011**, **Jiang and Su, 2009**].

The prorogued programming paradigm improves collaborative software development by reducing interpersonal and cross-team dependencies. One team can continue development by proroguing method calls across components without having to wait for a fixed interface to be supplied by the team writing the underlying component. Also, prorogued programming facilitates component developments by means of proroguing stateful types in addition to simple methods. Moreover, it reduces the risk of stalling development while selecting third-party components since prorogued methods can proxy those components, allowing development to continue while a choice of component vendor is being made.

2.1.2. Main Contributions. This chapter makes the following contributions:

- We introduce a new programming paradigm that allows programmers to defer programming concerns and finish their current task. In so doing, it makes program construction more closely conform to how humans actually think when programming.

- We present the design and realization of the prorogued programming paradigm in C#. In particular, we discuss and motivate our design decisions of incorporating and supporting prorogued programming in a real-world programming language.
- We discuss software engineering implications of the prorogued programming paradigm and show its power, applicability, and universality through a collection of case studies.
- We discuss open issues, such as utility and usability, applicability, program evolution and reification, and possible approaches for addressing them.

The rest of this chapter is organized as follows. We first use an example to motivate prorogued programming and illustrate its use (section 2.2). Next, we formalize prorogued programming for a small functional language (section 2.3). section 2.4 describes our design and realization of prorogued programming for a real-world language, C#. We then use a few examples to highlight the utility of prorogued programming (section 2.5). section 2.6 discusses a few open issues and opportunities for prorogued programming. Finally, section 2.7 surveys related work, and section 2.8 concludes.

2.2. Illustrating Example

To motivate and illustrate the utility of the prorogued programming paradigm, we describe a scenario in which a programmer, call her Lily, builds an application that reads a file named `"mail.txt"` containing a simplified raw

```
static void Main() {
  string input =
    prorogue ReadFile("mail.txt");
  PrintEmail(input);
}
static void PrintEmail(string input) {
  string from =
    prorogue GetHeader(input, "From");
  string subject =
    prorogue GetHeader(input, "Subject");
  string body =
    prorogue GetBody(input);
  Console.WriteLine("From: " + from);
  Console.WriteLine("Subject: " + subject);
  Console.WriteLine(body);
}
```

FIGURE 2.1. The simple mail parser in Prorogued C#.

email message. It pretty-prints the relevant parts of the message such as its sender, subject, and body. Lily first writes the high-level aspects of the mail parser application; she decomposes her task into the methods `ReadFile`, `GetHeader`, and `GetBody`, then prints the parsed output. Even though these methods do not yet exist, Lily may wish to experiment with her high-level design. Two options exist: 1) she can fully implement these missing methods or 2) provide stubs for them.

At this point, she only wishes to experiment with the high-level implementation decisions and ignore the low-level details of how to implement these missing methods. Thus, the first option would be unnecessarily disruptive. So she takes the second option and writes stubs that merely return the empty string for the missing methods. Unfortunately, this option is also disruptive: 1) she

```
static void Main() {
  var db = prorogue
    new UserDatabase { Server = "server1" };
  db.ConnectionTimeout = 1000;
  var userName = Console.ReadLine();
  var password = Console.ReadLine();
  if (db.Authenticate(userName, password)) {
    string input =
      prorogue ReadFile("mail.txt");
    PrintEmail(input);
  } else Console.WriteLine
                       ("Authentication failed.");
}
```

FIGURE 2.2. Mail program using a prorogued mock database.

needs to write the stubs; 2) the stubs, although simple, can contain errors, which she would have to fix; and 3) the stubs must return artificial values because Lily does not know with which inputs they may be invoked. In summary, neither option is ideal.

While writing these functions is not particularly hard and Lily can implement them with a couple of regular expressions, she will probably need to look for and read about the regular expressions API, then experiment with her regular expression to ensure it is correct. As Jamie Zawinski famously said, she now has two problems. At the very least, refining her regular expressions will distract her from her current task, forcing her to context switch and work at a different level of abstraction.

Now, let us see how Prorogued C# can aid Lily. As above, Lily first writes the high-level aspects of the program, but with the power to prorogue the details. Figure 2.1 depicts this initial draft of Lily's mail parser. The Prorogued C#

compiler compiles this code, even though the methods `ReadFile`, `GetHeader`, and `GetBody` do not yet exist. Here, we assume that Lily has in mind a small set of emails she can use as input while testing her incomplete program and from which she can quickly extract the appropriate output. The first time the program runs, the prorogue dispatcher initiate hybrid computation and asks Lily for return values of each prorogued method call, and continues execution and prints out the values received from her at lines 8, 10, and 12. The next time the program runs, it simply prints out those values and exits, since the previous run saved Lily's responses and the method arguments were unchanged. When this happens, the prorogue dispatcher simply returns the saved values.

Prorogued programming allows Lily to prorogue types as well as methods. For instance, Lily can use prorogued programming to instantiate a mock database and begin to flesh out her authentication logic, as shown in Figure 2.2. We apply the `prorogue` keyword to the constructor call on line 3 to create a mock object on which all method calls that are not already implemented in `UserDatabase` type, such as the call to `Authenticate` on line 7, are prorogued, like those in Figure 2.1. The assignments on line 3 and line 4 simply create properties within the `db` instance. After each change, prorogued programming allows Lily to immediately compile, execute, and experiment with the refined, albeit still partial, program.

Later, Lily discovers a built-in method to read a text file and return its contents as a `string`. To use it, Lily replaces the `ReadFile` invocation with the framework-provided method, removing the first prorogued call: `string input = File.ReadAllText("mail.txt");`. After this change, the program actually reads the `"mail.txt"` file. The program interacts with the user to produce a result every time the file contents is changed. Again, it persists the result of that hybrid computation for reuse in subsequent calls.

The fact that we were able to prorogue the `ReadFile` method highlights a useful property of prorogued programming: the programmer can continue testing a program that relies on an external resource or module when that resource or module is not readily available. The program can execute and be debugged without having to resort to explicit mocking techniques, simply by proroguing a call that depends on the external resource. To make it easier to debug the program during the construction phase, we can prorogue the invocation to `File.ReadAllText`, *shadowing* the existing method implementation: `string input = prorogue File.ReadAllText("mail.txt");`. When it encounters `prorogue` applied to a call to a preexisting function, the compiler warns the developer that it will ignore that function's existing implementation and treat it as a prorogued method. When it reaches the shadowing prorogued function call, the program presents its inputs and prompts the programmer for a return value. The programmer can then choose a return value that drives execution to a particular program point.

We can leverage the input/output pairs captured in the interactive process to generate code via *reification*. Reification removes the `prorogue` keyword from call sites and generates code in the form of an `if-else` chain that, to handle unknown inputs, culminates in a prorogued call. For existing implementations, like the prorogued call to `File.ReadAllText`, reification simply removes the `prorogue` keyword and issues a warning. In an IDE with first-class support for prorogued programming, the reification tool will be integrated in the IDE. The

$$
\begin{array}{rl}
\textit{Program} & p ::= p,\ \text{fun } f(x) = e \mid \varepsilon \\
\textit{Expression} & e ::= n \mid x \mid e_1 \text{ op } e_2 \\
 & \quad \mid\ \text{if } e\ e_1\ e_2 \\
 & \quad \mid\ \text{let } x = e_1 \text{ in } e_2 \\
 & \quad \mid\ f(e) \\
 & \quad \mid\ \textbf{prorogue } f(e)
\end{array}
$$

FIGURE 2.3. The syntax of the simple prorogued language $\mathscr{F}_\text{p}$, which adds, to a standard expression language, the new syntactic construct "**prorogue** $f(e)$."

resulting program is immediately runnable. Typically, the programmer fills in the final implementation details of each, formerly prorogued, method.

It is possible to perform a global reification as well as selectively specifying a set of methods to reify. If the function is naturally a direct mapping between a small set of inputs and outputs (*e.g.* a function returning a string representation for `enum` values), the reified implementation might be immediately useful. For functions exhibiting more complex behavior, the programmer can use the generated code as a skeleton and write code for the custom behavior. The pairs collected by running a prorogued program can also be used to generate unit tests automatically (section 2.6). Programmers can use these tests to ensure that the behavior of the method's implementation matches the expected behavior as collected when the method was prorogued.

2.3. A Prorogued Programming Language

This section formalizes the syntax and semantics of a small prorogued programming language $\mathscr{F}_\text{p}$ to clarify our presentation. Our actual implementation (Section 2.4) is an extension to C#.

$$\langle p,\sigma,\kappa,n\rangle \Downarrow \langle\kappa,n\rangle \qquad \text{[const]}$$

$$\langle p,\sigma,\kappa,x\rangle \Downarrow \langle\kappa,\sigma(x)\rangle \qquad \text{[var]}$$

(A) Semantics of const and var.

$$\frac{\begin{array}{c}\langle p,\sigma,\kappa,e_1\rangle \Downarrow \langle\kappa_1,v_1\rangle\\ \langle p,\sigma,\kappa_1,e_2\rangle \Downarrow \langle\kappa_2,v_2\rangle\end{array}}{\langle p,\sigma,\kappa,e_1 \text{ op } e_2\rangle \Downarrow \langle\kappa_2,v_1 \llbracket\text{op}\rrbracket\, v_2\rangle} \quad \text{[op]}$$

(B) Semantics of op.

$$\frac{\begin{array}{c}\langle p,\sigma,\kappa,e\rangle \Downarrow \langle\kappa_1,0\rangle\\ \langle p,\sigma,\kappa_1,e_2\rangle \Downarrow \langle\kappa_2,v_2\rangle\end{array}}{\langle p,\sigma,\kappa,\text{if } e\; e_1\; e_2\rangle \Downarrow \langle\kappa_2,v_2\rangle} \quad \text{[if-false]}$$

(C) Semantics of if-false.

$$\frac{\begin{array}{c}\langle p,\sigma,\kappa,e\rangle \Downarrow \langle\kappa_1,1\rangle\\ \langle p,\sigma,\kappa_1,e_1\rangle \Downarrow \langle\kappa_2,v_1\rangle\end{array}}{\langle p,\sigma,\kappa,\text{if } e\; e_1\; e_2\rangle \Downarrow \langle\kappa_2,v_1\rangle} \quad \text{[if-true]}$$

(D) Semantics of if-true.

$$\frac{\begin{array}{c}\langle p,\sigma,\kappa,e_1\rangle \Downarrow \langle\kappa_1,v_1\rangle\\ \langle p,\sigma[v_1/x],\kappa_1,e_2\rangle \Downarrow \langle\kappa_2,v_2\rangle\end{array}}{\langle p,\sigma,\kappa,\text{let } x = e_1 \text{ in } e_2\rangle \Downarrow \langle\kappa_2,v_2\rangle} \quad \text{[let]}$$

(E) Semantics of let.

FIGURE 2.4. The simple prorogued language $\mathscr{F}_\text{p}$, which adds, to a standard expression language, the new syntactic construct "**prorogue** $f(e)$." Its dynamic semantics is specified in the big-step style, where 1) $\llbracket\text{op}\rrbracket$ denotes the semantic interpretation of op, 2) Φ_f the oracle for a prorogued function f, and 3) $\oplus$ function overriding: $\kappa_2 = \kappa_1 \oplus ((f,v),v_1)$ iff $\kappa_2(f,v) = v_1$ and $\kappa_2(f',v') = \kappa_1(f',v')$ for all $(f',v') \neq (f,v)$. Continued in Figure 2.5

2.3.1. Syntax and Semantics of $\mathscr{F}_\text{p}$. $\mathscr{F}_\text{p}$ extends a standard core expression language; Figure 2.3 shows its syntax and Figure 2.4, its semantics. An $\mathscr{F}_\text{p}$ program consists of a list of functions, each of which has a single integer

$$\frac{\langle p, \sigma, \kappa, e\rangle \Downarrow \langle \kappa_1, v\rangle \qquad p(f) = \lambda x.e_1 \qquad \langle p, \sigma[v/x], \kappa_1, e_1\rangle \Downarrow \langle \kappa_2, v_1\rangle}{\langle p, \sigma, \kappa, f(e)\rangle \Downarrow \langle \kappa_2, v_1\rangle} \text{ [call]}$$

(A) Semantics of call.

$$\frac{\langle p, \sigma, \kappa, e\rangle \Downarrow \langle \kappa_1, v\rangle \qquad \kappa_1(f, v) = v_1}{\langle p, \sigma, \kappa, \textbf{prorogue } f(e)\rangle \Downarrow \langle \kappa_1, v_1\rangle} \text{ [p-call-old]}$$

(B) Semantics of p-call-old.

$$\frac{\langle p, \sigma, \kappa, e\rangle \Downarrow \langle \kappa_1, v\rangle \qquad \kappa_1(f, v) = \bot \qquad \Phi_f(v) = v_1 \qquad \kappa_2 = \kappa_1 \oplus ((f, v), v_1)}{\langle p, \sigma, \kappa, \textbf{prorogue } f(e)\rangle \Downarrow \langle \kappa_2, v_1\rangle} \text{ [p-call-new]}$$

(C) Semantics of p-call-new.

FIGURE 2.5. Continued from Figure 2.4

argument and an expression as its body. With the exception of the **prorogue** construct, the expression sublanguage is standard. An integer literal is n and x is a variable, over integers. We use op to denote a primitive operation whose semantics is given by $[\![\text{op}]\!]$, *e.g.*, $[\![+]\!]$ is integer addition. As usual, if and let denote the conditional and local binding constructs. Function invocation is $f(e)$ and **prorogue** $f(e)$ denotes a *prorogued* function invocation, whose semantics we formalize next.

Figure 2.4 give the dynamic, big-step semantics of $\mathscr{F}_{\text{p}}$. The value domain is $\mathit{Value} = \mathbb{Z} \cup \{\bot\} = \mathbb{Z}_\bot$. Evaluation judgments have the form $\langle p, \sigma, \kappa, e\rangle \Downarrow \langle \kappa', v\rangle$ where

- the program p maps a function name f to its definition: 1) $p(f) = \lambda x.e$ if p contains "fun $f(x) = e$", and 2) $p(f) = \perp$ otherwise;
- the state $\Sigma \ni \sigma : \mathit{Var} \rightarrow \mathit{Value}$ maps variables to values;
- the IO store $\kappa : \mathbb{F} \times \mathit{Value} \rightarrow \mathit{Value}$ maps a *prorogued* function f and an argument i to an output o, *i.e.*, $\kappa(f,i) = o$ (where $\mathbb{F}$ denotes the set of functions);
- e is the expression being evaluated;
- κ' is the updated IO store after evaluating e; and
- v is the result of evaluating e.

The evaluation rules are straightforward. For conditionals, we let 0 denote false and 1 denote true. The term $\sigma[v/x]$ denotes an updated state σ' where $\sigma'(x) = v$ and $\sigma'(y) = \sigma(y)$ for all $y \neq x$.

The [p-call-old] and [p-call-new] rules are specific to prorogued programming. The programmer prorogues a function to defer its implementation. When code containing a prorogued call of the function f executes, if f has been invoked with the argument v, the previously returned value v_1 stored in the IO store κ_1 is returned (as shown in the [p-call-old] rule). Otherwise the oracle Φ_f is consulted, as shown in rule [p-call-new], and the IO store is updated to yield the new κ_2 (via $\oplus$, the function override operator). Initially, the IO store κ is empty; calls to the oracle Φ_f populate it, so its contents are correct. To realize the oracle, we apply our hybrid computation principle, and involve the programmer. We describe a concrete realization of this interaction in section 2.4.

From the above discussion, we see that $\mathscr{F}_\text{p}$ naturally supports the three principles of prorogued programming: 1) **prorogue** $f(e)$ allows the programmer to prorogue the "concern" of implementing the function f (proroguing concerns); 2) humans realize the formal oracle Φ and compute the result of prorogued function invocations (hybrid computation); and 3) an $\mathscr{F}_{\mathrm{p}}$ program, starting from the minimal "**prorogue** *main*()", is executable at each refinement step, as it evolves (executable refinement).

In this simple functional language, prorogued functions are pure. A programmer who wishes to update state, such as a global, must use assignment to write the return value of a prorogued function into the desired location, as with $g :=$ **prorogue** $f(x)$.

THEOREM 1 (Correctness of Prorogued Semantics). *For any program p, state σ, and expression e,*

$$\forall \kappa, v(\langle p, \sigma, \Phi, e\rangle \Downarrow \langle \kappa, v\rangle \Rightarrow \ \langle p, \sigma, \emptyset, e\rangle \Downarrow \langle \_, v\rangle).$$

2.3.2. Reifying Prorogued Functions. As a programmer stepwise refines a prorogued program, that programmer will, in general, implement a prorogued function and remove the **prorogue** keyword from its call sites to convert them into standard method calls. We call this process *reification*. Although it only makes sense in the context of a prorogued language, reification is orthogonal to the prorogued programming paradigm, since it is, in essence, an instance of stepwise refinement [**Wirth, 1971**]. That said, reification will be integral to a programmer's workflow when using a prorogued language. Beyond the manual implementation of the prorogued function, we discuss a few rewriting strategies that assist the programmer in replacing prorogued calls:

1) deploy a version of the program that still contains prorogued calls; 2) convert the IO store into code; 3) leverage test-based code search techniques [**Reiss, 2009, Lemos et al., 2007, Lemos et al., 2011, Jiang and Su, 2009**] to find reusable implementations; and 4) employ synthesis by example techniques [**Harris and Gulwani, 2011, Gulwani, 2011, Witten and Mo, 1993, Lau et al., 2003a, Lau et al., 2000, Lau et al., 2003b**] using the IO store as input.

The first strategy leaves the prorogued calls untouched. It may be applicable for programs containing functionality that can be approximated by a set of input/output pairs and complex enough not to be profitable to implement. The second strategy reifies the set of IO pairs as an `if-else` chain or a `switch` statement. Studying the set of IO pairs in this executable and modifiable format may help a programmer gain insight into how to devise an algorithm that abstracts the behavior encoded in the set. The last two strategies rest on the observation that the IO stores that a prorogued program produces may provide a fertile new source of applications and problems for test-based code search and program synthesis.

THEOREM 2 (Correctness of Reification). *For any program p, state σ, and expression e,*

$$\forall \kappa, v(\langle p, \sigma, \emptyset, e\rangle \Downarrow \langle \kappa, v\rangle \Rightarrow \forall \theta_\kappa \langle \theta_\kappa(p), \sigma, \emptyset, \theta_\kappa(e)\rangle \Downarrow \langle \emptyset, v\rangle)$$

where θ_κ denotes any correct reification strategy w.r.t. κ, i.e., $\theta_\kappa(f)(i) = \kappa(f,i)$ for all f and i with $\kappa(f,i) \neq \bot$.

2.4. Design and Realization of Prorogued C#

To experiment with prorogued programming, we extended the Mono C# compiler [**de Icaza et al., 2010**], an open source implementation of C#, a popular, real-world language. We chose a statically typed language to demonstrate the universality of the prorogued programming paradigm. This section discusses the design choices we made and interesting implementation details.

2.4.1. The Language. To realize the prorogued programming paradigm in C#, we amended the C# grammar [**Hejlsberg et al., 2010**] to include `prorogue` as a keyword and added the production

prorogued-invocation-expression ::=

`prorogue` *primary-expression* `(` *argument-list* `)`

to decorate invocation expressions. In the case of a chain of method invocations, `prorogue` binds to the first invocation in the chain: the Prorogued C# compiler parses `prorogue a().b().c()` as `(prorogue a()).b().c()`. While a programmer might prefer `prorogue` to bind to the last call in the chain than the first, this design decision is a more natural fit to C#, since it is consistent with left-associativity of the dot operator and other constructs. As usual, the programmer can resort to parentheses to override this behavior.

By default, a simple prorogued call like `prorogue Foo()` assumes the callee is a static method in the current type. To prorogue a method call in another type, that type must qualify the method name:

`prorogue FooNamespace.BarClass.Baz( arg1, arg2)`. The above expression prorogues a call to the static `Baz` method in the context of `BarClass` declared in the `FooNamespace` namespace.

To prorogue an instance method, a programmer must prepend the `prorogue` keyword to an instance method invocation expression: `prorogue obj.InstanceMethod(arg)`. Of course, an arbitrary expression yielding a value can replace `obj`. In the above example, we assume that `InstanceMethod` is an instance method in the context of the static type of the receiver expression, `obj`. Proroguing an instance method of the type in which a prorogued call appears is a special case, in which the programmer uses `this` as the receiver: `prorogue this.InstanceMethodInCurrentType(arg)`.

2.4.2. Prorogued Types. Prorogued C# also supports proroguing types. This is achieved by prepending an *object-creation-expression* with the `prorogue` keyword which is supported by the

prorogued-creation-expression ::=
    `prorogue` *object-creation-expression*

production in the grammar. Extending the idea of proroguing concerns from methods to an entire type, potentially with mutable state, enables the programmer to prorogue the design of a module or component while writing the client code that consumes it.

A prorogued type is instantiated using a regular type that it extends. It acts as a proxy, dispatching implemented methods to the underlying type while treating the rest as prorogued calls. Supporting prorogued types complicates

```
1 var msg = prorogue new Message {
2   From = "from@email.com",
3   To = "to@email.com",
4   Delivered = false
5 };
6 msg.Send(login, passwd);
7 Console.WriteLine(msg.Delivered);
```

FIGURE 2.6. Example of a prorogued type; the UI interaction for the prorogued call on line 6 happens on line 7 where the value of `msg.Delivered` can mutate as a result of the interaction.

lazy evaluation, discussed below in subsection 2.4.5, and requires the handling of mutable state, in contrast to simple prorogued functions that are pure value-to-value transformations. In principle, a method invocation on a prorogued type can still be thought of as a value-to-value transformation in which the state mutation is an element in the return tuple. The prorogue dispatcher then dissects the return tuple and mutates the state of the prorogued instance. Of course, the user interface is smart enough to hide this implementation detail and lets the user manipulate state as if the function itself, as opposed to prorogue dispatcher, was mutating state.

Sometimes, a prorogued type relies on global state, external input, or state that is not implemented yet. For instance, while mocking an object that represents a network stream, we might want to make two consecutive `ReadLine()` calls return two distinct values, despite the fact that it is called with the same set of arguments, *i.e.* none, both times. The canonical pattern for preventing the prorogue dispatcher from simply returning the cached value from the first call in response to the second, when no explicit state change has occurred, is to introduce one, *i.e.* change the value of a dummy property in the user interface to implicitly capture the state of the object during the execution of the program. Of course, this solution will not scale to complex interactions with the mocked object, but recall that prorogued programming's purpose is to record and replay a relatively small set of behaviors from the developer to allow that developer to continue her current task. In this case, the user, upon returning from the first call, assigns the value 1 to a property named `readCount` of the instance in the UI. Since the instance does not have such a member, it is added to the type on the fly, which causes the prorogue dispatcher to ask for a new return value when it dispatches the second call, since the receiver object's state has changed.

```
var netStream = prorogue
  new NetStream { Host = "server", Port = 80 };
string line1 = netStream.ReadLine();
Console.WriteLine("Line 1: " + line1);
string line2 = netStream.ReadLine();
Console.WriteLine("Line 2: " + line2);
```

2.4.3. The IO Store. The IO store maps input to outputs. The design question it presents is to decide what it should accept as inputs and outputs. Should IO store contain code (including values) or only values? If only values, should it store instances of user-defined types or only instances of system types?

Code vs. Values. Binding code to a prorogued call would allow a programmer the flexibility of handling some inputs with code, while simply returning values for the rest. Unfortunately, binding code to a prorogued function in the IO store would come at some cost. It would make programs more complicated

```
T_0 transmute(T_1 x, T_2 y) {
  if (x < 0)
    return 0;
  else
    return prorogue transmute(x, y);
}
```

FIGURE 2.7. In prorogued languages, a developer can partially implement a previously prorogued function to handle part of its input domain and prorogue the rest, reusing the IO store populated before partial implementation.

and harder to understand by scattering executable logic across the prorogued program and the IO store. One would have to decide whether or not to allow nested prorogued calls and, if so, their execution semantics. It would prevent lazy dispatch of prorogued calls. The ability to write code in response to a query from a prorogued call may distract a programmer into doing just that, defeating the principle of proroguing concerns. Finally, it violates the principle of simplicity and, in the end, is unnecessary, as we demonstrate next.

When a programmer is ready to partially implement a prorogued function, that programmer has two choices: 1) reify the prorogued function's IO into code, as described in subsection 2.4.7, and edit the result or 2) define the formerly prorogued function in the host language, making prorogued calls to the function as desired. Figure 2.7 depicts this latter case. In essence, the programmer writes logic to directly handle some cases, while proroguing the rest to the previously populated IO store. Partial implementation allows a developer to suppress unwanted interaction with a prorogued function. For instance, if a programmer learns that a frequently called, prorogued function should return 0 whenever its first input is negative, the programmer simply defines `transmute` as shown in Figure 2.7. This strategy of pushing down a prorogued call into a partially implemented function is always possible. Therefore, a prorogued language loses no expressive power by restricting prorogued functions to values. Indeed, an IDE for a prorogued language could provide a developer with the illusion of an IO store that intermixes code and values by maintaining that store as a non-prorogued function that makes prorogued calls as appropriate. Implementation of a function is complete when the prorogued call is detritus.

System vs. User-defined Types. The next question is whether to allow the IO store to contain instances of user-defined types or restrict it to system-defined values, instances of values defined by type in a prorogued languages default distribution of libraries. The argument for the latter is mainly simplicity: working with values defined over a fixed set of types may allow optimized layout of the IO store and restrict the complexity of queries, forcing the programmer to deconstruct a potentially complex input into values defined over a prorogued language's constituent, well-known types. This restriction might also address the problem with objects pointed to by reference type arguments mutating between a call site and lazy dispatch (subsection 2.4.5) since, in principle, we could traverse any referenced data structure. This design choice has two problems. First, it violates the principle of least surprise by handling system types, the set of which is not even clearly defined, differently than user-defined types, a distinction that C# itself does not make. Second, it does not give the programmer sufficient power to abstract inputs, *e.g.* into intervals. For example say the programmer knows that $[0..10] \to 5$. If Prorogued C# restricted its user to system types, encoding this fact into the IO store would require 10 tedious and distracting interactions, dragging out the handling of this concern

```
int foo(int x) {
  if (isPrime(x))
    return prorogue primeFoo();
  else
    return <previous logic>;
}
```

FIGURE 2.8. Extending existing functions with `prorogue`.

and violating the first principle of prorogued programming which is to alleviate distraction by allowing programmers to defer work. Worse, what if the range were over floating-point numbers? Of course, the developer could resort to partially implementing a prorogued method, writing `if (x >= 0 || x <= 10) y = prorogue foo(5);` but this too would be cumbersome and run counter to prorogued programming's central goal of concern deferment.

Thus, we decided to restrict the IO store to map values to values, over arbitrary types. To persist across runs, these types must be serializable. User-defined types give the programmer the power to abstract inputs into classes that can arbitrarily partition the space of underlying values. The programmer can simply abstract a partition into a class and pass instances of that class to a prorogued function. So for instance, a developer could define `incomeInterval` as `new Interval { Start = (int)Math.Floor(income / 1000), End = (int)Math.Ceiling(income / 1000) }`, then use the resulting interval in a prorogued call — `var taxRate = prorogue GetTaxRate(incomeInterval);`. User-defined types give the programmer similar power over the output. Indeed, nothing prevents the programmer from defining a prorogued method that returns an expression tree, which the program executes.

Extending Existing Functions. A consequence of our decision to disallow placing code in a prorogued function's data store is that prorogued methods are restricted to leaf nodes in the call graph. To prorogue an existing function whose functionality you want to extend, you add a prorogued call into its function body along the path you wish to extend. You may even need to add that path. For instance, imagine that you wanted a function `foo` that previously did not distinguish between composites and primes to handle primes differently. You would modify `foo` in the host language to add the path that makes a leaf call to a prorogued function: Of course, we could also implicitly resort to user-defined types here and rewrite this example as `return prorogue foo(prorogue isPrime(x));`.

The design decision to implement prorogued functions as value-to-value transformations under the hood makes prorogued programs simpler, prevents the scattering of executable logic across the program and its IO stores, and allows prorogued functions to be pure, which allows the caching of results and the lazy evaluation of calls at the cost of reference and output parameters. Without giving up simplicity, prorogue can leverage the abstraction of user-defined types that the host language provide to reclaim any expressive power lost by restricting IO stores to values, as opposed to executable code.

2.4.4. Typechecking. To typecheck a prorogued call, we could 1) force the programmer to declare its signature, 2) infer the signature, or 3) use a generic signature. Two principles guided our design here: proroguing concerns and coexisting naturally with the host language's type system. In this context, the principle of proroguing concerns implies that our choice should not distract the programmer with concerns other than the one on which she is currently focused. This principle leads us to reject the first choice, that of forcing the programmer to declare each prorogued function's signature, since, in general, a programmer might prorogue a function precisely because they wish to defer deciding its signature.

One could infer the signature of a prorogued function from the types of the arguments at a prorogued call site. One might be tempted to treat one of the call sites specially and extract a signature from it. However, there is no principled, general way to do so, short of revisiting the first design choice and involving the programmer. Thus, we extract a signature from each call. For example, in `var var1 = prorogue Foo(5);` the type of `Foo` is `int` $\rightarrow$ `dynamic`, while that of `var var2 = prorogue Foo("hello, world");` is `string` $\rightarrow$ `dynamic`. This design choice implicitly overloads `Foo` whenever the compiler encounters a new signature, and therefore creates a different prorogued function with its own IO store. To avoid unintended method overloading, the programmer would have to tediously cast each call to the desired base class; for `var2`, the example is `var var2 = prorogue Foo((object)"hello, world");`. Not only is this cumbersome, forcing unnatural, explicit casts to a shared ancestor, but it runs counter to the spirit of prorogued programming, since it distracts the programmer with details from a concern other than the one she is working on, thereby defeating some of the benefits of prorogued programming.

We could bypass C#'s type system and build our own that infers a prorogued function's signature from all the calls to it. For instance, we could experiment with equality-based unification. However, this approach violates our design principle of peaceful coexistence with the host language, so we do not consider it further. Another approach is to infer the signature from all the prorogued calls to a particular name. In C#'s class-based subtype system, every type is a subtype of `object`. Thus, this approach would be unable to distinguish between type errors and intended polymorphism because all types unify at `object`, if not before.

This last approach to signature inference is effectively indistinguishable from the third choice but requires more work, so, for simplicity, we choose the third option: in Prorogued C#, the type of a prorogued invocation expression is `dynamic` and the type of its parameters is always `object`. As a consequence of the fact that its parameters all have type `object`, overloading prorogued methods is possible only if the number of parameters vary. Since its return type is `dynamic`, the return value of a prorogued method call is implicitly convertible to any type. With this assumption, the Prorogued C# compiler typechecks a program with C#'s existing type system. While employing `dynamic` types is a simple way to implement the prorogued programming paradigm and is consistent with our design principles, it is important to point out that our paradigm is by no means restricted to languages that support dynamic invocation. In a language without a similar feature, a prorogued invocation could return a special type that the compiler could convert to any other type. The compiler could then typecheck the program and generate code to invoke the prorogue dispatcher when it encounters such a type conversion.

2.4.5. Lazy Evaluation of Prorogued Calls. An execution of a prorogued program that prompts the programmer to populate IO stores too frequently

```
var value = prorogue foo();
if (condition)
  Console.WriteLine(value);
```

FIGURE 2.9. Lazy evaluation of prorogued return values.

would tediously undermine the utility of prorogued programming. To mitigate this threat, Prorogued C# lazily evaluates prorogued calls. When a prorogued call executes, the prorogue dispatcher immediately returns an implicit future [**Baker and Hewitt, 1977**] to represent that call, along with its arguments. To minimize the number of user interactions, it is not until the first time the return value is *used* in the program that a user interaction might be necessary. In Figure 2.9, if the `condition` evaluates to `false`, user interaction is avoided altogether.

The following constitute use of a prorogued return value. First, there is casting the return value to another type, as implicitly with `string s = prorogue Foo();` or explicitly with `int i = (int)prorogue Bar();`. Second, one can pass the return value as an argument to a prorogued function. For example, we first execute `var input = prorogue GetInput();` and later `input` is used and therefore evaluated in `sqrt = prorogue SquareRoot(input);`. Finally, the return value can be used in an expression in which it does not appear alone: `int sum = 5 + prorogue Bar();`.

Since prorogued calls cannot have global side effects, the only way for a prorogued function to affect the program state is through its return value. Further, lazy evaluation of their returns should not affect program behavior in most cases. Reference types are problematic, however: we copy their reference

```
1 void PrintFinalInvoice(int price) {
2   var discount = prorogue CalculateRebate(price);
3   if (price < 1000)
4     discount = 0;
5   price += GetSalesTax(price);
6   Console.WriteLine(price - discount);
7 }
```

FIGURE 2.10. Capturing the context of a call.

by value to save time and because the prorogue dispatcher cannot traverse arbitrary data structures. Thus, the referenced object may change between the time the prorogued call is encountered and the time it is actually dispatched, changing its behavior and possibly violating the program's semantics.

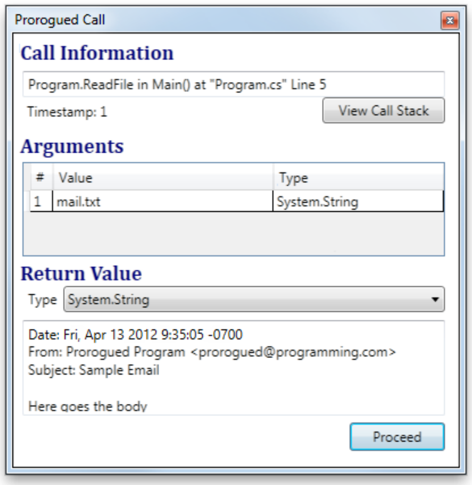
The situation is a more complicated with regard to prorogued types. Method invocations on prorogued types can mutate the internal state of the instance. Operations on prorogued types are kept track of by the prorogue dispatcher and will dispatch in order the first time the instance is being read from or cast to a non-prorogued type.

Laziness brings up another potential issue: the order of user interactions may not correspond to the order in which prorogued calls were visited during execution. In Figure 2.10, the call to `CalculateRebate` is prorogued, but the dispatcher does not prompt the user until execution reaches line 5. If `price` is less than 1000, the user is not prompted for that call. However, the prorogued call to `GetSalesTax` on line 5 is always evaluated when the call returns, due to the implicit cast to `int`. Consequently, the user may be prompted for the second prorogued call *before* the first one. To help the user distinguish the two calls and identify their relative execution order, the dialog that prompts the user for an output contains the source file name, line number, and timestamp when the prorogued call was encountered. The coordinates and timestamp of a call are especially helpful while debugging a prorogued program.

In a multithreaded program, the dispatcher queues and sequentially makes prorogued calls. Execution of the thread making a prorogued call stalls until its user interaction completes. Since user interaction can take an indefinite amount of time, to the other running threads, the user interface thread runs very slowly. This fact can adversely impact programs that rely on timing information or use timeouts, making prorogued calls unsuitable to specific regions of such programs. In a race-free, multithreaded program that does not rely on timing, prorogued program behavior matches its non-prorogued version.

2.4.6. User Interaction. When a prorogued value is first used, the prorogue dispatcher must provide a concrete value to the program and may need to interact with the programmer. To prompt the user, the prorogue dispatcher displays the arguments to, coordinates of, and timestamp of a call, either graphically if the programmer is using an integrated development environment (IDE) or on the command line, if her workflow is terminal-based. Figure 2.11 shows the Prorogued C#'s user interface.

To capture the return value from the user, we need to provide her with a way to express it. For simple types, this is easy: a string representation of the type will do. More complex types require a more powerful, yet still human-readable serialization. XML is too verbose and inconvenient to write. A better solution is JavaScript Object Notation (JSON) serialization, which represents hierarchical object graphs in a concise, readable, and easy to write way. Since JSON is



Call Information

Program.ReadFile in Main() at "Program.cs" Line 5

Timestamp: 1

Arguments

#	Value	Type
1	mail.txt	System.String

Return Value

Type: System.String

```
Date: Fri, Apr 13 2012 9:35:05 -0700
From: Prorogued Program <prorogued@programming.com>
Subject: Sample Email

Here goes the body
```

FIGURE 2.11. Prorogued C#'s user interface.

a popular serialization scheme, flexible libraries and frameworks that handle complex type serialization and deserialization in JSON are readily available.

While persisting prorogued functions works for all argument types that support JSON serialization and is not limited to primitive types, some types are not serializable: persistence is meaningless for types like file and process handles. In many cases, it is better to pass only a fine-grained subset of the state of input to the prorogued function, rather than passing the reference to the complex instance: calling `prorogue GetPrice(c.Make, c.Model, c.Year);` instead of `GetPrice(c)`.

2.4.7. Program Refinement. Reifying a prorogued method into code is a natural phase of the prorogued programming paradigm. When the programmer specifies that they want to reify a prorogued method, the IO pairs are written out as a sequence of `if` statements, usually into the receiver class specified in the call, where the programmer can then edit them. The `prorogue` keyword is then

```
int fib(int x) {
  if (x == 0) return 1;   // (0, 1)
  if (x == 1) return 1;   // (1, 1)
  if (x == 2) return 2;   // (2, 2)
  if (x == 3) return 3;   // (3, 3)
  if (x == 4) return 5;   // (4, 5)
  return prorogue; // fallback
}
```

FIGURE 2.12. Generated code for a reified method.

removed from the call sites of the reified method. In an IDE, the programmer simply selects a prorogued call site, right-clicks and selects reify. Reifying a prorogued class simply iterates over and reifies the prorogued method calls it contains. At the command line, the programmer can issue a command passing a list of classes or methods she wants to reify.

Prorogued C# also allows you to prorogue methods in interfaces and enumerations, that cannot have methods, and types that are not defined in the current project and are externally referenced. In the reification process, static and instance methods of a `class` or `struct`, defined in the current project, are added to the respective source files that define those classes. Instance methods of types that do not support the addition of methods or are declared outside the current project, are generated as static extension methods in a separate class, which is added to the current project, and the relevant call sites are redirected to the newly generated method. Prorogued C# does not infer types (subsection 2.4.4), but it does guess at the types based on the values in the reified function's IO store. In case the guessed signature does not match programmer's intent, the suggested types should be corrected manually. Figure 2.12 shows a simple reified method.

When methods are reified into a sequence of `if` statements, a default branch is added that executes whenever the function acts on arguments not encountered during its prorogued incarnation. This default branch invokes a fallback prorogued call that shadows the newly reified method. This prorogued fallback makes reification an iterative process that helps a program evolve naturally. Since program refinement is a core aspect of the prorogued programming paradigm, we have added a syntactic sugar for a prorogued call that represents fallback behavior: the appearance of the `prorogue` keyword by itself, not followed by an identifier name and parenthesis, is essentially equivalent to recursively proroguing a call to the current method using the arguments[3]. The only difference between the two is that `prorogue foo(x)` triggers a compiler warning about shadowing an existing implementation while `prorogue` is a known and common pattern that does not trigger the warning.

2.5. Applications

Prorogued programming is a practical paradigm that can improve many programming scenarios, ranging from simple to complex. In this section, we present a select few applications that highlight the strengths of this paradigm.

2.5.1. Impact on Software Engineering Practice. We believe prorogued programming will have wide-ranging impact on software engineering practice. Here, we outline how it may transform task assignment and unit testing, permit

[3]This syntax could have been used in Figure 2.7.

the deployment of incomplete programs, and open the door to new forms of development crowdsourcing.

Task Assignment. Program construction is difficult to parallelize [**Brooks, 1995**]. In large projects, where people of different experience and expertise work together, this problem becomes even harder. One way to parallelize the construction of a program is to separate the program into well-defined modules. However, existing paradigms are not as successful at separating concerns during program construction. As a result of natural interdependencies across modules, these paradigms generally impose a specific construction order to programmers. Thus, these paradigms often require careful planning that fixes a large fraction of design beforehand and necessitates writing stringent specifications that clearly communicate that design to the teams responsible for the different modules, all of which reduces agility, delays coding phase, and increases overhead, especially at the beginning of a project.

Prorogued programming separates concerns during program construction; it separates high-level design from low-level implementation details, without requiring a fixed specification before coding. This method of program construction is reminiscent of the way traditional hand-drawn animation used to be produced: senior animators created the *key frames* that represented the major movements of the characters, and after successfully experimenting with the high-level idea of the animation, assigned the task of drawing *inbetweens*, which made the animation smooth, to junior animators.

Besides task assignment based on expertise and experience, in practice, an organization may need to assign tasks based on trust. It may choose to have its security related code written by a handful of trusted security experts to prevent

```
string GetLocalizedWord(Word w, string lang) {
  if (lang == "en" || lang == "en-US")
    switch (w) {
      case Word.Hello: return "Hello";
      case Word.World: return "World";
      default: throw new ArgumentException("w");
    }
  throw new NotImplementedException();
}
```

FIGURE 2.13. Deploying an incomplete program in a non-prorogued language by hardcoding values.

potential exploits and backdoors. To stop leaks of product details before launch, a company may decide not to use a large portion of its human resources on a critical project. Prorogued programming shines in such scenarios because untrusted and outsourced programmers can be tasked to implement low-level aspects of the program without being aware of the overall design goal. Another key benefit of prorogued programming is prioritization of program construction tasks based on their criticality to the project rather than dependency satisfaction.

Deployment of Prorogued Programs. The software industry is highly competitive today. Success demands quickly reacting to customer demands and features that competitors introduce. Predicting the time and resources that a software project needs, even in isolation, is not an easy problem, and statistics show that a large percent of all software projects fail or are delivered late [**Brooks, 1995**]. Consequently, keeping software in a runnable state at all times during construction is a valuable asset and reduces the risk of a software firm. In practice, many software systems, especially custom software systems that are developed

```
string GetLocalizedWord(Word w, string lang) {
  return prorogue;
}
```

FIGURE 2.14. Prorogued localization.

in-house, are incomplete and depend on hardcoded values in code. Existing programming paradigms have not focused on addressing this problem.

For instance, to ship a program intended to eventually support 100 languages, but needed in English quickly, one may have hardcoded values in different parts of code, as shown in Figure 2.13. Had prorogued programming been available, the code could have been written as in Figure 2.14. In addition to demanding less code, the prorogued version is more flexible. To support another language, one need only ask speakers of the target language to run the program, which will harvest and store their responses in the IO store. Prorogued programming obviates hardcoded values, making the code easier to maintain. Furthermore, the system can be kept ready for deployment, "as is", with the languages it already supports. There is no need to interrupt normal development efforts to hardcode values to ship an incomplete program.

Unit Testing. Programmers usually consider writing unit tests to be tedious. The existence of unit tests, however, makes a program easier to maintain by assuring programmers that their changes do not adversely affect existing functionality.

In addition to easing implementation via reification, the prorogued programming paradigm promises "test cases for free": the input-output pairs in an IO store can be easily transformed into test cases, in a similar fashion to the “program testing assistant” proposed by David Chapman [**Chapman, 1982**], who recognized the value of collecting and preserving such test cases early on. As the program is being constructed, these collected unit tests can serve as an evolving foundation for the integrity of the program. Moreover, when coupled with outsourcing low-level tasks to junior or outsourced programmers, they serve as a correctness verification mechanism to ensure they have done their job correctly. Further, these test cases are likely to cover important behavior precisely because a developer took the trouble to enter them into the IO store while testing refinements during program construction. Finally, prorogued programming decreases the cost of writing test cases by enabling testers with less knowledge of programming to generate unit tests without having to write any code, just by running the program over and over again with different inputs.

Not only prorogued programming can help collecting test cases, it can be useful in setting up the environment for testing functions. Specifically, prorogued types can be leveraged as an alternate approach in place of mocking frameworks.

Crowdsourcing. By deferring some concerns until after running a program, prorogued programming can open the door to crowdsourcing. A partial program can be shipped to a number of end users who contribute values for the prorogued functions in the program, without having specialized programming knowledge or knowing the internal details of the system. The crowd can be end-users of the program, or gathered rather inexpensively by posting Human Intelligence Tasks on services such as Amazon Mechanical Turk. Contributing translations is amenable to this type of crowdsourcing. Facebook has successfully crowdsourced the localization of its user interface.

```
static void Main() {
  var input = File.ReadAllText("mail.txt");
  var from =
    prorogue GetHeader(input, "From");
  var subject =
    prorogue GetHeader(input, "Subject");
  var body = prorogue GetBody(input);
  if (prorogue IsSpam(body)) {
    Console.WriteLine("SPAM!\n");
  } else {
    Console.WriteLine("From: " + from);
    Console.WriteLine("Subject: " + subject);
    Console.WriteLine(body);
  }
}
```

FIGURE 2.15. Spam filtering with prorogued programming.

2.5.2. Case Studies. Here we use case studies to study the power and utility of prorogued programming.

Spam Filtering. Spam filtering is an excellent example of a problem at which humans can easily identify an instance[4], but a general implementation is tedious. We extend our motivating example (section 2.2) to handle spam filtering and print out "SPAM!" instead of the email body if it believes the message is a spam.

Adding spam filtering to our mail parser example required minimal structural modification to the program and without triggering an abstraction shift. The program is runnable and testable given a small set of input emails. Later, an automated spam filtering function can be written, or a spam filtering library can be used, to complete the program. Spam filtering is an example of a class of applications for which the prorogued programming paradigm is particularly

[4]To paraphrase Justice Stewart, we know it when we see it.

```
MsgStatus Send(string recipient, string msg) {
  try {
    return prorogue Sms.Send(recipient, msg);
  } catch {
    return prorogue;
  }
}
```

FIGURE 2.16. Evolving an existing API.

well-suited. Spell checking, parsing, as of email headers or configuration files, handling CAPTCHAs, and generally any functions that process or tag images, are other members of this class. This underscores the importance of the hybrid computation principle of prorogued programming. Usually, these problems are amenable to crowdsourcing for building the IO store.

Evolving API. By proroguing dependencies, a team working on a client to an API can work and iterate independently from the team that implements that API. Faster, untangled, iteration helps the API client team to have a more concrete idea about what they are going to need from the API implementors, so that they can provide feedback early in the API evolution process. Prorogued programming opens the way to the parallel development of coupled modules, while minimizing the need for coordination. The IO stores capture the behavior each team expects and can be examined by the other team. This extends the paradigm's power to support the execution of partial implementation and evolution of code from the individual programmer to a team of developers. Additionally, prorogued programming can be used to defer the concerns about the exact characteristics of an API, like the exceptions it may throw, to a later time. A common usage pattern for prorogued calls is in the catch blocks, shown in Figure 2.16.

```
void ChargeUser(Card cc, decimal amount) {
  if (prorogue cc.Charge(amount)) {
    var video =
      (int)Session["RequestedVideo"];
    var ip = Request.UserHostAddress;
    var server = FindContentServer(ip, video);
    var key = CreateAuthKey(server, video);
    Response.Redirect(
      GetVideoUrl(server, video, key));
  } else  Response.Redirect("/failed.html");
}
```

FIGURE 2.17. Debugging with prorogued invocations.

Shadowing while Debugging. Prorogued programming gives a programmer the power to temporarily decouple tightly coupled modules and interactively control (via IO store construction) the output of the shadowed method to drive execution as desired. This is especially useful if the call being shadowed depends on time or location the program is being run, or the call is costly, such as a paid web service. Simply prepending an existing call with `prorogue` decouples the caller from the callee and provides a means to inject values into the caller. Decorating a method declaration with the `prorogue` keyword is also supported and is semantically equivalent to annotating all call sites bound to the method with `prorogue`. Another use case of shadowing is bug localization: if the both caller and callee are complex methods and we wish to identify where the problem lies, we can prorogue the call and act as an oracle in between that returns correct values. For example, we can test the example program in Figure 2.17 without actually charging a real credit card, by proroguing the `cc.Charge` invocation in the `if` condition as shown.

Mocking External Resources. Another benefit of prorogued programming is the ability to prorogue external resources, as highlighted in the illustrating example Figure 2.2, we use a database to back our mail program. The program is functional without relying on a real database at all. The programmer is free to experiment with the data model and code; she can also just leave the database concerns to database experts.

2.6. Open Issues

Prorogued programming is the newborn fusion of three principles, *viz* prorogued concerns, hybrid computation, and executable refinement. To date, our focus has been on realizing and experimenting with a prorogued language, Prorogued C#. Much work lies before the fruition of prorogued programming. We must assess its utility and usability, work to identify application domains for which it is well-suited, and explore the evolution of prorogued programs.

Utility. New language features and paradigms are intrinsically hard to evaluate, especially at their debut when there is no data or experience to draw upon. Object-oriented programming (OOP) and aspect-oriented programming (AOP) faced similar difficulties in measuring their impact on programming practice when they arrived [**Kiczales et al., 1997**]. In section 2.5, we followed their lead and used case studies to illustrate the prorogued programming paradigm.

Quantifying the impact of prorogued programming on programmer productivity is an open issue. In general, a prorogued program runs slower than a version without prorogued calls. The magnitude of this slowdown is a function of the number of human interactions. As a developer gradually designs and refines a program during its construction, we contend that this slowdown will be more than offset by the productivity gains of catching errors because of the ease of testing refinements and from not having to write stubs in order to perform testing at all. Related is the question of quantifying the productivity gains from avoiding abstraction shifts.

Usability. User interaction underlies hybrid computation: the human must be efficiently and effectively involved both in populating the IO store and, during reification, in understanding and abstracting it into code. Thus, the user interface of any prorogued language will be critical to its success. Currently, complex objects are displayed in a hierarchical fashion in the user interface. Pattern recognition is a human strength, so IO tables should store and return graphical objects. For example, a human will not be able to solve a CAPTCHA if the interface does not display it. Perhaps we can harness frameworks, like Microsoft's debugger visualizer, to allow a programmer to write renderers for objects within an IO store. In future work, we will release a prorogued language to users, study how they use it in order to improve its realization, notably its user interface.

Understanding what functionality is, or is not, well-suited for proroguing requires more investigation. Human latency and excessive user interaction are two issues here. While it is natural to model human computation as a very slow thread in a concurrent application, not all concurrent applications can tolerate the resulting latency. Regarding excessive user interaction, the pertinent questions are "How many times can the system query the programmer?" and "How complex can each query be?"

A worse case for prorogued programming would be to prorogue a function that adds one to unique numeric parameters, since the human (at least one who did not instantly grasp the pattern) would be tediously involved in every call. For many functions, however, a small set of test cases can drive them to exhibit their critical behaviors. Even when a function has many behaviors, prorogued programming can help a programmer systematically explore subsets of them, by progressively, partially implementing (subsection 2.4.3) the handling of subsets of inputs and requiring the prorogue call to handle only a manageable set of test inputs.

Query complexity is a challenging problem. To begin, we note that, in our use of a prorogued language, we have observed simple queries. A programmer can find a query complex either 1) because it is complex for the human intellect, perhaps merely because of problem size, but not, in principle, for a computer, once an algorithm has been found and implemented or 2) because it is an intrinsically hard problem for both man and machine. Prorogued programming offers nothing special to tackle the latter problem; indeed, no silver bullet may exist. The former problem presents an opportunity: if a user feels a query is too complex, she can give the system a hint and ask for a simpler query that is also useful from the system's perspective. When faced by a complex query, a human can also consult other resources, such as other developers or even an SMT solver. Finally, an approach to the problem of a complex is to again avail ourselves of our principle of hybrid computation and interact with the user to simplify a query.

As with query quantity, a programmer could mistakenly prorogue a function that is better suited for a computer than a human. Clearly, a programmer will not learn very much from testing refinements when spending most of her time populating an IO store. However, these are programming errors, akin to unintentionally writing an infinite loop. Like an infinite loop, these errors can be quickly identified and addressed. For instance, a programmer could partially implement a prorogued call by wrapping it in logic in the host language as described in subsection 2.4.3. In short, the programmer decides whether or not and how often to prorogue some, as yet unimplemented, functionality.

Program Evolution. We conjecture that prorogued programs will typically evolve toward fewer prorogued calls throughout construction. Here, the open question is how to best leverage the knowledge captured in a prorogued function's IO store. One promising direction is to use IO stores as input to program synthesis techniques [**Gulwani, 2010**]. Here, if the IO store is insufficient, perhaps we could again apply our principle of hybrid computation and solicit human help and allow the synthesis algorithm to query the human for additional examples that resolve ambiguities. We all make mistakes; programmers will inevitably incorrectly answer a query and pollute a prorogued function's IO store. How do we allow the user to correct or update the IO store? Can we devise algorithms to detect errors in an IO store with high precision and recall? Finally, as code evolves, the signature of a prorogued method may change. Rather than repopulate that method's IO store from scratch, can we migrate the contents of an existing IO store to the new format?

2.7. Related Work

We are introducing a new programming paradigm, a perilous and ambitious endeavor since few paradigms gain traction. Two paradigms that also sought to change how programmers manage concerns and that succeeded are Object-Oriented Programming (OOP) and Aspect-Oriented Programming

(AOP) [**Kiczales et al., 1997**]. OOP defined a new way to design programs and to modularize concerns. AOP modularizes a concern that OOP did not capture, namely cross-cutting concerns, like logging. Prorogued programming differs from OOP and AOP along all three of its defining principles: by involving humans, its hybrid computation both enables proroguing concerns and experimenting with very fine-grained refinements.

Tinker, by Lieberman and Hewitt, is, in modern terms, an early integrated development environment (IDE) for Lisp that uses interactive memoization to integrate implementation and testing [**Lieberman and Hewitt, 1980**]. Lieberman and Hewitt focus on Tinker's menu-driven code entry and reversible debugger features, although they speculate that Tinker may aid top-down development. Prorogued programming also uses memoization; however, we do so to introduce a programming paradigm that aims to streamline development by allowing a programmer to control the order in which they work on tasks. To realize prorogued programming, we integrated its three features directly into the language, not as an IDE-overlay, and we targeted a statically typed, compiled language, to demonstrate the universality of prorogued programming.

We next describe two projects that share with prorogued programming a focus on enhancing programmer productivity during program construction. DuctileJ is a detyping transformation that allows the execution of type-incorrect Java programs [**Bayne et al., 2011**]. From the domain of dynamically typed languages, it focuses on bringing the ability to execute code at nearly any time to a statically typed language. The motivation is facilitate testing, during program construction while a programmer works to converge on a final, type-correct program. One aspect of prorogued programming shares this focus, *viz* its realization of executable and testable refinements. Beyond this, prorogued programming is quite different. Prorogued programming is about correct, but incomplete programs; DuctileJ is about incorrect programs. DuctileJ is unneeded in a dynamic language; in contrast, prorogued programming is universal.

Angelic programming shows how to employ Floyd's nondeterministic `choose` operator to assist program construction [**Bodik et al., 2010**]. Given an angelic program, an angelic solver controls the output of the `choose` operators within a program to search for safe traces, executions that terminate without failing an assertion. The programmer is expected to reason about the resulting safe traces to gain insight and discover algorithms that allow her to restructure the program toward its deterministic, `choose`-free final version. As with DuctileJ, prorogued programming is also concerned with improving program construction and is otherwise quite different. Prorogued programming relies on hybrid computation to execute incomplete programs, not backtracking or SAT-based solvers. The `prorogue` keyword decorates any function call and can return arbitrary collections of types; the `choose` operator is limited to producing boolean, integer, or address values. To help a developer shift through and control the production of traces, angelic programming relies on assertions. It is not clear that writing these assertions reduces, rather than simply shifts, the complexity of the programming task facing a developer. In contrast, prorogued programming's execution model, other than when it prompts the user to populate a prorogued method's IO store, is standard and allows a more traditional workflow: a developer is not required to (but can) use assertions to control a prorogued program execution. Although not related to prorogued programming's current realization, we note that angelic programming has been adapted to debugging [**Chandra et al., 2011**].

During the evolution of a prorogued program, prorogued functions are typically progressively reified and eliminated. The reification of a prorogued method may be quite difficult; a programmer may not gain very much insight from even a large IO store. All is not lost when this happens of course, since the programmer benefited from deferring the concern. Nonetheless, the tantalizing problem here is to leverage knowledge an IO store contains to ease the implementation of the deferred task. Next we discuss work that bears on this problem of facilitating the evolution of prorogued programs.

Mixed interpreters execute programs that *mix* specification and implementation [**Samimi et al., 2010**, **Milicevic et al., 2011**, **Freeman-Benson and Borning, 1992**].

When it encounters a specification, a mixed interpreter runs a solver and updates the heap with the solution if one is found. Writing specifications can be quite difficult, so mixed interpreters can impose precisely the disruptive abstraction shift that prorogue aims to obviate. During the evolution of a prorogued program, however, the programmer may decide to use formal specification to capture a prorogued function. Can an IO store's input/output pairs facilitate the writing of a specification?

When program sketching, a programmer needs to write only a skeleton, or sketch, of a desired implementation, leaving holes that a synthesizer fills in [**Solar-Lezama et al., 2008**, **Solar-Lezama et al., 2007**, **Solar-Lezama et al., 2006**, **Solar-Lezama et al., 2005**]. Prorogued calls can be seen as these holes. Example-guided synthesis, as its name suggests, uses examples to guide synthesis [**Harris and Gulwani, 2011, Gulwani, 2011**]. Obviously a prorogued method's IO store may be an excellent source of examples for this line of work, which may, in turn, help automate the reification of prorogued functions. Finally, a related line of work, previously discussed in section 2.3, is programming by example (aka programming by demonstration) [**Witten and Mo, 1993, Lau et al., 2003a, Lau et al., 2000, Lau et al., 2003b**]; this work too provides an opportunity for synergism with prorogued programming.

2.8. Conclusion

In this work, we have introduced a new programming paradigm, prorogued programming, founded on three principles — proroguing concerns, hybrid computation, and executable refinements. These principles interlock to form a new paradigm that lets a programmer compile and experiment with an incomplete program that invokes unimplemented functions. A user interface allows the programmer to control the behavior of these functions at runtime if they are actually invoked. This paradigm allows a programmer to prorogue the concern that an unimplemented function embodies and focus on and complete a task at a particular level of abstraction. In contrast, today's languages force the programmer, if she wishes to compile and experiment with their code, to immediately define at least a stub for an unresolved dependency, potentially derailing her train of thought. Prorogued programming also enables a programmer to interact and experiment with their implementation very early in its development, because each successive refinement is executable and testable. We believe that prorogued programming will facilitate program construction and enable new programming workflows that increase programmer productivity.

CHAPTER 3

Invigorating the Programming Canvas with Live Symbols

Programming is about expressing ideas. In mainstream programming languages, the expression of ideas often takes form in sequences of program statements. The composition of a program tokens dictates its behavior to the machine. Pieces of this composition are glued together by *identifiers*. Identifiers tell the compiler how statements relate to each other and how data and control flows throughout the program. While this is the principal purpose of identifiers from the machine's perspective, they serve a more important purpose when it comes to the interaction of the programmmer with the program. They are critical from the standpoint of program understanding and navigation to the point that one of the most primitive jobs of most program obfuscators is to rename identifiers.

Observation of obfuscated programs also shows us that the two purposes of identifiers are distinct. You can have a working program that is very difficult to understand. Programmers usually overload the mechanical purpose for identifiers and use them as signposts to aid with program understanding.

In this chapter, we propose consciously leveraging identifiers as hooks not just for gluing pieces of a program together, but as terminals to interact with the programmer, program documentation, knowledge external to the program, the programming environment, and serve as tools to manipulate the program itself in specialized ways. We illustrate how such systems would look like and explore the opportunities, and suggest solutions to some of the challenges faced in realizing such programming environments.

3.1. Introduction

Mainstream programming often involves typing and manipulating characters in a text editor. The goal is, however, rarely to manipulate individual characters. When modifying a program, we generally want to alter higher level chunks of the code. Traditionally, this problem has been alleviated by more powerful batch character manipulation tools. However, the programmer still has to map an intended higher-level change to deltas in character sequences, which these tools manipulate. The more savvy programmer might invoke a text editor shortcut to execute the delta in as few keystrokes as possible.

Ironically, even our compilers (for the most part) deal with higher level things than raw character sequences, such as abstract syntax trees.

To fix this problem, we need a better canvas for composing our ideas — we want to modernize our program editors and make them smarter. In this chapter, we outline our vision for how a first step towards that path might look like.

3.1.1. Main Contributions. This chapter makes the following contributions:

- We introduce Live Symbols as our vision for a modern programming canvas.
- We discuss our design, and the assumptions and choices made in the design process.
- Through a collection of case studies, we illustrate the power of our modern programming environment.

The rest of this chapter is organized as follows. We first discuss our design in section 3.2. Then we use a few examples to show its utility (section 3.3). Finally, section 3.4 discusses related work and section 3.5 concludes.

3.2. Design

Our vision for a new programming environment is based on the following assumptions:

- The current mainstream programming style is not the best way to solve all problems. That said, it might still be suitable for a great many of them, and many people are comfortable with it.
- We are not limited to keystrokes to interact with our programming workspace.
- Sometimes, applying tools on a problem is better than building complex abstractions.

With those in mind, we describe the pillars on which our design stands.

- **Contextual Extensibility:** We should be able to extend the environment by defining characteristics and operations on specific identifiers. These operations are defined alongside the type "package" and are loaded on-the-fly, as opposed to plug-ins preinstalled in the environment.
- **Domain Specificity:** Parts of the program that express domain-specific problems should be expressed in domain-specific languages or tools.
- **Interactivity:** The programming environment should be interactive whenever it makes sense and can leverage modern input/output technologies.

Live symbols work by defining *design-time metadata* and *operations to types or other program entities*. The environment uses the metadata to:

(1) display buttons and other user interface elements to invoke those custom operations on identifiers affected, *e.g.* display a color picker beside a value of type `Color`. Of course, a live symbol can handle arbitrarily complex interactions with the symbol in the programming environment, as long as they supply the necessary code.
(2) customize visualization of certain values or blocks of code in the program, like displaying a colored rectangle beside a value of type `Color`.

By giving arbitrary extensibility to live symbols, they can selectively pop up windows, providing tools at the programmer's fingertips, and perform complex interactions with the programmer. They can selectively generate code or customize code around their context as they respond to user interactions.

3.2.1. Realization. In practice, the design of a programming environment supporting live symbols depends on many factors. One of the most important factors is the programming language it is going to support. Here we discuss one way to design such programming environment for a typical object-oriented, imperative, programming language.

To engage with the extensibility hooks of the environment, the program entity (usually a class definition) that wants to provide interactive functionality for its users should annotate itself with a reference to a function or type that will serve as a plug-in to the environment that will be loaded on-the-fly when the type is used. Alternatively, a standard naming convention can be used in place of an explicit annotation that lets the environment to find the extensibility procedures.

Once a extensibility handler for the specific program entity is loaded from the library package, it can communicate back and forth with the environment via its normal plug-in mechanisms, getting information about the code context, displaying embedded widgets within the text editor or popping up external dialogs, and possibly inject code into the program itself.

3.3. Applications

To demonstrate the potential impact of live symbols on software engineering practice, we present a few case studies and applications.

3.3.1. Classifier Trainer. Assume that we have a piece of code for a basic neural network and a data set for training it. Without live symbols, we probably need to invoke a program offline, train the neural network using the data set, and incoporate the learned weights in the final program's source code before shipping it to the user who needs the classifier.

With live symbols, it could be as easy as starting typing:

```
classifier := NeuralNet{}
```

Depending on the exact user interface implementation, there could be a down arrow displayed when we finish typing `NeuralNet`, clicking on which will provide us with the option "Train from data set", clicking on which would ask for a file containing training data. We would simply load the file and the neural network will train itself offline and embed the final weights in our code under the hood. It is possible because the NeuralNet library has provided the code necessary to accomplish this and the environment has simply invoked that extension.

3.3.2. String Pattern Matching. Imagine having a live symbol enabled pattern matcher. Analogous to the neural network example, it could pop up a dialog that lets us experiment with various regular expressions. Alternatively, in that user interface, it could give us an option to discover a pattern based on a few examples that we provide, using program synthesis [**Le and Gulwani, 2014**].

3.3.3. Color Picker. The programming environment is able to visualize the color based on the arguments given to the constructor of a `Color(245, 255, 1)` type.

Additionally, it would be able to display a color picker when we click on the value in the editor and rewrite the underlying code when another color is picked.

3.3.4. Spreadsheet-Assisted Computation. Imagine writing a program that depends on a model designed by a domain expert who has implemented it in an spreadsheet. With a `SpreadsheetEvaluator` object, we could immediately open a spreadsheet editor from our programming environment, pull up the model developed by domain expert, pick cells that represent inputs and outputs to the model, and plug it in to certain variables in scope and we will be all set. This is possible because the `SpreadseetEvaluator` class is packaged with code necessary to invoke the spreadsheet engine and the user interface to generate the appropriate hooks when invoked as a live symbol.

3.3.5. Image Filter Application. If we want to write a program that loads a set of images and draws an X on each of them, traditionally we need to write a loop and then use drawing statements in our favorite graphics library to draw the X. With live symbols, the ImageFilter object can open up a Photoshop like environment where we apply a bunch of modifications to a sample image and it will learn the modifications, and initialize the object in our code accordingly. Then, calling the `Apply` method on our object will reapply the sample operations on the actual bitmap passed in at run time.

3.3.6. Querying a SQL Database. In a database client application, we need to issue a complex query, but we are not entirely familiar with the database schema. Invoking the "Design Query" operation associated with the database command object will not only bring up a designer and let us visually construct the query and test it, but also when we are done with it, will show the embedded SQL in code, but with syntax highlighting! Because the SQL command object comes with the design time operation to visualize the query appropriately, it can display the SELECT statement with proper syntax highlighting visually embedded right into the host language code editor.

3.3.7. LaTeX Editor. In a LaTeX programming environment supporting these techniques, we can define a set of commands for say, defining tables, and annotate the command definition to refer to a class that extends the environment by providing a visual table editor. When `\NewTable{}` is typed, the environment recognizes it and starts by looking at the source containing its definition and picking up its annotation referring to a function provided alongside the LaTeX library and invokes it. The function will then interact with the environment, presenting the user with a live table editor, embedded in the middle of the text editing environment.

3.4. Discussions and Related Work

A large body of work in various areas such as programming languages, integrated development environments (IDEs), interaction design, and end-user programming, have served as inspiration, are related to, or have the potential to be improved by the ideas presented in this chapter. In this section, we highlight several of those to better contextualize the work and put its potential in perspective.

3.4.1. Extensibility. Many of the popular text editing tools are extensible. Most notably, Emacs and Vim have over the years developed a vibrant community of users who write scripts that extend the editor in many ways and share them online, turning them into incredibly powerful tools. In the case of Emacs, many of the extensions have little to do with coding and are basically applications written to run in the Emacs environment, flashing a spotlight on the broad range of extensibility facilities offered by that environment. Many other extensions focus on making text editing more efficient and powerful, and are broadly useful for editing all kinds of textual data. There are language-specific extensions that do syntax highlighting and navigating through code, most of which are accomplished by quickly parsing outer levels of productions specified in language grammars (and often, to make it fast, finding a regular approximation of the grammar and simplifying parsing). Anecdotally, the main class of extensions that take the semantics of the code written into account are the ones that enable autocompletion. Live symbols can bring on-the-fly contextual extensibility to these editors.

3.4.2. Integrated Development Environment Features. IDEs such as Eclipse and Visual Studio come with a bunch of tools and services in addition to the main text editor. Similar to live symbols, some of these services help with domain-specific tasks like visually designing dialog boxes or database queries. However, unlike live symbols, these services are rarely blended directly into the text editor and feel more like separate tools bundled into the IDE and accessible via a shortcut. There is often an impedance-mismatch between the artifacts generated by those IDE services and the textual code. While the overarching vision of live symbols is hard to spot in mainstream IDEs, sparks of it can be seen nevertheless. For instance, Microsoft Visual Studio supports a feature dubbed “visualizers” [**Microsoft, 2015**] in its debugger, that lets the programmer write code to meaningfully display objects of that type when inspected in the debugger at run time. Similarly, the form designer supports customizing the design-time behavior of visual controls dropped into a form via metadata annotations on the custom control type.

There are other environments that have a fixed and limited set of domain-specific functionalities that live symbols can offer. One example is Android Studio [**Google Inc., 2013**] is an IDE for targeting the Android platform, that supports displaying actual localized strings within the text editor, embedded in code, in place of the "real" code that consists of a locale-independent resource key. Another example is Codea [**Two Lives Left, 2014**], a programming tool for iPad based on the Lua programming language specifically targeted at making games and simulations. When a user touches a value of a small, fixed, set of types, like colors and images, in the program text, Codea brings up a color or image picker, that lets the user easily manipulate the value. Live symbols offers a universal way to add these features to the environment without the need to hardcode them into the IDE or offering them as separately shipped plug-ins.

3.4.3. Live Programming. In some dynamic programming environments like Self [**Smith and Ungar, 1995**] and Smalltalk [**Goldberg and Robson, 1983**] development can happen in a living environment, in which program objects can be instantiated and manipulated and can potentially interact with the environment itself to serve as tools for the programmer, like live symbols do.

Bret Victor's "Inventing on Principle" talk [**Victor, 2012**] inspired the community to strive to bring in a more immediate connection between the programmer's ideas and the final product. It triggered a renewed interest in live programming environments. In particular, Light Table [**Kodowa, Inc., 2012**] and Xcode [**Apple Inc., 2014**] (via its Playgrounds feature) have strived to bring that vision closer to reality by displaying the runtime values of variables alongside program statements.

Live symbols are complementary to live programming. Raw runtime data can be too low level to be meaningful to the programmer. Live symbols can help abstract the raw data in a type-specific fashion and present it to the programmer in a concise and easy to comprehend way. Moreover, in the absence of actual runtime data, *e.g.* in a complex distributed system where it is hard to keep the complete system live, live symbols can ease understanding of the system just by looking at the code statically and presenting a static model of what would happen at runtime. Lastly, live programming environments do not necessarily provide the ability to use the live objects as interactive tools that help the programmer with the code. They often merely try to help the programmer tell what the code does, not more.

3.4.4. Hybrid Computation. Prorogued Programming [**Afshari et al., 2012**] is a programming paradigm that allows programmers to compile and run incomplete programs so they can experiment with and refine work-in-progress. It works by decorating invocation expressions in a program with the **prorogue** keyword and letting the compiler know that it is okay if the callee does not currently exist. At run time, if a prorogued call is encountered for the first time, it will pop up a dialog and asks the user to supply a return value for the program to continue execution, which it then caches and persists for future runs of the program if the same invocation is encountered with the same set of arguments. While prorogued programming is mainly concerned with giving the programmer the power to freely reorder concerns by untangling dependencies across program constructs, it also enables hybrid computation. Ideas highlighted in this chapter synergize with prorogued programming along two dimensions: first, they can empower the interactive query dialog when a prorogued call is encountered at runtime, by providing enhanced custom visualizations for the arguments passed to the function, and by enabling the programmer to construct complex custom objects by leveraging the same type-specific tools and operations that the object type carries with it; second, it extends the opportunity for hybrid computation from just leaf level functions when the program is run to the entire programming environment, by connecting external "computation engines" to the program via the tools provided by live symbols.

Live symbols make hybrid computation seamless, and may effectively make alternative ways to express ideas to a computer such as programming by example [**Halbert, 1984**] mainstream, allowing to switch from one form of expression to another, depending on the suitability to the task at hand.

Systems like MATLAB [**The MathWorks, Inc., 1984**] that are designed mainly for scientists, engineers, and non-professional programmers to accomplish domain-specific tasks across many domains are arguably instances of hybrid computation. It is conceivable that such systems could be implemented with a set of libraries by taking advantage of a programming environment that supports live symbols. Domain-specific toolboxes would be reduced to libraries and there would be little differentiation between a toolbox's "applications" and the functions provided by it.

3.4.5. Metaprogramming. Macros [**Leavenworth, 1966**] can be thought of as primitive, non-interactive, stripped-down version of live symbols. While they are not interactive, they can be thought of as tools for code generation. They are similar with live symbols in that they are a form of extensibility that gets carried over from the program being edited, not by direct installation in the environment, making them contextual and simple to add with little marginal cost.

A domain-specific language [**van Deursen et al., 2000**] (DSL) is yet another way to increase expressiveness of a program (or piece of a program) when it targets a specific problem domain. When working with a DSL as part of a program written in a mainstream language, the connection between the two becomes a challenge. To prevent this problem, a number of DSLs try to restrict themselves to valid syntactic constructs in the host language and merely "trick" the host compiler into generating the code that behaves as they want [**Ghosh, 2011**]. This resolves the embedding issue, but restricts the DSL to the syntax and semantics of the host language. Integrating free-form DSLs [**Bravenboer and Visser, 2004**] into a host language is possible, but the downside is that the host language compiler needs to be modified to support DSL hosting. Like macros, embedding a DSL without the appropriate tooling on the environment side, will result in a non-interactive solution that partially addresses the same problems as live symbols. Live symbol implementations can use this technique under the hood to persist the source files.

3.5. Conclusion

In this chapter, we described a novel way to think about the programming canvas, making it richer, more interactive, more contextual, and wary of the code being written. Transformation of the code editor into a hybrid canvas that hosts various tools opens up opportunities for lightweight and fast innovation in program construction tools. Furthermore, this transformation can change the traditional code editor from a tool used primarily by professional programmers into a versatile tool for creation and computation to accomplish domain-specific goals, via lightweight toolboxes that are simply “packaged libraries”. Finally, it sets the stage to escape from the traditional, mostly text-based, environment for creating programs, and toward more modern ways to create programs, taking more advantage of input and output devices other than the teletype and its modern incarnations and emulations. This is especially important considering the increasing popularity of touchscreen tablets and phones; they are the devices that the next generation of programmers grow up with.

We believe the path that takes us towards that vision will make program construction easier and the programmers more productive, and is an exciting avenue for future research.

CHAPTER 4

Building White-Box Abstractions by Program Refinement

Abstractions make building complex systems possible. Many facilities provided by a modern programming language are directly designed to build a certain style of abstraction. Abstractions also aim to enhance code reusability, thus enhancing programmer productivity and effectiveness.

Real-world software systems can grow to have a complicated hierarchy of abstractions. Often, the hierarchy grows unnecessarily deep, because the programmers have envisioned the most generic use cases for a piece of code to make it reusable. Sometimes, the abstractions used in the program are not the appropriate ones, and it would be simpler for the higher level client to circumvent such abstractions. Another problem is the impedance mismatch between different pieces of code or libraries coming from different projects that are not designed to work together. Interoperability between such libraries are often hindered by abstractions, by design, in the name of hiding implementation details and encapsulation. These problems necessitate forms of abstraction that are easy to manipulate if needed.

In this chapter, we describe a powerful mechanism to create *white-box abstractions*, that encourage flatter hierarchies of abstraction and ease of manipulation and customization when necessary: program refinement.

In so doing, we rely on the basic principle that writing directly in the host programming language is as least restrictive as one can get in terms of expressiveness, and allow the programmer to reuse and customize existing code snippets to address their specific needs.

4.1. Introduction

Programming is about expressing ideas. Ideas expressed in programs vary widely in complexity. Programs can exhibit small ideas or very complex ones. Complex ideas are built from simple ones. There are three ways to build complex ideas from simple ones: by combining them into a compound one, by comparing them with each other without unifying them, and via abstraction, *i.e.* distancing them from other ideas that accompany them in their concrete existence [**Locke, 1689**].

Complex programs, like ideas, are generally composed of smaller pieces. In order to make building complex systems tractable, and to be able to reuse these smaller pieces in different contexts, we need to rely on abstraction. Programming languages provide various ways to build abstractions, like procedures [**Abelson and Sussman, 1996**], abstract data types [**Liskov and Zilles, 1974**], classes [**Dahl et al., 1968**], objects [**Goldberg and Robson, 1983**], and actors [**Hewitt et al., 1973**]. Different programming languages provide different means of abstraction.

In practice, software systems often consist of complex abstractions composed of other complex abstractions, forming a deep hierarchy of abstractions, created by different people at different times to achieve different goals. While some of the nodes in this hierarchy are essential to the program, a deep hierarchy of abstractions have some obvious downsides.

One issue is imperfections of the abstractions themselves, *i.e.* they do not fully abstract away the related ideas that accompany the underlying concrete instantiations of the abstraction and the subtleties *leak* to the higher level observer, making it responsible for specifically working around such leaks, thereby hindering the reusability of the abstraction in arbitrary contexts.

Another concern is understandability of programs relying heavily on complex abstractions: by design, many of the language features and techniques to build abstractions, *e.g.* procedural abstraction and object orientation, aim to build *black-box* abstractions. Black-box abstractions are double-edged swords. The advantage of hiding the internals is that the component can be isolated and reasoned about as a separate unit with clear interfaces and boundaries. The disadvantage is that the interfaces can be arbitrary and lacking documentation, or worse, having incorrect documentation that does not perfectly reflect the subtleties of the implementation, causing confusion for the programmer. Anecdotally, sometimes reading the source code for the component, if available, can be the best path for understanding the subtleties of the implementation. Deep abstraction hierarchies can make this more difficult.

4.1.1. Abstraction by Refinement. In this chapter, we introduce a new mechanism for building abstractions: program refinement.

The core idea is treating a certain procedure (β) as the *base* template and letting the programmer modify it as they wish. The new, specialized, procedure (Γ) can be formally described of as a pair consisting of the original base procedure and its differences (Δ).

$$\Gamma = (\beta, \Delta)$$

Δ is the refinement applied by the programmer and captures the intent of the specialization on the base template. The way Δ is interpreted is implementation-dependent. In the simplest implementations, it can be a syntactic difference provided by a version control tool.

For it to be a proper abstraction, we need to be able to liberate Δ from being meaningful only in the context of that particular base, and be able to apply it to other base templates as well, computing a new specialization with the same delta over a new base template. This is formalized by a *merge* operation:

$$\Gamma_2 = \mathit{merge}(\Gamma, \beta_2) = \mathit{merge}((\beta, \Delta), \beta_2)$$

The actual behavior of a merge operation is also implementation dependent. In the simplest case, it is a version control-like syntactic automerge, but it can be made smarter and more semantic-oriented. The smartness of the merge operation is, in a way, representative of how capable the programming system is in capturing the programmer's intent.

A single base can serve as the template for many specializations. There can, in principle, be a nested tree of specializations. When the root base changes, the changes would propagate by applying successive merge operations. The merge operation can be unsuccessful. We will leave the discussion on how we resolve this issue to section 4.3.

We have used this simple formalization to describe the idea as an analogy to another known idea, and the differences between the two.

At this point, the description might sound similar to a macro system, or a template metaprogramming feature in a language like C++. Subtle, but key, differences, however, exist, as we describe.

Refinement vs. Macros. Macros are powerful abstraction tools. Similar to refinement, they offer specialization from a symbolic template. Macros, in languages that embrace them, like flavors of Lisp, operate at the abstract syntax tree level, and therefore make the full power for the underlying language accessible to the programmer. However, there are two key differences:

(1) Macro expansions are evaluated in the scope of the use site. In this manner, they are similar to copy-pasted code. A key feature in refinement abstractions is evaluation within the environment of the original base template.
(2) Macros need to be predefined. A programmer generally needs to think beforehand about what macros to write, and provide appropriate "holes" in them for the external arguments. The programmer would

often overgeneralize the macro before the complexity is actually needed in the program. The opposite can also occur, where the macro definition does not support parameterization of certain parts of itself. Clearly, arbitrary procedures do not become macros automatically, but you can apply refinement to any procedure in the program, without any special consideration when the procedure is being authored.

Refinement vs. Metaprogramming Templates. Metaprogramming template systems vary in design and function. To address the differences, we take C++ template system as a popular, concrete instantiation. In contrast with the C++ templates:

(1) Refinements, being syntactic, are constrained by the expressiveness of the host language only. Templates, however, can only be parameterized in certain areas. An arbitrary statement cannot be fed into a C++ template as an argument. The parameterization potential is usually limited to types, values, and function references.
(2) Refinements can be applied to any base procedure, whereas templates, like macros, need to be predefined as such.
(3) Depending on the way the template system is implemented, its expansions can exhibit the second limitation described for macros, *i.e.* redefinition of the environment in which the template is expanded.

4.1.2. Main Contributions. This chapter makes the following contributions:

- We introduce a new general paradigm for building abstractions by allowing the programmers to refine existing code.

- We present and discuss the design choices in GoCLR, our development environment for Go featuring abstraction by refinement.
- We illustrate the usefulness of this abstraction toolkit through a collection of case studies.
- We discuss open issues, such as usability, challenges in merging, interactions with external editors, and possible approaches for resolving them.

4.1.3. Chapter Outline. The rest of this chapter is organized as follows. First, in section 4.2, we use an illustrating example to motivate building abstractions by refinement and illustrate its use. In section 4.3, we describe our design and realization of a system with support for building abstractions via program refinement for a real-world language, Go. We then use a few examples in section 4.4 to highlight the utility of the paradigm. In section 4.5, we discuss a few open issues. Finally, section 4.6 surveys related work, and section 4.7 concludes.

4.2. Illustrating Example

To motivate and illustrate the utility of white-box abstractions created with program refinement, we use an illustrating example.

Imagine using a package that implements some graph operations, among other things. The package contains a public DFS function that does a depth-first traversal of a graph passed via a root node as an argument and marks them as visited. The end result is that all reachable nodes are marked as visible in some state variables internal to the package for future use, for instance to check graph connectivity.

```
func DFS(root *Node) {
  q := Stack{}
  q.Push(root)
  for !q.Empty() {
    node := q.Pop()
    if !visited(node) {
      markVisited(node)
      for adj := adjacentNodes(node) {
        if !visited(adj) {
          q.Push(adj)
        }
      }
    }
  }
}
```

FIGURE 4.1. The original depth-first search function provided by the library.

A programmer using this package is interested in the depth-first search functionality (Figure 4.1), but needs to perform a custom task when a new node is visited, like printing its satellite data.

Had the original author of the `DFS` function had the foresight that it would be used this way, they would have provided a generic way to pass in, say, a function pointer to the `DFS` function (Figure 4.2). The caller would have then supplied a function that takes a node and processes it as an argument to the `DFS` function. Note that providing this functionality is only possible if the underlying language has the required bells and whistles, like the ability to pass functions as arguments. Furthermore, this approach limits the degree of freedom of the client to intervene at the specific point after `visit` is called in the function. Any other functionality would still be unsupported. Realistically, the caller might

```
func DFS(root *Node, look func(*Node)) {
  q := Stack{}
  q.Push(root)
  for !q.Empty() {
    node := q.Pop()
    if !visited(node) {
      markVisited(node)
      look(node)
      for adj := adjacentNodes(node) {
        if !visited(adj) {
          q.Push(adj)
        }
      }
    }
  }
}
```

FIGURE 4.2. Depth-first search procedure extended to support a custom processing via a function reference.

want to use the DFS code to find back-edges in the graph, which requires more changes to `DFS` than just being able to pass the callback that would be run on every visit.

Nevertheless, the original author has not provided us with this functionality. We are stuck with a decision to copy and paste the `DFS` source code or modify it in-place.

There are a number of problems with explicit copying and pasting. First, the programmer needs to figure out where to paste the copied code. If the code is pasted in the caller context, it will not compile, because its identifiers refer to dependencies that are meaningless in the caller's scope. Therefore, the programmer needs to manually resolve the references, if possible. It is not always possible due to private identifiers within packages.

Pasting the code in the callee context is essentially forking the function into two. The first downside is doing that means you are essentially forking the dependency library and updates to the dependency will not be as straightforward to use from then on. Second, even with a good version control system, the updates and bug fixes to the original `DFS` would not propagate to the cloned implementation.

Modifying the code in-place has the obvious downside of potentially breaking the existing clients who are relying on the subtleties of the existing behavior of the function.

With our system, this problem is easily fixable by right-clicking on the `DFS` identifier in the programming environment. The system will let you specialize `DFS` function for that specific call site. That way, you can modify the body at will and add appropriate statements wherever needed. The language does not even need to support function pointers.

If the original library changes upstream, the system will automatically try to merge the specialized versions of the functions with the updates to them fetched from the upstream package source. Should the merge succeed automatically, the update would be seamlessly applied to all of the specialized versions of `DFS`.

Importantly, the visual footprint of the specialized `DFS` is confined to that particular caller only. It would not be visible when browsing the source code of the dependency package.

4.3. Design and Realization

To experiment with building abstractions by program refinements, we designed a prototype system, GoClr (pronounced “go clear”). In this section, we describe some of the design challenges we faced and how we tackled them and the rationales behind our choices.

4.3.1. Programming Environment. GoClr has its own custom programming environment that is based on the Go programming language [**The Go Authors, 2009**] and Git version control system [**Torvalds, 2005**] internally. Go was chosen as the programming language for the following technical and conventional reasons:

- Go is designed to understand the need for external packages and tools for package management.
- Go packages are conventionally distributed as source code and dependencies are often compiled in a static binary at build time.
- Go is a simple language and lacks many of the conventional mechanisms to build abstractions, like generics, making it a particularly suitable testbed for building abstractions with program refinement, due to a more acute need for alternate abstraction mechanisms, and minimal potential complexity arising from interference with existing language features.
- The Go community seems to strongly prefer lightweight abstractions and has a tendency to more strongly resist overengineering than some other mainstream languages.

4.3.2. Projects and Build Strategy. For simplicity, the initial version of GoClr is not designed to actively interoperate with other development environments. GoClr normally stores program pieces as separate Git objects and generates textual Go source files to be fed to the actual compiler toolchain only when the project is going to be built. For practical purposes, and for seamless working with dependencies, an import mechanism is provided. Importing existing Go source packages will parse and transform them to the internal data format, while preserving the connection between the original location of tokens and the abstract syntax tree. This connection is necessary to identify and propagate changes to the program when a dependency is updated, for instance.

4.3.3. Merging. A necessary feature for realization of our vision is support for propagation of changes when a piece of code that serves as the base for one or more refined specializations changes, either at the source code level in an external Git repository serving the dependency, or within the current GoClr project.

The prototype implementation of GoClr only supports simple automatic syntactic merges by piggy-backing on Git itself. While accurate and sensible automerging is a distinct problem from the general idea of abstraction with program refinement, a smart merge subsystem is critical for a good programmer experience, especially as the project and as a result, the quantity of specializations grow. To help solve this problem, we envision providing API hooks for smart mergers that can understand semantics of the differences and the programmer intent behind changesets. Such smart merge tools can then automatically reapply the modifications inferred on the new base version.

Of course, it is possible for automatic merge tools to fail. When that happens, the user has the option to manually do the merge—by effectively rewriting the specialization on a new base—, or disentangle the specialization from the base piece of code, thus creating a new copy of the code. Obviously, future changes to the base piece will not be propagated to the distinct specialized copy anymore and the new piece takes on its own independent life.

4.3.4. User Interface. The user interface is a critical piece of the solution. Specialized versions of the code should be hidden from the programmer except when they are explicitly looking for them or they are interested in a particular specialization relevant to a specific call site. Otherwise, the clutter caused by the visibility of many variations of a single procedure will make the system unbearable: imagine C++ programmers having to see template expansions for each specialized type.

It is also of utmost importance to properly highlight and indicate that a callee is specialized for a particular call site. GoCLR will let you provide a short comment when specializing a base template that will be visible to the reader under a specialized function name in the call site.

4.4. Applications

Building abstractions by program refinement is helpful in various ways to the programmer. In this section, we discuss a select few of its potential applications.

Debugging Aid. Often in debugging scenarios, the programmer might be interested in temporarily customizing the functionality of a procedure in a specific invocation, without having it behave differently for the program at large. GoCLR lets the programmer do exactly the customization they need per individual call site. The programmer can choose to customize a specific procedure when called from a specific location and add diagnostics and print statements to aid debugging, for instance.

The changes will affect only calls originated from a specific call site and the rest of the program executes as it normally would.

This debugging technique can be applied even if the programmer does not expect to specialize the function permanently. To get back to the original functionality, the programmer can just revert the specialization within the programming environment.

Customizing Generated Code. Automatic program synthesis tools and code generation tools expect the generated code file to not be edited manually, because the edits would be lost if the code generation tool is run again. Therefore, generated code is usually kept in a separate file and maintains a clean, minimal, interface to the other pieces of the program that are non-generated. This style may make sense for tools like parser generators, that have a very clear, isolated, functionality, but they effectively discourage the use of a class of automatic programming tools that require more customization on the output produced and are more entangled with the host program.

Naturally, we can simply consider the generated code a separate dependency and propagate changes in the output of the code generator to specialized versions of functions that are based on pieces of the generated output.

Exposing Hidden State in Dependencies. There are times when excessive focus on encapsulation cause problems. For instance, a concrete problem with the TLS package in older versions of the Go runtime library was the lack of an exposed connection identifier. In order to perform meaningful authentication over an established but unauthenticated TLS channel, while preventing man-in-the-middle attacks, you need a way to bind the underlying TLS channel to the higher level authentication sequence. There used to be no easy way to extract a connection identifier from the Go runtime library's TLS package. Forking that piece of the library and manually adding a method that exposes the internal state variable is a way to accomplish it, and it is a huge burden. Luckily, with a simple program refinement technique, a method can be effectively added to the package that reads the connection ID from the internal state variables and returns the value for use by the caller, effectively circumventing the overly strict encapsulation policies of the package, for good reason.

Lightweight Forking. Considering the vast variety of freely available source code on the web, sometimes all the programmer wants is to write a program whose functionality can leverage a subset of another program, with minor additions and differences. For instance, a static analysis tool might be based on a compiler toolchain that was not intended to be used as a library. GoClr can be used to help the programmer extend and manage the fork without severing the ties to the original program, *i.e.* future changes and bug fixes in the original program can still propagate through the derivative.

4.5. Open Issues

Refining programs is fundamentally a new way to define abstractions. The GoClr is in prototype stage and work should to be done to ensure seamless cohabitation of this concept with other language features present in more featureful languages. Furthermore, we must assess its utility and usability, work to identify the applications for which it is most useful and effective and how to ensure we maintain a delightful user experience.

Effectiveness. The utility of programming languages, techniques, paradigms, and tools are often subjective and difficult to evaluate. This is especially true for new paradigms and ideas that have not been widely applied. Established paradigms like Object-Oriented Programming and Aspect-Oriented Programming [**Kiczales et al., 1997**] faced a similar issue. In section 4.4, we follow their footsteps [**Kiczales et al., 1997, Afshari et al., 2012**] and illustrate the utility of using program refinements to infer abstractions with a few case studies.

Empirical studies are needed to quantify the impact of availability of different ways to build abstractions on programmer productivity. Unfortunately, a meaningful empirical study is hard to do before widespread adoption of a paradigm. While we believe there are compelling use cases for building abstractions on top of program refinements, there are concerns about syntactic modifications leading to the prolification of divergent specialized versions of a procedure that hardly resemble their original base and it may prove to be hard to reason about them as a general, unified, thing, which might lead to adverse effects on programmer productivity. Quantifying such effects is an open issue.

Interaction with External Editors. In order to capture refinements, propagate changes, and present the appropriate code specializations in their right context, the programming environment needs to store some metadata. In our implementation of GoClr, this metadata and the associated code is not meant to be modified outside the environment, therefore the developer is mostly confined to the GoClr editor. In order to resolve this problem, a standard format for persisting the specializations and the appropriate links and metadata should be developed. Even with such standard format that could be supported in alternative editors, some programmers strongly prefer sticking to their favorite plain text editors without additional functionality. It is conceivable that a useful implementation of the concepts presented in this chapter would require a smart editor to be effect. Not supporting plain text editors can hinder its adoption among some programming circles, therefore research into adapting the techniques to a text editor and command line tool-based environment remains an open issue.

Merge Conflicts. While syntax oriented merges can work well when the changes are spread away, as they become more granular, too many conflicts start to emerge. The burden of resolving conflicts is enough that if they are frequent, it will discourage people from using the system.

Programming language-aware merge tools can help alleviate this problem because they can take a more semantic oriented view at the language and do a better job at merging. That said, the merge problem is definitely one that has a lot of room for improvements.

4.6. Related Work

Kiczales [**Kiczales, 1992**] identifies the issue of leaky abstractions and the necessity for being able to reach into them at times. He observes that in practice, the implementation cannot always be hidden, citing performance characteristics show through in significant ways as an example of how abstractions can leak. This work discusses the deficiencies in mainstream abstraction frameworks and suggest application of a *metaobject protocol* technology to resolve the problems. The metaobject protocol is a reflection mechanism that lets the client reach into an abstraction and alter its behavior. In dynamic environments like Ruby and JavaScript, the “monkey patching” technique is commonly used to swap a value of an object property or a method body at run time to achieve the desired results. Reflection is often limited in its power in more static environments and commonly these all the prior techniques operate at the granularity of a method at best. In comparison, program refinement can work by syntactic manipulation of statements within method bodies. Since it is a syntactic tool, its power is effectively only limited by the expressivity of the host language itself.

Domain specific languages [**van Deursen et al., 2000**] provide an alternative path to managing complexity without the need to build a deep hierarchy of abstractions. Domain specific languages that are implemented as code generators synergize well with program refinements.

Embedded domain specific languages [**Bravenboer and Visser, 2004**] are basically language extensions for which the parsing is handled by some of the objects used in the program, depending on the context. These languages increase expressiveness and concision of programs, but still require careful upfront thinking by the library author. Of course, program refinement is not confined to any particular language, so in principle, the domain specific parts of a program can also be refined and specialized.

There is a body of work related to detecting cloned code [**Kim et al., 2005, Jiang et al., 2007**] and automatically propagating patches through them [**Toomim et al., 2004**]. One distinction of these systems from our work is that we do not increase the footprint of the codebase, whereas copying-and-pasting excessively increases the code size and visually cluttering to the programmer.

4.7. Conclusion

In this work, we have introduced a new, generic, way to build abstractions by program refinements. Program refinement is a powerful tool for building many forms of abstractions because it is limited only by the expressiveness of the host language.

The key insight about basing abstractions on modifications at the syntactic level is interpreting such changes in the context of the original definition, as opposed to the caller's scope, while the effect would be limited to a specific call site. This gives the programmer implementing the caller an easy way to reach into the implementation and customize the concrete code behind an abstraction to achieve the desired effect that is executed when necessary.

We believe that the ability of building custom abstractions via arbitrary syntactic manipulation of code is a powerful tool that can alleviate the need for narrower, more specific, abstraction tools that exist in some programming languages, and liberates the programmer from fighting with abstractions that confuse the programmer down the line and hinder program understanding and programmer agility.

CHAPTER 5

Conclusion

Programming is one of highly leveraged human activities in the modern society and it seems its importance is only increasing in the forseeable future, as computers get more deeply integrated with the fabric of lives. Therefore it is important that the programming experience keeps up with the new shapes and forms of computers, be more accessible to a wider audience, and makes the professional programmer more effective as well.

In this dissertation, we started by highlighting transforming ideas into software as a high level, long-term mission for software engineering, surveyed the work that has been done along that direction, and identified various paths forward to realize this long term vision.

In the next chapters, we further explored three distinct technqiues to concretely move forward in reimagining the modern programming experience. In chapter 2, we introduced a new programming paradigm, prorogued programming, to better fit the programmer's workflow and reduce distractions. In chapter 3, we described Live Symbols, an interactive technique to enhance the development environment to adapt itself to various context-specific tasks that can potentially look little like traditional programming, thereby making it easier for domain experts to leverage computing resources to accomplish their domain-specific goals, making programming more accessible to a wider audience beyond professional programmers. Lastly, we identified complicated layers of black-box abstractions as a hinderance in writing code and reusing external dependencies effectively and described program refinement as an alternative mechanism to build abstractions that are easily malleable simply by editing code snippets, relying on the programming system to propagate upstream changes to the modified pieces of code automatically.

As mentioned before, there is no silver bullet that can help us achieve the TIIS mission overnight, and ideas explored in this dissertation, together with future research across many domains, will collectively help reimagining the programming experience for the modern, connected, world with computers everywhere.

Bibliography

[Abelson and Sussman, 1996] Abelson, H. and Sussman, G. J. (1996). *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition.

[Afshari et al., 2012] Afshari, M., Barr, E. T., and Su, Z. (2012). Liberating the programmer with prorogued programming. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2012, pages 11–26, New York, NY, USA. ACM.

[Apple Inc., 2014] Apple Inc. (2014). Xcode. `https://developer.apple.com/xcode/`.

[Baker and Hewitt, 1977] Baker, Jr., H. C. and Hewitt, C. (1977). The incremental garbage collection of processes. In *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*, pages 55–59, New York, NY, USA. ACM.

[Bayne et al., 2011] Bayne, M., Cook, R., and Ernst, M. D. (2011). Always-available static and dynamic feedback. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 521–530, New York, NY, USA. ACM.

[Bodik et al., 2010] Bodik, R., Chandra, S., Galenson, J., Kimelman, D., Tung, N., Barman, S., and Rodarmor, C. (2010). Programming with angelic nondeterminism. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 339–352, New York, NY, USA. ACM.

[Bravenboer and Visser, 2004] Bravenboer, M. and Visser, E. (2004). Concrete syntax for objects: Domain-specific language embedding and assimilation without restrictions. In *Proceedings of* *the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 365–383, New York, NY, USA. ACM.

[Brooks, 1995] Brooks, F. (1995). *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, anniversary edition.

[Brun et al., 2010] Brun, Y., Holmes, R., Ernst, M. D., and Notkin, D. (2010). Speculative analysis: Exploring future development states of software. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, pages 59–64, New York, NY, USA. ACM.

[Chandra et al., 2011] Chandra, S., Torlak, E., Barman, S., and Bodik, R. (2011). Angelic debugging. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 121–130, New York, NY, USA. ACM.

[Chapman, 1982] Chapman, D. (1982). A program testing assistant. *Commun. ACM*, 25(9):625–634.

[Czerwinski et al., 2004] Czerwinski, M., Horvitz, E., and Wilhite, S. (2004). A diary study of task switching and interruptions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '04, pages 175–182, New York, NY, USA. ACM.

[Dahl et al., 1968] Dahl, O.-J., Myhrhaug, B., and Nygaard, K. (1968). Some features of the simula 67 language. In *Proceedings of the Second Conference on Applications of Simulations*, pages 29–31.

[de Icaza et al., 2010] de Icaza, M., Safar, M., Peterson, S., Maurer, B., Pouliot, S., Raja, H., and Baulig, M. (2010). Mono C# compiler. `http://www.mono-project.com/CSharp_Compiler`.

[Dijkstra, 1974] Dijkstra, E. W. (1974). On the role of scientific thought. Published as EWD:EWD447 `http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD447.PDF`.

[Freeman-Benson and Borning, 1992] Freeman-Benson, B. N. and Borning, A. (1992). Integrating constraints with an object-oriented language. In Madsen, O. L., editor, *ECOOP '92 – European Conference on Object-Oriented Programming*, pages 268–286, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Ghosh, 2011] Ghosh, D. (2011). DSL for the uninitiated. *Communications of the ACM*, 54(7):44–50.

[GitHub Inc., 2008] GitHub Inc. (2008). GitHub. `http://github.com`.

[Goldberg and Robson, 1983] Goldberg, A. and Robson, D. (1983). *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Google Inc., 2013] Google Inc. (2013). Android Studio. `http://developer.android.com/tools/studio`.

[Green et al., 2015] Green, T. J., Olteanu, D., and Washburn, G. (2015). Live programming in the LogicBlox system: A MetaLogiQL approach. *Proceedings of the 41st International Conference on Very Large Data Bases*, 8(12):1782–1791.

[Gu et al., 2012] Gu, Z., Barr, E. T., Schleck, D., and Su, Z. (2012). Reusing debugging knowledge via trace-based bug search. In *ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 927–942.

[Gu et al., 2014] Gu, Z., Schleck, D., Barr, E. T., and Su, Z. (2014). Capturing and exploiting IDE interactions. In *ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 83–94.

[Gulwani, 2010] Gulwani, S. (2010). Dimensions in program synthesis. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, PPDP '10, pages 13–24, New York, NY, USA. ACM.

[Gulwani, 2011] Gulwani, S. (2011). Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 317–330.

[Halbert, 1984] Halbert, D. C. (1984). *Programming by example*. PhD thesis, University of California, Berkeley.

[Harris and Gulwani, 2011] Harris, W. R. and Gulwani, S. (2011). Spreadsheet table transformations from examples. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 317–328.

[Hejlsberg et al., 2010] Hejlsberg, A., Torgersen, M., Wiltamuth, S., and Golde, P. (2010). C# language specification.

[Hewitt et al., 1973] Hewitt, C., Bishop, P., and Steiger, R. (1973). A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245.

[Iqbal and Horvitz, 2007] Iqbal, S. T. and Horvitz, E. (2007). Disruption and recovery of computing tasks: Field study, analysis, and directions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '07, pages 677–686, New York, NY, USA. ACM.

[Jiang et al., 2007] Jiang, L., Misherghi, G., Su, Z., and Glondu, S. (2007). Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 96–105, Washington, DC, USA. IEEE Computer Society.

[Jiang and Su, 2009] Jiang, L. and Su, Z. (2009). Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 81–92, New York, NY, USA. ACM.

[Kiczales, 1992] Kiczales, G. (1992). Towards a new model of abstraction in the engineering of software. In *Proceedings of the IMSA'92 Workshop on Reflection and Meta-level Architectures*.

[Kiczales et al., 1997] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In Akşit, M. and Matsuoka, S., editors, *ECOOP '97 European Conference on Object-Oriented Programming*, pages 220–242, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Kim et al., 2005] Kim, M., Sazawal, V., Notkin, D., and Murphy, G. (2005). An empirical study of code clone genealogies. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 187–196, New York, NY, USA. ACM.

[Kodowa, Inc., 2012] Kodowa, Inc. (2012). Light Table. `http://lighttable.com/`.

[Lau et al., 2003a] Lau, T., Domingos, P., and Weld, D. S. (2003a). Learning programs from traces using version space algebra. In *Proceedings of the 2Nd International Conference on Knowledge Capture*, K-CAP '03, pages 36–43, New York, NY, USA. ACM.

[Lau et al., 2003b] Lau, T., Wolfman, S. A., Domingos, P., and Weld, D. S. (2003b). Programming by demonstration using version space algebra. *Machine Learning*, 53(1):111–156.

[Lau et al., 2000] Lau, T. A., Domingos, P., and Weld, D. S. (2000). Version space algebra and its application to programming by demonstration. In *Proceedings of the Seventeenth International Conference on Machine Learning*, ICML '00, pages 527–534, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Le and Gulwani, 2014] Le, V. and Gulwani, S. (2014). FlashExtract: A framework for data extraction by examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 542–553, New York, NY, USA. ACM.

[Leavenworth, 1966] Leavenworth, B. M. (1966). Syntax macros and extended translation. *Communications of the ACM*, 9(11):790–793.

[Lemos et al., 2011] Lemos, O. A. L., Bajracharya, S., Ossher, J., Masiero, P. C., and Lopes, C. (2011). A test-driven approach to code search and its application to the reuse of auxiliary functionality. *Information and Software Technology*, 53(4):294 – 306. Special section: Software Engineering track of the 24th Annual Symposium on Applied ComputingSoftware Engineering track of the 24th Annual Symposium on Applied Computing.

[Lemos et al., 2007] Lemos, O. A. L., Bajracharya, S. K., Ossher, J., Morla, R. S., Masiero, P. C., Baldi, P., and Lopes, C. V. (2007). Codegenie: Using test-cases to search and reuse source code. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 525–526, New York, NY, USA. ACM.

[Lieberman and Hewitt, 1980] Lieberman, H. and Hewitt, C. (1980). A session with tinker: Interleaving program testing with program design. In *Proceedings of the 1980 ACM Conference on LISP and Functional Programming*, LFP '80, pages 90–99, New York, NY, USA. ACM.

[Liskov and Zilles, 1974] Liskov, B. and Zilles, S. (1974). Programming with abstract data types. In *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*, pages 50–59.

[Locke, 1689] Locke, J. (1689). *An essay concerning human understanding*.

[Microsoft, 2015] Microsoft (2015). Create custom visualizers of data. `http://msdn.microsoft.com/en-us/library/zayyhzts.aspx`.

[Microsoft Research, 2015] Microsoft Research (2015). Bing code search. `http://codesnippet.research.microsoft.com`.

[Milicevic et al., 2011] Milicevic, A., Rayside, D., Yessenov, K., and Jackson, D. (2011). Unifying execution of imperative and declarative code. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 511–520, New York, NY, USA. ACM.

[Murphy-Hill et al., 2012] Murphy-Hill, E., Jiresal, R., and Murphy, G. C. (2012). Improving software developers' fluency by recommending development environment commands. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 42:1–42:11, New York, NY, USA. ACM.

[.NET Foundation, 2015] .NET Foundation (2015). .NET Compiler Platform (Roslyn). `https://github.com/dotnet/roslyn`.

[Pérez and Granger, 2007] Pérez, F. and Granger, B. E. (2007). IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29.

[Reiss, 2009] Reiss, S. P. (2009). Semantics-based code search. In *2009 IEEE 31st International Conference on Software Engineering*, pages 243–253.

[Sadowski et al., 2015] Sadowski, C., Stolee, K. T., and Elbaum, S. (2015). How developers search for code: A case study. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 191–201, New York, NY, USA. ACM.

[Samimi et al., 2010] Samimi, H., Aung, E. D., and Millstein, T. (2010). Falling back on executable specifications. In D'Hondt, T., editor, *ECOOP 2010 – European Conference on Object-Oriented Programming*, pages 552–576, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Schachman, 2012] Schachman, T. (2012). Alternative programming interfaces for alternative programmers. In *ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 1–10.

[Smith and Ungar, 1995] Smith, R. B. and Ungar, D. (1995). Programming as an experience: The inspiration for Self. In Tokoro, M. and Pareschi, R., editors, *ECOOP '95 – European Conference on Object-Oriented Programming*, pages 303–330. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Solar-Lezama et al., 2007] Solar-Lezama, A., Arnold, G., Tancau, L., Bodik, R., Saraswat, V., and Seshia, S. (2007). Sketching stencils. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 167–178.

[Solar-Lezama et al., 2008] Solar-Lezama, A., Jones, C. G., and Bodik, R. (2008). Sketching concurrent data structures. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 136–148.

[Solar-Lezama et al., 2005] Solar-Lezama, A., Rabbah, R., Bodík, R., and Ebcioğlu, K. (2005). Programming by sketching for bit-streaming programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 281–294.

[Solar-Lezama et al., 2006] Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., and Saraswat, V. (2006). Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 404–415.

[Stack Overflow Inc., 2008] Stack Overflow Inc. (2008). Stack Overflow. `http://stackoverflow.com`.

[The Clang Team, 2007] The Clang Team (2007). Clang tooling. `http://clang.llvm.org/docs/Tooling.html`.

[The Go Authors, 2009] The Go Authors (2009). Go programming language. `https://golang.org/`.

[The MathWorks, Inc., 1984] The MathWorks, Inc. (1984). MATLAB - The Language of Technical Computing. `http://www.mathworks.com/products/matlab/`.

[Toomim et al., 2004] Toomim, M., Begel, A., and Graham, S. L. (2004). Managing duplicated code with linked editing. In *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*, pages 173–180.

[Torvalds, 2005] Torvalds, L. (2005). Git. `https://github.com/git/git`.

[Two Lives Left, 2014] Two Lives Left (2014). Codea. `http://twolivesleft.com/Codea/`.

[van Deursen et al., 2000] van Deursen, A., Klint, P., and Visser, J. (2000). Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36.

[Victor, 2012] Victor, B. (2012). Inventing on principle. `http://vimeo.com/36579366`.

[von Ahn et al., 2003] von Ahn, L., Blum, M., Hopper, N. J., and Langford, J. (2003). Advances in cryptology — EUROCRYPT 2003: International conference on the theory and applications of cryptographic techniques, warsaw, poland, may 4–8, 2003 proceedings. pages 294–311, Berlin, Heidelberg. Springer Berlin Heidelberg.

[von Ahn and Dabbish, 2004] von Ahn, L. and Dabbish, L. (2004). Labeling images with a computer game. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '04, pages 319–326, New York, NY, USA. ACM.

[von Ahn et al., 2008] von Ahn, L., Maurer, B., McMillen, C., Abraham, D., and Blum, M. (2008). reCAPTCHA: Human-based character recognition via web security measures. *Science*, 321(5895):1465–1468.

[Wirth, 1971] Wirth, N. (1971). Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227.

[Witten and Mo, 1993] Witten, I. H. and Mo, D. (1993). *Watch What I Do*, chapter TELS: Learning Text Editing Tasks from Examples, pages 183–203. MIT Press, Cambridge, MA, USA.