Liberating the Programmer with Prorogued Programming

Mehrdad Afshari Earl T. Barr Zhendong Su

Department of Computer Science, University of California, Davis
{mafshari,etbarr,su}@ucdavis.edu

Abstract

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; 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.

Categories and Subject Descriptors D.2.3 [*Software Engineering*]: Coding Tools and Techniques; D.2.5 [*Software Engineering*]: Testing and Debugging; D.2.6 [*Software Engineering*]: Programming Environments; D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Design, Languages, Experimentation, Human Factors

[1] A *concern* is any piece of interest or focus in a program [8].

Keywords prorogued programming, executable refinement, hybrid computation, human computation, managing concerns, workflow improvement

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 [3]. 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 [6, 15]. 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.

1.1 Prorogued Programming

We propose a new programming paradigm, *prorogued programming*, to lift these restrictions: it allows programmers to

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

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 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 [31–33]. Section 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 (Section 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 [34], 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 [11, 12, 18–20, 35] and test-based code search tools [16, 21, 22, 25].

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.

1.2 Main Contributions

This paper 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 paper is organized as follows. We first use an example to motivate prorogued programming and illustrate its use (Section 2). Next, we formalize prorogued programming for a small functional language (Section 3). Section 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 5). Section 6 discusses a few open issues and opportunities for prorogued programming. Finally, Section 7 surveys related work, and Section 8 concludes.

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 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 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.

```
 1 static void Main() {
 2   string input =
 3     prorogue ReadFile("mail.txt");
 4   PrintEmail(input);
 5 }
 6 static void PrintEmail(string input) {
 7   string from =
 8     prorogue GetHeader(input, "From");
 9   string subject =
10     prorogue GetHeader(input, "Subject");
11   string body =
12     prorogue GetBody(input);
13   Console.WriteLine("From: " + from);
14   Console.WriteLine("Subject: " + subject);
15   Console.WriteLine(body);
16 }
```

Figure 1: The simple mail parser in Prorogued C#.

```
 1 static void Main() {
 2   var db = prorogue
 3     new UserDatabase { Server = "server1" };
 4   db.ConnectionTimeout = 1000;
 5   var userName = Console.ReadLine();
 6   var password = Console.ReadLine();
 7   if (db.Authenticate(userName, password)) {
 8     string input =
 9       prorogue ReadFile("mail.txt");
10     PrintEmail(input);
11   } else Console.WriteLine
12                   ("Authentication failed.");
13 }
```

Figure 2: Mail program using a prorogued mock database.

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 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. 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 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 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.

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

Figure 3: The syntax of the simple prorogued language $\mathscr{F}_\mathrm{p}$, which adds, to a standard expression language, the new syntactic construct "**prorogue** $f(e)$."

The pairs collected by running a prorogued program can also be used to generate unit tests automatically (Section 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.

3. A Prorogued Programming Language

This section formalizes the syntax and semantics of a small prorogued programming language $\mathscr{F}_\mathrm{p}$ to clarify our presentation. Our actual implementation (Section 4) is an extension to C#.

3.1 Syntax and Semantics of $\mathscr{F}_\mathrm{p}$

$\mathscr{F}_\mathrm{p}$ extends a standard core expression language; Figure 3 shows its syntax and Figure 4, its semantics. An $\mathscr{F}_\mathrm{p}$ program consists of a list of functions, each of which has a single integer 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 $[\![\mathrm{op}]\!]$, *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 4 give the dynamic, big-step semantics of $\mathscr{F}_\mathrm{p}$. The value domain is $\textit{Value} = \mathbb{Z} \cup \{\perp\} = \mathbb{Z}_\perp$. 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 $\Sigma \ni \sigma : \textit{Var} \rightarrow \textit{Value}$ maps variables to values;
- the IO store $\kappa : \mathbb{F} \times \textit{Value} \rightarrow \textit{Value}$ 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 $\sigma[v/x]$

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

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

(a) Semantics of const and var.

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

(b) Semantics of op.

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

(c) Semantics of if-false.

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

(d) Semantics of if-true.

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

(e) Semantics of let.

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

(f) Semantics of call.

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

(g) Semantics of p-call-old.

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

(h) Semantics of p-call-new.

Figure 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 $(f', v') \neq (f, v)$.

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 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 $\mathscr{F}_\text{p}$ 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 3.1 (Correctness of Prorogued Semantics)**.** *For any program p, state σ, and expression e,*

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

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 [34]. 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 [16, 21, 22, 25] to find reusable implementations; and 4) employ synthesis by example techniques [11, 12, 18–20, 35] 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 3.2 (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 $\kappa(f,i) \neq \perp$.*

4. Design and Realization of Prorogued C#

To experiment with prorogued programming, we extended the Mono C# compiler [7], 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.

4.1 The Language

To realize the prorogued programming paradigm in C#, we amended the C# grammar [13] 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).

```
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 5: 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.

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).

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 lazy evaluation, discussed below in Section 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

```
T₀ transmute(T₁ x, T₂ y) {
  if (x < 0)
    return 0;
  else
    return prorogue transmute(x, y);
}
```

Figure 6: 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.

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);
```

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 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 Section 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 6 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 6. 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 (Section 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 $[0..10] \rightarrow 5$. 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 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

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

Figure 7: Extending existing functions with **prorogue**.

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.

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

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

Figure 8: Lazy evaluation of prorogued return values.

```
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 9: Capturing the context of a call.

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.

4.5 Lazy Evaluation of Prorogued Calls

An execution of a prorogued program that prompts the programmer to populate IO stores too frequently 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 [14] 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 8, 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 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.

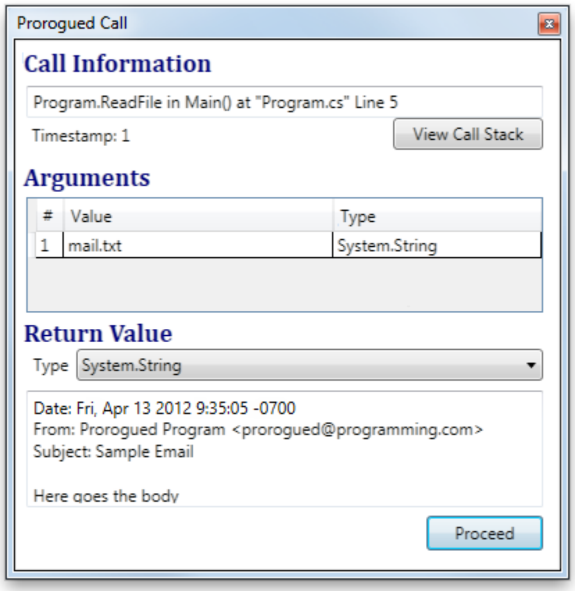
Laziness brings up another potential issue: the order of user interactions may not correspond to the order in which prorogued calls were visited during execution. In Figure 9, the call to `CalculateRebate` is prorogued, but the dispatcher does not prompt the user until execution reaches line 5. If `price` 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.

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



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

Timestamp: 1

#	Value	Type
1	mail.txt	System.String

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 10: Prorogued C#'s user interface.

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)`.

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 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 (Section 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 11 shows a simple reified method.

```
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 11: Generated code for a 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.

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.

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

Task Assignment Program construction is difficult to parallelize [3]. 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

[3] This syntax could have been used in Figure 6.

```
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 12: Deploying an incomplete program in a non-prorogued language by hardcoding values.

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 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 [3]. 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,

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

Figure 13: Prorogued localization.

many software systems, especially custom software systems that are developed 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 12. Had prorogued programming been available, the code could have been written as in Figure 13. 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 [5], 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

```
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 14: Spam filtering with prorogued programming.

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.

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) 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 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,

[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 15: Evolving an existing API.

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 15.

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 16 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, 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.

```
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 16: Debugging with prorogued invocations.

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 [17]. In Section 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 (Section 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 Section 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 [10]. 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?

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) [17]. 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 [23]. 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 [1]. 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 non-deterministic `choose` operator to assist program construction [2]. 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 [4].

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 [9, 24, 26]. When it encounters a specification, a mixed interpreter runs a solver and updates the heap with the solution if one is found[5]. Writing specifications can be quite difficult, so mixed interpreters can impose precisely the disruptive abstraction shift that prorogue aims to obvi-

[5] Samimi *et al.*'s Plan B "encounters" a specification when an implementation violates its contract.

ate. 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 [27–30]. Prorogued calls can be seen as these holes. Example-guided synthesis, as its name suggests, uses examples to guide synthesis [11, 12]. 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 3, is programming by example (aka programming by demonstration) [18–20, 35]; this work too provides an opportunity for synergism with prorogued programming.

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.

Acknowledgements

We thank Phillip Rogaway for helping name the "prorogued" programming paradigm, and Alex Aiken, Mark Gabel, Anders Møller, and Martin Rinard for helpful discussions and feedback on this work.

This research was supported in part by NSF CAREER Grant No. 0546844, NSF CyberTrust Grant No. 0627749, NSF CCF Grant No. 0702622, NSF TC Grant No. 0917392, NSF SHF Grant No. 1117603, and the US Air Force under grant FA9550-07-1-0532. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

References

[1] M. Bayne, R. Cook, and M. D. Ernst. Always-available static and dynamic feedback. In *ICSE*, 2011.

[2] R. Bodik, S. Chandra, J. Galenson, D. Kimelman, N. Tung, S. Barman, and C. Rodarmor. Programming with angelic nondeterminism. In *POPL*, 2010.

[3] F. P. Brooks. *The mythical man-month — essays on software engineering (2nd ed.)*. Addison-Wesley, 1995.

[4] S. Chandra, E. Torlak, S. Barman, and R. Bodik. Angelic debugging. In *ICSE*, 2011.

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

[6] M. Czerwinski, E. Horvitz, and S. Wilhite. A diary study of task switching and interruptions. In *CHI*, pages 175–182, 2004.

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

[8] E. W. Dijkstra. On the role of scientific thought. Published as EWD:EWD447, Aug. 1974.

[9] B. N. Freeman-Benson and A. Borning. Integrating constraints with an object-oriented language. In *ECOOP*, pages 268–286, 1992.

[10] S. Gulwani. Dimensions in program synthesis. In *FMCAD*, 2010.

[11] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, 2011.

[12] W. R. Harris and S. Gulwani. Spreadsheet table transformations from examples. In *POPL*, 2011.

[13] A. Hejlsberg, M. Torgersen, S. Wiltamuth, and P. Golde. C# language specification, 2010.

[14] J. Henry C. Baker and C. Hewitt. The incremental garbage collection of processes. In *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*, pages 55–59, 1977.

[15] S. T. Iqbal and E. Horvitz. Disruption and recovery of computing tasks: field study, analysis, and directions. In *CHI*, pages 677–686, 2007.

[16] L. Jiang and Z. Su. Automatic mining of functionally equivalent code fragments via random testing. In *ISSTA*, pages 81–92, 2009.

[17] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

[18] T. A. Lau, P. Domingos, and D. S. Weld. Version space algebra and its application to programming by demonstration. In *ICML*, pages 527–534, 2000.

[19] T. A. Lau, P. Domingos, and D. S. Weld. Learning programs from traces using version space algebra. In *K-CAP*, pages 36–43, 2003.

[20] T. A. Lau, S. A. Wolfman, P. Domingos, and D. S. Weld. Programming by demonstration using version space algebra. *Machine Learning*, 53(1-2):111–156, 2003.

[21] O. A. L. Lemos, S. K. Bajracharya, J. Ossher, P. C. Masiero, and C. V. Lopes. A test-driven approach to code search and its

application to the reuse of auxiliary functionality. *Information & Software Technology*, 53(4):294–306, 2011.

[22] O. A. L. Lemos, S. K. Bajracharya, J. Ossher, R. S. Morla, P. C. Masiero, P. Baldi, and C. V. Lopes. CodeGenie: using test-cases to search and reuse source code. In *ASE*, 2007.

[23] H. Lieberman and C. Hewitt. 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, 1980. ACM.

[24] A. Milicevic, D. Rayside, K. Yessenov, and D. Jackson. Unifying execution of imperative and declarative code. In *ICSE*, pages 511–520, 2011.

[25] S. P. Reiss. Semantics-based code search. In *ICSE*, pages 243–253, 2009.

[26] H. Samimi, E. D. Aung, and T. D. Millstein. Falling back on executable specifications. In *ECOOP*, pages 552–576, 2010.

[27] A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodik, V. Saraswat, and S. Seshia. Sketching stencils. In *PLDI*, 2007.

[28] A. Solar-Lezama, C. G. Jones, and R. Bodik. Sketching concurrent data structures. In *PLDI*, 2008.

[29] A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioğlu. Programming by sketching for bit-streaming programs. In *PLDI*, 2005.

[30] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, 2006.

[31] L. von Ahn, M. Blum, N. J. Hopper, and J. Langford. CAPTCHA: Using hard AI problems for security. In *EUROCRYPT*, pages 294–311, 2003.

[32] L. von Ahn and L. Dabbish. Labeling images with a computer game. In *CHI*, 2004.

[33] L. von Ahn, B. Maurer, C. McMillen, D. Abraham, and M. Blum. reCAPTCHA: Human-based character recognition via web security measures. *Science*, Sept. 2008.

[34] N. Wirth. Program development by stepwise refinement. *CACM*, 1971.

[35] I. H. Witten and D. Mo. *TELS: Learning text editing tasks from examples*, pages 183–203. MIT Press, Cambridge, MA, USA, 1993.