Toward a Formal Theory of Extensible Software*

Shriram Krishnamurthi   Matthias Felleisen
Department of Computer Science
Rice University
Houston, TX 77005–1892, USA
shriram@cs.rice.edu

Abstract

As software projects continue to grow in scale and scope, it becomes important to reuse software. An important kind of reuse is *extensibility*, i.e., the extension of software without accessing existing code to edit or copy it. In this paper, we propose a rigorous, semantics-based definition of software extensibility. Then we illustrate the utility of our definitions by applying them to several programs. The examination shows how programming style affects extensibility and also drives the creation of a variant of an existing design pattern. We consider programs in both object-oriented and functional languages to prove the robustness of our definitions.

1 Introduction

As software projects have continued to grow in scale and scope, it has become increasingly important to reuse program components. Reuse lowers software development costs by reducing development time, decreasing the number of errors, and increasing the consistency of software systems. In short, there are compelling reasons to study and understand software reuse.

Researchers have recognized the importance of software reuse and have made it the subject of numerous studies [13, 19, 20, 23, 28, 29]. Many of the studies cited in these surveys describe metrics for reuse. Others examine actual software artifacts and estimate the extent of reuse in those systems. Most of this research, however, suffers from two important shortcomings.

1. There are no rigorous definitions of reuse, which makes reusable software difficult to identify and classify, especially since methods such as "copy-and-paste" or "scavenging" are considered reuse techniques by some authors [19]. This also makes it impossible to express the reusability properties of programs and components in a precise manner.
2. Most of the methods of analyzing software reuse are *syntactic*. For example, reuse is often measured in terms of lines-of-code [28], which is a coarse measure that can vary widely across languages. Other syntactic measures suffer from similar problems.

In this paper, we focus on a specific kind of reuse called *extensibility*.[1] An extensible program can be adapted to new tasks without accessing its source code. In particular, our definition prevents two acts. The first is source modification, which can introduce unexpected behavioral and structural changes. The second is copying of code, which increases the clerical effort needed to maintain programs by introducing potential inconsistencies. Extensibility is particularly critical for a producer who wishes to market programs that clients can customize, but who does not want to reveal proprietary source code.

Our work provides a *semantic* definition for extensibility. Hence, we can state, prove and compare the extensible properties of programs on a rigorous basis. It also helps us focus on the behavior of the program, freeing us from both ambiguous and language-sensitive syntactic characteristics. With these formal definitions, we can even compare the extensible properties of programs across (semantically related) languages.

By using a rigorous characterization, software engineers can determine whether a given program is extensible in certain ways. Documenting programs with corresponding statements will be invaluable to the program design process since it enables design for anticipated classes of extensions. More significantly, it can be used to create a catalogue of extensible software that can simplify the process of designing large and complex systems.

Design patterns [15] are a step in this direction, but patterns and their extensible properties are usually only stated

*This research was partially supported by NSF grants CCR-9619756 and CCR-9708957, and by a Texas ATP grant.

To appear in the Sixth ACM-SIGSOFT *Symposium on the Foundations of Software Engineering*, November, 1998.

[1] This is colloquially called "black-box reuse" [19].

```
abstract class Subject {
  String name;
  String getName () {
    return name; } }

class Prince extends Subject {
  Prince (String name) {
    this.name = name; } }

class Princess extends Subject {
  Princess (String name) {
    this.name = name; } }

class Frog extends Subject {
  Frog (String name) {
    this.name = name; } }
```

Figure 1: $EditObj_1$

```
class Wizard extends Subject {
  Wizard (String name) {
    this.name = name; } }
```

Figure 2: $EditObj_2$ Extension

```
abstract class Subject {
  String name;
  String getName () {
    return name; }
  Subect kiss () {
    return this; } }

class Prince_k extends Subject {
  Prince_k (String name) {
    this.name = name; } }

class Princess_k extends Subject {
  Princess_k (String name) {
    this.name = name; } }

class Frog_k extends Subject {
  Frog_k (String name) {
    this.name = name; }
  Subject kiss () {
    return new Prince_k (name); } }
```

Figure 3: $EditObj_3$

informally. As we illustrate, these informal statements can be misleading. In addition, patterns are currently defined in terms of object-oriented designs, which narrows their applicability.

We first present our definitions in the context of object-oriented languages. Subsequently, we extend the definitions to apply to functional languages as well. This extension requires only one change, concerning the one definition that directly refers to the syntax of the underlying language. The remaining definitions carry over unchanged. This switch illustrates that our definitions are robust, i.e., they depend only minimally on the specifics of the language.

The rest of this paper is organized as follows. Section 2 illustrates the principle of extensibility through two series of programs produced by different design principles. Section 3 introduces our formal definition of extensibility. Section 4 applies the definitions from Section 3 to the examples in Section 2 and uses the results to define a new, more extensible version of one of the programs and a corresponding new design pattern. Section 5 shows the robustness of the definitions by extending them to functional languages. Section 6 discusses directions for future work. The last two sections describe related work and summarize the ideas in this paper.

2 A Motivating Example

We examine two contrasting approaches to software development through two sequences of programs: *EditObj* and *Int*. They implement two different representations of characters and actions in a fantasy game.[2] *EditObj* represents a "copy-and-paste" approach to program construction. *Int* uses the Interpreter pattern [15], which is claimed to permit programmers to add functionality without altering or duplicating existing code.

Each element in the two sequences is a collection of class definitions, which we call a *repertoire*. Both sequences begin with a common initial repertoire, shown in Figures 1 and 4. The repertoire represents all characters as instances of concrete subclasses of Subject, which contains one method: *getName*.

The game is a moderate market success, so we decide to extend it with new types of characters. Specifically, we add wizards, a new class of subjects in our imaginary land. Both sequences are adapted in the same manner: by adding

[2] The programs are written in Java [16] because it is a widely-understood object-oriented language with simple formal models [6, 12], but the results apply equally to other, semantically-related languages.

```
abstract class Subject {
  String name;
  String getName () {
    return name; } }

class Prince extends Subject {
  Prince (String name) {
    this.name = name; } }

class Princess extends Subject {
  Princess (String name) {
    this.name = name; } }

class Frog extends Subject {
  Frog (String name) {
    this.name = name; } }
```

Figure 4: Int_1

```
class Wizard extends Subject {
  Wizard (String name) {
    this.name = name; } }
```

Figure 5: Int_2 Extension

```
class Prince_k extends Prince {
  Prince_k (String s) {
    super (s); }
  Subject kiss () {
    return this; } }

class Princess_k extends Princess {
  Princess_k (String s) {
    super (s); }
  Subject kiss () {
    return this; } }

class Frog_k extends Frog {
  Frog_k (String s) {
    super (s); }
  Subject kiss () {
    return new Prince_k (name); } }
```

Figure 6: Int_3 Extension

a new concrete subclass, Wizard, of Subject, and endowing it with the appropriate methods. Figures 2 and 5 illustrate this extension.

This version is even more successful. To sustain player interest, we introduce the first behavioral change into the game: each subject must respond to being kissed. Most subjects ignore this advance, except frogs, who turn into princes. This change illustrates how the two sequences differ. For the first, *EditObj*, the resulting repertoire is shown in Figure 3.[3] (The subscript 'k' indicates that the class has a *kiss* method.) For the second, *Int*, we use subclassing to add the new method, as shown in Figure 6.

Based on the informal discussion in Section 1, we consider the elements of *Int* constructed through extension, which the elements of *EditObj* are not. We formalize this intuition in the next section.

[3] The class names in $EditObj_3$ mirror those in Int_3 so that they can support the same client expressions. This still forces changes in clients so that they can create instances of the right classes. This problem can be eliminated through the use of the Abstract Factory pattern [15] or a module system.

3 Extensibility in Object-Oriented Languages

In this section, we define extensibility in terms of a simple model of sequential object-oriented languages.

To define extensibility, we must first agree on (the abstract syntax of) a minimal syntactic core language. Programs consist of a tree of classes and a directed acyclic graph of interfaces. A class is defined incrementally through a sequence of *class extensions*. A class extension describes a collection of fields and methods relative to some superclass. The complete class is the aggregate of all these extensions, starting from some universal base class (called Object in Java). Similarly, interfaces are specified incrementally through *interface extensions*, starting with an empty interface. We assume the language has the standard collection of expressions (including, for example, conditionals and assignment).

Definition 1 (Definition) *Each well-formed class or interface extension is a* definition.

Definition 2 (Repertoire) *A* repertoire *is a well-formed set of definitions.*

Definition 3 (Program) *A* program *consists of (1) a repertoire r and (2) an expression e (notated $r \cdot e$) such that the resulting combination is closed (i.e., has no free variables).*

Given a semantics for the programming language [6, 12], we can express when two programs have equivalent observable behavior in terms of termination.

Definition 4 (Equivalence)

- *For the programs p_1 and p_2, p_1 is* functionally equivalent *to p_2 (notated $p_1 \cong p_2$) if p_1 halts without error iff p_2 halts without error.*
- *For repertoires r_1 and r_2, r_1 is* functionally equivalent *to r_2 (notated $r_1 \cong r_2$) if for all expressions e such that $r_1 \cdot e$ and $r_2 \cdot e$ are programs, $r_1 \cdot e \cong r_2 \cdot e$.*

Next we define when one program conservatively extends the behavior of another.

Definition 5 (Containment) *For the repertoires r_1 and r_2, r_2* contains *r_1 (notated $r_1 \subseteq r_2$) if for each definition d in r_1, d is in r_2.*

Definition 6 (Approximation)

- *The program p_1* approximates *the program p_2 (notated $p_1 \sqsubseteq p_2$) if p_2 halts without error whenever p_1 halts without error.*
- *For repertoires r_1 and r_2, r_1* approximates *r_2 (notated $r_1 \sqsubseteq r_2$) if for all expressions e such that $r_1 \cdot e$ and $r_2 \cdot e$ are programs, $r_1 \cdot e \sqsubseteq r_2 \cdot e$.*

Definition 7 (Behavioral Extension) *For repertoires r_1 and r_2, r_2* behaviorally extends *r_1 (notated $r_1 \lhd r_2$) if $r_1 \subseteq r_2$ and $r_1 \sqsubseteq r_2$.*

This definition requires the extension to mimic the behavior of the extended program on inputs common to both; thus it allows only *conservative* extensions.[4]

It suffices to use termination as a test for equality because equivalence and approximation are defined in terms of *all* expressions that are closed with respect to the repertoire. This includes expressions of the form

$$\begin{array}{l} \textbf{if}\ (\langle e \text{ has the desired value}\rangle) \\ \quad \langle \text{halt} \rangle \\ \textbf{else} \\ \quad \langle \text{loop forever} \rangle \end{array}$$

This expression reports, via termination, whether the expression e reduces to the desired value. If e does not terminate, it will not test equal to any value through this process.

With these definitions, we can define our key notion: relative extensibility.

[4] When a program is extended in a non-conservative fashion, it is impossible to determine how these changes in behavior will affect clients. Therefore, our definitions do not accommodate such extensions.

Definition 8 (Relative Extensibility) *For repertoires x, r and r', x is* an extensible version of r with respect to r' *if $r \cong x$ and there exists a repertoire x' such that $x \lhd x'$ and $x' \cong r'$.*

Relative extensibility is a property of one repertoire x with respect to two others, r and r'. Suppose x and r have the same observable behavior. (They may even be the same program.) There are many possible programs r' that can be produced starting from r. *The definition places no constraints on the relationship between r and r'*; r' may have been obtained through extension, by manual editing (like the *EditObj* sequence of Section 2), or even by replacing r entirely. For some of these r', we can produce x' as a behavioral extension of x such that x' and r' have the same behavior. That is, x' is x with some additional definitions. Therefore we call it an extensible version of r with respect to r'.

This definition has an important implication: extensibility is always defined *with respect to specific properties*. In our definition, the property is represented by the "difference" between r and r'. We believe it is meaningless to speak about extensibility without mentioning the properties in question.

4 Success and Failure of Patterns

To illustrate the utility of our definitions, we apply them to the examples of Section 2. This reveals the strengths and weaknesses of the Interpreter pattern in providing extensibility.

4.1 The *EditObj* and *Int* Sequences

Proposition 1 *$Int_1 \lhd Int_2 \lhd Int_3$.*

Proof Sketch By construction, $Int_1 \subseteq Int_2 \subseteq Int_3$. None of the repertoire extensions overrides a method, or contains a method that is invoked, in the repertoire that it extends. Therefore, the observable behavior of existing programs is not affected by any of the extensions. □

Proposition 1 shows that the *Int* sequence conforms to our definition of extensibility. Next we show that the first three repertoires in *Int* are equivalent to their *EditObj* counterparts.

Proposition 2 *$EditObj_1 \cong Int_1$ and $EditObj_2 \cong Int_2$.*

Proof Sketch This follows trivially since the repertoires are syntactically identical. □

Proposition 3 *$EditObj_3 \cong Int_3$.*

Proof Sketch Any expression that forms a program with both $EditObj_3$ and Int_3 can create instances of only the $\mathsf{Prince_k}$, $\mathsf{Princess_k}$ or $\mathsf{Frog_k}$ classes. The *kiss* method in $\mathsf{Frog_k}$ also creates instances of $\mathsf{Prince_k}$. Thus, the objects created by each program have the same methods with the

```
abstract class Subject {
  String name;
  String getName () {
    return name; }
  abstract Subject kiss ();
  abstract Subject spellCast (); }

class Prince_k,s extends Subject {
  Prince_k,s (String name) {
    this.name = name; }
  Subject kiss () {
    return this; }
  Subject spellCast () {
    return this; } }

class Princess_k,s extends Subject {
  Princess_k,s (String name) {
    this.name = name; }
  Subject kiss () {
    return this; }
  Subject spellCast () {
    return this; } }

class Frog_k,s extends Subject {
  Frog_k,s (String name) {
    this.name = name; }
  Subject kiss () {
    return new Prince_k,s (name); }
  Subject spellCast () {
    return this.kiss (); } }
```

Figure 7: *EditObj*$_4$

```
class Prince_k,s extends Prince_k {
  Prince_k,s (String s) {
    super (s); }
  Subject spellCast () {
    return this; } }

class Princess_k,s extends Princess_k {
  Princess_k,s (String s) {
    super (s); }
  Subject spellCast () {
    return this; } }

class Frog_k,s extends Frog_k {
  Frog_k,s (String s) {
    super (s); }
  Subject spellCast () {
    return this.kiss (); } }
```

Figure 8: *Int*$_4$ Extension

same implementations. Therefore they have the same observable behavior. □

To study the extensibility characteristics of the Interpreter pattern in more depth, we add one more action to our game: characters can now cast spells on one another. The details of spells are unimportant for our exposition; hence, the *spellCast* method always returns **this** except for Frogs, which behave as if they have been kissed. As before, in *EditObj*$_4$ we add the *spellCast* method directly to the existing source (Figure 7), while in *Int*$_4$ we add it through class extension (Figure 8). (The 's' subscript indicates the presence of the *spellCast* method.)

Now it is not true that *EditObj*$_4 \cong$ *Int*$_4$. The *kiss* method in *Int*$_4$ is inherited from *Int*$_3$, and any object created by that method will only create instances of $\mathsf{Prince}_\mathsf{k}$, not $\mathsf{Prince}_{\mathsf{k,s}}$. Hence, the two repertoires are not equivalent.

Proposition 4 *EditObj*$_4 \ncong$ *Int*$_4$.

Proof Sketch It suffices to present an expression e such that *EditObj*$_4$ terminates without an error while the program *Int*$_4 \cdot e$ raises an error. Consider

$$(\textbf{new}\ \mathsf{Frog}_{\mathsf{k,s}}\ (\texttt{"Kermit"})).\mathit{kiss}\ ()\ .$$

EditObj$_4$ creates an instance of $\mathsf{Prince}_{\mathsf{k,s}}$ that contains the *spellCast* method. In contrast, *Int*$_4$ invokes the *kiss* method inherited from Frog_k. This method creates an instance of $\mathsf{Prince}_\mathsf{k}$ that does not contain a *spellCast* method. Therefore, the expression

$$((\textbf{new}\ \mathsf{Frog}_{\mathsf{k,s}}\ (\texttt{"Kermit"})).\mathit{kiss}()).\mathit{spellCast}(\ldots)$$

results in an error in repertoire *Int*$_4$, but not in *EditObj*$_4$. □

Our propositions show that *Int*$_1$ is an extensible version of *EditObj*$_1$ with respect to each of *EditObj*$_2$ and *EditObj*$_3$. However, *Int*$_3$ is not an extensible version of *EditObj*$_3$ with respect to *EditObj*$_4$. The proof's counter-example shows that any extension of *Int*$_3$ that is equivalent to *EditObj*$_4$ must ensure that the *kiss* method creates instances of the most recent subclass of $\mathsf{Prince}_\mathsf{k}$. The next section describes a combination of patterns that increases the reuse potential of program components based on the Interpreter pattern.

```
class Prince_k extends Prince {
  Prince_k (String s) {
    super (s); }
  Subject kiss () {
    return this; } }

class Princess_k extends Princess {
  Princess_k (String s) {
    super (s); }
  Subject kiss () {
    return this; } }
```

```
class Frog_k extends Frog {
  Frog_k (String s) {
    super (s); }
  Subject makePrince (String s) {
    return new Prince_k (s); }
  Subject kiss () {
    return makePrince (name); } }
```

Figure 9: *ExtInt*$_3$ Extension

```
class Prince_k,s extends Prince_k {
  Prince_k,s (String s) {
    super (s); }
  Subject spellCast () {
    return this; } }

class Princess_k,s extends Princess_k {
  Princess_k,s (String s) {
    super (s); }
  Subject spellCast () {
    return this; } }
```

```
class Frog_k,s extends Frog_k {
  Frog_k,s (String s) {
    super (s); }
  Subject makePrince (String s) {
    return new Prince_k,s (s); }
  Subject spellCast () {
    return this.kiss (); } }
```

Figure 10: *ExtInt*$_4$ Extension

4.2 A More Extensible Interpreter Pattern

Instead of hard-coding the object instantiation in the *kiss* method of Frog_k, we should decouple object creation from other processing done by the method. We can encode this idea by introducing a *virtual constructor* (sometimes called a Factory Method [15]) to perform the creation, and overriding the constructor to reflect extensions to $\mathsf{Prince}_\mathsf{k}$. A new series of repertoires, *ExtInt*, illustrates this pattern. The first two repertoires in the series are the same as those in *EditObj* and *Int*. In the third repertoire, we add the *kiss* method (Figure 9) and introduce the virtual constructor. The fourth implements *spellCast* and overrides the virtual constructor (Figure 10).

Proposition 5 *$EditObj_i \cong ExtInt_i$ for $i \in 1, 2, 3, 4$.*

Proof Sketch The proof exploits the fact that the virtual constructor is always overridden to create instances of the most recent extension. Other than this, it resembles the proof for the equivalences of *EditObj* and *Int*. □

From these results, we can conclude the following.

Corollary 1

- *$ExtInt_1$ is an extensible version of $EditObj_1$ with respect to $EditObj_2$, $EditObj_3$ and $EditObj_4$.*
- *$ExtInt_3$ is an extensible version of $EditObj_3$ with respect to $EditObj_4$.*

Our attempt to prove the relative extensibility of the *Int* sequence thus yields three results. First, it proves that some members of *Int* are indeed extensible (with respect to the *EditObj* sequence), validating the intuition behind the Interpreter pattern [15, pages 246–247]. Second, it identifies where the extensibility of the Interpreter pattern fails. Finally, it corrects this failure and suggests a new composite design pattern.

5 Extensibility in Functional Languages

In a typical functional programming language such as ML, Haskell or Scheme, a program consists of a sequence of function and data definitions followed by an expression over the definitions which initiates evaluation. This structure differs only slightly from our definitions in Section 3, when a definition was either a class or interface extension. Now we also allow function and data (or type) definitions.

Definition 9 (Definition) *Each function and data or type description is a* definition.

The remainder of our definitions of extensibility can stay unchanged since they are effectively independent of the constructs in the ambient programming language.

```
;; Exp = ∅

;; interp-1 : Exp ⟶ Num
(define (interp-1 expr)
  (error 'interp "no semantics for ~s" expr))
```

Figure 11: *EditFun*$_1$

```
;; Exp = num (val) | plus (lhs rhs)

(define-struct num (val))
(define-struct plus (lhs rhs))

;; interp-2 : Exp ⟶ Num
(define (interp-2 expr)
  (cond
    ((num? expr) (num-val expr))
    ((plus? expr) (+ (interp-2 (plus-lhs expr))
                     (interp-2 (plus-rhs expr))))
    (else (error 'interp "no semantics for ~s" expr))))
```

Figure 12: *EditFun*$_2$

```
;; Exp = num (val) | plus (lhs rhs) | minus (lhs rhs)

(define-struct num (val))
(define-struct plus (lhs rhs))
(define-struct minus (lhs rhs))

;; interp-3 : Exp ⟶ Num
(define (interp-3 expr)
  (cond
    ((num? expr) (num-val expr))
    ((plus? expr) (+ (interp-3 (plus-lhs expr))
                     (interp-3 (plus-rhs expr))))
    ((minus? expr) (− (interp-3 (minus-lhs expr))
                      (interp-3 (minus-rhs expr))))
    (else (error 'interp "no semantics for ~s" expr))))
```

Figure 13: *EditFun*$_3$

Let us illustrate the meaning of the revised definitions with a series of interpreters for an arithmetic language. Each member of the series adds new terms to the language and defines their meaning. Our examples are written in Scheme [1], extended with a mechanism called **define-struct** for defining structures.

The first sequence is called *EditFun*. These interpreters are built in the same spirit as the *EditObj* sequence of Section 2. The language of *EditFun*$_1$ (Figure 11), the first interpreter, is empty. Therefore, it raises an error for all inputs. The second interpreter, *EditFun*$_2$, shown in Figure 12, understands numbers and an addition operation. Finally, *EditFun*$_3$ (Figure 13) also processes subtraction.

The *EditFun* sequence does not reuse code, even though much of it is repeated from one stage to the next. To remedy this we create a second sequence, *MidFun*, which reuses existing interpreters and only adds the implementation of new operations. The first interpreter (which is identical to *EditFun*$_1$) is shown in Figure 14. Figures 15 and 16 present extensions which handle the added language features. In each extension, the outer function—*make-interp-2* in Figure 15 and *make-interp-3* in Figure 16—accepts an argument that represents the next interpreter to invoke for unrecognized terms. The inner function defines the interpreter proper.

Unfortunately, though the *MidFun* sequence of interpreters is better from the perspective of reuse, it is not equivalent to *EditFun*. Specifically, the term

```
(interp-3 (make-plus (make-minus (make-num 1)
                                 (make-num 2))
                     (make-num 3)))
```

results in an error in *MidFun*$_3$ but not in *EditFun*$_3$ because the recursive calls which evaluate the arguments to the addition operator in *MidFun*$_2$ invoke *interp-2*, which cannot handle subtraction. Therefore, *MidFun*$_2$ is not an extensible version of *EditFun*$_2$ with respect to *EditFun*$_3$.

The solution to this problem is presented in the *Fun* sequence (Figures 17, 18 and 19). The repertoires in this sequence are similar to those in *MidFun* but with a key difference: each inner interpreter accepts *two* arguments. The first argument is the expression, as before. The second argument is the interpreter that should be used for all recursive calls. The latter interpreter is also passed to the interpreter invoked for unrecognized terms. *pre-interp-2* and *pre-interp-3* are the values corresponding to the inner function declarations.[5]

This programming pattern emulates two properties:

[5]The successful extensibility of Emacs [32], which is one of the most widely used extensible products, uses "hooks" to enable extensions. One can understand hooks as a weak form of the *next* protocol in our functional pattern.

```
;; Exp = ∅
```

```
;; interp-1 : Exp ⟶ Num
(define (interp-1 expr)
  (error 'interp "no semantics for ~s" expr))
```

Figure 14: *MidFun*$_1$

```
;; Exp = ... | num (val) | plus (lhs rhs)

(define-struct num (val))
(define-struct plus (lhs rhs))

;; interp-2 : Exp ⟶ Num
(define (interp-2 e)
  ((make-interp-2 interp-1) e))
```

```
;; Interp = Exp ⟶ Num

;; make-interp-2 : Interp ⟶ Interp
(define (make-interp-2 next)
  ;; the-interp-2 : Interp
  (define (the-interp-2 expr)
    (cond
      ((num? expr) (num-val expr))
      ((plus? expr) (+ (interp-2 (plus-lhs expr))
                       (interp-2 (plus-rhs expr))))
      (else (next expr))))
  the-interp-2)
```

Figure 15: *MidFun*$_2$ Extension

```
;; Exp = ... | minus (lhs rhs)

(define-struct minus (lhs rhs))

;; interp-3 : Exp ⟶ Num
(define (interp-3 e)
  ((make-interp-3 interp-2) e))
```

```
;; Interp = Exp ⟶ Num

;; make-interp-3 : Interp ⟶ Interp
(define (make-interp-3 next)
  ;; the-interp-3 : Interp
  (define (the-interp-3 expr)
    (cond
      ((minus? expr) (- (interp-3 (minus-lhs expr))
                        (interp-3 (minus-rhs expr))))
      (else (next expr))))
  the-interp-3)
```

Figure 16: *MidFun*$_3$ Extension

inheritance The *next* argument of each interpreter dictates which interpreter's behavior it inherits. Technically, this construction implements an "extensible conditional", which object-oriented languages provide automatically by means of inheritance and dispatching.

modularity The interpreter provided as a second argument to each interpreter is expected to handle the entire language. Thus, interpreters do not need to be aware of the rest of the language, so long as they are given an extended interpreter and pass it appropriately when they make calls. This pattern has the same effect as the virtual constructor described in Section 4.2.

We can now show that the *Fun* sequence is extensible with respect to the members of *EditFun*. This validates our design pattern for extensible functional interpreters. More importantly, it illustrates the robustness of our definitions, which have a minimal dependence on the language's syntax.

6 Directions for Future Work

Ideally, we would like to characterize whole groups of programs, not just individual ones. Design patterns are a useful starting point, since they provide a convenient classification of programs and program fragments. Unfortunately, current formalisms for patterns [4, 8, 30] are based largely on their syntactic shape, not on their semantic properties. Since these descriptions do not account for the behavior of programs, they are probably incompatible with our definitions. In addition, the syntactic nature of these formalisms also commits them to certain language models, typically object-oriented ones. We intend to re-classify patterns based on semantic properties, such as those outlined above. Then we can formalize the extensibility of patterns with respect to certain properties.

It is fruitful to identify the facets of programs or patterns that designers might want to keep extensible. For instance, the Interpreter pattern defines a *datatype* (an abstract class

```
;; Exp = ∅

;; interp-1 : Exp ⟶ Num
(define (interp-1 e)
  (pre-interp-1 e pre-interp-1))
```

```
;; Interp = Exp × Interp ⟶ Num

;; pre-interp-1 : Interp
(define (pre-interp-1 expr int)
  (error 'interp "no semantics for ~s" expr))
```

Figure 17: *Fun*$_1$

```
;; Exp = ... | num (val) | plus (lhs rhs)

(define-struct num (val))
(define-struct plus (lhs rhs))

;; pre-interp-2 : Interp
(define pre-interp-2
  (make-interp-2 pre-interp-1))

;; interp-2 : Exp ⟶ Num
(define (interp-2 e)
  (pre-interp-2 e pre-interp-2))
```

```
;; Interp = Exp × Interp ⟶ Num

;; make-interp-2 : Interp ⟶ Interp
(define (make-interp-2 next)
  ;; the-interp-2 : Interp
  (define (the-interp-2 expr int)
    (cond
      ((num? expr) (num-val expr))
      ((plus? expr) (+ (int (plus-lhs expr) int)
                       (int (plus-rhs expr) int)))
      (else (next expr int))))
  the-interp-2)
```

Figure 18: *Fun*$_2$ Extension

```
;; Exp = ... | minus (lhs rhs)

(define-struct minus (lhs rhs))

;; pre-interp-3 : Interp
(define pre-interp-3
  (make-interp-3 pre-interp-2))

;; interp-3 : Exp ⟶ Num
(define (interp-3 e)
  (pre-interp-3 e pre-interp-3))
```

```
;; Interp = Exp × Interp ⟶ Num

;; make-interp-3 : Interp ⟶ Interp
(define (make-interp-3 next)
  ;; the-interp-3 : Interp
  (define (the-interp-3 expr int)
    (cond
      ((minus? expr) (- (int (minus-lhs expr) int)
                        (int (minus-rhs expr) int)))
      (else (next expr int))))
  the-interp-3)
```

Figure 19: *Fun*$_3$ Extension

like Subject) with a set of *variants* (the concrete subclasses like Prince). (Datatypes and variants are well-understood, formal entities in functional languages such as Haskell [17] or ML [24].) For each datatype, the pattern provides a set of *tools* (the methods in an object-oriented language, or functions in a functional one). It is clearly useful to keep the collection of variants and tools extensible.

We can then verify whether programs incorporate these properties in an extensible manner. For example, we conjecture that our version of the Interpreter pattern can support both datatype and tool addition in an extensible fashion. The proof highlights the steps that an implementation of the pattern must observe, such as delegating construction of variants to virtual constructors and updating them in concert with class extension.

Once we have identified a set of extensible properties, we can design programming constructs that automatically endows the program with these properties and maintain code dependencies automatically. The design of such constructs should be guided by identifying properties, as discussed above. The approach is outlined by Krishnamurthi, et al. [18]. We have prototyped this idea, and will present the details in a forthcoming publication.

7 Related Work

One of the earliest expositions of the idea of reusable software components is due to McIlroy. In his influential paper [22], McIlroy anticipated off-the-shelf software components that programmers could combine to produce complete programs.

Design patterns [15] are a modern variation on McIlroy's idea of software components. They are *designs* that programmers can combine to produce implementations (of systems). There are now numerous efforts to collate such patterns. These efforts, however, do not provide formal specifications of the properties of patterns, so users of patterns must convince themselves of the extensibility of individual patterns. Though there have been a few efforts to formalize patterns [4, 8, 30], these formalisms are effectively syntactic. Software architectures [31] are a more formal approach to the problem of categorizing software designs, and sometimes offer tool support [2]. However, they are concerned with general problems of software structure, not the specific domain of extensibility.

Many authors [19] have informally defined "black-box" reuse, which is the foundation for our notion of extensibility. O'Malley and Batory [3] define a formal semantics of program components in terms of their exported types, and use this to study principles like composition and reuse. In particular, they define two kinds of reuse: algorithmic and class-based. The latter captures the same essence as black-box reuse, but is still defined in terms of the syntactic structure of the system. We do not directly address algorithmic reuse.

Two groups of authors have attempted to provide formal specifications of extensibility that are comparable to ours. Frappier, Mili and Ben Ayed [14] use relational calculi to specify programs. They then compile the specifications into a conventional programming language. They compare the specifications for syntactic and semantic similarity, which are similar to our notions of containment and approximation, as well as proximity. However, they do not discuss relative extensibility. In addition, their approach requires the designer to write specifications in a specialized language, and their calculi do not include language features for control and program organization such as those found in most major contemporary languages. Nierstrasz, Schneider and Lumpe [26] attempt to use programming language semantics to formalize software compositions. The language they use is a variant on the π-calculus. They also mention the benefits of formalizing design patterns, but do not present concrete examples of formalisms.

In contrast to both these works, our underlying formalism [12], which uses a class-typed model [27] covers constructs such as classes, interfaces and mixins, and is closely related to calculi for control and state [10]. We also believe it is simpler and more tractable.

Our functional language example (Section 5) was implemented in Scheme, which does not have a static typing discipline. In typed functional languages such as as ML or Haskell, the solution is complicated because these languages do not directly offer user-definable extensible datatypes to implement the expression datatype. The solution can be encoded using ML's exceptions [11] or proposed mixin modules [7], or with Haskell's type classes [21].

Our work was inspired by studies of the expressive power of programming languages. Those works provide frameworks and proofs for classifying which language constructs can be expressed in terms of others. Felleisen [9] defines expressive language features to be those which cannot be expressed by local macro transformations. Mitchell's characterization [25] uses the types of terms to arrive at a different characterization of expressiveness. Both works employ the underlying language's observational equivalence relation as we do in Section 3, and both use contexts, which are implicit (and constrained to disallow definitions) in our work. Our notion of extension is closely related to Felleisen's *conservative extension* and Mitchell's *language extension*.

Krishnamurthi, Felleisen and Friedman [18] noticed the problem with the extensibility of the Interpreter pattern. The Extensible Interpreter presented in Section 4 is a variation on their Extensible Visitor. The *Fun* sequence in section Section 5 is based on the protocol presented by Cartwright and Felleisen [5].

8 Summary

We have described a notion of program reuse called *extensibility*. Extensible systems are important for two major reasons. First, they provide commercial software producers a way to distribute systems that can be customized by clients without access to source. Second, since extensibility is usually achieved only through careful design, satisfying our notion of extensibility forces programmers to embrace and exploit key software engineering principles such as abstraction and delegation. For example, programs are usually made more extensible by using inheritance, overriding and late-binding in object-oriented languages and by the use of parameterization and higher-order procedures in functional languages. It also compels both producers and clients to advertise and adhere to interfaces, instead of exploiting internal knowledge about the system.

Our definition characterizes *relative extensibility*. It determines whether a program can be extended to emulate the behavior of another. We briefly show how the theory can be used to drive the design of a new and useful design pattern. We also demonstrate that our definition is robust in that it applies with little modification to both functional and object-oriented languages.

Acknowledgments

We thank Cormac Flanagan for stimulating discussions. The anonymous referees provided helpful comments on the presentation.

References

[1] Abelson, H. and G. J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, MA, 1985.

[2] Allen, R. and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.

[3] Batory, D. and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, October 1992.

[4] Brown, K. G. Design reverse-engineering and automated design pattern detection in smalltalk. Master's thesis, North Carolina State University, 1996.

[5] Cartwright, R. S. and M. Felleisen. Extensible denotational language specifications. In Hagiya, M. and J. C. Mitchell, editors, *Symposium on Theoretical Aspects of Computer Software*, pages 244–272. Springer-Verlag, April 1994. LNCS 789.

[6] Drossopoulou, S. and S. Eisenbach. Java is type safe—probably. In *European Conference on Object-Oriented Programming*, pages 389–418, 1997.

[7] Duggan, D. and C. Sourelis. Mixin modules. In *International Conference on Functional Programming*, pages 262–273, May 1996.

[8] Eden, A. H., J. Gil and A. Yehudai. Precise specification and automatic application of design patterns. In *Automated Software Engineering*, 1997.

[9] Felleisen, M. On the expressive power of programming languages. *Science of Computer Programming*, 17:35–75, 1991.

[10] Felleisen, M. and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 102:235–271, 1992.

[11] Findler, R. B. Modular abstract interpreters. Unpublished manuscript, Carnegie Mellon University, June 1995.

[12] Flatt, M., S. Krishnamurthi and M. Felleisen. Classes and mixins. In *Symposium on Principles of Programming Languages*, pages 171–183, January 1998.

[13] Frakes, W. and C. Terry. Software reuse: Metrics and models. *ACM Computing Surveys*, 28(2):415–435, 1996.

[14] Frappier, M., A. Mili and R. Ben Ayed. A relational calculus for software reuse. In *IJCAI Workshop on Formal Approaches to the Reuse of Plans, Proofs and Programs*, 1995.

[15] Gamma, E., R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Personal Computing Series. Addison-Wesley, Reading, MA, 1995.

[16] Gosling, J., B. Joy and G. L. Steele, Jr. *The Java Language Specification*. Addison-Wesley, 1996.

[17] Hudak, P., S. Peyton Jones and P. Wadler. Report on the programming language Haskell: a non-strict, purely functional language. *ACM SIGPLAN Notices*, 27(5), May 1992. Version 1.2.

[18] Krishnamurthi, S., M. Felleisen and D. P. Friedman. Synthesizing object-oriented and functional design to promote re-use. In *European Conference on Object-Oriented Programming*, pages 91–113, 1998.

[19] Krueger, C. W. Software reuse. *ACM Computing Surveys*, 24(2):131–183, 1992.

[20] Leach, R. J. *Software Reuse: Methods, Models, and Costs*. McGraw-Hill, 1997.

[21] Liang, S., P. Hudak and M. Jones. Monad transformers and modular interpreters. In *Symposium on Principles of Programming Languages*, pages 333–343, 1992.

[22] McIlroy, M. D. Mass produced software components. In Naur, P. and B. Randell, editors, *Report on a Conference of the NATO Science Committee*, pages 138–150, October 1968.

[23] Mili, H., F. Mili and A. Mili. Reusing software: Issues and research directions. *IEEE Transactions on Software Engineering*, 21(6):528–562, June 1995.

[24] Milner, R., M. Tofte and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

[25] Mitchell, J. C. On abstraction and the expressive power of programming languages. *Science of Computer Programming*, 212:141–163, 1993.

[26] Nierstrasz, O., J.-G. Schneider and M. Lumpe. Formalizing composable software systems — a research agenda. In *IFIP Workshop on Formal Methods for Open Object-based Distributed Systems*, 1996.

[27] Palsberg, J. and M. I. Schwartzbach. *Object-Oriented Type Systems*. Wiley, 1994.

[28] Poulin, J. S. *Measuring Software Reuse: Principles, Practices, and Economic Models*. Addison-Wesley, 1997.

[29] Prieto-Díaz, R. Status report: Software reusability. *IEEE Software*, pages 61–66, May 1993.

[30] Sefika, M., A. Sane and R. H. Campbell. Monitoring compliance of a software system with its high-level design models. In *International Conference on Software Engineering*, 1995.

[31] Shaw, M. and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[32] Stallman, R. M. *GNU Emacs Manual*. Free Software Foundation, Cambridge, MA, 1993.