12-2012

Connotational Subtyping and Runtime Class Mutability in Ruby

Ian S. Dillon
East Tennessee State University

Connotational Subtyping and Runtime Class Mutability in Ruby

A thesis

presented to

the faculty of the Department of Computer and Information Sciences

East Tennessee State University

In partial fulfillment

of the requirements for the degree

Master of Science in Computer Science

---

by

Ian Dillon

December 2012

Dr. Phil Pfeiffer, Chair

Dr. Martin Barrett

Dr. Christopher Wallace

Keywords: ruby, type systems, class mutability

ABSTRACT

Connotational Subtyping and Runtime Class Mutability in Ruby

by

Ian Dillon

Connotational subtyping is an approach to typing that allows an object's type to change dynamically, following changes to the object's internal state. This allows for a more precise representation of a problem domain with logical objects that have variable behavior. Two approaches to supporting connotational subtyping in the Ruby programming language were implemented: a language-level implementation using pure Ruby and a modification to the Ruby 1.8.7 interpreter. While neither implementation was wholly successful the language-level implementation created complications with reflective language features like `self` and `super` and, while Ruby 1.8.7 has been obsoleted by Ruby 1.9 (YARV), the results suggest that Chambers-style, predicate-based runtime type inference could be incorporated into Ruby with only some reduced interpreter performance.

ACKNOWLEDGMENTS

My deepest gratitude to Dr. Donald Sanderson, who introduced me to my subject and whose guidance helped me through the early, faltering steps of my thesis work. The university is poorer for his passing.

CONTENTS

1 INTRODUCTION

A common strategy for software system construction treats software as a model of a problem domain that characterizes the domain's entities and their relationships [10]. One such strategy, *object-oriented programming* (OOP), treats a domain as a collection of logical objects with interfaces that define the behaviors that other objects may invoke. The OOP approach supports the use of intrinsic and extrinsic strategies for modeling behavior. Intrinsic strategies model behaviors as properties of an object's type, otherwise known as *is-a* relationships. Extrinsic strategies model behaviors using auxiliary objects that act on behalf of a first object, otherwise known as *has-a* relationships. Is-a and has-a relationships are used routinely to model problem domains, often in the same system. Their use, moreover, should be indistinguishable to an observer of a properly implemented system.

Even so, has-a and is-a relationships have distinct properties from the standpoint of language and systems implementation. Historically, is-a relationships for objects with changing behaviors have been avoided by designers of mainstream programming languages. Mainstream languages like C++, C#, and Java simplify the management of is-a relationships by assigning static types to language objects, thereby precluding the need for type systems that model changing behaviors. For such languages, multiple approaches have been devised for using has-a relationships to approximate dynamic is-a relationships. One such approach, the Strategy design pattern, models changing behaviors by acquiring and/or dropping references to a set of auxiliary objects, each of which implements one of the fluctuating behaviors [12]. A second, the State design pattern, models changes of behavior with a state machine, with each set of expressed behaviors implemented as a distinct class. These state classes, which are referenced by the original changing object, perform the expected behavior based on the referencing object's internal state [12]. Such approaches, while serviceable, yield object models that obscure the domain's relationships, along with a type system that can't assure the type safety of these dynamic changes in object behavior.

These concerns about the limitations of has-a relationships has fostered extensive research into type systems that allow objects' types to vary at runtime. This includes the languages Fickle, Cecil, ***e***, EXPRESS, and modifications to the Java runtime to support forms of object evolution. This thesis investigates the practicality of applying such research to a popular contemporary language, Ruby.

1.1 Connotational Subtyping: Rationale

Connotational subtyping is an approach to typing that allows an object's class to be reclassified dynamically, following changes to the object's internal state. In connotational subtyping, an object with varying behaviors can maintain its identity throughout a computation while assuming the type most appropriate for the object's current state. These changes in type can include changes to that object's external interface and properties: a capability that yields a more precise characterization of objects with changing behaviors than can be obtained with static types, as in languages like C++ and Java. Full implementations of connotational subtyping also detect changes in object type and check the validity of operations on objects automatically, relieving programmers of having to code these checks by hand or risk doing without.

To appreciate the benefits of connotational subtyping, consider its potential application in a drawing package that supports polygon objects with a variable number of edges. Modeling polygons as subclasses of a Polygon class with edge-count-based subtypes (e.g., triangle, rectangle, pentagon) would allow for polygon objects whose area methods changed automatically as their number of sides changed. The resulting class structure would reflect the actual taxonomy of polygons, in terms of the relationship between type and method of area computation. This close correspondence between object type and object behavior is especially useful in languages that support reflection, as a polygon's current behavior could be detected by inspecting its external interface. By contrast, a typical design for this package in statically typed languages would use has-a relationships to approximate these runtime changes in object type. An arbitrary set of characteristics would first be associated with a "baseline" abstract object that models each kind of polygons that a user could create. Changes in object type would be modeled by adding, subtracting, and/or modifying behaviors or characteristics that vary according to that object's states: typically, by either

- defining a concrete strategy class for each area method and manually updating references to the appropriate area classes in a baseline shape class (cf. [12], Strategy pattern) or,
- defining a set of state classes for each shape and allowing a base polygon class to select which to invoke, based on its current number of edges (cf. [12], State pattern).

This has-a-based approach sacrifices the ability to define precise, type-based characterizations relationships among polygons: i.e., one where `Square` objects are subtypes of `Rectangle` objects and `Triangle` objects can assume one of five subtypes, corresponding to the five different types of triangles.

Design patterns like Strategy and State become unnecessary under connotational subtyping. With State, for example, a state designation could be replaced by a class predicate method that determines an object's class from that object's state at the point of method invocation. In effect, the State pattern becomes a feature of the model's design. The same is true for Liskov's abstract devices [18], which rely on exception signaling to respond to non-implemented behaviors.

Connotational subtyping can also provide a clearer characterization of which guard clauses and sanity checks to enforce at key points in an object's lifetime. Instead of associating type-related checks with hand-coded assessments of object state, these checks could be associated with those types to which they apply. An inferencing algorithm for determining an object's current subtype could then apply the checks when determining an object's type. This centralizes the implementation of the checks and can reduce redundant code.

1.2 Connotational Subtyping: Difficulties

One difficulty in connotational subtyping is maintaining a computation's type safety. Implementations of statically typed systems typically check that an object's type in context `C` is a valid subtype of a type that is permissible at `C`. Such checks are facilitated by requiring every object to have exactly one type throughout a computation. In connotational subtyping, where an object's type can vary, the types that an expression's objects will assume relative to one another may be difficult or impossible to determine at compile time. For this reason, implementations of connotational subtyping typically defer checks of dynamically varying references to run time. Similar considerations hold for method invocation, insofar as type systems must ensure that each object can respond to every type of message it receives.

Other difficulties with changing an object's class involve class invariants. When an object's class changes, its attributes may need to change to meet the new class's requirements. An implementation must manage these changes in attributes. This could include initializing newly added data items, removing access to deleted items, and transforming items whose representation and/or value needs to change.

1.3 Implementing Connotational Subtyping in Ruby

This thesis explored strategies for implementing connotational subtyping in the Ruby programming language. Ruby is a dynamically typed, interpreted object-oriented programming language with many similarities to Smalltalk [2]. Ruby was selected due to its popularity, its open source interpreter, its extensive support for runtime metaprogramming, and its consistent object model.

Two strategies were explored for implementing connotational subtyping in Ruby. One, a language-level implementation, leveraged Ruby's dynamic programming capabilities and the Ruby/DL extension library with an unmodified Ruby interpreter. This language-level implementation should work with any recent, unmodified Ruby interpreter. The other strategy modified the Ruby 1.8.7-p174 interpreter to directly support connotational subtyping as a feature of its object model.

Both implementations succeeded, for the most part. The language level implementation introduced complications with language features like `self` and `super` due to the mechanics of the metaprogramming techniques used.

Both implementations also reduced interpreter performance, with the interpreter level outperforming the language level implementation. In hindsight, performance could have been improved with the use of better implementation strategies: i.e., by using a different class manipulation method for the language-level implementation and by making dynamic typing optional on a per-class basis for the interpreter-level implementation, as was done for the language level implementation.

2 BACKGROUND

2.1 Object-Oriented Programming

Object-oriented programming is a programming paradigm that uses *objects* as the primary units of computation. These objects are instances of *classes*, which define its objects' component elements (*attributes*), including those elements that implement its behaviors (*methods*) along with its externally accessible attributes (*interface*). Classes may *inherit* part of their definition from other classes, allowing for *polymorphism*. Each object in the program's runtime has its own state, identity, behaviors, interface, and lifespan. [3] [21]

2.2 Type

2.2.1 Type Proper

In programming languages, a *type* is a set of shared characteristics for one of a language's entities. These characteristics, which commonly include shared operators and attributes, help to determine what combinations of entities are valid for a computation's expressions.

The framework a language implements to define types and identify appropriate operations for types is called a *type system*. Determining if the interactions between types within a program are valid is called *type checking*.

Type systems and type checking algorithms are typically classified as *static* or *dynamic*. A static type system determines a type for every expression through a compile-time analysis of a program's properties. Such an analysis is referred to as a *static* analysis of a program's types. A dynamic type system, by contrast, determines the types of a computation's expressions and checks them for correctness at runtime. In a typical dynamically typed system, values are typed but identifiers are untyped: i.e., identifiers may refer to values of differing types throughout the variable's lifespan. [5]

2.2.2 Type In Object-Oriented Type Systems

Types and *classes* are related but distinct concepts. In object-oriented programming (OOP), a *class* is a factory for instantiating a family of related objects, each of which has a class-specified state, identity, interface, behavior, and lifetime. A *type* describes a common set of behaviors and characteristics that may be shared by multiple entities, including classes. In most OOP languages type and class are closely related, yet distinct. In classic type systems, an object's class is static but its type may change depending on the context of use. An object may be considered a member of any type for which the object's class satisfies the defined characteristics of that type.

A *subtype* is a type whose behaviors and characteristics are explicitly related to those of an existing type, called the subtype's *supertype*. Typically, subtypes are created from their supertypes by incrementally changing the supertype's behaviors or characteristics: i..e, by adding attributes or deleting or modifying existing attributes. In OOP languages, an object's type and applicable supertypes are normally denotational, since an object's class is declared at the point of its instantiation. Connotational subtyping differs from denotational typing by allowing for the dynamic recognition of an object's class relative to the context in which that object is used.

2.3 Runtime Class Mutability

2.3.1 Definition

The core mechanism for connotational subtyping is *runtime class mutability*: the ability for an object's declared class to change during execution while the object retains its identity.

2.3.2 Mutability Proper

Most languages and systems that support runtime class mutability do so by directly manipulating an object's internal class reference. It is this manipulation that introduces the difficulties of ensuring type safety and maintaining class invariants.

Runtime class mutability, while uncommon in major programming languages, is directly supported in two languages: the Common Lisp Object System (CLOS) and Smalltalk. CLOS supports runtime class mutability through the generic function `change-class`. This function takes two parameters: an object and a class that the object is to become. The change operation occurs "in place", meaning all references to that object remain unchanged, as does that object's location in memory. The `change-class` function retains any attributes in common between the instance's old class and new class and initializes any new attributes defined in new class. CLOS provides an optional method, `update-instance-for-different-class`, for changing how these attributes are initialized. The CLOS runtime system invokes this method on the modified object before returning control to the initial `change-class` point of invocation. CLOS, however, leaves it to the developer to ensure that the object can respond to all method calls supported by that object's new type [23].

Smalltalk supports runtime class mutability through its `become:otherObject` function. Smalltalk objects, however, lose their identity when modified. A call of `foo become:bar` changes all references to `foo` in the runtime environment to references to `bar`. This gives the appearance of a class change in those contexts that held a reference to `foo`. As per usual in Smalltalk, the developer must ensure type safety by ensuring the new object can handle any future messages [13].

Neither Smalltalk nor CLOS restrict the target class or target object. Neither language, moreover, attempts to ensure the change operation's type safety beyond the measures provided by the language's normal safeguards.

2.3.3 Object Evolution

Some extensions to existing object oriented languages support safe runtime class mutability by restricting a class change's target types to subclasses of the object's current class. This helps ensure type safety by only extending the available methods and never removing any methods from the object's interface, as subclasses can only override inherited methods or define new methods. This approach is generally called *object evolution* [7].

One such extension, Schlack's `evolveto` [22], is implemented as an extension of the Java Virtual Machine (JVM) that operates directly on objects. A second JVM-based system, described in [19], alters class definitions at runtime instead of individual objects, mutating all of a class's instances at once. This *dynamic classes* system is also discussed in [15], including concerns related to the granularity of class redefinition: i.e., which instances of the changing class are affected.

Drossopoulou et al.'s Fickle language [8, 9] introduces a *reclassification* operation that changes an object's class membership while maintaining the object's identity. Type safety is maintained by supporting two separate class types: *state classes*, which may be targets for reclassification, and *root classes*, the parent classes of all state classes. Root classes contain all the attributes their state subclasses have in common.

For a more comprehensive discussion of object evolution, different types of object evolution, and potential implementation strategies, see [7].

2.3.4 Predicate Classes

The work most similar to connotational subtyping is Chambers's *predicate classes.* Chambers's type system introduces a new class type that extends normal classes with a predicate statement that characterizes objects from that class. Any group of predicate classes that is a subclass of a normal class represents a subset of the instances of the superclass that satisfy their individual predicate statements. As an instance of the common superclass changes state throughout its lifespan, that object's type changes to whatever predicate subclass is best satisfied by that object's predicate statement. Class membership changes, which occur without programmer intervention, are determined at runtime during method dispatch. While an object may lose or gain methods and fields as its type changes, the changes are restricted to the predicate subclasses of the object's declared normal class [6].

2.4 The Ruby Language

Ruby is an object-oriented programming language developed by Yukihiro Matsumoto in the mid-1990's [2]. Its object model is similar to Smalltalk, in that all values are objects and methods are invoked through message passing rather than function calls. Ruby allows classes to handle runtime message passing errors by implementing `method_missing`, similar to Smalltalk's `doesNotUnderstand:` method. The Ruby interpreter invokes `method_missing` when an object is passed a message to which it doesn't respond. Classes in Ruby are "open", in that methods may be re-defined, added, or removed at any point after the class's initial declaration. This allows developers to add functionality to classes defined externally or in base Ruby classes like `String` or `Fixnum`. Ruby also supports first-class functions through its `lambda` construct. The combined use of Ruby's reflective capabilities, open classes, and first-class functions allows for flexible metaprogramming.

Ruby is dynamically typed, with all type checking performed at runtime. Ruby is *duck typed*: i.e., an object that supports those methods that are invoked in a given context is deemed valid for that context [24]. Calling an unsupported method results in a runtime error. Ruby also supports *type introspection*, the ability to query an object's type at runtime. This allows a developer to ensure an object being acted upon supports a specific implementation of a required method.

2.5 An Example

Gamma et al. describe the use of the State pattern to implement objects for managing TCP connections [12]. In their design a class, `TCPConnection`, models the connection with a remote server as perceived by one of that server's client processes (see Figure 1). This `TCPConnection` class maintains a reference to a state class, `TCPState` in the example, which defines an abstract interface for the allowable actions for the client-server connections current state. These actions are implemented in `TCPState`'s concrete subclasses: i.e., `TCPEstablished`, `TCPListen`, and `TCPClosed`. As the `TCPConnection` object's state changes during its lifespan, the `TCPState` reference held by `TCPConnection` is modified to reference a new concrete state object. Gamma et al.'s design, shown in Figure 1, requires the `TCPConnection` class to define a method for every action that can be taken at any point during a client-server session, including actions, like message sending, that are not permissible for all states.

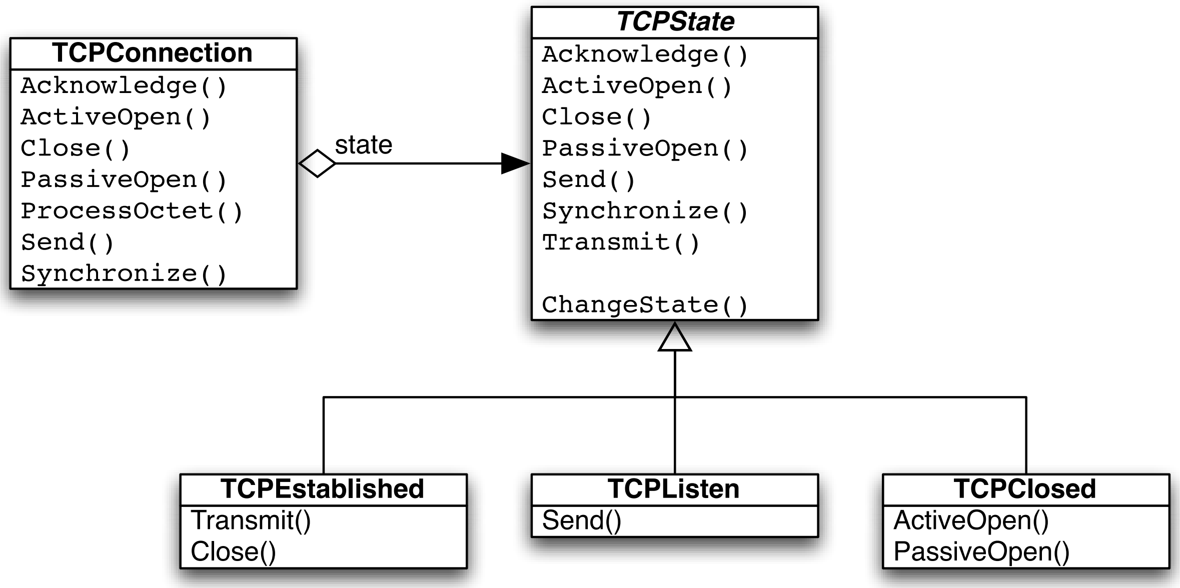


Figure 1: TCPConnection State Pattern Class Structure

In a connotational subtyping-based implementation of this functionality, the connection's state changes could be modeled directly and managed by the runtime system (see Figure 2). This simplifies the class structure, eliminating the need for a separate `TCPState` class and also the need to model all possible actions in the abstract `TCPConnection` class.

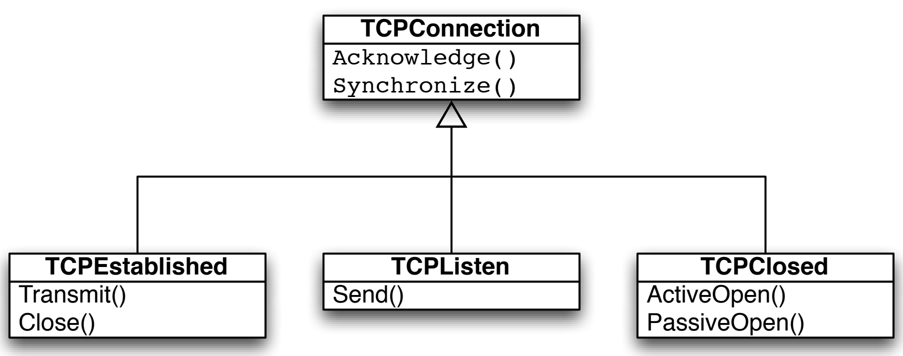


Figure 2: TCPConnection Connotational Subtyping Class Structure

This example assumes that the class-determining code is present in the `TCPConnection` parent class. Using connotational subtyping, an object would be initially created as an instance of the base `TCPConnection`. As the object's internal state changes, its class would transition from `TCPEstablished` to `TCPListen` and, finally, to `TCPClosed`. Eliminating the need for a common interface allows the connotationally subtyped object's public interface to model just the appropriate actions for the object's current state. Thus, introspection and reflection can be used to accurately determine an object's current abilities and the object's implemented behaviors would be appropriate for its current state.

This use of connotational subtyping also eliminates a form of coupling that the State pattern requires. In the State pattern example, the concrete state classes are aware of their sibling classes as each class must handle state transitions through the `ChangeClass` method. Since class transitions in the connotational subtyping example are automatic, the knowledge of state transitions can be centralized in the class determination code in `TCPConnection`, eliminating the need for concrete state subclasses to reference their sibling classes.

3 METHODOLOGY

3.1 Goals

This investigation sought to discover the difficulties and feasibility of implementing connotational subtyping in a modern object-oriented programming language. Starting with an established programming language allowed the work to begin from an established base and to observe how connotational subtyping would integrate into an existing object model. This research's criteria for evaluating this implementation are impact on interpreter performance, correctness, ease of use, and implementation portability; i.e., the ability to reuse the implementation with different architectures or versions of the language.

3.2 Experimental Design

The research was conducted by completing and benchmarking two different implementations of connotational subtyping. One, a language-level implementation, used standard Ruby programming language constructs to implement connotational subtyping. The other, an interpreter-level implementation, modified Ruby's internals to incorporate connotational subtyping into the language itself. The implementations were then benchmarked for performance, reviewed for correctness, and evaluated for ease of use.

3.3 Implementation

3.3.1 Requirements

A programming language suitable for implementing connotational subtyping must be dynamically typed, as the type of expressions involving connotationally subtyped object may be indeterminate. This research also used a free, open-source implementation, due to funding constraints. Other key requirements for language selection included support for an interpreter for ease of manipulation and support for metaprogramming, due to the experimental design, which involved language-level implementation and the need to support runtime changes to classes.

3.3.2 Realization

Ruby was selected as the language for the thesis work because of its open source interpreter, extensive support for runtime metaprogramming, and consistent object model. The 1.8.7 version of Ruby was chosen for this thesis because of the simplified abstract syntax tree (AST) evaluation used in Ruby 1.8.

Ruby 1.9 proved less well suited for this work, due to the introduction of the YARV (Yet Another Ruby VM) byte-code compiler and virtual machine in the interpreter's implementation. YARV complicated object manipulation by introducing bytecode compilation and more complex strategies for AST evaluation, such as inline method caching.

3.3.2.1 Language Level

Connotational subtyping was implemented at the language level by first developing a method for safely modifying an object's declared class at runtime, then combining it with a second method for intercepting method calls on objects. Intercepting method calls provides a place to re-evaluate and change the receiving object's class per the class determining method of the connotationally typed receiver.

This work used the strategy for modifying an object's class employed by Florian Groß's Evil Ruby project [14] and Jeremy Evans's evilr extension [11]. Evil Ruby uses the Ruby/DL extension, which provides access to the dynamic linker, to extend Ruby functionality by directly manipulating the running Ruby interpreter's structures in memory. Evil Ruby extends Ruby's `Object` and `Class` classes to support the manipulation of object flags, modification of inheritance chains, and instance variable sharing between two objects. The class-changing ability is provided by extending the `Object` class with the `class=` method. Evilr replicates the features of Evil Ruby, but is written as a C extension instead of pure Ruby.

Ruby's internal object model is based on the three core structures `RBasic`, `RObject`, and `RClass`:

```c
struct RBasic {
    unsigned long flags;
    VALUE klass;
};

struct RObject {
    struct RBasic basic;
    struct st_table *iv_tbl;
};

struct RClass {
    struct RBasic basic;
    struct st_table *iv_tbl;
    struct st_table *m_tbl;
    VALUE super;
};
```

Ruby uses variables of type `VALUE`, an alias for the `unsigned long` data type, to reference internal structs. Ruby casts `VALUE` to the specific struct type as needed. Ruby's `RObject` struct is one of several built-in Ruby internal representations of user-created objects in Ruby. `RObject`, like `RString`, `RArray`, `RHash` and other built-in Ruby primitive representations, has its own internal type.

All of Ruby's internal type structs contain an `RBasic` struct. This struct's `klass` pointer references the instance of the `RClass` struct that represents the object's declared class. The `RClass` struct's `super` pointer, which references a class's superclass, helps to define an object's inheritance hierarchy. These relationships are depicted in Figure 3, which shows the in-memory struct relationships created by this example of a class declaration and instantiation:

```ruby
class ExClass
  attr_accessor :name
end

ex_object = ExClass.new
```

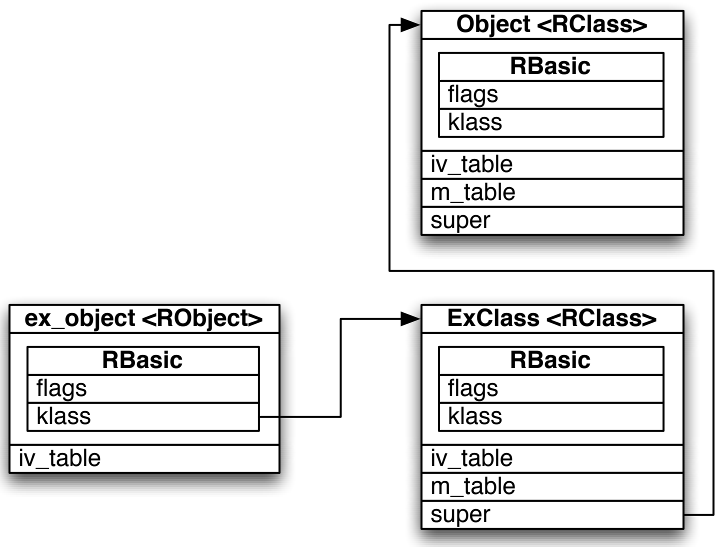


Figure 3: RObject and RClass in Memory

Evil Ruby and evilr modify an object's class at runtime by changing the `klass` pointer to reference the `RClass` of another class, with some restrictions. Ruby uses internal type flags to identify built-in types like String, Hash, and Array, each of which has a distinct internal struct. Ruby treats these base types as mutually incompatible, restricting class changes to classes with the same internal struct. For example, an object with a class derived from `String` may not change to `Hash` because the missing internal `RHash` struct would cause a segmentation fault on the next attempted access.

Ruby metaprogramming techniques were used to intercept method calls on connotationally typed objects. Class declarations for connotationally typed classes were augmented with a module, `ConnotationalSubtyping`, that intercepts and alters the definitions of newly defined methods as those methods are added to a class's definition. The `ConnotationalSubtyping` module exploits two metaprogramming-related features of Ruby's interpreter. The first, a class method hook called `method_added`, is invoked when a new method is defined in a class. The other, the `alias_method`, allows an existing method to be overridden but maintained under a new name. The module that intercepts and alters method definitions essentially replaces a method with a new "wrapper" method that first calls the method `cs_det`, which determines the current object's current class. This wrapper method then changes the current object's class handle, if necessary, before invoking the original method. This mechanism could be described as a simplified form of aspect-oriented programming that treats method invocation as a join point and `cs_det` as the advice [17]. The completed module, Listing 1, uses the `Module.included` method to add the `method_added` class method to classes that import the module.

Listing 1: ConnotationalSubtyping Module

```
module ConnotationalSubtyping
  def self.included(base)
    base.extend ClassMethods
  end

  module ClassMethods
    def method_added(method_name)
      return if [:initialize,:cs_det,:cs_class_changed].include?(method_name) or @added
      @added = true
      self.class_eval <<-END
        alias_method "__cs_#{method_name}", "#{method_name}"
        def #{method_name}(*args)
          klass = cs_det
          if self.class != klass
            old_klass, self.class = self.class, klass
            self.cs_class_changed(old_klass) if self.respond_to?(:cs_class_changed)
          end
          self.send(:__cs_#{method_name}, *args)
        end
      END
      @added = false
    end
  end
end
```

The ConnotationalSubtyping module assumes that the class determining code is contained in a method called cs_det that returns a result of type Class. The module also invokes the method cs_class_changed when an object's class is changed. This gives the object an opportunity to react to a class change: e.g., to initialize instance variables or checking class invariants.

This language level implementation of connotational subtyping allows a developer to selectively and explicitly apply connotational subtyping to a class hierarchy. The ConnotationalSubtyping module only affects classes and the descendants of classes that import it, making these class structures easier to manage while reducing the overhead on the overall system. The language level implementation is also portable across multiple Ruby interpreter versions and requires no modifications to the interpreter.

3.3.2.2 Interpreter Level

Like the language level implementation, modifying the Ruby interpreter to support connotational subtyping required combining the methods for changing an object's class at runtime and intercepting method calls on objects. Similar to language level implementation, an object's class is modified by manipulating the klass pointer of the object's underlying RObject struct to reference a different RClass struct.

To intercept method calls on objects, the interpreter's NODE_CALL handler, which evaluates nodes in the Ruby AST, was changed to determine if the current receiver is connotationally subtyped. Because this check is performed on all program objects, a mechanism was needed to determine whether an object was connotationally subtyped. The current implementation tests for connotationally subtyped objects by checking if an object responds to the method cs_det. Once the interpreter determines that an object is connotationally subtyped it calls the object's `cs_det` method to determine that object's class. If the call to `cs_det` returns a class that differs from an object's current class, that object's `klass` pointer is changed to refer to the Class returned by `cs_det`. If the current receiver's class was changed and the object responds to `cs_class_changed` then it is also called on the receiver. Method invocation then proceeds normally and the original method is called on the receiver.

4 RESULTS

4.1 Correctness

The modified interpreters support the creation of connotationally subtyped class structures like the `TCPConnection` example from Figure 2 (page 17). The following code in Listing 2 implements the class structure from Figure 2, using the state of the @socket instance variable to determine the appropriate subclass for a `TCPConnection` object. Unlike the *Design Patterns* example in Figure 1 (page 17), which uses separate instances of each concrete state class, an instance of the connotationally subtyped `TCPConnection` class below maintains its identity and instance variables as its class changes.

Listing 2: Ruby TCPConnection Example

```
class TCPConnection
  def initialize
    @socket = :open
  end

  def cs_det
    case @socket
      when :open
        return TCPListen
      when :established
        return TCPEstablished
      when :closed
        return TCPClosed
      else
        raise "TCPConnection in unknown state."
    end
  end

  def acknowledge
    ...
  end

  def synchronize
    ...
  end
end

class TCPListen < TCPConnection
  def send
    # send SYN, receive SYN, ASK, etc.
    @socket = :established
  end
end

class TCPClosed < TCPConnection
  def active_open
    # send SYN, receive SYN, ASK, etc.
```

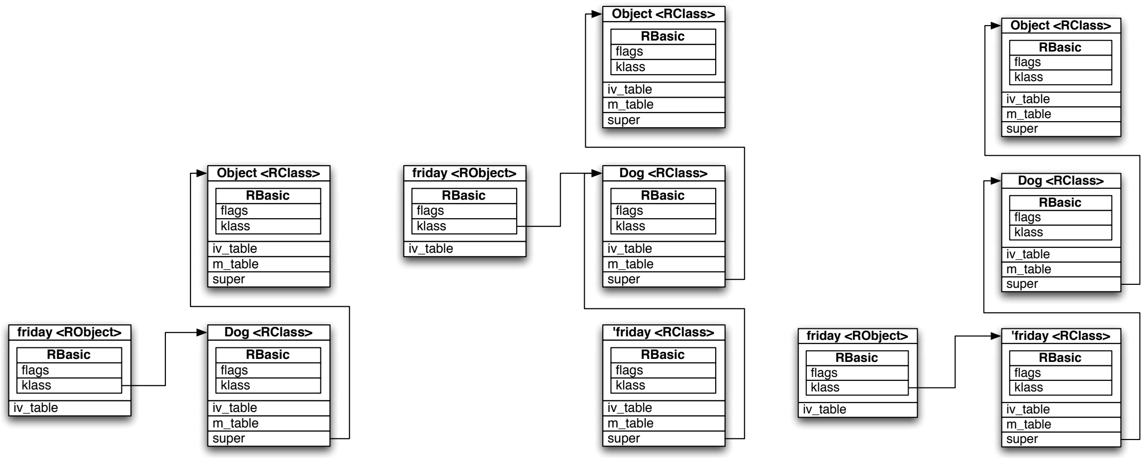
```ruby
    @socket = :established
  end

  def passive_open
    @socket = :open
  end
end

class TCPEstablished < TCPConnection
  def transmit(octet)
    @socket.process_octet octet
  end

  def close
    # send FIN, receive ACK of FIN
    @socket = :closed
  end
end
```

The strategies used here for implementing connotational subtyping, however, can also cause certain otherwise correct programs to fail. The `ConnotationalSubtyping` module could cause correctly terminating programs that consume at least half of an interpreter's call stack to fail due to stack overflow. Invoking a wrapped method in a class that includes the `ConnotationalSubtyping` module consumes two stack frames, one for the core method and one for the wrapping code. This doubling of call stack space requirements could cause recursive calls to connotationally subtyped methods to exhaust the interpreter's stack depth in half the number of calls that would be required by a non-connotationally subtyped method. Reaching the maximum call stack depth causes a fatal error in the interpreter, halting program execution. Increasing the call stack depth requires either using `ulimit` where available or re-compiling the Ruby interpreter with a higher stack size flag.

Stack overflow errors can also result from self-referential calls to Ruby's `cs_det` method. A call on connotationally subtyped class method `C#m` from a class C's `cs_det` method invokes `C#cs_det` due to `C#m`'s being wrapped by the ConnotationalSubtyping module. Any reference to `self#m` in `C#cs_det` will cause `C#cs_det` and `C#m` to repeatedly call one another until the interpreter's maximum call stack depth is reached. This problem doesn't occur in the modified interpreter implementation, as methods invoked on `self` do not trigger class re-evaluation so `cs_det` is not called. This reduces the potential confusion of an object's class changing while its state is in flux.

A similar situation arises with references to `super`, due to `ConnotationalSubtyping`'s use of the `Object#send` method to invoke the originally called method in its method aliasing code. The problem occurs because `Object#send` uses standard method dispatch, which invokes a method's "lowest" implementation. If a method that has been aliased by `ConnotationalSubtyping` invokes `super` and the superclass's implementation of the method has also been aliased, then the superclass's wrapping code's call to `Object#send` will invoke the "lowest" implementation of the method, leading to a recursive loop when the `super` invocation is reached again. This error does not occur in the modified interpreter implementation.

A final concern relates to the failure of Evil Ruby and evilr to correctly manage *singleton classes*. Ruby allows developers to define new methods that are specific to a particular instance of a class, as well as to override existing methods. These object-specific methods are called *singleton methods*. The Ruby interpreter implements singleton methods by creating a hidden Class object, a *singleton class*, whose method table contains the definition for the singleton methods. This new singleton class is then inserted in the class inheritance chain between the object and the object's original class (cf. Figure 4).



(a) Initial state (b) Singleton class created (c) Object's klass pointer updated

Figure 4: A singleton class being created

The interpreter's method dispatch for a modified object checks the singleton class before moving up the class's `super` pointer to the originally declared class. So any method definitions specific to an object would first be found in the singleton class's set of method definitions.

Neither Evil Ruby nor evilr checks the object's `klass` pointer to determine if the immediate Class being referenced is a singleton class. So when the `class=` method modifies the object's `klass` pointer the object loses the singleton class reference, removing all singleton methods that had been defined. The modified interpreter implementation, however, maintains the references to all singleton classes for the object being modified. If an object's immediate `klass` reference is a singleton class, denoted by the `FL_SINGLETON` flag, the singleton class's `super` pointer is changed to reference the object's new `RClass` struct. This results in the object's class being changed while maintaining the object's `klass` reference to an associated singleton class.

4.2 Performance

To determine how the connotational subtyping modifications affected the interpreter, the modified and unmodified Ruby 1.8 interpreters were compared using the Ruby Benchmark Suite (RBS) and its micro-benchmarks group of tests [4]. The micro-benchmarks test set is intended to measure the overall performance of core Ruby functionalities in different implementations and does not use any of the features of connotational subtyping. All benchmarks were run on a 1.6Ghz Intel Atom CPU with 2GB of RAM running Ubuntu 11 and the Linux 3.0 kernel. All benchmarks were run ten times and the value of these runs was recorded as the result.

The results of the RBS benchmarks of the modified and stock Ruby 1.8 interpreters are included in Appendix A (Ruby Interpreter Benchmarks) on page 44. An impression of the results can be obtained from Figure 5, which compares the total percentage of benchmarks at or below the percentage change in benchmark completion time.

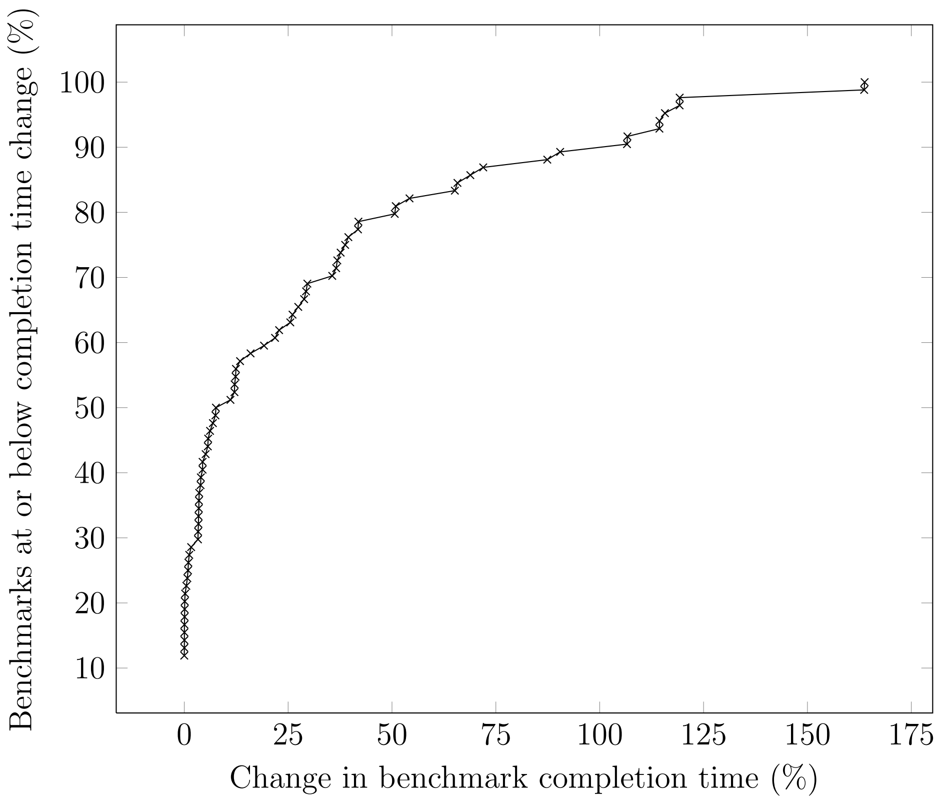


Figure 5: RBS results, comparing percentage of benchmarks at or below percentage change in completion time

To compare the performance of the connotational subtyping features, a new single benchmark was developed. This benchmark iterated a single object's class membership through three classes in series, e.g. `ClassA`→`ClassB`→`ClassC`→`ClassA`, etc., with each class change being considered an iteration. This new single benchmark was then executed for 100, 1,000, and 10,000 iterations for each of the following four Ruby configurations:

- The modified Ruby 1.8 interpreter (v1.8-connsub)
- The stock Ruby 1.9 interpreter using the ConnotationalSubtyping module and the `evilr` extension for class mutability (v1.9 evilr)
- The stock Ruby 1.8 interpreter using the ConnotationalSubtyping module and the

`evilr` extension for class mutability (v1.8 evilr)

- The stock Ruby 1.8 interpreter using the ConnotationalSubtyping module and the Evil Ruby extension for class mutability (v1.8 Evil Ruby)

The results of this initial execution produced the results table in Figure 6 and graph in Figure 7.

Input Size	v1.8 EvilRuby	v1.8 evilr	v1.9 evilr	v1.8-connsub
100	1.16961	0.001755	0.000858881	0.000156
1,000	11.982608	0.017863	0.00799318	0.001341
10,000	120.784384	0.18203	0.083585888	0.013164

Figure 6: First connotational subtyping benchmark completion time, in seconds

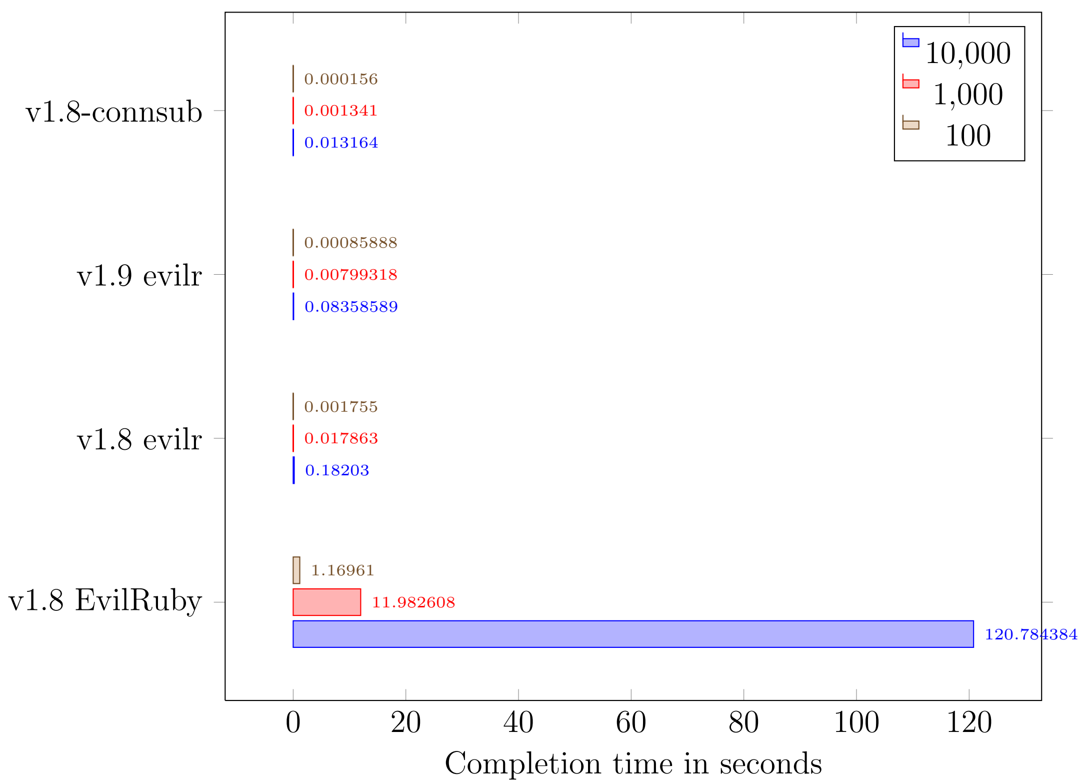


Figure 7: First connotational subtyping benchmark completion time, in seconds

A second, more intensive round of tests was then conducted with evilr v1.8, evilr v1.9, and v1.8-connsub. These tests, which increased the number of class change iterations to 10,000, 100,000, and 1,000,000, produced the results table in Figure 8 and graph in Figure 9.

Input Size	v1.8 evilr	v1.9 evilr	v1.8-connsub
10,000	0.1633	0.082	0.0119
100,000	1.6382	0.8449	0.119
1,000,000	16.4835	8.4994	1.1945

Figure 8: Second connotational subtyping benchmark completion time, in seconds

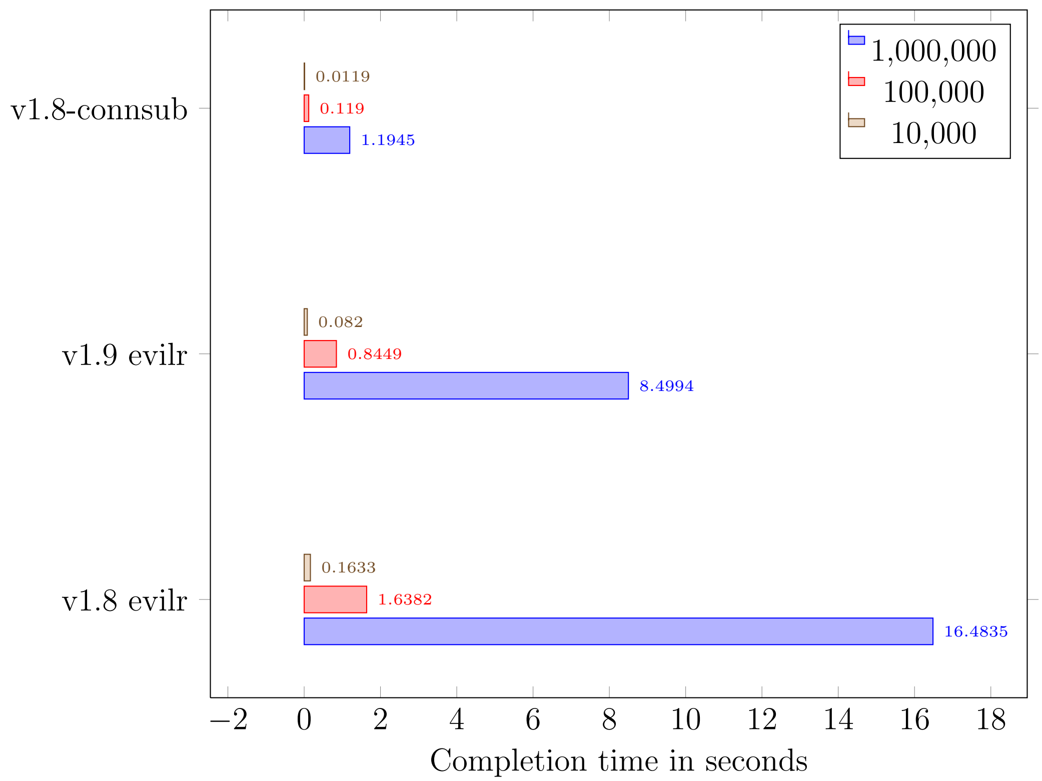


Figure 9: Second connotational subtyping benchmark completion time, in seconds

The increased completion times were consistent with the linear rate of growth seen in the first test. The results of all benchmarks are analyzed in Section 5.2, Performance.

5 ANALYSIS

5.1 Correctness

Although the language-level and interpreter-level implementations of connotational subtyping supported automatic runtime class changes, each had shortcomings that should be addressed in future work. The native Ruby implementation should be improved by modifying the `class=` method of Evil Ruby and `evilr` to account for singleton classes while walking the target object's class hierarchy. A filter should also be included with the `ConnotationalSubtyping` module that selects, either by pattern matching method names or by a static list, the set of methods that the `ConnotationalSubtyping` module should "wrap" with `alias_method` to intercept method invocation for class re-evaluation. This would improve performance by limiting aliasing to the methods to which class re-evaluation should apply while making it possible to safely alias inherited methods.

5.2 Performance

Roughly speaking, the impact of connotational subtyping on interpreter performance decreased in proportion to the number of *immediate* objects that a benchmark manipulated. Immediate objects are frequently used objects that are stored directly in the pointers that Ruby uses to reference its object heap, rather than in the heap proper. Immediate objects are of types `Fixnum` and `Symbol` and values `true`, `false`, and `nil`.

The impact of connotational subtyping on interpreter performance is less noticeable for those benchmarks that primarily manipulate immediate objects or whose processing is dominated by syscalls, i.e. sockets and filesystem I/O. The impact increases as the number of non-primitive, heap stored objects being manipulated increases. A few benchmarks showed an improvement in the modified Ruby 1.8 interpreter of less than 1%. These slight improvements can be attributed to slight testbed environment and environment differences as the minimum and maximum results of these benchmarks overlap for the two tested interpreters. The two anomalous results in `eval.rb` and `read_large.rb`, however, were consistent across multiple runs of either benchmark and may possibly be attributed to different compiler optimizations or system call library performance.

The performance comparison of the connotational subtyping implementations show the inefficiency of Evil Ruby's Ruby-based in-memory struct manipulation as compared to the native `C` extensions used by `evilr`. The second set of connotational benchmark results make the base performance improvements of Ruby 1.9 apparent, with the Ruby 1.9 benchmarks completing in roughly 50% of the time of the same `ConnotationalSubtyping` configuration in Ruby 1.8. However, the modified Ruby 1.8 interpreter still outperforms the Ruby 1.9 `ConnotationalSubtyping` module configuration, due to the overhead of language-level method wrapping.

5.3 Ease of Use

One point of confusion with both implementations is determining at runtime if a particular class or object is connotationally subtyped. The modified interpreter treats objects that respond to `cs_det` as connotationally subtyped and *all* external invocations of methods on that object trigger `cs_det`. Checking if an object responds to `cs_det` can be done at runtime using `Object#responds_to?`. This means that any descendants of a class that implement `cs_det` are also considered connotationally subtyped, as they inherit their ancestor's `cs_det`. Connotational subtyping can also be restricted to a particular *object* via a singleton method implementation of `cs_det`.

One point of confusion with both implementations is determining at runtime if a particular class or object is connotationally subtyped. The modified interpreter relies on convention for determining connotational subtyping: if an object responds to `cs_det` then it is considered connotationally subtyped and *all* external invocations of methods on that object trigger `cs_det`. Checking if an object responds to `cs_det` can be done at runtime using `Object#responds_to?`. This means that any descendants of a class that implements `cs_det` are also considered connotationally subtyped, as they have inherited their ancestor's implementation of `cs_det`. Connotational subtyping can also be restricted to a particular *object* via a singleton method implementation of `cs_det`.

The language-level implementation, however, requires the explicit inclusion of the `ConnotationalSubtyping` module into a class. Once this module has been imported then its effects also apply to any descendant classes. It's possible to determine if a class or any of its ancestors include `ConnotationalSubtyping` through the `Kernel#include?` method. Another possible point of confusion in the language-level implementation is the `ConnotationalSubtyping` module's use of the `method_added` callback method. The `method_added` class hook only updates methods that are defined after a class imports `ConnotationalSubtyping`. Methods that are present in a class prior to the import won't be wrapped by the module and thus won't trigger the class evaluation code. This includes all methods inherited from a superclass. So while it's possible to determine if a class has included the `ConnotationalSubtyping` module, it's difficult to determine what subset of the class's methods can trigger class re-evaluation. Additionally, any other code that overrides `method_added` in a class that includes the `ConnotationalSubtyping` module will silently break the module's wrapping of newly defined methods.

5.4 Applicability to Other Contexts

5.4.1 Ruby 1.9

The language-level implementation in the `ConnotationalSubtyping` module will work on any Ruby version supported by Evil Ruby or `evilr`. This currently includes Ruby 1.8.x and Ruby 1.9.2.

The modifications made to the Ruby 1.8.7 interpreter for connotational subtyping are specific to the 1.8.x versions of Ruby and will not work in the Ruby 1.9.x versions due to the introduction of the YARV virtual machine in the baseline interpreter. Further research would be needed to determine how to best implement connotational subtyping in Ruby 1.9, given the introduction of the Ruby virtual machine and the added runtime performance enhancements like inline method caching, stack caching, and specialized compiled virtual machine instructions.

The modified interpreter would best be improved by introducing a new class type, `csclass`, with associated internal flags. This would make it unnecessary to check for `cs_det` to determine if type re-evaluation is appropriate. Currently, the necessity of checking for `cs_det` at every method invocation and the inability to cache the checks' results are the main drags on interpreter performance. Instead, the interpreter could simply check the internal type of the receiver's class, a much simpler and faster operation.

5.4.2 Other Languages

This research was originally inspired by the draft proposals of the EXPRESS standards working group. The initial research advisor, the late Dr. Donald Sanderson, was a member of this committee. The EXPRESS data modeling language was standardized as part of the

ISO 10303 for the representation and exchange of product manufacturing information [16]. During the WG11's draft proposals for the next version of EXPRESS, the *connotational subtype* was introduced.

> A connotational subtype allows an instance of a particular supertype to be used as if it were a subtype instance also even though the instance is not declared to be of that particular subtype. The connotational subtype construct directs an information base to treat a supertype instance as if it were a subtype instance if it meets all constraints specified in the subtype. [20]

The constraints of a connotational subtype were predicates on its supertype's attributes, as connotational subtypes could not contain explicit attribute declarations. Variables declared as a connotational subtype's supertype would become members of the connotational subtype when the state of the variable's attributes satisfied the declared predicate statements of the connotational subtype's definition. At that point the variable would be a valid value for functions that used values of the connotational subtype as parameters. The working group's draft included an example of a `pensioner` connotational subtype of a `person` supertype and a `pension` function that would only accept valid `pensioner` parameters.

```
ENTITY person;
        name : personal_name;
        age  : natural;
        ...
END_ENTITY;

ENTITY pensioner
CONNOTATIONAL SUBTYPE OF (person);
WHERE
        old: age > 65;
END_ENTITY;
```

```
FUNCTION pension (subject : pensioner) : REAL;
    ...
END_FUNCTION;
```

Ultimately, connotational subtype declarations were removed from the working group's draft proposal.

The connotational subtyping implementation presented here may also be applicable to another popular interpreted programming language, Python [1]. Similar to the `class=` method used in this work, Python supports object class mutability by allowing an object's `__class__` attribute to be changed after instantiation. Python's metaprogramming support would also allow the same aspect-oriented programming approach to wrap method invocation as used in the Ruby language-level implementation.

6 CONCLUSION

The research presented here shows connotational subtyping as a promising, feasible addition to the Ruby environment. While both connotational subtyping implementations implemented the base features of connotational subtyping, the language-level connotational subtyping implementation potentially impacted Ruby language features and reduced interpreter performance more than the interpreter-level implementation. The language-level implementation, however, was more portable than the interpreter-level modifications.

The techniques used in this research could be most improved by determining a new connotational determinant approach other than the `cs_det` convention, like a new declared class type or internal type flag. This would allow for selective application of connotational subtyping and the elimination of object checks in the interpreter-level implementation, reducing the performance impact of the interpreter modification.

BIBLIOGRAPHY

[1] Python programming language. http://www.python.org/, Jan 2011.

[2] Ruby programming language. http://www.ruby-lang.org/en/, Jan 2011.

[3] Abadi, M., and Cardelli, L. *A Theory of Objects*. Springer-Verlag, Secaucus, NJ, 1996.

[4] Cangiano, A. Ruby benchmarks suite. https://github.com/acangiano/ruby-benchmark-suite, Jan 2011.

[5] Cardelli, L., Donahue, J., Jordan, M., Kalsow, B., and Nelson, G. Type systems. In *The Computer Science and Engineering Handbook* (1997), CRC Press, pp. 2208–2236.

[6] Chambers, C. Predicate classes. In *Proceedings of the 7th European Conference on Object-Oriented Programming* (London, UK, 1993), ECOOP '93, Springer-Verlag, pp. 268–296.

[7] Cohen, T., and Gil, J. Y. Three approaches to object evolution. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java* (New York, NY, USA, 2009), PPPJ '09, ACM, pp. 57–66.

[8] Drossopoulou, S., Damiani, F., Dezani-Ciancaglini, M., and Giannini, P. Fickle: Dynamic object re-classification. In *Proceedings of the 15th European Conference on Object-Oriented Programming* (London, UK, UK, 2001), ECOOP 01, Springer-Verlag, pp. 130–149.

[9] Drossopoulou, S., Damiani, F., Dezani-Ciancaglini, M., and Giannini, P. More dynamic object reclassification: Fickle ii. *ACM Trans. Program. Lang. Syst. 24* (March 2002), 153–191.

[10] Evans. *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[11] Evans, J. evilr. `https://github.com/jeremyevans/evilr`, Jan 2011.

[12] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, Boston, MA, USA, 1994.

[13] Goldberg, A., and Robson, D. *Smalltalk-80 : The Language*. Addison-Wesley, Reading, MA, 1989.

[14] Gross, F. evil-ruby. `http://code.google.com/p/evil-ruby/`, Jan 2011.

[15] Hjálmtýsson, G., and Gray, R. Dynamic c++ classes: a lightweight mechanism to update code in a running program. In *Proceedings of the annual conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 1998), ATEC '98, USENIX Association, pp. 6–6.

[16] ISO 10303-11:2004. *Industrial automation systems and integration - Product data representation and exchange - Part 11: Description methods: The EXPRESS language reference manual*. ISO, Geneva, Switzerland.

[17] Kiczales, G., and Hilsdale, E. Aspect-oriented programming. *SIGSOFT Softw. Eng. Notes 26* (September 2001), 313–.

[18] Liskov, B. Keynote address - data abstraction and hierarchy. *SIGPLAN Not. 23* (January 1987), 17–34.

[19] Malabarba, S., Pandey, R., Gragg, J., Barr, E., and Fritz Barnes, J. Runtime support for type-safe dynamic java classes. In *ECOOP 2000 Object-Oriented Programming*, E. Bertino, Ed., vol. 1850 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2000, pp. 337–361.

[20] N81, I. *ISO/WD 10303-11 Product data representation and exchange: Description methods: The EXPRESS language reference manual.* ISO, Geneva, Switzerland, 1999.

[21] Page-Jones, M. *Fundamentals of object-oriented design in UML.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 2000.

[22] Schalck, R. Object evolution: Adding runtime class mutability to the jvm. Master's thesis, Massachusetts Institute of Technology, January 2005.

[23] Steele, G. L. *Common LISP: The Language*, 2nd ed. Digital Press, Bedford, MA, 1990.

[24] Thomas, D., Fowler, C., and Hunt, A. *Programming Ruby 1.9: The Pragmatic Programmers' Guide*, 3rd ed. Pragmatic Bookshelf, 2009.

APPENDICES

Appendix A: Ruby Interpreter Benchmarks

Results are completion times in seconds.

Benchmark File	Input Size	Ruby 1.8.7	Ruby 1.8.7-connsub	Completion Time Change (%)
read_large.rb	100	21.5054344	17.8400463	-17.04
eval.rb	1000000	11.8469212	11.3008363	-4.61
dirp.rb	10000	5.4305368	5.3851004	-0.84
app_tak.rb	9	24.5429233	24.3950791	-0.6
app_tak.rb	8	9.3381135	9.3064831	-0.34
so_ackermann.rb	7	1.8677968	1.8617144	-0.33
app_tarai.rb	3	11.1119703	11.0759606	-0.32
app_tak.rb	7	3.2231326	3.2173925	-0.18
lucas_lehmer.rb	19937	321.9907925	321.8465024	-0.04
simple_connect.rb	1	5.0470602	5.0452239	-0.04
lucas_lehmer.rb	9941	41.2846923	41.2932507	0.02
simple_connect.rb	100	5.0853819	5.0869957	0.03
simple_server.rb	1	5.0434871	5.0451347	0.03
simple_server.rb	100	5.0484228	5.0504225	0.04
lucas_lehmer.rb	11213	59.0862135	59.1315689	0.08
lucas_lehmer.rb	9689	38.2324514	38.2651066	0.09
pi.rb	10000	9.4993914	9.5098154	0.11
simple_connect.rb	500	5.2545103	5.2652134	0.2
app_tarai.rb	4	13.3509637	13.4174644	0.5
app_fib.rb	30	4.5319245	4.5653849	0.74
app_fib.rb	35	50.15053	50.6084618	0.91
regex_dna.rb	20	13.3037538	13.4352638	0.99
app_tarai.rb	5	16.1093574	16.3034291	1.2
mbari_bogus2.rb	1	1.5213026	1.546515	1.66
count_shared_thread.rb	1	0.2452542	0.2533421	3.3
count_shared_thread.rb	4	0.2446285	0.2527832	3.33
count_shared_thread.rb	16	0.2454466	0.2538465	3.42
count_multithreaded.rb	4	0.0981475	0.1015154	3.43
count_multithreaded.rb	8	0.1962461	0.2031188	3.5
count_multithreaded.rb	2	0.0491283	0.0508628	3.53
count_multithreaded.rb	1	0.0246464	0.0255306	3.59
so_exception.rb	500000	36.9332461	38.384544	3.93
count_shared_thread.rb	2	0.2438142	0.2535413	3.99
count_multithreaded.rb	16	0.3945088	0.4118931	4.41
count_shared_thread.rb	8	0.2444724	0.255249	4.41
list.rb	10000	15.1278532	15.9059888	5.14
quicksort.rb	1	14.9775395	15.8269178	5.67
simple_server.rb	100000	9.9024801	10.472057	5.75
mbari_bogus1.rb	1	0.8884958	0.9437024	6.21
reverse_compliment.rb	1	20.3273011	21.7283162	6.89
socket_transfer_1mb.rb	10000	6.4084512	6.8887912	7.5

socket_transfer_1mb.rb	1000000	6.425749	6.9157005	7.62
so_count_words.rb	100	10.7745652	11.9730053	11.12
gc_array.rb	1	111.3326223	124.7799331	12.08
open_many_files.rb	50000	1.3877207	1.555942	12.12
pi.rb	1000	0.1189384	0.1336277	12.35
write_large.rb	100	1.5902022	1.7880267	12.44
nsieve.rb	9	56.8482811	64.4837949	13.43
so_sieve.rb	4000	201.2101978	233.3200739	15.96
sum_file.rb	100	48.3162139	57.5910469	19.2
knucleotide.rb	1	5.912883	7.2048609	21.85
word_anagrams.rb	1	30.7776168	37.8031013	22.83
observ.rb	100000	3.107488	3.9006581	25.52
list.rb	1000	0.1716147	0.2163399	26.06
cal.rb	500	7.1927796	9.1665633	27.44
so_object.rb	1500000	15.6182469	20.1241125	28.85
so_object.rb	1000000	10.3666143	13.4047793	29.31
so_object.rb	500000	5.1654646	6.6961243	29.63
fannkuch.rb	10	343.9747739	466.5089542	35.62
fannkuch.rb	8	2.4506278	3.3465067	36.56
fannkuch.rb	6	0.0271181	0.0371088	36.84
gc_mb.rb	3000000	10.6502768	14.6588186	37.64
nsieve_bits.rb	8	70.0853772	97.2319317	38.73
gc_mb.rb	1000000	3.4374878	4.7952594	39.5
gc_mb.rb	500000	1.6160679	2.2919708	41.82
meteor_contest.rb	1	113.9658035	161.7425033	41.92
binary_trees.rb	1	171.4013777	258.2266448	50.66
primes.rb	3000	16.6459297	25.1207017	50.91
app_pentomino.rb	1	281.3741009	433.944056	54.22
so_array.rb	9000	20.1325928	33.2439908	65.13
gc_string.rb	1	23.6143011	39.142657	65.76
spectral_norm.rb	100	2.6598786	4.4913595	68.86
app_mandelbrot.rb	1	6.8674468	11.809018	71.96
fasta.rb	1000000	104.9711296	196.6912696	87.38
monte_carlo_pi.rb	10000000	47.7505442	90.9536009	90.48
so_matrix.rb	60	5.7104449	11.796188	106.57
nbody.rb	100000	28.1060031	58.0879835	106.67
mergesort_hongli.rb	3000	14.7900841	31.7051712	114.37
mandelbrot.rb	1	191.3900902	410.3141781	114.39
fractal.rb	5	17.3982213	37.5277137	115.7
mergesort.rb	1	6.9734991	15.2878785	119.23
partial_sums.rb	2500000	67.0803714	147.0711519	119.25
so_lists.rb	1000	28.4659886	75.0532411	163.66
so_lists_small.rb	1000	5.6987926	15.0316634	163.77

Appendix B: Ruby 1.8.7 ConnSub Patch

```
diff -rupN /usr/local/rvm/src/ruby-1.8.7-p174//eval.c ruby-1.8.7-p174-connsub//eval.c
--- /usr/local/rvm/src/ruby-1.8.7-p174//eval.c
+++ ruby-1.8.7-p174-connsub//eval.c
@@ -3500,6 +3501,38 @@ rb_eval(self, n)

      ruby_current_node = node;
      SET_CURRENT_SOURCE();
+
+     VALUE o_itype = TYPE(recv);
+     if(recv != self && o_itype != T_FIXNUM && o_itype != T_NIL && o_itype != T_FALSE &&
    →o_itype != T_TRUE && o_itype != T_UNDEF && o_itype != T_SYMBOL) {
+       ID cs_det = rb_intern("cs_det");
+       if(rb_obj_respond_to(recv, cs_det, Qtrue)) {
+         if(o_itype == T_DATA) {
+           rb_raise(rb_eTypeError, "%s has an internal type of T_DATA and cannot be 
    →Connotationally Subtyped.", rb_class2name(CLASS_OF(recv)));
+         }
+
+         ID cs_klass = rb_funcall(recv, cs_det, 0);
+         ID old_klass = rb_class_real(CLASS_OF(recv));
+         if(TYPE(cs_klass) != T_CLASS) {
+           rb_raise(rb_eTypeError, "%s::cs_det must return an instance of type Class",
    →rb_class2name(CLASS_OF(recv)));
+         }
+         if(cs_klass != old_klass) {
+           if(o_itype != TYPE(rb_obj_alloc(cs_klass))) {
+             rb_raise(rb_eTypeError, "Internal types of %s and %s not compatible for 
    →Connotational Subtyping.", rb_class2name(old_klass), rb_class2name(cs_klass));
+           }
+
+           if(FL_TEST(CLASS_OF(recv), FL_SINGLETON)) {
+             RCLASS(RBASIC(recv)->klass)->super = cs_klass;
+           } else {
+             RBASIC(recv)->klass = cs_klass;
+           }
+
+           ID cs_change_class = rb_intern("cs_class_changed");
+           if(rb_obj_respond_to(recv, cs_change_class, Qtrue)) {
+             rb_funcall(recv, cs_change_class, 1, old_klass);
+           }
+         }
+       }
+     }
      result = rb_call(CLASS_OF(recv),recv,node->nd_mid,argc,argv,0,self);
    }
    break;
diff -rupN /usr/local/rvm/src/ruby-1.8.7-p174//version.c ruby-1.8.7-p174-connsub//version.c
--- /usr/local/rvm/src/ruby-1.8.7-p174//version.c   2008-05-31 09:37:06.000000000 -0400
+++ ruby-1.8.7-p174-connsub//version.c   2012-04-30 16:20:28.539500956 -0400
@@ -39,7 +39,7 @@ Init_version()
    rb_define_global_const("RUBY_PLATFORM", p);
    rb_define_global_const("RUBY_PATCHLEVEL", INT2FIX(RUBY_PATCHLEVEL));

-   snprintf(description, sizeof(description), "ruby %s (%s %s %d) [%s]",
+   snprintf(description, sizeof(description), "ruby %s (%s %s %d connsub) [%s]",
         RUBY_VERSION, RUBY_RELEASE_DATE, RUBY_RELEASE_STR,
         RUBY_RELEASE_NUM, RUBY_PLATFORM);
    ruby_description = description;
```

VITA

IAN DILLON

Education:	B.S. Computer Science, East Tennessee State Univerity, Johnson City, Tennessee 2004
	M.S. Computer Science (conc. Applied Computer Science), East Tennessee State University, Johnson City, Tennessee 2012
Honors and Awards:	Upsilon Pi Epsilon Honors Society