5-1-2011

Static Analysis for Ruby in the Presence of Gradual Typing

Michael Edgar
Dartmouth College

STATIC ANALYSIS FOR RUBY IN THE PRESENCE OF GRADUAL TYPING

MICHAEL JOSEPH EDGAR

Department of Computer Science

Dartmouth Computer Science Technical Report TR2011-686

William McKeeman, Ph.D.
Thesis Advisor

STATIC ANALYSIS FOR RUBY IN THE PRESENCE OF GRADUAL TYPING

by

MICHAEL JOSEPH EDGAR

THESIS

Presented to the Faculty

in Partial Fulfillment

of the Requirements

for the Degree of

BACHELOR OF SCIENCE

Department of Computer Science

DARTMOUTH COLLEGE

May 2011

Abstract

Dynamic languages provide new challenges to traditional static analysis techniques, leaving most errors to be detected at runtime and making many properties of code difficult to infer. Ruby code usually takes advantage of both dynamic typing and metaprogramming to produce elegant yet difficult-to-analyze programs. Function *eval*() and its variants, which usually foil static analysis, are used frequently as a primitive runtime macro system. The goal of this thesis is to answer the question:

> *What useful information about real-world Ruby programs can be determined statically with a high degree of accuracy?*

Two observations lead to a number of statically-discoverable errors and properties in parseable Ruby programs. The first is that many interesting properties of a program can be discovered through traditional static analysis techniques despite the presence of dynamic typing. The second is that most metaprogramming occurs when the program files are loaded and not during the execution of the "main program."

Traditional techniques, such as flow analysis and Static Single Assignment transformations aid extraction of program invariants, including both explicitly programmed constants and those implicitly defined by Ruby's semantics. A meaningful, well-defined distinction between *load time* and *run time* in Ruby is developed and addresses the second observation. This distinction allows us to statically discern properties of a Ruby program despite many idioms that require dynamic evaluation of code. Lastly, gradual typing through optional annotations improves the quality of error discovery and other statically-inferred properties.

Table of Contents

Appendix

Chapter 1

Introduction

1.1 Static Analysis

When creating a program in a Turing-complete programming language, the programmer often wishes to determine certain properties of the code as written. Depending on the language in use, some properties are discovered with simple algorithms, such as the type of a variable in C. Some properties, such as termination, are undecidable in the general case [Tur37], yet even automated proof of termination has seen success with Microsoft's TERMINATOR project [CPR06, ACPR]. Errors in a program are the most common property we wish to discover, though there exist additional properties that are tractable to determine statically that provide meaningful information about a program. Java, during compilation, determines precisely which exceptions a method may raise during its execution, and ensures the programmer has annotated a certain subset of those exceptions (so-called *checked exceptions*) in the method definition [JBGG96].

1.2 Dynamic Typing

Dynamically-typed languages introduce challenges to static analysis. In such languages, the types of variables, such as the arguments to a procedure, may not be known until runtime. The type of a variable can change within a loop or even based on user input. While statically-typed languages provide constructs such as subtype polymorphism both for code re-use and for dynamic binding, dynamically-typed languages have implicit parametric polymorphism: variables are assumed to have some correct type for the operations performed on them, and in the general case, only attempting the operation at runtime can determine the validity of that assumption. Type errors are thus difficult to discover statically. Additionally, if function resolution is allowed to depend on the types of the arguments (including the receiver with common object-oriented syntax), as is common in such languages, the program flow also depends on the runtime types of the arguments.

1.3 Ruby's Dynamic Capabilities

Ruby is dynamically typed, and it allows nearly all aspects of the running program to change at runtime. *(NB: I assume for the purposes of this paper that the reader is familiar with, or can readily learn, common aspects of Ruby and its dynamic features.)* For example, Ruby provides constant identifiers as a language feature and disallows setting them in method bodies. Implementations always generate a runtime warning upon rebinding the constant identifier, while executing the rebinding. This warning can be avoided entirely: one can use `remove_const` to delete the constant, and `const_set` to rebind it, all without generating a warning.

As an object-oriented language in the Smalltalk tradition, every value in Ruby is an object: an instance of a class. Classes can be created or removed at any time, as can instance methods. When a method is called that an object cannot respond to, its `method_missing` method is called; programmers use this callback to implement delegation patterns and to generate methods at runtime. For example, the popular Ruby on Rails web framework can generate database-searching methods on the fly when a method call is performed matching the regular expression `find_all_by_\w+`.

Ruby's object model includes a *metaclass* or *eigenclass* for every object. Every object is an instance of its metaclass inheriting from the object's traditional class. This hierarchy allows the programmer to add or remove methods to individual objects at runtime.[1]

[1] Exception: small integer constants (less than $2^3$0), which are commonly represented internally as tagged pointers, do not have mutable metaclasses.

All of these dynamic features challenge static analysis. Ruby programmers commonly make use of these dynamic features, forcing any successful static analysis to be prepared for their implications.

1.4 Syntax Aids Analysis Greatly

Some highly dynamic languages, such as the Lisp family of languages, have little syntax. However, most Lisp dialects have numeric literals, strings, $\#t$ to represent truth, and so on, and an analyzer can determine the types of those literals. Some dialects have special-purpose macros such as `defun` for function definition; an analyzer might use knowledge of these macros to assume that the name being bound will refer to a function unless rebound. Beyond these examples, dataflow analysis must take over if an analysis intends to determine types or other properties of a given binding.

Ruby, on the other hand, uses a complicated but structured syntax to simplify common programming tasks. Many constructs overlap in functionality: there exist four conditional syntaxes, two ways to create classes, and so on. Special syntax exists for defining methods, creating classes and modules, passing a single anonymous procedure to a method, invoking the specially passed procedure, interacting with an object's metaclass, receiving a variable number of arguments, and so on. It has literal syntax for strings (with and without arbitrary code interpolation), regular expressions (also including interpolation), integers of arbitrary precision, floating point values, arrays, hash tables, ranges, symbols, and procedures.

These syntactic forms allow an analyzer to determine wide-ranging information about a program without any dataflow analysis. The information cannot always be relied on with exact certainty, due to the dynamic features described previously (Listing 1.1). A compiler would be required to account for such dynamism precisely. In designing an analyzer and linter,[2] however, I can choose to not fight this battle.

[2] I use "linter" to mean a static analysis tool that warns against errors, questionable practices, and poor style choices.

Listing 1.1: Dynamic class modifications

```
class A < String
end
if rand > 0.5
  Object.class_eval { remove_const :A }
  A = Class.new(Integer)
end
```

1.5 Emulating Load-time Metaprogramming

As noted earlier, metaprogramming is used commonly in Ruby to reduce code duplication and to provide convenient interfaces to libraries. Methods are generated primarily through `Module#define_method` and `Kernel#{class_,module_,instance_,`ϵ`}eval`. Detecting calls to these methods statically is necessary to have a full knowledge of which methods exist on which types.

As an example, `Module#attr{_reader,_writer,_accessor,`ϵ`}`, which generate getters and setters for instance variables, are simple methods generating new methods based on their arguments. Since they are used to generate getters and setters for instance variables, they are primarily called with a constant string or symbol as parameters. The `attr` family of methods are pure in that they use only their arguments as input; when they are called with constant arguments their effects can be predicted statically. They are also most often called in class definitions, which are typically executed exactly once, when the containing file is loaded into the interpreter. Indeed, the RDoc documentation tool accompanying Ruby attempts to discover direct calls to these methods (and others) by inspecting each identifier token that does not occur inside a method definition. It then uses this information to create documentation for the reader and writer methods that will be generated.

This technique is insufficient once one ventures beyond these four methods. If one writes new methods that generate methods, or re-implements the above methods with a different name, the tool will fail to predict the side-effects, as it has hard-coded this analysis.

If a tool can perform static method resolution, and can emulate a predictable subset of Ruby, it can emulate load-time method calls that it resolves precisely and that have constant arguments. While emulating, it recursively analyzes strings passed to calls to `eval`, and intercepts calls to `define_method`. This technique can discover properties of a subset of dynamically-created methods statically. Intuition leads us to think this subset is an important one, and the limited research into Ruby has supported this intuition [FAF09]. The size of this subset, relative to the set of all dynamically-created methods, should be measured to determine the efficacy of top-level emulation.

1.6 Static Single Assignment

A significant issue in analysis is determining, given a use of a variable (temporary or explicit) in a program, where that variable was defined. Given multiple assignments to the same variable along disjoint paths, possibly including assignments inside loops, answering this question efficiently is nontrivial.

One technique commonly used is transforming the control-flow graph (CFG) of the program into *Static Single Assignment Form* (SSA). This form modifies the original CFG to satisfy the following constraints [CFR+89]:

1. Each programmer-specified use of a variable is reached by exactly one assignment to that variable.
2. The program contains ϕ-functions, that distinguish between values of variables transmitted on distinct incoming control flow edges.

ϕ-functions are commonly called *ϕ-nodes* as well, a convention I will use. If a program is written such that each variable is assigned to only once, each use of that variable can be traced precisely to its definition. While a full discussion of implementing the SSA transformation is not within the scope of this thesis, it is worth noting that it can be implemented efficiently: the algorithm chosen for implementation here is linear for non-pathological programs, regardless of language [Mor98].

1.7 Constant Propagation

Constant Propagation (CP) is the dataflow problem of determining which variables and values are constant in a program. Common among optimizing compilers, constant propagation allows a static analyzer to prune branches conditioned on proven constants, which improves the accuracy and efficiency of all other analyses that consider the control flow of a program. In general, CP is undecidable [Hec77], yet there exist sets of constants that are decidable, as well as polynomial-time algorithms to find instances of those constants in programs.

An important example illustrating the need for CP in static analysis of Ruby stems from Ruby's history of breaking backward compatibility: library authors often write separate code based on the variable `RUBY_VERSION` to either resolve incompatible differences or to backport new features (see Listing 1.2).

Listing 1.2: Conditionally Backporting a String Method

```
if RUBY_VERSION < "1.9"
  class String
    def start_with?(prefix)
      self[0...prefix.size] == prefix
    end
  end
end
```

Ruby's rich syntax for literals (including classes, modules, and procedures) mean the primary structure of a Ruby program is actually a series of instructions involving constants. Inferring the results of these instructions statically not only presents the opportunity for optimization but also reveals the object-oriented structure of the program. A CFG-based analysis with constant propagation can discover classes, modules, and methods declared statically using literal syntaxes, and could potentially discover such constructs when created dynamically, without literal syntaxes, but with constant arguments (see Listing 1.3: discovering the `start_with?` method requires some form of constant propagation through local variables).

Listing 1.3: Backporting `start_with?` using Dynamic Code w/ Constants

```
if RUBY_VERSION < "1.9"
  klass = String
  klass.class_eval do
    new_method = :start_with?
    define_method new_method do |prefix|
      self[0...prefix.size] == prefix
    end
  end
end
```

Constant Propagation is typically framed as assigning a value to each variable drawn from the set $C \cup \{\top, \bot\}$, where C is the set of all constants, $\top$ is read as “undefined”, and $\bot$ is read as “varying.” All variables in a function are initially considered to be $\top$, with formal parameters to a function considered $\bot$ (unless interprocedural analysis indicates the parameter is always the same constant—we disregard this opportunity). CP terminates when it has assigned every variable either a constant from C or $\bot$. CP algorithms differ by which constants are proven and in their running times.

In particular, Wegman and Zadeck’s *Conditional Sparse Simple* (CSS) algorithm finds all **simple constants** in a program, as well as constants that are constant when ignoring untaken branches on constants [WZ91]. A **simple constant** is a value that can be proven to be constant without assumptions about the path taken through the program, and by only maintaining one value along each path. CSS combines the single-assignment portion of SSA with earlier work in pruning branches based on constants. Intuitively, SSA simplifies CP by ensuring that only one assignment occurs to each variable, though additional rules are required for ϕ-nodes. Wegman and Zadeck’s algorithm runs in time linear with the number of variables in the program [WZ91]. Due to the SSA transformation, the number of variables in the CFG may be superlinear with respect to the size of the input program; drastic increases in variable count during SSA transformation are uncommon outside of pathological input code.

1.8 Ruby Blocks: A Prime Analysis Target

A method in a Ruby program may always receive a *block* as an argument. This block is simply a closure, created by the caller, using one of two syntaxes. The callee may invoke this closure directly by using the `yield` keyword or by capturing the block as an explicit argument. It may also forward the closure on to another method. Widespread use of blocks is a defining characteristic of idiomatic Ruby code.

Some methods require a block, such as the benchmarking method `Benchmark.measure`. Calling this method without a block argument results in a `LocalJumpError` exception being raised, which occurs when the `yield` keyword is executed without a block argument.

Some methods do not require a block, but will use a block if provided one. If a programmer uses `File.open` without a block, the requested file is opened and the file handle is returned. If `File.open` is called with a block, then the file is opened, the block is run with the open file handle as an argument, and then the file is closed. This technique, inherited from Lisp (`with-open-file`), is a common idiom for handling resource management.

Naturally, some methods will never use a block. One may write `Dir.pwd { |x| puts x }`, but the block will not be run. More importantly, it will be *silently ignored.*

How a method uses block arguments is a defining part of its API, yet conformance is at best enforced through runtime errors, and at worst ignored. Statically inferring this portion of the API implies the potential for static enforcement.

1.8.1 Block Use Mechanics are Diverse

In order to implement an efficient block-use analysis, we must have a full understanding of how a callee can determine whether a block argument is available, how it can attempt to invoke a block, how such attempts can fail, and how it can obtain a reference to the block object (if any).

1.8.2 Determining the Presence of a Block

Without obtaining a reference to the block, there are two ways to determine whether a block argument is available in the current stack frame:

- The method `Kernel#block_given?` returns true if there is a block argument available, and false otherwise. This is the most common approach.
- The `defined?` operator takes an expression as an argument and returns true if the expression is "defined"; `defined?(yield)` is considered "defined" if there is a block argument. Thus `defined?(yield)` is equivalent to `Kernel#block_given?`. This is a rarely used technique.

Neither of these methods can be implemented in pure Ruby. Thus, we need only identify invocations of the `block_given?` method. While `defined?` in general is not fully supported by our present implementation, `defined?(yield)` is lowered during CFG construction to equivalent branch and assignment instructions. Thus, we can detect whether the programmer is using either technique purely through analysis local to the method in question.

1.8.3 Invoking a Block

Without capturing the block as a `Proc` object, there is one way to invoke the block: the `yield` keyword, which accepts an argument list (possibly varying). If executed when a block is present, the block is invoked with the given arguments. When the block terminates, execution continues after the `yield`. When executed when a block is *not* present, a `LocalJumpError` exception is created and raised. This exception can be caught like any other, and catching that exception introduces a technique to create methods that optionally use their block.

`yield` is tied to the method body and frame in which it occurs: `yield` cannot invoke a block argument passed to a different stack frame[3]. Thus, `yield` can be implemented more

[3] `yield`, combined with `eval()`, can be used to invoke a block from a different scope: as we discuss later, `Binding` objects allow the programmer to violate this encapsulation. We propose a solution to the resulting complications later.

efficiently and analyzed purely locally.

1.8.4 Capturing a Reference to a Block

The previous sections discuss invoking a block without a reference to it. A method may obtain a reference to the block in two ways:

- Prefixing the final argument in the method's formal argument list with an ampersand (&). Upon method entry, the named argument will contain a reference to the block, if any.
- `Proc.new`, when called with no normal arguments and no block argument, will return a reference to the block, if any.

When a block is not present, both of these techniques will give `nil`, and when a block is present, they give the block as an instance of the `Proc` class. The `Proc` class provides several methods which invoke the closure, such as `#call`, `#[]`, and `#===`.

When we later characterize block use, we will require a notion of "attempting to invoke the block argument;" Keyword `yield` is one way. If a reference to the block is exposed in such a way, this notion manifests as attempting to capture the block as a reference, not checking if it is `nil`, and attempting to invoke it by one of the above methods. Doing so results not in a `LocalJumpError` as above, but instead a `NoMethodError`.

Since the closure obtained through either technique is an object reference, it may be passed to other procedures, and can escape the frame's execution entirely if stored on the heap. This presents a greater challenge for analyzing block use.

Chapter 2

Prior Art

Ruby has been the subject of research in both static and dynamic analysis in recent years, primarily focused on type inference and discovering type errors. The prior work that we build upon, both directly (through implementation) and intellectually, is listed below.

2.1 Ripper

Ripper is a SAX-style Ruby parser library provided with Ruby 1.9 which can also convert Ruby code into a simple, array-based abstract syntax tree. The library is constructed by embedding special instructions into the same `bison` grammar file used by the language implementation. The resulting trees still retain some features of the concrete syntax tree,[1] and only specific node types carry line and column information. It also has the flaw of parsing local variable references and no-argument, no-receiver, no-parentheses method calls to the same node type. Distinguishing the two after the fact is not difficult, but as local variables are created by the parser, Ripper should make this distinction. Finally, it is still considered "experimental" by its authors and is not bug free; three bugs were discovered in which Ripper discarded parse trees for which it lacked parsing rules. They were patched by the authors and applied to the main Ruby repository [Edg11b, Edg11a].

While competing AST formats and libraries exist, such as `ruby_parser` [Dav11], only Ripper is officially supported. Additionally, `ruby_parser` is developed primarily by one individual, and contrasted with Ripper, at times performs *too much* semantic interpretation

[1] For example, parsing "foo.bar" and "foo::bar"—which are equivalent—results in different parse trees, which include the separator token as a node.

of the tree.[2]

The input to the analyzer is thus a sequence of ASTs conforming to the format output by Ripper.

2.2 Type Inference: Diamondback Ruby

Diamondback Ruby is the most prominent research project attempting to bring the benefits of static analysis to Ruby. Focused on static typing, Diamondback Ruby (DRuby) combines type annotations, static analysis and dynamic analysis to type wide ranges of Ruby code [FAFH09b]. DRuby is primarily driven by an OCaml parser and analyzer that produces a unified AST in what the authors call Ruby Intermediate Language (RIL), a high-level intermediate representation that reduces some of the implicit complexities of Ruby code while enabling source-to-source transformations [FAFH09c]. It uses temporary variables to aid dataflow analysis, but is primarily intended to reduce Ruby code to a simplified, less flexible language with a mapping back to Ruby code. While Ruby offers four different conditional syntaxes,[3] RIL transforms all of these syntaxes to the traditional `if .. else ... end` syntax.

DRuby also warns the user when a function is called with incorrect arity, or symbols fail to resolve, but it primarily focuses on typing [FAFH09a]. It uses a custom constraint-based type solver to implement its type system.

Static analysis of RIL failed to produce significant results on real-world code, so the DRuby authors augmented it to allow for dynamic profiling and added more "refactorings" from untypable code to typable code [FAFH09b]. Profile-guided dynamic analysis saw greater success and led to the creation of DRails [ACF09], which extends DRuby to analyze web applications written using the Ruby on Rails web framework. Ruby on Rails is known

[2] For example, upon parsing the `rescue A => err_name` construct, ruby_parser discards `err_name` as part of the syntax tree and instead inserts a node in the rescue body representing `err_name = $!`. For a static analyzer, having the parser perform this level of semantic interpretation is unacceptable.

[3] if and unless, with guard versions of both.

for taking full advantage of highly dynamic and unpredictable Ruby features; successful analysis of such programs represents significant success in Ruby analysis.

Unfortunately, mainstream developers have not adopted DRuby. In part, the use of OCaml instead of Ruby as the implementation language, despite its performance benefits, limits DRuby's ability to integrate into the Ruby ecosystem. Its focus on typing and its annotation syntax may also contribute to its failure to break through: Ruby is often used for rapid development and prototyping, and a lack of static typing is often considered a strength in this regard.

2.3 Type Inference: Cartesian Product Algorithm

A primary difficulty of type inference in dynamically-typed languages stems from the implicit parametric polymorphism of all code: a method called with arguments of different types may result in new types and new polymorphic calls in the method body. In the presence of nested scopes, common in many dynamically-typed languages, the quantity of type information to infer can grow exponentially with respect to the size of the input program, though not all code will exhibit such complex polymorphic behavior.

The Cartesian Product Algorithm (CPA) developed by Ole Agesen [**(year?)**] is an adaptive, single-pass algorithm which tracks type propagation throughout a dynamically-typed program. It was originally written for the Self and Smalltalk programming languages, which share many of Ruby's dynamic capabilities. The algorithm creates a copy of each method body for each possible combination of types for its arguments, and simulates the flow of typed data in the method with the given combinations. As the set of argument types grows, the method is re-simulated with new possible type combinations. This has the effect of precisely inferring parametric polymorphism in a single pass through the program.

CPA has also been implemented for the Python language in Dufour's Shed Skin optimizing Python-to-C++ compiler [Duf], where it is combined with Pleyvak's algorithm for inferring data polymorphism [PC94]. Python shares many of Ruby's dynamic capabilities, though Python programmers rarely rely on *eval*() or dynamic method definition, unlike common Ruby practice.

2.4 Type Inference: Ecstatic

Ecstatic is the title of Kristian Kristensen's master's thesis from 2007 which applies type inference to vanilla Ruby code [Kri07]. Ecstatic implements the CPA algorithm described earlier while simulating a small set of Ruby constructs. It effectively inferred parametric polymorphism in real-world benchmark scripts and Ruby libraries.

However, it was limited in its support. Recursion was unsupported, and only the most straightforward syntax for using a block was supported. While valuable as a proof-of-concept and excellent evidence that Ruby programs can be type-inferred by CPA, Kristensen acknowledges several specific areas where accuracy was limited.

Notably, like the RIL used by DRuby, Ecstatic uses the AST of the Ruby program, and not a lowered representation.

2.5 YARD

YARD is a documentation tool for Ruby that is designed to be modular, handle metaprogramming, and allow arbitrary, searchable documentation tags attached to methods, parameters, classes, and so on. In this respect it is successful: it correctly analyzes all the cases that RDoc, the standard documentation tool, can analyze, and several more. When run using Ruby 1.9, it uses the actual parse tree from Ripper (see 2.1) instead of relying on a mere token stream as RDoc does. It does go beyond RDoc's abilities: for example, the developer provides a plugin that analyzes and documents test cases. This author has also written a patch that supports the detection of classes created with `Struct.new`.

However, in discovering properties of the program in question (primarily classes and methods), YARD too suffers from the fact that each new dynamic feature must be explicitly supported through additional analysis code. If one defines a custom `attr_accessor`, one must write a YARD plugin that supports the new definition.

Chapter 3

Construction and Low-Level Analysis of a Ruby CFG

The following chapter details the construction and analysis of a low-level control flow graph for Ruby code. The early sections of this chapter overlap with prior work (see [FAFH09c]), yet is provided for completeness.

3.1 Resolving Scopes

Before any specific analysis is performed, the static scopes of the Ruby program are computed, with as many bindings created and resolved as possible. Ruby has both open and closed scopes for local variables (open scopes can reference variables in enclosing scopes; closed scopes cannot). These bindings can be resolved statically. Local variable bindings are created by Ruby "upon use", which can interact oddly with various language constructs.[1] Constants use open scopes yet can be referenced directly from outside the scope in which they were declared. Class variables, a seldom-used, often-confusing feature of Ruby distinct from static variables, use lexical scope for resolution, and so too are statically resolvable. Instance variables rely on the dynamic value of `self` to resolve and thus cannot necessarily be resolved statically. Global variables are trivially resolved.

[1] Example: variables defined by a rescue handler definition (`rescue Exception => err`) are available in the ensure block and after the handler has executed.

3.2 Building the Control Flow Graph (CFG)

Next, we construct a full control-flow graph of Ruby code, in a lowered representation. Several cues were taken from Morgan's *Building an optimizing compiler* [Mor98]. Since the Ripper (2.1) AST defines roughly 80 different node types, the walking procedures are quite large, though nearly all node types are simply translated into a low-level intermediate representation (IR) using traditional means.

3.2.1 CFG Intermediate Representation

Each instruction generated receives a reference to the AST node currently being processed; this is done to simplify the process of relating information about instructions to the source code that generated it. In order to simplify analysis of the IR, we use as few instruction types as possible. The instruction types used are:

1. Local Assignment
2. Method Call: fixed or variable arity
3. Invoke Super: fixed or variable arity
4. Return
5. Raise
6. Jump
7. Branch

Typical Ruby implementations compile Ruby source to a bytecode with far more opcodes than we use. For example, the YARV 1.9 bytecode compiler uses fresh opcodes for class definitions, basic operators, and common operations on core library classes.

We implement some operations as method calls on a hidden, implementation-level class (much like Ruby 1.9's `FrozenVM` object), but many operations, such as class definitions, are lowered entirely to sequences of assignments, branches, and calls to public methods. This has the effect of explicitly encoding the implicit semantics of Ruby code in a low-level form.

The downside is that operations with complex semantics require much IR code: for example, declaring an empty class at the top level requires 8 basic blocks and 30 instructions.

3.2.2 Handling Blocks (Closures)

Ruby's blocks raise an interesting question in the design of the CFG. Ruby's blocks are quite frequently immediately executed by the callee using the `yield` keyword, any number of times. Blocks may also be captured and stored for later execution by specifying the block as an explicit argument in the callee's definition. It is then converted to an object of class `Proc` and may be invoked at any time, again any number of times.

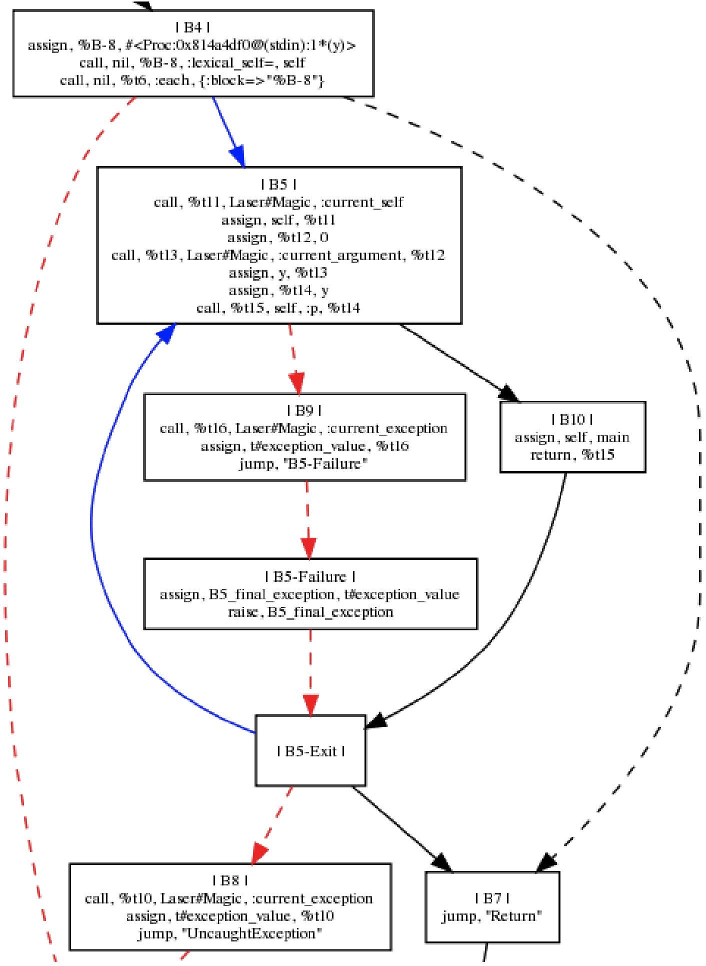
Thus there are edges from a block-bearing method call to the block used, with the block exiting via the same `raise` and `return` instructions that exit a method. There are edges at the end of the block's CFG indicating the return to the callee as well as to re-invoke the block. (See Figure 3.1)

This construction is sufficient for inline blocks, but does not address the potential edges from other blocks captured as values. Method calls that may invoke a closure not passed as an inline block must also have edges entering and returning from the closure's CFG. We do not implement this, but recommend using a simple alias analysis to first create conservative edges into local closure graphs, and then allow later analysis to remove extraneous edges. This is likely to create a very dense graph at first, analogous to the CFGs produced by computed `goto` constructs in C/Fortran.

3.2.3 Fake Edges

Many of the standard algorithms that work on Control Flow Graphs expect that the designated *Enter* block dominate each block and that the *Exit* block post-dominate each block. In order to maintain this invariant, "fake" edges are inserted under specific conditions. Fake edges, described by Morgan, are placeholder edges that only exist to preserve the graph structure. In the context of optimization, when an instruction is placed on a fake edge, the

Figure 3.1: A call with a block that prints its argument


| B4 |
assign, %B-8, #<Proc:0x814a4df0@(stdin):1*(y)>
call, nil, %B-8, :lexical_self=, self
call, nil, %t6, :each, {:block=>"%B-8"}
| B5 |
call, %t11, Laser#Magic, :current_self
assign, self, %t11
assign, %t12, 0
call, %t13, Laser#Magic, :current_argument, %t12
assign, y, %t13
assign, %t14, y
call, %t15, self, :p, %t14
| B9 |
call, %t16, Laser#Magic, :current_exception
assign, t#exception_value, %t16
jump, "B5-Failure"
| B10 |
assign, self, main
return, %t15
| B5-Failure |
assign, B5_final_exception, t#exception_value
raise, B5_final_exception
| B5-Exit |
| B8 |
call, %t10, Laser#Magic, :current_exception
assign, t#exception_value, %t10
jump, "UncaughtException"
| B7 |
jump, "Return"

instruction is simply ignored.

When an unconditional jump such as *return*, *next*, *break*, and so on are generated, a new basic block begins that has no predecessors: it is unreachable. A fake edge is inserted from the loop header block to this unreachable block.

When *redo* is executed in a loop context, it restarts the loop body while ignoring any loop header that may be present. This can create an infinite loop in the CFG during construction - before removing any edges. While this is almost surely an error on the programmer's part, we must add a fake edge or we cannot analyze the code properly using standard algorithms. We conservatively add a fake edge for all *redo* statements.

3.3 Top-Level Simulation

Any attempt to analyze a Ruby program requires discovering the classes, modules, and methods defined in the program. The Ecstatic project walks the AST with inference rules for certain node types, such as `class`, `module`, `def`, and so on [Kri07]. DRuby's static analyses use a similar approach [FAFH09b]. As these projects readily acknowledge, there is a limit to the success of this approach: metaprogramming (the Ecstatic paper calls it "dynamic programming") is employed commonly in Ruby, wherein methods are created to define methods, classes, include modules, and so on. Hard-coding rules for certain methods is not enough.

As all top-level code is executed to achieve its effect, including class definitions and method definitions, our approach is to simulate all top-level code that has a predictable effect. This includes global mutations, such as the setting and getting of global variables, the creation of classes (which creates new globally-accessible identifiers), defining new methods, and so on. In order to capture these effects (especially the creation of classes, modules, and methods) we intercept all calls to certain core Ruby methods, such as `Class.new` and `define_method`. We then implement these methods within a sandbox and return compatible results. Once we reach code which is unpredictable - a method call involving I/O, for example - we cease simulation and begin further analysis.

In effect, we have implemented a interpreter for a deterministic subset of Ruby. The need for this is clear: previous efforts (Ecstatic, RDoc, YARD) all work by simulating a much smaller subset of Ruby, and they fail to discover methods and classes when user code goes beyond that subset.

Performing full simulation introduces new challenges:

1. Top-level code may not terminate. While this is likely a major error on the user's part, it must not cause the analyzer to fail to terminate. As a safety measure, our analyzer terminates simulation after a threshold number of basic blocks have executed.
2. The analyzer requires a knowledge of "unpredictable" methods, such as `gets` or `rand`. We blacklist unpredictable methods that are written in C, and assume unpredictable Ruby code will use these methods. The correctness of this approach is unproven.

Despite these challenges, top-level simulation successfully identifies metaprogramming constructs that previously required manual intervention. As an example, the `attr` family of methods are supplied to the analyzer as user code, without special casing its behavior.

3.4 Static Single Assignment

Creating the Static Single Assignment form of a Control-Flow graph is a well-known, simple algorithm. However, it may require adaptation to the language in question, especially regarding the semantics of uninitialized variables. Ruby creates local variables when the parser first sees an assignment to them, and they are available to be read any point lexically after that assignment. However, that assignment could be conditional, and thus, not executed, leaving the variable in an uninitialized state. This has a well-defined meaning in Ruby: when read, an uninitialized variable will have the value `nil`. In the example listing 3.1, the program will either print “5”, if the branch is successful, or “0” (as `nil.to_i` = 0).

Listing 3.1: A potential use of an uninitialized variable

```
if rand > 0.5
  y = 5
end
puts y.to_i
```

On the path in the CFG where that variable is not written, its first use may occur before any assignments to the variable. This will confuse the straightforward SSA renaming algorithm, which expects to have created a path-local name for the variable before encountering a read from that variable. The stack of names will be empty along the uninitialized path.

When this occurs, we must insert an assignment to `nil` immediately before the variable is used (see Figure 3.4). This restores the property that all variables are written before they are read, explicitly encodes Ruby’s semantics in the CFG, and allows SSA renaming to continue. If the use is a normal instruction, we insert the assignment in the same

```
if x > 10
  y = 2
else
  a = y
end
```

Figure 3.2: Relevant source code

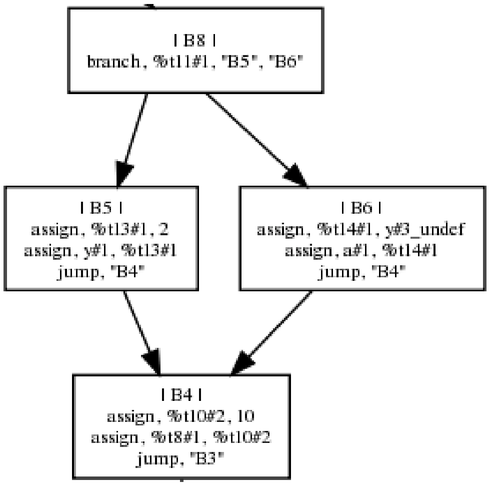


Figure 3.3: Before correction

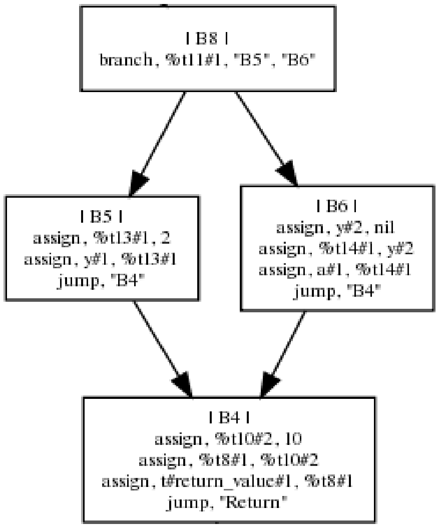


Figure 3.4: After correction

basic block, immediately preceding the use instruction. If the use is a ϕ-node, then it is "executed" immediately upon entering the block, so any new assignment must occur in a preceding block. We insert the assignment instruction on the preceding edge corresponding to the uninitialized argument to the ϕ-node (see Figure 3.7).

```
if x > 10
  y = 2
end
p y
```

Figure 3.5: Relevant source code

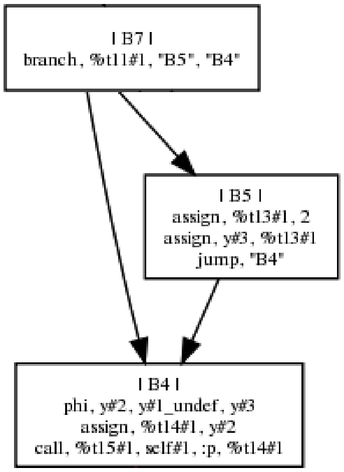


Figure 3.6: Before correction

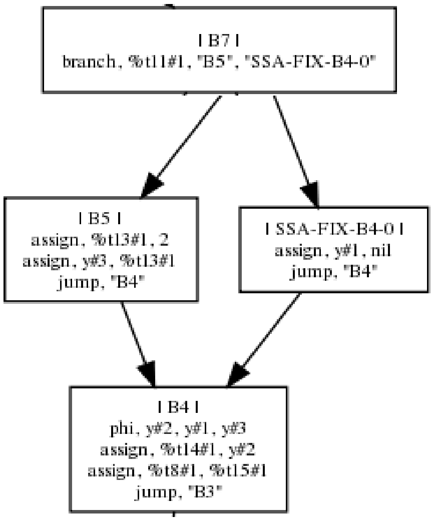


Figure 3.7: After correction

3.5 Type Inference: Cartesian Product Algorithm

The Cartesian Product Algorithm (CPA) is a single-pass algorithm for inferring the potential concrete types in a dynamically-typed program. CPA can be readily described as: ([Age95])

1. Instantiate type variables for all expressions
2. Initialize type variables for expressions with known pre-execution types (eg. string literals)
3. Identify constraints between type variables
4. Propagate known concrete types through these constraints

The Ecstatic project implements this at the AST level for a large subset of Ruby [Kri07]. They also discuss the issues with flow-sensitive versus flow-insensitive analyses, and implement a flow-insensitive analysis.

By reducing the AST to a SSA-form CFG with a minimal instruction set, much of the work of defining our CPA analysis is already complete. We implement the steps of (3.5) as follows:

1. **Instantiate type variables**: The CFG's temporary objects bear a publicly-manipulatable instance variable `inferred_type`, which defaults to $\varnothing$.
2. **Initialize type variables**: Constants appear in our CFG as raw Ruby objects (such as `nil` or small integers). Their type is determined by simply calling `.class`.
3. **Identify constraints**: Only four instructions create type constraints: assignment, method-call, super-call, and ϕ. All four have the same constraint: their target temporary's type depends on their operands' types.
4. **Propagate types**: Assignment instructions copy type from source to target, and ϕ-nodes assign the union type of their operands to the target temporary. Method-call and super-call instructions use the traditional CPA type calculation technique. When the type of a temporary changes, it changes at its definition instruction; all instructions using that temporary are then reconsidered.

The type propagation step, when applied to an SSA CFG, is nearly identical to the CSS constant propagation algorithm. Indeed, our implementation runs CPA simultaneously with CSS. Where CPA differs substantially from the CSS algorithm is in its complexity. CSS guarantees $O(V)$ runtime by only visiting each temporary's definition at most twice, changing a temporary t's inferred value from $\top$ to a constant C_t, then from C_t to $\bot$. However, CPA may require changing t's type $O(T)$ times, where T is the total number of types in the program.

Unlike Ecstatic, we achieve a flow-sensitive analysis by using an SSA CFG. The SSA renaming algorithm distinguishes between type variables on distinct paths, and ϕ-nodes join those types as necessary.

3.6 Detecting Purity

We will soon confront the issue of determining whether we may simulate a resolved method. We must only simulate *pure methods*. In an imperative language like Ruby, a *pure* method is one which:

1. uses only its actual arguments (including the receiver) to compute its result
2. does not overwrite memory that was initialized before the method began.

For the first condition, we provide an exception for external constants, named using Ruby's syntax for constants, which we have shown to only be assigned once. This is necessary to access common classes such as `String` and `Array`. A full purity analysis is nontrivial, especially given lack of static type information. We employ a simple metric. We conservatively denote a method impure if it:

1. explicitly writes to a non-local variable
2. explicitly reads a non-constant binding from outside the local scope
3. potentially calls an impure method via dynamic dispatch

One exception is made for `#initialize`, which is called during allocation of new objects: it may write instance variables, but may not read nonlocal variables.

Some methods are implemented in C, not in Ruby. This is especially important in the Ruby core library, where crucial classes such as `Hash` and `Fixnum` are implemented in C to improve performance. For these, we provide a comment-based annotation, which informs the analyzer that the method is pure. The annotation is not granular by overload, as no instance of a method was found whose purity changed based on arity or argument types.

3.7 Constant Propagation

Wagmen and Zadeck's CSS algorithm is attractive as it not only finds simple constants, but also prunes untaken branches. The algorithm makes no assumptions about the source language or the instructions present in the CFG; CSS simply distinguishes between operations "predictable" at compile-time versus "unpredictable" operations. Typical presentations of the algorithm are based on statically-typed languages such as C or FORTRAN, where the CFG contains distinct instructions for basic mathematic operations (+, ×, <<) and function calls. Special rules may be provided taking advantage of algebraic identities and these basic operations.

However, in Ruby, unary and binary operators are implemented as methods with full dynamic dispatch. In our CFG, there are extremely few fundamental instruction types. It becomes clear that the effectiveness of CSS for Ruby depends closely on what operations we can conservatively "predict". If we disregard method calls entirely as unpredictable, our implementation of CSS will not even prove that upon seeing $x_1 = call(3, +, 4)$, x_1 equals 7! As a result, if our CP algorithm will prove useful, we must make an effort to distinguish between methods we may safely predict, and those we cannot.

3.7.1 Constant Propagation: Method Calls

We consider a method call's components: the method name, the receiver object, and the arguments provided. If any argument or the receiver of the method call is $\perp$, then in general we cannot predict the result of the operation (algebraic identities violate this; we consider them shortly). Thus, any method call we might predict has a constant receiver. If the receiver is constant, then we know its concrete type, as is true for the arguments to the method call. Given the receiver's type and the method name provided, we consult the `LaserClass` corresponding to the receiver's type, and look up the method by name. Given successful resolution, we then consider the method call's arity and the concrete types of the arguments, and ensure the arguments comply with arity and any annotated type requirements.

Finally, we must decide whether we should attempt to simulate the resolved method. Two concerns arise: the method must be *pure*, and it may not halt [2]. We have already described our conservative purity analysis (see 3.6), but we note that if any constant argument is an impure procedure itself, then the method call itself will be impure if the impure procedure argument is called. We tentatively assume conservatively, without further analysis, that such a call is always impure and its result is $\perp$. Next, we turn to the issue of termination.

If the method in question has been implemented in C, and its purity inferred by annotation, then it is assumed to terminate. In this case, we simply call the method directly with the receiver and arguments; this is a trivial operation as the analyzer is written in Ruby itself.

If the method in question is implemented in Ruby, and we know it to be pure, we may simply instantiate the actual parameters and walk the CFG of the method for a capped number of steps, X. If the simulation executes a `return` instruction then the return value is constant. If the simulation executes a `raise` instruction then the method call always exits abnormally. If we execute X instructions without reaching the Exit node, we consider

[2] The method may also have guaranteed termination, but simply run longer than we wish during analysis.

the call unpredictable.

No matter how simulation is performed, if the direct call terminates normally, the return value is assigned to the target register of the call instruction in the caller CFG, and if the call has a CFG edge corresponding to an exception, that edge is removed. If it terminates abnormally, then the method call will always raise; we remove the edge from the CFG corresponding to the call's successful execution and assign $\top$ to the target register of the call.

3.7.2 Constant Propagation: Types and Branches

In Ruby, only `nil` and `false` are considered *false* in a boolean context. These two objects each have their own corresponding metaclasses, `NilClass` and `FalseClass`, respectively. Thus, by inferring types along with constants, we can prune branches predicated on $t = \perp$ if we know t's type to not include (or to only include) those classes. The normal CSS algorithm requires branches be predicated on a constant to prune their edges, as it is the *value* and not the *type* that traditionally controls branching.

3.7.3 Constant Propagation: Lambda Definitions

A crucial feature of common Ruby code is pervasive use of anonymous functions. As the CFG is built, when an anonymous function is discovered, its body's CFG is built, and an instance of `LaserProc` is created and assigned to a new temporary. That `LaserProc` instance is a small wrapper around:

1. references to the Enter and Exit blocks of the function's CFG
2. a description of the function's formal arguments
3. a reference to the register holding the lexical value of `self`

Since anonymous procedures are only introduced as a block argument to a method call, the method call then adds an edge from its basic block to the Enter block of the function's CFG (as described in 3.2.2).

When is a function itself constant? By itself, a function is an immutable value. However, bear in mind that in Ruby, anonymous functions are *closures*: a combination of a function, and its environment. The function may read and write variables from the environment that are not defined in the function body. A closure may or may not be constant due to this interaction.

Consider an anonymous function that reads a variable from its enclosing scope, and uses it in the computation of its result. If that variable's value is always a constant, then the function could be rewritten without referring to that variable, by substituting all references with the constant value. If that variable's value is $\perp$, then such a substitution is impossible, and the function both varies and is impure.

If an anonymous function merely writes variables in the environment, then it is a constant value. However, this may affect constant propagation. Edges exist into the body of the anonymous function anywhere it may be activated, so these writes are handled by CSS as simply a control-flow issue.

3.7.4 Constant Propagation: Supporting Algebraic Identities

Thus far, we have considered only instructions with constant operands to potentially result in a constant result. However, there exist operations between constants and $\perp$ that also result in a constant.

The simplest case, in a statically-typed language, is that $0 \times \perp = 0$. Similar identities exist for exponentiation: $1^{\perp} = 1$ and $\perp^{0} = 1$. Such identities naturally hold true in Ruby as well, yet we cannot implement them directly due to dynamic typing. Consider Listing 3.2, which could set $x \leftarrow 5 \times 0 = 0$ or sets $x \leftarrow$ “`Hello`″ $\times\, 0 =$ “″.

Not only would assuming $x = 0$ be an incorrect inference, if the operands to the multiplication were reversed, such an approach would be ignoring a type error and exception. However, such an approach *does* work if we are guaranteed to only have arguments of type `Numeric`. Thus, due to dynamic dispatch, if we wish to infer constants using such identities, we must:

Listing 3.2: Dynamic typing defeating traditional algebraic identities

```
if rand > 0.5
  y = 5
else
  y = 'Hello'
end
x = y * 0
```

1. Use the call's constant arguments and method name to identify an identity
2. Ensure the potential concrete types of the varying arguments all match the required types of the identities

While listing 3.2 demonstrates a program for which identities cannot be applied, listing 3.3 shows one we may approach, even though $y = \bot$ and the types of its possible values differ. The variable y is always `Numeric`, which means we may apply the $\bot \times 0 = 0$ identity. The only difference between this example and the previous is the type of y. $\{\texttt{Fixnum}, \texttt{String}\} \subsetneq \texttt{Numeric}$ in (3.2), yet $\{\texttt{Fixnum}, \texttt{Complex}\} \subseteq \texttt{Numeric}$ in (3.3).

Listing 3.3: Dynamic typing compatible with traditional algebraic identities

```
if rand > 0.5
  y = 5
else
  y = Complex(2, 4.5)
end
x = y * 0
```

Since CPA runs during constant propagation, the potential type sets for each temporary grow incrementally at each step. At first, in the negative example (3.2), CPA will infer that y has type $\{\texttt{Fixnum}\}$, and thus CSS may apply the $\texttt{Numeric} \times 0 = 0$ identity to infer that $x \leftarrow 0$. However, CPA must eventually determine that y has type $\{\texttt{Fixnum}, \texttt{String}\}$, triggering a re-evaluation of $y \times 0$ by CSS. CSS will fail to match an identity and assign $\bot$ to x.

In the positive example, y's type will never exceed $\{\texttt{Fixnum}, \texttt{Complex}\}$, which is a subtype of `Numeric`. Thus the identity $\texttt{Numeric} \times 0 = 0$ will hold throughout constant propagation.

3.7.5 Constant Propagation: Binding Complications

Until this point we have concerned ourselves with traditional interactions between variables and procedures, and for these cases, CSS is correct as long as our "predictability" estimates are correct. However, there exist dynamic features of Ruby that affect this propagation technique.

Function `eval()` is the simplest complicator, and when it is called with a varying value, we must assume all variables defined in blocks dominated by `eval()` are varying, as `eval()` could redefine them unpredictably. When called with a constant value, we could attempt to compile the code and analyze it; our implementation does not implement this in method bodies.

Additionally, at any point in a Ruby program, one may obtain a `Binding` object which encapsulates the context of execution at that point. This includes the values of local variables, the value of `self`, and any block argument in scope. The `Binding` class provides one method: `eval()`, which takes a string to be executed given the Binding object's context. Thus, using a Binding object, one can set local variables in a caller's call frame. This, like direct use of `eval()`, complicates constant propagation.

In the body of a given method, there are two ways to expose the binding of a caller to a callee: calling the method `Kernel#binding` and providing a pointer to that binding somehow, or passing to another method a reference to an anonymous function created in the caller's context. The `Proc#binding` method could then be used similarly.

Both of these behaviors are disastrous for analysis, yet they are exceedingly rare. Additionally, none of these methods can be implemented in pure Ruby. Our tool could potentially prove the absence of a call to `binding` in some cases; we do not implement a solution to this issue.

Chapter 4

High-Level Analyses

4.1 Inferring Block Use Patterns

As illustrated in (1.8), block use patterns are a prime target for analysis: methods that use their block argument typically fit simple patterns, but no tools exist to infer these patterns, and some errors are silently ignored by Ruby. As blocks can be activated through two mechanisms, we consider them separately. Note: since the algorithms described work on the CFG composed of *basic blocks*, I will refer to basic blocks as "nodes" to reduce confusion between Ruby's blocks.

4.1.1 Characterizing Block Use

We begin by formally defining the property we wish to estimate. We have discussed several intuitive types of methods: those requiring blocks, those which optionally allow blocks, and those which ignore their blocks.

A method requiring a block is simple to define:

Definition 1. *A method $M \in$* Block-Required $\subset$ Methods *if M* may *terminate via an exception raised by attempting to invoke the block argument when none is provided.*

Note the word "may": a Block-Required method need not always raise when called without a block, but it might. These methods, to be used safely, must be called with a block.

Again from the set of all methods, we define Block-Optional as follows:

Definition 2. *A method $M \in$* Block-Optional $\subset$ Methods *if M* never *terminates via an exception raised by attempting to invoke the block argument when none is provided.*

This leads nicely to the definition of Block-Ignored:

Definition 3. *A method $M \in$ Block-Ignored $\subset$ Block-Optional if $M \in$ Block-Optional and M* never *invokes its block argument when a block* is *provided.*

The symmetry in these definitions leads to a final, unlikely method type, Block-Foolish:

Definition 4. *A method $M \in$ Block-Foolish $\subset$ Block-Required if $M \in$ Block-Required and M* never *invokes its block argument when a block* is *provided.*

We consider a method M shown to be in Block-Foolish to merit a warning from the analyzer, as such a method is likely the result of an error.

The following table summarizes the four types of methods as we have described them:

		Block Provided	
		May Yield	Never Yields
No Block	May Yield	Block-Required	Block-Foolish
	Never Yields	Block-Optional	Block-Ignored

4.1.2 Complexity of Characterization

Determining if a method $M \in$ Block-Required is, intuitively, a question of whether the program ever enters a certain state. Such questions are, in the general case, *Turing-recognizable* but *undecidable.* We can construct a program that can confirm that a method is Block-Required, but if the method is not, then our program may never terminate. A proof that the language Required_M is undecidable is included in Appendix A.1.

Determining if a method $M \in$ Block-Optional is thus a question of whether the program *never* enters a certain state. Such questions are, in the general case, *Turing-unrecognizable*: we cannot even construct a program definitively confirming that $M \in$ Block-Optional. A proof that the language Optional_M is Turing-unrecognizable is also included in Appendix A.2.

This should not surprise us: most interesting questions about the properties of programs are, in the general case, undecidable or unrecognizable. We choose to concern ourselves with subsets of Block-Required and Block-Optional that we *can* identify rigorously.

4.1.3 Inferring Block Use: `yield`

The simplest mechanism to call a block argument is to use the `yield` keyword. When `yield` is encountered during CFG construction, its semantics may be directly implemented: the arguments are computed, and then a branch is taken on the method's block register, which is initialized in the method prologue. If the block register is `nil`, a branch is taken which creates an exception and raises it, following an abnormal CFG edge to the nearest exception handling target. If the block register is a `Proc` object, its `call` method is invoked with the arguments calculated earlier. An optimized CFG of a method `foo` which simply yields its first argument `x` is given (4.1) to illustrate this structure. (*Note: after this point, the actual graphs become too large to provide verbatim; simplified versions will be substituted*)

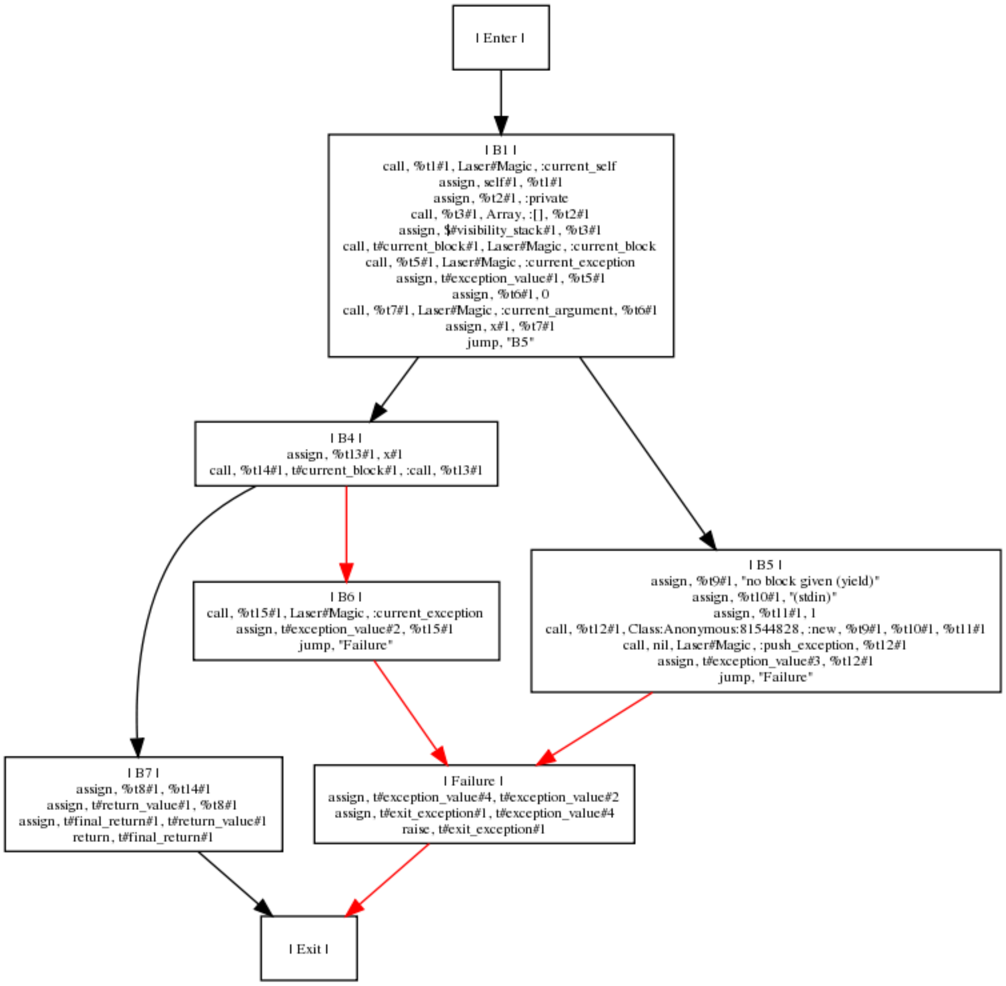
We hope to find a structural property of the graph that characterizes the block use of the modeled method. In this naïve formulation of the CFG, we see that the "Failure" node, which postdominates all method failures, has an edge coming directly from a node implementing yield-failure. During CFG construction, we might choose to add extra flags to these edges, and use their presence to characterize the potential for calling yield without a block. An early implementation did just this, but it is foiled by the ability to catch the `LocalJumpError` raised by yield failure. Not only does catching `LocalJumpError` remove the characteristic structure identifying yield failure, it will interfere with constant propagation at the rescue handler (Figure 4.2).

Instead, we create a separate exception-handling path for yield failure (Figure 4.2). This separate path duplicates the catch-checking logic, but is guaranteed to only consider exceptions of class `LocalJumpError`. This will make the yield-failure exception path highly amenable to the CSS Constant Propagation algorithm we have already implemented.

Given this CFG structure, and no capturing of the block to a variable, we can characterize many methods' block use based on answering two questions, each of which can be answered efficiently:

- If we assume the method has no block, and `block_given?` returns false, does CSS eliminate the edge in the CFG between the yield-failure exception path and the

Figure 4.1: A simple Block-Required method's CFG


| Enter |
| B1 |
call, %t1#1, Laser#Magic, :current_self
assign, self#1, %t1#1
assign, %t2#1, :private
call, %t3#1, Array, :[], %t2#1
assign, $#visibility_stack#1, %t3#1
call, t#current_block#1, Laser#Magic, :current_block
call, %t5#1, Laser#Magic, :current_exception
assign, t#exception_value#1, %t5#1
assign, %t6#1, 0
call, %t7#1, Laser#Magic, :current_argument, %t6#1
assign, x#1, %t7#1
jump, "B5"
| B4 |
assign, %t13#1, x#1
call, %t14#1, t#current_block#1, :call, %t13#1
| B6 |
call, %t15#1, Laser#Magic, :current_exception
assign, t#exception_value#2, %t15#1
jump, "Failure"
| B5 |
assign, %t9#1, "no block given (yield)"
assign, %t10#1, "(stdin)"
assign, %t11#1, 1
call, %t12#1, Class:Anonymous:81544828, :new, %t9#1, %t10#1, %t11#1
call, nil, Laser#Magic, :push_exception, %t12#1
assign, t#exception_value#3, %t12#1
jump, "Failure"
| B7 |
assign, %t8#1, %t14#1
assign, t#return_value#1, %t8#1
assign, t#final_return#1, t#return_value#1
return, t#final_return#1
| Failure |
assign, t#exception_value#4, t#exception_value#2
assign, t#exit_exception#1, t#exception_value#4
raise, t#exit_exception#1
| Exit |

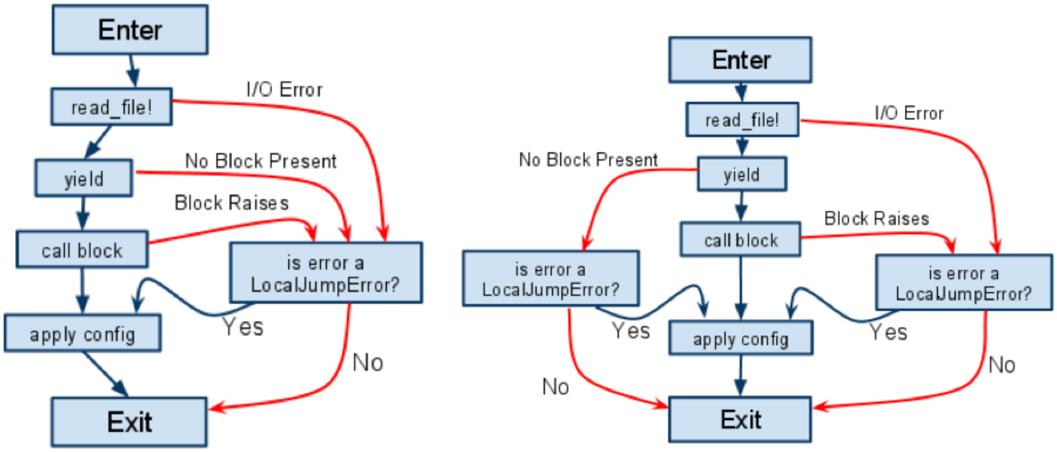
method exit?

- If we assume the method has a block, and `block_given?` returns true, are there any instructions in the CFG of the form $call(t\#current\_block, \text{“}call\text{''}, ...)$?

The first question offers an opportunity to prove that `yield` is not called when no block is provided: if CSS can optimize away the entire yield failure path, we can be sure that the yield is guarded by the use of simple constants and `block_given?`.

The second question offers an opportunity to prove that `yield` is not called when the block *is* provided. The simplest case is if `yield` simply never appears in the body of the method, but there may also be a yield guarded by simple constants. Additionally, if the

Figure 4.2: Naïve CFG construction masks yield failure behind exception handlers



method is ∈ Block-Foolish, the second question may be in the negative despite the presence of a call to `yield`. We can encapsulate these two questions into two algorithms, YieldsWhenAbsent (line 1) and YieldsWhenPresent (line 6). We will use these algorithms later to characterize the method wholly.

Algorithm 1 Inferring Yield Use

```
function YieldsWhenAbsent(G)
    CSS(G, Kernel#block_given: false, block: false)
    $G' \leftarrow$ TrimDeadEdgesAndBlocks($G$)
    return YieldFailPostDominator $\in G'$
end function

function YieldsWhenPresent(G)
    CSS(G, Kernel#block_given: true, block: $\beta$)
    $G' \leftarrow$ TrimDeadEdgesAndBlocks($G$)
    for all instruction $\in G'$ do
        if instruction calls $t\#current\_block.call$ then
            return true
        end if
    end for
    return false
end function
```

4.1.4 Inferring Block Use: `Proc#call`

The second form of invoking a block is to capture it and call it. A simple example is illustrated in (Listing 4.1), implementing `map` by naming the block argument in the list of formal arguments.

Listing 4.1: A simple invoking a block using `Proc#call`

```
def map(list, &blk)
  result = []
  list.each do |element|
    result.push(blk.call(element))
  end
  result
end
```

`Proc.new()` with no block argument returns the active frame's block argument, as well.

Without this functionality, a block could only be invoked by the callee, a significant limitation. However, it introduces new challenges for analysis. By introducing the block as the variable `blk`, any object with a reference to `blk` may invoke the block argument, and it may even be invoked after the callee terminates. The `Kernel#lambda` method in fact simply returns its block argument to its caller, permitting the *caller* to call the block.

Thus, when a procedure acquires a reference to its block parameter, our analysis revolves around potential aliases to that parameter. Any method call that has the block as a parameter could invoke the block, and in the general case, any other parameter could obtain a reference to the block. Any method call involving an argument that may reference the block could call the block as well. If a reference is written to shared memory, such as an instance variable or global variable, analysis becomes even more difficult.

With `yield`, the callee uses `block_given?` to check if there is a block argument to call. If the callee uses one of the above techniques to obtain a reference to its block, it may also compare against `nil` to check if a block was actually provided. Thus, if we wish to determine if a method is Block-Optional, we will need to successfully prune such `nil` comparisons in the negative case. This will, unsurprisingly, be an instance of the Constant Propagation problem.

To infer block use with an explicit block parameter or a call to `Proc.new()`, the following framework provides a conservative analysis:

- Conservatively identify potential aliases to the block parameter.
- For $B \in \{nil, \beta\}$:
  - Run CSS, fixing the block argument to B.
  - Identify reachable invocations of `#call`, `#[]`, and `#===` on a potential alias. These are guaranteed block invocations.
  - Identify method calls with a potential alias as the block argument. The block is potentially invoked if any potential dispatch target is Block-Requried.
  - Identify method calls with a potential alias as an argument. The block is potentially invoked, unless interprocedural analysis (eg. inlining) proves otherwise.

This provides the equivalent information as the information derived during `yield` analysis, with weaker accuracy due to the alias analysis dependency.

4.1.5 Inferring Block Arity

Given a set of potential block call sites, we develop an estimate of how many arguments will be provided to the block argument. Each call may be of either fixed or variable arity. The total arity is thus a (potentially infinite) set of integers. Fixed-arity block invocations add their arity to that set. Variable-arity invocations add to the set all possible values for the number of arguments they provide; inferring this requires tracking potential sizes of array objects throughout the method in question. Our current implementation does not attempt to infer this; variable-arity invocations imply the block could be called with all potential arities under our analysis.

4.1.6 Inferring Block Argument Types

Block call-sites are simply method calls, so much like any method call, it participates during type inference (CPA). The built-in `Proc#call` method does not have any type information accompanying the analyzer, so during CPA the block will be presumed to return an object of any type. We do not consider how to efficiently discern this information. However, CPA may infer the types of the arguments passed *to* the block, which is a relevant portion of the method's block API. For example, the analyzer correctly infers that the `times` method (Listing 4.2) calls its block argument with an object of type $\{\texttt{Fixnum}, \texttt{Bignum}\}$.

Listing 4.2: Example: Inferable block argument types

```
def times(n)
  i = 0
  while i < n
    yield(i)
    i = i + 1
  end
end
```

Variable arity again obstructs this analysis, as it requires knowing the types of elements inside an `Array`. DRuby addresses this with the addition of *Tuple* types to its type system. [FAFH09a]

4.1.7 Discovering Errors

We developed this analysis in order to report errors to the programmer who wrote the methods in question. The analyses discussed so far imply a number of potential errors that could be uncovered about a method M and its behavior with respect to blocks:

- M $\in$ Block-Foolish.
- M $\in$ Block-Ignored; M is called with a block.
- M $\in$ Block-Required; M is called without a block.
- A = Block-Arity(M), $A \subset \mathbb{Z}$; M is called with a block which accepts $\alpha \subset \mathbb{Z}$ arguments; $\alpha \cap A = \varnothing$.
- M is called with a block, and M calls the block with arguments of types $T = (T_1, T_2, ..., T_k)$. The block's computation is invalid with argument types T.

We implement and test conservative analyses for the first four error types above. The analyzer will incidentally discover some errors of the fifth kind during constant propagation, but no additional analysis is implemented to directly find such errors.

4.2 Exit Behavior Analysis

A method M may exit in one of several ways in Ruby. The following list considers them roughly in an intuitive order of prevalence in Ruby code:

1. Normally, with a return value
2. Abnormally, due to an exception
3. Via non-local jump (a block returns from its enclosing scope)
4. via `throw`/`catch`
5. By invoking a continuation

Given a method M, how do we characterize how it exits? Like block behavior, this is a relevant feature of how one uses a method. We consider a smaller, but important question: does a method M exit due to an exception *never*, *sometimes*, or *always*? CPA already provides a mechanism for parameterizing methods by argument types, so we can consider this question about M both in its general form, and when it is provided specific argument types.

4.2.1 Characterizing Exception Behavior

As we did during Block-use analysis, we break methods into sets based on their behavior:

Definition 5. *A method* $M \in \text{RAISE-NEVER} \subset \text{METHODS}$ *if* M never *exits bearing an exception.*

Definition 6. *A method* $M \in \text{RAISE-MAYBE} \subset \text{METHODS}$ *if* M sometimes *exits bearing an exception.*

Definition 7. *A method* $M \in \text{RAISE-ALWAYS} \subset \text{METHODS}$ *if* M always *exits bearing an exception.*

If a CFG equivalent to M's CFG has no exception-bearing edges leading to M's "Exit" block, then $M \in$ Raise-Never. Similarly, if *only* exception-bearing edges lead to M's "Exit" block, then $M \in$ Raise-Always. Raise-Maybe is thus characterized by the presence of both types of exit edges. Non-terminating methods are in Raise-Never.

However, any method call in user code will, during the construction of the CFG, create an exception-bearing edge. Only the most trivial of methods will, after CFG generation, lack an exception-bearing edge to the Exit. All methods will have a "normal" exit edge immediately after construction.

4.2.2 Raise Analysis During Constant Propagation

Given that methods will rarely reveal much about their exit behavior given a conservative CFG, we must remove some exception and normal edges in the CFG based on knowledge of the methods called. For example, the type-checking method `Module#===` never raises, and `Kernel#raise` always raises. If we identify calls to these methods, we may trim exception-bearing and normal edges; this approach applies to both built-in methods and user code.

Presently, we implement a simple heuristic. During constant propagation (see 3.7), exception edges in the CFG are encountered under two conditions: a failed `yield`, and exiting a method call. We wish to determine which exception edges are executable, which constant propagation is already responsible for. Normally, if a method call is found to be executed, all edges leaving it are marked executable.

Instead, we mark the edges leaving a method call by the following heuristic. If the set of potential methods called is $\mathbb{M}$:

- The normal edge is executable if $\exists M \in \mathbb{M}, M \in$ Raise-Never $\cup$ Raise-Maybe
- The exception-bearing edge is executable if $\exists M \in \mathbb{M}, M \in$ Raise-Always$\cup$Raise-Maybe

If a method $M \in \mathbb{M}$ is implemented in C, its raise behavior must be annotated manually. If it is implemented in Ruby, then the analysis proceeds recursively, just as CPA does.

This analysis could also be applied to consider the argument types provided to the method. Many methods, such as the basic arithmetic operators, only raise when provided non-`Numeric` arguments. This would improve the accuracy of the analysis at the expense of time and space. Since exceptions are likely to be raised due to improper argument types, specializing this analysis on argument types improves results and allows accurate modeling of standard library methods.

4.3 Unreachable Code Analysis

Unreachable code elimination is one of the simpler compiler optimizations one can perform given a CFG. However, we delay performing this analysis until after constant propagation, as CSS's edge elimination may improve the results of dead code elimination.

Any node not reachable from the start node is dead. By computing the depth-first search tree starting at the *Enter* block (ignoring fake edges, critically) one obtains a set of blocks that *are* reachable; subtracting this set from the set of all blocks yields the set of unreachable basic blocks. Any such blocks can be removed, reducing the size of the resulting binary and potentially simplifying generated code (branches becoming jumps, for example). For a linter, however, such nodes represent an error by the programmer, and should be reported. Deleting the block is not an option.

Given an approach to identify unreachable CFG instructions, we must define precisely how to determine that source-level code is unreachable. The CFG is unsuitable for this task, but the AST which generated it is close to the source representation.

Definition 1. *A node in an Abstract Syntax Tree is* ***unreachable*** *if and only if all instructions in the CFG implementing that node's semantics are unreachable.*

As an AST node carries with it source line and column information, if we determine which nodes are unreachable, we create a warning at that line and column. Any node which is unreachable will be the root of a subtree of unreachable nodes. With all nodes in the

AST annotated as such, simply depth-first-searching for unreachable nodes will discover all such subtrees.

Algorithm 2 Incorrect, naïve reachability algorithm

```
Input: $G = (V, E, Enter, Exit)$, a control-flow graph
  for $node \in G.AST$ do
      $node.reachable \leftarrow false$
  end for
  for $block$ ∈DFS(G.Enter) do
      for $instruction \in block$ do
          $instruction.node.reachable \leftarrow true$
      end for
  end for
```

The naïve approach, in which all nodes are marked unreachable first, then all reachable instructions mark their nodes as reachable, seen in algorithm (2), fails, though. It fails to account for AST nodes that themselves generate no instructions when walked. A `paren` node is one example: upon reaching a `paren` node when building the CFG, we simply walk all of its children in sequence. No instructions specific to the `paren` node will be generated, so the `paren` node will be left marked *unreachable*. This result came from an insufficient definition of an unreachable AST node (4.3). The revised definition follows:

Definition 2. *A node in an Abstract Syntax Tree is unreachable if and only if all instructions in the CFG implementing that node's semantics are unreachable,* ***and the set of such instructions is nonempty****.*

A revised algorithm only marks nodes unreachable if they have instructions in unreachable basic blocks. We can find all unreachable blocks in $O(V)$ time by depth-first searching from *G.Enter* to find reachable blocks, then subtracting this set from the set of all blocks to get the set of unreachable blocks. The correct algorithm, 3, implements this.

If any instruction that is generated for a given AST node lies in a reachable block, then the final loop will mark that node as reachable. Yet, if no instructions exist for a given node, it will be marked reachable unconditionally.

Algorithm 3 Correct reachability algorithm

```
Input: $G = (V, E, Enter, Exit)$, a control-flow graph
  $ReachableBlocks \leftarrow \phi$
  for $block \in DFS(G.Enter)$ do
      $ReachableBlocks \leftarrow ReachableBlocks \cup \{block\}$
  end for
  $DeadBlocks \leftarrow G.V - ReachableBlocks$
  for $node \in G.AST$ do
      $node.reachable \leftarrow true$
  end for
  for $block \in DeadBlocks$ do
      for $instruction \in block$ do
          $instruction.node.reachable \leftarrow false$
      end for
  end for
  for $block \in DFS(G.Enter)$ do
      for $instruction \in block$ do
          $instruction.node.reachable \leftarrow true$
      end for
  end for
```

Given such an annotated AST, reporting unreachable code is simple. As noted earlier, all children of an unreachable AST node are also unreachable, yet we only wish to warn against the root of each subtree; additional warnings are unnecessary. A modified depth-first search applies these warnings (see Algorithm 4).

Algorithm 4 Generate Warnings for Unreachability

```
  procedure DEADDFS(node)    ▷ node: AST node, whose subtree is annotated as above
      if $node.reachable$ then
          for $child$ ∈CHILDREN(node) do
              DEADDFS(child)
          end for
      else
          WARN($`deadcode', node.line\_number$)
      end if
  end procedure
```

4.4 Unused Variable Analysis

Finding unused variables as an automated analysis technique goes at least to Johnson's original *Lint* for C. He elegantly describes the motivations for warning programmers about unused variables [JI77]:

> As sets of programs evolve and develop, previously used variables and arguments to functions may become unused; it is not uncommon for external variables, or even entire functions, to become unnecessary, and yet not be removed from the source. These "errors of commission" rarely cause working programs to fail, but they are a source of inefficiency, and make programs harder to understand and change.

Since then, this analysis has continued to be implemented in compilers [Fou11], linters alike [Cro11, Sof11], and editors [Fou10].

Morgan's Static Single Assignment algorithm tracks both the (only) definition of each variable in SSA form as well as the instructions that use each variable. These are the $Definition(T)$ and $Uses(T)$ sets, respectively. Once the SSA transformation is complete, these sets are populated. However, the SSA transformation was a prerequisite for many of the analyses performed thus far, which may have pruned edges not taken from the CFG. Any uses of a variable in an unreachable block will still be present in the $Uses(T)$ set. Thus, for each variable, we must ensure the variable is used at least once in a reachable code block. Algorithm (5) summarizes this approach cleanly, finding unused variables and pruning unreachable instructions from $Uses(T)$.

This algorithm finds each variable which is defined and then never read in a reachable block. It is $O(V \times U)$, where U is the number of times a variable is used.

However, there exist further variables that may represent an error: given an unused variable y, all variables whose only use is in computing y may also be considered unnecessary. Consider listing 4.3, where x, y, and z are unnecessary, but a is not:

Algorithm 5 Algorithm Warning for Unused Variables

```
function SimpleUnused
    $result \leftarrow Variables$
    for $var \in Variables$ do
        for $use \in Uses(var)$ do
            if $use.block \in ReachableBlocks$ then
                $result \leftarrow result - \{var\}$
            else
                $Uses(var) \leftarrow Uses(var) - \{use\}$
            end if
        end for
    end for
    return $result$
end function
```

Listing 4.3: Additional Unused Variables

```
x = 'hi'
y = x * 300

a = 'EOF'
z = gets(a)
```

The above algorithm correctly identifies y and z as unused local variables, as they are never used. Since y was defined by a side-effect-free computation, removing the entire line $y = x * 300$ has no effect on the overall program. If one did so, x is then unused, which a programmer might find concerning. We cannot apply the same reasoning to eliminate $z = gets(a)$, as this would change the meaning of the program. Thus, the calling of an *pure* method may be removed, as can a simple assignment. However, the calling of an *impure* method may not be removed.

We rely on method resolution and purity analysis [3.6] to determine which method calls we may kill. Assignments and ϕ-nodes are always killable. This gives us a simple worklist algorithm that propagates removal of uses of variables backward (Algorithm 6). It is worth noting that this algorithm also infers unused formal arguments.

Algorithm 6 Algorithm Warning for Chained Unused Variables

```
function KILLABLE(I)
    if $I.type \in \{assign, lambda\}$ then
        return true
    else if $I.type = call$ then
        return $AllPure(I.possible\_methods)$
    else
        return false
    end if
end function

function CHAINEDUNUSED
    $worklist \leftarrow$ SIMPLEUNUSED()
    $result \leftarrow \phi$
    while $worklist \neq \phi$ do
        $unused\_var \leftarrow pop(worklist)$
        $result \leftarrow result \cup \{unused\_var\}$
        WARN($unused\_var$, $Definition(unused\_var).line\_number$)
        if KILLABLE($Definition(unused\_var)$) then
            for all $operand \in Operands(Definition(unused\_var))$ do
                $Uses(operand) = Uses(operand) - \{Definition(unused\_var)\}$
                if $Uses(operand) = \phi$ then
                    $worklist \leftarrow worklist \cup \{operand\}$
                end if
            end for
        end if
    end while
end function
```

Chapter 5

Warnings and Errors Discovered

This chapter details the diagnostics the analyzer is capable of generating when it encounters inferable errors and warnings in Ruby code. For the purposes of this discussion, a "warning" indicates a potential error by the user. Some of these diagnostics are also emitted at runtime by the Ruby interpreter, and some are not. Some runtime errors more readily discoverable than others without the aid of analysis tools. We attempt to clarify for each diagnostic precisely how it would be traditionally encountered.

5.1 Load-time Errors

The following sections consider program errors whose negative effects occur immediately upon loading the program into the Ruby interpreter.

5.1.1 Top-Level Nontermination or Stack Overflow

Class and module definitions are executed, and as such, metaprogramming techniques commonly utilize loops or recursion to modify the class creation process. This presents the possibility of introducing an infinite loop or recursion during class creation - preventing the program from even beginning to process its input. Infinite recursion will result in a `StackOverflowError` and likely terminate the Ruby interpreter; infinite loops are naturally more difficult to detect.

As we directly interpret predictable top-level code (3.3), we will ourselves observe a `StackOverflowError` if one occurs. We catch this error and output this information to the user. We terminate analysis at this point, though we could potentially attempt to continue in an undefined state. In addition, during analysis, we track how many steps we have taken, to prevent the analyzer itself from looping infinitely; if a capped number of steps X are taken, a `SimulationNonterminationError` is raised and reported to the user. By default, $X = 100,000$ basic blocks.

5.1.2 Double Module Inclusion

When a class or module `include`s or `extend`s a module, the module is inserted into a well-defined location in the class's inheritance hierarchy. If the module is `include`d or `extend`ed again, Ruby ignores the request, as such a request has no effect. We consider this to merit a warning. Since most module inclusions occur in top-level class and module definitions, we will observe them during top-level simulation (3.3). We report this as a `DoubleIncludeError` and continue analysis.

The Ruby interpreter never emits a diagnostic for this behavior.

5.1.3 Mismatched Superclasses

Ruby's *open classes* present opportunities for error. When a class is defined with superclass C_1 and later re-opened with superclass C_2, an error is immediately raised. When we build our CFG, we expand these semantics explicitly; during top-level simulation, the appropriate raise instructions will be executed if a specified superclass is incorrect. In fact, with CFG optimization enabled, the CFG will be reduced to a deterministic exception, with the rest of the code in the file optimized away. If the error occurs during simulation, or if the reduced CFG implies a guaranteed exception, we report this error to the user.

This error would be discovered by a programmer immediately upon running the program.

5.1.4 Opening a Class as a Module (and vice-versa)

Another issue raised by open classes is that one might confuse a class for a module or vice-versa. This mistake primarily stems from the fact that both classes and modules can be used as namespaces. Attempting to open a class using module syntax, or a module using class syntax, results in an exception. This exception is also present in the lowered representation of the `class C ... end` and `module M ... end` syntactic constructs. It is observed and reported in the same manner as mismatched superclasses.

This error would be discovered by a programmer immediately upon running the program.

5.2 Run-time Errors

The remaining errors may occur anywhere in a program, and are observed and reported in the same way regardless of their presence at the top-level or in method bodies. A common theme of the following errors is that many will rely on method resolution, and thus their efficacy will improve with type inference, either via CPA, or through user-provided annotations.

5.2.1 Unnecessary Block Arguments

As described in Chapter 4 (4.1), Ruby silently ignores when a block is provided and not used. This is because Ruby does not know that the block will not be used. Our analysis can prove some methods are in Block-Ignored, and report an error when a block is provided to them. When we encounter a method call during constant propagation, we already conservatively calculate the possible dynamic dispatch targets. If all of these targets are in Block-Ignored, and a block is provided to the call, we issue an `UnnecessaryBlockError` and continue analysis.

5.2.2 Missing Block Argument

In the same vein as unnecessary block arguments, we may also warn when a called method is potentially in Block-Required and no block is provided. This is prone to false positives: our analysis infers a method to be in Block-Required when we cannot prove it to be either Block-Optional or Block-Foolish. As both of those sets are undecidable, we cannot be sure that no block was provided at a call-site because the method was really in Block-Ignored. However, Block-Ignored methods are typically constructed with a relatively simple set of patterns, and we correctly infer these patterns. We are thwarted primarily by block capture and forwarding; analyzing this case precisely depends on precise type information.

Regardless, we issue a `MissingBlockError` if all potential targets of dynamic dispatch are in Block-Required and no block argument is provided. This error may be found at runtime if the callee attempts to invoke the (not provided) block argument.

5.2.3 Incorrect Arity

Method resolution provides an avenue for checking arity as well. When a method is called with a fixed number of arguments, and we know all possible targets of that method call, we can ensure at least one will accept that number of arguments. Variable arity is currently not considered, as the analyzer lacks any facility for tracking the potential size of an array throughout a CFG. An `IncompatibleArityError` is issued when a method call is found with a number of arguments that is not compatible with any potential dynamic dispatch target.

Ruby raises an `ArgumentError` when this error occurs at runtime.

5.2.4 Privacy Errors

Ruby offers both `protected` and `private` methods, and the implementation differs somewhat from other languages: to execute a `private` method, the user must not use a receiver in the method call at all, even `self`. This is an easy error to commit when working with private methods. Much like the previous sections would indicate, this error is found when all potential dynamic dispatch targets have privacy modifiers incompatible with the invocation. A `DisallowedPrivateError` or `DisallowedProtectedError` is issued when this error is discovered.

Ruby raises a `NoMethodError` with an appropriate message when this error occurs at runtime.

5.2.5 Missing Method Errors

If a callsite is found with no matching dynamic dispatch targets, a `NoSuchMethodError` is issued. Among the typical causes of such an error, calling a non-existent method is especially commonly due to typos; incorrectly typing the name of a local variable will be considered a method call with `self` as the receiver.

Ruby raises a `NoMethodError` with an appropriate message when this error occurs at runtime.

5.2.6 Improperly Overriding Core Methods

A number of methods exist in Ruby that have expected return types. The `to_s` method should always return a `String`, and `!` should always return a boolean. These methods may be overridden, but if the return type is incorrect, then they may interact improperly with the expectations of core language constructs. A small set of methods have their expected types hard-coded into the analyzer. If a method with a name in that set is found to have an incompatible return type, then an `ImproperOverloadTypeError` is issued.

5.3 Warnings

The following are potential sources of logic error that we consider worth reporting to the user. They do not necessarily indicate a error, and do not cause exceptions at runtime. The analyzer may be directed to ignore these warnings via comments in analyzed code.

5.3.1 Catching `Exception`

A `rescue` clause in Ruby uses a list of potential handlers to determine whether to catch a raised exception. These handlers are any object that responds to `#===`, but are nearly always subclasses of `Exception`. A common error in Ruby is to use `rescue Exception` to catch "any exception", but this is usually not correct. `Exception` is the root class in the exception hierarchy, but its subclasses include `SystemExitError`, which is raised by the standard `exit()` method. `SyntaxError` is also a subclass of `Exception`, as is `NoMemoryError`, `SystemStackError`, and `SignalException`. While there exist reasons to catch all of these exceptions, it is unlikely that most programmers wish to do so. When one wishes to catch "any exception", they typically mean `TypeError`, `ArgumentError`, and so on, not interpreter-level errors. Thus, the correct idiom is `rescue StandardError`, the base class of all these common user-level exceptions.

We discover this potential error during CFG construction, and report it as a `RescueExceptionWarning`.

5.3.2 Dead Code

As discussed in Chapter 4, we discover unreachable code after trimming edges in constant propagation. We attempt to minimize a flood of warnings by only reporting the lines of code corresponding to the highest-level unreachable AST nodes. One issue with reporting dead code is that the errors listed previously result in exceptions, which redirect control flow. If analysis uncovers a `NoMethodError` in a block of code, all dominated expressions will not execute due to the guaranteed exception. This is an instance of the *cascading error* problem affecting both compilers and linting tools alike. We do not presently address this issue.

5.3.3 Unused Variables

As discussed in Chapter 4, we issue `UnusedVariableWarning`s upon the discovery of unused variables. Like dead code discovery, variables will be found to be unused if they follow guaranteed exceptions. As with dead code, we do not presently attempt to address the resulting cascading errors.

Chapter 6

Concluding Remarks

6.1 Our Results

Our top-level simulation successfully emulates widely used metaprogramming techniques, including those in Ruby's standard library (`Struct`, `DelegateClass`). Combining all of the analyses, we successfully infer the block use patterns of many test examples and standard library methods, as well as their raise behavior. We warn statically against new errors which otherwise would fail only at runtime, or be silently ignored.

The code for the analyzer can be found at https://www.github.com/michaeledgar/laser/, hosted by GitHub, a free code hosting service using PKI authentication.

6.2 Limitations

A small but significant subset of Ruby is not supported, including the "special" Ruby variables ($&, $1, etc.), `super` in dynamically-defined methods, and all but one block argument semantics. The stubbed-out standard library is not complete. Recursion varying by local block types is not supported (*recursive customization* [Age95]).

6.3 Future Work

This work opens the door to further analyses. Ruby has many tricky corner cases the analyzer could warn against, including the semantics of implicit super, mismatched parallel assignment, and for-loops (namely, their lack of a new scope). A block's arguments shadow local variables with the same name; this is warned against by Ruby, but we could easily warn against it ourselves statically. Determining precisely which exception types a method might raise is within the reach of our existing exception analysis.

The analyzer could be integrated with documentation tools, editors, and/or compilers to improve the results of each.

List of Algorithms

References

[ACF09] J. An, A. Chaudhuri, and J. S. Foster. Static Typing for Ruby on Rails. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 590–594. IEEE Computer Society, 2009.

[ACPR] R. C. Andreas, B. Cook, A. Podelski, and A. Rybalchenko. Terminator: Beyond safety. *In Computer Assisted Verification '06, LNCS*, 4144:415–418.

[Age95] O. Agesen. The Cartesian Product Algorithm. In *ECOOP'95—Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995*, pages 2–26. Springer, 1995.

[CFR$^+$89] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An Efficient Method of Computing Static Single Assignment Form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 25–35. ACM, 1989.

[CPR06] B. Cook, A. Podelski, and A. Rybalchenko. Termination Proofs for Systems Code. *ACM SIGPLAN Notices*, 41(6):415–426, 2006.

[Cro11] Douglas Crockford. Jslint: The javascript code quality tool, April 2011.

[Dav11] Ryan Davis. parse_tree & ruby_parser, March 2011.

[Duf] M. Dufour. *Shedskin: An Optimizing Python-to-C++ Compiler*. PhD thesis, Master's thesis, Delft University of Technology, 2006.

[Edg11a] Michael Edgar. Ripper loses MLHS variables in the presence of an LHS splat, February 2011.

[Edg11b] Michael Edgar. Ripper.sexp should return an :array node for words/qwords, February 2011.

[FAF09] M. Furr, J. D. An, and J. S. Foster. Profile-Guided Static Typing for Dynamic Scripting Languages. In *Proceeding of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, pages 283–300. ACM, 2009.

[FAFH09a] M. Furr, J. D. An, J. S. Foster, and M. Hicks. *Diamondback Ruby Guide*. University of Maryland, Computer Science Department, April 2009.

[FAFH09b] M. Furr, J. D. An, J. S. Foster, and M. Hicks. Static Type Inference for Ruby. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, pages 1859–1866. ACM, 2009.

[FAFH09c] M. Furr, J. D. An, J. S. Foster, and M. Hicks. The Ruby Intermediate Language. In *Proceedings of the 5th symposium on Dynamic languages*, pages 89–98. ACM, 2009.

[Fou10] Eclipse Foundation. Java compiler errors/warnings preferences, June 2010.

[Fou11] Free Software Foundation. Warning options - using the gnu compiler collection (gcc), April 2011.

[Hec77] M. S. Hecht. Flow Analysis of Computer Programs. The Computer Science Library Programming Language Series, 1977.

[JBGG96] G. James, J. Bill, S. Guy, and B. Gilad. The Java Language Specification, 1996.

[JI77] Johnson and Bell Telephone Laboratories Inc. *Lint, a C Program Checker*. 1977.

[Kri07] K. Kristensen. Ecstatic: Type Inference for Ruby Using the Cartesian Product Algorithm. *Master's thesis, Aalborg University*, 2007.

[Mor98] R. Morgan. *Building an Optimizing Compiler.* Digital Press Newton, MA, USA, 1998.

[PC94] J. Plevyak and A. A. Chien. Precise Concrete Type Inference for Object-Oriented Languages. *ACM SIGPLAN Notices*, 29(10):324–340, 1994.

[Sof11] Sureshot Software. Sureshot software, March 2011.

[Tur37] A.M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(1):230, 1937.

[WZ91] M. N. Wegman and F. K. Zadeck. Constant Propagation with Conditional Branches. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(2):181–210, 1991.

Appendix A

Appendix A: Proofs of Related Theorems

A.1 Required$_M$ is Undecidable

As described earlier (4.1.2), determining if a method is Block-Required essentially asks if the method ever enters a particular state. We define the decision formulation of this question as the language:

Required$_M$ = $\{\langle M \rangle \mid M$ is a method and may terminate via an exception raised by attempting to invoke the block argument when none is provided.$\}$

This is, in the general case, an undecidable language, which we may show by reducing a known undecidable language to Required$_M$.

Proof. We reduce A_{TM} to Required$_M$ as follows: [1]

Input: M is a TM, $w \in \Sigma^*$

Output: N is a Ruby method

[1] Recall the definition of A_{TM}: $A_{TM} = \{\langle M, w \rangle \mid M$ is a TM and terminates on w with *accept*$\}$

Algorithm:

1. Create a Ruby method N taking no arguments, which when run:
   (a) Runs $M(w)$.
   (b) If M is in the accept state, `yield`.
   (c) Otherwise, `return`.
2. Return N.

We must show that N is in $\textsc{Required}_M$ if and only if M accepts w.

- If M accepts w, then step 1 of N terminates in finite time. At step 2, the method yields to its block argument. However, if N is called without a block argument, it will still *yield*. Thus, $N \in \textsc{Required}_M$.
- If M rejects w, then step 1 of N terminates in finite time. It then returns at step 3. This method never yields, so it never raises an exception based on yielding. Thus, $N \notin \textsc{Required}_M$.
- If M loops forever on w, then step 1 of N never terminates. This method never yields, so it never raises an exception based on yielding. Thus, $N \notin \textsc{Required}_M$.

As A_{TM} is undecidable, and we have reduced it to $\textsc{Required}_M$, $\textsc{Required}_M$ too must be undecidable.

□

A.2 $\textsc{Optional}_M$ is Turing-unrecognizable

Determining if a method is Block-Optional essentially asks if the method *never* enters a particular state: the state in which it yields to the block argument. We define the decision formulation of this question as the language:

$\textsc{Optional}_M = \{\langle M\rangle \mid M$ is a method and *never* terminates via an exception raised by attempting to invoke the block argument when none is provided.$\}$

This is, in the general case, a Turing-unrecognizable language, which we may show by reducing a known unrecognizable language to $\textsc{Optional}_M$.

Proof. We reduce $\overline{A_{TM}}$ to $\textsc{Optional}_M$ using the exact same reduction as the proof of the undecidability of $\textsc{Required}_M$. (See A.1) [2]

We must show that N is in $\textsc{Optional}_M$ if and only if M rejects w or M runs forever on w.

- If M accepts w, then step 1 of N terminates in finite time. At step 2, the method yields to its block argument. However, if N is called without a block argument, it will still *yield*. Thus, $N \notin \textsc{Optional}_M$.
- If M rejects w, then step 1 of N terminates in finite time. It then returns at step 3. This method never yields, so it never raises an exception based on yielding. Thus, $N \in \textsc{Optional}_M$.
- If M loops forever on w, then step 1 of N never terminates. This method never yields, so it never raises an exception based on yielding. Thus, $N \in \textsc{Optional}_M$.

As $\overline{A_{TM}}$ is Turing-unrecognizable, and we have reduced it to $\textsc{Optional}_M$, so too must $\textsc{Optional}_M$ be Turing-unrecognizable.

□

[2] Recall the definition of $\overline{A_{TM}}$: $\overline{A_{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and either } M \text{ terminates on } w \text{ with } \textit{reject}, \text{ or } M \text{ runs forever on } w\}$