

Chapter 1

Introduction

Software development is time consuming. When building large and complex pieces of software, programmers rely on developer tools, especially ones that ‘understand’ the semantics of the programming language, to keep up productivity and avoid performing mechanical tasks which can be automated. Program slicers are tools which, amongst other things, could aid in the debugging process. This dissertation investigates program slicing in the relatively young programming language Swift.

1.1 What is program slicing?

Mark Weiser introduced the idea of ‘program slicing’ in his 1979 thesis [22]. Given a source program, a set of variables, and a location in the source program a program slice is a subset of the program with only those instructions that can possibly affect the value of any of those variables at that location. We call the set of variables, along with the location, the ‘slicing criterion’ and I shall refer to tools that perform program slicing as ‘program slicers’. Figure 1.1 demonstrates a short program and its slice with respect to the criterion $\langle \text{sum}, \text{line 13} \rangle$. This example is adapted from Frank Tip’s 1994 survey of slicing techniques [19].

1.1.1 Uses of slicing

The obvious next question is what these slices can be used for. Many applications for program slicing have been proposed and implemented. Each application may require different performance, precision, and features of the program slicer. Here I discuss just a few of the proposed applications.

Debugging

Weiser argues in [21] that the natural ‘work-backwards’ approach to debugging, in which the programmer finds the point in the program at which the error is first seen and then follows the chain of events backwards to find the source of the issue, is closely related to the task of program slicing. He explored the idea that expert programmers use slices as a mental abstraction when debugging complex programs rather than only considering contiguous chunks of the program. This hypothesis

1 <code>let</code> <code>n</code> = <code>readInt()</code>	1 <code>let</code> <code>n</code> = <code>readInt()</code>
2 <code>var</code> <code>sum</code> = 0	2 <code>var</code> <code>sum</code> = 0
3 <code>var</code> <code>product</code> = 1	3
4 <code>var</code> <code>i</code> = 1	4 <code>var</code> <code>i</code> = 1
5 <code>while</code> <code>i</code> < <code>n</code> {	5 <code>while</code> <code>i</code> < <code>n</code> {
6 <code>sum</code> = <code>sum</code> + <code>i</code>	6 <code>sum</code> = <code>sum</code> + <code>i</code>
7 <code>product</code> = <code>product</code> * <code>i</code>	7
8 <code>i</code> = <code>i</code> +1	8 <code>i</code> = <code>i</code> +1
9 }	9 }
10 <code>print</code> (<code>product</code>)	10
11 <code>print</code> (<code>sum</code>)	11 <code>print</code> (<code>sum</code>)

(a) Original program
(b) The slice with respect to `<sum, line 11>`

Figure 1.1: An example of a program slice

was tested by having expert programmers debug three programs and then testing the programmers' recognition of various contiguous and non-contiguous subsets of the program, some of which were slices with respect to the point of error ('relevant slices'). 20 participants were used to reach the conclusion of this study.

Weiser concludes that, especially in programs where the structure of the program is unclear, the programmer does recognise slices though he does not show that they recognise them more so than relevant contiguous chunks. He concludes that programmers do, on some level, use slices as an abstraction when debugging.

One can argue from this that accurate slices displayed explicitly in a development environment can only aid the debugging process. Since slicing is a mechanical task and requires keeping track of many points at once, cognitive load can be reduced by automating the process.

Program analysis

Slicing can also be used to detect certain classes of bugs automatically. Dead and unreachable code analyses are common diagnostic and optimisation analyses built in to compilers. Code that cannot possibly be reached at runtime or that is redundant (e.g. because every variable it defines is redefined before use) typically indicates a bug and so this makes for a useful warning to the programmer. Slicing can go one step further to ask whether every line in the program appears in the slice of some output statement (a program statement that prints to the console or otherwise emits a value to the end user). If there is no output statement whose slice contains a given line then that line does not contribute to the result of the program. Again, this may well be informative to the programmer.

Testing and Software Maintenance

We've so far only considered *backward* slicing i.e. the statements that can *affect* the values at the slicing criterion. We may also think about the *forward* slice i.e. the statements that can *be affected by* the criterion.

If a change is made to a source file at some point then it's desirable to run regression tests to ensure the system still functions as expected. The forward slice can be used to find only the tests that may be affected by the change. In large codebases, running all test cases becomes detrimental to productivity and so this pruning can be highly beneficial.

Code Quality Metrics

Weiser [20] suggested many novel code metrics that could be defined through the use of slices. Ott and Biemen [16] explored the measurement of the existing concept of 'cohesion' through the use of slices. Cohesion concerns the arrangement of modules within a codebase. A cohesive modularisation of a codebase is one where the components of a given module are contributing towards a common goal and inter-module communication is minimised[?]. We can also talk about 'functional cohesion', whether the code is split up into functions in such a way that calls between different functions are minimised. Both of these metrics were previously loosely and subjectively evaluated. Ott and Bieman use slicing as an abstraction for defining a concrete measure of functional cohesion.

1.1.2 Types of slicing

Considering the applications above we can conceptually split types of program slicing into several different dimensions.

Executable vs. Non-executable

The output of a program slicer is a subset of the instructions in the original program, intended to capture only the statements relevant to the slicing criterion. The question remains whether or not this subset should be able to compile and run independently of the rest of the program. Whether this is a desirable property or not depends on the application of the program slice. Consider the following example:

```
1 var x = 0
2 var y = 0
3 y = 1
4 x += y
5 print(x)
```

Taking a slice with respect to line 5, clearly the value assigned to `y` in line 2 is irrelevant to the final value of `x` but, in Swift, we need to retain the declaration of `y` if the resulting slice is to be executable. If the slice is being used for debugging purposes, we might argue that line 2 should be omitted from the slice since the value it defines is not relevant to whatever bug might have appeared at line 5. If the slice is being used for, say, extracting some piece of functionality then the slice needs to be executable and so we need to keep line 2.

Syntax-preserving vs. Amorphous

The above example also points to the benefit of being more flexible with our transformation. We've so far considered taking subsets of the original programs instructions

i.e. the only action our transformation can perform is statement deletion. With this restriction, the optimal executable slice with respect to line 5 is the entire original program. If we permitted the rewriting of declarations we could produce this slice:

```
1 var x = 0
2
3 var y = 1
4 x += y
5 print(x)
```

This avoids the unnecessary reassignment of `y`. This form of slicing where we permit other semantics-preserving transformations than deleting statements is known as ‘amorphous slicing’ [6]. Examples of other amorphous transformations which under certain conditions can preserve semantics of a slice include:

- Replacing an `if` statement with one of its clauses
- Removing a parameter from a function
- Replacing a tuple value with one of its constituent values

Static vs. Dynamic

If one is debugging a program and has found an unexpected value at some line, they may wish to look at the slice to find where the program is not meeting expectations. However in this scenario we have the additional information that in this particular run of the program with these specific inputs, the value went awry. That means we can consider only the statements that we depended on in *this specific execution* of the program, potentially narrowing down the search for the bug in the program significantly. This is known as ‘dynamic’ slicing as opposed to static slicing which does not consider the inputs to the program.

Intraprocedural vs. Interprocedural

It’s often useful to consider slices across procedure boundaries. i.e. one might want the slice to include not just the lines within the procedure that are used but also the relevant lines in other callee procedures. This is called *interprocedural* slicing. This is useful for a variety of application including automated testing, code quality metrics, and program analysis.

Forward vs. Backward

We have already discussed the idea of forward and backward slices in the context of automated testing. When we refer to ‘slices’, the backward form is typically implied.

For this project, the goal is to build a syntax-preserving, static, intraprocedural, backward slicer for Swift producing non-executable slices with the application of debugging in mind.

1.1.3 Approaches to slicing

At the high level, there are two prominent approaches to developing program slicers.

Data-flow Equations

Weiser’s original algorithm for generating program slices was based on a backward *data-flow analysis*. Data-flow analysis in general is a technique where we have some property of each statement in a program that we want to compute and we do so by initialising the values at each statement to some starting point, defining a relation between the values of a given statement and its preceding or following statements, and then using this relation over the whole program, updating the values at the nodes until the values reach a fixed point. For the case of slicing, Weiser’s data-flow equations compute the sets of *relevant variables* and then *relevant statements*.

Informally, the *relevant variables* to a statement are those which are defined before this statement and are eventually important in the evaluation at the *criterion*. The *relevant statements* are those which define a variable which is *relevant* in a statement that may be executed directly after it.

Dependence graphs Slicing can also be described as a reachability problem on a graph representation of the program. We construct a graph in which nodes are the statements of the program and the edges represent either *control dependencies* or *data dependencies*. Informally, a statement S_2 is control dependent on a statement S_1 if S_1 *controls* whether or not S_2 gets executed. For example, the statements within an `if` statement are control dependent on its predicate. A statement S_2 is data dependent on S_1 if S_1 defines a variable that S_2 references and there is a path of execution in which that variable does not get redefined between S_1 and S_2 . If we construct this graph, we get a *program dependence graph*. The slice with respect to any node is then the set of nodes backward reachable (using either type of edge) from that node.

1.2 What is Swift?

Swift is a general-purpose programming language designed and maintained by Apple and first released in 2014. It is designed with mobile and desktop application development in mind and largely used for those purposes. In late 2015 the Swift compiler was open-sourced along with the standard library.

1.2.1 Compilation stages

Swift source compiles to native code via LLVM (Low Level Virtual Machine) bitcode (see Fig 1.2). Static slicers exist for LLVM bitcode but no prominent slicer exists for Swift. This project aims to implement a static slicer for a significant subset of the Swift language and compare the precision, that is the lack of irrelevant statements, of the slices produced by slicing Swift source and compiling the results to LLVM and compiling the Swift source to LLVM and then slicing the results (see Fig 1.3).

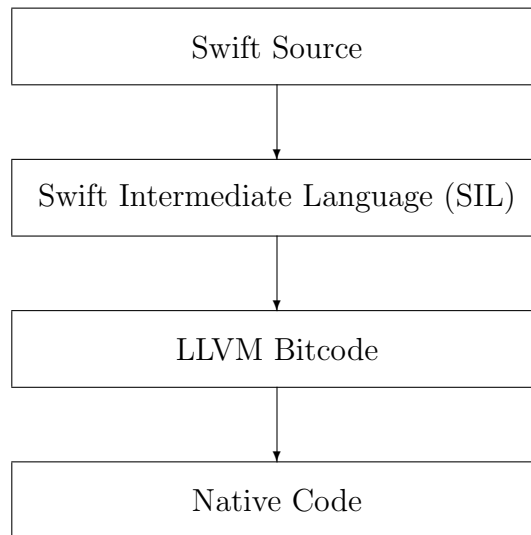


Figure 1.2: The Swift compilation pipeline.

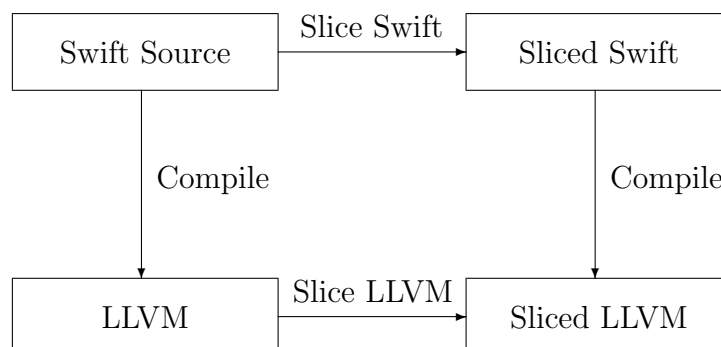


Figure 1.3: Two routes to obtaining the sliced LLVM of a Swift source file

1.2.2 Interesting semantics

As a modern programming language which has taken inspiration from multiple programming languages across multiple paradigms, Swift has a wide range of interesting features. Here I discuss a select few that are especially relevant to control flow and to slicing.

Exhaustive switch statements

`switch` statements will be familiar from other languages. In Swift the compiler will only allow switch statements that are exhaustive i.e. have at least one case statement that will match each possible value of the switch variable.

```
1 let num = Int(readLine())!
2 switch num {
3     case 1: print("That's a one!")
4     case 2: print("That's a two")
5 }
6 // error: switch must be exhaustive
```

The above is an example of a non-exhaustive `switch` statement and this code will not compile. This can be remedied by adding a `default` case. Swift has enumeration types which allow the user to declare a type with a discrete set of cases. The compiler is smart enough to recognise exhaustivity over enumeration cases:

```
1 enum CardSuit {
2     case hearts
3     case diamonds
4     case clubs
5     case spades
6 }
7 let mySuit = CardSuit.clubs
8 ...
9 switch(mySuit) {
10     case .hearts: print("That's a heart")
11     case .diamonds: print("That's a diamond")
12     case .clubs: print("That's a club")
13     case .spades: print("That's a spade")
14 }
```

In the case above, the compiler recognises exhaustivity and this code compiles successfully. In error-free programs, we can assume that at least one case of every `switch` statement is executed. This has implications for data dependency analysis. For example if every case of a `switch` statement assigns a value to a variable `x`, then a statement after the `switch` statement which only references `x` and no other variable cannot be data dependent on anything before the `switch` statement.

Never type

Swift defines a return type `Never` that can be used to indicate that a function will never return to its caller. This can either be because it will cause an unrecoverable

error or because it will run indefinitely. The most common use of this type is by the standard library `fatalError` function that will force a crash of the program, used for example in development when the program calls a function that is not yet implemented. Another example of this could be a server run loop:

```
1 func runWebServer() -> Never {
2     // Initiate web server...
3 }
```

This has implications for control dependency. Picture an `if` statement that calls a function with return type `Never` in its ‘then’ clause. Now not only are the statements within the `if` statement control dependent on the condition of the `if` statement but so are all statements *after* the `if` statement.

Control transfer statements

Swift has control transfer statements as will be familiar from other languages:

- **break** – Break out of the current loop and go to the next statement
- **continue** – Stop the current iteration of a loop and go to the next iteration
- **throw** – Throw a given error, propagating the error until it is handled or reaches the top level.
- **return** – (In a function) Return a value to the caller

There is also a `fallthrough` statement that is used to allow execution to fall through from one switch case to the next. Since it is possible to bind variables in a case statement, the compiler restricts the use of `fallthrough` to cases where control is being transferred to case statements that *do not* declare variables.

Swift also allows the user to label loops so as to indicate *which* loop a control transfer statement should apply. This is useful when we have nested loops:

```
1 rowLoop: for rowIndex in 0 ..< height {
2     colLoop: for colIndex in 0 ..< width {
3         // Symmetrical matrix so only fill in the bottom left corner
4         if colIndex > rowIndex { continue rowLoop }
5         // Populate cell...
6     }
7 }
```

Clearly each of these control flow mechanisms have implications for control dependency.

Guard statements

A **guard** statement is similar to an `if` statement in that it has a boolean expression condition. Rather than defining a ‘then’ and an ‘else’ clause, a **guard** statement contains only an `else` clause that is run if the condition is not met. If the condition is met then execution continues in the current scope. The compiler checks that program control is transferred out of the current scope after the `else` clause through use of either a control transfer statement or a function that returns the `Never` type.


```

1 func divide(dividend: Int, divisor: Int) throws -> Int {
2     guard divisor != 0 else { print("Can't divide by zero") }
3     ...
4 }
5 // This will not compile

```

In this example, the `else` clause prints a message to the user but does not transfer control to exit the scope (the function in this case) so this code does not compile. The following is an example that does transfer control to exit the scope and does compile:

```

1 func divide(dividend: Int, divisor: Int) throws -> Int {
2     guard divisor != 0 else { throw Error.divisionByZero }
3     ...
4 }

```

This requirement means that anything in the same scope as a `guard` statement that comes after the `guard` statement will be control dependent on the `guard` condition.

Mutating functions on value types

Swift makes a distinction between ‘value types’ and ‘reference types’ i.e. those that are passed by value and those that are passed by reference respectively. `structs` are an example of a value type in Swift. One is able to define mutating methods on value types, methods that change the value of the instance. These must be explicitly annotated as mutating with the `mutating` keyword:

```

1 struct IntStack {
2     private var backingStore: [Int] = []
3     mutating func push(element: Int) {
4         backingStore.append(element)
5     }
6     mutating func pop() -> Int? {
7         return backingStore.popLast()
8     }
9 }

```

This information is useful for slicing since we can efficiently determine if a call to a method can mutate the instance or not.

Computed properties

Sometimes it is useful to describe a property in terms of other properties of an object. For example, a rectangle’s centre could be defined in terms of its upper-left-corner position and size. This can be achieved with a function rather than a property in most object oriented languages. If however we knew where we wanted to move the centre of a rectangle to, it would be useful to describe that movement in terms of its centre only rather than in terms of the position of its upper-left corner. This is one scenario where computed properties come in useful. The following example is reproduced from ‘The Swift Programming Language (Swift 4.1)’:

```

1  struct Point {
2      var x = 0.0, y = 0.0
3  }
4  struct Size {
5      var width = 0.0, height = 0.0
6  }
7  struct Rect {
8      var origin = Point()
9      var size = Size()
10     var center: Point {
11         get {
12             let centerX = origin.x + (size.width / 2)
13             let centerY = origin.y + (size.height / 2)
14             return Point(x: centerX, y: centerY)
15         }
16         set(newCenter) {
17             origin.x = newCenter.x - (size.width / 2)
18             origin.y = newCenter.y - (size.height / 2)
19         }
20     }
21 }
22 var square = Rect(origin: Point(x: 0.0, y: 0.0),
23                   size: Size(width: 10.0, height: 10.0))
24 let initialSquareCenter = square.center
25 square.center = Point(x: 15.0, y: 15.0)
26 print("square.origin is now at (\\(square.origin.x), \\(square.origin.y)
27 // Prints "square.origin is now at (10.0, 10.0)"

```

The presence of computed properties means that setting or even accessing properties could result in multiple other stored properties being modified.

Optional binding

Swift has the notion of ‘optional’ values. Values that are either of a given type or are `nil`. The compiler forces us to consider both the value case and the `nil` case. For example, the built in `Int` type has an initialiser that takes a string and returns an `Int?` (an *optional* `Int`). If the string is composed only of digits and can be represented by an integer, it will have an `Int` value, otherwise it evaluates to `nil`. If we try to use this value directly then we see an error:

```

1  Int("3") + 1
2  // error: value of optional type 'Int?' not unwrapped

```

One way to handle the two cases is through the use of ‘optional binding’ where we check if an optional value is `nil` and ‘unwrap’ its value in one step:

```

1  if let userInput = Int("3") {
2      print(userInput + 1)
3  } else {

```

```
4     print("Please input an integer")
5 }
```

This code is saying: ‘If the result of `Int("3")` is not `nil` then bind the `Int` value to `userInput` and enter the then clause, otherwise enter the else clause’. `userInput` is only in scope in the then clause.

Chapter 2

Preparation

2.1 Understanding slicing approaches

In this section I go in to detail on the two main approaches to slicing, data-flow equations and dependence graphs.

2.1.1 Control Flow Graphs

A concept critical to both approaches is that of the *control flow graph* (CFG). A control flow graph describes the possible sequences of execution of a program. The control flow graph of a program p is a graph in which the nodes are statements from p and the edges are directed and go from a statement, S_1 , to another, S_2 , if S_2 can be run directly after S_1 in the execution of p . Fig 2.1 presents an example of a control flow graph. Notice that every node in the CFG is reachable from the **START** node and the **END** node is reachable from every node. This property is often assumed when working with CFGs. The literature refers to programs that meet this condition as ‘well-formed’. Typically it would be the role of the compiler to catch programs that don’t meet this ‘well-formed’ condition. Notice also that in this case we will never enter the ‘hello’ clause at runtime since x is hardcoded to be 2 and so will never be greater than 3. Our CFG construction is purely *syntactic*, it does not consider arithmetic or logical identities that could lead to smaller graphs. Whilst we could make our CFG construction arbitrarily better at dealing with this, in general it will not be possible to decide whether a branch could ever be taken at runtime.

Control flow graphs can be constructed directly from the AST in a single traversal. Which types of statements to consider a node and which to break down further into constituent parts is dependent on application and implementation. This issue is revisited in the ‘Implementation’ section.

2.1.2 Dependence Graph approaches

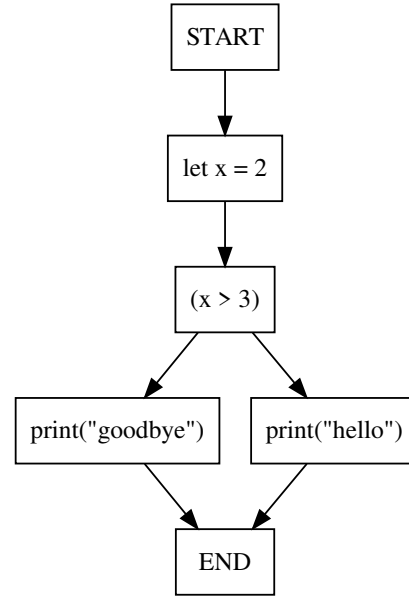
The majority of recent work on slicing algorithms uses dependence graph-based approaches. This involves constructing a graph in which nodes are program statements and edges are ‘dependences’.

```

1  let x = 2
2  if (x > 3) {
3    print("hello")
4  } else {
5    print("goodbye")
6  }

```

(a) Original program



(b) The CFG of the program

Figure 2.1: An example of a control flow graph

As well as the control flow information, we also require an indication of the variables that are referenced and defined at each node x . We call these sets $\text{REF}(x)$ and $\text{DEF}(x)$ respectively. For now, we consider only ‘scalar’ variables, that is, variables that hold only one value at a time. For arrays and objects we could once again get arbitrarily clever about how fine-grained our REF and DEF approximations are, for example we could treat each index of an array as a separate variable, but since we can dynamically index into an array at runtime it is not going to be possible to precisely determine the set of indices that an assignment may affect.

Program dependence graphs directly represent *data dependencies* and *control dependencies*. We can define these dependencies formally in terms of the CFG, $\text{REF}(x)$ and $\text{DEF}(x)$:

Data dependency

A node y is data dependent on a node x iff there exists a variable v such that:
 $v \in \text{DEF}(x)$,
 $v \in \text{REF}(y)$,
 and there exists a path (x_0, x_1, \dots, x_k) in the CFG with $x_0 = x, x_k = y$ and for all $0 < i < k, v \notin \text{DEF}(x_i)$.

That is, there is at least one path in the CFG from x to y where v is not redefined.

Control dependency

In the control flow graph of Fig 2.1b, the only control dependencies are that of the print statements on the if condition. All other statements will be run in every possible execution of the program. With this intuition, we can formalise the definition of control dependency.

First we define *postdominance* in the CFG. A node y postdominates a node x iff every path from x to the **END** node contains y . Note that this definition means every node postdominates itself and that the **END** node postdominates every node.

A node y is control dependent on a node x iff y postdominates at least one of x 's children in the CFG but *does not* postdominate x . The set of nodes in a CFG that a node is control dependent on is otherwise known as its ‘postdominance frontier’, the set of nodes where its postdominance ends.

Now a program dependence graph is constructed as follows:

- Define the set **NODES** as the nodes of the CFG subtracting the **START** and **END** nodes. No node is control or data dependent on either the **START** or **END** nodes.
- Define the set **DATADEPEDGES** as $\{(x, y) | y \text{ is data dependent on } x\}$ (note the direction)
- Define the set **CONTROLDEPEDGES** as $\{(x, y) | y \text{ is control dependent on } x\}$

These three sets describe a program dependence graph. Note that this graph is not necessarily connected. Some implementations will choose to include the **START** node and make every node control dependent on the **START** node. For the purpose of slicing, this is not an important difference.

Graph-based approaches to slicing tend to define their slicing criteria simply as a node in the PDG (i.e. a statement in the program). To find the slice with respect to a given node, we find all backward-reachable nodes from that node:

$$\text{PARENTS}(y) := \{x | (x, y) \in \text{DATADEPEDGES} \vee (x, y) \in \text{CONTROLDEPEDGES}\}$$

$$\text{BACKWARDSREACHABLE}(y) := \text{transitive closure of PARENTS}$$

With these definitions, the set $\text{BACKWARDSREACHABLE}(y)$ contains all the nodes that belong in the slice for y . Notice that this definition of a slicing criterion does not match up with our previous definition which also included a set of variables that the user is interested in. Modifications to this technique exist to deal with criteria which also specify sets of variables of interest. Given a criterion node, all nodes with definitions that reach this node can be found. The slice is then the union of the backward reachability of all of these nodes.

2.1.3 Data-flow Equations

Recall that our slicing criterion C , in the original sense, consisted of a statement n and a set of variables V i.e $C = (n, V)$. In the data-flow approach, we consider the set $\text{RELVAR}_C(y)$ (‘relevant variables’) of variables whose value *on entrance* to statement y can influence the value of one of the variables in V at n through some chain of assignments. Consider the following example:

```

1  var x = 1
2  var y = x
3  var a = 2
4  var z = y + x
5  print(z)

```

In this simple case, for criterion $(5, \{z\})$ we would say:

- Only the variable z is relevant at line 5. This is the criterion statement and z is the only variable in the set of criterion variables.
- The variables x and y are relevant at line 4 since they are both used here to define z which is relevant at a direct CFG successor (line 5). Recall that we are interested in the variables that are relevant on the entrance to the statement so z is not relevant at line 4.
- The variables x and y are still relevant at line 3 since they are both relevant at a direct CFG successor and are not redefined by line 3.
- Only the variable x is relevant at line 2. It is used here to define y which is relevant at a direct successor (line 3).
- No variable is relevant at line 1 since there are no variables defined on entrance to line 1.

Notice in this example that we have not considered any branching statements, statements that have more than one CFG successor like an `if` condition or a `while` condition. This will be addressed in a moment. For now we consider only the ‘direct’ relevance and we will denote this with a superscript 0 to indicate 0 levels of indirectness e.g. $\text{RELVAR}_C^0(n)$. We can now define RELVAR_C^0 with data-flow equations. Intuitively, $\text{RELVAR}_{(n,V)}^0(n)$ must include all the variables in V so we use this as our initialisation point, with all other RELVAR sets starting empty. We then take the least fixed point of the equation:

$$\text{RELVAR}_C^0(y) := \bigcup_{s \in \text{Succ}(y)} (\{v \mid v \in \text{RELVAR}_C^0(s) \wedge v \notin \text{DEF}(y)\} \cup \{v \mid v \in \text{REF}(y) \wedge \text{DEF}(y) \cap \text{RELVAR}_C^0(s) \neq \emptyset\})$$

Now define the set RELSTAT_C^0 (‘relevant statements’) of *statements* y that define a variable that is directly relevant at a CFG successor of y . Once we account for branching statements, the RELSTAT_C will become our set of statements in the slice.

These directly relevant statements are all that we need for ‘straight-line’ programs with no branching but won’t suffice for even the simplest of examples with branching:

```

1  var pred = readBool()
2  var x = 0
3  if pred {
4    x = 1
5  }
6  print(x)

```


With the criterion $(6, \mathbf{x})$, the direct relevance equations will not pick up the definition of **pred** as a relevant statement, despite it clearly affecting the value of \mathbf{x} .

The set RELBRANCHES_C^0 is defined as the set of branching statements that influence whether a directly relevant statement is executed (i.e. the relevant statement is control dependent on the branching statement). Now we can define each of these sets for higher levels of indirectness.

$$\text{RELVAR}_C^{k+1}(y) := \text{RELVAR}_C^k(y) \cup \bigcup_{b \in \text{RELBRANCHES}_C^k} \text{RELVAR}_{(b, \text{REF}(b))}^0(b)$$

$$\text{RELSTAT}_C^{k+1} := \text{RELBRANCHES}_C^k \cup \{i | j \in \text{SUCC}(i) \wedge \text{DEF}(i) \cap \text{RELVAR}_C^{k+1}(j) \neq \emptyset\}$$

$$\text{RELBRANCHES}_C^k := \{i | j \text{ is control dependent on } i \wedge j \in \text{RELSTAT}_C^k\}$$

The fixed point of RELSTAT_C is the slice for the criterion C .

2.1.4 Discussion and comparison of approaches

There is a striking overlap between these two approaches. Both require the control dependency analysis to be pre-calculated. This is itself an interesting task and is discussed at length in section 3.4.2. Both also assume that the control flow information is available though they don't necessarily require building a CFG as an intermediate step. Steenkamp demonstrates in [18] that a performant slicer can be built without using an intermediate CFG.

A modification to the data-flow approach presented here can compute a slice in $O(v \times n \times e)$ where v is the number of variables in the program, n is the number of nodes in the CFG, e is the number of edges in the CFG. The program dependence graph approach can compute a slice in $O(e \times n + E)$ where E is the number of edges in the PDG. In the intra-procedural slicing case, we can expect to be dealing with numbers of nodes that mean that minor asymptotic complexity improvements are unimportant compared to reducing the constant overhead. See Tip [19] for a thorough survey of complexity analyses of various techniques.

The program dependence graph approach is computing more information since it does not narrow its computation to the criterion values. In fact, since the program dependence graph is criterion-agnostic, once created it can be sliced by many different criteria. In the case of debugging, where the clearest use-case is finding the slice of a statement which has an incorrect value and trying to find all the places where that bug could have been introduced, it's likely that only one criterion would be used. It is conceivable however that a user may narrow the search iteratively by slicing with respect to one of the nodes that is in the slice of the erroneous statement (which is necessarily a subset of the slice of the erroneous statement). In this case, the criterion-agnostic construction of the PDG could provide a performance benefit.

The precision of graph-based slicing and data-flow-based slicing are equivalent for the simple case of intra-procedural slicing with only scalar variables [19]. We consider only intra-procedural slicing with scalar variables for the purposes of this project.

Program dependence graphs can be used for tasks other than slicing such as program differencing, program integration [7], and various optimisations [5]. The data-flow equations are computing a more specific set of information.

The program dependence graph is an approach that can be easily interpreted visually. This makes program dependency graph approaches arguably easier to implement since errors can be inspected more easily.

Both approaches are extensible to more complex language features such as non-scalar variables (e.g. arrays, tuples) and pointers [9] [1] as well as to *inter*-procedural slicing [2] [8].

2.2 Understanding Swift

In this section I will discuss some of the Swift tooling relevant to slicing.

2.2.1 The compiler

The Swift compiler, `swiftc` takes Swift source files and emits executable files along with a package of debugging symbols (in a `.dSYM` directory) which includes mappings from mangled names to original symbol names and mappings from executable statements to source statements. It also has modes that allow for the dumping of the internal abstract syntax tree or the emission of intermediate representations include Swift Intermediate Language and LLVM bitcode.

An abstract syntax tree (AST) representation of programs is required for slicing. The Swift compiler, with the compiler flag `dump-ast`, prints a text representation of the internal AST. This unfortunately lacks some key information for use in analysis. For instance, nodes are tagged with source ranges but the end points of the ranges they are tagged with are the *starting points* of the last *token* relevant to that node. This points to the fact that this AST dump is intended for internal debugging and not particularly appropriate for parsing for other purposes. One might consider working with the internal AST directly rather than via this serialised format and indeed that would be a sensible option if the goal were to implement a slicer natively within the compiler. Learning the compiler architecture for a production language is outside of the scope of this project as the goal is to produce a standalone slicing tool for Swift.

2.2.2 SourceKit

Swift is a young language and thus far tooling outside of Apple's Xcode IDE has not been a key focus. That said, there does exist a first-party library called SourceKit that aims to aid IDEs in supporting languages including Swift. The functionality it provides includes:

- Code completion

- Documentation generation
- ‘Cursor information’ – This includes information about the entity at a given offset in a Swift file including its name and type
- Index – This includes high-level information about a file including various structural elements such as functions and `if` statements

Whilst SourceKit provides the high-level structure of programs such as the location of structured control flow statements like `if` statements, it does not provide much insight into unstructured control flow statements such as `break` statements nor does it provide locations or types of expressions. The information provided by SourceKit could not prove a substitute for a full AST. Cursor information and index information proved useful for determining REF and DEF sets.

2.2.3 dg

‘dg’ is a dependence graph and inter-procedural program slicer for LLVM bitcode [3]. The dg slicer takes bitcode as input and an LLVM function name as the criterion. It will then find the slice with respect to all call sites of the given function. It is typically recommended to slice with respect to assertions (with the LLVM function `__assert_fail`).

Whilst the Swift compiler does allow emission of the intermediate bitcode (with the flag `-emit-bc`), this complicates the embedding of debugging information. Swift also uses a modified version of LLVM. This makes comparing a source-level Swift slicer to dg somewhat more of a challenge than expected. This will become an extension goal and my primary evaluation technique will compare to the baseline of slicing by eye. This will give a better idea of precision of my slicer.

2.2.4 swift-ast

A third-party project named ‘swift-ast’ [17] has endeavoured to provide an AST representation of Swift source programs that is appropriate for program analysis and transformation. The AST produced represents directly what is present in the source file and no reductions or optimisations are immediately performed. Information is embedded to locate each AST node in the source program. Since swift-ast does not use the Swift compiler, the results will be more fragile in the sense that any changes in the compiler will have to be reflected in this project for results to remain accurate. That said, swift-ast is built with source-level analysis projects in mind and so can provide a solid starting point for this project.

2.3 Choice of slicing approach

I have chosen to implement the slicer using the program dependence graph approach for its flexibility and interpretability. The difference in the form of the criteria (where data-flow slicing criteria include a set of variables and PDG slicing criteria do not) should not

make much practical difference in the case of debugging. In what follows, slicing criteria will be assumed to be PDG nodes.

2.4 Development Environment and Process

This project has been developed under the Git version control system. Development was segmented into cohesive chunks of work and committed with self-contained messages for ease of browsing the history of the project.

Xcode was used as a development environment when appropriate for its good Swift support. Sublime Text was also used for speed when Xcode's features weren't required.

Builds were managed by Swift Package Manager and tests were written using the XCTest unit testing framework. The project makes use of both unit tests for verifying each part of the implementation and integration tests to verify the high-level behaviour of the slicer.

Development work was done primarily on my personal computer which is regularly backed up to an external local disk. The project itself is also hosted remotely on GitHub and updated with every significant modification to the project. Travis CI was used to run the test suite on the remote repository to ensure that the remote repository is kept in a stable state.

Development was done according to the 'spiral model'. Topics of particular uncertainty were addressed as priority and a working product was iterated on to support new language features as development proceeded.

Chapter 3

Implementation

In this chapter I'll describe the various steps of producing a slicer for Swift, the implementation details and algorithms used, and practical difficulties.

3.1 Retrieving the AST

To appropriately analyse the source programs, access to an abstract syntax tree representation of the program is required. This was surprisingly difficult to obtain from the Swift compiler directly and so a third party tool ('swift-ast', See 2.2.4) had to be used.

I extended swift-ast with a line/column to character offset translation so as to be compatible with SourceKit which deals with . I also added a utility for easily viewing the related source file content for each AST node. This is necessary for providing the slice to the user.

3.2 Constructing the CFG

I implemented a bottom-up construction of the CFG. I endeavoured to handle control flow such as **break** and **continue** statements. To this end, in my construction of the CFG, I distinguish between a 'partial' and a 'complete' CFG. A complete CFG is a CFG as described above. A 'partial' CFG does not have a **START** node (but does have a node that is labelled as the entry point) and rather than pointing directly to other nodes, edges may point to **break** or other 'placeholder nodes' which will have to be resolved to point to a real node later in the construction.

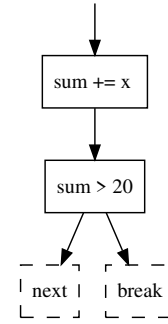
Fig 3.1 shows an example of these placeholder nodes in use. I construct my CFG in a bottom-up fashion. I perform a (depth-first) traversal of the AST. If I reach a statement that has a non-trivial control flow structure such as a **for** statement, the following steps are necessary:

- Recursively get the partial CFGs of the statement's children
- Add the additional nodes necessary for this particular structure (e.g. a condition node)

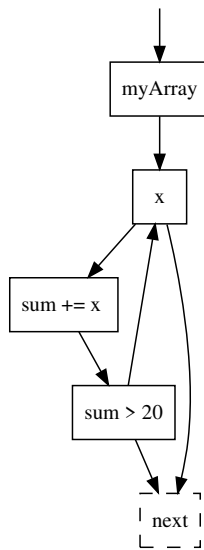
```

1  for x in myArray {
2      sum += x
3      if sum > 20 { break }
4  }
```

(a) A snippet featuring ‘semi-structured’ control flow



(b) The partial CFG of the *contents* of the **for in** statement



(c) The partial CFG of the whole snippet

Figure 3.1: An example of a partial CFG

- Add edges from these structural nodes according to the semantics of Swift (e.g. the condition of an **if** statement could either transfer control to the then clause or the else clause)
- Resolve placeholder nodes according to the semantics of Swift (e.g. a **break** placeholder should be resolved to a **next** placeholder when constructing the CFG of a **while** statement from the partial CFGs of its children)

In the specific case of the **for in** statement in Fig 3.1:

- A node is added to represent the evaluation of the sequence expression
- A node is added to represent the binding of **x** to an element of the sequence which can either succeed (if there are remaining elements of the sequence to bind) or fail (if there are none).

- For the success case, an edge is added from the binding node to the entry point of the partial CFG of the body of the `for in` statement.
- For the failure case, an edge is added from the binding node to the `next` placeholder (since if there is nothing left to bind to the `x`, execution should continue to the next statement).
- Any `break` placeholders in the child CFG are resolved to the `next` placeholder (since if we break out of a `for in` loop, execution should continue to the next statement).
- Any `next` placeholders in the child CFG are resolved to point to the binding node so that we can bind the next sequence element if one exists.

Results of this construction can be seen in Fig 3.6b. Similar constructions are provided for all supported control flow structures (`for in`, `guard`, `if`, `repeat while`, `switch`, `while`).

Whenever a statement with trivial control flow is encountered in the traversal, its CFG has only one node which is the statement itself which is marked as the entry point and has an edge from that node to the `next` placeholder. My implementation considers any expression to have ‘trivial control flow’. This is a source of imprecision when the expressions are actually rather complex, for example when the expression includes a long closure.

Finally, when a partial CFG has been constructed for each top-level statement, they are chained together as expected to produce a complete CFG. A `START` node is made to point to the entry point of the first partial CFG. Any `next` placeholders of a given partial CFG are made to point to the entry point of the next partial CFG if there is one and an `END` node otherwise. Any remaining, unresolved, placeholder nodes are reported to the user and an error is thrown. This can happen if, for example, a `fallthrough` statement is used outside of a `switch` statement. An example result is shown in Fig 3.2.

In total this construction performs a single depth-first traversal, visiting each statement once and so takes time linear in the number of statements. The nodes and edges of child CFGs are reused (rather than copied) by parent CFGs so each node and edge is only created and stored once in the construction. Thus, memory complexity is $O(|\text{NODES}| + |\text{EDGES}|)$.

3.3 Determining Ref and Def sets

One of the drawbacks of using `swift-ast` for the AST is that type information is not embedded. It is not known whether a certain variable is a reference to an object or is of a value type. If the reference is to an object, function calls can mutate the object. If the variable is of a value type, the variable will have been marked `var` as opposed to `let` and any functions that mutate it will have this information embedded in their type information.

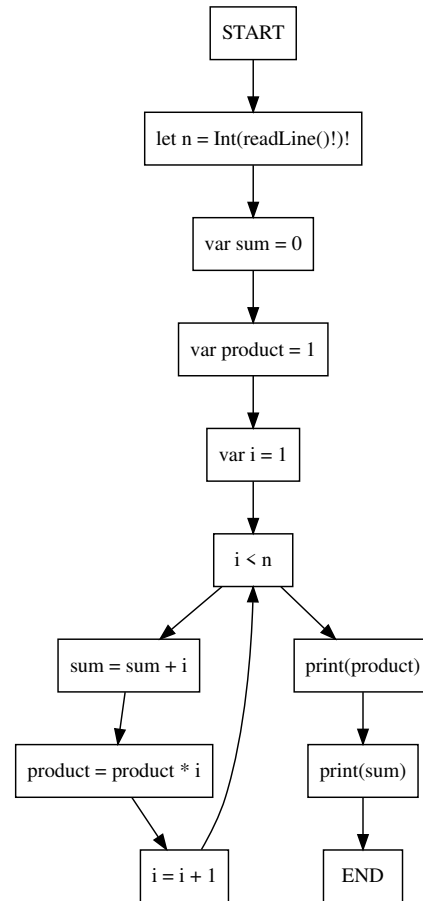
To get around this, my slicer makes a coarse approximation to the REF and DEF sets by using the assumptions that (a) variables are only defined in assignment operations,

```

1  let n = readInt()
2  var sum = 0
3  var product = 1
4  var i = 1
5  while i < n {
6      sum = sum + i
7      product = product * i
8      i=i+1
9  }
10 print(product)
11 print(sum)

```

(a) A simple Swift program



(b) The CFG of this program produced by Caramel

Figure 3.2: An example of Caramel’s CFG construction.

(b) all symbols appearing in a statement that aren’t the symbol/symbols being defined are being referenced, (c) all variables are scalar values. These assumptions break down in some of the situations that follow:

```

1  var str = "Hello"
2  str.append(", world!")

```

Assumption (a) breaks down here since we use a mutating function on `str`. This is a source of incorrectness in my slicer in the presence of mutating functions (an unsupported feature). A potential way around this would be to find a way to extract the type information to be able to distinguish mutating from non-mutating functions. Unfortunately, this still would not suffice for reference types (classes) since these do not have annotated function types. Extracting the type information was outside the scope of this project.

```

1  var n = 5
2  var x = 0
3  for i in 1 ..< n {
4      x += 1
5      n += 1
6  }

```



```
7 print(x)
```

Assumption (a) breaks down in the case of `for` in statements. Here the expression “`1 ..<n`” defines a sequence that is then iterated over in the `for` loop. That is to say that the identifier `i` should be data dependent on the result of the sequence expression but under these assumptions, the slicer would not recognise either that the sequence expression defines a value or that the identifier is data dependent on that value. In particular this would result in the slice at line 7 not including line 1 because of the missed dependency. A naive solution to this might be to make a single CFG node to represent both the identifier `i` and the sequence expression. This would lead to imprecision. Since Swift evaluates the sequence expression first, the subsequent assignments to `n` within the body of the `for` loop do not change the sequence that is being iterated over and thus the slice at line 7 should not contain line 5. This issue was resolved in Caramel by creating a unique ID for the value created and artificially adding this ID to the DEF set of the sequence node and the REF set of the identifier node. A similar issue exists for `switch` statements which was solved similarly.

```
1 var arr = [1, 2, 3]
2 arr[2] = arr[1] * 2
```

Assumption (c) breaks down here since arrays are not scalar values, rather they are a structure of many values. In this case, the slicer will see that an assignment is being performed on `arr` and see that `arr` is being referenced but will not attempt to understand which indices are being referenced. This is a source of imprecision in my slicer.

```
1 var arr = [1, 2, 3]
2 arr[2] = 4
```

Assumption (c) can also lead to incorrectness in the presence of non-scalar values. Here, line 2 will be seen as overwriting the value of `arr` where it is actually only overwriting the value at a single index. This might lead to sliced excluding line 1 when it should actually be retained. The use of non-scalar values is permitted in my slicer as long as assignments are made to the entire structure at once and with the caveat that referencing indices of a non-scalar value will lead to imprecision.

With these assumptions, I define for each expression a ‘reference range’ and a ‘definition range’. For assignment expressions, the definition range is the left hand side of the assignment operator and for other expressions it is empty. I then use SourceKit to scan these ranges for symbol references. It will return symbol names in their mangled form to avoid issues of shadowing or overloading. Note that sometimes the reference and definition ranges will overlap:

```
1 var x = 40
2 x += 2
```

Here `x` is being both referenced and defined on line 2. My slicer supports all the standard library assignment functions. The definition range for these is the left hand side of the operator and the reference range is the whole expression.

3.4 Constructing the PDG

3.4.1 Determining data dependences

The problem of determining data dependences is more widely known as the problem of finding ‘define-use chains’ (or, equally, ‘use-define chains’). The approach I take is that of computing the set of reaching definitions for each node n , the set of nodes whose definitions may reach n .

This can be done using a standard data-flow approach. We say a variable definition from node x ‘reaches’ a node y if there is a path from node x to node y in which that variable is not redefined. The approach maintains a set of definitions that reach the start of a node n , $\text{REACHIN}(n)$ and a set of definitions that reach the end of a node n , $\text{REACHOUT}(n)$. $\text{REACHIN}(n)$ is defined as the union of $\text{REACHOUT}(x)$ over its predecessors, x . $\text{REACHOUT}(n)$ is defined as $\text{REACHIN}(n)$, subtracting any of those definitions that are overwritten by n . This process is iterated until the results converge.

Then, the set of data dependencies is found for a node n by finding each reaching definition that defines a variable in $\text{REF}(n)$.

Each iteration requires the union of the REACHOUT sets of all predecessors of every node. Given that the union of two sets of size m and n takes $O(\min(m, n))$ time, for all nodes with more than one incoming edge we have to form the union in $O(r)$ where r is the number of reaching definitions in the predecessor with the fewest reaching definitions. Given that r is bounded by the number of definitions in the program d , and that the number of nodes with more than one incoming edge is bounded by N , the number of nodes in the graph, we have that an iteration is bounded by $O(Nd)$. If we assume that there is at most 1 definition at each node (which turns out not to be the case in general in Swift) then we see a worst case of $O(N^2)$ time required for each iteration.

Assuming the DEF set is precomputed for each node, and assuming only one definition at each node, subtracting the definitions takes constant time for each node and $O(N)$ time overall. Finally, when the reaching definition have been computed, the intersecting this with REF for each node takes $O(k)$ where k is the number of reference in the program in total. Thus the total time complexity for an iteration is $O(N^2)$.

In the worst case, the number of definitions grows linearly with the length of the program and so the number of data dependencies can grow quadratically with the size of the program:

```

1  var x = 0
2  if true { x += 1 }
3  if true { x += 1 }
4  ...
5  if true { x += 1 }
6  print(x)
```

In this example, the statements in the body of each if statement are data dependent on those in all previous if statements and the declaration of x , thus we have a number of data dependencies quadratic in the number of lines of the program.

3.4.2 Determining control dependences

We have already discussed the definition of control dependence in terms of the CFG by the postdominance frontier in 2.1.2. Classical iterative data-flow approaches to the problem of finding postdominance frontiers involve maintaining a matrix of booleans indicating whether a node is postdominated by another. This matrix is initialised to be the identity matrix (since by definition each node postdominates itself) and is updated iteratively with the understanding that if a node n_2 postdominates all children of a node n_1 then n_2 also postdominates n_1 . This is a backwards data-flow analysis and eventually converges on a matrix representing the postdominance relation. Another pass over this matrix can find us the postdominance frontiers. If a node n_2 does *not* dominate a node n_1 but does dominate one of its children, n_1 is in the postdominance frontier of n_2 .

Lowry and Medlock observe in [14] that if a node z is dominated by two other nodes x and y then either x dominates y or vice versa. This leads to the idea of an *immediate* dominator of a node z , the unique dominator of z that is not itself dominated by any other dominator of z . With this fact, we can represent the dominators in an entire graph more efficiently than our previous matrix representation by a *dominance tree* in which we relate each node to its immediate dominator. Since postdominance is simply dominance in the reversed CFG, any results regarding dominance in arbitrary flow graphs also apply to postdominance. Fig 3.3 gives an example of a *postdominance tree* for a CFG.

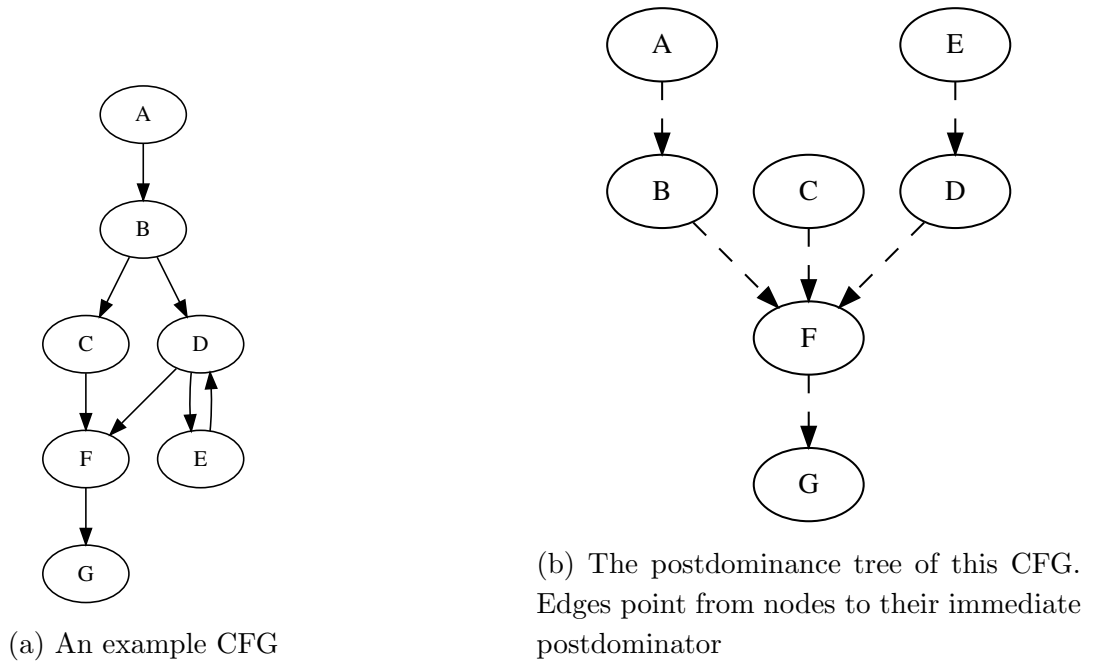


Figure 3.3: An example of a postdominance tree.

Langauer and Tarjan describe an algorithm in [13] which finds the dominance tree of a CFG in $O(e \cdot \alpha(e, n))$ time, where e is the number of edges in the CFG, n is the number of nodes in the CFG $\alpha(m, n)$ is a functional inverse of Ackermann's function, that is to say that its value is 'almost constant', making the algorithm overall 'almost linear'. This algorithm is widely implemented and used for the finding of dominance frontiers.

Cooper, Harvey and Kennedy observe in [4] that, in practice, it requires CFGs with on the order of 30,000 nodes before the asymptotic advantage of the algorithm in [13] catches up with a well-engineered iterative approach. They claim that this size of CFG is unreasonably large. To expand upon ‘well-engineered’ they present a specific implementation of the iterative approach and justify its performance. I implemented their approach and I reproduce here the implementation of [4], rephrased for the finding of postdominance rather than dominance.

- Start by performing a backwards postorder depth first traversal of the flow graph starting from the **END** node, assigning numbers to each node. That is to say, the **END** node should have the highest number. See Fig 3.4 for an example. The iterative re-evaluation will be done in reverse order of this numbering as per the popular data-flow formulation of Kam and Ullman in [10]. Following this formulation allows us to make some claims about correctness and performance later on.

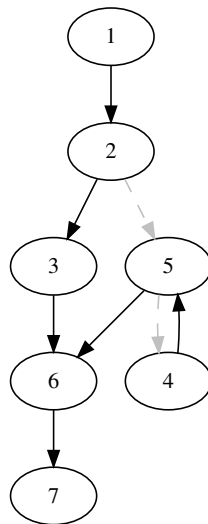


Figure 3.4: A depth-first traversal postorder numbering of the CFG in 3.3a. Dark edges are those in the depth-first spanning tree. Dashed edges are edges from the CFG not in the depth-first search tree.

- The dominance tree, represented as a map `ipdoms` from node to immediate postdominator, should be initialised such that each nodes estimated immediate postdominator is undefined except for the **END** node whose estimated immediate postdominator should be initialised to itself.
- *Observation:* In the depth-first spanning tree (DFST), there is a path from every node n to the **END** node. Since all postdominators of n will appear on this path by definition, we know that any postdominator of n has a numbering greater than or equal to n .
- Define a function `commonPostdominator(n, m)` that aims to find the closest common postdominator of two nodes with numbers n and m . We implement this by ‘walking’ down the paths from n to **END** and from m to **END** and finding the point where they

meet (which does exist since at the very least both paths end at the same node). Start two ‘fingers’: $\text{finger1}=n, \text{finger2}=m$. Whilst $\text{finger1} \neq \text{finger2}$, with our observation above, we know that any common postdominator of n and m will be strictly greater than $\min(\text{finger1}, \text{finger2})$ so advance the lower finger one step along the tree. For example: $\text{finger2} := \text{ipdoms}[\text{finger2}]$.

- Now we can define an iteration. In order of descending node number, set the updated immediate dominator estimate for each node n to be the common postdominator of all CFG successors s_1, s_2, \dots, s_k of n .
- Keep performing iterations until nothing changes. The algorithm will now have converged on the correct postdominance tree.
- To find the nodes whose postdominance frontier a node n is in (equivalently, the nodes that are control dependent on n) one simply walks the DFST from each CFG successor of n towards the END node, recording each node touched until the immediate postdominator of n is reached.

This process is summarised in the pseudocode of Fig 3.5.

To assess the correctness and algorithmic complexity we turn to Kam and Ullman.

We first observe that the postdominance equations form a *distributive* data-flow problem in the sense of [10]. That is to say we can define a function, f_b , for each node b such that:

$$\text{out}(b) = f_b\left(\bigcap_{x \in \text{succ}(b)} \text{out}(x)\right)$$

and f_b has the distributive property:

$$f_b(x \cap y) = f_b(x) \cap f_b(y)$$

For the postdominators problem, we find the following f_b

$$f_b(x) := x \cup \{b\}$$

Which says that the postdominators of a node are the nodes that postdominate all of its children plus b itself. We see that f_b has the necessary distributive property:

$$f_b(x \cap y) = (x \cap y) \cup \{b\} = (x \cup \{b\}) \cap (y \cup \{b\}) = f_b(x) \cap f_b(y)$$

This result allows us to use the generalised framework of Kam and Ullman to prove that this algorithm converges upon the correct result and does so in a number of iterations linear in the *loop-connectedness* of the flow graph. Loop-connectedness is a property of a flow graph which is calculated with respect to a DFST, T , by separating the edges of the graph into ‘forward’ edges that respect the order of T , back edges that go back up the tree, and cross-edges which cross branches of T . Loop-connectedness then is the maximum number of back edges in any cycle-free path of the graph. This is hard to intuit but is bounded by a related quantity called the derived sequence length which can be thought of

intuitively as the maximum loop nesting depth in the graph. With a loop-connectedness number lc , the number of iterations required before convergence is $lc + 1$. Since in general we do not know lc in advance, another iteration is required to recognise that the process has converged, giving $lc + 2$ in total. These results are due to Kam and Ullman [10] and a very readable explanation of these results can be found in [15].

Knuth’s empirical study of FORTRAN programs [11] provides insight into use of the FORTRAN programming language in the 70s and he observes that the flow graphs of the studied programs have a derived sequence length of on average 2.75 steps and at most 6 steps. Knuth’s study has a relatively small sample size of only 50 programs which were obtained from a local (Stanford) source. Whilst his results are still insightful, a modern-day empirical analysis with a wider and larger sample would be valuable to better inform the analyses of algorithms such as this.

In this analysis we have only considered *reducible* graphs, those whose edges can be separated into disjoint sets `FORWARDEDGES` and `BACKWARDEDGES` such that `FORWARDEDGES` forms a directed acyclic graph and for every edge (u, v) in `BACKWARDEDGES`, v dominates u .

By inspection over the possible structured control flow mechanisms, we see that in the absence of unstructured control flow, we have at most 2 control dependencies for each node and as such the total number of control dependencies in a graph is linearly bounded. In an arbitrary control flow graph however, we end up with the conservative quadratic bound on the number of control dependencies. Indeed, with the unstructured control flow mechanisms in Swift, we can construct programs with a quadratic growth of control dependencies:

```

1  switch 1 {
2  case 1: fallthrough
3  case 2: fallthrough
4  ...
5  case n: fallthrough
6  default: print("Lots of control dependencies!")
7  }
```

Each case here is control dependent on every case that comes before it. Thus there are $n(n + 1)/2$ control dependencies in this $n + 3$ line program.

3.5 Slicing

I implemented slicing using the typical backward-reachability approach. This requires being able to efficiently map a node to its predecessors so this map should be stored explicitly (rather than being computed from the forward mapping). Caramel follows both control and data dependence edges to find the nodes in the slice. Caramel provides a minimal frontend for testing purposes. It accepts a file path, a line number, and a column number and finds the (unique) PDG node at that location. Slices are given as highlighted source code to draw attention to critical areas. An example slice output is given in Fig 3.6

```

for all nodes, b {
  ipdoms[b] := Undefined
  ipdoms[end_node] := end_node
  changed := true
  while (changed) {
    changed := false
    for all nodes, b, in reverse postorder (except end node) {
      new_ipdom := first (processed) predecessor of b /* (pick one) */
      for all other predecessors, p, of b {
        if ipdoms[p] != Undefined /* i.e., if ipdoms[p] already calculated */ {
          new_ipdom := commonPostdominator(p, new_ipdom)
        }
      }
      if ipdoms[b] != new_ipdom {
        ipdoms[b] := new_ipdom
        changed := true
      }
    }
  }
}

function commonPostdominator(n1, n2) returns node
  finger1 := b1
  finger2 := b2
  while (finger1 != finger2)
    while (finger1 < finger2) finger1 = doms[finger1]
    while (finger2 < finger1) finger2 = doms[finger2]
  return finger1
}

function buildFrontiers(ipdoms) {
  for all nodes b {
    pdomfrontier[b] = {}
  }
  for all nodes b {
    for successor in successors(b) {
      walker := successor
      while walker != ipdoms[b] {
        pdomfrontier[walker].insert(b)
        walker = ipdoms[walker]
      }
    }
  }
}

```

Figure 3.5: An algorithm for computing the postdominance tree and frontiers, reproduced from [4]

```
1 let n = Int(readLine()!!)
2 var sum = 0
3 var product = 1
4 var i = 1
5
6 while i < n {
7     sum = sum + i
8     product = product * i
9     i = i + 1
10 }
11
12 print(product)
13 print(sum)
```

(a) A simple Swift program.

```
let n = Int(readLine()!!)
var sum = 0
var product = 1
var i = 1

while i < n {
    sum = sum + i
    product = product * i
    i = i + 1
}

print(product)
print(sum)
```

(b) The highlighted source code output from Caramel.

Figure 3.6: An example slicing output from Caramel

Chapter 4

Evaluation

The success criteria I set out for this project were as follows:

1. Correctness of the slices produced by my slicer for the subset of Swift identified as verified by the devised test suite
2. ‘Reasonable performance’ of my slicer. It should take less than 5 seconds to slice a 1000 line program
3. For equivalent slicing criteria, my slicer should slice away at least half as much as the lower level slicer achieves

In this section I will evaluate the project against these criteria as well as discussing overall feasibility of the tool in practice.

4.1 Correctness

Defining correctness of program slices is not a trivial task. Many definitions have been used throughout the literature which cater to different tasks. Weiser’s original notion of correctness involved the slicer returning an executable slice that produced the same state at the criterion, for all possible input values, as the original program. For the debugging use-case, executable slices can be overly verbose, including structural or boilerplate statements that are required for the program to be executable but not salient to the debugging task. This means that I can’t use this definition of correctness for the evaluation of my non-executable slices.

Caramel’s correctness is evaluated against a test suite of programs. This suite includes programs created for the purpose of verifying the behaviour of Caramel for the range of supported language features, as well some select programs from GitHub. Those whose licenses allow have been reproduced in the repository, others are analysed (under fair use) but are omitted from the hosted repository. The test suite specifies the ranges that are expected to be in the slice and passes if those returned by Caramel are a superset of those expected (i.e. it has not missed any node that could, in fact be relevant to the criterion). The nodes that are expected have been specified by hand. This test suite passes on all tests. Of course, this is not the end of the story since a slicer that returns *all* of the

program statements would pass this correctness test suite. We also need to ensure it has reasonable precision, that is it doesn't return a significant amount of nodes that aren't relevant to the criterion. This is addressed in Section 4.2.

Correctness of implementation details of Caramel is verified via unit tests. Unit tests were written for CFG construction to ensure adequate construction for all supported language features. Unit tests were written for both the data dependency analysis and the control dependency analysis. Integration tests were written for the PDG construction. Of course, these tests do not prove correctness in any formal sense, they only go some way towards improving our confidence of the correctness.

My slicer provides correct results for a subset of Swift. The subset chosen was:

- Declaration of constants
- Declaration of variables
- For in statements
- Guard statements (including optional binding)
- If statements (including optional binding)
- Repeat while statements (including optional binding)
- Switch statements
- While statements (including optional binding)
- Assignment expressions
- Binary operator expressions

All subject to the condition that variables are scalar (i.e. they have a single value which is updated in full whenever it is updated).

The program restricts the use of other statement types but does not restrict the use of other expression types since this would be counter productive. As long as all expressions used are non-mutating and do not depend on variables other than their arguments, slices remain correct (according to my test suite). Note that in all test cases slices are non-trivial. That is, the slice contains fewer than the full set of nodes.

4.2 Precision

Precision of slices was measured as the number of relevant nodes returned in the slice divided by the number of nodes in the slice. Results can be seen in the table of Fig 4.1. In all cases, precision was measured at 100%.

On several occasions, for particularly complex files, nodes that were not marked as expected slice members were returned by Caramel which, upon further inspection did indeed belong in the slice. The test suite was updated to reflect this but these incidents speak to the fact that the data the test suite is testing against cannot be guaranteed

File	Criterion Location	Precision	Time taken	#PDG Nodes
multiplyAndAdd.swift	(13:9)	1.0	0.0088152885437011719	11
multiplyAndAdd.swift	(12:9)	1.0	0.0085849761962890625	11
suite/for.swift	(10:5)	1.0	0.08491206169128418	10
suite/for.swift	(11:5)	1.0	0.0069401264190673828	10
suite/guard.swift	(13:9)	1.0	0.10601305961608887	12
suite/guard.swift	(12:9)	1.0	0.010329246520996094	12
suite/repeatWhile.swift	(13:9)	1.0	0.10000491142272949	11
suite/repeatWhile.swift	(12:9)	1.0	0.0082919597625732422	11
suite/switch.swift	(15:5)	1.0	0.10333609580993652	16
github/j.swift	(16:20)	1.0	0.23354530334472656	27
github/p.swift	(18:19)	1.0	0.17990684509277344	18
github/d.swift	(81:17)	1.0	0.35867094993591309	44
github/d.swift	(98:17)	1.0	0.080527067184448242	44
github/s.swift	(17:19)	1.0	0.11873888969421387	15
github/t.swift	(38:15)	1.0	0.14450907707214355	13

Figure 4.1: Results from the Caramel test suite

complete since my by-hand slicing cannot be guaranteed complete. Comparison against another slicer on the same source files would therefore be informative. Initial plans to compare to existing LLVM slicers could not be carried out since I was unable to find a technique of tracing back to Swift source from sliced, compiled LLVM bitcode. I expect this is possible and future work may wish to explore this idea in greater depth.

4.3 Performance

For a slicer to be usable in practice it has to produce results almost instantly from the time it is given the criterion and the source program. My original goal for performance was to produce slice results for a 1000 line file within 5 seconds. A brief survey of a sample of Swift files that have been committed to GitHub since September 2017 revealed that this is not going to be a useful measure of performance. Of 4081 files analysed, selected at random from the files that were found to compile successfully, the longest function body found was that of a cryptographic hashing algorithm and was 155 lines long.

The Caramel test suite includes its longest file of 88 lines ('d.swift'). It can be seen in the results in Fig 4.1 that retrieving slices from this file took no longer than 0.4 seconds¹. This is an adequately brief period that the tool could feasibly be used in a development environment for debugging functions of this size.

As well as function length, the performance will also depend on the number of data dependencies. A program of 221 lines with worst-case growth of data dependencies for

¹Note that the discrepancy between the duration of the (81:17) and the (98:17) criteria tests can be put down to the caching mechanism built in to SourceKit which Caramel utilises

testing purposes. Slices of this file (`‘mediumSliceable.swift’`) could be found in 2.7 seconds.

4.4 Feasibility

In selecting programs to test the slicing behaviour of Caramel with, certain high-level trends were clear:

- The majority of the functions encountered were within either classes or structs
- The majority of function are too short to benefit from slicing
- Mutating functions were common
- Non-scalar values, namely structs and classes, where common

Each of these facts leads to Caramel being applicable to fewer files and limits its feasibility in practice. More features will need to be supported for the tool to have practical appeal.

Chapter 5

Conclusion

In this project I have successfully created a program slicer for a subset of the Swift programming language. I've discussed the precision and performance of my slicer and also looked at the feasibility of using this tool or a similar tool in practice.

5.1 Future work

To make this tool cater to a wider variety of real-world Swift programs, support for certain common features is crucial. In particular, non-scalar variables and reference types were of particular importance.

The use of a third party Swift parser makes this tool fragile to changes in the Swift syntax. Though a more challenging task, integrating this slicing functionality into the Swift compiler would provide more resilience to future changes in Swift and would also allow the slicer to make use of crucial type information.

The control flow behaviour of **break**, **continue**, and **fallthrough** has been modelled in Caramel's CFG but the dependence handling for these may be considered incomplete. Caramel does not make nodes control dependent on these statements despite the fact that removing these statements can change the behaviour of a program. Kumar and Horwitz offer a compelling approach to account for these dependencies appropriately in [12]. They augment the CFG with edges which can't ever be traversed at run time but that point to the statements that could be executed next if the, say, **break** statement was not there. Caramel could be extended using this approach to support highlighting of these statements.

If this tool is to provide a time saving for developers it must be usable seamlessly within a development environment. In this work, I have focused on providing a backend for a debugging-centric program slicer. Future work could experiment with possible frontend interfaces for Caramel within a popular IDE for Swift such as Apple's Xcode.