DISS. ETH NO. 20181

Localizing and Understanding Verification Errors

A dissertation submitted to
ETH Zurich

for the degree of
Doctor of Sciences
(Dr. sc. ETH Zurich)

presented by
Joseph Nicholas Ruskiewicz
M. Sc. Software Engineering, Carnegie Mellon University
B. Sc. Mathematics, University of Wisconsin – La Crosse
B. Sc. Computer Science, University of Wisconsin – La Crosse

born on
December 18, 1976

citizen of
United States of America

accepted on the recommendation of
Prof. Dr. Peter Müller
Prof. Dr. Thomas Gross
Dr. Tom Ball

2012

Zusammenfassung

Automatische Methoden zur Programmverifikation versuchen während der Übersetzung eines Programms zu beweisen, dass dieses auch korrekt ist. Ein Programm wird dabei als korrekt betrachtet, wenn die Ausführung dessen zu keinem Fehler führt, was wiederum der Fall ist, wenn die Spezifikation zu jedem Zeitpunkt während der Ausführung hält. Für den Fall dass die verwendete Verifikationsmethode nicht in der Lage ist die Korrektheit des Programms zu zeigen, wird der Programmierer von einem möglichen Fehler im Programm in Kenntnis gesetzt. Es liegt damit in der Verantwortung des Programmierers den tatsächlichen Grund für das Fehlschlagen der Verifikation zu bestimmen, wobei drei Typen von Ursachen unterschieden werden: (a) Ein fehlerhaftes Programm, also ein Programm welches tatsächlich einen Programmierfehler enthält, (b) eine fehlerhafte Spezifikation, wobei das Programm sich nach der Intention des Programmierers verhält, oder (c) die Verifikationsmethode war nicht in der Lage die Korrektheit des Programms zu beweisen – ein häufig auftretender Fall bei der Verwendung automatischer Verifikationsmethoden. In allen drei Fällen ist der Stand der Technik limitiert auf manuelle Inspektion durch den Programmierer oder systematisches Ausprobieren, wenn eine deduktive Verifikationsmethode verwendet wird. In dieser Dissertation wird der Stand der Technik verbessert indem neue Methoden zur Lokalisierung sowie zum Verständnis von Verifikationsfehlern vorgeschlagen werden. Um dem Programmierer dabei zu helfen einen Verifikationsfehler zu lokalisieren, wird Program Slicing verwendet. Dabei handelt es sich um eine effiziente Methode zum Entfernen von Programmteilen welche für das Fehlschlagen der Verifikation keine Rolle spielen. Existierende Methoden die auf Program Slicing basieren unterstützen *partielle* Anweisungen nicht oder inkorrekt. In dieser Dissertation wird deshalb eine neue Methode, welche diese Anweisungen korrekt behandelt, präsentiert. Für den Fall dass Progam Slicing nicht in der Lage ist dem Programmierer dabei zu helfen einen Verifikationsfehler zu verstehen, wir einen neue Methode vorgeschlagen, welche nicht nur mit dem Debugger (dem de-facto Werkzeug des Programmverständnisses) arbeitet, sondern den Verifikationsfehler auch validiert. Dies wird erreicht indem ein ausführbares Programm generiert wird, welches die Verifikationssemantik auf den Zuständen in einem Gegenbeispiel zur Korrektheit simuliert. Durch die Kombination dieser beiden Ansätze wird es möglich dem Programmie-

rer dabei zu helfen Verifikationsfehler effizient zu lokalisieren, sowie die Ursache dessen besser zu verstehen.

Abstract

Automatic program verification attempts to prove the correctness of a program at compile time. A program is considered correct if the execution of the program does not lead to a program failure. A program fails if a specification does not evaluate to true during the execution of the program. If the program verifier is unable to ascertain the correctness of the program, the programmer will be notified of a possible failure. It now becomes the responsibility of the programmer do determine the cause of this verification error. This error may result from (a) incorrect program, that is, the error is in the program and not the specification, (b) incorrect specifications, that is, the program does what the programmer intended and it is the specification that is incorrect, or (c) the verifier was simply too weak and was not able to verify the correctness of the program; a common case in the context of automatic program verification. For each of these cases, the state of the art is limited to trial and error and manual inspection when using a program verifier based on deductive reasoning. In this thesis we extend the state of the art to provide better tools to help the programmer localize and understand the cause of verification failures. To help the programmer locate the cause of the verification failure we apply the technique of program slicing. Program slicing is an efficient approach to removing those parts of the program that do not play a role in the behavior of the verification failure. However, applying this approach in the context of automatic program verification in the presence of *partial* commands has an omission with consequences both in theory and practice, which we aim to rectify in this thesis. If program slicing is unable to help the programmer understand the verification error, we present a new approach that will not only work with the de-facto tool for understanding programs, the program debugger, but will also validate the verification error. We achieve this by constructing an executable program that simulates the verification semantics of the program with the states given by the counterexample. By combining our two approaches we provide an invaluable aid to the programmer that will help them efficiently and systematically locate and understand the cause of verification errors.

Acknowledgements

First and foremost I would like to give my upmost gratitude and thanks to my *Doktorvater* Prof. Dr. Peter Müller. Without his support and insight this thesis would not have existed. I also wish to extend this gratitude to my reviewers Dr. Tom Ball and Prof. Dr. Thomas Gross for their time and effort.

The time of this thesis was an incredible journey and I wish to acknowledge all the great friends and colleagues that have taken this journey with me. In particular, I wish to thank Stephanie Balzer, Christoph Wintersteiger, Werner Dietl, Aresnii Rudich, Herman Lehner, and Jean-Raymond Abrial for various forms of discussion and entertainment. I also wish to thank my officemates over the years — Ádám Darvas, Alex Summers, Ioannis Kassios, and Pietro Ferrara — for putting up with my various idiosyncrasies. To the rest of the group, thank you! A very warm thanks is in due for Marlies Weissert for being more than her position. Thank you so much for everything you have ever done for me.

I would like to thank the everlasting love and support of my family. Doing a Ph.D. is a trying experience and living so far does not make it easy to stay connected. I would like to thank my grandparents Nick and Kathleen for always being there for me. For my parents Mary, Pete, and Nick; thanks for all the support you have given me. My amazing sisters Maria and Margaret, you are just thanked for being in my life. To the rest of the family, thanks for always lending me a hand, a van, or a bike whenever I am in town.

To the amazing friends I have made of over the years here in Switzerland. Thank you! For the friends I have left back home, thank you for your support. I hope to see you all again someday soon! To Paul Ross for reminding me of the priorities in life and Ladd Van Tol for all the great times we always have, thank you both.

I wish to acknowledge the great CS department at UW – La Crosse that has molded me into the creature I am today. In particular, a special thanks to Prof. Dr. David Riley, Prof. Dr. Steven Senger, Prof. Dr. Kasi Periyasamy, and Dr. Thomas Gendreau.

Finally, I wish to thank my partner Mai Matsushita for being in my life. The best part of my time here in Switzerland has been with you. Thank you for everything you do.

Contents

List of Figures

Chapter 1

Introduction

Automatic program verification takes a program and a specification and attempts to verify that the program satisfies the specification. This process is usually transparent to the programmer and is done during the compilation phase of a program. When the verification succeeds the programmer is given a guarantee that their program will never fail. However, when verification fails, the programmer is left with the responsibility to understand why the program has failed. A verification attempt fails for one of the following reasons:

1. The program is incorrect, that is, the program does not satisfy its specification, and the *specification* expresses what the programmer intended. A typical example is a runtime error such as division by zero.

2. The specification is incorrect or incomplete, that is, the program does not satisfy its specification, and the *program* expresses what the programmer intended. A typical example is a loop invariant that is too weak.

3. The prover was too weak to validate the condition, that is, the verification error is a false positive, called a *spurious error*.

All three causes occur frequently in program verification; in particular, incorrect and incomplete specifications are as common as errors in

programs. Spurious errors are less common, but are more difficult to understand as they usually happen when the program or specification is too complex for the program verifier thus misleading the programmer into locating the cause of an error for a valid program.

A common approach to automatic program verification is to compute verification conditions, logical formulas whose validity entails the correctness of the program. The verification conditions are then passed to an automatic theorem prover, typically an SMT solver such as ‘ [22] or Z3 [20]. If the prover can establish the validity of the verification condition, then verification succeeds (case 1). Otherwise, verification fails and the programmer is notified of this failure along with an indication as to the specification the solver was unable to verify (cases 2 and 3).

Besides the location of the failing specification, the main support that verifiers based on SMT solvers provide for understanding a verification error is a counterexample that illustrates why the verification condition is not valid. A counterexample essentially contains a value for each variable in each execution state, and therefore characterizes an execution of the program being verified. This information is essential for understanding a verification error. However, for programs with non-trivial states, for instance heap data structures of modern programming languages, counterexamples can be many times the size of the program being verified, and thus not comprehensible by a programmer. Moreover, if a verification condition is beyond the capabilities of the SMT solver, then the counterexample may be invalid and not representative of a valid program execution, thus misleading the programmer.

In this thesis, we present two approaches to help programmers localize and understand failed verification attempts. If the programmer has determined that the failing specification reported by the program verifier is what they intended, then the cause of the verification failure must be located in the program. Experienced programmers who understand the behavior of a subset of the variables in a program do not inspect all the commands of the program, only those who contribute to the states of those variables. That is, they slice away the commands of a program that are irrelevant to the variables in the criterion. We use this concept of program slicing to help programmers localize the verification error by removing those commands and specifications from the program that do not contribute to the failing specification. As the program slice is *smaller*

than the original program, it is therefore easier for a programmer to locate the cause of the program error.

However, for programs that work on heap like data structures, program slicing is very challenging and may not always localize the cause of the failure. The second approach that we present in this thesis is a technique that enables programmers to use standard debuggers to inspect program verification and counterexamples just as they use debuggers to inspect program executions and execution states. Our technique enables programmers to step through the verification of a method, check the validity of assertions, and observe the evolution of the state described by the counterexample. It detects verification failures caused by all three reasons mentioned in the introduction and notifies the programmer of spurious errors and invalid counterexamples. This tool support allows programmers to understand, locate, and fix verification errors more easily. We believe that applying a familiar tool for this task is crucial for making program verification more efficient and for increasing acceptance among practitioners. Our approach is implemented within the Spec# programming system.

1.1 Overview of the Thesis

In this thesis we present our approach to localize and understand errors in failed verification attempts. The first part of this thesis will focus on localizing the possible cause of the verification failure. Our approach is based on the idea of program slicing except that we will extend the approach to handle assume commands, which are used to represent a larger set of commands that are called partial commands. The second part of this thesis will focus on understanding the cause of the error in a verification failure. Programmers use debuggers to understand why programs fail and we extend this idea to using program debuggers to help the programmer understand verification failures.

First we will present preliminaries that are required to understand both parts of this thesis. We will present the language used and the semantic characterization of this language using the predicate transformers wp and wlp. To ensure that these predicate transformers *make sense* we will present the classical healthiness rules [24] and give some indication to the correctness of these transformers when applied to the rules. With

the healthiness rules, we will present partial commands and understand the impact these commands have in a programming language and how they are used in the context of verification. We present our model of a sound program verifier and how it is related to the predicate transformer wp. To complete this chapter we will show how our basic language can be used as an intermediate language to encode the semantics of more sophisticated languages such as Spec#.

With the preliminaries finished we will present our approach to program slicing for total commands. The current literature contains many approaches and correctness results of varying degrees of confidence, but none has captured what we feel it means to directly slice a program and prove that this program is indeed a correct program slice. We will present a data dependency function that will give us a set of relevant variables for a program and we will use this data dependency function in our presentation of our program slicing function. Our program slicing function is a program transformer, thus it is a direct way of representing what a program slicer should achieve when inspecting programs. We will conclude the chapter with a discussion on the correctness of program slicing and a direct proof of this correctness using the predicate transformers. We believe that this is the first direct proof of the correctness of program slicing. This chapter will also introduce some basic concepts that will be used in the next chapter on how we slice partial commands.

After we present our approach to slicing programs for total commands we will turn our focus to the slicing of partial commands. To preserve the correctness of slicing for total commands we are forced to keep all partial commands in the program. However, by not removing those commands that do not play a role in the understanding of a verification failure appears to go against the spirit of program slicing. To slice partial commands, we weaken the definition of program slicing and we proceed to construct a program slicer for partial commands. As these commands commute freely over sequential composition, we are required to extend our data dependency function to also include anti-dependencies in the program. As a dependency can occur at various places in a program, we extend our dependency function to iterate through the program until a fixpoint is reached. With this fixpoint we use our program slicer to remove those assumptions from the program that are irrelevant to the failed verification attempt. We formalize the correctness of the weakened

notion of program slicing and prove that our program slicer produces semantically correct program slices. We conclude the chapter with a presentation on how we can utilize the program slicer to remove assumptions on the background theory used in program verification.

After we understand how we can slice partial commands we will present how the partial command program slicer can be used to help the programmer locate the cause of the verification failure. As our program slicer is not complete, a negative response may not indicate necessarily an invalid program and we have to take this into account when we give guarantees to the programmer of what they can expect from the program slice. As modern program verifiers give a counterexample in the form of a program trace, we discuss how this can be used to drive a dynamic program slicer to produce even finer program slices. We discuss some issues when slicing the intermediate form of a more sophisticated language and finally conclude with some insights into our implementation for our constructed program slice in the Boogie verification environment. This concludes the first part of the thesis.

In the second part of the thesis we turn our focus to understanding verification failures. As programmers use program debuggers to understand program failures, we extend this to also enabling the programmer to understand verification failures with the same familiar program debugger. To validate the error and the states of the counterexample, we construct an executable program that will simulate the verification semantics of the program with the states given by the counterexample. This constructed program can then be attached to a program debugger which will allow the programmer to observe the failure and the sequence of states that lead to the failure. By using a program debugger it is understood that our approach will offer a level of usability that will enable the programmer to efficiently understand failed verification attempts.

We will finally conclude this thesis with a short summary and some indication as to the next steps that can be taken with the results of this thesis.

Chapter 2

Preliminaries

A standard approach to program verification is to transform a given program and specifications into a set of *verification conditions*. These conditions are logical formulas whose validity implies that the program satisfies its specifications. The verification conditions are usually validated with the help of an automatic theorem prover, typically a satisfiable modulo theories (SMT) solver. Similar in the approach used by modern compilers, the complex task of generating verification conditions for modern programming languages is better performed by transforming the program into an intermediate form [6, 44]. This intermediate form is closer to the logical formulas used by the verification condition generator but still preserves some structure of programs, thus allowing for more precise analysis and programmer understanding.

Consider the Spec# program in Figure 2.1. This program declares a type `Node` and a method `Add` that represents the addition of a node to a link-list. The method takes a node `n` and either assigns the field `next` to point to the node `n`, or it passes the node `n` to the next node if it is not null. The method contains a postcondition `Contains` that will ensure that when the method terminates the node `n` is in the list. Encoding the Spec# program into an intermediate language will give us the program in Figure 2.2. In this intermediate language we can see that everything about our introductory example has been made explicit. Namely, the reading and writing to the heap locations of `this.next` has been made

explicit through the use of the array $\mathcal{H}$. We have removed aliasing from the program by using an array to model aliasing, thus simplifying the handling of the program by a program verifier.

```
1  class Node {
2    Node next;
3    public void Add (Node n)
4      ensures Contains (n);
5    {
6      if (this.next != null)
7        next.Add (n);
8      else
9        next = n;
10   }
11 }
```

Figure 2.1: A typical Spec# program that takes a node `n` and adds it to the end of the list. This program contains a post-condition that asserts that the list contains `n` when the method finishes execution.

```
1  type Node;
2  var H:[ref,name] ref;
3  var Node.next : name;
4
5  proc List.Add (this:ref, n:ref)
6      ensures Contains (this,n);
7    assume H[this,Node.next] != null;
8    call List.Add(this,n)
9  ||
10   assume H[this,Node.next] == null;
11   H[this,List.next] := n
```

Figure 2.2: An intermediate form of our Spec# program. The run-time heap is made explicit with the variable $\mathcal{H}$ and the method call to `Add` is now a call to the procedure `Add`. With everything made explicit this language is considerably easier to generate verification conditions for the theorem prover.

In this chapter we present the language and semantics of a basic intermediate language that is designed to accommodate the encoding of verification conditions for imperative, object-oriented programs. We give a semantic characterization of this language using the predicate transformers wp and wlp. We state some healthiness properties of our predicate transformers and give some reasoning why we can ignore the classical property of strictness when we verify programs. We construct a generic model program verifier and define what we can expect from a sound verifier. As this is a basic intermediate language, we give some source language commands (`if`,`while`,`call`) and show how their *meanings* can be encoded in our intermediate language.

2.1 Language

We use an intermediate language based on the Boogie language [44, 6]. The language consists of two parts, the mathematical part and the imperative part. The mathematical part consists of declarations of types, constants, first-order functions, and axioms. This part is commonly referred to as the *background* part. The imperative part of our language contains declarations of global mutable variables and procedures whose bodies are allowed to update and check properties on these variables.

2.1.1 Background Theory

Our language represents an intermediate language that can be used for the verification and analysis of more complicated source languages such as C#. To encode the semantics of these source languages, we allow the programmer to describe certain properties of the language that are assumed to hold when the program executes. These properties correspond to static type information of the source language or more sophisticated concepts such as abstract data types. We call this part of a program declaration the *background* theory of the program. It is used to define the signature for the declarative part. A background theory consists of the following declarations

$$\texttt{type T} \mid \texttt{const C : T} \mid \texttt{func F (T) returns (T}') \mid \texttt{axiom E}$$

Types In addition to the builtin types `int` and `bool`, our language allows for programmer defined type declarations. A programmer can declare a new type constructor by declaring a new identifier with the keyword `type`. If the programmer wishes to declare a new type `T`, they can use the declaration

```
type T
```

This declaration defines a new type constructor named `T`. To be valid the name `T` must be unique. Every type represents a nonempty set of elements that belong to this type. For example, the type declaration

```
type Fruit
```

declares a new type constructor that represents all possible fruits. Along with these programmer declared types, programmers can also make use of the primitive types `int` and `bool` and array types. The primitive type `int` denotes the mathematical integers and `bool` denotes the set of Boolean values. The array type represents heterogeneous arrays. The domain of the array type is listed within square brackets followed by a range type. For example, [`int`]`Fruit` declares an array type that has `int` as its domain and `Fruit` as its range. We can declare multiple domains for an array by simply delimiting each type by a comma. The set of all types in a program is contained in the set $\mathcal{T}$.

Constants and Functions A constant is a type of variable that does not vary in its value. A programmer can declare a constant C with the `const` keyword. Along with a declaration of the constant the programmer must provide a valid type for the constant. For example, the declaration

```
const C : T
```

declares a constant C with type T. To be valid the name of C must be unique and T must either be a user defined type or one of the builtin types {`int`,`bool`}. For example, a declaration of the form

```
const banana : Fruit
```

declares a constant `banana` of type `Fruit`. The Boogie language allows the programmer to decorate the declaration with a `unique` keyword. Adding this keyword to our `banana` declaration ensures that no other constants of type `Fruit` can be equal to `banana`. We will use $\mathcal{C}$ to represent the set of constants in a program.

Along with constants, a programmer can also declare functions. These functions represent mathematical functions. A declaration of the form

```
func F (T) returns (T')
```

declares a function named `F` that takes as input an element of type `T` and returns an element of type `T'`. A programmer can declare more than one input and output by separating each type by a comma. The name of the function must be unique. The function declaration

```
func isRipe (Fruit) returns (bool)
```

declares a function called `isRipe` that takes some parameter of type `Fruit` and returns a `bool` type. The encoding of this function into its mathematical counterpart is

$$\text{isRipe} : \text{Fruit} \rightarrow \text{bool}$$

where $\rightarrow$ has the usual function denotation in mathematics. We assume that all functions are *total*. We use the set $\mathcal{F}$ as the set of all functions in a program.

Axioms Declaring a constant or a function is not very useful unless it is possible to postulate some properties on the constant or function. Axioms are used to declare properties on constants and functions. An axiom declaration has the form

```
axiom E
```

where `E` is a first-order expression of type Boolean. It is used to express properties about the program's constants and functions. For example, the axiom declaration

```
axiom forall f : Fruit :: age(f) <= 20 ==> isRipe(f)
```

states that every fruit with an age less than or equal to `20` must be ripe. This axiom is assumed to hold in all states as constants and functions are not mutable. As with axioms of logic, it is possible to write inconsistent axioms that are equivalent to false. We will use $\mathcal{A}$ to represent the set of axioms for a program $\mathcal{P}$.

2.1.2 Imperative Programming

The background part of our programming language allows the programmer to express assumptions on the environment of the program and construct new theories that can be used to model programming concepts. Our language also allows the programmer to write an imperative part that allows the programmer to declare global variables and instruct the verifier to either modify these variables or state assertions on them.

Mutable State

The imperative part of our programming language operates on a state space Σ. The program state space is the Cartesian product of all variables in the program. A state $\sigma \in \Sigma$ is an assignment of a value to each variable. For example,

$$\{x \mapsto 7, y \mapsto 3\}$$

represents a state where `x` maps to the value `7` and `y` maps to the value `3`. Programmers can extend the state of the program by declaring a new variable `x` of type `T` using the declaration

```
var x : T
```

which declares a variable `x` of type `T`. This declaration essentially extends the state of the program point-wise to include the variable `x`. The name of the variable must be unique in the program. For example, the declaration

```
var apple : Fruit
```

declares a modifiable variable `apple` with the type of `Fruit`. The value of `apple` is some arbitrary value of `Fruit`. We will use the set **Vars** to represent the set of sets of variables in a program. That is, **Vars** is equal to the powerset of variables in the program.

Expressions

Expressions in our language include constants, functions, variables, equality and arithmetic relations, Boolean connectives, simple arithmetic operators, and an ordering operator used for typing. In our language, this is essentially first-order logic with an extension for arithmetic. Valid expressions must be typed correctly. Equality between two expressions `E` and `F` is expressed as

$$\texttt{E} == \texttt{F}$$

where an expression is either a Boolean expression, an arithmetic expression, or simply a variable name. A Boolean expression can contain the basic connectives `and`, `or`,=> and the quantifiers `exists` and `forall`. An arithmetic expression is built from the usual connectives <,>,<=,

and >=. An expression can introduce quantification over a type.news For example,

$$\texttt{exists f : Fruit . f != banana}$$

states there exists an element of type `Fruit` that is not a `banana`. Expressions may include array types. The expression `m[j]` access the value of array `m` at the index `j`. An expression is valid for a state $\sigma \in \Sigma$ if the evaluation of the expression point-wise for each variable is valid. For example, given a state σ the expression

$$\texttt{x} < \texttt{y}$$

is only valid if $\texttt{x}.\sigma$ is strictly less then $\texttt{y}.\sigma$ where $v.\sigma$ is the value of v in σ. Given a state σ we can evaluate the value of a variable v in σ by using the dot notation $v.\sigma$ which will give us the value of v in σ. For example,

$$y.\{x \mapsto 7, y \mapsto 3\}$$

will give us the value `3` as this is the value mapped to by `y`. If we apply a state σ' to another state σ, we treat this as an overriding of the variables contained in σ. For example,

$$\{x \mapsto 7, y \mapsto 3\}.\{x \mapsto 0\}$$

is equivalent to

$$\{x \mapsto 0, y \mapsto 3\}$$

where the value of `x` has been overridden to now have the value `0` and the value of `y` has been unchanged.

Procedures

Procedures are reusable units that a programmer can use in their programs. A programmer can declare the signature of a procedure `P` by declaring the input variables and types, output variables and types, and specifications on the global variables, input parameters, and output parameters. A declaration of the form

```
proc P (ins : T) returns (outs : T)
    requires E
    modifies m
    ensures E
```

defines a procedure `P` with `ins` of type `T` as input parameters and `outs` of type `T` as output parameters. To avoid any ambiguity, we assume that input and output variable names are disjoint from any constants or global variable names. The programmer can specify a precondition of the method with the `requires` declaration and an expression `E`. The variables mentioned by `E` must only contain constants, global variables, and input parameters.

To specify which global variables may be modified by a procedure, a programmer can also specify the *frame* of the procedure with a `modifies` declaration. A global variable not mentioned in the specification may not be modified by the procedure. The `ensures` declaration represents a condition on the post-state of the procedure. The expression of the `ensures` declaration is allowed to mention all global variables, input variables, and output variables. It is also allowed to mention the pre-state of all global variables with the use of the `old` keyword. We assume that any reference to the pre-state of a global variable already has been de-sugared into a temporary variable, thus freeing us from having to explicitly deal with old variables.

The procedure declaration

```
proc Eat (f : Fruit) returns (b : bool)
    requires isRipe(f)
    modifies isFull
    ensures b == true ==> isFull == true
```

declares a procedure `Eat` that has one input parameter `f` of type `Fruit` and one output parameter `b` of type `bool`. The procedure declares that `f` must be ripe when the procedure is called. The declaration also states that it may modify the global variable `isFull`. Finally, this declaration ensures that upon exiting the procedure if the value of `b` is true, then the value of `isFull` will be true.

Optionally, a programmer can declare a body within the procedure. This body consists of local variable declarations and an imperative program. This imperative program consists of a combination of commands that read and write to state.

Commands

The imperative part of our language tests and mutates program states. We mutate and test program states through the use of commands. With commands we can update a variable, state an assertion or assumption on the current state, or construct a larger set of commands using either sequential composition or non-deterministic choice. The commands that we will use in our language are primitive, but still sophisticated enough to capture almost any source language semantics. The basic commands are the following

$$\texttt{skip} \mid \texttt{x} := \texttt{E} \mid \texttt{havoc x} \mid \texttt{assert E} \mid \texttt{assume E} \mid S0; S1 \mid S0||S1$$

We will us the set **Prgs** to represent the set of commands.

No-Op The command `skip` behaves as a no-op. It does nothing. Executed in a state σ will produce the exact same state.

Assignment The basic assignment command evaluates the expression on the right hand side when executed and assigns the values to the identifiers on the left hand side of the assignment. We assume a single identifier on the left side of the assignment, but generalizations can be made. The type of the right hand side must conform to the type on the left hand side. The assignment

$$\texttt{x} := \texttt{E}$$

assigns to `x` the value of the expression `E`. For example, if we assign to `x` the value of $\texttt{y} * \texttt{z}$, we first look up the values of `y` and `z`, multiply them together and then replace the current value of `x` with this new value.

An assignment command of the form $\texttt{m}[\texttt{i}] := \texttt{E}$ first evaluates the expression on the right hand side and then assigns the values into the index `i` of the array `m`. This has the effect of not only changing the value of the indexed location but also changing the state of the array such that all other indexes are preserved. We assume that an index `i` will always be within the domain of the array.

Havoc The havoc command is used to define an unknown value. That is, the command assigns an arbitrary value to the referenced variable.

For example, the command

```
havoc x
```

arbitrarily chooses an element from the type of `x` and assigns it to `x`.

Assertions The assert command expresses a desired condition that should hold upon execution of the command. If the condition does hold for the state currently executing, the assert command behaves like a no-op and execution continues without any modification to the state. If the condition does not hold, then the command aborts the program. For example, the assertion

```
assert x < y
```

asserts that the value of `x` is strictly less than the value of `y`. If we execute this command in a state where the value of `x` is indeed strictly less than `y`, then the command is equivalent to `skip`. The expression given by an assert command must be of type Boolean and can refer to any constant, function, or other variable.

Assumptions The assume command expresses an assumption made by a programmer upon the state of the program. If the condition does hold in the current state, then the assume command behaves like a no-op. If the condition does not hold, then the command produces a miracle and the program is said to trivially satisfy any state the programmer wishes to achieve. That is, when an assumption does not hold, it is said that a *miracle* occurs. The assumption

```
assume x < y
```

assumes that the value of `x` is strictly less than the value of `y`. If we execute this program in a state where the value of `x` is indeed strictly less than `y`, then the program is equivalent to `skip`. However, if `x` is not strictly less than `y` then the program gets into an impossible state and a miracle is said to have occurred. Once a miracle has occurred every postcondition is valid, even the impossible condition `false`. The expression given by an assume command must be of type Boolean and can refer to any constant, function, or other variable.

Sequential Composition Basic commands are not very interesting until you can compose them together. Sequential composition does exactly this. Given a command $S0$ and a command $S1$, sequential composition first executes $S0$ and if $S0$ terminates then $S1$ will execute with the resulting state from executing $S0$. For example, the program

$$\texttt{x} := \texttt{y} * \texttt{z}; \texttt{assert } \texttt{0} <= \texttt{x}$$

assigns `x` the value of `y` times `z` and then checks that the state of `x` is greater than or equal to `0`. As assignment always terminates, the next command will always be executed. However, assertions may not always terminate and our assertion will not terminate if `x` is not greater than or equal to `0`. As sequential composition is a command, we can construct arbitrarily but finite sequences of commands.

Non-deterministic Choice Given a command $S0$ and a command $S1$, the non-deterministic choice operator chooses which command it should execute non-deterministically. This choice is completely arbitrary. For example, the execution of

$$\texttt{x} := \texttt{y} * \texttt{z} \mid\mid \texttt{assert } \texttt{0} <= \texttt{x}$$

will either choose to execute the assignment of `x` to the value of `y` times `z` or it will choose to assert that the value of `x` is greater than `0`. It will only choose one branch, it will not execute both.

2.1.3 Program Labels

To be able to inspect a program and differentiate between equal commands that occur in different positions in the program we make use of program labels. Program labels are unique declarations given to every basic command in the program. To represent a program label we will use the set of natural numbers. The program labels can either be given by the programmer or automatically given by a compiler. For example, given a command $\mathcal{S}$ and a program label ℓ, we decorate the command as

$$\ell : \mathcal{S}$$

which states that command $\mathcal{S}$ is located at program label ℓ. When necessary, we will make the labels in a program explicit by prepending each

command or background declaration with a label. For example, if a program contains the command $\mathtt{x} := \mathtt{y} + \mathtt{z}$ it will be decorated with a program location, say $\mathtt{3}$, that gives us an indication of the location in the program. For example,

$$\mathtt{3}\colon \mathtt{x} := \mathtt{y} + \mathtt{z}$$

tells us that the command $\mathtt{x} := \mathtt{y} + \mathtt{z}$ has the program label $\mathtt{3}$. We assume a function LAB that given a program will give us the set of program labels in that program. We assume a set of labels $\mathbb{L}$ for every given program $\mathcal{P}$ such that $\mathrm{LAB}(\mathcal{P})$ is equal to $\mathbb{L}$. A program label ℓ is valid if and only if $\ell \in \mathbb{L}$. Program labels are only an auxiliary decoration for a command and play no role in the semantics of a program.

Given a valid program label ℓ in a program $\mathcal{P}$, the command located at program location ℓ can be given by the function call to $\mathcal{P}[\ell]$. Letting $\mathcal{P}$ be our previous example, the query $\mathcal{P}[\mathtt{3}]$ will give us the command $\mathtt{x} := \mathtt{y} + \mathtt{z}$.

As with natural numbers, we assume that program labels are well-ordered under $\leq$. If a label ℓ is less than another program label ℓ', then the execution of the command $\mathcal{P}[\ell]$ will always happen before the execution of the command $\mathcal{P}[\ell']$.

2.1.4 Free Variables

A useful function for this thesis will be a function FV that will tell us what variables a program either updates or reads from. The set of variables that a command updates or reads from is called the *free variables* of the command. We compute the set of free variables FV as follows

Definition 1 ($\mathrm{FV} : \mathbf{Prgs} \to \mathbf{Vars}$)**.**

$$\begin{aligned}
\mathrm{FV}(\mathtt{skip}) &= \varnothing \\
\mathrm{FV}(\mathtt{x} := \mathtt{E}) &= \{x\} \cup \mathrm{FV}(E) \\
\mathrm{FV}(\mathtt{havoc\ x}) &= \varnothing \\
\mathrm{FV}(\mathtt{assert\ E}) &= \mathrm{FV}(E) \\
\mathrm{FV}(\mathtt{assume\ E}) &= \mathrm{FV}(E) \\
\mathrm{FV}(S0; S1) &= \mathrm{FV}(S0) \cup \mathrm{FV}(S1) \\
\mathrm{FV}(S0 || S1) &= \mathrm{FV}(S0) \cup \mathrm{FV}(S1)
\end{aligned}$$

where $\mathrm{FV}(E)$ follows the usual free-variable analysis of first-order expressions. A program $\mathcal{P}$ must have the property that

$$\mathrm{FV}(\mathcal{P}) \subseteq \mathrm{dom}(\Sigma)$$

where Σ is the possible states of the program mapping variables to values and dom returns the domain of the function Σ. It is also customary to separate the variables read by a program and the variables modified by a program. The function $\mathrm{def}(\mathcal{P})$ returns all variables that are updated by a program $\mathcal{P}$.

2.2 Predicate Transformers

In this thesis we will use the wp and wlp predicate transformers [23, 24] to give a semantic meaning to the imperative part of our program. Given program $\mathcal{P}$ and a postcondition R, the weakest precondition transformer (wp) characterizes those initial states for which every execution of the program will terminate in a state satisfying the postcondition R. The weakest liberal precondition transformer (wlp) characterizes those initial states that will either not lead to termination or if it does terminate then it will satisfy the postcondition R.

The program in Figure 2.3 assigns the value of `y` to the variable `total`, asserts if the state of `sum` is equal to `0` and then assigns `x` the value of `total`. If the value of `sum` is not equal to `0`, then the program will abort and not terminate.

```
1 total := y;
2 assert sum == 0;
3 x := total;
```

Figure 2.3: A program that will abort if `sum` is not equal to `0` or terminate normally otherwise.

The weakest precondition for this program with the postcondition $0 \leq$ `x` will characterize those initial states where the value of `y` must be

greater than `0` *and* the value of `sum` must be equal to `0`. If the value of `y` is not greater than `0`, then the program may terminate in a state where `x` is not greater than `0` as the value of `x` is defined by the value of `y`. If the value of `sum` is not equal to `0`, then the assertion that `sum == 0` will abort and will not satisfy the semantics of the weakest precondition.

However, if the programmer is not interested in termination they may apply the weakest liberal precondition transformer to the program. For the given postcondition $0 \leq \mathtt{x}$, the weakest liberal precondition will characterize those states where `y` must be greater than `0` *or* the value of `sum` must not be equal to `0`. That is, if we execute the program in a state where the value of `y` is greater than `0` or if we execute it in a state where `sum != 0`, the program will either terminate in a state satisfying the postcondition or the program will abort.

The predicate transformers wp and wlp are defined over the commands in our language. There is very little difference between the two except when it comes to commands that may not terminate. The set **Preds** is the set of all predicates. We defying wp as

Definition 2 (wp : **Prgs** $\times$ **Preds** $\rightarrow$ **Preds**)**.**

$$\begin{aligned}
\mathrm{wp}(\mathtt{skip}, R) &= R \\
\mathrm{wp}(\mathtt{x} := \mathtt{E}, R) &= R[x := E] \\
\mathrm{wp}(\mathtt{havoc\ x}, R) &= \forall x \cdot R \\
\mathrm{wp}(\mathtt{assert\ Q}, R) &= Q \wedge R \\
\mathrm{wp}(\mathtt{assume\ Q}, R) &= Q \Rightarrow R \\
\mathrm{wp}(\mathtt{S0}; \mathtt{S1}, R) &= \mathrm{wp}(\mathtt{S0}, \mathrm{wp}(\mathtt{S1}, R)) \\
\mathrm{wp}(\mathtt{S0} || \mathtt{S1}, R) &= \mathrm{wp}(\mathtt{S0}, R) \wedge \mathrm{wp}(\mathtt{S1}, R)
\end{aligned}$$

where $R[x := E]$ denotes the simultaneous capture-avoiding substitution of the expression `E` for the the variable `x` in R. We define wlp similarly as wp.

Definition 3 (wlp : **Prgs** $\times$ **Preds** $\rightarrow$ **Preds**)**.**

$$\begin{aligned}
\mathrm{wlp}(\mathtt{skip}, R) &= R \\
\mathrm{wlp}(\mathtt{x} := \mathtt{E}, R) &= R[x := E]
\end{aligned}$$

$$\begin{aligned}
\text{wlp}(\texttt{havoc x}, R) &= \forall x \cdot R \\
\text{wlp}(\texttt{assert Q}, R) &= \neg Q \vee R \\
\text{wlp}(\texttt{assume Q}, R) &= Q \Rightarrow R \\
\text{wlp}(\texttt{S0}; \texttt{S1}, R) &= \text{wlp}(\texttt{S0}, \text{wlp}(\texttt{S1}, R)) \\
\text{wlp}(\texttt{S0} || \texttt{S1}, R) &= \text{wlp}(\texttt{S0}, R) \wedge \text{wlp}(\texttt{S1}, R)
\end{aligned}$$

where $R[x := E]$ denotes the simultaneous capture-avoiding substitution of the expression `E` for the the variable `x` in R. The only difference between the two predicate transformers is how the expression of an assertion is handled. In the wp the expression is conjoined with the postcondition thus strengthening the condition given to the programmer. In the wlp the expression is assumed to hold, thus weakening the precondition given to the programmer. Both transformers work *backwards* through a program.

The relationship between the weakest precondition transformer and the weakest liberal transformer is called the *pairing* condition. This condition states

$$\text{wp}(\mathcal{P}, R) \equiv \text{wlp}(\mathcal{P}, R) \wedge \text{wp}(\mathcal{P}, \top)$$

must hold for all states. This condition makes explicit that the wp computes the necessary preconditions to ensure termination and that the wlp computes the necessary preconditions to satisfy the postcondition. When we use $\top$ as the postcondition of the wp, we are given the weakest precondition to ensure termination of the program.

2.2.1 Healthiness Properties

Any function that takes a predicate and returns a predicate is defined as a predict transformer. However, the predicate returned must make sense in the context of program semantics. To ensure that a predicate transformer does make sense for program semantics, some *healthiness* properties [24] have been defined that wp and wlp must satisfy.

The first and most important healthiness property is monotonicity. The idea is that if we know that every state in Q is also in R, then for an arbitrary program $\mathcal{P}$, every initial state that satisfies the precondition $\text{wp}(\mathcal{P}, Q)$ should also satisfy the weakest precondition $\text{wp}(\mathcal{P}, R)$.

That is, wp (and wlp) should both be monotonic in its second argument. This property allows the programmer to reason about the composition of programs.

Proposition 1 (Monotonicity)**.** *For any program $\mathcal{P}$ and post-conditions Q and R such that*

$$Q \Rightarrow R \quad \textit{for all states}$$

then we also have

$$\mathrm{wp}(\mathcal{P}, Q) \Rightarrow \mathrm{wp}(\mathcal{P}, R) \quad \textit{for all states}$$

The second healthiness property ensures that wp is conjunctive. If a program executed in a state that satisfies $\mathrm{wp}(\mathcal{P}, Q \wedge R)$, then we know that $\mathcal{P}$ will terminate in a state satisfying $Q \wedge R$. A transformer is conjunctive if the initial state for $\mathrm{wp}(\mathcal{P}, Q \wedge R)$ also satisfies the preconditions $\mathrm{wp}(\mathcal{P}, Q)$ and $\mathrm{wp}(\mathcal{P}, R)$. That is, reasoning about a program that satisfies the postcondition $Q \wedge R$ is the same as reasoning about a program that satisfies the postcondition Q and the postcondition R. It should also be that if we reason about the postcondition Q and R, then we should also be able to reason about $Q \wedge R$.

Proposition 2 (Conjunctivity)**.** *For any program $\mathcal{P}$ and postcondition Q and R, we have*

$$\mathrm{wp}(\mathcal{P}, Q) \wedge \mathrm{wp}(\mathcal{P}, R) \equiv \mathrm{wp}(\mathcal{P}, Q \wedge R)$$

for all states.

The third healthiness property is about disjunctive predicate transformers. This property ensures that if we are in a state where the execution of $\mathcal{P}$ will satisfy Q or the execution of $\mathcal{P}$ will satisfy R, then from the same state we should be able execute the program to satisfy the postcondition $Q \vee R$. Unlike conjunctivity, this property does not hold in the other direction in the presences of non-determinism. We will refer to this as *weak* disjunctivity.

Proposition 3 (Weak Disjunctivity)**.** *For any program $\mathcal{P}$ and postcondition Q and R, we have*

$$\mathrm{wp}(\mathcal{P}, Q) \vee \mathrm{wp}(\mathcal{P}, R) \Rightarrow \mathrm{wp}(\mathcal{P}, Q \vee R)$$

for all states.

A program is *deterministic* if the final state of a program will always be the same for a given initial state. If a program is deterministic then we can strengthening the disjunctivity of wp to also include the other direction of the implication. That is, if we know that if a program terminates in a state that satisfies the postcondition $Q \vee R$, then we know that the program will either terminate in a state that satisfies Q, or it will terminate in a state satisfying R. We will refer to this as *strong* disjunctivity.

Proposition 4 (Strong Disjunctivity)**.** *For any deterministic program $\mathcal{P}$ and postcondition Q and R, we have*

$$\text{wp}(\mathcal{P}, Q) \vee \text{wp}(\mathcal{P}, R) \equiv \text{wp}(\mathcal{P}, Q \vee R)$$

for all states.

The proof of these properties on wp (and wlp) is straightforward and left out as they can be found in the literature [24]. By the pairing condition, every property on the wp also holds for the wlp. A reader familiar with the *classical* healthiness properties of the predicate transformers would have noticed by now that we have left out the property of strictness.

2.2.2 Strictness and Feasibility

Our language allows for the use of partial commands. Partial commands [56, 53, 4] are commands that do not satisfy the *Law of Excluded Miracle* [23]. This law states that every command in the programming language must be *total*. That is, the semantic function describing the command must be total for all states. If it is not total, then the semantic function describing the command may not be valid for some states in the program. Our programming language admits such partial commands through the assume command.

Partial commands can generally not be realized by a computing machine and are justifiably left out of computer languages [23]. However, partial commands are used extensively in the modeling and verification of programs [47] as they allow the verification environment to abstract away from implementation details. The assume command represents the simplest of the partial commands. The command simply checks the state

for the given condition and does nothing if the condition is satisfied in the state. If the condition is not satisfied, a *miracle* is said to occur. Once a miracle has occurred, anything is possible and any postcondition is satisfiable. As such, this command has no direct counterpart in existing programming languages as it is not implementable. For example, the program

$$\texttt{assume x} == \texttt{0}$$

is not strict as there is a state, any value of `x` not equal to `0`, that will produce the impossible postcondition $\bot$. We define strictness using the wp transformer.

Definition 4 (Strict)**.** For any program $\mathcal{P}$, if the program satisfies the condition

$$\text{wp}(\mathcal{P}, \bot).\sigma \equiv \bot$$

for all states σ, then the program is called *strict* or *total*.

If a program is not strict, then we say it is *partial*. Every command except for assumptions are total in our language. Partial programs admit another category of programs depending if the program is only occasionally partial or if it is everywhere partial. If the program admits at least one initial state that does not produce a miracle, then the program is said to be *feasible*. If the program contains no initial states that will not lead to a miracle, then the program is *infeasible*.

Definition 5 (Feasible)**.** For any program $\mathcal{P}$, if there exists a state σ such that

$$\text{wp}(\mathcal{P}, \bot).\sigma \equiv \bot$$

then $\mathcal{P}$ is said to be feasible. If there does not exist such a state, the program is said to be infeasible.

For example, the program

$$\texttt{assume x} == \texttt{x} + \texttt{1}$$

is infeasible as there does not exist a value for `x` such that `x` be equal to itself plus one. However, the program

$$\texttt{assume x} == \texttt{0}$$

is feasible as there does exist a state for `x` that does not produce a miracle, namely `0`. A total program is feasible for all initial states.

2.2.3 Everywhere Operator

In this thesis we will make use of the "everywhere" operator [25]. This operator saves us from the notational overhead of having to make explicit the states of the program. Given a condition $\mathtt{E}$, the everywhere operator takes $\mathtt{E}$ and determines if $\mathtt{E}$ holds for all possible states. We use the notation $[\mathtt{E}]$ to mean that $\mathtt{E}$ holds everywhere. This is the same as $\forall \sigma \in \Sigma \cdot \mathtt{E}$ where $\mathrm{FV}(\mathtt{E}) \subseteq \mathrm{dom}(\Sigma)$. When using the predicate transformers we will make explicit the variables used in the state. For example, $[x = y]$ will be equivalent to writing $\forall x, y \cdot (x = y)$. We can also qualify this operator as $v : [\mathtt{E}]$ to mean that the expression $\mathtt{E}$ holds for all states contained in v. For example, $\{x\} : [x = y]$ is the same as $\forall x \cdot (x = y)$. The everywhere operator enjoys all the same properties as universal quantification.

2.3 Automatic Program Verification

Given the background theory and procedure declarations of a program, an automatic program verifier will attempt to verify the body of every procedure by checking that for every execution of the procedure body no specifications will fail. Most verifiers will attempt this by assuming the procedure's preconditions and asserting that this condition implies the weakest precondition of the procedure's body. An automatic program verifier will attempt to do this without any input from the programmer. If the verifier is unable to ascertain the validity of the program, a *sound* program verifier will notify the programmer of the possibly failing specification. Along with this notification, modern program verifiers are also able to provide the programmer location of the failure and a trace through the procedure that contains the failure [45].

We use Ξ to model a generic automatic program verifier. Given a procedure body $\mathcal{P}$ our model verifier will return a positive response $\top$ if it is able to ensure that for all executions of $\mathcal{P}$ no specifications will fail. If the verifier is unable to ensure the validity of the program program, then the verifier will respond with a negative response $\bot$ along with a tuple containing the location of the failure and a counterexample trace that represents a witness to the failure. For this thesis we will treat assert commands as failing specifications, an extension to other commands is trivial. We model an automatic verifier as a function from simple pro-

grams to positive results or negative results with counterexample traces.

Definition 6 (Automatic Program Verifier)**.** An automatic program verifier Ξ takes a global declaration and a procedure body $\mathcal{P}$ as input and attempts to verify that every execution of the body will never fail. If it is able to ascertain this, Ξ will return a positive result $\top$ meaning *success*. If the verifier is unable to ascertain the validity of the procedure body it will return a negative result $\bot$ with the program location of the failure and a program trace leading up to the failure. We model Ξ as

$$\Xi \in \mathbf{Prgs} \to \{\top\} \cup (\{\bot\} \times (\mathbb{L} \times (\mathbb{N} \to (\mathbb{L} \times \Sigma))))$$

where **Prgs** is a procedure body, Σ is a state in the procedure, $\mathbb{L}$ is the set of program labels and $\mathbb{N}$ is the natural numbers

A program verifier is considered sound if it will never report $\top$ for a procedure body that may possibly fail. If the verifier is unable to show the validity of a procedure body, for example, due to the limitations of automatic analysis, it must be conservative and return a negative response. We allow the program verifier the flexibility to use any approach it sees fit to verify a program, however it must only return a positive response $\top$ if the procedure body will never fail.

Definition 7 (Sound Verifier)**.** A program verifier is considered sound if and only if for every procedure p in the program

$$\Xi(\mathcal{T}, \mathcal{C}, \mathcal{F}, \mathcal{A} \vdash \texttt{assume Pre(p)}; \text{body}(p); \texttt{assert Post(p)}) = \top$$
$$\Rightarrow$$
$$\mathcal{T}, \mathcal{C}, \mathcal{F}, \mathcal{A} \vdash [\text{Pre}(p) \Rightarrow \text{wp}(\text{body}(p), \text{Post}(p)]$$

where $\mathcal{T}$ is the declared types, $\mathcal{C}$ are the constants, $\mathcal{F}$ are the functions, and $\mathcal{A}$ are the axioms of $\mathcal{B}$.

It is important to note that the other direction does not have to necessarily hold. This property is called *completeness* and is generally not necessary as the analysis required to verify interesting programs is beyond the limitations of known computational measures [30]. This has the consequences that semantically correct programs in the wp may not be verifiable by Ξ. From this point forward when we make use of **Prgs** we will be referring to only the imperative part of a program.

2.4 Target Languages

The imperative part of our programming language is rather primitive. It does not contain the usual programming language commands such as conditional choice, looping, aliasing, or procedure calls. However, the language that we have presented is sufficiently complex to capture the semantic meaning of these more complicated commands. In this section we introduce a function Tr[] that given a program in a source language will produce a program in our intermediate language that preserves the *meaning* of the source program.

2.4.1 Conditional command

The conditional command allows us to make a choice between which program segments should be executed based on a test of the current state. Given a command of the form `if` E `then` $S0$ `else` $S1$ `fi` the command will execute $S0$ if the evaluation of `E` in the current state evaluates to true. If it does not evaluate to true, then the command $S1$ will be executed. For example, in the conditional

$$\texttt{if } x < 0 \texttt{ then } x := x + 1 \texttt{ else } x := x - 1 \texttt{ fi}$$

the assignment to `x` with the value of `x` + `1` will be executed if the initial value of `x` is strictly less than `0`. If the value of `x` is not strictly less than `0`, then the conditional command will execute the assignment to `x` with the value of `x` − `1`. We refer to the expression `x` < `0` as the *guard* of the command.

Translating this into our intermediate language is straightforward. We model the branches with a non-deterministic choice operator and we model the guards by assuming that either it holds, executing the first branch, or that the guard does not hold, executing the last branch.

$$\begin{aligned}&\mathrm{Tr}[\texttt{if (E) } S0 \texttt{ else } S1 \texttt{ fi}] \mathrel{\widehat{=}} \\ &\quad \texttt{assume E}; \mathrm{Tr}[S0] \;||\; \texttt{assume !E}; \mathrm{Tr}[S1]\end{aligned}$$

where `!E` is the negation of the expression `E` and the translation preserves the labels from the original program.

2.4.2 Repetition Command

In addition to languages with a conditional command, most modern programming languages offer a `while` command which allows the program to iterate the body of the command a number of times until the guard of the loop no longer holds. An example of such a command as it is found in modern programming languages is

$$\texttt{while } x < 0 \ \{x <= 0\} \ \texttt{do } x := x + 1 \ \texttt{od}$$

where the condition `x < 0` is the guard of the loop, the condition `x <= 0` is the loop invariant and the assignment `x := x + 1` is the body of the loop. The program first checks that the loop invariant `x <= 0` holds. If this check does not abort, the guard `x < 0` of the loop is evaluated. If the guard does not hold, the command terminates. If the guard does hold, then the body `x := x + 1` of the command is executed. When the body of the loop terminates, the invariant is then checked again. If the invariant holds, we then again check the guard. If the guard is false, the command terminates. If the guard holds, we continue with the loop execution.

Checking that the command terminates requires a variant function that describes a property that becomes smaller with each execution of the loop. However, these variant functions are extremely hard to write and verify. Instead, we only are interested to know that if the loop terminates, then it satisfies the postcondition of the program. To encode this into our intermediate language, we employ the encoding of [8]. This encoding ensures that we verify both the correctness of the body of the loop and the loop invariant. Given a repetition command we translate it as follows

$$\begin{aligned}
&\mathrm{Tr}[\texttt{while } E \ \{I\} \ \texttt{do } S \ \texttt{od}] \mathrel{\widehat{=}} \\
&\quad \texttt{assert I}; \texttt{havoc } \mathrm{def}(\texttt{S}); \texttt{assume I}; \\
&\quad (\texttt{assume E}; \mathrm{Tr}[S]; \texttt{assert I}; \texttt{assume false} \ || \ \texttt{assume !E})
\end{aligned}$$

where $\mathrm{def}(S)$ is the variables that are defined by the body of the repetition. This translation checks that the loop invariant holds and then it havocs the variables in the body of the loop. This arbitrary assignment to these variables simulates an arbitrary execution. The invariant is then

assumed to hold before we either do one execution of the body of the loop, then checking the invariant or we simply assume that we are finished with the end of the loop and we exit. As with the other translations, we assume that all labels are preserved in the translation. This encoding may surprise the reader at first, but upon inspection the reader will find that this encoding does do the trick for the verification of loop invariants and correctness of the body of the looping command.

2.4.3 Aliasing

Our intermediate language does not contain any aliasing of variables. That is, when we update a variable, we know that the value of every other variable has not changed. This property of variable *coincidence* is what makes our language well suited for program verification. However, with the use of array types we can simulate the aliasing of variables commonly found in target languages such as Spec#.

When we see an assignment of the form `x = o.f` in an object-oriented language, we know that this command updates the object pointed to by `x` to now contain the evaluation of the field `f` of the object pointed to by `o`. If we see an assignment of the form `o.f = x` then we know that this updates the reference of the field `f` of the object pointed to by `o` to the object referenced to by `x`. For example, the assignment of `this.next = n` assigns to the field `next` of `this`, as declared by the static type of `this`, the reference pointed to by `node`. In a `C` like language, this is changing the object being pointed to by `this.next` to the address of `this`. We can encode this aliasing by making explicit the properties of the run-time heap. We do this by declaring an array $\mathcal{H}$ that goes from object locations and field names to object locations or primitive values. We then use this array to read and write to any variables or expressions that use an aliased variable. For our example, we would look up the instance of `this` with the field name `next` and then assign to it the instance of of `node`. That is,

$$\mathcal{H}[\texttt{this}, \texttt{Node.next}] := \texttt{node}$$

where `Node` is the declared type for `this`. A similar translation is performed when we also read from an aliased variable. We will distinguish between the reading and the writing for our translation function. That

is when we read from an aliased variable we translate this as

$$\begin{aligned}&\mathrm{Tr}[\mathtt{x} = \mathtt{o.f}] \mathrel{\widehat{=}} \\ &\quad \mathtt{x} := \mathcal{H}[\mathtt{o}, \mathrm{typeOf}(\mathtt{o}).\mathtt{f}]\end{aligned}$$

where typeOf is a function that given a reference will return the type that declares the reference. When we write to a reference, we translate it as

$$\begin{aligned}&\mathrm{Tr}[\mathtt{o.f} = \mathtt{x}] \mathrel{\widehat{=}} \\ &\quad \mathcal{H}[\mathtt{o}, \mathrm{typeOf}(\mathtt{o}).\mathtt{f}] := \mathtt{x}\end{aligned}$$

The encoding for a full blown language such as Spec# is more complicated than what we have presented here [9, 7]. However, our encoding is sufficient to capture the main ideas and problems when tackling aliasing in target programming languages.

2.4.4 Procedure Calls

A programmer can call a procedure to modify any global state and can then save the results from the procedure. A programmer calls a procedure `P` with the command

$$\mathtt{call\ x} := \mathtt{P(E)}$$

where `E` is some expression and `x` is the result of the procedure `P`. A modular verification of a procedure does not actually *call* the procedure, but instead reasons about the procedure using its specifications. That is, the meaning of a procedure call for verification is to use the specifications of the procedure, and not the body of the procedure. For example, if we consider our example declaration of the procedure `Eat`

```
proc Eat (f : Fruit) returns (b : bool)
    requires isRipe(f)
    modifies isFull
    ensures b == true ==> isFull == true
```

and we have a call to the procedure `Eat`, we do not actually call this procedure. Instead we verify that the precondition holds, then we havoc the state of all the variables in the modifies clause and then we assume

the postcondition of the procedure. For example, calling our procedure `Eat` with the input `Banana` and storing the return value into the variable `x` will be translated into

```
assert isRipe(Banana);
havoc IsFull;
assume x == true ==> IsFull == true
```

The translation for procedure calls is

$$\mathrm{Tr}[\texttt{call x := P(E)}] \mathrel{\widehat{=}}$$
$$\texttt{assert Pre(P); havoc Mod(P); assume Post(P)})$$

where $\mathrm{Pre}(P)$ is the precondition of procedure `P` with the formal parameters replaced by the expression `E`, $\mathrm{Mod}(P)$ is the set of all global variables modified by `P`, and $\mathrm{Post}(P)$ is the postcondition of `P` with all occurrences of `result` replaced by `x`. All program labels are preserved in the translation.

2.5 Summary

We have presented a basic language that can be used to encode the semantics of more complicated languages such as Spec#. We have presented the semantics of our language using the predicate transformers wp and wlp. For the predicate transformers we have shown the healthiness properties that both transformers enjoy. We have shown how our language violates the property on strictness of programs and have shown why this is okay for the context of program verification. We have defined an automatic program verifier and what we can expect from it. Finally, we have given some example source language commands and have shown how they can be encoded in our programming language.

Our language and semantics are based on the Boogie programming language [21, 6]. The Boogie language offers the programmer more sophisticated commands (`call forall`,`where`); however these commands can be translated into our intermediate language. Our model program verifier Ξ is based on the Boogie program verifier; however, our model is applicable to all automatic program verifiers that are able to produce counterexamples for failed verification conditions.

Part I

Localizing Program Errors

Chapter 3

Slicing Total Commands

A large and complicated program is more easily understood when broken into smaller pieces. A programmer interested in the behavior of only a few variables of the program does not always need to inspect the entire program as only some parts of the program affect the behavior of those few variables. Experienced programmers do this automatically when understanding program behavior as they ignore those commands that do not affect the variables. Program slicing [68, 69, 65] automates this approach by systematically removing those commands from the program that have no effect on the variables the programmer is interested in. It is understood that by presenting the programmer with a simpler program, the programmer will be able to understand and debug the programmer more efficiently.

The program in Figure 3.1 assigns an arbitrary value to `y` using a havoc command and then assigns value `0` to `total` and the value `0` to `sum`. The program asserts that the state of `x` is equal to `z` and then non-deterministically chooses to assign the value of `y` to `sum` or to assign an arbitrary value to `z` and the value of `x * y` to `total`. A programmer trying to understand the behavior of `total` upon the exit of the program will search for those commands that updated the value of `total`. The programmer will first inspect if either of the branches are relevant to the value of `total`. Inspecting the bottom branch containing the assignment to `total` the programmer will observe that the value of `total` depends

on the values `x` and `y` and will now search for any modification to the values of `x` or `y`. As the havoc to `z` does not modify the value of `x` or `y` the programmer can safely ignore the havoc. With this branch finished the programmer can now inspect the top branch that updates the value of `sum` to `y` for any modifications to `total`. As the assignment to `sum` has no effect on on `total` the command can be ignored by the programmer.

With both branches inspected the programmer may continue inspecting the program for any more updates to either the variables `x` and `y` or the variable `total`. The assertion on `x` can be ignored as assertions do not modify the value of any variables. The assignment to `sum` can also be safely ignored by the programmer as it does not affect `x`,`y`, or `total`. However, the assignment to `total` the value of `y` cannot be ignored as the programmer is interested in the value of `total`. As this is the last command to be inspected, the programmer has now produced a *slice* of the program and will no longer see the program in Figure 3.1 but the program in Figure 3.2.

Program slicing automates this *removal of commands* by analyzing the program backwards and removing those commands that do not affect the values of the variables the programmer is interested in. Similar to the approach taken by a programmer, program slicing will proceed by inspecting both branches of our introductory program (Figure 3.1). In the bottom branch the program slicer will first inspect the assignment to `total` and as it is interested in the behavior of `total` the slicer will keep the command and remove `total` from the list of variables whose state it is interested in. The slicer will then take interest in the values of `x` and `y` and continue inspecting the program. The havoc on `z` has no effect on the value of `x` or `y` and will be removed by the program slicer.

The program slicer will also inspect the next branch for any updates to `total`. As the only command is an update to `sum` the program slicer will remove the command and finish inspecting this branch. As there are no commands in the top branch that update the value of `total`, the program slicer will remove this branch from the program. Now finished with the branches, the program slicer will continue inspecting the program for any updates to `x`, `y`, or `total`. The slicer will remove the assertion on `x` as assertions do not modify the value of any variables but keep the assignment to `total` and the havoc on `y` as it is interested in both of these variables.

```
1 havoc y;
2 total := 0;
3 sum := 0;
4 assert x == z;
5   sum := y
6 ||
7   havoc z;
8   total := x * y
```

Figure 3.1: A program that nondeterministically assigns `total` either the value `0` or the value of `x` times `y`.

```
1 havoc y;
2 total := 0;
3 skip
4 skip
5   skip
6 ||
7   skip
8   total := x * y
```

Figure 3.2: A program slice that preserves only the values of the variable `total` when executed. Commands that are removed from the program are modeled as being replaced with the identity command `skip`.

In this chapter we present our version of a program slicer. Our version of a program slicer differs from the existing program slicers in that our slicer works directly on the commands of the program and not on an intermediate representation. Once we have presented our program slicer we will formalize what it means to be a correct program slice and then use this definition to prove that our program slicer will always produce correct program slices. We will conclude with a discussion on related work and why we believe that we have the first direct proof of program slicing.

3.1 Program Slices

In this section we present our working definition of a program slice. A *program slice* is a syntactically correct subset of a program that preserves the behavior of the original program for a limited set of program variables [68]. This set of program variables is called the *slicing criterion*. Typically a slicing criterion consists of a program label and a set of variables. However, for brevity, we drop the program label from the criterion and

refer only to the set of variables as the slicing criterion.

Definition 8 (Slicing Criterion)**.** A slicing criterion $\mathcal{C}$ of a program $\mathcal{P}$ is a set of variables v where v is a subset of variables in $\mathcal{P}$ whose final values the programmer is interested in.

In this thesis we will use classical backwards program slicing [68, 65]. This type of slicing is a static analysis that works backwards through the program removing those commands that do not play a role in the behavior of the slicing criterion. A static backwards program slice must preserve two properties of the original program: firstly, the program slice must be obtainable from the original program by only removing commands. Secondly, the program slice must preserve the behavior of the original program for the slicing criterion *if* the original program terminates. The first property is a syntactic property and simply states that the program slice must be a subset of the original program. In this thesis we will model the removal of commands from the original program by replacing them with the no-op command `skip`.

The second property is a semantic property and will be formalized later in this thesis. For now it suffices to know that the second property states that for whatever input state that the original program is executed in, the program slice and the original program must have the same final value for each variable in the criterion. In our introductory program (Figure 3.1) the final value of `total` after assignment it to the value `0` will always have the value `0` in the program slice regardless of the input states of the program.

The semantic property of program slicing specifies that the original program and the program slice must both terminate and have the same final value for each of the variables in the criterion for the same input values. However, a program containing non-determinism cannot ever fulfill this semantic property as we can never know what value is assigned by a havoc or what branch is taken by the program. The final value of `total` in our introductory program (Figure 3.1) may have any allowable value of natural numbers as the havoc to `y` replaces the initial value of `y` with an arbitrary value. As we cannot know from the initial values what the final value of `total` may be we cannot enforce the classical semantic property of program slicing. We redefine this classical property of program slicing to take into account the non-determinism of programs

by requiring that a program slice must preserve all *possible* final values for each variable in the criterion.

Definition 9 (Program Slice)**.** A program $\mathcal{P}'$ is called a program slice of another possibly non-deterministic total program $\mathcal{P}$ for a criterion $\mathcal{C}$ if and only if

1. $\mathcal{P}'$ is a valid program in **Prgs** and $\mathcal{P}'$ can be obtained from $\mathcal{P}$ by only replacing the commands in $\mathcal{P}$ with `skip`, and
2. whenever $\mathcal{P}$ terminates for some input $\mathcal{I}$, then $\mathcal{P}'$ must also terminate for the input $\mathcal{I}$ and both programs must have the same possible values for each variable in $\mathcal{C}$.

There is more than one slice for a given criterion. There always exists at least one slice of a program, the program itself.

3.2 Data Dependency

Before we can present our program slicer, we must first define an auxiliary function γ that given a program $\mathcal{P}$ and a set of variables $\mathcal{C}$ will return the set of variables whose values define the variables in $\mathcal{C}$. If a command *defines* the value of a variable in the criterion $\mathcal{C}$ then the variables that the command *uses* must be included in set of variables that is returned from the function. For every variable v in the set of variables returned by the function there exists a variable in the criterion that is data dependent on v. We call this function γ and define it as a function from commands and sets of variables to sets of variables.

Definition 10 ($\gamma : \mathbf{Prgs} \times \mathbf{Vars} \to \mathbf{Vars}$)**.**

$$\gamma(\texttt{skip}, v) = v$$

$$\gamma(\texttt{havoc x}, v) = \begin{cases} v \setminus \{\texttt{x}\} & \text{if } \texttt{x} \in v \\ v & \text{otherwise} \end{cases}$$

$$\gamma(\texttt{x} := \texttt{E}, v) = \begin{cases} \mathrm{FV}(\texttt{E}) \cup (v \setminus \{\texttt{x}\}) & \text{if } \texttt{x} \in v \\ v & \text{otherwise} \end{cases}$$

$$\gamma(\texttt{assert Q}, v) = v$$

$$\gamma(S0;S1, v) = \gamma(S0, \gamma(S1, v))$$
$$\gamma(S0||S1, v) = \gamma(S0, v) \cup \gamma(S1, v)$$

Program labels do not affect the function and are left out for brevity. The command `skip` does not define any variables. The command `havoc x` defines the variable `x` by assigning an arbitrary value to it. If `x` is in the criterion, we remove it as we are no longer interested in the assignments to `x`. If `x` is not in the criterion, we treat it as `skip` and return the criterion untouched. The assignment command `x := E` assigns the value of the expression `E` to `x`. If `x` is in the criterion we remove it and add the free variables of `E` to the criterion as these variables are used to define the value of `x`. Otherwise, we treat it as `skip` and return the criterion. Assertion commands do not modify any state and are treated the same as `skip`. Sequential composition of the form $S0; S1$ works by first analyzing $S1$ for any dependencies and then uses these newly computed dependencies for the criterion for analyzing $S0$. This backwards analysis ensures that we get all transitive dependencies of variables that may have an affect on the slicing criterion. As we cannot know which branch of a non-deterministic choice is taken, γ will analyze both branches of a non-deterministic choice and merge the results together, thus ensuring that we know all possible dependencies that my be introduced by either branch.

Applying γ

To understand how γ works we will compute the data dependencies for our introductory program (Figure 3.1) for the initial criterion {`total`} manually. To manage the complexity of the computation we will analyze the individual blocks – sequential units of commands – of the program separately. We will refer to the first block as the header of the program and we will call it $P0$. For brevity, we leave out the program labels as they have no effect on the results. We define $P0$ as

$$P0 \mathrel{\widehat{=}} \texttt{havoc y}; \texttt{total} := 0; \texttt{sum} := 0; \texttt{assert x} == \texttt{z}$$

We will refer to the bottom branch of the program as $P1$. We define $P1$ as

$$P1 \mathrel{\widehat{=}} \texttt{havoc z};\ \texttt{total} := \texttt{x} * \texttt{y}$$

and we will refer to the top branch of the program as $P2$.

$$P2 \mathrel{\widehat{=}} \texttt{sum} := \texttt{y}$$

Using our abbreviations $P0$, $P1$, and $P2$ we can present how the analysis works for the entire program $P0; (P1||P2)$. Given the criterion $\{\texttt{total}\}$, the analysis $\gamma(P0; (P1||P2), \{\texttt{total}\})$ first works by analyzing $P1$ and $P2$ and then joining the results. Applying the definition of γ for non-deterministic choice we are given the step in the analysis where $\gamma(P0, \gamma(P1, \{\texttt{total}\}) \cup \gamma(P2, \{\texttt{total}\}))$. The following steps are used to compute $\gamma(P1, \{\texttt{total}\})$:

$$\begin{aligned}
&\gamma(\texttt{havoc z; total} := \texttt{x} * \texttt{y}, \{\texttt{total}\}) \\
=\ &\{\text{definition of } \gamma \text{ for assignment, } \texttt{total} \in \{\texttt{total}\}\} \\
&\gamma(\texttt{havoc z}, \{\texttt{x}, \texttt{y}\}) \\
=\ &\{\text{definition of } \gamma \text{ for havoc, } \texttt{z} \notin \{\texttt{x}, \texttt{y}\}\} \\
&\{\texttt{x}, \texttt{y}\}
\end{aligned}$$

which gives us the new criterion $\{\texttt{x}, \texttt{y}\}$. Computing $\gamma(P2, \{\texttt{total}\})$ gives us the steps

$$\begin{aligned}
&\gamma(\texttt{sum} := \texttt{y}, \{\texttt{total}\}) \\
=\ &\{\text{definition of } \gamma \text{ for assignment, } \texttt{sum} \notin \{\texttt{total}\}\} \\
&\{\texttt{total}\}
\end{aligned}$$

which will give us the new criterion $\{\texttt{total}\}$ as the assignment to `sum` does not affect the value of `total`. With the results of analyzing $P1$ and $P2$ for the criterion $\{\texttt{total}\}$ we are now in the state of the analysis where $\gamma(P1, \{\texttt{x}, \texttt{y}\} \cup \{\texttt{total}\})$ which is equal to $\gamma(P1, \{\texttt{x}, \texttt{y}, \texttt{total}\})$. By analyzing the non-deterministic choice with γ and the criterion $\{\texttt{total}\}$ gives us the new criterion $\{\texttt{x}, \texttt{y}, \texttt{total}\}$. Using this new criterion to analyze $P0$ will give us the following steps.

$$\gamma(\texttt{havoc y; total} := 0; \texttt{ sum} := 0; \texttt{assert x} == \texttt{z}, \{\texttt{x}, \texttt{y}, \texttt{total}\})$$

$= \{\text{definition of } \gamma \text{ for assertions}\}$
$\gamma(\texttt{havoc y; total} := 0; \texttt{sum} := 0, \{\texttt{x}, \texttt{y}, \texttt{total}\})$
$= \{\text{definition of } \gamma \text{ for assignment, } \texttt{sum} \notin \{\texttt{x}, \texttt{y}, \texttt{total}\}\}$
$\gamma(\texttt{havoc y; total} := 0, \{\texttt{x}, \texttt{y}, \texttt{total}\})$
$= \{\text{definition of } \gamma \text{ for assignment, } \texttt{total} \in \{\texttt{x}, \texttt{y}, \texttt{total}\}\}$
$\gamma(\texttt{havoc y}, \{\texttt{x}, \texttt{y}\})$
$= \{\text{definition of } \gamma \text{ for havoc, } \texttt{y} \in \{\texttt{x}, \texttt{y}\}\}$
$\{\texttt{x}\}$

where $\{\texttt{x}\}$ is the only variable whose initial state will have an influence on the variables of the original criterion $\{\texttt{total}\}$. This is because `total` has been assigned the constant `0` and the value of `y` was assigned by the havoc, thus removing them from the criterion. Given the analysis of our program $\gamma(P0; (P1||P2), \{\texttt{total}\})$ we have the final criterion $\{\texttt{x}\}$.

3.3 A Total Command Slicer

With a working definition of a program slice and our auxiliary function γ, we can construct our version of a program slicer. A program slicer works by inspecting each command of the program and determining if the command could possibly define a variable in the slicing criterion. If the slicer is able to determine that the command does not have an effect on any variable in the criterion it will remove the command from the program. We call our version of a program slicer Δ. We model Δ as a program transformer. To simplify the presentation we model the removal of basic commands by replacing the command with the command `skip`. We define the slicer for every command in **Prgs**.

Definition 11 ($\Delta : \textbf{Prgs} \times \textbf{Vars} \to \textbf{Prgs}$)**.**

$$\Delta(\texttt{skip}, v) = \texttt{skip}$$

$$\Delta(\texttt{havoc x}, v) = \begin{cases} \texttt{havoc x} & \text{if } \texttt{x} \in v \\ \texttt{skip} & \text{otherwise} \end{cases}$$

$$\Delta(\texttt{x} := \texttt{E}, v) = \begin{cases} \texttt{x} := \texttt{E} & \text{if } \texttt{x} \in v \\ \texttt{skip} & \text{otherwise} \end{cases}$$
$$\Delta(\texttt{assert Q}, v) = \texttt{skip}$$
$$\Delta(S0;S1, v) = \Delta(S0, \gamma(S1, v));\Delta(S1, v)$$
$$\Delta(S0||S1, v) = \Delta(S0, v)||\Delta(S1, v)$$

The command `skip` has no effect and is its own slice. The havoc command assigns an arbitrary value to the variable that is being havoced. If the variable is in the criterion, the program slicer keeps the havoc. If the variable is not in the criterion, then the program slicer replaces it by `skip`. The assignment `x := E` updates the variable `x` with the evaluation of `E`. If the variable `x` is in the criterion, then the slicer keeps the assignment. If `x` is not in the criterion then the program slicer replaces the command by `skip` as it has no effect on the variables in the criterion. The slicer replaces all assertions from the program as they only use variables and do not define them. For sequential composition $S0; S1$, the program slicer works backwards through the program first slicing $S1$ and then slicing $S0$ with the data dependencies computed by γ for the criterion in $S1$. The non-deterministic choice operator is sliced by slicing each branch of the choice and then merging the slices back using the choice operator. The program slicer does not modify the labels of command and preserves them when slicing.

For array updates and reads we treat an update or read in an index in an array `m` as an update and read to the entire array `m`. This has the consequence that if we are only interested in a value contained in the array at an index `E`, we must keep all updates to the array even though these updates may not affect the index `E` we are interested in. The problem is that it generally is not possible to compute if two indexes are equivalent or not, thus it generally is not possible to determine if an update to an array at index `E` has an effect on the value of the index `F` into the array we are interested in. This treatment of arrays is the approach taken in the program slicing literature [68, 65, 50] and not a limitation of this thesis. Applying a points-to [13] or similar static analysis will help produce finer grained slices in the presence of arrays.

3.3.1 Applying Δ

To understand how Δ works we will slice our introductory example (Figure 3.1) with the criterion $\{\texttt{total}\}$. To manage the complexity of the computation we will reuse the program blocks $P0$, $P1$, and $P2$ from the working example of γ. The first step in the program slicing of $\Delta(P0; (P1||P2), \{\texttt{total}\})$ is to slice the program $P1$ and the program $P2$ with the criterion $\{\texttt{total}\}$. The steps to slice the program $P1$ for the criterion $\{\texttt{total}\}$ works as follows:

$$
\begin{aligned}
&\Delta(\texttt{havoc z}; \texttt{total} := \texttt{x} * \texttt{y}, \{\texttt{total}\}) \\
=\ &\{\text{definition of } \Delta \text{ for composition}\} \\
&\Delta(\texttt{havoc z}, \gamma(\texttt{total} := \texttt{x} * \texttt{y}, \{\texttt{total}\})); \Delta(\texttt{total} := \texttt{x} * \texttt{y}, \{\texttt{total}\}) \\
=\ &\{\text{definition of } \Delta \text{ for assignment, } \texttt{total} \in \{\texttt{x}, \texttt{y}, \texttt{total}\}\} \\
&\Delta(\texttt{havoc z}, \gamma(\texttt{total} := \texttt{x} * \texttt{y}, \{\texttt{total}\})); \texttt{total} := \texttt{x} * \texttt{y} \\
=\ &\{\text{definition of } \gamma \text{ for assignment, } \texttt{total} \in \{\texttt{total}\}\} \\
&\Delta(\texttt{havoc z}, \{\texttt{x}, \texttt{y}\}); \texttt{total} := \texttt{x} * \texttt{y} \\
=\ &\{\text{definition of } \Delta \text{ for assignment, } \texttt{x} \notin \{\texttt{x}, \texttt{y}\}\} \\
&\texttt{skip}; \texttt{total} := \texttt{x} * \texttt{y}
\end{aligned}
$$

The slice of $P1$ keeps the assignment to `total` as it is in the criterion. The steps to slice $P2$ for the criterion $\{\texttt{total}\}$ is as follows.

$$
\begin{aligned}
&\Delta(\texttt{sum} := 0, \{\texttt{total}\}) \\
=\ &\{\text{definition of } \Delta \text{ for assignment}\} \\
&\texttt{skip}
\end{aligned}
$$

After we slice both $P1$ and $P2$ for the criterion $\{\texttt{total}\}$ we can join the two program slices using the non-deterministic choice operator and then continue slicing the rest of the program. We are currently at the step in the analysis where we have sliced both branches of the program and need to compute the data dependency of the branches for $\{\texttt{total}\}$

so that we can slice $P0$. From our example analysis for γ, we know that the set of data dependent variables given by γ for both branches for the criterion $\{\texttt{total}\}$ is the new criterion $\{\texttt{x}, \texttt{y}, \texttt{total}\}$. With this new criterion we slice $P0$ as follows.

$\Delta(\texttt{havoc y}; \texttt{total} := 0; \texttt{sum} := 0, \{\texttt{x}, \texttt{y}, \texttt{total}\})$
$= \{\text{definition of } \Delta \text{ for composition}\}$
$\Delta(\texttt{havoc y}; \texttt{total} := 0, \gamma(\texttt{sum} := 0, \{\texttt{x}, \texttt{y}, \texttt{total}\})); \Delta(\texttt{sum} := 0, \{\texttt{x}, \texttt{y}, \texttt{total}\})$
$= \{\text{definition of } \Delta \text{ for assignment, } \texttt{sum} \notin \{\texttt{x}, \texttt{y}, \texttt{total}\}\}$
$\Delta(\texttt{havoc y}; \texttt{total} := 0, \gamma(\texttt{sum} := 0, \{\texttt{x}, \texttt{y}, \texttt{total}\})); \texttt{skip}$
$= \{\text{definition of } \gamma \text{ for assignment, } \texttt{sum} \notin \{\texttt{x}, \texttt{y}, \texttt{total}\}\}$
$\Delta(\texttt{havoc y}; \texttt{total} := 0, \{\texttt{x}, \texttt{y}, \texttt{total}\}); \texttt{skip}$
$= \{\text{definition of } \Delta \text{ for composition}\}$
$\Delta(\texttt{havoc y}, \gamma(\texttt{total} := 0, \{\texttt{x}, \texttt{y}, \texttt{total}\})); \Delta(\texttt{total} := 0, \{\texttt{x}, \texttt{y}, \texttt{total}\}); \texttt{skip}$
$= \{\text{definition of } \Delta \text{ for assignment, } \texttt{total} \in \{\texttt{x}, \texttt{y}, \texttt{total}\}\}$
$\Delta(\texttt{havoc y}, \gamma(\texttt{total} := 0, \{\texttt{x}, \texttt{y}, \texttt{total}\})); \texttt{total} := 0; \texttt{skip}$
$= \{\text{definition of } \gamma \text{ for assignment, } \texttt{total} \in \{\texttt{x}, \texttt{y}, \texttt{total}\}\}$
$\Delta(\texttt{havoc y}, \{\texttt{x}, \texttt{y}\})); \texttt{total} := 0; \texttt{skip}$
$= \{\text{definition of } \Delta \text{ for assignment, } \texttt{y} \in \{\texttt{x}, \texttt{y}\}\}$
$\texttt{havoc y}; \texttt{total} := 0; \texttt{skip}$

Once the program slicer has finished slicing the block $P0$ it will take the results from slicing $P1$ and $P2$ and produce the program slice $\Delta(P0, \{\texttt{x}, \texttt{y}, \texttt{total}\}; (\Delta(P1, \{\texttt{total}\}) || \Delta(P2, \{\texttt{total}\}))$ for our introductory program. Observing our program slice for our introductory program we can see that it satisfies both the syntactic and semantic properties of program slicing. It is syntactically correct as all commands that have been removed have been replaced by `skip`. And, it is semantically correct as it preserves all possible values for `total` for any given value of `x`.

3.4 Correctness of the Approach

In the previous section we constructed a program slicer Δ and showed how it is used to slice our introductory program. In this section we will formalize our working definition of a program slice and then use the formalization to prove that our program slicer Δ only produces valid program slices. We will first formalize the syntactic property of program slicing and show that Δ will always satisfy this property, regardless of the input program or criterion. After this we will then formalize the semantic property of program slicing and prove that Δ will always return semantically valid program slices for any program and criterion.

3.4.1 Syntactic Correctness

The first property of our working definition of program slicing is that the program slice must be a program obtainable from the original program by only replacing commands by `skip`. This property is a very simple property and is very easy to formalize and prove. However, it is worth the effort to do so as it gives us a way to talk about one program being *simpler* than another program.

A program $\mathcal{P}'$ is considered to be simpler to understand than a program $\mathcal{P}$ if both programs share the same set of labels and for every shared program label the command at the program label in $\mathcal{P}'$ is the same as the command at the program label in $\mathcal{P}$ or the command is just `skip`. We will use the symbol $\propto$ to mean "is simpler than" and $\mathcal{P} \propto \mathcal{P}'$ to mean that the program $\mathcal{P}'$ is simpler than the program $\mathcal{P}$. We define $\propto$ here.

Definition 12 ($\propto$: **Prgs** $\times$ **Prgs** $\rightarrow \{\top, \bot\}$)**.** Suppose $\mathcal{P} \in$ **Prgs**, $\mathcal{P}' \in$ **Prgs**, and $\mathcal{L} \in \mathbb{L}$. Then,

$$\mathcal{P} \propto \mathcal{P}' \mathrel{\widehat{=}} \forall \ell \in \mathcal{L} \cdot \mathcal{P}[\ell] \neq \mathcal{P}'[\ell] \Rightarrow \mathcal{P}'[\ell] = \texttt{skip}$$

where $\mathcal{P}[\ell]$ returns the command located at program location ℓ in program $\mathcal{P}$.

The definition of $\propto$ is reflexive, transitive, and anti-symmetric; thus making $\propto$ a partial ordering over the programs in **Prgs**. It is trivial to see that the program slice for our introductory program trivially satisfies this property as every command is either kept or replaced by `skip` in the program slice. We can show this holds for all programs produced by Δ.

Theorem 1. *Suppose* $\mathcal{P} \in \mathbf{Prgs}$*, and* $v \in \mathbf{Vars}$*. Then,*

$$\mathcal{P} \propto \Delta(\mathcal{P}, v)$$

Proof. By induction on the structure of $\mathcal{P}$. As Δ either keeps the command or it replaces it by `skip`, the proof is trivial. □

By producing only syntactically correct programs that are ordered by $\propto$, our program slice preserves the first property of classical program slicing; that program slices must be obtainable from the original program by only replacing commands by `skip`. However, it does not offer any indication as to the semantic correctness of a program slice as a slicer that only preserves the $\propto$ ordering can produce programs with all basic commands replaced by `skip` without any regard to the meaning of the program.

3.4.2 Semantic Equivalence

The second property of our working definition of program slicing is that when the original program terminates for some input, then the program slice must also terminate and have the same possible final values as the original program for the variables in the criterion. A program slicer that produces only program slices that satisfy this property and the syntactic property are said to be *correct*. To show that our program slicer is correct we have to prove that it satisfies this semantic property. However, we have to first define what this semantic property means for our programming language and our semantics.

We believe the predicate transformer semantics are well suited for the task of defining the semantic equivalence for program slicing. However, it appears to be challenging to come up with the right equivalence using the predicate transformer semantics [15, 14, 66]. To motivate why we believe that our equivalence is the right equivalence we will proceed to construct our semantic equivalence in three parts: what are the conditions that ensure that a program slice will terminate when the original program terminates, how do we project away unnecessary state from the predicate transformer, and finally, how do we ensure that both programs agree on all possible values of the variables in the criterion.

Termination Conditions

The first part of our semantic property is that if the original program $\mathcal{P}$ terminates, then the program slice $\mathcal{P}'$ must terminate. The predicate transformer wp computes a precondition that characterizes the set of input states that will lead to termination and satisfy some predicate R given by the programmer. By using $\top$ as the predicate for R, we can compute the necessary terminating conditions for the program using wp. That is, $\text{wp}(\mathcal{P}, \top)$ characterizes those states that always will lead to $\mathcal{P}$ terminating. To enforce that the program slice terminates whenever the original program terminates we use the refinement ordering $\Rightarrow$.

$$[\text{wp}(\mathcal{P}, \top) \Rightarrow \text{wp}(\mathcal{P}', \top)]$$

For our introductory program (Figure 3.1) computing wp for the predicate $\top$ will give us the condition $0 \leq x$. If the initial state of `x` is equal to `sum`, then the program is guaranteed to terminate as the assertion is the only possibly non-terminating command. The program slice for our introductory program (Figure 3.2) has $\top$ as the terminating condition. That is, this program will always terminate as there are no possibly non-terminating condition in the program, thus satisfying our termination property.

Projected Predicates

When we slice a program we are interested in a subset of the program's behavior. Specifically, we are only interested in the behavior of the variables in the criterion. With the predicate transformers we express behavior of a program using a postcondition R. By limiting the free variables of R we can exhibit the projection of program state to only those states that we are interested in.

Comparing a program and a program slice using the predicate transformers requires that both programs can achieve the same postcondition R. This postcondition R is an arbitrary predicate that specifies some property on all of the possible states in the program. When we want to compare a program and a program slice, we are interested in comparing those states that are contained in the slicing criterion. We achieve this in the predicate transformers by simply limiting the free variables of the

postcondition to only contain those variables in the slicing criterion. By limiting these variables we are projecting away state that we are no longer interested in. A postcondition R that limits the free-variables is called a projected predicate.

If we compute the weakest precondition for our introductory program (Figure 3.1) for an arbitrary R, we will have an equation that contains all the variables in the program as R may possibly contain those variables.

$$\forall y \cdot (x = z \wedge (R[sum := y] \wedge \forall z \cdot (R[total := x * y]))[sum := 0][total := 0])$$

However, if we limit the free variables of the predicate R to those contained in the criterion $\{\texttt{total}\}$ we get a very different precondition. This precondition is free from any of the variables that `total` is not dependent on, such as `sum` or `z`. Limiting the postcondition R such that $\mathrm{FV}(R) \subseteq \{\texttt{total}\}$ for our introductory program we have the precondition

$$\forall y \cdot (x = sum \wedge R[total := 0] \wedge R[total := x * y])$$

that only speaks about those variables that we are interested in. The assignment to `sum` is not captured by R as `sum` is not a free variable of R. The havoc on `z` is also not captured by R as $\forall z \cdot R$ is equal to R. Whenever we make us of a projected predicate, we will make it explicit with the qualifier $\mathrm{FV}(R) \subseteq \mathcal{C}$ where $\mathcal{C}$ is some set of variables.

Egli-Milner Ordering

The semantic property of program slicing states that a program slice should preserve the final terminating values of the variables in the slicing criterion. In the program refinement calculus [4] a program $\mathcal{P}'$ is said to preserve the partial behaviors of another program $\mathcal{P}$ whenever $[\mathrm{wlp}(\mathcal{P}, R) \Rightarrow \mathrm{wlp}(\mathcal{P}', R)]$ holds for an arbitrary predicate R. This tells us that whenever the program $\mathcal{P}$ does not terminate or satisfies the predicate R, then the program $\mathcal{P}'$ must not terminate or satisfy the predicate R. If we pair this condition with our terminating condition we get the equation

$$[\mathrm{wp}(\mathcal{P}, R) \Rightarrow \mathrm{wp}(\mathcal{P}', R)]$$

In the refinement calculus this is called a *total* program refinement. This implication works for proving that the program $\mathcal{P}'$ preserves at least the

behaviors of the program $\mathcal{P}$. However, it does not limit the behavior of $\mathcal{P}'$ to only those behaviors of $\mathcal{P}$. For example, if we remove a branch from the program $\mathcal{P}$ and get the program $\mathcal{P}'$, this satisfies this relationship. However, we do not want the program slice to have more behaviors than the original program. We achieve this by limiting what $\mathcal{P}'$ can do by simply reversing the implication

$$[\text{wlp}(\mathcal{P}', R) \Rightarrow \text{wlp}(\mathcal{P}, R)]$$

where R is again some arbitrary predicate. Now, we do not want to assume the termination of $\mathcal{P}'$ to help us prove the termination of $\mathcal{P}$, so this other direction is as strong as we want it to be. A program $\mathcal{P}$ and a program slice $\mathcal{P}'$ are said to satisfy the Elgi-Milner ordering [56] if they satisfy the two properties stated above. With a projected predicate R, we have now formalized the semantic property of program slicing.

3.4.3 Semantic Correctness

A program $\mathcal{P}'$ is a valid program slice for a criterion $\mathcal{C}$ of a program $\mathcal{P}$ if and only if it satisfies the Egli-Milner ordering for a projected predicate R such that $\text{FV}(R) \subseteq \mathcal{C}$. We call a program slicer *correct* if and only if for every program slice $\mathcal{P}'$ it produces this program is a semantically valid program slice of the original program. In this section we prove that our program slicer Δ produces only valid program slices when given an arbitrary program $\mathcal{P}$ and criterion $\mathcal{C}$, thus proving that our program slicer Δ is *correct*.

Bounded Dependencies

If we slice a program $\mathcal{P}$ for the criterion $\mathcal{C}$, the variables used by the resulting program should be contained within the set of variables computed by the data dependency function γ for the program $\mathcal{P}$ and criterion $\mathcal{C}$. If this property did not hold, when a program slice inspects a sequential composition command, it may remove a command that updates the state of a variable that is used to define a variable in the criterion later in the program.

For example, if we applied our data dependency operator γ on the havoc to variable `y` in our introductory program (Figure 3.2) and it did

not contain `y` in the result, then the program slicer would remove the havoc on `y`, although `y` is used in the program slice for assigning to `total`. That is, we would have an incorrect program slice. The set of variables used by a program to define the variables in $\mathcal{C}$ is equivalent to the set of free variables of the precondition computed by wp for a projected predicate R such that $\mathrm{FV}(R) \subseteq \mathcal{C}$ and we will use the free variables of the weakest precondition from now on.

To ensure that our γ function does not miss any variables that are required by the slicer, we have to show for an arbitrary program $\mathcal{P}$, criterion v, and predicate R that the set of free variables of the resulting precondition of the program slice is contained within the data dependency analysis of the program. Again, we could have simply used the set of variables used by the program in place of the weakest precondition. We state this as a lemma and give its proof here.

Lemma 1. *Suppose* $\mathcal{P} \in \mathbf{Prgs}$, $R \in \mathbf{Preds}$, $v \in \mathbf{Vars}$. *Then,*

$$\mathrm{FV}(R) \subseteq v \Rightarrow \mathrm{FV}(\mathrm{wgp}(\Delta(\mathcal{P}, v), R)) \subseteq \gamma(\mathcal{P}, v)$$

for $\mathrm{wgp} = \{\mathrm{wp}, \mathrm{wlp}\}$.

Proof. By structural induction on $\mathcal{P}$.

Case (`skip`). Trivial.

Case (`havoc x`, $x \in v$). Assume $\mathrm{FV}(R) \subseteq v$ and $x \in v$. Suppose $y \in \mathrm{FV}(\mathrm{wgp}(\Delta(\texttt{havoc x}, v), R))$. Then $y \in \mathrm{FV}(\forall x \cdot R)$ by definition of Δ and wgp. By the properties of FV, $y \in \mathrm{FV}(R) \setminus \{x\}$. From $\mathrm{FV}(R) \subseteq v$ we can conclude $y \in (v \setminus \{x\})$. By the definition of γ, we may also conclude $y \in \gamma(\texttt{havoc x}, v)$ when $x \in v$.

Case (`havoc x`, $x \notin v$). Trivial.

Case ($\texttt{x} := \texttt{E}$, $x \in v$). Assume $\mathrm{FV}(R) \subseteq v$ and $x \in v$. Suppose $y \in \mathrm{FV}(\mathrm{wgp}(\Delta(\texttt{x} := \texttt{E}, v), R))$. Then $y \in \mathrm{FV}(R[x := E])$ by definition of Δ and wgp. By the properties of FV, $y \in (\mathrm{FV}(R) \setminus \{x\} \cup \mathrm{FV}(E))$. From $\mathrm{FV}(R) \subseteq v$ we can conclude $y \in (v \setminus \{x\} \cup \mathrm{FV}(E))$. By the definition of γ, we may also conclude $y \in \gamma(\texttt{x} := \texttt{E}, v)$. Since y was an arbitrary element we can conclude $\mathrm{FV}(\mathrm{wgp}(\Delta(\texttt{x} := \texttt{E}, v), R)) \subseteq \gamma(\texttt{x} := \texttt{E}, v)$ when $x \in v$.

Case ($\texttt{x} := \texttt{E}$, $x \notin v$). Trivial.

Case (`assert Q`). Trivial.

Case ($S0;S1$). Assume $\mathrm{FV}(R) \subseteq v$. Assume the induction hypothesis on $S0$ and $S1$. Suppose $y \in \mathrm{FV}(\mathrm{wgp}(\Delta(S0;S1, v), R))$. By definition of Δ and wgp we know $y \in \mathrm{FV}(\mathrm{wgp}(\Delta(S0, \gamma(S1, v)), \mathrm{wgp}(\Delta(S1, v), R)))$. From the assumption $\mathrm{FV}(R) \subseteq v$ and the induction hypothesis on $S1$, we can conclude $\mathrm{FV}(\mathrm{wgp}(\Delta(S1, v), R)) \subseteq \gamma(S1, v)$. Instantiating the induction hypothesis on $S0$ with $\mathrm{FV}(\mathrm{wgp}(S1, R))$ and $\gamma(S1, v)$ we can conclude $\mathrm{FV}(\mathrm{wgp}(\Delta(S0, \gamma(S1, v)), \mathrm{wgp}(\Delta(S1, v), R))) \subseteq \gamma(S0, \gamma(S1, v))$. Assuming $y \in \mathrm{FV}(\mathrm{wgp}(\Delta(S0, \gamma(S1, v)), \mathrm{wgp}(\Delta(S1, v), R)))$, we can assume $y \in \gamma(S0, \gamma(S1, v))$. By definition of γ for sequential composition, $y \in \gamma(S0; S1, v)$. Since y was an arbitrary element, we can conclude $\mathrm{FV}(\mathrm{wgp}(\Delta(S0; S1, v), R)) \subseteq \gamma(S0; S1, v)$.

Case ($S0||S1$). Assume $\mathrm{FV}(R) \subseteq v$. Assume the induction hypothesis on $S0$ and $S1$. Suppose $y \in \mathrm{FV}(\mathrm{wgp}(\Delta(S0||S1, v), R))$. Either $y \in \mathrm{FV}(\mathrm{wgp}(\Delta(S0, v), R))$ or $y \in \mathrm{FV}(\mathrm{wgp}(\Delta(S1, v), R))$. Let us suppose $y \in \mathrm{FV}(\mathrm{wgp}(\Delta(S0, v), R))$. From $\mathrm{FV}(R) \subseteq v$ and the induction hypothesis on $S0$ we can conclude $\mathrm{FV}(\mathrm{wgp}(\Delta(S0, v), R) \subseteq \gamma(S0, v)$. From this we can conclude $y \in \gamma(S0, v)$. Suppose $y \in \mathrm{FV}(\mathrm{wgp}(\Delta(S0, v), R)$. From $\mathrm{FV}(R) \subseteq v$ and the induction hypothesis on $S1$ we can conclude $\mathrm{FV}(\mathrm{wgp}(\Delta(S1, v), R) \subseteq \gamma(S1, v)$. From this we can conclude $y \in \gamma(S1, v)$. As y is either in $\gamma(S0, v)$ or $\gamma(S1, v)$ we can conclude $y \in \gamma(S0, v) \cup \gamma(S1, v)$. By definition of non-deterministic choice this is equivalent to $y \in \gamma(S0||S1, v)$. Since y was an arbitrary element, we can conclude $\mathrm{FV}(\mathrm{wgp}(\Delta(S0||S1, v), R)) \subseteq \gamma(S0||S1, v)$ holds for all elements.

□

Proof of Correctness

Given an arbitrary program $\mathcal{P}$ and an arbitrary criterion $\mathcal{C}$, we want to know that Δ will always produce a program that satisfies the semantic property of program slicing. We proceed to do this by assuming an arbitrary program, criterion, and projected predicate for the criterion and showing that the program produced by Δ satisfies our Egli-Milner ordering. We will break this proof into two parts, the first part $\Rightarrow$ comparing the wp results of the program and its slice and the second part $\Leftarrow$ which states the correctness in terms of the wlp. The proof of both of them will give the proof of our Egli-Milner ordering.

Theorem 2. *Suppose* $\mathcal{P} \in$ **Prgs**, $R \in$ **Preds**, *and* $v \in$ **Vars**. *Then,*

$$\mathrm{FV}(R) \subseteq v \Rightarrow [\mathrm{wp}(\mathcal{P}, R) \Rightarrow \mathrm{wp}(\Delta(\mathcal{P}, v), R)]$$

Proof. The proof proceeds by structural induction on the commands of $\mathcal{P}$.

Case (`skip`). Trivial.

Case (`assert Q`). Suppose wp(`assert Q`, R) holds in a state σ. Then by definition of wp, we can assume $(Q \wedge R).\sigma$ also hold. From $(Q \wedge R).\sigma$ we can conclude $R.\sigma$. By the definition of wp for `skip` and Δ for assertions we can conclude that wp(Δ(`assert Q`, R)).σ holds. As σ was arbitrary, wp(Δ(`assert Q`, R)) holds for all states.

Case (`havoc x`, `x` $\in v$). Trivial.

Case (`havoc x`, `x` $\notin v$). Assume $\mathrm{FV}(R) \subseteq v$. Suppose wp(`havoc x`, R) holds in some state σ. By definition of wp $\forall x \cdot R.\sigma$ also holds. Since $x \notin v$ and $\mathrm{FV}(R) \subseteq v$ it follows that $x \notin \mathrm{FV}(R)$. As x is not free in R, we can conclude $R.\sigma$ also holds. By the definition of wp for `skip` and the definition for Δ for havocs when $x \notin v$ we can conclude wp(Δ(`havoc x`, v), R).σ holds. As σ was arbitrary, wp(Δ(`havoc x`, v), R) holds in all states.

Case (`x` := `E`, `x` $\in v$). Trivial.

Case (`x` := `E`, `x` $\notin v$). Assume $\mathrm{FV}(R) \subseteq v$. Suppose wp(`x` := `E`, R) holds in some state σ. By definition of wp $R[x := E].\sigma$ also holds. Since $x \notin v$ and $\mathrm{FV}(R) \subseteq v$ it follows that $x \notin \mathrm{FV}(R)$. As x is not free in R, we can conclude $R.\sigma$ holds. By the definition of wp for `skip` and the definition for Δ for updates when $x \notin v$ we can conclude wp(Δ(`x` := `E`, v), R).σ holds. As σ was arbitrary, wp(Δ(`x` := `E`, v), R) holds in all states.

Case ($S0$;$S1$). Assume $\mathrm{FV}(R) \subseteq v$. Assume the induction hypothesis for $S0$ and for $S1$. Suppose wp($S0$;$S1$, R) holds for some state σ. From the definition of wp, we can conclude wp($S0$, wp($S1$, R)).σ also holds . From the assumption $\mathrm{FV}(R) \subseteq v$ and the induction hypothesis on $S1$ we can conclude wp($S1$, R)$\Rightarrow$wp($\Delta(S1, v)$, R) holds for all states. By monotonicity of wp , we can conclude wp($S0$, wp($\Delta(S1, v)$, R)).σ from the condition wp($S0$, wp($S1$, R)).σ. By Lemma 1 and our assumption $\mathrm{FV}(R) \subseteq v$ we can conclude $\mathrm{FV}(\mathrm{wp}(\Delta(S1, v), R)) \subseteq \gamma(S1, v)$. Instantiating the induction hypothesis on $S0$ with wp($\Delta(S1, v)$, R) and $\gamma(S1, v)$ and we can assume wp($S0$, wp($\Delta(S1, v)$, R)) $\Rightarrow$ wp($\Delta(S0, \gamma(S1, v))$, wp($\Delta(S1, v)$, R))

holds for all states. Assuming $\mathrm{wp}(S0, \mathrm{wp}(\Delta(S1, v), R)).\sigma$ holds, then $\mathrm{wp}(\Delta(S0, \gamma(S1, v)), \mathrm{wp}(\Delta(S1, v), R)).\sigma$ also holds. By definition of Δ and wp we have that $\mathrm{wp}(\Delta(S0; S1, v), R).\sigma$ also holds. As σ was arbitrary, we can conclude $\mathrm{wp}(S0; S1, R) \Rightarrow \mathrm{wp}(\Delta(S0; S1, v), R)$ holds for all states.

Case $(S0||S1)$. Assume $\mathrm{FV}(R) \subseteq v$. Assume the induction hypothesis for $S0$ and $S1$. Suppose $\mathrm{wp}(S0||S1, R)$ holds in some state σ. From the definition of $||$ for wp we can conclude $\mathrm{wp}(S0, R).\sigma \wedge \mathrm{wp}(S1, R).\sigma$ holds. By the induction hypothesis on $S0$ and the assumption $\mathrm{FV}(R) \subseteq v$ we can conclude $\mathrm{wp}(S0, R) \Rightarrow \mathrm{wp}(\Delta(S0, v), R)$ holds for all states. As σ satisfies $\mathrm{wp}(S0, R)$ we can conclude $\mathrm{wp}(\Delta(S0, v), R).\sigma$ also holds. By the induction hypothesis on $S1$ and the assumption $\mathrm{FV}(R) \subseteq v$ we can conclude $\mathrm{wp}(S1, R) \Rightarrow \mathrm{wp}(\Delta(S1, v), R)$ holds for all states. As σ also satisfies $\mathrm{wp}(S1, R)$ we can conclude $\mathrm{wp}(\Delta(S1, v), R).\sigma$ also holds. From $\mathrm{wp}(\Delta(S0, v), R).\sigma$ and $\mathrm{wp}(\Delta(S0, v), R).\sigma$ and the definition of wp and Δ we can conclude $\mathrm{wp}(\Delta(S0||S1, v), R).\sigma$ also holds. As σ was arbitrary we can conclude $\mathrm{wp}(S0||S1, R) \Rightarrow \mathrm{wp}(\Delta(S0||S1, v), R)$ holds for all states.

□

Theorem 3. *Suppose* $\mathcal{P} \in$ **Prgs**, $v \in$ **Vars**, *and* $R \in$ **Preds**. *Then,*

$$\mathrm{FV}(R) \subseteq v \Rightarrow [\mathrm{wlp}(\Delta(\mathcal{P}, v), R) \Rightarrow \mathrm{wlp}(\mathcal{P}, R)]$$

Proof. We proceed by induction on the structure of $\mathcal{P}$.

Case (`skip`). Trivial.

Case (`assert Q`). Suppose $\mathrm{wlp}(\Delta(\texttt{assert Q}, v), R)$ holds in a state σ. Then by definition of Δ we conclude $\mathrm{wlp}(\texttt{skip}, R)\sigma$ also holds in the same state. By definition of wlp, $R\sigma$ also holds. Since $(R \Rightarrow \neg Q \vee R).\sigma$ for any $Q.\sigma$, we conclude that $(\neg Q \vee R).\sigma$ holds. By definition of wlp for assertions we conclude $\mathrm{wlp}(\texttt{assert Q}, R).\sigma$ holds. As the state was arbitrary, we can conclude $\mathrm{wlp}(\Delta(\texttt{assert Q}, v), R) \Rightarrow \mathrm{wlp}(\texttt{assert Q}, R)$ holds for all states.

Case (`havoc x`, $\texttt{x} \in v$). Trivial.

Case (`havoc x`, $\texttt{x} \notin v$). Assume $\mathrm{FV}(R) \subseteq v$. Suppose the condition $\mathrm{wlp}(\Delta(\texttt{havoc x}, v), R)$ holds in a state σ. By definition of Δ and the assumption that `x` is not contain in v we can conclude $\mathrm{wlp}(\texttt{skip}, R).\sigma$ also

holds. By definition of wlp, $R.\sigma$ also holds. Since $\mathtt{x} \notin v$ and $\mathrm{FV}(R) \subseteq v$ it follows that $\mathtt{x} \notin \mathrm{FV}(R)$. As $\mathtt{x}$ is not free in R, we can conclude $\forall x \cdot R.\sigma$ also holds. By definition of wlp, we can conclude $\mathrm{wlp}(\mathtt{havoc\ x}, R).\sigma$ holds. As σ was arbitrary, we can conclude $\mathrm{wlp}(\Delta(\mathtt{havoc\ x}, v), R) \Rightarrow \mathrm{wlp}(\mathtt{havoc\ x}, R)$ holds for all states.

Case ($\mathtt{x} := \mathtt{E}$, $\mathtt{x} \in v$). Trivial.

Case ($\mathtt{x} := \mathtt{E}$, $\mathtt{x} \notin v$). Assume $\mathrm{FV}(R) \subseteq v$. Suppose $\mathrm{wlp}(\Delta(\mathtt{x} := \mathtt{E}, v), R)$ holds in some state σ. By definition of Δ for updates when $\mathtt{x} \notin v$ we can assume $\mathrm{wlp}(\mathtt{skip}, R).\sigma$ also holds. By definition of wlp, $R.\sigma$ holds. Since $\mathtt{x} \notin v$ and $\mathrm{FV}(R) \subseteq v$ it follows that $x \notin \mathrm{FV}(R)$. As x is not free in R, $R[x := E].\sigma$ also holds. By definition of wlp for assignment, we can conclude $\mathrm{wlp}(\mathtt{x} := \mathtt{E}, R).\sigma$ also holds. As σ was arbitrary, we can conclude $\mathrm{wlp}(\Delta(\mathtt{x} := \mathtt{E}, v), R) \Rightarrow \mathrm{wlp}(\mathtt{x} := \mathtt{E}, R)$ holds for all states.

Case ($S0;S1$). Assume $\mathrm{FV}(R) \subseteq v$. Assume the induction hypothesis for $S0$ and for $S1$. Suppose $\mathrm{wlp}(\Delta(S0;S1, v), R)$ holds for some state σ. By definition of Δ and wlp we can make the assumption that $\mathrm{wlp}(\Delta(S0, \gamma(S1, v)), \mathrm{wlp}(\Delta(S1, v), R)).\sigma$ holds. Using the assumption $\mathrm{FV}(R) \subseteq v$ and instantiating the Lemma 1 with $S1$ we can conclude $\mathrm{wlp}(\Delta(S1, v), R) \subseteq \gamma(S1, v)$ holds. Instantiating the induction hypothesis on $S0$ with $\mathrm{wlp}(\Delta(S1, v), R)$ and $\gamma(S1, v)$ we can assume the hypothesis $\mathrm{wlp}(\Delta(S0, \gamma(S1, v)), \mathrm{wlp}(\Delta(S1, v), R)).\sigma \Rightarrow \mathrm{wlp}(S0, \mathrm{wlp}(\Delta(S1, v), R))$ also holds for σ. As we assume $\mathrm{wlp}(\Delta(S0, \gamma(S1, v)), \mathrm{wlp}(\Delta(S1, v), R)).\sigma$ holds, we can conclude $\mathrm{wlp}(S0, \mathrm{wlp}(\Delta(S1, v), R)).\sigma$. From the induction hypothesis on $S1$ we can conclude $\mathrm{wlp}(\Delta(S1, v), R).\sigma \Rightarrow \mathrm{wlp}(S1, R)$ holds for σ. From $\mathrm{wlp}(S0, \mathrm{wlp}(\Delta(S1, v), R)).\sigma$ and monotonicity of wp we can conclude $\mathrm{wlp}(S0, \mathrm{wlp}(S1, R)).\sigma$. By definition of Δ and wlp we have that $\mathrm{wlp}(\Delta(S0; S1, v), R).\sigma$ holds. As the current state was arbitrary, we can conclude $\mathrm{wlp}(S0; S1, R) \Rightarrow \mathrm{wlp}(\Delta(S0; S1, v), R)$ holds for all states.

Case ($S0||S1$). Assume $\mathrm{FV}(R) \subseteq v$. Assume the induction hypothesis for $S0$ and $S1$. Suppose $\mathrm{wlp}(\Delta(S0||S1, v), R)$ holds in some state σ. From the definition of wlp for choice and Δ for choice we can assume $\mathrm{wlp}(\Delta(S0, v), R) \wedge \mathrm{wlp}(\Delta(S1, v), R).\sigma$ holds. By the induction hypothesis on $S0$ and $\mathrm{FV}(R) \subseteq v$ we can conclude $\mathrm{wlp}(\Delta(S0, v), R) \Rightarrow \mathrm{wlp}(S0, R).\sigma$. As $\mathrm{wlp}(\Delta(S0, v), R).\sigma$ holds we can conclude $\mathrm{wlp}(S0, R).\sigma$ also holds. By the induction hypothesis on $S1$ and $\mathrm{FV}(R) \subseteq v$ we can conclude $\mathrm{wlp}(\Delta(S1, v), R) \Rightarrow \mathrm{wlp}(S1, R).\sigma$ holds. As we assume $\mathrm{wlp}(\Delta(S1, v), R).\sigma$

we can conclude $\text{wlp}(S1, R).\sigma$. From $\text{wlp}(S0, R).\sigma$ and $\text{wlp}(S1, R).\sigma$ and the definition of wlp and Δ we can conclude $\text{wlp}(S0||S1, R)\sigma$ also holds. As the σ was arbitrary we can conclude $\text{wlp}(\Delta(S0||S1, v), R) \Rightarrow \text{wlp}(S0||S1, R)$ holds for all states.

□

As we have shown that Δ preserves both $\Rightarrow$ and $\Leftarrow$ of the Egli-Milner ordering we know that Δ will always produce programs that satisfy the semantic property of program slicing for an arbitrary program and criterion.

3.5 Summary and Related Work

In this chapter we have presented our version of program slicing. Our program slicer Δ works directly on the syntax of the commands in **Prgs** and reflects the *direct* nature of the predicate transformers wp and wlp. We have shown how the predicate transformers can be used to define the semantic correctness of program slicing by limiting the variables of the postcondition and using an Egli-Milner ordering. With our semantic definition of program slicing we have proved that our program slicer Δ produces only correct program slices. There have been various proofs in the literature, however by making the analysis syntax directed and using the predicate transformers wp and wlp we feel that we have presented the first direct and convincing proof for the correctness of program slicing for total commands.

The original definition of program slicing that was introduced by Weiser [68] and later fixed by Leung and Reghbati [48] is based on an iterative solution of dataflow equations that work on the control flow graph of the program. A slicing criterion consists of a pair (n, V) where n is a node in the control flow graph of the program and V is the set of variables. The iterative approach first computes the set of directly relevant variables for each node by taking only the data dependencies into account. From this set, a set of directly relevant commands is then derived by inspecting the set of variables defined by a node i and checking to see if it has an influence on a set of directly relevant variables of a node dominated by i. The process then takes into account any control

dependencies and then reiterates this process until a fixpoint is reached; which exists.

A proof of correctness for such an approach is incredibly hard to show. The general gist of how to show the correctness for such an approach is to first show that the program slice control flow graph is a sub-graph of the original program control flow graph and then to show that for every node in the program slice that it is a bi-simulation of the corresponding node in the original control flow graph [3, 67]. If two control graphs are semantically equal, then we know that the programs they are representing are also semantically equal; thus proving the correctness of the approach. The complexity of this proof is in the choice of the analyses used to compute the program slice. Verifying the correctness of analysis based on data-flow equations is known to be hard [27] and alternative approaches have been investigated [10] to aid in showing the correctness of such an analysis. By defining our program slicer Δ as a program transformer we are able to remove most of this complexity and have what we believe to be a simple and intuitive proof of the correctness of program slicing.

An alternative approach to dealing with this complexity has been to use program dependency graphs [28] as the intermediate representation for programs. Program dependency graphs extend control flow graphs to include dataflow edges for each variable in the program. If there is a data dependency in the program between two commands, then the program dependency graph will have an edge for the data dependency between the two nodes representing the two commands. Program slicing using program dependency graphs essentially boils down to removing those nodes that cannot be reached from either control flow or data flow edges from the node specified by the criterion [40, 59]. As the semantics of program dependency graphs are well understood [12], showing the correctness of the program slice is to simply show that the program dependency graph of the program slice is a subset of the original program dependency graph. As with control flow graphs, if two program dependency graphs are semantically equal for the slicing criterion; then the programs they represent are also semantically equal for the slicing criterion. This approach, as with the control flow graph approach, is considered to be an *indirect* approach as both the analysis and the proof are based on an intermediate representation of the program and not on the actual program itself. Our approach works on the program itself and reasons about the semantics of

the program without an intermediate representation. Moreover, we show that the program program slice is correct without having to go through an indirect proof on on indirect representation.

Found late in the work on this thesis was the work on denotational slicing [36]. Denotational slicing uses the denotational semantics of a programming language to drive the construction of a functional slicer that is similar to our program slicer Δ. By using the denotational semantics to drive the construction of the program slicer the author argues that the constructed slicer is correct, but does not prove it. The work on providing a lazy semantics for program slicing [17] points this out but fails to show the correctness of the approach according to their version of denotational semantics. We feel that using the predicate transformers is a better fit than the denotational semantics to show the correctness of a functional slicer such as Δ.

Using predicate transformers to define the semantic property of program slicing is not new. The work on p-slicing [15] defines a valid program slice $\mathcal{P}'$ of a program $\mathcal{P}$ as satisfying the equivalence

$$\mathrm{wp}(\mathcal{P}, R) \equiv \mathrm{wp}(\mathcal{P}', R)$$

for a predicate R. However, constraining a program slicer to only produce valid slices using this equivalence will result in program slices that are overly large. A program slicer attempting to remove a possibly non-terminating command from the program will have to first verify that the removal of the command will not have an impact on the terminating condition of the program. For example, the assert command `assert x = z` in our introductory program could not be removed from the program slice without having to show that the postcondition R is stronger than the assertion. That is, the program slicer would have to ensure that $R \Rightarrow x = z$ holds before it can safely remove the command. However, as this is an unreasonable expectation for a program slicer to show such conditions; a slicer has no choice but to keep all possibly non-terminating commands in the program slice.

The work on specification slicing [14] weakens the equality of p-slicing by only requiring one direction of the equivalence. Given a program $\mathcal{P}$, they define the equivalence for a valid program slice $\mathcal{P}'$ as

$$\mathrm{wp}(\mathcal{P}, R) \Rightarrow \mathrm{wp}(\mathcal{P}', R)$$

for a predicate R. However, this equivalence is too weak. A program slicer that produces program slices for this equivalence has the freedom to remove branches from the program that may have an effect on the predicate R. Given our introductory program (Figure 3.1), a program slicer that satisfies this equality would be allowed to remove the top branch of the program. The (de-sugared) weakest precondition for our introductory program is

$$R[total := 0] \wedge \forall y \cdot R[total := x * y]$$

If we remove a branch from the program slice, say the branch that updates the value of `total` with the value of `x * y`, then we have the equation

$$R[total := 0]$$

which trivially satisfies this equality for program slicing. Allowing the program slicer to remove branches from the program can only mislead the programmer understanding the behavior of the variables in the criterion.

The work on program slicing as a program transformation [66] uses contextual refinement[1] to define the equivalence for a program slice. Given a program $\mathcal{P}$, a program slice $\mathcal{P}'$ must satisfy the equivalence

$$\text{wp}(\mathcal{P}, R) \equiv \text{wp}(\mathcal{P}, \top) \wedge \text{wp}(\mathcal{P}', R)$$

for a predicate R. This allows the program slicer to remove possibly non-terminating commands from the program slice as the equivalence allows the proof of $\Leftarrow$ to assume the termination conditions of the original program. It also ensures that the program slicer does not remove branches that update any variables in the criterion. This equivalence is also known as a Smyth powerdomain. The difference between the Egli-Milner ordering and a Smyth powerdomain is in the $\Leftarrow$ part of the equivalence. In the Smyth powerdomain the proof of $\Leftarrow$ is allowed to assume the termination of the program slice whereas in the Egli-Milner ordering we do not assume termination of the program slice. Thus, the Egli-Milner ordering is a stronger result than the Smyth powerdomain.

[1] The authors claim it is new concept called *semi-refinement*, although it has already existed in many forms under the name *contextual* refinement.

Chapter 4

Slicing Partial Commands

Partial commands are used in the modeling and verification of programs. These commands allow us to abstract away implementation details and simplify the reasoning of programs. They are considered partial as the semantic function that describes them is not total for all states of the program [4, 56]. In the previous chapter we constructed a program slicer Δ for the total commands in our programming language, defined the semantic property of program slicing using the predicate transformers and proved that Δ only produces program slices that satisfy our semantic property of program slicing. In this chapter we will turn our focus to the slicing of partial commands.

Our introductory example for this chapter in Figure 4.1 is similar to the introductory example of the previous chapter, except that instead of assigning the value `0` to `total` and assigning the value `0` to `sum`, we assume that the states of `total` and `sum` are both equal to `0`. In the previous chapter we removed the assignment to `sum` when we sliced the program for the criterion {`total`} as the value of `sum` did not define `total`. If we mimic this removal, we will have the potential program slice in Figure 4.2 where the assumption on `sum` as been removed from the program.

Removing the assumption on `sum` appears to be a reasonable as the resulting program slice (Figure 4.2) for our introductory program keeps the assignment to `total` and the assumption that `total` has the value `0`. However, by removing the assumption on `sum` we are strengthening

```
1 assume total == 0;
2 assume sum == 0;
3   sum := y
4 ||
5   havoc z;
6   total := x * y
```

Figure 4.1: Instead of assigning the variables `total` and `sum` with the value `0`, we instead assume their initial value to be `0`.

```
1 assume total == 0;
2 skip
3   skip
4 ||
5   skip
6   total := x * y
```

Figure 4.2: A potential program slice for our introductory example. Removing the assumption that `sum` has the value `0` seems reasonable as `sum` is not in the criterion {`total`}.

the weakest precondition of the program and invalidating the semantic property of program slicing. Given the projected predicate R such that $\mathrm{FV}(R) \subseteq \{\texttt{total}\}$, the weakest precondition for our introductory program is

$$total = 0 \wedge sum = 0 \Rightarrow R \wedge R[total := x * y]$$

and the weakest precondition for our potential program slice is

$$total = 0 \Rightarrow R \wedge R[total := x * y]$$

We can observe that the weakest precondition of the potential program slice is stronger than the weakest precondition of the original program which violates the semantic property that the weakest precondition of the original program should be just a strong as the weakest precondition of the program slice. If we execute the original program in a state where `sum` has the value of something other than `0`, then a miracle occurs and we can satisfy any behavior of `total`. However, if we execute the program slice in a state where `sum` has the value of something other than `0`, no miracle occurs as there is no assumption on `sum` and we can no longer satisfy the behavior of `total` in the original program.

However, if we assume that `sum` will always have the value `0`, then we can paint a different picture of what it means to be a valid program slice. If a program is feasible, we know that there exists an initial state for the

program that will not produce a miracle. If we assume that we execute the program in this initial state, then we know that `sum` will always have the value `0` and the assume statement will behave exactly as `skip`. If we weaken our semantic property of program slicing to only consider a program slice valid if we only consider feasible states, then our potential program slice becomes a valid program slice.

In this chapter we will redefine our semantic property of program slicing to allow us the flexibility to remove some assumptions from the program. With our new definition of program slicing we will construct a new program slicer that we believe captures the programmer's intuition when understanding programs that include partial commands. We will formalize our new definition of program slicing and prove that our constructed slicer will only produce program slices that satisfy this new definition. We will conclude with some discussion and related work.

4.1 Preserving Program Slicing

We have seen that removing an assumption from a program strengthens the weakest precondition and invalidate the semantic property of program slicing. In this section we will first investigate what it takes for a program slicer to preserve the semantic property of program slicing in the presence of partial commands.

When an assume command `assume Q` is executed in a state where `Q` fails to hold, a miracle occurs and any behavior of any variable in the program can be observed. That is, an assume command has the potential to affect the behavior of all variables in the program, not just those contained within the expression if the assumption. For example, the command `assume x == x + 1` states that the value of `x` is equal to `x + 1`. Executing this command in any state will always produce a miracle and allows the programmer to express conditions on any variables in the program.

As we cannot know if an assumption will or will not produce a miracle, we must include the variables of the expression used by the assumption in the data dependency analysis. That is, the value of these variables define, albeit indirectly, the values of the variables of the criterion. We

extend our data dependency function γ to include this possibility.

$$\gamma(\texttt{assume Q}, v) = \mathrm{FV}(\texttt{Q}) \cup v$$

As the execution of the command, irrelevant of the variables contained in the expression of the command, may produce a miracle, we must keep all assumptions in the program. Although this command does not directly define the variables in a criterion, even for those that it shares, it definitely affects the behavior of the variables in the criterion. We extend our total command slicer to keep all assumptions in the program.

$$\Delta(\texttt{assume Q}, v) = \texttt{assume Q}$$

Using Δ to slice our introductory program (Figure 4.1) with the criterion $\{\texttt{total}\}$ will give us the program slice in Figure 4.3. This program slicer removes the havoc on `z` and the assignment to `sum` and keeps the assumption on `sum`, thus satisfying the semantic property of program slicing.

```
1 assume total == 0;
2 assume sum == 0;
3   skip
4 ||
5   skip
6   total := x * y
```

Figure 4.3: Keeping all assumptions in the program will preserve the semantic property of program slicing.

Computing the weakest precondition for this program with the projected predicate R such that $\mathrm{FV}(R) \subseteq \{\texttt{total}\}$ gives us the weakest precondition

$$total = 0 \wedge sum = 0 \Rightarrow R \wedge R[total := x * y]$$

which trivially satisfies the semantic property of program slicing. By treating all assumptions in the program as possible definitions for the

variables in the criterion and keeping these assumptions in the program slice we will always satisfy our semantic property of program slicing. The proof is trivial and is of very little consideration here.

If the goal of program slicing in the presence of partial commands is to preserve the semantic property of program slicing, then we are finished. There is nothing more we can do with assumptions when we need to satisfy the semantic property of program slicing. However, if we are willing to weaken the semantic property of program slicing, we can remove those assumptions from the program that we feel do not capture the programmer's intuition when understanding programs that contain partial commands.

4.2 Redefining Program Slicing

Keeping all assumptions in the program slice preserves the semantic property of program slicing. However, this goes against the spirit of program slicing; that is, removing commands that an experienced programmer is not interested in. As partial programs are only useful in the context of program verification, a programmer understanding a program with partial commands will attempt to verify that the program behaves as they expected using an assertion. If the assertion is satisfied, then the programmer has understood the program and has very little use for a program slicer.

However, if the assertion fails to hold, it is the responsibility of the programmer to understand why the program failed. When a programmer inspects a program that has failed to be verified, they inspect the program for assignments and assumptions on the variables of the failing assertion. A programmer does not inspect the program for assumptions that do not contribute directly to the state of the variables in the assertion as the programmer works under the assumption that a miracle did not happen, otherwise the program would not have failed verification.

For example, a programmer asserting that `total` has the value `0` upon exiting of our introductory program (Figure 4.1) will have to inspect the program as it is not the case that this assertion will always hold. A programmer will keep the assignment to `total` the value of $\texttt{x} * \texttt{y}$ in their mind and remove the havoc on `z` in the bottom branch and remove the assignment to `sum` in the top branch. When the programmer inspects

the assumption on `sum`, they will remove this as the value of `sum` does not define the value of `total`. The experienced programmer will however keep the assumption on `total` as this does define values of `total`.

Recall that a program is feasible if there exists at least one initial state of the program that will not produce a miracle when executed. If a programmer cannot prove an assertion R, then a programmer will not be unable to prove the stronger condition $\bot$. That is, a failed verification attempt tells us that the program is feasible for some state. If we assume that we are in this state when we compare a program with its program slice, we can remove assumptions from the program as they behave like `skip`. In our introductory example, if we assume that `sum` will always be in the state `0`, then the program slicer can safely remove the assumption as `0 == 0` is always true and will not produce a miracle.

However, we have to be careful not to remove too many assumptions from the program. If we assume that the program is executed in a feasible state, then the program slicer would have the freedom to remove all assumptions from the program, including those assumptions that define the state for the variables in the criterion. If we assume a feasible state for `total` then the program slicer has the freedom to remove the assumption on `total` as it will not produce a miracle. However, we feel that this does not capture the intuition of an experienced programmer who is trying to understand why a program has failed verification.

We need a middle ground that allows us to remove the assumptions on variables we are not interested in but keep the assumptions on the variables that we are interested in. We do this by assuming a feasible state for all those variables that the criterion does not have a dependency on and ensuring that the semantic property enforces the program slicer to preserve the values of all variables not shared by a dependency. A dependency can be created by either an assignment, as in the previous chapter, or by sharing an assumption with another variable, thus sharing the same cone of influence. As `total` is not dependent on `sum`, we assume a feasible state for `sum` and require that the program slice preserves all possible values for `total`.

We redefine the definition of program slicing to take into account this middle ground for the semantic property. We will call this new definition of program slicing *partial* program slicing.

Definition 13 (Partial Program Slice)**.** A program $\mathcal{P}'$ is called a partial program slice of another program $\mathcal{P}$ for a criterion $\mathcal{C}$ if and only if

1. $\mathcal{P}'$ is a valid program in **Prgs** and $\mathcal{P}'$ can be obtained from $\mathcal{P}$ by only replacing commands in $\mathcal{P}$ with `skip`, and

2. if $\mathcal{P}$ is feasible and $\mathcal{P}$ terminates for some input $\mathcal{I}$, then $\mathcal{P}'$ must also terminate for the input $\mathcal{I}$ and both programs must have the same possible final values for each variable in $\mathcal{C}$

We will refer to the standard definition of program slicing as *total* program slicing.

4.3 Partial Program Slicer

Weakening the semantic property of program slicing allows us to construct a program slicer that has the freedom to remove those assumptions from the program that do not define feasible initial states for the variables in the criterion. To differentiate this new slicer from the total program slicer Δ we will put a little hat on top of it and call it $\widehat{\Delta}$. We will do the same for the dependency function γ as this will also have to change. For all commands except assume commands these functions will behave identical to their total program counterparts.

If an assume command contains an expression which mentions one of the variables in the criterion, we are interested in this assumption. If the assumption contains variables that are not in the criterion, we are also interested in the value of these variables as they are used to help define the possible values for the variable in the criterion. For example, if our introductory program instead assumed `total` == `sum`, then we know that whatever value `total` has it is equal to `sum`. And, whatever value `sum` has, the value of `total` will be equal to that value. There is a dependency between the value of `total` and `sum`. We construct $\widehat{\gamma}$ to include all the free variables of an assume command if and only if it shares at least one variable with the criterion.

$$\widehat{\gamma}(\texttt{assume Q}, v) = \begin{cases} \mathrm{FV}(\texttt{Q}) \cup v & \text{if } \mathrm{FV}(\texttt{Q}) \cap v \neq \varnothing \\ v & \text{otherwise} \end{cases}$$

An experienced programmer only is interested in assumptions that state some property on the variables in the criterion. We construct the program slicer $\widehat{\Delta}$ to first check if the assume command shares any variables with the criterion. If the command does share some variables, then the slicer keeps the command. If the assumption does not share any variables with the criterion, it is removed from the program slice. We define $\widehat{\Delta}$ as

$$\widehat{\Delta}(\texttt{assume Q}, v) = \begin{cases} \texttt{assume Q} & \text{if } \mathrm{FV}(\texttt{Q}) \cap v \neq \varnothing \\ \texttt{skip} & \text{otherwise} \end{cases}$$

Applying $\widehat{\Delta}$ to our introductory program (Figure 4.1) will produce the introductory program slice (Figure 4.2). The slicer will remove the assumption on `sum` as $\mathrm{FV}(\texttt{sum} == \texttt{0})$ does not share any variables with the criterion $\{\texttt{total}, \texttt{x}, \texttt{y}\}$ but will keep the assumption that `total` as the value `0` as we are interested in the value of `total`.

The only noticeable difference between the analysis of a total command assigning the value `0` to `total` and the analysis of a partial command that assumes the value of `total` is equal to `0` is that the initial state of `total` does not play a role in the final value of `total` when it is assigned the value `0`. When we assume that `total` has the value of `0`, we are still interested in the initial value of `total` as it has not been defined to be `0`, only assumed.

4.3.1 Missed Dependencies

Applying $\widehat{\Delta}$ to our introductory program produces a correct partial program slice. When inspecting the assumption on `sum` the slicer was able to know that the assumption could be safely removed as there was not a dependency between `sum` and `total`. However, this may not always be the case and $\widehat{\Delta}$ may remove assumptions that are needed to preserve the semantic property of partial program slicing.

The program in Figure 4.4 assumes that the initial value of `total` will have the same value as `sum`. It then assumes that the initial value of `sum` is equal to `0`. By transitivity, this assumption has in effect also stated that the initial value of `total` must also be equal to `0`. Slicing this program with $\widehat{\Delta}$ and the criterion $\{\texttt{total}\}$ will give us the incorrect program slice in Figure 4.5.

```
1 assume total == sum;
2 assume sum == 0;
3   sum := y
4 ||
5   havoc z;
6   total := x * y
```

Figure 4.4: Assuming that the value of `total` is equal to `sum` and that the value of `sum` is equal to `0` also assumes that the state of `total` is equal to `0`.

```
1 assume total == sum;
2 skip
3   skip
4 ||
5   skip
6   total := x * y
```

Figure 4.5: Removing the assumption that the value of `sum` is equal to `0` also removes the assumption that `total` has the value `0`, invalidating the semantic property of partial program slicing.

The problem here is that $\widehat{\Delta}$ did not know that there was a dependency between `sum` and `total` when it inspected the second assumption on `sum` on line `2`. As it did not know of this dependency, it removed the assumption and violated the semantic program of partial program slicing as `total` is no longer assumed to have the value `0` in the program slice. For $\widehat{\Delta}$ to correctly handle this assumption, it needs to know of this dependency before it inspects the program. That is, this dependency cannot be known inline as the dependency between `total` and `sum` happens after the analysis of the assumption on `sum`. We must compute this dependency offline before we begin slicing the program.

Using the data dependency function $\widehat{\gamma}$ to inspect the program with the criterion $\{\texttt{total}\}$ will give us the set of all variables that are used to define the possible values of `total` in the program. Applying $\widehat{\gamma}$ to our example program will give us the set $\{\texttt{total}, \texttt{sum}, \texttt{x}, \texttt{y}\}$ where `sum` is included in the criterion. If we use this new criterion as the criterion for slicing the example program, $\widehat{\Delta}$ will correctly keep the assumption on `sum` as `sum` is in the criterion. First analyzing the set of dependencies in the program before we slice will allow $\widehat{\Delta}$ to correctly slice the program.

However, only applying $\widehat{\gamma}$ once to the program may not get all the dependencies necessary for $\widehat{\Delta}$ to correctly slice the program. This only

computes the first step of the data dependencies for a given criterion. If our example program contained an intermediate value `z` such that either `sum` or `total` was dependent on `z`, applying $\widehat{\gamma}$ once would miss this dependency as it would take another step to find this dependency. Using the result of the first analysis of $\widehat{\gamma}$ to analyze the program again would ensure that we pick up this intermediate dependency. The transitive dependencies between variables form a dependency graph where each iteration of $\widehat{\gamma}$ is one more level in the graph. We define a function $\widehat{\gamma}^n$ that will compute the n^{th}dependency of a program for a given criterion.

Definition 14 ($\widehat{\gamma}^n : \mathbf{Prgs} \times \mathbf{Vars} \to \mathbf{Vars}$)**.**

$$\widehat{\gamma}^n(\mathcal{P}, v) = \widehat{\gamma}(\mathcal{P}, \widehat{\gamma}^{n-1}(\mathcal{P}, v))$$
$$\widehat{\gamma}^0(\mathcal{P}, v) = v$$

For our example program computing the n^1 dependency worked as the n^1 dependency also included all the dependencies of the n^0 analysis. However, this is generally not the case as the n^{th}dependency may not be a super-set of the $n-1$ dependency due to the killing of variables by assignments. What we essentially want is the transitive closure of the data dependencies of a program before we begin slicing. To do this we define a function $\widehat{\gamma}^\infty$ that take the union of all n^{th}dependencies of a set of variables for a given program.

Definition 15 ($\widehat{\gamma}^\infty : \mathbf{Prgs} \times \mathbf{Vars} \to \mathbf{Vars}$)**.**

$$\widehat{\gamma}^\infty(\mathcal{P}, v) = \bigcup_{n=0}^{\infty} \widehat{\gamma}^n(\mathcal{P}, v)$$

The function $\widehat{\gamma}^\infty$ forms an ascending chain over the variables of a program and as such is guaranteed to terminate as the set of all variables **Vars** of a program is finite. Given a program and an initial criterion, this function will compute the set of all data dependencies in a program for a given criterion.

For example, the program in Figure 4.6 assigns the value of `sum` to `total` and then assumes that `sum` has the value `z`. The program then assumes that `z` has the value `0`, which gives us the assumption that `sum`

has the value 0, which in turn gives us the assumption that total has the value 0. Computing $\widehat{\gamma}^{\infty}$ for the criterion {total} will give us the set {total, sum, z, x, y}. It does this by first computing $\widehat{\gamma}^{1}$ for the criterion {total} which will give us the set {sum, x, y}. It will then use this set as the criterion for calling $\widehat{\gamma}^{2}$, which will give us the set {sum, z, x, y}. Joining the results of the initial criterion with the results of $\widehat{\gamma}^{1}$ and $\widehat{\gamma}^{2}$ will give us the set {total, sum, z, x, y}. Using this criterion to compute the $\widehat{\gamma}^{3}$ dependency and joining the results from $\widehat{\gamma}^{2}$ will not change from joining the results of $\widehat{\gamma}^{2}$ and $\widehat{\gamma}^{1}$, thus $\widehat{\gamma}^{\infty}$ has reached a fixpoint. Using this fixpoint to slice the program will give us the correct program slice in Figure 4.7.

```
1 total := sum;
2 assume sum == z;
3 assume z == 0;
4   sum := y
5 ||
6   havoc z;
7   total := x * y
```

Figure 4.6: An example program that contains a two step dependency between the variable z and total.

```
1 total := sum;
2 assume sum == z;
3 assume z == 0;
4   sum := y
5 ||
6   havoc z;
7   total := x * y
```

Figure 4.7: Slicing the program with the criterion computed by $\widehat{\gamma}^{\infty}$ gives us a correct, albeit large program slice.

Computing the transitive closure of the data dependencies between variables and the criterion will ensure that we can give the program slicer the knowledge it needs to correctly slice a program. However, as we can see in the program slice in Figure 4.7 the program slicer has kept more commands than it needed to ensure the correct behavior for total.

4.3.2 Dependency Context

When our analysis $\widehat{\gamma}^{\infty}$ adds a data dependency to the criterion it does not preserve the context of the dependency. That is, it does not record if the dependency is between the initial value of a variable, or the final

value, or some intermediate value. As this information is not preserved, a program slicer may produce a program slice that is either overly large or does not preserve the final values of the variables in the criterion.

The program in Figure 4.8 is similar to the program in Figure 4.4 except that the havoc on `z` has been replaced by a havoc on `sum` at program location `5`. The data dependency analysis $\widehat{\gamma}^{\infty}$ for the criterion $\{\texttt{total}\}$ will give us the set $\{\texttt{total}, \texttt{sum}, \texttt{x}, \texttt{y}\}$ for this program. Using this set to slice the program will give us the incorrect program slice in Figure 4.9 where the assumption on `sum` has been removed.

```
1 assume total == sum;
2 assume sum == 0;
3   sum := y
4 ||
5   havoc sum;
6   total := x * y
```

Figure 4.8: An example program where `sum` is killed in both branches of the program.

```
1 assume total == sum;
2 skip
3   sum := y
4 ||
5   havoc sum;
6   total := x * y
```

Figure 4.9: The value of `total` is dependent on the initial value of `sum`. However, the program slicer has already removed `sum` from the criterion as `sum` is defined in both branches of the program.

The problem here is that the program slicer did not know that the dependency between `total` and `sum` was on the initial value of `total` and `sum` and not the final value of `total` and `sum`. The program slicer kept the assignment to `sum` in the top branch and the the havoc to `sum` in the bottom branch. As `sum` was defined in both branches of the program, the slicer removed `sum` from the criterion as the value of `sum` was no longer needed. When it inspected the assumption on `sum` it was no longer interested in the value of `sum` and removed the assumption, producing a program slice that no longer defines `total` to have the value `0`.

A program in static single assignment (SSA) form [61, 2] solves this problem. The SSA form of a program makes explicit the definition con-

text of variables by only assigning to a variable once. The SSA form of a program can be constructed by iterating forward through the program replacing every variable definition with a unique variable incarnation and replacing every variable read with the current incarnation. To propagate the current value of a variable incarnation in each branch, the SSA form of a program will contain a special Φ operator. We treat the Φ operator as a special non-deterministic assignment. For example, $\texttt{total}_2 := \Phi(\texttt{total}_0, \texttt{total}_1)$ is treated semantically as $\texttt{total}_2 := \texttt{total}_0 \;||\; \texttt{total}_2 := \texttt{total}_1$.

The program in Figure 4.10 is the SSA form program for our example program in Figure 4.8. The assumption on the initial value of `total` and `sum` is now made explicit with the variables $\texttt{total}_0$ and $\texttt{sum}_0$. The assumption on the initial state of `sum` is now between an assumption on $\texttt{sum}_0$ and `0`. The assignment to `sum` in the top branch is to a different variable $\texttt{sum}_1$, representing a different state of `sum` than the initial state. In the bottom branch of the program we havoc the value of $\texttt{sum}_2$ and we assign to the variable $\texttt{total}_1$ which also represents current incarnation for `total`, $\texttt{total}_0$. The program concludes with the Φ operator.

```
1 assume total0 == sum0;
2 assume sum0 == 0;
3   sum1 := y1
4 ||
5   havoc sum2;
6   total1 := x0 * y1
7 total2 := Φ(total0, total1);
8 sum3 := Φ(sum1, sum2)
```

Figure 4.10: A single static assignment form program of our example program in Figure 4.8. Each incarnation makes explicit the context of the dependency.

```
1 assume total0 == sum0;
2 assume sum0 == 0;
3   skip
4 ||
5   skip
6   total1 := x0 * y1
7 total2 := Φ(total0, total1);
8 skip
```

Figure 4.11: Slicing the SSA form of the program with the results of $\widehat{\gamma}^{\infty}$ for the criterion $\{\texttt{total}_2\}$. Translating back out of SSA form gives us the program slice that will preserve the final values of `total`.

Applying $\widehat{\gamma}^{\infty}$ to the SSA form of the program with the variable incarnation for the criterion variable $\{\texttt{total}\}$ will give us the set of variables

$\{\texttt{total}_2, \texttt{total}_1, \texttt{total}_0, \texttt{sum}_0\}$. Using this as the criterion for slicing the SSA form of the program will give us the correct program slice in Figure 4.11. The program slicer will no longer keep the definition of `sum` in the branches as the data dependency between `total` and `sum` in the original program is now a data dependency between $\texttt{total}_0$ and $\texttt{sum}_0$ and not $\texttt{sum}_1$ nor $\texttt{sum}_2$ in the SSA form of the program. We can take the program slice out of SSA form by replacing all variable incarnations with the original variables and removing all instances of the Φ operator. By translating back out of SSA form we will have a semantically correct program slice for our example program in Figure 4.8 for the criterion $\{\texttt{total}\}$.

For the remainder of this thesis we will not make SSA form explicit unless we need to. Now that we have added dependency context to our data dependency analysis we are almost finished understanding how to correctly slice programs with partial commands.

4.3.3 Anti-Dependencies

We have constructed a transitive data dependency function $\widehat{\gamma}^{\infty}$ that given a program and a criterion computes the set of all data dependencies for the given criterion in the program. For most programs this set contains all the dependencies needed to correctly slice a program. However, there is one dependency that this does not capture, the anti-dependencies. An anti-dependency is created when a variable in an assumption is dependent on the value of a variable in the criterion, giving an assumption on the variable in the criterion.

The program in Figure 4.12 contains such an anti-dependency. The program assigns to the variable `sum` the value of $\texttt{total}_0$ and then either assumes that the value of $\texttt{sum}_1$ is equal to `0` or havocs the value of $\texttt{z}_1$ and then assigns $\texttt{total}_1$ the value of $\texttt{x}_0 * \texttt{y}_0$. The evaluation of the assumption on $\texttt{sum}_1$ in the top branch of the program is dependent on the value of $\texttt{total}_0$ as $\texttt{sum}_1$ has been assigned the value of $\texttt{total}_0$. Slicing this program for the criterion $\{\texttt{total}_2\}$ will give us the incorrect program slice in Figure 4.13.

The program slicer will remove the assumption on $\texttt{sum}_1$ and the assignment to $\texttt{sum}_1$ as $\texttt{total}_2$ does not have a dependency on $\texttt{sum}_1$. However, by removing the assumption and assignment to $\texttt{sum}_1$ we are invalidating

```
1 sum1 := total0;
2   assume sum1 == 0
3 ||
4   havoc z1;
5   total1 := x0 * y0
6 total2 := Φ(total0, total1)
7 z2 := Φ(sum0, sum1)
```

Figure 4.12: A program that contains an anti-dependency between $\texttt{total}_0$ and $\texttt{sum}_1$. The assumption on the state of $\texttt{sum}_1$ is also an assumption on the state of $\texttt{total}_0$.

```
1 skip
2   skip
3 ||
4   skip
5   total1 := x0 * y0
6 total2 := Φ(total0, total1)
7 skip
```

Figure 4.13: Slicing the program for the criterion $\{\texttt{total}_2\}$. Removing the assumption and assignment to $\texttt{sum}_1$ gives us an incorrect program slice.

the semantic property of partial program slicing. This can be seen if we compare the weakest preconditions of the program and the program slice. The weakest precondition of the program for the projected predicate R such that $\mathrm{FV}(R) \subseteq \{\texttt{total}_2\}$ is the condition

$$total_0 = 0 \Rightarrow R \wedge R[total_1 := x_0 * y_0]$$

whereas the weakest precondition for the program slice is the condition

$$R \wedge R[total_1 := x_0 * y_0]$$

We can see from the weakest precondition of the program slice that we have removed the assumption that $\texttt{total}_0$ has the value $\texttt{0}$. As we have removed this assumption from the program slice, we have violated the semantic property of partial program slicing.

When we inspect an assumption we need to know not only if there is a data dependency between the variables in the criterion and the variables in the assumption, but we also need to now if there is also an anti-dependency between these variables. We extend our $\widehat{\gamma}$ function to include an anti-dependency analysis. To differentiate it from $\widehat{\gamma}$ we will add a $+$ to the bottom right of the function. This new function $\widehat{\gamma}_+$ will behave the same as $\widehat{\gamma}$ for the data dependencies. When $\widehat{\gamma}_+$ inspects an assignment

it will add the target of the assignment to the set of anti-dependencies if there is a shared variable between the expression of the right hand side and the criterion. If an assumption shares a variable with an anti-dependency, we add the free variables of the assumption to the set of data dependencies. That is, we turn the anti-dependency into a data dependency.

Definition 16 ($\widehat{\gamma}_+ : \mathbf{Prgs} \times \mathbf{Vars} \times \mathbf{Vars} \to \mathbf{Vars} \times \mathbf{Vars}$)**.**

$$\widehat{\gamma}_+(\texttt{x} := \texttt{E}, v, v') = \begin{cases} \widehat{\gamma}(\texttt{x} := \texttt{E}, v), v' \cup \{x\} & \text{if } \mathrm{FV}(E) \cap v \neq \varnothing \\ \widehat{\gamma}(\texttt{x} := \texttt{E}, v), v' & \text{otherwise} \end{cases}$$

$$\widehat{\gamma}_+(\texttt{assert Q}, v, v') = v, v'$$

$$\widehat{\gamma}_+(\texttt{assume Q}, v, v') = \begin{cases} \mathrm{FV}(\texttt{Q}) \cup \widehat{\gamma}(\texttt{assume Q}, v), v' & \text{if } \mathrm{FV}(\texttt{Q}) \cap v' \neq \varnothing \\ \widehat{\gamma}(\texttt{assume Q}, v), v' & \text{otherwise} \end{cases}$$

$$\widehat{\gamma}_+(S0;S1, v, v') = \widehat{\gamma}_+(S0, \widehat{\gamma}_+(S1, v, v'))$$

$$\widehat{\gamma}_+(S0||S1, v, v') = \widehat{\gamma}_+(S0, v, v') \cup \widehat{\gamma}_+(S1, v, v'))$$

We have a function $\widehat{\gamma}_+^n$ that computes the set of n^{th}iteration data and anti-dependencies of a program and a function $\widehat{\gamma}_+^\infty$ that computes the transitive closure of all data and anti-dependencies of a program. The function $\widehat{\gamma}_+^\infty$ forms a point-wise ascending chain and is guaranteed to terminate. These functions do not differ very much from their counterparts $\widehat{\gamma}^n$ and $\widehat{\gamma}^\infty$. Having the program in SSA form is just as crucial to $\widehat{\gamma}_+^\infty$ as it was for $\widehat{\gamma}^\infty$.

Computing $\widehat{\gamma}_+^\infty$ for the criterion $\{\texttt{total}_2\}$ and the initial set of anti-dependencies $\varnothing$ for our example program will work as follows: in the first iteration of the program the analysis will add `sum` to the set of anti-dependencies as $\texttt{total}_0$ is contained in the set of dependencies and is being assigned to `sum`. In the second iteration of the program the analysis will inspect the assumption on `sum` and see that `sum` is an anti-dependency. It will then add `sum` to the set of data dependencies of the program. After the second iterating through the program the analysis will reach a fixpoint and terminate giving us a new criterion containing the variable `sum`.

Slicing our program with the new criterion computed by $\widehat{\gamma}_{+}^{\infty}$ will give us a program slice that preserves the semantic program of partial program slicing. Specifically, it will keep the assumption on and the assignment to `sum`, thus preserving the behavior of $\texttt{total}_2$. With $\widehat{\gamma}_{+}^{\infty}$ we can compute the set of all possible data dependencies that will allow the program slicer $\widehat{\Delta}$ to correctly remove or keep partial commands in the program slice.

To summarize, we slice a program that contains partial commands as follows: First, we transform the program $\mathcal{P}$ into SSA form and replace the variables in the criterion $\mathcal{C}$ with the last incarnation of the variables used in the SSA form of the program. We then compute the fixpoint of $\widehat{\gamma}_{+}^{\infty}$ with the criterion $\mathcal{C}$ and the empty set. We take the resulting criterion $\mathcal{C}'$, ignoring the set of anti-dependencies and use this criterion to slice the program with $\widehat{\Delta}$. With the resulting program slice we transform the program back out of SSA form and have our final program slice. In the next section we will formalize the semantic property of partial program slicing and show that this procedure produces only semantically correct program slices for programs that contain partial commands.

4.4 Semantic Correctness

Relaxing the semantic property of program slicing allowed us the flexibility to construct a program slicer Δ that removes some of the assumptions from the program. To ensure that we do not remove any assumptions that are needed to describe the behaviors of a criterion we have constructed a data dependency analysis $\widehat{\gamma}_{+}^{\infty}$ that will give us the set of data and anti-dependent variables for the criterion in a program. In this section we formalize the semantic property of partial program slicing and prove that for any fixpoint of $\widehat{\gamma}_{+}^{\infty}$ and any program $\mathcal{P}$, $\widehat{\Delta}$ only produces program slices that satisfy the semantic property of partial program slicing.

4.4.1 Partial Egli-Milner Ordering

By removing an assumption from a program we are possibly strengthening the weakest precondition of the program slice; thus invalidating the semantic property of total program slicing. To accommodate for this removal we have redefined the semantic property of program slicing to only consider a program slice to be valid if the original program is feasible.

If we assume that we execute the program in a feasible state, then we can remove assumptions from the program as they will behave as `skip` and will not have an effect on any variables, including those in the criterion. However, we do not want to allow a program slicer the flexibility to remove all assumptions from the program as we want to keep those assumptions that define the states of the variables in the criterion. We do this by requiring that the program and the program slice preserve all states of the variables in the criterion, not just the feasible states that we executes the program in.

The semantic property of total program slicing required that every program slice must satisfy the Egli-Milner ordering. That is, given a program $\mathcal{P}$, the program slice $\mathcal{P}'$ must satisfy the condition

$$\mathrm{FV}(R) \subseteq \mathcal{C} \Rightarrow [\mathrm{wp}(\mathcal{P}, R) \Rightarrow \mathrm{wp}(\mathcal{P}', R)]$$

and the condition

$$\mathrm{FV}(R) \subseteq v \Rightarrow [\mathrm{wlp}(\mathcal{P}', R) \Rightarrow \mathrm{wlp}(\mathcal{P}, R)]$$

As we remove assumptions from the program slice, we are possibly going to violate the first condition ($\Rightarrow$) and we need to redefine what this condition means for the semantic property of partial program slicing. The other condition ($\Leftarrow$) is trivially satisfied if we remove assumptions from the program and does not need to be modified.

A program $\mathcal{P}$ is feasible when there exists a state where $\mathrm{wp}(\mathcal{P}, \bot)$ does not evaluate to true. If we assume that we execute the program in this state we will have the flexibility to remove assumptions from the program as a miracle will not happen. We achieve this by assuming that the program $\mathcal{P}$ must be feasible before we compare the weakest precondition of the program and the program slice. If the program we are slicing is infeasible, then this condition will be trivially satisfied and the program slicer will have the freedom to produce any program it desires. This has the effect of freeing the program slicer from having to preserve program slices for programs that are always infeasible, even if the variables that always produce the miracle are the variables in the criterion. However, this is not a practical issue because a program that is infeasible satisfies every assertion and a programmer only needs to slice a program if their assertion failed. As with total command slicing, we assume the initial state to be also terminating.

Assuming a feasible and terminating state allows us to weaken the weakest precondition of the program and remove assumptions from the program. However, we want to limit this to only those states that we are not interested in, specifically all the variables that the criterion does not share a transitive data dependency or anti-dependency with. We can achieve this by assuming a feasible and terminating state and then erasing the knowledge of this state for the dependent variables of the criterion. This will limit the flexibility given to the program slicer and require it to preserve all possible values of the variables in the criterion. Let $\mathcal{P}$ be a program, $\mathcal{P}'$ a possible program slice, v the criterion and v' the set of all data dependencies computed by $\widehat{\gamma}_+$ for $\mathcal{P}$ with the criterion v, we have the condition

$$(\mathrm{FV}(R) \subseteq v) \Rightarrow [\neg\, \mathrm{wp}(\mathcal{P}, \bot) \wedge \mathrm{wp}(\mathcal{P}, \top) \Rightarrow v' : [\mathrm{wp}(\mathcal{P}, R) \Rightarrow \mathrm{wp}(\mathcal{P}', R)]]$$

where $\neg\, \mathrm{wp}(\mathcal{P}, \bot)$ is the condition for a feasible state and $\mathrm{wp}(\mathcal{P}, \top)$ is the weakest precondition for a terminating program, and $v' : [R]$ quantifies only over the variables contained in v'. For example, $\{\texttt{total}\} : R$ is equivalent to the predicate $\forall \mathit{total} \in \mathrm{typeof}(\mathit{total}) \cdot R$ where all other variables are free in R. Along with the wlp condition, we call this a *partial* Egli-Milner ordering and this is the semantic property of partial program slicing.

4.4.2 Blind Alleys

A feasible state for a non-deterministic program that contains partial commands does not preclude the absence of miracles. A feasible program only ensures that there exists at least one initial state and one path in the program that will not produce a miracle. A program can choose to execute any choice of a non-deterministic operator without any consideration for the initial state. If the program makes a choice that leads to a miracle, although the initial state was feasible is called a *blind alley* [38].

The program in Figure 4.14 non-deterministically chooses to either assume that `sum` has the value of `z` and then assign the value `0` to `total` or it will choose to assign the value `7` to `z` and the value of `x` times `y` to `total`. Computing the feasibility condition for this program tells us that every state is a feasible state for the program. The program slice for the

criterion {`total`} in Figure 4.15 is the program slice we want, however it violates our semantic property of partial program slicing.

```
1  assume sum == z;
2  total := 0
3 ||
4  z := 7;
5  total := x * y
```

Figure 4.14: A program that is feasible for all states. However, executing this program in a state where `sum` is not equal to `z` will produce a miracle when the program chooses to execute the assumption command that `sum` is equal to `z`.

```
1  skip
2  total := 0
3 ||
4  skip
5  total := x * y
```

Figure 4.15: The program slice for the criterion {`total`}.

Every initial state for every variable is feasible for the program in Figure 4.14. Activating the original program in a feasible state where `sum` is not equal to the value of `z` will produce a miracle if the program chooses the branch containing the assumption that `sum` must be equal to `z`. Removing the assumption that `sum` is equal to `z` feels like the correct program slice, but it does not preserve the final values of `total`. This can be observed when we attempt to prove that the program slice satisfies the wp part of the semantic property of partial program slicing. Computing the wp for the program in Figure 4.14 and the projected predicate R such that $\mathrm{FV}(R) \subseteq \{\texttt{total}\}$ will give us the weakest precondition

$$(sum = z \Rightarrow R[total := 0]) \wedge R[total := x * y][z := 7]$$

and computing wp for the program slice with the same predicate R will give us the weakest precondition

$$R[total := 0] \wedge R[total := x * y]$$

To show that these conditions are equal we need to show

$$(sum = z \Rightarrow R[total := 0][z := 7]) \Rightarrow R[total := 0]$$

holds for all states. However, in order to show that this implication holds we need to assume that the initial value of `sum` is equal to the initial value of `z`. However, as this program is feasible for all initial states of `sum` and `z`, we cannot use this knowledge to prove this implication. The program slice does not satisfy preserve the final values of `total`. However, it is the program slice that we want for this program and criterion {`total`}. Mixing non-determinism and partial commands gives us program slices that violate our semantic property, but achieves what we want from a program slicer that handles partial commands. To fix this we need to either remove the non-deterministic part or the partial command part of slicing partial commands. We focus on removing the non-determinism by limiting ourselves to only feasible program paths.

A program path is simply a program that does not contain any non-deterministic choices. In our language a program path can only be constructed from assertions, assumptions, assignment, and sequential composition. A program $\mathcal{P}$ can contain many program paths. Our example program Figure 4.14) contains exactly two paths, the path that executes the bottom branch and the path that executes the top branch. We will use the notation $\rho \in \mathcal{P}$ to say that a program ρ is a program path of a program $\mathcal{P}$. Program paths have the property that for every path ρ in a program $\mathcal{P}$ the weakest precondition of $\mathcal{P}$ implies the weakest precondition of ρ for a postcondition R and the conjunction of the weakest preconditions of all the paths in a program imply the weakest precondition of the program for all post-conditions R.

Focusing only on the feasible program paths of a program allows us to remove assumptions in the presence of non-determinism as we have removed non-determinism from the semantic property of partial program slicing. This weakens the semantic property, but it is necessary as it will allow us to show that our example program is a valid program slice. Letting $\rho \in \mathcal{P}$ mean a program path of $\mathcal{P}$, our semantic property then becomes

$$\begin{aligned} & \forall \rho \in \mathcal{P} \cdot \mathrm{FV}(R) \subseteq v \\ \Rightarrow & \\ & [\neg\, \mathrm{wp}(\rho, \bot) \wedge \mathrm{wp}(\rho, \top) \Rightarrow v' : [\mathrm{wp}(\rho, R) \Rightarrow \mathrm{wp}(\rho', R)]] \end{aligned}$$

where ρ' is the program slice of ρ. Using this semantic property to show that our program slice (Figure 4.15) will now succeed. For the path that

contains the top branch of the program we will have the condition

$$\begin{array}{l} \quad sum = z \\ \Rightarrow \\ \quad \{total, x, y\} : [(sum = z \Rightarrow R[total := 0][z := 7] \Rightarrow R[total := 0])] \end{array}$$

which is now trivially satisfied as we can now assume that we execute the program in a state where `sum` is equal to `z`. This also works for the havoc command. If we treat the havoc command `havoc x` as a non-deterministic choice between the assignment of all the possible values of the variable being havoc-ed, such as $\forall y \in \text{typeOf}(x) \cdot || \ \texttt{x} := \texttt{y}$, then we can also reason about the various paths through a havoc. However, it is of little value to worry about this and we will simply keep the command around when we slice the paths of a program given by $||$.

4.4.3 Non-Killing Dependency Analysis

The proof that our program slicer $\widehat{\Delta}$ produces only correct program slices for a given program and criterion contains many moving parts. To limit these moving parts we redefine our primitive data dependency analysis $\widehat{\gamma}$ for the assignment command. This will give us monotonicity over sequential composition. When $\widehat{\gamma}$ inspects an assignment command it will remove the target variable of the assignment from the criterion if the criterion contains the variable as the analysis is no longer interested in the variable. By removing the variable from the criterion, we are unable to share a set of variables when we reason about sequential composition in the proof.

A program in SSA form ensures that every variable incarnation is assigned to only once. The analysis $\widehat{\gamma}$ removes the variable from the criterion as it is no longer interested in the value of the variable. In SSA form, every variable is assigned to only once, so there is no longer any reason to remove the variable from the criterion. Keeping the variable we are given the mathematical property that $\widehat{\gamma}$ is monotonic over sequential composition. We redefine $\widehat{\gamma}$ such that it no longer removes the target of the assignment from the criterion. We call this the non-killing definition of $\widehat{\gamma}$.

$$\widehat{\gamma}(\texttt{x} := \texttt{E}, v) = \begin{cases} \text{FV}(E) \cup v & \text{if } x \in v \\ v & \text{otherwise} \end{cases}$$

This redefinition will also affect our data and anti-dependency function $\widehat{\gamma}_+$. Without removing any variables from the criterion we now have monotonicity of γ over sequential composition.

Lemma 2. *Suppose* $S0; S1 \in \mathbf{Prgs}$ *and* $v \in \mathbf{Vars}$. *Then,*

$$\widehat{\gamma}(S1, v) \subseteq \widehat{\gamma}(S0; S1, v)$$

Proof. Trivial. □

This lemma also applies to $\widehat{\gamma}_+$ as it uses $\widehat{\gamma}$ for the data dependency analysis and the anti-dependency analysis is already monotonic over sequential composition. Computing the fixpoint of $\widehat{\gamma}_+^\infty$ for a program in SSA form with our redefined $\widehat{\gamma}$ analysis will give us a set of data dependency variables such that we no longer need to compute the intermediate variables of a sequential composition. We can simply remove $\widehat{\gamma}_+$ from the analysis when we reason about sequential composition. We redefine $\widehat{\Delta}$ for sequential composition as slicing $S0$ with the fixpoint of $\widehat{\gamma}_+^\infty$ and composing the results with the program slice for $S1$ with the same fixpoint.

Lemma 3. *Suppose* $S0; S1 \in \mathbf{Prgs}$, $v \in \mathbf{Vars}$, *and* $v' \in \mathbf{Vars}$. *Then,*

$$\begin{aligned} & \widehat{\gamma}_+^\infty(S0; S1, v, \varnothing) \subseteq v, v' \\ \Rightarrow & \\ & \widehat{\Delta}(S0; S1, v) = \widehat{\Delta}(S0, v); \widehat{\Delta}(S1, v) \end{aligned}$$

Proof. Trivial. □

Removing $\widehat{\gamma}$ from the definition of $\widehat{\Delta}$ for sequential composition will help us prove the correctness of our program slices. It is important to note that using this non-killing version of $\widehat{\gamma}$ will not have an effect on the size of the resulting program slice. As long as the program is in SSA form, the use of the killing version of γ and the non-killing version will result in the same program slices. From this point forward, we will only make use of the non-killing definition of $\widehat{\gamma}$.

Understanding how $\widehat{\gamma}$ works is useful for understanding how it bounds the variables of the predicate transformers. Similar to Lemma 1 for γ we show an analogous result for $\widehat{\gamma}$.

Lemma 4. *Suppose* $\mathcal{P} \in \mathbf{Prgs}$, $R \in \mathbf{Preds}$, $v \in \mathbf{Vars}$, *and* $v' \in \mathbf{Vars}$. *Then,*

$$\begin{aligned} & \mathrm{FV}(R) \subseteq v \wedge \widehat{\gamma}_+^{\infty}(\mathcal{P}, v, v') \subseteq v, v' \\ \Rightarrow & \\ & \mathrm{FV}(\mathrm{wp}(\widehat{\Delta}(\mathcal{P}, v))) \subseteq \mathrm{prj}_1(\widehat{\gamma}_+^{\infty}(\mathcal{P}, v, v')) \end{aligned}$$

Proof. Proof is analogous to Lemma 1 except for sequential composition. *Case* (S0;S1). Assume $\mathrm{FV}(R) \subseteq v$ and assume $\widehat{\gamma}_+^{\infty}(S0; S1, v, v') \subseteq v, v$. Assume the property holds for $S0$ and $S1$. As $\widehat{\gamma}_+^{\infty}$ is increasing we can assume $\widehat{\gamma}_+^{\infty}(S1, v, v') \subseteq \widehat{\gamma}_+^{\infty}(S0; S1, v, v')$. By transitivity of $\subseteq$ we can assume $\widehat{\gamma}_+^{\infty}(S1, v, v') \subseteq v, v'$. From our induction hypothesis on $S1$ we can conclude $\mathrm{FV}(\mathrm{wp}(\widehat{\Delta}(S1, v), R)) \subseteq v$. Instantiating our induction hypothesis on $S0$ with $\mathrm{wp}(\widehat{\Delta}(S1, v), R)$ we can conclude $\mathrm{FV}(\mathrm{wp}(\widehat{\Delta}(S1, v), R)) \subseteq v \wedge \widehat{\gamma}_+^{\infty}(S0, v, v') \subseteq v, v'$ implies $\mathrm{FV}(\mathrm{wp}(\widehat{\Delta}(S0, v), \mathrm{wp}(\widehat{\Delta}(S1, v), R))) \subseteq v$. It follows from our assumptions and that $\widehat{\gamma}_+^{\infty}$ is a fixpoint we can conclude $\mathrm{FV}(\mathrm{wp}(\widehat{\Delta}(S0, v), \mathrm{wp}(\widehat{\Delta}(S1, v), R))) \subseteq v$. By definition of wp and $\widehat{\Delta}$ we can conclude $\mathrm{FV}(\mathrm{wp}(\widehat{\Delta}(S0; S1, v), R)) \subseteq v$ holds.

□

4.4.4 Useful Properties of wp

There are a couple of useful properties of the wp that will help us prove that our program slicer only produces semantically valid program slices. The first useful property states that if we execute a deterministic program $S0; S1$ in a feasible state, then either the program $S0$ will not terminate or the state after executing $S0$ will be a feasible initial state for $S1$.

Lemma 5. *Suppose* $S0; S1 \in \mathbf{Prgs}$ *and* $S0; S1$ *is deterministic. Then,*

$$[\neg\, \mathrm{wp}(S0; S1, \bot) \Rightarrow \neg\, \mathrm{wp}(S0, \top) \vee \mathrm{wp}(S0, \neg\, \mathrm{wp}(S1, \bot))]$$

Proof. Suppose $\neg\, \mathrm{wp}(S0; S1, \bot)$ holds in some state σ. By definition of wp we can assume $\neg\, \mathrm{wp}(S0, \mathrm{wp}(S1, \bot)).\sigma$. Suppose $\mathrm{wp}(S0, \top).\sigma$ holds. Since wp is a function we can assume $\mathrm{wp}(S0, \mathrm{wp}(S1, \bot) \vee \neg\, \mathrm{wp}(S1, \bot)).\sigma$ also holds. Since $S0$ is deterministic it is also universally disjunctive. Since $S0$ is universally disjunctive we can assume $\mathrm{wp}(S0, \mathrm{wp}(S1, \bot)).\sigma$

or $\mathrm{wp}(S0, \neg\,\mathrm{wp}(S1, \bot)).\sigma$ holds. From $\neg\,\mathrm{wp}(S0, \mathrm{wp}(S1, \bot)).\sigma$ we can conclude $\mathrm{wp}(S0, \neg\,\mathrm{wp}(S1, \bot)).\sigma$. Since σ was arbitrary we can conclude $\neg\,\mathrm{wp}(S0; S1, \bot) \Rightarrow \neg\,\mathrm{wp}(S0, \top) \vee \mathrm{wp}(S0, \neg\,\mathrm{wp}(S1, \bot))$ holds for all states. □

The next property states that wp is *sub-monotonic*. That is, if we know the postcondition $P \Rightarrow Q$ holds for all terminating states of a program, then we know that for all initial states of the program that satisfy the postcondition P, they must also satisfy the postcondition Q. In the presence of partial commands we must also assume that the program is both feasible and terminating. This property trivially holds for total commands.

Lemma 6 (Sub-monotonicity of wp)**.** *Suppose* $\mathcal{P} \in$ **Prgs**, $R \in$ **Preds**, *and* $P \in$ **Preds**. *Then,*

$$[\neg\,\mathrm{wp}(\mathcal{P}, \bot) \wedge \mathrm{wp}(\mathcal{P}, \top) \wedge \mathrm{wp}(\mathcal{P}, [P \Rightarrow R]) \Rightarrow [\mathrm{wp}(\mathcal{P}, P) \Rightarrow \mathrm{wp}(\mathcal{P}, R)]]$$

Proof. By structural induction on $\mathcal{P}$.

Case ($\mathtt{x} := \mathtt{E}$). Trivial.

Case (`assert Q`). Suppose $\mathrm{wp}(\mathtt{assert\ Q}, [P \Rightarrow R])$ holds in some state σ. From the definition of wp for assertions we can assume $Q.\sigma \wedge [P \Rightarrow R].\sigma$ holds. Suppose $\mathrm{wp}(\mathtt{assert\ Q}, P).\sigma.\sigma'$ holds for some state σ'. From the definition of wp for assertions we can conclude $Q.\sigma.\sigma' \wedge P.\sigma.\sigma'$ holds. From $[P \Rightarrow R].\sigma$ we can conclude $P.\sigma.\sigma' \Rightarrow R.\sigma.\sigma'$ holds. From $P.\sigma.\sigma'$ we can conclude $R.\sigma.\sigma'$. From $Q.\sigma.\sigma'$ we can conclude $Q.\sigma.\sigma' \wedge R.\sigma.\sigma'$. As σ' was arbitrary we can conclude $[\mathrm{wp}(\mathtt{assert\ Q}, P) \Rightarrow \mathrm{wp}(\mathtt{assert\ Q}, R)].\sigma$. As σ was arbitrary we can conclude that the condition $\mathrm{wp}(\mathtt{assert\ Q}, [P \Rightarrow R]) \Rightarrow [\mathrm{wp}(\mathtt{assert\ Q}, P) \Rightarrow \mathrm{wp}(\mathtt{assert\ Q}, R)$ holds for all states.

Case (`assume Q`). Suppose $\neg\,\mathrm{wp}(\mathtt{assume\ Q}, \bot)$ holds in some state σ. By definition of wp we can conclude $Q.\sigma$. Suppose $\mathrm{wp}(\mathtt{assume\ Q}, [P \Rightarrow R]).\sigma$ holds. From the definition of wp we can conclude $Q.\sigma \Rightarrow [P \Rightarrow R].\sigma$ holds. Suppose $\mathrm{wp}(\mathtt{assume\ Q}, P).\sigma.\sigma'$ holds for some state $\sigma.\sigma'$. From the definition of wp we can assume $Q\sigma.\sigma' \Rightarrow P\sigma.\sigma'$ holds. Suppose $Q.\sigma.\sigma'$. We can conclude $P.\sigma.\sigma'$ holds. From $Q.\sigma$ we can conclude $[P \Rightarrow R].\sigma$. From $P.\sigma.\sigma'$ we can conclude $R.\sigma.\sigma'$. Since σ' was arbitrary we can conclude

$[\mathrm{wp}(\texttt{assume Q}, P) \Rightarrow \mathrm{wp}(\texttt{assume Q}, R)]$. Since σ was arbitrary we can conclude $\mathrm{wp}(\texttt{assume Q}, [P \Rightarrow R]) \Rightarrow [\mathrm{wp}(\texttt{assume Q}, P) \Rightarrow \mathrm{wp}(\texttt{assume Q}, R)$ holds for all states.

Case $(S0; S1)$. Suppose $\neg\, \mathrm{wp}(S0; S1, \bot).\sigma$ holds and suppose the condition $\mathrm{wp}(S0; S1, \top).\sigma$ holds and suppose $\mathrm{wp}(S0; S1, [P \Rightarrow R]).\sigma$ holds. By definition and monotonicity of wp we may conclude $\mathrm{wp}(S0, \top).\sigma$ also holds. From Lemma 6 and $\neg\, \mathrm{wp}(S0; S1, \bot).\sigma$ we may conclude $\mathrm{wp}(S0, \neg\, \mathrm{wp}(S1, \bot)).\sigma$ holds. Instantiating our induction hypothesis on $S1$ we may assume the condition $\neg\, \mathrm{wp}(S1, \bot) \wedge \mathrm{wp}(S1, \top) \wedge \mathrm{wp}(S1, [P \Rightarrow R]) \Rightarrow [\mathrm{wp}(S1, P) \Rightarrow \mathrm{wp}(S1, R)]$ holds for σ. By monotonicity of wp we may assume $\mathrm{wp}(S0, \neg\, \mathrm{wp}(S1, \bot) \wedge \mathrm{wp}(S1, \top) \wedge \mathrm{wp}(S1, [P \Rightarrow R])) \Rightarrow \mathrm{wp}(S0, [\mathrm{wp}(S1, P) \Rightarrow \mathrm{wp}(S1, R)])$ also holds for σ. By conjunctivity of wp we can assume the condition $\mathrm{wp}(S0, \neg\, \mathrm{wp}(S1, \bot)) \wedge \mathrm{wp}(S0, \mathrm{wp}(S1, \top)) \wedge \mathrm{wp}(S0, \mathrm{wp}(S1, [P \Rightarrow R])) \Rightarrow \mathrm{wp}(S0, [\mathrm{wp}(S1, P) \Rightarrow \mathrm{wp}(S1, R)])$ holds for σ. By our assumptions $\mathrm{wp}(S0, \neg\, \mathrm{wp}(S1, \bot)).\sigma$, $\mathrm{wp}(S0, \mathrm{wp}(S1, \top)).\sigma$, and $\mathrm{wp}(S0, \mathrm{wp}(S1, [P \Rightarrow R])).\sigma$ we can conclude from our implication that $\mathrm{wp}(S0, [\mathrm{wp}(S1, P) \Rightarrow \mathrm{wp}(S1, R)]).\sigma$ holds. Instantiating the induction hypothesis on $S0$ with $\mathrm{wp}(S1, P)$ and $\mathrm{wp}(S1, R)$ we can assume $\neg\, \mathrm{wp}(S0, \bot).\sigma \wedge \mathrm{wp}(S0, \top).\sigma$ and $\mathrm{wp}(S0, [\mathrm{wp}(S1, P) \Rightarrow \mathrm{wp}(S1, R)]).\sigma \Rightarrow [\mathrm{wp}(S0; S1, Q) \Rightarrow \mathrm{wp}(S0; S1, R)].\sigma$. From feasibility of $S0; S1$, monotonicity of wp, and contraposition we can assume $\neg\, \mathrm{wp}(S0, \bot).\sigma$. From this, $\mathrm{wp}(S0, \top).\sigma$, and $\mathrm{wp}(S0, [\mathrm{wp}(S1, P) \Rightarrow \mathrm{wp}(S1, R)]).\sigma$ we can conclude $[\mathrm{wp}(S0; S1, P) \Rightarrow \mathrm{wp}(S0; S1, R)].\sigma$.

□

This lemma also holds when we bound the quantifier to only those variables that we are interested in.

Lemma 7 (Bounded sub-monotonicity of wp)**.** *Suppose* $\mathcal{P} \in \mathbf{Prgs}$*,* $R \in \mathbf{Preds}$*,* $P \in \mathbf{Preds}$*,* $v \in \mathbf{Vars}$*, and* $v' \in \mathbf{Vars}$*. Then,*

$$\mathrm{FV}(R) \subseteq v \wedge \widehat{\gamma}_{+}^{\infty}(\rho, v, v') \subseteq (v, v')$$
$$\Rightarrow$$
$$[\neg\, \mathrm{wp}(\mathcal{P}, \bot) \wedge \mathrm{wp}(\mathcal{P}, \top) \wedge \mathrm{wp}(\mathcal{P}, v\!:\![P \Rightarrow R]) \Rightarrow v\!:\![\mathrm{wp}(\mathcal{P}, P) \Rightarrow \mathrm{wp}(\mathcal{P}, R)]]$$

Proof. Analogous to the proof in Lemma 6. □

We are now ready for the proof that given a valid program $\mathcal{P}$ and a fixpoint from $\widehat{\gamma}_+^\infty$ for the criterion v, $\widehat{\Delta}$ only produces program slices that satisfy the semantic property of partial program slicing.

4.4.5 Proof of Correctness

We are now ready to prove that for an arbitrary program $\mathcal{P}$ and an arbitrary set of variables v such that $\widehat{\gamma}_+^\infty(\mathcal{P}, v, \varnothing) \subseteq v, v'$, $\widehat{\Delta}$ will produce a valid program slice for every path in $\mathcal{P}$. The proof proceeds by structural induction over the possible commands in $\mathcal{P}$, however we will only treat the partial command `assume Q` and sequential composition as these are the only commands affected by the presence of partial commands. We prove the two conditions of the semantic property separately.

Theorem 4. *Suppose* $\mathcal{P} \in \mathbf{Prgs}$, $v \in \mathbf{Vars}$, $v' \in \mathbf{Vars}$, *and* $R \in \mathbf{Preds}$. *Then, for every path* $\rho \in \mathcal{P}$,

$$\begin{aligned} &\mathrm{FV}(R) \subseteq v \wedge \widehat{\gamma}_+^\infty(\rho, v, v') \subseteq (v, v') \\ \Rightarrow\ &[\neg\,\mathrm{wp}(\rho, \bot) \wedge \mathrm{wp}(\rho, \top) \Rightarrow v : [\mathrm{wp}(\rho, R) \Rightarrow \mathrm{wp}(\widehat{\Delta}(\rho, v), R)]] \end{aligned}$$

Proof. We proceed by structural induction on ρ. The only commands that are not analogous to Theorem 3 are assume commands and sequential composition.

Case (`assume Q`,$\mathrm{FV}(Q) \cap v = \varnothing$). Suppose $\neg\,\mathrm{wp}(\texttt{assume Q}, \bot).\sigma$ holds. By definition of wp we can assume $Q.\sigma$ holds. Suppose the condition $\mathrm{wp}(\texttt{assume Q}, R).\sigma.\sigma'$ holds for some σ' where $\mathrm{dom}(\sigma') = v$. By definition of wp and we can conclude $Q.\sigma.\sigma' \Rightarrow R.\sigma.\sigma'$ holds. Since $\mathrm{FV}(Q) \cap v = \varnothing$ we can conclude $Q.\sigma \Rightarrow R.\sigma.\sigma'$. From $Q.\sigma$ we can conclude $R.\sigma.\sigma'$. From $\mathrm{FV}(Q) \cap v = \varnothing$ and definition of $\widehat{\Delta}$ we can conclude $\mathrm{wp}(\widehat{\Delta}(\texttt{assume Q}, v), R).\sigma.\sigma'$. Since σ' was arbitrary we can conclude $v : [\mathrm{wp}(\texttt{assume Q}, R) \Rightarrow \mathrm{wp}(\widehat{\Delta}(\texttt{assume Q}, v), R)].\sigma$. Since σ was arbitrary we can conclude that the condition $\neg\,\mathrm{wp}(\texttt{assume Q}, \bot) \wedge \neg\,\mathrm{wp}(\texttt{assume Q}, \bot) \Rightarrow v : [\mathrm{wp}(\texttt{assume Q}, R) \Rightarrow \mathrm{wp}(\widehat{\Delta}(\texttt{assume Q}, v), R)]$ holds for all states.

Case (`assume Q`,$\mathrm{FV}(Q) \cap v \neq \varnothing$). Trivial.

Case $(S0; S1)$. Suppose $\mathrm{FV}(R) \subseteq v$. Suppose $\widehat{\gamma}_+(S0; S1, v, v') \subseteq v, v'$. Assume the induction hypothesis for $S0$ and $S1$. Suppose the condition $\neg\, \mathrm{wp}(S0; S1, \bot).\sigma$ and $\mathrm{wp}(S0; S1, \top).\sigma$ hold for an arbitrary state σ. We must show $v : [\mathrm{wp}(S0; S1, R) \Rightarrow \mathrm{wp}(\widehat{\Delta}(S0; S1, v), R)].\sigma$ holds. From Lemma 2 and transitivity of $\subseteq$ we can assume $\widehat{\gamma}_+(S1, v, v') \subseteq v, v'$. Instantiating the induction hypothesis on $S1$ we can conclude $\neg\, \mathrm{wp}(S1, \bot) \wedge \mathrm{wp}(S1, \top) \Rightarrow v : [\mathrm{wp}(S1, R) \Rightarrow \mathrm{wp}(\widehat{\Delta}(S1, v), R)]$ holds for σ. By monotonicity of wp we can also assume $\mathrm{wp}(S0, \neg\, \mathrm{wp}(S1, \bot) \wedge \mathrm{wp}(S1, \top)) \Rightarrow \mathrm{wp}(S0, v : [\mathrm{wp}(S1, R) \Rightarrow \mathrm{wp}(\widehat{\Delta}(S1, v), R)])$ holds for σ. By conjunctivity of wp we can assume $\mathrm{wp}(S0, \neg\, \mathrm{wp}(S1, \bot)).\sigma \wedge \mathrm{wp}(S0, \mathrm{wp}(S1, \top)).\sigma \Rightarrow \mathrm{wp}(S0, v : [\mathrm{wp}(S1, R) \Rightarrow \mathrm{wp}(\widehat{\Delta}(S1, v), R)]).\sigma$. From termination of $S0; S1$ and monotonicity of wp we can assume $\mathrm{wp}(S0, \top).\sigma$. As our path is deterministic we can apply Lemma 6 and assume $\neg\, \mathrm{wp}(S0; S1, \bot).\sigma \wedge \mathrm{wp}(S0, \top).\sigma \Rightarrow \mathrm{wp}(S0, \neg\, \mathrm{wp}(S1, \bot)).\sigma$. By our assumption that $S0; S1$ is feasible and $S0$ terminates we can conclude $\mathrm{wp}(S0, \neg\, \mathrm{wp}(S1, \bot)).\sigma$. This and termination of $S0; S1$ allows us to conclude from $\mathrm{wp}(S0, \neg\, \mathrm{wp}(S1, \bot)).\sigma \wedge \mathrm{wp}(S0, \mathrm{wp}(S1, \top)).\sigma \Rightarrow \mathrm{wp}(S0, v : [\mathrm{wp}(S1, R) \Rightarrow \mathrm{wp}(\widehat{\Delta}(S1, v), R)]).\sigma$ the condition $\mathrm{wp}(S0, v : [\mathrm{wp}(S1, R) \Rightarrow \mathrm{wp}(\widehat{\Delta}(S1, v), R)]).\sigma$. Application of Lemma 7 with $S0$, $\mathrm{wp}(S1, R)$, and $\mathrm{wp}(\widehat{\Delta}(S1, v), R)$ allows us to assume $\neg\, \mathrm{wp}(S0, \bot).\sigma \wedge \mathrm{wp}(S0, \top).\sigma \wedge \mathrm{wp}(S0, v : [\mathrm{wp}(S1, R) \Rightarrow \mathrm{wp}(\widehat{\Delta}(S1, v), R)])$ holds for σ. From this condition and our assumption we are able to conclude $v : [\mathrm{wp}(S0, \mathrm{wp}(S1, R)) \Rightarrow \mathrm{wp}(S0, \mathrm{wp}(\widehat{\Delta}(S1, v), R))].\sigma$. As we have already assumed each antecedent we can assume $v : [\mathrm{wp}(S0, \mathrm{wp}(S1, R)) \Rightarrow \mathrm{wp}(S0, \mathrm{wp}(\widehat{\Delta}(S1, v), R))].\sigma$. Instantiating the induction hypothesis on $S0$ with $\mathrm{wp}(\widehat{\Delta}(S1, v), R)$ we can assume $\mathrm{FV}(\mathrm{wp}(\widehat{\Delta}(S1, v), R)) \subseteq v \wedge \widehat{\gamma}_+^{\infty}(S0, v, v') \subseteq v, v' \wedge \neg\, \mathrm{wp}(S0, \bot).\sigma \wedge \mathrm{wp}(S0, \top).\sigma$ implies that the condition $v : [\mathrm{wp}(S0, \mathrm{wp}(\widehat{\Delta}(S1, v), R)) \Rightarrow \mathrm{wp}(\widehat{\Delta}(S0, v), \mathrm{wp}(\widehat{\Delta}(S1, v), R))]$ also holds for σ. From Lemma 4 we can assume $\mathrm{FV}(\mathrm{wp}(\widehat{\Delta}(S1, v), R)) \subseteq v$ and from Lemma 2 with definition of $\widehat{\gamma}_+^{\infty}$ we can assume $\widehat{\gamma}_+^{\infty}(S0, v, v') \subseteq v, v'$. We know $S0$ is feasible and terminates, so we can assume $v : [\mathrm{wp}(S0, \mathrm{wp}(\widehat{\Delta}(S1, v), R)) \Rightarrow \mathrm{wp}(\widehat{\Delta}(S0, v), \mathrm{wp}(\widehat{\Delta}(S1, v), R))].\sigma$. From this condition and transitivity of $\Rightarrow$ we can conclude $v : [\mathrm{wp}(S0, \mathrm{wp}(S1, R)) \Rightarrow \mathrm{wp}(\widehat{\Delta}(S0, v), \mathrm{wp}(\widehat{\Delta}(S1, v), R))].\sigma$. By definition of wp we can conclude $v : [\mathrm{wp}(S0; S1, R) \Rightarrow \mathrm{wp}(\widehat{\Delta}(S0, v); \widehat{\Delta}(S1, v), R)].\sigma$. From Lemma 3 and the fact that v is a fixpoint of $\widehat{\gamma}_+^{\infty}$ we can conclude our intended goal

$v : [\mathrm{wp}(S0; S1, R) \Rightarrow \mathrm{wp}(\widehat{\Delta}(S0; S1, v), R)].\sigma.$

□

Now that we have shown that the program slicer $\widehat{\Delta}$ only produces programs that satisfy the $\Rightarrow$ part of the semantic property, we can now show that it also satisfies the $\Leftarrow$ part of the semantic property.

Theorem 5. *Suppose* $\mathcal{P} \in \mathbf{Prgs}$*,* $v \in \mathbf{Vars}$*,* $v' \in \mathbf{Vars}$*, and* $R \in \mathbf{Preds}$*. Then, for every* $\rho \in \mathcal{P}$*,*

$$\begin{array}{l} \quad \mathrm{FV}(R) \subseteq v \wedge \widehat{\gamma}_+(\rho, v, v') \subseteq (v, v') \\ \Rightarrow \\ \quad [\mathrm{wlp}(\widehat{\Delta}(\rho, v), R) \Rightarrow \mathrm{wlp}(\rho, R)] \end{array}$$

Proof. We proceed by structural induction on ρ. The only commands that are not analogous to Theorem 3 are assume commands and sequential composition.

Case (`assume Q`, $\mathrm{FV}(Q) \cap v = \varnothing$). Suppose $\mathrm{wlp}(\widehat{\Delta}(\texttt{assume Q}, v), R).\sigma$ holds. By definition of $\widehat{\Delta}$ and wlp we can conclude $R.\sigma$ holds. From $R.\sigma$ we can conclude $Q.\sigma \Rightarrow R.\sigma$. By definition of wlp we can assume $\mathrm{wlp}(\texttt{assume Q}, R).\sigma$. As σ was arbitrary we can conclude that the condition $\mathrm{wlp}(\widehat{\Delta}(\texttt{assume Q}, v), R) \Rightarrow \mathrm{wlp}(\texttt{assume Q}, R)$ holds for all states.

Case (`assume Q`,$\mathrm{FV}(Q) \cap v \neq \varnothing$). Trivial.

Case $(S0; S1)$. Suppose $\mathrm{FV}(R) \subseteq v$ and $\widehat{\gamma}_+(S0; S1, v, v') \subseteq (v, v')$. Assume the induction hypothesis for $S0$ and $S1$. Suppose the condition $\mathrm{wlp}(\widehat{\Delta}(S0; S1, v), R)$ holds for an arbitrary state σ. By definition of $\widehat{\Delta}$ and wlp we can assume $\mathrm{wlp}(\widehat{\Delta}(S0, v), \mathrm{wlp}(\widehat{\Delta}(S1, v), R)).\sigma$. By Lemma 2 we know $\widehat{\gamma}_+(S1, v, v')$ is a subset of $\widehat{\gamma}_+(S0; S1, v, v')$. From the assumption $\widehat{\gamma}_+(S0; S1, v, v') \subseteq (v, v')$ and transitivity of $\subseteq$ we may assume $\widehat{\gamma}_+(S1, v, v') \subseteq (v, v')$. Instantiating the induction hypothesis on $S1$ we may assume $\mathrm{wlp}(\widehat{\Delta}(S1, v), R) \Rightarrow \mathrm{wlp}(S1, R)$ holds for σ. From Lemma 4 and our assumptions we may assume $\mathrm{FV}(\mathrm{wp}(\widehat{\Delta}(S1, v), R)) \subseteq v$. Instantiating our induction hypothesis on $S0$ with $\mathrm{wlp}(\widehat{\Delta}(S1, v), R)$ allows us to assume $\mathrm{wlp}(\widehat{\Delta}(S0, v), \mathrm{wlp}(\Delta(S1, v), R)).\sigma \Rightarrow \mathrm{wlp}(S0, \mathrm{wlp}(\widehat{\Delta}(S1, v), R)).\sigma$. From this condition, monotonicity and definition of wlp we can conclude $\mathrm{wlp}(\widehat{\Delta}(S0; S1, v), R).\sigma \Rightarrow \mathrm{wlp}(S0; S1, R).\sigma$. As σ was arbitrary,

$\text{wlp}(\widehat{\Delta}(S0; S1, v), R) \Rightarrow \text{wlp}(S0; S1, R)$ holds for all states, which was our intended goal.

□

And this concludes the proof that $\widehat{\Delta}$ only produces valid partial program slices for feasible program paths.

4.5 Slicing Background Theory

We have constructed a program slicer $\widehat{\Delta}$ for the imperative part of our programming language and have proven that the slicer will produce only semantically correct program slices. However, our programming language contains more than this imperative part, it also contains a declarative part where type declarations, constant and function declarations, and axioms on these constants and functions are declared. A programmer understanding the behavior of an imperative program may be interested in understanding the background part that defines the background theories of the imperative program. In this section we show how we can safely remove type declarations, constant and function declarations, and axioms that do not play a role in the behavior of the program slice in the imperative part.

4.5.1 Global Variables and Types

Global variable declarations define the state space of the imperative program. Every variable declaration must be a type that is defined by a type declaration. Computing the fixpoint of $\widehat{\gamma}_+^\infty$ for the imperative part of a program gives us the set of all variables that play a role in the behavior of the criterion the programmer interested in. We can remove all variable declarations that are not contained within this set as their state has no effect on the sliced program. Moreover, we can remove all type declarations that are not used by the global variables that are kept.

For example, the declarations in Figure 4.16 declares a type `uint` and a type `Fruit`. It also declares the global variables `total` and `sum` of type `uint` and the global variable `banana` of type `Fruit`. If we assume that this is the declaration for some example program and we assume computing $\widehat{\gamma}_+^\infty$ for our example program gives us the set {`total`} we can

remove the variables `sum` and `banana` as they are not contained within the set {`total`}. However, we keep the variable declaration on `total` as it is in the set. As we do not keep any variables of type `Fruit`, we can also remove this declaration and give the programmer the background part in Figure 4.17.

```
1 type uint
2 type Fruit
3
4 var total : uint
5 var sum : uint
6 var banana : Fruit
```

Figure 4.16: A background part that declares two types `uint` and `Fruit`. It also declares the global variables `total` and `sum` of type `uint` and `banana` of type `Fruit`.

```
1 type uint
2 type Fruit
3
4 var total : uint
5 var sum : uint
6 var banana : Fruit
```

Figure 4.17: Removing all variables that are not free in the program slice and all type declarations that are not used by the global variable declarations.

The set of free variables of a program slice are contained within the fixpoint of $\widehat{\gamma}_{+}^{\infty}$. Keeping all variable declarations for all the variables in the fixpoint of $\widehat{\gamma}_{+}^{\infty}$ ensures that we have a declaration for all the free variables in the program slice. Removing all the variable declarations that are not contained in the fixpoint of $\widehat{\gamma}_{+}^{\infty}$ is safe to do as none of these variables are used by the program slice. For our example we removed the variable declaration on `sum` and `banana` as we know from the fixpoint of $\widehat{\gamma}_{+}^{\infty}$ that neither `sum` nor `banana` is used by the program slice.

Once we have removed all the global variables declarations that are not needed by the program slice we can remove all the type declarations that are not used by those global variable declarations. This is done by simply iterating through the global variables that we keep in the background part and removing those type declarations that are not needed by the variables. For our working example, we can safely remove the type declaration of `Fruit` as we have no variable declarations that are of type `Fruit`. By keeping all variables required by the program and keeping all types required by the variables we keep all types that are used by the

program for global variables.

4.5.2 Constants and Functions

We have an analysis $\widehat{\gamma}_{+}^{\infty}$ that computes the set of all variable used in a program slice and we use this set to remove any global variable declarations that do not define a variable in this set. To know what constants and functions we can remove from the background part, we iterate through the program to compute the set of constants and functions used to define the behavior of the criterion, similar to $\widehat{\gamma}_{+}^{\infty}$ for variables. When we inspect an assignment we query the right hand side for any constants and functions used and add them to our set of relevant constants or functions. When we inspect an assumption or an assertion, we inspect the expression for all constants and functions and add them to the relevant set if the free variables of the expression are shared with the criterion. We do this iteration through the entire program until we have found all constants and functions that are relevant to the behavior of the criterion.

The program in Figure 4.18 declares the constants `x` and `y` of `uint` and the constant `f` of type `Fruit`. It also declares the functions `add` and `sub`, both of which take a `uint` and return a `uint`, and a function `isRipe` that takes a `Fruit` element and returns a Boolean. The program also contains an imperative part that first assumes that the value of the variable `sum` is equal to the value of the constant `y`. The program then assigns the value of `sum` to `total` and then assumes that the application of the function `add` to `x` and to `total` are equal. This imperative part does not use the constant `f` nor the functions `sub` and `isRipe`. We can remove them from the background declarations to get the program in Figure 4.19.

Computing the set of all constants and functions that are used in the imperative program gives us the set $\{\texttt{x}, \texttt{y}, \texttt{add}\}$. The analysis proceeds by inspecting the assumption on `total` and observing that the function `add` and the constant `x` are used to define the values of `total`. It adds these to the relevant constants and functions and proceeds. As the assignment to `total` is given by the value of `sum`, which is assumed to have the value of the constant `y`; the analysis also adds the constant `y` to the relevant constants and functions. As it was the last command in the imperative part, the analysis stops and gives us the set of constants $\{\texttt{x}, \texttt{y}\}$

```
1 const x : uint
2 const y : uint
3 const f : Fruit
4
5 func add (uint) returns (uint)
6 func sub (uint) returns (uint)
7 func isRipe (Fruit) returns (bool)
8
9 assume sum == y;
10 total := sum;
11 assume add(x) == add(total)
```

Figure 4.18: A background part declaration for the constants `x`,`y`, and `f` and the functions `sub`, `add`, and `isRipe`. The imperative part makes use of the constant `y` and `x` and the function `add`, but not the constant `f` nor the functions `sub` and `isRipe`.

```
1 const x : uint
2 const y : uint
3 const f : Fruit
4
5 func add (uint) returns (uint)
6 func sub (uint) returns (uint)
7 func isRipe (Fruit) returns (bool)
8
9 assume sum == y;
10 total := sum;
11 assume add(x) == add(total)
```

Figure 4.19: Removing the declarations for the constant `f` and the functions `sub` and `isRipe`.

and functions {`add`}.

Once we have computed the set of all relevant constants and functions we can remove those declarations from the background theory that declare a constant or function not contained in the relevant sets. As with $\widehat{\gamma}_{+}^{\infty}$, the free constants and functions of a program slice is contained within these relevant sets. Keeping all constants and functions in the relevant sets will ensure that every constant and function used by the program slice are declared. Removing the declarations on the constant `f` and the functions `sub` and `add` from the background part gives us the program in Figure 4.19.

Every constant and function is declared using a type. For all constants and functions that are kept in the background theory, similar to global variables, we must keep all type declarations that are used by those constants and functions in the background theory. For our example, we must keep the type declaration `uint` as the constants `x` and `y` are declared as being of type `uint`. The function `add` also only uses `uint` as its input and output types, so we must be sure to keep this declaration. However,

we can remove type declaration `Fruit` as no constants are of type `Fruit` and no functions use type `Fruit` as an input or an output type.

4.5.3 Axioms

Axioms declare properties on constants and functions in the background part of the program. The imperative part can use these axioms when reasoning about an expression that contains a constant or a function. To preserve the behavior of the variables in the criterion, we must preserve the behavior of the constants and functions that are relevant for the variables in the criterion. As with assumptions and variables, we first analyze the axioms for any dependencies between the constants and functions before we can begin removing axioms from the background part to ensure that we do not remove any behavior of a constant or a function from the background.

In Figure 4.20 the axiom declarations state that every element of type `uint` should have a value greater than or equal to the constant `0` and every element of type `Fruit` should be ripe. The declarations also state that for every element of `uint` the value of applying `add` to the value of applying `sub` must be equal and that for every element of `uint`, the application of `mod` must be greater than or equal to the element. Finally, there is a declaration stating that the constant `x` is equal to the application of `sub` to the constant `y`. If we assume that the imperative part of the program only uses the constant `x` and the function `add`, we can remove the some of the declarations and have only those declaration in Figure 4.21.

```
1 axiom forall i:uint :: 0 <= i
2 axiom forall f:Fruit :: isRipe(f)
3 axiom forall i:uint ::
4   sub(add(i)) == i
5 axiom forall i:uint :: i <= mod(i)
6 axiom x == sub(y)
```

Figure 4.20: A background declaration stating axioms on the constants, functions, and global variables.

```
1 axiom forall i:uint :: 0 <= i
2 axiom forall f:Fruit :: isRipe(f)
3 axiom forall i:uint ::
4   sub(add(i)) == i
5 axiom forall i:uint :: i <= mod(i)
6 axiom x == sub(y)
```

Figure 4.21: Removing the axioms that do not contain any constants or functions that the imperative part requires.

The analysis of axioms is similar to that for assumptions in that we remove all axioms that do not share the same cone of influence with the criterion. It works as follows: First we inspect the axioms for any dependencies between the constants and functions that are used by the imperative program and those that are expressed in the axioms. We add these dependencies to the set of constants and functions and keep iterating until we have reached a fixpoint. Once we have reached a fixpoint, we remove the axioms that do not express a property on any of the constants or functions contained in our set. For our example declarations and imperative program we start with the set of constants and functions $\{\texttt{x}, \texttt{y}, \texttt{add}\}$. Inspecting the axiom stating that the value of `x` is equal to the value of the application of `sub` to the constant `y`, we add the constant `y` and the function `sub` to our set as they help to define the state of `x`.

Continuing to iterate through the axioms we will reach a fixpoint with the set of constants and functions $\{\texttt{x}, \texttt{y}, \texttt{add}, \texttt{sub}\}$. Using this set we can now take a second pass through the axioms removing those that do not contain any of the variables in the our set of constants and functions. We remove the axiom stating that every element of type `unit` is less than or equal to the value of the application of `mod` to the element as the function `mod` is not in our set of constants and functions. Although the axiom speaks about all the elements of `uint`, nothing it says about a `uint` is important for the program slice as we have assumed that all global variables and constants are feasible, so the axiom no longer contributes to the values of the variables or criterion as we have already assumed that they satisfy the axiom, so we can safely remove these axioms from the background declarations. We also remove the axiom stating that every `uint` is greater than or equal to `0` as the constant `0` is not in our set. The axiom stating that every element of `Fruit` is removed as we `isRipe` is also not in our set. The rest of the axioms are kept as the do contain at least one constant or function in our set of relevant constants and functions.

The correctness of this approach is similar to the correctness of using $\widehat{\gamma}$ and $\widehat{\Delta}$ for variables and assumptions. Once we have computed the set of all relevant constants and functions for the axioms, we must also be sure to keep all definitions for those constants and functions as we may have picked up some new constants and functions from analyzing the axioms. In the same vein as for global variables, we must also be sure to keep all

type declarations that are used to declare those constants and functions. As the proof follows the same vein of reasoning for assumptions we do not present it here.

By removing all global variable, constant, function declarations, and axioms from the background part that the imperative program does not depend on will help the programmer understand the background theories that contribute to the behaviors of the imperative program. We have shown how to slice away each of these types of declarations and preserve the semantic property of partial program slicing.

4.6 Summary and Related Work

In this chapter we have relaxed our working definition of program slicing to allow the possibility of removing partial commands from programs. We have constructed a program slicer $\widehat{\Delta}$ that will remove those partial commands that do not contribute to the behaviors of the variables in the criterion. To ensure that $\widehat{\Delta}$ does not accidentally remove any partial commands, we have constructed a function $\widehat{\gamma}_{+}^{\infty}$ that will iterate through the program and compute the set of all dependent variables for a given criterion. To ensure that we preserve the context of every data dependency we require the program to be in SSA form before we perform this analysis. We have formalized our relaxed semantic property of program slicing and have proven that for every feasible program path in an arbitrary program, the program slice will always produce programs that satisfy this semantic property.

Existing work on program slicing has not addressed the issue of slicing partial commands. The work on slicing as a program transformation [66] includes assume commands in their background section, however they fail to address them in their approach of program slicing. Path slicing [41] includes assumptions in their language, but their treatment does not include the possibility that the assumption may be partial. Their model of assumptions is used to model the guards of alternative commands (`if then else`) which are total. For our introductory example (Figure 4.1) their analysis would remove the assumption stating that the initial value of `total` is equal to `0`, which will violate the semantic property of partial program slicing.

Our approach uses assume commands as an instance of the more generic concept of partial commands. The generalized demonic update command is a generic command that is sufficient to describe all partial commands [4]. It is used in the theoretical refinement calculus to give a normal form to programs that include partial commands. The command has the given form $[\mathtt{x} := \mathtt{x}'|\mathtt{b}]$ where $\mathtt{x}'$ is bound to $\mathtt{b}$ and has the semantic meaning

$$\mathrm{wp}([\mathtt{x} := \mathtt{x}'|\mathtt{b}], R) = \forall x' \cdot (b \Rightarrow R[x := x'])$$

If we treat this command as an assignment to $\mathtt{x}$, an assumption on the free variables of $\mathtt{b}$, and a havoc on the variable $\mathtt{x}'$; we can construct a corresponding $\widehat{\gamma}$ that will treat the assumption on $\mathtt{b}$ as a normal assumption and the assignment to $\mathtt{x}$ as an update. The corresponding $\widehat{\Delta}$ would keep the command if $\mathtt{b}$ mentioned any variables in the criterion or if the target of the assignment $\mathtt{x}$ was included in the criterion. As we can handle this generic command, our approach is applicable to any language that includes partial commands.

Chapter 5

Localizing Errors in Failed Verification Attempts

Given a program $\mathcal{P}$, an automatic program verifier will attempt to verify that $\mathcal{P}$ will never fail. A program fails when an assertion in the program can evaluate to false when executed from a valid initial state. A sound program verifier such as Ξ must notify the programmer of a possible failure when it is unable to verify the program. An assertion validated by the verifier shows that the programmer has understood the program and there is no reason for the programmer to inspect the program. However, if an assertion is not validated by the program verifier, then it is up to programmer to inspect the program to understand why the assertion failed. A program slicer such as $\widehat{\Delta}$ can help the programmer with this inspection.

The program in Figure 5.1 assigns the value `0` to `total`, asserts that `sum` has the value `0`, either assumes `sum` to have the value of `y` or it assigns an arbitrary value to `sum` and assigns `total` the value of `z`, then it checks after both branches that `total` has a value greater than or equal to `0`. As the initial state of `sum` may not have the value of `0` or the initial state of `z` may have a value that is not greater than or equal to `0`, a sound program verifier will notify the programmer of either of these failing assertions.

A programmer trying to understand a failing assertion will first inspect the assertion to check that the assertion expresses what they have

```
1 total := 0;
2 assert sum == 0;
3   assume sum == y
4 ||
5   havoc sum;
6   total := z
7 assert 0 <= total
```

Figure 5.1: A sound program verifier will report to the programmer either the assertion on `sum` or the assertion on `total` may possibly fail.

```
1 total := 0;
2 skip;
3   skip
4 ||
5   skip;
6   total := z
7 assert 0 <= total
```

Figure 5.2: Slicing the program for the failing assertion on `total`.

intended. If the assertion is what the programmer has intended then it is not the cause of the verification failure. The programmer will then inspect the program for the cause of the error by searching for commands that either update a variable in the failing assertion or state a property on the variables of the assertion. For our example, if the failing assertion is the assertion on `total` then the programmer will inspect the program for assignments or assumptions on `total`. An experienced programmer will keep the assignment to `total` and remove the havoc to `sum` and the assumption on `sum` as they do not affect the value of `total`. The assertion will be removed as the programmer as assertions do not effect the state of the program. The assignment of `0` to `total` will also be kept. An experienced programmer will not inspect the entire program, they will only inspect the program in Figure 5.2 for the cause of the failed verification attempt and see immediately that it is the initial value of `z` that causes the failed verification attempt.

A programmer understanding a failed verification attempt can automate this removal of commands by using a program slicer. Given the set of free variables of the failing assertion as the initial criterion, a programmer can use $\widehat{\Delta}$ to produce the program slice that an experienced programmer would use when understanding failed verification attempts. Given the failed verification attempt and the program, this can be entirely

automated without any interaction from the programmer. The resulting program slice exhibits the same behavior of the original program for the failing assertion.

In this chapter, we show how $\widehat{\Delta}$ can be used to help programmers localize the cause of the error. We give some indication as to what properties a programmer can expect from this program slice and what they can expect from modifying this program slice to satisfy the failing assertion. A failed verification attempt from a verifier such as Ξ gives us more than the location of the failing assertion, it also gives us a possible program trace for the failed assertion. We show how this trace can be used to give finer program slices. We have implemented our approach and give some indication on the usefulness of program slicing in the wild. We finally conclude with some discussion and related work.

5.1 Localizing Verification Errors

Programmers use assertions to understand programs. An assertion that is not reported as failing by a sound program verifier suggests that the programmer has understood the program and no longer needs to understand the behavior of the program. However, if the program verifier notifies the programmer that an assertion may fail, it is the responsibility of the programmer to understand why the program has failed. Along with the location of the failing command, our model program verifier Ξ will give a trace leading to the failing command. From this and the program location of the failing command we can automate the slicing of the program and present to the programmer a smaller, simpler program.

Given our introductory program (Figure 5.1), the program verifier Ξ will notify the programmer that the command at program location `7` may fail with the possible program trace $\mathcal{T}$, where $\mathcal{T}$ contains the program locations `1`, `2`, `5`, `6` and `7`. Inspecting the program at the location `7` will give us the assertion that `total` is greater than or equal to `0`. Using the trace given by the program verifier we can compute the path that the program uses to reach the failing assertion. For our example program the path for the failing assertion is the path that only takes the bottom branch of the non-deterministic choice. Given the path and the free variables of the failing assertion, our approach to helping locate the cause of the error is to invoke the program slicer to remove those commands in the path that

do not contribute to the behavior of the variables in the failing assertion and present to the programmer the sliced program path.

Given a negative response $\langle \bot, \ell, \mathcal{T} \rangle$ by a program verifier such as Ξ for a program $\mathcal{P}$, we automate our approach to helping the programmer locate the verification error as follows: First, we compute a program path from the given program trace $\mathcal{T}$. In our language a program trace is simply a program path, but for a language with looping or recursion computing the path is done by determining which path through the program the trace took. Once we have computed the program path $\rho \in \mathcal{P}$, we inspect the failing command at the program location ℓ given by the program verifier. We take the free variables of the failing command $\rho[\ell]$ at program location ℓ and invoke $\widehat{\gamma}_{+}^{\infty}(\rho, \mathrm{FV}(\rho[\ell]), \varnothing)$ with the program path ρ. Once $\widehat{\gamma}_{+}^{\infty}$ is done computing the set of data and anti-dependent variables, we take only the data dependencies (viz. projection) and use this as the set of criterion for calling $\widehat{\Delta}$ on our program path ρ. Once $\widehat{\Delta}$ has finished computing the program slice for $\widehat{\Delta}$, we present this program slice along with the assertion that failed.

In summary, given a failed verification attempt $\langle \bot, \ell, \mathcal{T} \rangle$ from Ξ for a program $\mathcal{P}$, we automate the entire approach and present to the programmer the program slice computed by

$$\widehat{\Delta}(\rho, \mathrm{prj}_1(\widehat{\gamma}_{+}^{\infty}(\rho, \mathrm{FV}(\rho[\ell]), \varnothing)))$$

where ρ is the program path computed from the trace given by Ξ and prj_1 returns the first projection of the tuple.

5.1.1 Validity of the Slice

A sound automatic program verifier ensures that if a positive response is given to the programmer, then the weakest precondition of the program is valid for all initial states of the program. A negative response from an automatic program verifier, however does not give any conditions on the program other than the fact that Ξ could not verify the program. That is, the error may be spurious and not an indication that the program is feasible. The semantic property of partial program slicing is guarded by feasibility of the original program and the failing assertion is implied by the projected postcondition of the program slice. As the failed verification attempt does not ensure feasibility, we rethink what it means for

equivalence between the program and its slice when applied to automatic program verification.

The program in Figure 5.3 assigns the variable `sum` the value `1` and then assumes that the value of `sum` is equal to `0`. It then either assumes that `sum` has the value of `y` or it havocs the value of `sum` and then assigns the value of `z` to `total`. The program then checks that the value of `total` is greater than or equal to `0`. A sound program verifier such as Ξ can notify the programmer that the assertion at program location `7` is possibly failing, even though a miracle always occurs as the program assumes the value of `sum` to be both `1` and `0` at the same time.

```
1 sum := 1;
2 assume sum == 0;
3   assume sum == y
4 ||
5   havoc sum;
6   total := z
7 assert 0 <= total
```

Figure 5.3: A sound program verifier can report to the programmer the assertion on `total` may possibly fail, although the program contains a miracle by the assignment of `1` to `sum` and assuming `sum` has the value `0`.

```
1 skip;
2 skip;
3   skip
4 ||
5   skip;
6   total := z
7 assert 0 <= total
```

Figure 5.4: Removing the miracle induced by assuming `sum` is equal to both `1` and `0`.

As there is always a miracle in the program, the program slicer is free to remove or keep any commands that the slicer desires. Our semantic property for partial program slicing states that either the program is infeasible or not terminating or both programs must satisfy the same projected postcondition when they terminate. If the program is infeasible, then we cannot be sure if the program slice preserves the same projected predicates of the original program as all predicates are satisfied in the original program and may not be satisfied in the program slice.

If we know that the program fails for the predicate transformer wp for

some predicate R, then we know that the wp will also fail the impossible state $\bot$ by anti-monotonicity of wp and the program will have a feasible state. However, a sound verifier only ensures that if it gives a positive response, then wp will hold for the program. If a sound verifier gives a negative response, then either the weakest precondition does or does not hold. That is, we cannot be sure if the program is actually invalid or if the program verifier was simply unable to verify that the program was valid. Although an infeasible program trivially satisfies our semantic property of program slicing, it does not seem right to give the programmer a program that may not satisfy all projected predicates of the original program. Generally, SMT solvers will pick up on this infeasibility and will verify most programs that always contains miracles, however this is not a hard rule and we cannot be ensured of this by the solver.

However, a programmer will generally not think of comparing the two programs using the semantic property of program slicing, they will think of comparing two programs against the verifier. If an assertion fails in the original program, they expect the assertion to fail in the program slice with the same results as the failure in the original program. If the program verifier was unable to determine that the original program contained a miracle and the miracle is taken out in the program slice, then the program verifier knows no better and will be is unable to verify the program slice. Assuming that our verifier Ξ is stable, we can ensure that if the original program failed verification for the command located at program location ℓ, then the command located at location ℓ in the program slice will also fail to verify.

Proposition 5. *Suppose* $\rho \in \mathcal{P}$, $v \in \mathbf{Vars}$, $v = \mathrm{prj}_1(\widehat{\gamma}_+^\infty(\rho, \mathrm{FV}(\rho[\ell]), \varnothing))$, *and* $\rho' = \widehat{\Delta}(\rho, v)$. *Then,*

$$\Xi(\rho) = \langle \bot, \ell, \mathrm{prj}_v(\mathcal{T}) \rangle \Rightarrow \Xi(\rho') = \langle \bot, \ell, \mathcal{T} \rangle$$

where $\mathrm{prj}_v(\mathcal{T})$ *is the projection of all the variables in* v *for the trace* $\mathcal{T}$.

The machinery require to show the correctness of such a proposition is beyond the limits of this thesis, however the basic idea would be to show that Ξ will always behave the same when the commands of the program work on the same states and both lead to the same failing assertion. The contra-positive of this property states that if Ξ returns a positive response

for a program slice, then it will return a positive response for the original program. This allows the programmer the possibility to fix the cause of the verification failure in the program slice and know that this fix when put back into the original program will lead to a successful verification.

We cannot in general provide semantically correct program slices that are unverifiable by an incomplete program verifier. For example, if the verifier were able to determine that the assumption on `sum` always produces a miracle, it would then verify our example program in Figure 5.3. The program slice for the criterion {`total`} would remove the assumption on `sum`, thus removing the miracle. By removing the miracle we are producing a program slice where any assertion on `total` is may no longer be verifiable as the miracle induced by the assumption on `sum` has been removed.

5.2 Accidental Triggers

A sound program verifier can employ any trick it wants to help with the verification of the program as long as a positive response ensures that the weakest precondition of the program is satisfiable. A programmer attempting to understand why an assertion succeeds can employ the use of our program slicer to remove those commands that do not contribute to the evaluation of the variables in the assertion. If the program is infeasible for variables not contained in the data dependencies of the criterion, we may give back to the programmer a program where the assertion no longer is verifiable by the program verifier. However, if the original program is known to be feasible, such as for programs containing only total command, then slicing a verifiable program should produce a verifiable program when sliced.

However, depending on the analysis used by a program verifier, this may not necessarily hold. Automatic program verifiers make use of various techniques to help the underlying SMT solver verify the correctness of a program. One such technique is to encode the verification conditions of the program so that their logical structure helps the prover verify their correctness. Provers based on SMT solving [22, 20] make use of this structure to help determine which triggers are needed to verifying the correctness of these conditions. One such trick is to treat assertions as both an assertion and an assumption. That is, given an assertion

`assert Q` and a postcondition R, instead of proving $Q \wedge R$, a program verifier like Ξ may encode the assertion as $Q \wedge (Q \Rightarrow R)$ where the proof of R may use the assumption on Q. This analysis is still sound as its correctness implies $Q \wedge R$, but helps the solver find the right trigger to prove the postcondition R.

The program in Figure 5.5 executes some commands and then asserts that the value of `s` is equivalent to the sum of all values contained within the array `a`. It then asserts that the value at position `n` in the array `a` also contains the summation of all values in `a` up to the index of `n`. The program then adds the $\texttt{n}^{th}$ value to `s` and increments `n`. The program then asserts that `s` is now the sum of all indexes in the array up to the new value of `n`, which is equivalent to the old value of `n` plus one. We assume the first two assertions are provable and we assume that the program verifier is able to prove the final assertion on `s`.

```
1 ...;
2 assert s ==
3   sum i:(0:n-1) :: a[i];
4 assert a[n] ==
5   sum i:(0:n-1) :: a[i];
6 s := s + a[n];
7 n := n + 1;
8 assert s ==
9   sum i:(0:n-1) :: a[i]
```

Figure 5.5: A program that makes use of the builtin `sum` operator. An expression of the form `sum i : (j : k) :: E` behaves exactly like summation in mathematics. We assume the ... is a program that is able to prove the correctness of the first two assertions.

```
1 ...;
2 skip;
3
4 skip;
5
6 s := s + a[n];
7 n := n + 1;
8 assert s ==
9   sum i:(0:n-1) :: a[i]
```

Figure 5.6: Removing the assertions on `s` and `a[n]` prevents the program verifier from using the right triggers.

If we assume that the original program (Figure 5.5) is feasible, we can slice it and produce the program in Figure 5.6. This program preserves our semantic property of partial program slicing, however if we attempt to verify this program with a sound program verifier such as Ξ we will find that the program will no longer verify. Removing the assertions on `s` and `a[n]` has an impact on the verifier proving the assertion at the end of the program as we have removed the structure of the formulas that are used by the solver.

To ensure that a program slicer preserves the verifiability of a program slice, it must keep all commands, including assertions, that mention any of the variables in the criterion. By keeping all assertions we can guarantee to the programmer that

$$\Xi(\rho) = \top \Rightarrow \Xi(\rho') = \top$$

will hold for a given feasible path ρ and slice ρ'.

5.3 Exploiting Counterexample Traces

Automatic program verifiers based on SMT solvers can return not only the location of the failing command but also a program trace that represents an execution of the program that is a *witness* to the failing command. Static program slicing is generally unable to slice away updates to complex data types such as arrays as the slicer is unable to determine if an update to the array has an effect on the value of an array position in the criterion. To this end, an update to an index into an array is treated as an update to the entire array. This approach ensures that the program slice is correct, but could potentially lead to overly large program slices that may contain array updates and assumptions that are not relevant to the slicing criterion.

Dynamic program slicing [1, 65] offers an approach to handle the problem of program slicing in the presence of complex data types such as arrays. Given an assignment of the form `m[i] := E`, the criterion {`m[j]`} and a program trace $\mathcal{T}$, dynamic program slicing will inspect the trace $\mathcal{T}$ for the value of the index `i` and the index `j` at the position of the assignment and compare the to values. If the two indexes are the same then we know that the assignment to `m[i]` will have an effect on `m[j]` and

we keep the assignment. However, if the trace states that `i` is not equal to `j`, then dynamic program slicing will remove the assignment from the program.

The program in Figure 5.7 havocs the array `h` at position `y`, assumes that the position `sum` in `h` has the value `0`, and then either updates `h[sum]` with the value of `y` or it havocs the position at `z` and then assigns `h[total]` the value of `h[x]` times `h[y]`. A program verifier will return a negative response for the assertion on `h[total]`. Slicing this program for the criterion {`h[total]`} will provide no benefit to the programmer as we cannot determine statically if the value of `total` is different than `z` or `sum`, thus keeping all updates or assumptions on the array `h`. However, given a program trace we can determine if an update to index `sum` or `z` also is an update or assumption on the index `total`, and if it is not a shared update then we can remove the update as we know that the values are not the same.

```
1 havoc h[y];
2 assume h[sum] == 0;
3   h[sum] := y
4 ||
5   havoc h[z];
6   h[total] := h[x] * h[y]
7 assert h[total] == 7;
```

Figure 5.7: Slicing this program with a static slice will provide no benefit to the programmer.

```
1 havoc h[y];
2 skip
3   skip
4 ||
5   skip
6   h[total] := h[x] * h[y]
7 assert h[total] == 7;
```

Figure 5.8: Dynamically slicing the program with a counterexample trace where `total` is not equal to `z` nor `sum`, allowing the slicer to remove the havoc on `h[z]` and the assumption on `h[sum]`.

A dynamic program slicer will take the program in Figure 5.8, a trace $\mathcal{T}$, and the criterion {h[total]} and produce the program in Figure 5.8. If the path given by the program verifier is the top branch in the program, a dynamic program slicer will first inspect the assignment to h[total]. As h[total] is in the criterion this command is kept and h[x] and h[y] are added to the criterion, while removing h[total]. When the dynamic program slicer inspects the havoc to h[z], it will inspect the program trace for all possible values of x, y, and z at program location 5 and compare them. If the set of values of z are disjoint from x and y, then the program slicer will remove the havoc on h[z] as this does not affect the value of h[x] nor h[y]. If they are not disjoint, then the havoc must be kept as it does have an effect on the value of h[x] or h[y] at some point in the trace. For our example, let us assume that the values are disjoint and the dynamic program slicer can remove the havoc to h[z]. The next command, the assumption on h[sum] proceeds in the same way. If the values of sum at program location 2 are disjoint from x and y we can remove the assumption on h[sum] as it has no effect on the values of h[x] and h[y]. Let us assume the variables are disjoint and the slicer removes the assumption and gives us the program in Figure 5.8.

It would appear that using the trace given by the program verifier to dynamically slice the program would be a good idea to help the programmer understand the failed verification attempt. And, if we could trust that an error given by the program verifier was an actual error in the program, this would be true. However, we are dealing with an incomplete verifier Ξ and this verifier gives no guarantees to the validity of a failed verification attempt. Static program slicing produces a program slice that is valid for all possible traces in the program. Slicing the program with the trace given by the program verifier may remove commands from the program that are the cause of the verification failure, as the error given by the verifier may be spurious and the trace represents a valid trace in the program; thus misleading the programmer.

For example, in our introductory program (Figure 5.7) the trace given to us by the program verifier allowed us to remove the assumption on h[sum] as the values given by the trace for sum and x or y were disjoint. However, if the value of x or y had the same values of sum, then the program would definitely fail as h[total] will have the value 0 when assigned the product of h[x] and h[y]. By not presenting this assumption to the

programmer, we are doing one of two things: either we are presenting a dynamic program slice for an actual program failure, such as when `h[x]` has the value `0`, or we are presenting a dynamic program slice for a program trace that does not fail. Removing commands from the program that contribute to the actual failure produces an unsound program slice and will mislead the programmer.

Knowing that a program trace is an actual valid failing trace and performing dynamic program slicing *is* a really good idea to help the programmer localize the cause of the verification failure in languages that include aliasing. However, this is only useful if the program verifier is complete or if the counterexample has been validated posterior. A sound verifier may produce a counterexample that is spurious and when used by a dynamic program slicer may remove commands that are necessary for understanding the failure.

5.4 Slicing Target Languages

Encoding the semantics of a programming language such as Spec# into a program verifier is very complicated and prone to many errors. By using an intermediate language, this encoding is simplified as the intermediate language is closer to the source language while resembling the language used by a program verifier. Performing an analysis in our intermediate language should preserve the meaning of the analysis in the target language. That is, an analysis should commute over the translation. Slicing a program that has been translated from a target language may break this commutativity.

The program in Figure 5.9 assigns the value of `z` to `sum` and then checks if the value of `sum` is equal to `0`. If `sum` is equal to `0`, then the program executes the then branch and assigns `y` to `sum`. If it is not equal to `0` then the program executes the else branch of the program and assigns the value of `x` times `y` to `total` and then terminates. Slicing this program for conditional commands [68, 36] with the criterion {`total`} will give us the program slice in Figure 5.10.

A program slicer will first inspect the branches of a conditional command. If any command within the body of a conditional command is kept by the program slicer, then the guard of the command must also kept as the guard controls which branches of the program are executed.

```
1 sum := z;
2 if
3   sum == 0
4 then
5   sum := y
6 else
7   total := x * y
```

Figure 5.9: An example program containing a conditional choice.

```
1 sum := z;
2 if
3   sum == 0
4 then
5   skip;
6 else
7   total := x * y
```

Figure 5.10: Program slice of the conditional command for {`total`}. The guard on `sum` is kept as is the assignment.

By controlling which branch gets executed, the values of the variables in the guard have an indirect effect on the values in the criterion. For our example, the guard controls if the assignment to `total` happens. As the variables in the guard indirectly play a role in the value of `total`, we also add these variables to the criterion. The program slicer finishes the program slice by keeping the assignment to `sum` as its value effects the evaluation of the guard of the command and gives us the program slice in Figure 5.10.

Translating this conditional command program into our intermediate language gives us the program in Figure 5.11. The translated program assigns the value of `z` to `sum` and then non-deterministically chooses to either assume that the guard `sum` $==$ `0` holds and assigns `y` to `sum` or to assume that the guard does not hold and execute the assignment to `total` the value of `x` times `y`. Slicing this program with the criterion {`total`} will give us the program slice in Figure 5.12.

The weakest precondition of the conditional command [24] for the a projected predicate R where $\mathrm{FV}(R) \subseteq \{\texttt{total}\}$ is

$$(z = 0 \Rightarrow R) \wedge (z \neq 0 \Rightarrow R[total := x * y])$$

which is identical to the weakest precondition of the translation of the conditional command into our intermediate language. That is, the *meaning* of both of the programs are identical. The weakest precondition for

```
1 sum := z;
2   assume sum == 0;
3   sum := y
4 ||
5   assume sum != 0;
6   total := x * y
```

Figure 5.11: The translation of our example program into our intermediate language.

```
1 skip
2   skip
3   skip
4 ||
5   skip
6   total := x * y
```

Figure 5.12: Slicing the intermediate form of the program for the criterion {`total`}. The assumptions on `sum` are removed.

the program slice of our conditional command does not differ and gives us the exact same weakest precondition for the projected predicate R. That is

$$(z = 0 \Rightarrow R) \wedge (z \neq 0 \Rightarrow R[total := x * y])$$

however, the weakest precondition for our program slice of the intermediate program is

$$R[total := x * y]$$

thus breaking any relationship between the program slice of the target program and the program slice of our intermediate program.

The problem is that when $\widehat{\Delta}$ inspects the assumption on `sum` it treats it as a data abstraction on `sum`, not a control abstraction and removes it from the program slice. The program slicer does not know that the assumption is necessary to the execution of the program trace. To have $\widehat{\Delta}$ correctly slice these *control* assumptions we need to tell the program slicer whether an assumption is necessary or not for control flow of the program. We do this by including extra information in the translation that records whether an assumption models control flow or not. We extend the slicer $\widehat{\Delta}$ to keep all assumptions that are marked as necessary for the control flow of the program trace. We also include the *body* of the guards when marking the assumption so that the program slicer can remove the assumption if no commands in the body are kept in the program slice.

The assumptions `sum == 0` and `sum != 0` in our example program will both be marked as modeling control flow. Marking each assumption with the program labels $\{3, 6\}$ will tell the program slicer the assignments to `sum` at label `3` or `total` at label `6` are both control dependent on these assumptions. If the slicer keeps either of these assignments, then it also must keep the assumptions. If our source language contains a repetition command, we must also include additional information in the translated program that will instruct the program slicer to iterate through the body of the command until it reaches a fixpoint [68].

A similar problem occurs when a translation of a target program introduces separate commands that operate on different states into the intermediate program. Slicing the target program would keep the command that updates separate states as it is a single command that updates on both the states, but slicing in the intermediate program may only keep one update to part of the state as the other state may not be part of the criterion, also breaking the commutativity of slicing for target programs. The solution for this type of problem is to associate a set of program labels with each command that states to the program verifier that any inspection of the command should also include inspection of the other commands whose labels are included in the set.

5.5 Implementation Details

We have presented our approach to the localizing of program failures in our language using program slicing and what can be expected from our approach in the context of automatic program verification. The Boogie programming language [21] is a similar intermediate language that is used in the modeling and verification of various source languages [9]. The Boogie programming verifier [6] takes a Boogie program and transforms it to verification conditions that are then passed over to the Z3 theorem prover [20]. Our language is a subset of the Boogie language, but we have extended it to the entire Boogie language. In this chapter we present the implementation of our approach within the Boogie program verifier and an approach to help deal with the complexity of slicing heap like structures such as those given to us by the target language such as Spec#.

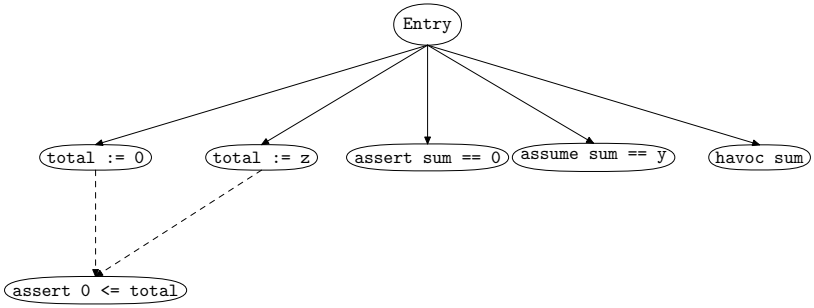

Figure 5.13: Program dependency graph for our introductory program. The solid arrows represent data flow dependencies and the dashed arrows represent control dependencies.

5.5.1 Program Dependency Graph

Applying our algorithms over and over again for the same program and the same verification failure can be inefficient. Unless the program or the variables in the specification have changed, we only apply our data flow and control flow analyses only once and we construct a program dependency graph. A program dependency graph (PDG) [28] represents a program as a graph in which the nodes are commands or predicate expressions and the edges represent either data dependencies or control dependencies. Program slicing in a PDG boils down to removing those commands that are not connected by either a control or a data flow edge to the criterion we are slicing [39].

The example graph in Figure 5.13 represents the program dependency graph for our introductory program in Figure 5.1. The bold-lined arrows represent data dependency and the dashed arrows represent control dependencies. The special node `ENTRY` represents the initial entry node for the program. The graph makes explicit that there is no data dependency between the assignment to `total` and the assertion on `sum`. It however makes explicit that there is a data-dependency between the assertion on `total` and the assignments to `total` in the bottom branch and the assignment to `total` in the header of the program, without any dependency on the havoc to `sum` or the assumption on `sum`.

To slice the program for the failing assertion on `sum` we select the

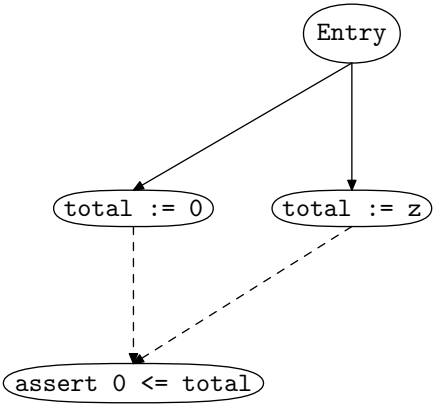

Figure 5.14: The program slice for the node containing the assertion on `total` gives us the sub-graph containing only the data dependencies of the node.

node for the criterion and remove all nodes that have no control or data dependency from the node. As the assertion has no data dependencies, the program dependence graph for the program slice of the assertion gives us only the singleton node. Slicing the program graph for the assertion on `total` will give us a more interesting graph. Inspecting the data dependencies, we find that the command has a data dependency on the assignment to `total` from the bottom branch of the program and a data dependency on the assignment to `total` in the header of the program. As there are no other control or data dependencies of the node, we are done slicing the program and will have the program dependency graph in Figure 5.14.

To handle the anti-dependencies introduced by partial commands, we extend the program dependency graph with a special anti-dependency edge that is used to include all nodes that define an anti-dependency between variables and partial commands in the program. For translations from a conditional command we include control edges that define a control dependency between the command in one node and an expression in the control node. For translations from a repetition command we also include the special edges loop-carried and loop-independent so that we

can correctly determine if a data dependency results from an iteration of the loop or if the data dependency is independent from the loop.

5.5.2 Prover Assistance

Program slicing in the presences of complex data types, such as arrays, possibly leads to overly-large program slices. When inspecting an array update `m[i] := E` for the criterion `{m[j]}` forces the program slicer to keep the assignment as it is unable to determine if the evaluation of `i` is equal to the evaluation of `j`. As this is generally an unsolvable problem, static program slicing over-approximates and keeps all assignments and assumptions that either assign to or define the array `m` for any index. Using the counterexample given by a negative result of the program verifier gives us an evaluation for every index and allows us to compare two indexes and determine if the array assignment has an effect on the index we are interested in. However, the trace may not be indicative of an actual error, and may mislead the programmer into inspecting the correct part of a program.

Program slicing suffers tremendously when dealing with programs that make extensive use of structures such as arrays. Translating target languages like Spec# into our intermediate form requires the use of a global array to model the heap of the program. As every step in a Spec# program has the potential to modify this heap, slicing in the presences of heap like structures offers very little benefit to programs such as Spec#. Whenever we inspect an update to the heap, we have to treat this as an update to every index in the heap.

One approach we have tried in our implementation was to make use of the SMT solver to help the program slicer determine if an update to or assumption on an array can be removed from the program slice. When we inspect an update of the form `m[i] := E` with the criterion `{m[j]}` we generate an assertion checking to see if the expression `i` is not equal to the expression `j`. We then compute the weakest liberal precondition for this assertion and query the prover for the validity of the assertion. If prover is able to prove the condition, then we know that for all possible traces `i != j` and we can safely remove the update from the program slice. If the prover is unable to prove the condition, then either `i == j` or the prover was simply unable to prove the condition `i != j`, we keep

the assignment and continue slicing the program.

For some examples this approach did produce a finer program slice with an acceptable performance. However, for interesting Spec# programs this approach provided very little benefit to the programmer. Either the approach took too long, which was the case for heap intensive programs or the approach provided very little benefit as the program simply allowed for the possible sharing of the references. We found that when a program updates two or more references of the same type, it was usually not the case that the program assumed (viz. precondition or invariant) that the references did not point to the same location. Without such an assumption, trying to prove that the indexes point to different locations in the model array of the heap is futile.

More work needs to be done for program slicing in the presence of complex data types such as heaps and arrays. The use of the prover to help determine if two references share the same location appears to be a good idea, but a more sophisticated analysis such as a points-to analysis [62] may help to give better results. However, it appears that our approach may already be on par with existing approaches [37] and there is simply more work to be done in this field.

5.6 Summary and Related Work

In this chapter we have presented our approach to how we can use a program slicer in the context of automatic verification to help the programmer locate the cause of the failed verification attempt. We have shown how our approach can be automated from a negative response from a sound program verifier. We have presented what the programmer can expect from the program slice in the context of sound program verification and what they can do with this program slice. Given the counterexample from the verifier we can use dynamic program slicing to produce finer program slices, as long as the programmer is wiling to accept an occasional invalid program slice. Translating from a target language requires more information than a simple translation to correctly slice these translated programs and we have presented the necessary information needed to ensure this. We have presented some insight into our implementation into the Boogie program verifier and gave some indication to the usefulness of the approach when applied to understanding

verification errors in Spec# like programs.

Localizing the cause of the verification failure using program slicing is an efficient approach only if the cause of the error is in the program slice. However, if the cause of the error is not located in the program slice, say an omission or mistyping of an assignment or an assumption, then program slicing may mislead the programmer by focusing their attention away from the actual cause of the error. For example, the program in Figure 5.15 fails to verify and the program verifier will tell us the assertion on `total` will possibly fail. Presenting the programmer with the slice in Figure 5.16 may mislead the programmer into believing that the cause of the verification failure is located in this program, and not a mistyping on the variable `sum`.

```
1 total := 0;
2 z := 0;
3   sum := y
4 ||
5   havoc z;
6   total := sum
7 assert 0 <= total
```

Figure 5.15: The assertion on `total` fails verification as the programmer mistyped `sum` for `z`.

```
1 total := 0;
2 skip;
3   skip
4 ||
5   skip;
6   total := sum
7 assert 0 <= total
```

Figure 5.16: The program slice for the failing assertion. The assignment to `z` is left out of the program slice and the programmer is mislead into believing that it is the value of `sum`, not the mistyping that has led to the failure.

Let us assume that the cause of this failed attempt is that the programmer mistyped the variable `sum` for the variable `z` when typing in the assignment to `total` at program location `2` in the original program. As this assignment to the variable `z`, of which `total` does not have a dependency on, is left out of the program slice, the programmer may be led to believe that the cause of the failure has to do with the value of `sum` and

not the mistyping from z. However, we do not believe that this is a very large issue in practice. Industrial verification of programs typically start with an existing program and add specifications to the program, thus checking if the specifications satisfy the program. That is, modern program verification is an a posteriori approach when it comes to applying specification driven development. As such, it is usually the specification that is incorrect, not the program. So omissions and mistypings are rarely a problem in these situations.

The idea of localizing the cause of the verification failure is not new. By comparing successful traces with error traces, the work on localizing errors in counterexample traces [5] is able to narrow down the cause of the error to particular branches in the program. Given a successful trace and a failing trace they compare the intersection looking for branches that differ between the two traces. If a branch choice has lead to a successful trace in one branch, but an error trace in the other branch; then we know that the cause of the error has to be in this branch. This approach works very well in programs that includes extensive control flow graphs, such as devices drivers and for specifications that are known to be correct. However, this approach does not work well for straight-line programs as there are no branches to compare or when the error is in the specification and not the program. Our approach is applicable to both in-line programs and programs where the specification is the cause of the failure, not the program.

The work by Groce [32, 31] generalizes this approach to help the programmer not only localize but also understand the failed verification attempt. They do this by comparing all positive (successful) traces against all negative (failing) traces and generalizing the result into either a program invariant or a precondition for a failing method. Their approach is based on bounded model checking, not on deductive verification such as Ξ. Computing the set of all positive and negative traces is a side-effect of model checking, so applying their approach is straightforward. However, unlike model checking, applying their approach in a deductive verification setting would not work well as computing the set of all positive and negative traces would requires us to produce one verification condition for every possible trace in the program and then invoke the theorem prover to determine if this trace is positive or not.

Part II

Debugging Program Errors

Chapter 6

Understanding the Cause of Verification Failures

When a program fails verification it is the responsibility of the programmer to determine the cause of the verification failure. Applying a technique such as program slicing may help the programmer localize the cause of the failure, but it may not help them understand *why* the program failed verification. When a verification attempt fails, SMT solvers provide a counterexample that illustrates why the verification condition is not valid. A counterexample essentially contains a sequence of program labels and values for each variable in each execution state and characterizes an execution of the program being verified. For programs with non-trivial states, for instance heap data structures, counterexamples can be magnitudes larger than the program being verified and not very helpful for the programmer to understand the verification error. If the verification error is spurious the counterexample might not even represent a failing execution.

Consider the example Spec# [46, 9] program in Figure 6.1. The method `AddSorted` of class `SortedList` adds the `value` parameter to the list of integers and then sorts the list. The specification requires that after the execution of `AddSorted`, the list be sorted (by an object invariant) and that it contains `value` (by a postcondition). A modular verifier such as Spec# verifies each method individually and reasons about

method calls in terms of the callee's specification, not its implementation. The specification of `Sort` states only that the elements of the list will be sorted, not that they will be preserved. Consequently, the verifier is unable to prove that `value` is still contained in `list.Elements` after the call to `list.Sort` and, thus, that the postcondition of `AddSorted` holds.

```
1  class IntList {
2    int[] Elements;
3    int Count;
4
5    void Add (int value)
6      modifies Count;
7      modifies Elements;
8      modifies Elements[*];
9      ensures Contains
10       (Elements, value)
11   { ... }
12
13   void Sort ()
14     modifies Elements[*];
15     ensures Sorted (Elements);
16   { ... }
17 }
```

```
1  class SortedList {
2    IntList list;
3
4    // The list is sorted
5    invariant Sorted (list.Elements);
6
7    void AddSorted (int value)
8      ensures Contains
9        (list.Elements,value);
10   {
11     list.Add (value);
12     list.Sort ();
13   }
14 }
15
16
17
```

Figure 6.1: Spec# is unable to verify `AddSorted`. The notation `Sorted(a)` abbreviates the condition that array a is sorted and `Contains(a, v)` abbreviates that v is contained in `a`. Both conditions can be expressed in Spec# via quantification over the indexes of a. The modifies-clauses specify frame properties by listing the locations a method is allowed to modify. For brevity, we omit the method bodies in class `IntList`, access modifiers, as well as Spec#'s ownership and non-null annotations.

A programmer who may not understand the cause of this failure can query the program verifier for a counterexample. For our introductory program, the counterexample for the failed verification attempt is over 1,200 lines of text. This counterexample represents the raw view of how the underlying SMT solver, Z3 in this instance, sees the Spec# program after it has been translated into our intermediate language. A programmer manually inspecting this counterexample will be at a loss as it is not amenable to any kind of manual inspection. Moreover, the counterexample may not even be representative of an actual error, thus possibly

misleading the programmer into understanding something different then an actual error in the program.

The literature contains several proposals for extracting useful information from counterexamples, but these proposals are not sufficient in the context of deductive program verification. In particular, they do not support the programmer in detecting too weak specifications and spurious errors, which are common reasons for verification failures. Some verifiers apply heuristics to extract those parts of a counterexample that are likely to be relevant for the verification error. However, it seems difficult to tune these heuristics such that they provide all necessary information without swamping the programmer with irrelevant details. For instance, for method `AddSorted`, the Spec# verifier filters too aggressively and provides only the following excerpt from the counterexample, which does not help the programmer understand the cause of the error: `(initial value of: value) == 0`.

Another approach is to construct a test case from a failed verification attempt, using the initial state of the counterexample as test input [16, 11, 63]. This approach is helpful if the test leads to a runtime error or if the violated specification can be found by a runtime assertion checker; in both cases, the generated test case will fail and provide useful information to the programmer. However, when static verification fails because of weak specifications, or when the violated specification is not checked at runtime, for instance when quantifying over all objects, or when the error is spurious, the test case will complete successfully. In these very common cases, the test does not help the programmer to determine the cause of the verification error and might even mislead the programmer into believing that the error is spurious. In our example, executing method `AddSorted` as part of a test case will succeed because the body of `Sort` does preserve the elements in the array, it is the weak specification that causes static verification to fail. However, this is not revealed by the test and the programmer is misled into believing this is a spurious error.

Finally, alternative forms of presenting the counterexample to the programmer, such as those based on graph visualization [58, 71] are limited by the size of the state presented. It is unclear how to present large counterexamples that contain heap like states. Moreover, this approach is unable to check the validity of the error, thus misleading the programmer into debugging a possibly spurious error.

In this chapter we present a second approach to helping programmer understand failed verification attempts. We do this by enabling programmers to use program debuggers to inspect program verification and counterexamples just as they used debuggers to inspect program executions and execution states. We construct an executable program that will enable the programmer to step through the verification of a method, check the validity of failing assertions, and observe the evolution of the state described by the counterexample. We will first present how our approach can be used to understand the verification failure in our introductory example. We will then go into the details of our approach and understand how we construct program stubs that reproduce the state of the counterexample and simulate the verification semantics of Spec#. We will see how we can extend the runtime checker to check for all failing specifications in Spec# and how we can use this to validate failed verification attempts. We will briefly discuss our implementation before we give a systematic debugging procedure that we have found to be effective in using our approach to understand failed verification attempts. We will then conclude this chapter with a summary and some discussion on related work.

6.1 Approach

Given a Spec# program and a counterexample produced by Z3, we construct an executable .NET program that simulates the verification semantics and reproduces states given by the counterexample. The constructed program can be executed in the Visual Studio program debugger, allowing the programmer to systematically and efficiently explore the counterexample and observe the failure. By executing the constructed program, we are able to detect spurious errors and validate failed verification attempts. The three key features of our approach are as follows:

(1) The constructed program *simulates the verification semantics* of the program as defined by the verifier rather than the concrete execution semantics as defined by the .NET platform. The semantics used by a program verifier is typically an abstraction of the execution semantics. Loops are typically verified via loop invariants rather than by considering the actual iterations, and modular verifiers reason about method calls in terms of method specifications rather than the implementation of the

called method. By simulating the verification semantics rather than the execution semantics, we can detect verification errors caused by incorrect or incomplete specifications.

(2) The constructed program *reproduces the states given by the counterexample*. We execute the constructed program in the initial state described by the counterexample. For each statement whose verification semantics differs from the execution semantics, we reproduce the effect of executing the statement by creating a program stub that alters the state as described by the counterexample. This allows programmers to use the debugger to explore and navigate through the counterexample.

(3) The constructed program *contains runtime checks for specifications* that are relevant for the verification error. For those specifications that generally cannot be checked efficiently at runtime (for instance, frame specifications, which universally quantify over all allocated objects), we use the counterexample to determine which objects are relevant for the verification error and focus the runtime checks on those. Moreover, checking the relevant specifications at runtime allows us to determine whether or not a verification error is spurious. This is the case if the constructed program terminates without a runtime error or specification violation.

Our approach enables the programmer to understand the failed verification attempt in method `AddSorted` as follows: We extract the initial state from the counterexample and construct a program driver that will create a `SortedList` object that contains an `IntList` object (in field `list`) with a list containing the elements, say, `0` and `1`. We then rewrite the body of `AddSorted` so that it simulates Spec#'s verification semantics. That is, we replace the calls to `Add` and `Sorted` with program stubs that change the program state to the state given by the counterexample. The stub for the call to `Add` changes `list.Elements` to contain the elements $[0, 1, -3]$[1]. The stub for the call to `Sort` updates the state of `list.Elements` to some sorted array, say $[7, 7, 7]$. We finish by constructing a runtime check for the invariant of `SortedList` and the postcondition (and modifies clause) of `AddSorted`. For each step in the construction, we insert debugger directives that allow the programmer to control the execution of the *original* program, but observe the states of the *constructed*

[1] Given the weak specification of `Add`, the counterexample could provide any array that contains the initial value of `value`, which we assume here to be -3.

program.

A programmer using our approach is presented with the original implementation of `AddSorted` highlighted by the program debugger. The programmer can either use the debugger to inspect the initial (counterexample) state or execute the method until either the runtime assertion checker notifies them of a failing assertion or the method terminates, notifying the programmer of a spurious error. In our example, the runtime assertion checker will notify the programmer of the failing postcondition `AddSorted`, thus confirming the verification failure. The programmer can then inspect the post-state of the method and observe the value $[7, 7, 7]$ for `list.Elements`. However, the initial state contained the state $[0, 1]$ for `list.Elements` and -3 for `value`. The programmer can now single-step through the body of `AddSorted` inspecting the (counterexample) state of each step. Stepping over the call to `list.Add` adds `value` to `list.Elements`, as expected. Stepping over the call to `list.Sort` changes `list.Elements` to $[7, 7, 7]$. This unexpected change points the programmer to the cause of the verification failure, the incomplete specification of `Sort`. Note that it is the simulation of the verification semantics that enables us to identify the incomplete specification as the cause of this verification error. Using the execution semantics, for instance in a test case, could exhibit only errors in the code.

6.2 State Construction

To simulate the verification semantics of the failing method we replace each command whose verification semantics differs from the execution semantics by a program stub that alters the state as prescribed by the counterexample. In doing so we are essentially lifting the SMT counterexample to the source level of the program. Both for this purpose and to set up the initial state of the method execution, we extract information from the counterexample and construct the corresponding state.

A counterexample contains values for all local variables in each execution state; we use those to extract the method arguments. Moreover, it contains function interpretations, in particular, for the select and store functions that are used to encode the heap; we use those to extract field values. As the counterexample only contains primitive types or arrays, the extraction is relatively simple and works for all counterexamples. For

primitive variables the extraction is done by looking up the value of the variable and for array variables we inspect the function interpretations of the array and use their values to recursively extract the elements of the array.

In this section we describe the construction of *mock types* that replace the original types in the program with versions that enable flexible initialization, of *program stubs* that construct the state given by the counterexample, and of the entry point to the failing method, the *driver*.

6.2.1 Partial Type Mocking

For variables of builtin types such as `int`, `bool`, and arrays, constructing the state consists of straight forward assignments. For variables of user-defined types such as classes and interfaces, the state construction involves the creation of objects and the initialization of their fields according to the state given in the counterexample.However, fully initializing an object is not possible if we do not know the values of all the fields of the object, due to visibility, or if the state construction makes use of user-defined types based on interfaces or abstract classes that do not provide a suitable constructor.

Mock objects [51, 29] are used to simulate the behavior of external interfaces in a controlled way by extending an interface or class with a new subtype that will simulate the behavior of the object being mocked. Since not all types have default constructors, automatically constructing a mock object requires inspecting all type constructors and determining which constructor will initialize the mock object as closely as possible to the state described by the counterexample [63, 64]. However, such a "close enough" construction of objects cannot in general reproduce the exact state described as we may not know how to construct such an object. Our approach depends on being able to construct the exact state of the counterexample, not a close approximation. We achieve this by constructing a *partial* mock type where we do not require a fully constructed object, only an object whose fields are relevant to the failing specification.

We replace each user-defined type in the program by a mock type—a concrete class—that contains: (1) a parameterless constructor with an empty body, which allows the program stubs to instantiate the class;

(2) a declaration for each field that is accessible to the failing method or that is mentioned in a specification. If the field is of a user-defined type, we also construct a corresponding mock type and replace the field with the newly constructed type. We declare all fields of mock types public, which allows the program stubs to initialize them according to the counterexample via field assignments. As we replace all method calls with a program stub there is no more construction do be done on the mock type. This type replacement is done on the .NET level and is transparent to the programmer.

6.2.2 Program Stubs

We replace each statement s whose verification semantics differs from the execution semantics by a program stub. This stub simulates the verification semantics of s by constructing the state after the execution of s as described by the counterexample. For this purpose, we extract the state before and after the execution of s from the counterexample. For each variable or field in which these two states differ, the program stub contains an assignment that updates the variable to reflect the state change.

When updating variables of reference types, we must preserve any alias properties contained in the counterexample, that is, when two variables contain the same symbolic reference in the counterexample, they must also contain the same reference in the constructed state. So when we update a variable of a reference type, we first check if we have already constructed an object for the symbolic reference in the counterexample. If so, we assign a reference to that object. If not, we create and initialize a new object, making use of type mocking.

6.2.3 Driver

To begin executing the failing method we have to generate a *driver*, which constructs the initial state, attaches itself to the program debugger, and then calls the failing method. The initial state consists of values for the receiver, the method arguments, static global data, and all objects reachable from any of these instances. This construction is a special case of the state construction described in the previous subsection with the

only difference being that the driver needs to construct the entire state as there is no changes from a previous state. The programmer does not see this constructed driver, they only see the initial states produced by the driver with the program debugger attached to the failing method.

The driver for our example creates the initial state for the failing method `AddSorted`, in particular, the receiver of type `SortedList` (Figure 6.2, left column). In order to initialize this object, it first constructs and initializes an `IntList` object that will be assigned to the receiver's `list` field. For this purpose, we create an integer array of the length given in the counterexample (`2`) and directly initialize its elements with the values from the counterexample ([`0`, `1`]). We use this array to initialize the new `IntList` object. After the initialization of the `IntList` object, the driver creates and initializes the receiver of the failing method.

```
1 // Construct the array for IntList.Elements
2 int[] Elements = new int[2];
3 Elements[0] = 0;
4 Elements[1] = 1;
5
6 // Construct an instance of IntList
7 IntList list = new IntList ();
8 list.Elements = Elements;
9 list.Count = 2;
10
11 // Construct receiver of failing method
12 SortedList rcvr = new SortedList ();
13 rcvr.list = list;
14
15 // Attach to the program debugger
16 Debugger.Launch ();
17
18 // Set the first step of the debugger
19 Debugger.Step ("rcvr.AddSorted (-3)");
20
21 // Call the failing method
22 rcvr.AddSorted (-3);
```

Figure 6.2: The driver for our example first constructs the initial state, launches the debugger, and calls the failing method. The types `IntList` and `SortedList` denote the mock types generated for the classes with the same names, which declare parameterless constructors and public fields.

After the initial state construction, the driver launches the debugger, and then calls the failing method `AddSorted` on the constructed receiver with the argument value from the counterexample, -3 (Figure 6.2, right column).

6.3 Verification Semantics

Program verifiers such as Spec# reason about a program in terms of its verification semantics, which differs slightly from the concrete execution semantics. The two main differences are to reason about method calls in terms of a method's specification rather than its implementation (to achieve modularity, to handle recursion, and to handle dynamic method binding) and to reason about loops in terms of a loop invariant rather than actual iterations (to avoid inefficient fix- point computations). To detect common verification errors caused by incorrect or incomplete specifications, we model the verification semantics of the failing method. For this purpose, we rewrite its body, in particular, calls and loops, as described in this section.

Even though we rewrite the body of the failing method, we present the original failing method to the programmer in the debugger; the rewriting is completely transparent to the programmer. We achieve this effect by injecting debugger directives, in the form of calls to `Debugger.Step`. These directives highlight the code in the failing method body and allow the programmer to control the execution of the stubs from the original method body.

6.3.1 Method Calls

The verification semantics of a call to a method m is (1) to assert the precondition of m, (2) to havoc all memory locations that may be changed by m, according to its modifies clause, and then (3) to assume the postcondition of m. To simulate the verification semantics, we replace each call to a method m in the failing method, including recursive calls and constructor calls, with a program stub that contains: (1) a runtime check for the precondition of m (2) code that updates the state of the program to reflect the state given by the counterexample, and (3) a runtime check

for the postcondition of m; which will ensure that we present only valid states to the programmer.

Method `AddSorted` contains calls to `list.Add` and `list.Sort`. For each method call the counterexample contains a state describing the effect of the call. We replace these method calls with the program stubs in Figure 6.3. The program stub for the call to `list.Add` (left column) creates a new `Element` array, assigns the elements `0`,`1`, and−`3` into the array and then updates `list.Count` with the value `3`. We constructed a new `Elements` array as the symbolic reference in the counterexample differed from the pre-state of the call in the counterexample. For the call to `list.Sort`, we do not construct a new `Elements` array as the symbolic reference between the post-state of the call to `list.Add` is the same as the post-state of the call to `list.Sort`. However, the elements of the array have changed and we update each of the elements with their new values `7`,`7`, and `7`. The program stub then checks the postcondition of `list.Add`; which will be explained in more depth in Section 6.5.2.

Both program stubs contain a debugger directive to highlight the method call in the original failing method.

6.3.2 Method Calls in Specifications

Spec# as well as other specification languages for object-oriented programs such as JML [43] and Eiffel [52] allow specifications to contain calls to side-effect free methods. These *pure* methods are useful specification constructs that can be used to extend the axioms used by the program verifier [18]. Modular verifiers reason about pure method calls in terms of their specifications, not their implementation. For method calls in specifications, this is achieved by encoding the pure method as a mathematical function, whose meaning is derived from the specification of the pure method [19]. The verification semantics of a pure method call within a specification is then to apply the mathematical function.

For example, the postcondition of `AddSorted` in our introductory program (Figure 6.1) declares that there must exist an index into the list of elements that contains the value of `value`. If we assume that the class of `IntList` contains the pure method `Contains` that checks if the value is contained within the list and returns true if `value` is in the list of elements and false if not, we can instead have a postcondition of `AddSorted`

```
1 // Step over the method list.Add
2 Debugger.Step (list.Add);
3
4 // Construct the post-state of list.Add
5 int[] Elements = new int[3];
6 Elements[0] = 0;
7 Elements[1] = 1;
8 Elements[2] = -3;
9 list.Elements = Elements;
10 list.Count = 3;
11
12 // Check postcondition of list.Add
13 assert list.Contains(value);
14
15 // Step over the method list.Sort
16 Debugger.Step (list.Sort);
17
18 // Construct the poststate of list.Sort
19 list.Elements[0] = 7;
20 list.Elements[1] = 7;
21 list.Elements[2] = 7;
22
23 // Check postcondition of list.Sort
24 assert list.Sorted (value);
```

Figure 6.3: The program stubs replacing the calls to `list.Add` and `list.Sort` in the failing method `AddSorted`. The debugger directives instruct the program debugger to highlight the calls. The stubs construct the post-states of the calls given by the counterexample.

that now mentions the pure method `Contains` instead.

As with normal method calls, calling a pure method in a runtime assertion checker may not correctly simulate the verification semantics. An actual call to the pure method would return the value computed by its implementation, not the value given by the counterexample. To ensure that we present the correct state to the programmer, we extract the value of every pure method call from the counterexample and replace the occurrence with its value. This replacement is relatively simple and straightforward as there are no changes to the heap of the program.

The rewritten method body of `AddSorted` will end with a runtime check for the postcondition of the method. Replacing the existential

```
1 [Pure] bool Contains (int value)
2   ensures result ==
3     exists {int i in (0:Count);
4       Elements[i] == value};
```

Figure 6.4: The specification of `IntList`'s method `Contains`. The [`Pure`] attribute expresses that the method is side-effect free and may be used in specifications.

```
1 void AddSorted (int value)
2   modifies list.Count, list.Elements[*];
3   ensures list.Contains (value);
4   {
5     list.Add (value);
6     list.Sort ();
7   }
```

Figure 6.5: Replacing the exists expression with a pure method `Contains`.

postcondition with a pure method will require that we not only check that the postcondition holds for the value given by the counterexample, but we must also check that this value satisfies the specification of the pure method, thus validating the states of the counterexample. The program stub in Figure 6.6 is the replacement program for the postcondition of `AddSorted` using the pure method `Contains`. We create a new temporary value `result` for the value of the pure method call according to the counterexample and then assert that this temporary value satisfies the postcondition of `Contains`. We then replace the call to `Contains` with the result given by the counterexample. For this example the result given by the counterexample for the call to `Contains` has the boolean value *false*, thus the generated assertion for the postcondition of `AddSorted` will fail during executing, validating the verification failure.

If the pure method of `Contains` included a precondition, we would also generate an assertion to check the precondition with the value of `value`.

```
1  // Step into the postcondition
2  Debugger.Step("ensures list.Contains(value)");
3
4  // Construct result of list.Contains (value)
5  bool result = false;
6
7 // Check postcondition of Contains
8 assert result ==>
9   exists {int i in (0:Count);
10    list.Elements[i] == value};
11
12 // Check the postcondition of AddSorted
13 assert result == true;
```

Figure 6.6: Using the pure method `Contains` for the postcondition of `AddSorted` requires checking the validity of the value given by the counterexample for the call to `Contains`.

6.3.3 Loops

The verification semantics of a loop is: (1) to assert the loop invariant before the loop, (2) to simulate the state after an arbitrary number of (possibly zero) loop iterations by assigning arbitrary values to all locations that may be modified by the loop and assuming that the resulting state again satisfies the loop invariant. The verification semantics then considers two possibilities to continue the execution: (3) an arbitrary execution of the loop body by assuming that the condition of the loop holds, executing the loop body, and asserting that the loop invariant holds again after the body, or (4) exiting the loop by assuming that the condition of the loop does not hold and proceeding to the statement after the loop. Checking an arbitrary iteration of the loop suffices to ensure that any execution of the loop preserves the loop invariant.

To simulate the verification semantics for loops we replace each loop in the program with a program stub that contains: (1) a runtime check for the loop invariant, and (2) code that updates the state of the program to reflect the state given by the counterexample and another runtime check for the loop invariant, which we discuss the reason for in Section 6.5. From the counterexample, we know whether the verification error occurred on the path that contains the arbitrary loop iteration

(branch (3)) or the path that exits the loop (branch (4)). In case (3), the stub contains a runtime check for the loop condition , the loop body (replacing any method calls or inner loops), another runtime check for the loop invariant, and then terminates the execution of the method. In case (4), the stub just contains a runtime check for the negation of the loop condition and then proceeds with the code following the loop.

As we mentioned above, a programmer using our approach will not see the program stubs, but only the effect they have on the state of the program. If the error is located in the loop body, the execution as presented to the programmer enters the loop body; upon entry, the programmer will observe a sudden change of the state to the arbitrary state prescribed by the counterexample, satisfying the loop invariant and the loop condition. If the error is located after the loop, execution skips the loop entirely, also with a sudden change of the state to an arbitrary state that satisfies the loop invariant and the negation of the loop condition.

6.4 Extended Runtime Checking

We rely on the runtime assertion checker to reproduce failed verification attempts. An execution of the rewritten failing method that does not lead to an assertion violation indicates a spurious error. Most assertions in Spec# programs are executable. In particular, quantifiers that range over finite integer intervals, such as array indexes, are checked by iterating over the range. However, the verification semantics of Spec# also makes use of assertions that quantify over possibly unbounded sets, for instance, over all allocated objects. This occurs in the assertions for modifies clauses and object invariants of Spec#. Such assertions cannot be checked efficiently at runtime. To be conclusive about the spuriosity of an error, we must be able to check any failing assertion.

Our introductory program (Figure 6.1) contains an unbounded quantifier in the verification semantics. The modifies clause specifies which variables are allowed to be modified by a method body, and implicitly specifies which variables are not allowed to be modified by a method. In the verification semantics, this is encoded as an assertion over all possible variables in the program by checking that their pre-state value is the same as their post-state value. The modifies clause of `Add` in `IntList` specifies

that the method body is allowed to modify the value of `Count`, the reference location of `Elements`, and the elements contained within the array of `Elements`. The modifies clause of `Sort` specifies that it may modify the elements of the array `Elements`, but it is not allowed to modify `Count` nor the location of `Elements`. The modifies clause of `AddSorted` specifies that the method is allowed to modify the value of `list.Count` and it is allowed to modify the elements of `list.Elements`, but not the location nor any other variable in the program. The Spec# program verifier will notify the programmer that it is unable to verify the modifies clause of `AddSorted`.

The problem is that method `AddSorted` does not satisfy its modifies clause because the call to `list.Add` may modify `list.Elements`, but this location is not specified by the modifies clause of `AddSorted`. The counterexample for this error contains instantiations for the quantified variables in the assertion for `AddSorted`'s modifies clause. Here, these instantiations indicate that the `Elements` field of the object `list` is being modified without permission by the modifies clause. That is, we are given the actual instantiation that has violated the universal quantifier used to verify modifies clauses. Using this instantiation we can generate a runtime check for this verification failure that is efficient and will ensure that we can check if the error is spurious or not. We do this by generating a local variable that stores the initial value of `list.Elements` upon entry to `AddSorted`. At the end of the method we insert a runtime check, an assertion, that will check if the pre-state of `list.Elements` has not been modified. If the assertion fails, then we have validated the error from the verifier. If the assertion does not fail, then we have found a spurious error and we can notify the programmer. For our example, the program stub for the method call to `list.Add` changes the value of `list.Elements` by assigning a new array to the location. As the reference location is not preserved by the method, the assertion will fail and the programmer will be notified of the error.

A programmer debugging a modifies verification failure can localize the error efficiently by attaching a data breakpoint to the variable in the failing modifies clause and execute the program. If a statement modifies the variable, the program debugger will stop execution and notify the programmer of the modification. In our example, if we attach a data breakpoint to `list.Elements` and execute the program, the program de-

bugger will notify the programmer of a modification to `list.Elements` at the call site to `list.Add`. The programmer is now aware of the location of the verification failure and may now inspect the modifies specification of `list.Add` and see that the call may modify the location of `list.Elements`. With the knowledge the programmer can now fix the verification failure by weakening the modifies clause of `AddSorted` to include for the modification to `list.Elements`.

This approach works for all unbounded *universally* quantified specifications; such as Spec#'s verification semantics for object invariants. However, if the specification failure contains an unbounded *existential* quantifier, no efficient runtime can exist as the negation of the example would require us to check all instances of the unbounded quantifier. Moreover, the counterexample for a failing unbounded existential quantifier cannot contain any useful information about the failing assertion as it too would have to contain a possibly unbounded domain of the quantifier. However, this is generally not an issue as these unbounded existential quantifiers pose a large problem for program verification and do not exist in Spec# nor any other languages that target automatic verification.

6.5 Error Validation

A sound program verifier such as Ξ will return a positive response only if the program is verifiable by the wp predicate transformer. A sound verifier will return a negative response if the program is not verifiable by the wp predicate transformer or if the verifier was unable to ensure that the program does satisfy the wp predicate transformer. If the verifier returns a negative response for a verifiable program, then we say that the verifier has returned a *spurious* error. Along with the knowledge that the program may not satisfy the weakest precondition of wp, we are also given a counterexample that should represent an execution of the program that causes the program to fail. However, when we force the SMT solver to give us a counterexample, we may sometimes end up with program trace that is not contained in the set of possible traces for the program.

In this section we present how we identify spurious errors and validate the states of the counterexample.

6.5.1 Spurious Errors

Since the validity of verification conditions is undecidable, SMT solvers cannot always determine whether a verification condition is valid or not. Whenever the SMT solver does not provide a conclusive result, a sound verifier needs to be conservative and report a verification error. If the program is valid, but not verifiable by the SMT solver we say that the error is *spurious*. Spurious errors occur frequently in automatic program verification and are especially problematic as the harder the program or the specification, the more the likelihood of an error being spurious. For example, assertions that contain many quantifiers or non-linear arithmetic increases the likelihood of a possible spurious error.

By extending the runtime assertion checker to handle *all relevant* failing assertions in Spec#, we are able to check and validate verification failures. If the execution of the rewritten failing method terminates without a failed runtime assertion check for the failing specification, we can safely conclude that the error is possibly spurious and notify the programmer. As we only check for one execution trace of the program, we can only notify the programmer that the error may be spurious as the error may still be a valid error, the counterexample was just unable to validate the error. However, it is generally the case that a valid error will have a counterexample that represents the error and a spurious error will generally have a counterexample that does not fail.

Knowing that an error is possibly spurious is an invaluable piece of knowledge for a programmer understanding a failed verification attempt. When a programmer is understanding a failed verification attempt they inspect the program and specifications for the cause of the error, however they do this with a grain of salt as they know that there is the possibility that the program verifier was simply too weak to prove their program correct. If they are unable to find the cause of the error, even possibly missing the cause, they begin to check the limits of the verifier by decorating the program with assertions. This can be a very time consuming process. By executing the program and checking the failing specification we are able to free the programmer from this time trap and get them focused immediately on how to get the verifier to verify their program.

6.5.2 Invalid Counterexamples

A counterexample is supposed to satisfy all assumptions that are being made in the verification semantics of a program. For instance, the initial state in a counterexample is supposed to satisfy the precondition of the failing method. However, if the assumptions contain formulas that are beyond the capabilities of the prover, the programmer may be presented with an invalid counterexample that contradicts the assumptions in the program. For example, if a program method contains a precondition that uses non-linear arithmetic, say `x/y > 0`, most automatic provers will not be able to handle the precondition and may produce a state -563 for `x` and `4` for `y`; thus invalidating the assumption. Simulating the execution described by an invalid counterexample and checking assertions in states states extracted from an invalid counterexample is not helpful for the programmer to understand verification errors and may only mislead them.

We extract states from the counterexample in three cases: (1) to set up the initial state in the driver, (2) to reproduce the state changes made by a method call, (3) and to reproduce the state changes made by a loop iteration. For these cases, the verification semantics of Spec# makes the following assumptions about the expected state: (1) the precondition of the failing method, (2) the postcondition and modifies clause of a called method, and (3) the loop invariant and the loop condition. To protect against invalid counterexamples we introduce assertion check for each of these assumptions. When one of assertions fail, it indicates that the state of the counterexample does not satisfy the assumption and that the counterexample is invalid.

The verification of our introductory method `AddSorted` assumes the precondition of the method as well as the postconditions and modifies clauses of the called methods `Add` and `Sort`. After we construct the initial state in the driver stub we check the validity of the initial state by asserting the precondition of `AddSorted`. For our example, `AddSorted` has no precondition so this check is trivial. After the program stub for the method call to `Add` we check the postcondition of `Add` by asserting that there exists an index into the array of elements that contains `value`. After we construct the program stub for the call to `list.Sort` we assert that the list of elements are sorted. If any of these assertions fail, the programmer will be notified of the invalid states given by the counterexample and that

they should no longer trust the execution of the program.

Assumptions on the state may also come from the axioms that are used by the program verifier to reason about the encoding of Spec# programs. These assumptions are encoded as background axioms for the verifier and they specify the ranges of builtin types and the ownership relationships between object instances. Checking the validity of builtin types is done by first assigning the value to an object and then checking if the typecast to the builtin type succeeds. If the typecast succeeds, the value is valid for the type and we continue execution. If it does not succeed, the programmer is notified of the possibly invalid counterexample *or* a possibly incorrect encoding of the Spec# program into the program verifier. To check the axioms we construct an assertion in the constructed program for each axiom that we wish to be checked. However, these checks usually are not practical as the verifier rarely is spurious under the background axioms, which have been chosen carefully to be a correct encoding of Spec# and generally do not confuse the SMT solver.

We are able to validate most assumptions in the verification semantics during runtime, but not all of them. Assumptions that are not checked during the execution are those assumptions that make use of unbounded universal quantification, such as the modifies clause of a called method. Our extended runtime check for failing assertions cannot be applied here as the counterexample does not provided the instantiations for the quantifier that are needed to check the assertion. However, we do not view this as a large issue as we have not observed any invalid counterexamples that where caused by the modifies clause specification in Spec# or any other unbounded universal specifications that were assumed.

6.6 Experience

Using our implementation we have applied our approach to debug the various verification failures found in examples from the Spec# tutorial [46], the Spec# test suite (see `http://specsharp.codeplex.com`), and our own test suite[2]. In this section, we outline a systematic procedure that we have found to be effective for using our approach to locate the

[2]Also included in the download of our tool.

cause of verification failures. We also summarize and evaluate our experiences using this procedure.

The main observations of our experiments are: (1) Our approach is helpful for understanding most of the verification failures in the examples. In particular, we were able to effectively and efficiently detect bugs in the implementation as well as incorrect or incomplete specifications. The examples where our approach did not provide any benefit were fairly obvious errors in small methods. For those verification failures, the error message provided by Spec# was sufficient to localize and fix the error. (2) Our set of examples contained very few spurious errors and invalid counterexamples because we took them mostly from the Spec# tutorial and test suite, both of which focus on examples that are handled well by the verifier. Nevertheless, our runtime checks identified all of the spurious errors and invalid counterexamples. (3) Most verification failures can be debugged systematically with a simple procedure, which we outline below.

These initial results are very promising. However, our evaluation may be biased in two ways. First, the examples were written for Spec# demonstrations and might not be representative of real application code. Second, the evaluation was performed by people who are familiar with Spec#'s program verifier; it is possible that programmers might struggle with issues that are obvious to us. Nevertheless, we are confident that our positive experience will be confirmed by programmers working on application code.

Debugging Procedure. We have found the following steps to be an efficient way to localize and understand the cause of a verification failure. If the verifier reports several errors for the same method, we debug them in the order of their source location.

1. Use the error message to check the method for obvious errors. For very simple programs and specifications our approach usually requires more effort than simply inspecting the failing method. This is often the case for programs that contain neither method calls nor loops, which reduces the likelihood that the verification failure is caused by an incorrect or incomplete specification.

2. Run the rewritten program in the debugger and observe the failure. Before attempting to localize the error, one should first confirm that the verifier has found a valid error by running the rewritten program in the debugger. This run will either result in an assertion violation (confirming

the validity of the error), in a failed assumption check (indicating an invalid counterexample), or in a message that suggests that the error is spurious. In the latter two cases, the programmer needs to find an alternative way of expressing the program or its specification and re-verify the program. In the former case, the debugging procedure continues with the next step.

3. Inspect the state in which the assertion failed. The runtime check for an assertion fails either because the assertion is incorrect or because the assertion was checked in a state the programmer did not expect. We recommend to inspect the assertion and the state in which the runtime check failed to determine which case applies. If the assertion is incorrect, we can fix it and re-verify the method. If the state contains unexpected values, we determine their origin in the next step.

4. Step through the rewritten program and observe changes to the relevant variables. From step 3, we know which assertion fails. It is helpful to track the values of the variables in this assertion to detect unexpected values, for instance, caused by a weak precondition or loop invariant. This tracking is best performed by adding these variables to the variable watch window of the debugger and then single-stepping through the rewritten method. Unexpected initial values point us to a weak precondition; unexpected modifications during a single step require further investigation, described in step 5. Single-stepping through the method is likely also to reveal errors in the code such as incorrect control flow or the absence of a necessary assignment.

A variation of step 4 is more efficient when the failing assertion contains only a small number of variables, such as the runtime check for a modifies clause which focuses on only one heap location (see Section 6.4). In this case, one can avoid the single-stepping and instead add data breakpoints for the relevant variables. We can then run the rewritten method in the debugger and get notified whenever a variable of interest gets updated.

5. Analyze unexpected modifications. Step 4 determines where a variable receives an unexpected value. If this happens during a method call or in a loop, we have identified the method's specification or the loop invariant as the cause of the unexpected value and can amend them. If the unexpected value comes from an assignment then we may also need to track the variables in the right-hand side expression by adding them

to the watch window and repeating from step 4.

6.7 Summary and Related Work

We have presented our approach to help programmers to understand failed verification attempts. We generate an executable program that reproduces the verification error by encoding the verification semantics of the program and by using variable values from a counterexample. We extend the runtime assertion checker to reproduce all relevant verification errors, identify spurious errors, and detect invalid counterexamples. Executing the generated program inside a debugger allows the programmer to systematically and efficiently explore the counterexample; which is crucial for understanding, and fixing the verification failure. Our implementation is entirely automatic and transparent to the programmer.

We have implemented our approach in Spec#, but it is applicable to all program verifiers based on automatic provers that provide counterexamples. Our experience using our approach is very promising; we are able to understand and fix verification errors effectively and efficiently. We have also found that our approach is useful to debug the encoding of Spec# in the program verifier. By checking the validity of the constructed states we are able to ensure that the values of the variables conform to the declared type in the encoding. We have indeed found an error in the Spec# verifier; when inspecting a counterexample in our tool, we noticed that a variable of type `uint` contained a negative value, which pointed us to an omission in the encoding of Spec# programs.

In the previous chapter we discussed how we can use dynamic program slicing to help the programmer localize the error, however as the counterexample trace may be spurious or invalid; a dynamic program slice may mislead the programmer by removing commands that are required for the failure. Executing our rewritten program and validating the failure, step 2 of our debugging procedure, also validates the counterexample trace given by the program verifier. Once we know that the trace is indeed valid, then performing dynamic slicing on the program with the validated trace will produce a valid program slice that will not mislead the programmer. Extending our approach to include dynamic slicing would help the programmer with step 4 of our debugging procedure.

As mentioned in the introduction, some verifiers such as Spec# apply heuristics to extract those parts of a counterexample that are likely to be relevant for the verification error. However, it is difficult to tune the heuristics such that they provide all necessary information without swamping the programmer with irrelevant details. For instance, Spec# filters too aggressively for the introductory method `AddSorted` and it provides only the following excerpt from the counterexample: `(initial value of: value) == -3`. It is unclear how useful this is to the programmer understanding the verification failure in our introductory example.

Another approach is to construct a test case from a failed verification attempt, using the initial state of the counterexample as test input [16, 11, 63]. This approach is only helpful if the test leads to a runtime error or if the violated specification can be found by a runtime assertion checker. However, when static verification fails because of incomplete specifications, or when the violated specification is not checked at runtime, for instance, when the specification contains unbounded quantification over objects, or when the error is spurious, the test case will succeed. A successful test for a verification failure will not help the programmer understand the cause of the verification error and might even mislead the programmer into believing that the error does not exist [16]. Constructing a test case for our introductory program `AddSorted` with the initial state from the counterexample will succeed as it is not the implementation of `list.Sort` that causes the verification error, it is the weak postcondition of `list.Sort`. Successful test cases are inconclusive about the presence and cause of verification errors.

Verification techniques based on symbolic execution assist the programmer in understanding failed verification attempts by presenting the programmer with the symbolic states used during the verification process [33, 34]. Inspecting a symbolic state would be very helpful to a verification expert who is familiar with the symbolic representation of the program, however it is unclear how helpful it would be to a programmer as this approach exposes the programmer to the particular encoding of the program in the verifier. Our approach appears to be more appropriate for programmer as we offer the programmer concrete states of the program that they can inspect with the program debugger. Moreover, using the symbolic states of the program will not help the programmer in detecting

spurious errors as checking the validity of an assertion is ultimately left up to the verifier.

Alternative techniques based on visualizing the counterexample, such as those based on graph visualization [58, 71], are limited by the size of the state presented and do not help the programmer to identify spurious errors and invalid counterexamples, thus possibly misleading the programmer.

Chapter 7

Conclusions

In this thesis we have presented our approaches to help the programmer localize and understand the cause of verification failures. The first part of the thesis presented our approach to helping the programmer localize the cause of the error by removing those commands, partial or total, that do not play a role in the behavior of the failing specification. By presenting a smaller program to the programmer, it is generally understood that the programmer will be able to locate and fix the cause of the error more efficiently than with the larger, un-sliced program.

If the program slice is insufficient in helping the programmer understand the cause of the verification failure, we take the counterexample given by the program verifier and construct an executable program that simulates the verification semantics of the program with the states given by the counterexample. This constructed program attaches itself to a program debugger and enables the programmer to control the execution and observe the states of the counterexample. By extending the runtime checker of the executable program, the programmer can validate failed verification attempts by executing the failing specification. With the executable program, we have derived a debugging procedure we have found to be useful in locating the cause of the failed verification attempt.

Program slicing in the presence of heap-like data structures suffers greatly from the inability of analysis to determine if an assignment to an aliased variable effects the value of any variables in the criterion. Using

the counterexample given by the program verifier for a failed verification attempt may produce a finer program slice, but may also remove the command that is the cause of the verification failure as the error may be spurious. Determining if the error is not spurious or not requires us to construct a program that simulates the verification semantics and reproduces the states of the counterexample and executing this program. However, it is unclear how much of an issue this is in real practice. One could envision setting up an experiment where we check various failing programs, determine if the error is spurious or not, and compare the dynamic program slices to determine how much of an issue this really is.

Another possibility to handle the overly large program slices would be to investigate the interaction between a program slicer and the verification methodology used to verify the program. For example, in Spec# we have a ownership system that ensures a hierarchy of objects in the system. If we know, for example, that the variables of the criterion are not in the same ownership tree as the variable being assigned to by an assignment, then we can safely remove this assignment as we know that it will never share an alias with any of the variables in the criterion. It would also be worth checking the interaction between an approach such as dynamic frames [42] or separation logic [60] with program slicing. If we know that the memory locations of the assignment are not in the same frame or heaplet as the variables in the criterion, then we can remove the assignment as we know it cannot have an effect on these variables.

In the second part of the thesis we have presented our approach to understanding failed verification attempts. However, we have not made any guarantees as to the correctness of the constructed program. That is, we have not formalized that the constructed program correctly simulates the verification semantics or that we have correctly constructed the counterexample states, modulo the data abstractions, of the verification semantics. Without such a formalization, we cannot ensure to the programmer that the resulting execution trace will validate the error or the states of the counterexample. Such a formalization would be straightforward and should not require any deep insights to show the correctness of the approach.

We have made the assumption that using the program debugger to understand the failed verification attempt is easier and more intuitive than not using the program debugger for understanding verification failures.

This assumption is based on the intuition we have about how programmers think. It would be very interesting to perform a study on actual programmers to qualify this assumption. Such a study would have to take two groups of programmers who know how to specify programs, not necessarily verification experts, and have them perform a series of programming and specification exercises. One group would use the Spec# program verifier without any extra support while the other group would have access to using our tool. The measurement we would be interested in is the average time from the verification failure reported to the programmer to the time it takes for the programmer to locate and fix the cause of the verification error.

Bibliography

[1] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *PLDI*, PLDI '90, pages 246–256, New York, NY, USA, 1990. ACM.

[2] Bowen L. Alpern, Mark N. Wegman, and Frank K. Zadeck. Detecting equality of values in programs. In ACM, editor, *POPL*, pages 1–11, 1988.

[3] Torben Amtoft. Correctness of practical slicing for modern program structures. Technical Report CIS TR 2007-3, Kansas State University, 2007.

[4] Ralph-Johan Back and Joakim von Wright. *Refinement calculus: A systematic introduction.* Graduate Texts in Computer Science. Springer-Verlag, New York, 1998.

[5] Thomas Ball, Mayur Naik, and Sriram K. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In *POPL*, pages 97–105. ACM, 2003.

[6] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, September 2006.

[7] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3:2004, 2004.

[8] Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *PASTE*, pages 82–87. ACM, 2005.

[9] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS*, volume 3362 of *LNCS*, pages 49–69. Springer, 2004.

[10] Nick Benton. Simple relational correcntess proofs for static analyses and program transformations. In *POPL*. ACM, January 2004.

[11] Dirk Beyer, Adam. J. Chlipala, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumbar. Generating tests from counterexamples. In *ICSE*, pages 326–335. IEEE, 2004.

[12] Robert Cartwright and Matthias Felleisen. The semantics of program dependence. In *PLDI*, pages 13–27. ACM, 1989.

[13] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *POPL*, pages 232–245, New York, NY, USA, 1993. ACM.

[14] In Sang Chung, Wan Kwon Lee, Gwang Sik Yoon, and Yong Rae Kwon. Program slicing based on specification. In *Proceedings of the 2001 ACM Symposium on Applied Computing*, pages 605 – 609. ACM, 2001.

[15] Joseph J. Comuzzi and Johnson M. Hart. Program slicing using weakest preconditions. In *Proceedings of the 3rd International Symposium of Formal Methods Europe*, volume 1051 of *Lecture Notes in Computer Science*, pages 557 – 575. Springer, 1996.

[16] Christoph Csallner and Yannis Smaragdakis. Check 'n' Crash: Combining static checking and testing. In *ICSE*, pages 422–431. ACM, 2005.

[17] Sebastian Danicic, Mark Herman, John Howroyd, and Lahcen Ouarbya. A non-standard semantics for program slicing and dependence analysis. *Journal of Logic and Algebraic Programming*, 72(2):191–206, July – August 2007.

[18] Ádám Darvas and K. Rustan M. Leino. Practical reasoning about invocations and implementations of pure methods. In *FASE*, volume 4422 of *LNCS*, pages 336–351. Springer, 2007.

[19] Ádám Darvas and Peter Müller. Reasoning about method calls in interface specifications. *Journal of Object Technology*, 5(5):59–85, June 2006.

[20] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[21] Robert DeLine and K. Rustan M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, March, 2005.

[22] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Laboratories, Palo Alto, 2003.

[23] Edsgar W. Dijkstra. Guarded commands, non-determinancy and formal derivation of programs. *Communications of the ACM*, 18(8):453 – 457, 1975.

[24] Edsgar W. Dijkstra. *A Discipline of Programming.* Prentice Hall, October 1976.

[25] Edsger Dijkstra. The everywhere operator once more, 1977.

[26] Edsger W. Dijkstra. A simple axiomatic basis for programming language constructs. circulated privately, May 1973.

[27] Private discussion with Uday Khedker.

[28] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9:319–349, July 1987.

[29] Steve Freeman, Nat Pryce, Tim Mackinnon, and Joe Walnes. Mock roles, not objects. In *OOPSLA*, pages 236–246. ACM, 2004.

[30] Kurt Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme. *Monatshefte für Mathematik und Physik*, 38(1):173–198, 1931.

[31] Alex Groce and Willem Visser. What went wrong: Explaining counterexamples. In *In SPIN Workshop on Model Checking of Software*, pages 121–135, 2003.

[32] Alex Groce and Willem Wisser. What went wrong: Explaining counterexamples. Technical Report 02-08, RIACS, 2002.

[33] Reiner Hähnle, Marcus Baum, Richard Bubel, and Marcel Rothe. A visual interactive debugger based on symbolic execution. In *ASE*, pages 143–146. ACM, 2010.

[34] Robert J. Hall and Andrea Zisman. Validating personal requirements by assisted symbolic behavior browsing. In *ASE*, pages 56–66. IEEE, 2004.

[35] Mark Harman, Rob Hierons, Chris Fox, Sebastian Danicic, and John Howroyd. Pre/post conditioned slicing. In *Proceedings of the 17th International Conference on Software Maintence*, pages 138 – 147, 2001.

[36] Philip A. Hausler. Denotational program slicing. In *Proceeedings of the Twenty-Second Annual Hawaii International Conference on System Sciences*, volume 2, pages 486 – 494, January 1989.

[37] Michael Hind. Pointer analysis: haven't we solved this problem yet? In *PASTE*, pages 54–61, New York, NY, USA, 2001. ACM.

[38] C. A. R. Hoare. Some properties of predicate transformers. *J. ACM*, 25:461–480, July 1978.

[39] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating noninterfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11:345–387, 1989.

[40] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *SIGPLAN Not.*, 39:229–243, April 2004.

[41] Ranjit Jhala and Rupak Majumdar. Path slicing. In *PLDI*, volume 40, pages 35 – 47, June 2005.

[42] Ioannis T. Kassios. The dynamic frames theory. *Formal Aspects of Computing*, 23(3):267–289, 2011.

[43] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David R. Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, and Daniel M. Zimmerman. JML Reference Manual. Available from `http://www.jmlspecs.org`, September 2009.

[44] K. Rustan M. Leino. This is Boogie 2. Manuscript KRML 178, working draft 24 June 2008, 2008.

[45] K. Rustan M. Leino, Todd Millstein, and James B. Saxe. Generating error traces from verification-condition counterexamples. *Sci. Comput. Program.*, 55:209–226, March 2005.

[46] K. Rustan M. Leino and Peter Müller. Using the Spec# language, methodology, and tools to write bug-free programs. In *Advanced Lectures on Software Engineering*, volume 6029 of *LNCS*, pages 91–139. Springer, 2010.

[47] K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking Java programs via guarded commands. In *Formal Techniques for Java Programs*, 1999.

[48] Hareton K.N. Leung and Hassan K. Reghbati. Comments on program slicing. *IEEE Transactions on Software Engineering*, 13(12):1370 – 1371, December 1987.

[49] Francesco Logozzo and Manuel Fähndrich. On the relative completeness of bytecode analysis versus source code analysis. In *Proceedings of CC '08, LNCS*, 2008.

[50] James R. Lyle and David Binkley. Program slicing in the presence of pointers (extended abstract).

[51] Tim Mackinnon, Steve Freeman, and Philip Craig. Endo-testing: Unit testing with mock objects. In *Extreme Programming Explained*, pages 287–302. Addison-Wesley, 2000.

[52] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.

[53] Carroll Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3):403 – 419, July 1988.

[54] Carroll Morgan. *Programming from Specifications*. Computer Science. Prentice Hall, 1990.

[55] Peter Müller and Joseph N. Ruskiewicz. Using debuggers to understand failed verification attempts. In M. Butler and W. Schulte, editors, *Formal Methods (FM)*, volume 6664 of *Lecture Notes in Computer Science*, pages 73–87. Springer-Verlag, 2011.

[56] Greg Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517 – 561, October 1989.

[57] Tomohiro Oda and Keijiro Araki. Specification slicing in formal methods of software development. In *Proceedings of the 17th Annual Computer Software and Applications Conference*, pages 313 – 319, November 1993.

[58] Derek Rayside, Felix Sheng-Ho Chang, Greg Dennis, Robert Seater, and Daniel Jackson. Automatic visualization of relational logic models. *ECEASST*, 7, 2007.

[59] Thomas Reps and Genevieve Rosay. Precise interprocedural chopping. *SIGSOFT Softw. Eng. Notes*, 20:41–52, October 1995.

[60] John Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.

[61] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of POPL*, pages 12–27, 1988.

[62] Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL*, pages 32–41, New York, NY, USA, 1996. ACM.

[63] Nikolai Tillman and Wolfram Schulte. Mock-object generation with behavior. In *ASE*, pages 365–368. IEEE, 2006.

[64] Nikolai Tillmann and Jonathan de Halleux. Pex - white box testing generation for .NET. In *TAP*, volume 4966 of *LNCS*, pages 134–153. Springer, 2008.

[65] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3), 1995.

[66] Martin Ward and Hussein Zedan. Slicing as a program transformation. *ACM Transactions on Program Languages and Systems*, 29(2), April 2007.

[67] Mark Weiser. *Program Slices: Foraml Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, University of Maryland, 1979.

[68] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439 – 449. IEEE Press, 1981.

[69] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.

[70] Fangjun Wu. Generalized program slicing applied to z specifications. In *Computer Engineering and Technology, 2009. ICCET '09. International Conference on*, volume 1, pages 338 –342, jan. 2009.

[71] Andreas Zeller and Dorothea Lütkehaus. DDD—a free graphical front-end for UNIX debuggers. *SIGPLAN Notices*, 31(1):22–27, 1996.