REVISITING PROGRAM SLICING WITH OWNERSHIP-BASED INFORMATION FLOW

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Will Crichton
August 2022

Abstract

Program slicing is the technique of automatically identifying the subset of a program that is relevant to a particular piece of code. In theory, slicing addresses one of the core challenges of modern-day programming: sorting through large quantities of information to find what is relevant to the task at hand. However, very little is known about how people would actually use a slicer while programming. This dearth of evidence is compounded by the fact that no practical program slicer exists for programmers to use today.

This dissertation contributes to the theoretical and practical foundations of program slicing in three ways. First, I provide evidence for the cognitive basis of slicing. I report on a series of experiments that demonstrate how a person's working memory significantly limits their ability to remember information about a program while engaging in a variety of comprehension tasks. These experiments support the design of tools to reduce working memory load while programming, such as program slicing.

Second, I describe the theory and implementation of a modular program slicer for the Rust programming language. This slicer is based on a novel information flow analysis, Flowistry, that leverages Rust's type system, namely ownership types, to approximate the behavior of unknown code solely from its static type. With these approximations, code can be analyzed more efficiently (no whole-program analysis) and robustly (no library source code needed). I show that this approximation is provably sound and reasonably precise in practice.

Finally, I describe the design of a new program slicing interface, Focus Mode, that interactively visualizes slices as the user changes their focus in a program. I report on a user study of Rust developers debugging programs with Focus Mode.

We find that slices do help programmers find relevant code, for example by focusing their attention on code with unexpected side effects. However, slices may also exclude code which is cognitively relevant to understanding a bug, suggesting the need for further work on the design of program analyses and user interfaces for code relevance.

Flowistry and Focus Mode are both free and open-source tools that have been downloaded by thousands of Rust developers. The tools are publicly available at https://github.com/willcrichton/flowistry

Acknowledgments

Academia is a world marked by dichotomies: do you work on systems or theory? Humans or algorithms? Logic or statistics? But my answer to all these questions is simply: "yes!" So finding my own path through the Ph.D. has not been easy. I am immensely grateful to the many people who guided me along the way.

As an undergraduate at Carnegie Mellon in 2015, I knocked on the office door of Kayvon Fatahalian to ask about doing systems research. Little did I know that as I crossed the threshold into his office, I was entering the *Pat Hanrahan Cinematic Universe*. Kayvon is Pat's former student, as is my co-advisor Maneesh Agrawala. My academic career has therefore been deeply shaped by Pat's way of thinking from the very beginning.

Kayvon showed me how to think like a systems researcher: to see the world in goals, constraints, trade-offs, and design decisions, as well as to pinpoint and articulate the most important parts of a complex system. Maneesh showed me how to think like a human factors researcher: to relentlessly prioritize the user's task over the system's mechanisms, as well as how to turn models of human behavior into design insights. I am grateful to both Kayvon and Maneesh for investing inordinate amounts of their valuable time into helping me develop as a researcher.

I find it harder to give a pithy description for Pat, since he operates on a conceptual plane that's inconceivable to his merely mortal advisees. But I think his guidance can be best summarized as showing me the nature of true wisdom. At so many points in my Ph.D., Pat would simply know how to nudge me in the right direction. "I think adding this to the paper will be the extra bit to help it get accepted" (it gets accepted). "Everyone seemed really excited about this new work, you should consider

focusing on it" (I change research fields and finish the Ph.D.). I am both eternally grateful for this advice, and I hope one day to emulate a fraction of Pat's wisdom.

About halfway through the Ph.D., I completely changed fields from graphics & systems & machine learning into programming languages & human-computer interaction. Shortly thereafter, the world was struck by a global pandemic, which did not ease the transition. I am therefore especially thankful to the many people who helped me ramp up in a new field. Anna Zeng, Georgia Sampaio, and Connor Fogary were stellar collaborators on the work that presaged my dissertation. Shriram Krishnamurthi invited me to many stimulating discussions about PL/CSEd, and he also hosted SNAPL 2019 which cemented my change in research direction. Marco Patrignani helped me refine the formal aspects of the information flow analysis into a publishable form. Sarah Chasins, Elena Glassman, and Josh Sunshine organized the PLATEAU workshop which became a yearly pilgrimage for me to meet other people interested in PL/HCI. Along the way, I have received many words of encouragement from graduate students in PL through summer schools and Twitter.

All of the inhabitants of Gates 3B have enlivened the day-to-day of my Ph.D., especially the folks in Kayvon's, Maneesh's, and Pat's labs (I have been to *many* lab meetings). I am indebted to the administrators who helped me graduate despite my pathological inability to file paperwork on time: Jay Subramanian, Helen Buendicho, Yari Lara, Andrea Kudik, Nan Hwa Aoki, and Kaila Jimenez.

My Ph.D. would not have been possible with the love and support of my wife, Maryyann, and our respective families. My parents supported my decision to go to grad school, even though getting a job at Jane Street would have been more financially sound. Maryyann stuck with me even through three years of long-distance throughout her MFA. Our wedding was truly the highlight of my Ph.D. (even better than a paper acceptance!). This dissertation is dedicated to her, and to many more years of research and art together.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Programming is a challenging activity. Programmers today have to learn dozens of arcane tools, reason about the behavior of complex systems, and work within sprawling teams where no one understands their entire codebase. As a result, studies of programmers show their working hours are largely spent reading or navigating through code, not writing or maintaining code [1, 2]. The quality of the code also suffers — software bugs cost an estimated $2 trillion each year in the US alone [3].

However, collectively improving the process of programming is no small feat due to the inherent variability of the domain. For example, does a front-end software engineer writing webapps in Typescript share anything in common with an embedded systems developer writing kernels in C? If the answer is no, then progress is doomed to discovering piecemeal solutions that must be reinvented for every new problem domain. But the human process of programming seems to have some common ground across domains. Programm*ers* have a shared cognitive architecture that influences how they approach programming problems. Programm*ing* has a core shared vocabulary of concepts like variables and functions.

Therefore my research approach is to combine *cognitive psychology* and *programming language theory* to create tools that simplify programming across a wide variety of tasks and domains. Cognitive psychology helps us understand *what makes tasks hard for people*: having to visually parse a complicated display, having to juggle competing tasks, having to remember too many thing at once. And programming

language theory helps us understand *what properties a program has*: whether a program has undefined behavior, whether a function satisfies its specification, whether one variable depends on another. Together, these fields can provide a principled foundation for the design of practical programming tools.

This dissertation represents one instantiation of this idea. It describes my process of combining the cognitive theories of memory and perception with the programming language theories of information flow and ownership types to design a new kind of program slicer that helps developers find code relevant to their task.

1.1 Perception and Memory

The main aspects of cognition that I investigate in relation to programming are *perception* and *memory*. Together, they form the cognitive foundation for how we understand the world. A person perceives the world (or a program) through the five senses, and then they persist information about the world in their memory. The study of perception and memory has influenced several fields of applied cognitive psychology such as industrial design [4], information visualization [5], and human-computer interaction [6] — and so programming may too benefit from this lens.

As a case study on the influence of perception and memory, imagine a person is in their kitchen making dinner by following a recipe. The person needs to determine the next step in the recipe. To understand the state of their recipe, the person can:

- Visually survey which ingredients are on the countertop, or the color of the object being cooked.
- Pick up containers and feel their weight to judge the volume of their contents.
- Listen for indicative sounds like the bubbling of boiling water or sizzling of oil.
- Smell the aroma of food baking out of view in the oven.

In each case, the person can quickly and effortlessly perceive information relevant to the state of the recipe. They do not need to *remember* this information because it is readily perceivable.

Now imagine an alternate scenario where the cook is not actually inside the kitchen, but rather inside a separate room, remotely controlling a cooking robot with a joystick. The cook cannot see, hear, smell, taste, or touch anything in the kitchen. By default, the only feedback they receive is a notification of catastrophic failure, e.g. if the robot sets the kitchen on fire. The cook must then hold all relevant information in their head: a mental image of the kitchen, a mental note of the robot's location and the state of the recipe, and so on.

You can imagine that a person in this nightmarish scenario would be very unlikely to prepare a good meal, and very likely to set the kitchen on fire. The core reason is that human memory is subject to many severe limitations. For example, short-term (or "working") memory has a very small operating capacity and can be easily interfered with distractions or other stimuli [7]. To perfectly remember everything about the robot and the recipe would be nigh impossible.

Yet, this scenario is more or less how a programmer must perform their job! A programmer writes a program, asks the computer to execute it, and waits until either receiving the expected output or notification of failure. In the interim, the state of the computer is wholly invisible to the programmer, unless carefully prodded with a `printf` or inspected in a debugger. Programmers frequently have to "play computer" in their heads to understand why a program executes as it does. We can therefore make two predictions:

1. A core cognitive challenge of programming is its load on working memory (supported by the experiments in Chapter 3).

2. Tools which make visible information that would otherwise be mentally deduced may reduce the cognitive challenge of programming (supported by the user study of our tool in Chapter 6).

1.2 Cognitive Support for Programmers

Today, the main tool designed for providing such assistance to programmers is the *integrated development environment*, or IDE. A modern IDE contains dozens of both

ambient and on-demand visualizations of program information. These visualizations are designed to assist the programmer when reading and writing programs, as shown in Figure 1.1. The features are a form of *cognitive support*, in that they externalize information that the programmer would otherwise need to deduce or remember.

Each IDE feature is implemented as a form of *program analysis* to varying degrees of sophistication:

- At one end are purely textual features such as the minimap (Figure 1.1-6), which require no understanding of the program at hand. These features can be applied to any programming language or arbitrary text file.
- In the middle are features that rely on knowledge of syntactic structure such as syntax highlighting (Figure 1.1-2) and module location (Figure 1.1-1). These features work for most programming languages.
- At the far end are features such as autocomplete (Figure 1.1-5), inlay type hints (Figure 1.1-3), and mutability indicators (Figure 1.1-7). These features require a language to have a static type system with e.g. type inference and mutability modifiers.

However, even the most semantic of these program analyses still only provides low-level information about the program's behavior. For instance, the fact that `snippet` is of type `String` does not indicate the role of that variable in the program, or how `snippet` is defined, and so on. It is therefore an open problem to design IDE features that support high-level cognitive tasks in programming, while still being generally applicable to a large class of programs.

1.3 Relevance and Program Slicing

One such high-level concern is *relevance*. Any sufficiently large codebase weaves together a number of features, concerns, modules, classes, data sources, and so on. But for a given programming task, only a small subset of the total code in the codebase

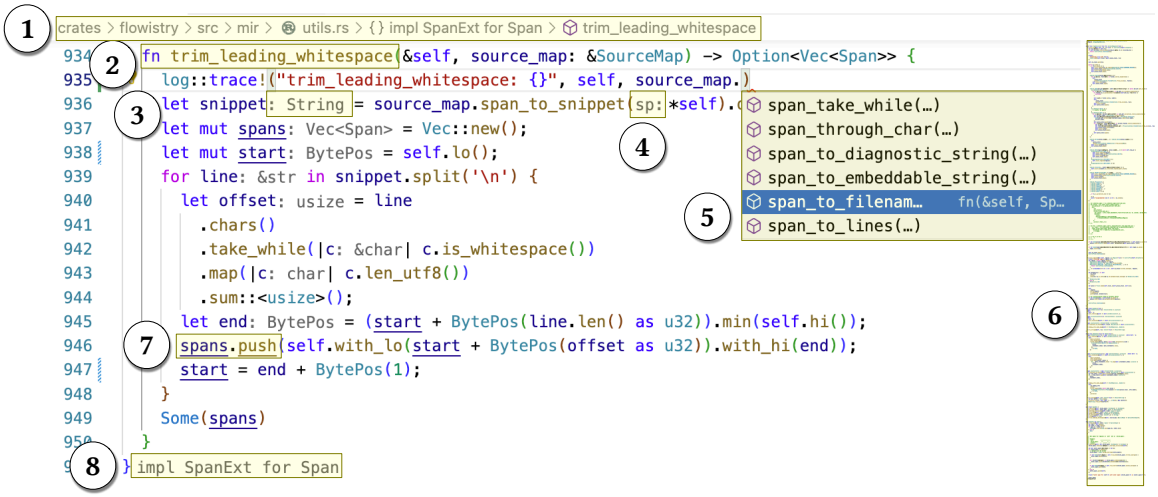

Figure 1.1: Examples of cognitive support through visualization provided by a modern IDE, specifically Visual Studio Code with the Rust Analyzer extension. Each example is highlighted in yellow and described by the corresponding item below.

1. Breadcrumbs show the position of the current function in the codebase. This helps the user understand how local changes are positioned in a global context.

2. Syntax highlighting uses color to indicate the syntactic role of a piece of code. This helps the user discern the role of a piece of code at a glance [8].

3. The inferred type of a variable is inserted as a "ghost" annotation. This prevents the user from needing to infer the type themselves.

4. The names of positional arguments are visualized in a function call. This means the user does not need to remember the order of arguments to the function.

5. When writing code to call a method, an autocomplete box shows the set of possible methods based on the receiver's type. This means the user does not need to exit the editor, find the type's documentation, read it, and return back.

6. A minimap shows a zoomed-out version of the current file's contents. This helps the user editing a large file remember where they are.

7. Objects that are mutable and methods that mutate are underlined. This helps the user quickly discern possible locations where side effects occur.

8. A closing brace is annotated with the object it closes. This helps a programmer remember which syntactic scope they are in.

is likely to be relevant. Therefore a key cognitive task of programming is identifying relevant code.

However, programmers are not always effective at distinguishing the relevant from the irrelevant. In one study of Java developers fixing a bug in a GUI library, Ko et al. [9] found that programmers spent 36% of their time inspecting irrelevant code. Conversely, programmers may ignore relevant code, such as by not recognizing hidden dependencies [10]. Therefore programmers may benefit from IDEs providing cognitive support targeted at identifying relevant code.

A mechanism that has long been hypothesized to assist programmers with finding relevant code is *program slicing*. The (backward) program slice of a given variable is the set of code that can influence the value of that variable, and a program slicer is an algorithm for computing program slices. For example, if a programmer is debugging an assertion failure like `assert(*x == 0)`, then the set of code relevant to the bug is (in theory) the backward slice of `*x`. A program slicing tool could visualize which code is inside the slice, thereby reducing the size of the programmer's working set and theoretically lowering the cognitive load of the task at hand.

It was this cognitive connection that inspired me to pursue the study of program slicing — can slicing actually be practically useful in helping programmers find relevant code? In fact, slicing was originally formulated in the 1980s by Weiser [11], and it was the subject of hundreds of papers in the ensuing years. Yet, practical algorithms for slicing have remained elusive. No program slicer remains in use today. The last two human-centered studies of program slicers took place in 1986 [12] and 2002 [13].

One possible factor in the scarcity of slicers is the inherent difficulty in analyzing the popular languages of the day. In the period from the late 1980s to early 2000s, those languages were primarily C and Java. But these languages are notoriously difficult to analyze. For instance, the backward slice of `*x` depends on what `x` could point to. Pointer analysis itself has been the subject of yet more hundreds of papers! And despite decades of research in static analysis, these tools still fail to catch memory safety issues in widely used C/C++ software. No amount of sophistication will likely be sufficient to practically analyze constructs like pointers in unrestricted languages

like C. Higher-level analyses like slicing may therefore be hopeless!

1.4 Ownership-based Information Flow

Yet, C and Java are no longer the only popular languages in 2022. Advances in static type systems have slowly seeped into the mainstream. New and old languages alike have gained features like generics, algebraic datatypes, type inference, and so on. Modern PLs with sophisticated type systems restrict the space of expressible programs in exchange for expanding the space of statically verifiable program properties.

Concurrently, the PL research community has developed a number of new theoretical frameworks for reasoning about programs. Researchers have designed programming models centered on static analysis of properties like runtime performance [14], communication protocols [15], and more. These models move past the traditional type-safety goal of eliminating undefined behavior toward enabling more deeply semantic analyses.

The theoretical contribution of my dissertation is connecting two advances in these parallel trends to program slicing, namely: ownership types and information flow. Ownership types are a discipline for ensuring memory safety in languages without garbage collection, popularized in Rust. Information flow is the property of whether learning about one variable provides information about another, commonly applied in security research to verify that sensitive data cannot leak publicly.

In short, the relationship between these topics is that program slicing is a special case of information flow analysis, as established by Abadi et al. [16]. I demonstrate in this dissertation that ownership types provide the foundation for a modular static analysis of information flow. Therefore it is practically feasible to compute program slices for a language with ownership types, namely Rust.

The technical contribution of my dissertation is the design and implementation of Flowistry, a Rust static analyzer that puts this theoretical insight into practice. Flowistry integrates with the Visual Studio Code IDE to provide a novel interface called Focus Mode, where a user can interactively view the slice of code under their cursor, as shown in Figure 1.2. The goal of Focus Mode is to externalize program

```
fn trim_leading_whitespace(&self, source_map: &SourceMap) -> Option<Vec<Span>> {
  let snippet: String = source_map.span_to_snippet(sp: *self).ok()?;
  let mut spans: Vec<Span> = Vec::new();
  let mut start: BytePos = self.lo();
  for line: &str in snippet.split('\n') {
    let offset: usize = line
      .chars()
      .take_while(|c: &char| c.is_whitespace())
      .map(|c: char| c.len_utf8())
      .sum::<usize>();
    let end: BytePos = (start + BytePos(line.len() as u32)).min(self.hi());
    spans.push(self.with_lo(start + BytePos(offset as u32)).with_hi(end));
    start = end + BytePos(1);
  }
  Some(spans)
}
```

Figure 1.2: An example of the Focus Mode interface within Flowistry. When the user selects a piece of code, like the variable `end`, then all code outside the forward and backward slice of the variable is faded out. For example, neither `spans` or `offset` can influence `end`, and they are therefore faded out.

slices in a manner that is quickly perceivable to the programmer and does not distract from the higher-level task at hand.

1.5 Relevance ≠ Information Flow

Since the inception of program slicing, papers on the subject have routinely referred to a slice as the set of code "relevant" to a particular variable. But it is not inherently clear whether relevance viewed as a *logical* property of a program (i.e. information flow) is the same as relevance viewed as a *cognitive* property of a program (whether reading a piece of code is helpful to the programmer in accomplishing their task).

Therefore the final experimental contribution of my dissertation is a formative user study exploring whether program slicing (as implemented in Focus Mode) actually helps programmers find relevant code. I recruited 18 participants familiar with Rust and asked them to debug a series of short Rust programs both with and without Focus Mode. I qualitatively coded my observations from each participant's session,

and searched for patterns of observations within each condition.

Consistent with Green's theory of hidden dependencies in the Cognitive Dimensions of Notation [17], I found Focus Mode helpful in surfacing unexpected dependencies between code, often induced by tricky side-effects in imperative-style programs. Focus Mode also helped de-emphasize code irrelevant to the task in some situations.

However, another key finding is that in some tasks, code outside the slice of a variable was relevant to understanding a bug involving that variable. For instance, say a variable was defined as `y = f(x)` and the bug is that the code should say `y = f(z)`. Then `z` may be outside the slice of `y`, but it should not be! A slicer like Focus Mode can therefore mislead a programmer into ignoring relevant code outside a given slice. This finding suggests that the longstanding definition of slices as "relevant code" does not match reality.

1.6 Dissertation Roadmap

The remainder of this dissertation is organized as follows:

- Chapter 2 surveys the history of cognitive psychology as applied to programming, with a focus on working memory.
- Chapter 3 describes a set of experiments about how working memory influences a person's ability to remember state about a program.
- Chapter 4 reviews the history of program slicing and the evidence for how the technique relates to cognition, as well as the connection to recent advances in type systems and information flow analysis.
- Chapter 5 describes a technique for the modular static analysis of information flow that leverages ownership types, providing a proof of theoretical soundness and experiments supporting its empirical precision.
- Chapter 6 reports on a user study of an IDE-integrated Rust program slicer

built on the technique in Chapter 5, finding that the slicer helps programmers find relevant code in some situations.

- Chapter 7 concludes the dissertation and suggests some directions for future work.

Chapter 2

Cognition and Programming

Every programming system must, at some point, come into contact with a person. That person needs to understand and manipulate the system — both cognitive tasks. In fact, research from across the spectrum of computer science will often cite cognition as an explicit motivation or design principle for systems. For example, in programming languages (emphasis added):

> "Silq is the first quantum language to provide intuitive semantics." — Bichsel et al. [18]

And high-performance computing:

> "This paper explores a system architecture designed to make it easy for users to reason about performance bottlenecks." — Ousterhout et al. [19]

And distributed systems:

> "We believe that Raft [...] is simpler and more understandable than other algorithms." — Ongaro and Ousterhout [20]

That is to say, in each of these systems (and many others) the authors were principally motivated by how their system design related to cognition. Yet despite these good intentions, very few of these papers produced conclusive evidence for their

cognitive claims. Bichsel et al. [18] showed that programs in their language are on average smaller than in a competing language. Ousterhout et al. [19] showed that their tool correctly predicts system performance in a variety of configurations. Neither evaluation involved a human.

Ongaro and Ousterhout [20] is particularly interesting in that their consensus algorithm, Raft, has been widely adopted in industry, arguably due to its simplicity and understandability. Yet, they were only able to show in a user study that participants had a small increase in quiz scores when learning about Raft versus another popular consensus algorithm. So in all of these cases, there was a large gap between a researcher's intuition for how a tool relates to cognition, and the methodologies available to the researcher for demonstrating that relationship.

It was the invariable recurrence of this gap that initially spurred my interest in the topic of cognition and programming. Why is it so hard for computer scientists to back up human-centered claims about their software? Especially in the context of programming language design, many widespread folk theories have never been demonstrated to a significant degree of rigor, such as "static type systems are valuable for productivity in large codebases" or "message-passing concurrency is easier for programmers to reason about than shared-memory concurrency."

Therefore early in my Ph.D., I set out to understand: what do we know about programming as a cognitive process? Why has this knowledge not found widespread use in the design of programming tools? And what would it take to make it practically useful for researchers, devtool builders, and programmers?

2.1 The History of Cognitive Psychology and Programming

The human factors of programming have been relevant as long as humans have been programmers. Many of the most influential ideas in computer science stemmed from human factors on some level. For example:

- Grace Hopper invented compilers because she didn't like programming in octal [21].
- Edsger Dijkstra argued for structured programming based on the cognitive limitations of people to "visualize processes evolving in time" [22].
- Don Knuth invented profiling because his intuition for which compiler optimizations mattered was different than what people actually needed[1] [23].
- Tony Hoare argued that "the primary purpose of a programming language is to help the programmer in the practice of his art" [24].

It was not until the 1970s however that human factors research on programming started to evolve from experience reports and expert intuition into a proper science. Researchers started conducting controlled experiments. They borrowed methodologies and theories from other disciplines. And that tradition has continued: human-centered programming research can be found in the fields of software engineering, computer science education, human-computer interaction, and even sometimes programming language design.

However, a researcher in 2022 looking back on those early experiments in the 1970s and 80s would find a wholly alien world. Researchers then focused much more heavily on cognitive psychology, on perception and memory, and generally on the construction of *theory* as much as *systems*. Therefore to understand this alien perspective, this section will chronicle the rise and fall of cognitive psychology in programming research.

2.1.1 The First Experiments

The earliest attempts to understand programming as a cognitive activity started in the 1970s, often through the lens of adjudicating questions of language design. Researchers investigated topics such as: structured programming vs. GOTO [25, 26, 27],

[1] This paper was, in fact, the first empirical analysis of code "in the wild." Since Knuth didn't have access to GitHub in 1971, he had to resort to methods such as pulling programs out of a literal trash can, and stationing a research assistant by the printer to ask for copies of other peoples' programs.

static vs. dynamic typing [28], the use of flowcharts [29], and the use of indentation [30].

In the 1981 article "The Psychological Study of Programming," Sheil [31] provides an excellent review and scathing critique of this early work, best summarized in this excerpt:

> "Although some psychological theory is very suggestive, it usually lacks the robustness and precision required to yield exact predictions for behavior as complex as programming. As a result, the psychological work on programming consists mainly of atheoretical evaluations motivated directly by the concerns of contemporary computing practice. [...]
>
> [These experiments] are unsatisfactory in that they are methodologically weak, the effects they report are small, and yet they are presented as if they establish claims that go far beyond their data. [...] High individual variances, strong practice effects, and (consequently) weak findings are exactly what one would expect from studying the average performance of highly learned skills across diverse collections of individuals."

That is to say, while there were many experiments about the psychology of programming during this time, it is difficult to conclude what general lessons may be learned. Perhaps the most important takeaway is that programming is simply too complex a cognitive task to approach without any theoretical basis.

2.1.2 Developing Theories of the Mind

In some ways, those early experiments on programmers reflected a *behaviorist* approach to psychological research. Within strict behaviorism, the mind is treated as a black box, and psychological theories are developed by observation of a stimulus/response relationship. Behaviorism was the prevailing methodology of psychology until cognitive psychology became a significant force in the 1970s.

By contrast to behaviorism, a central goal of cognitive psychology is to investigate the idea of mental states — to model the unobservable mind. One such foundational

model is *working memory*, specifically its limitations proposed by George Miller in his famous paper "The Magical Number Seven, Plus or Minus Two" [32]. Miller drew on Claude Shannon's information theory to explain the results of several then-recent psychological experiments in information-theoretic terms.

When viewing human perception as an information receiver, a person can generally receive about 2.5 bits of information in a single stimulus. For instance, if the average person is given ahead of time a set of N tones, then prompted with one of the N tones, their ability to identify which tone they just heard falls off around $N = 6$ (i.e. $2^{2.5}$). The same is true when viewing human memory as information storage. If the average person is read aloud a series of N digits and then asked to repeat them in order (a "digit span" task), they will only remember up to about $N = 7$.

The key difference for memory is that the information capacity is defined in terms of *items*, or more commonly *chunks*. A person can perform equally well in remembering binary digits, decimal digits, or words. However, each kind of item contains a different amount of information (e.g. 1 bit for a binary digit, 3.3 bits for decimal, many bits for a word). Therefore a person can actually remember *unbounded amounts of information* depending on the size of the item in memory! For example, Chase and Ericsson [33] trained a Carnegie Mellon undergraduate for two years (!) to eventually achieve $N = 80$ (!!) on a digit span task by use of mnemonics to cluster the digit sequence.

Since the formulation of Miller's initial theory, cognitive psychologists have shown that expertise in many domains involves efficient use of working memory via careful chunking. Chase and Simon [34] showed that expert chess players could remember a chess board at a glance by encoding high-level game features ("this is the Sicilian Defense but with..."), while novice chess players would struggle to remember the same information. This observation led to a crucial insight: *that expert knowledge could be interrogated through chunking.*

Therefore going into the 1980s, many studies about the nature of programming expertise used chunking to examine cognition. These experiments have roughly the following form:

1. Create an object (e.g. a program) in a domain (e.g. FORTRAN).

2. Have a person study the object so its contents fill their working memory.
3. Remove the object from the person's view.
4. Ask the person to tell you about the the object, i.e. fetching its contents from working memory.
5. Analyze the order of object features in the response. Features that consistently appear close together are considered part of a single chunk.

For instance, here are two such studies from the time:

- Adelson [35] had expert and novice programmers recall lines of code after being shown three functions scrambled together. Novice programmers chunked lines based on syntactic similarity, while expert programmers mentally reconstructed and chunked together lines from each function.
- McKeithen et al. [36] had expert and novice programmers recall language syntax keywords after studying a set of flashcards. Novice programmers chunked keywords based on syntactic similarity (e.g. "SHORT", "STEP", and "STRING"), while expert programmers chunked keywords based on their semantic similarity (e.g. "IF", "THEN", "ELSE").

These initial findings suggested that, indeed, expert programmers did acquire some kind of knowledge that changed how they encoded information about a program in their working memory. Novices remembered programs "syntactically" while experts remembered them "semantically". But the nature of these semantics remained unclear, so researchers sought to develop more nuanced theories of the structure of programming knowledge.

2.1.3 Schemas, Plans, and the Structure of Knowledge

The most popular theory of knowledge in the 1980s was *schemas*, or hierarchical templates that describe relationships between the abstract slots in the template [37]. For example, a face schema would contain an eye, nose, mouth, etc. and prescribe

that a face has two eyes, the eyes are above the mouth, so on. Each component would itself have its own schema. Cognitive psychologists postulated that people constructed schema over time and applied them to understand the world.

One notable application of schema theory is the interpretation of the Wason selection task [38]. In this task, participants are given a logical rule about two-faced cards. For instance, if a card has a letter on one side and a number on the other, then a rule could be "if the letter is a vowel, then the number must be even." Participants are then shown one side of a particular card and asked: do you need to flip the card to know whether the card satisfies the rule? Wason demonstrated that with this kind of logical reasoning task, participants do very poorly, getting the answer correct less than 10% of the time.

However, later research showed that changing the problem setting while preserving the logical structure could significantly impact performance. Griggs and Cox [39] rephrased the problem as: a card has a person's age on one side and a drink order on the other side. The rule is: if a person is under 21, they cannot order a beer. In this setting, performance jumps from 10% to 74%! The schema theory interpretation of this result is that people naturally build a schema for logical deduction in social situations, like for rules around alcohol consumption. While people are generally bad at abstract logical reasoning (see Evans [40]), they are much better at schematic reasoning, and therefore accomplish the beer-themed selection task at astoundingly higher rates.

Several studies have provided evidence for the existence of schemas in programming. For example, Letovsky [41] ran a thinkaloud study while experienced programmers tried to add a feature to a database implemented in FORTRAN. During the experiment, participants would ask questions reflecting their thought process, such as: "Ok, `dbase` is an array: 200, 7. Why seven? Seven fields, I'll bet." Letovsky qualitatively analyzed patterns in the participants' questions.

For example, when reading this piece of code that searches the database for a record:

```
1 If (oldnme .EQ. name) THEN
2   iptr = ioldp
```

```
3   RETURN
4     ELSE
5   CALL srch2(dbase, ifinal, iptr, name)
```

Then participants would frequently understand this code as an instantiation of the generic schema:

```
1 If cheap solution applies
2   THEN use cheap solution
3   ELSE use expensive solution
```

If a participant identified that the two branches had the same ultimate outcome, then this fact would cue them into testing whether this performance-optimization schema explained the code at hand.

Closely related to (and sometimes synonymous with) schema are *plans*. A plan is a hierarchical decomposition of a task into a series of subgoals that can be concretely acted upon. Within programming psychology research, plans usually refer to program templates, program abstractions, or other means of decomposing programming problems.

One popular research direction in the 1980s was an attempt to capture schema/-plan knowledge as a series of formal rules (a "production system") that could computationally emulate the behavior of a human programmer. For instance, Anderson et al. [42] analyzed transcripts of people learning to program Lisp and created a model that could produce programs through similar means, e.g. generating one function by analogy to a similar one. However, the complete set of rules used was never specified and the code never released, so efforts like this are largely lost to history.

Another research direction focused on the role of plans in how students learned basic programming skills. Spohrer et al. [43] demonstrated that a common source of bugs for novice programmers was not in the translation of an individual plan to code, but rather the *composition* of multiple plans — specifically a *merging* composition where code from each plan is interleaved together in the same block. These researchers argued that plans should more generally serve as the basis for a redesigned CS curriculum [44], but this idea does not appear to have ever come to fruition[2].

[2]Recent work within the CS education research community has started to renew interest in plan

More generally, these intersecting lines of research on working memory, chunking, schemas, plans, etc. seem to have dried up by the early 1990s. Many of the authors cited above collaborated on the publication of a excellent survey titled *Psychology of Programming* [48], which I highly recommend to anyone seeking a deeper dive into these topics. But after the book's publication in 1990, the research community largely lost interest in building a cognitive theory of programming.

2.1.4 The End of the Cognitive Era

Why did research on cognition and programming die out? One possible explanation is the contemporaneous backlash to cognitivism within the broader field of human-computer interaction research. This position is best summarized by Landauer [49]:

> "For the most part, useful theory is impossible, because the behavior of human-computer systems is chaotic or worse, highly complex, dependent on many unpredictable variables, or just too hard to understand. Where it is possible, the use of theory will be constrained and modest, because theories will be imprecise, will cover only limited aspects of behavior, will be applicable only to some parts of some systems, and will not necessarily generalize; as a result, they will yield little advantage over empirical methods. [...] Direct empirical models, rules of thumb, and formative evaluation together are a more-than-adequate base for important inventions and advances."

This critique starts with the same premise as Sheil's [31] critique of early programming psychology research: human-computer interaction (programming or otherwise) is wickedly hard to understand due to its complexity. But the conclusion is different. Sheil argued for the construction of theory from first principles, whereas Landauer argued for the wholesale rejection of theory in favor of pure empiricism.

Landauer's critique stems from a simple observation: a cognitivist approach to understanding HCI had not produced much knowledge that could be applied to the

composition, such as: Fisler et al. [45], Duran et al. [46], Cunningham [47]

design of tools for users. HCI theory such as GOMS [6] could precisely model low-level actions like mouse movements and changes in gaze, but could not make predictions about higher-level tasks. The construction of elaborate theory took significantly more effort to produce significantly less insight than just iteratively designing a product with frequent user testing.

Applying this idea to programming, take any individual result, such as McKeithen et al. [36] showing that expert programmers semantically chunk programming language keywords while novice programmers syntactically chunk them. How can this observation inform the practice of programming? It's not clear. This research was performed by cognitive psychologists, whose principle motivation was *understanding people* and not *improving programming*, so the paper itself offers few applications. Later interpretations of this work offer some more practical ideas: Hermans [50] cites McKeithen to say that programmers should strive to write "chunkable" code. She further cites Prechelt et al. [51] to suggest that design patterns improve chunkability. But the connection from the ultimate insight to the original theory is still quite loose — reading code written with a visitor pattern is quite far from recalling language keywords.

In my view, the only major idea that survived from pre-2000s programming/cognition research is the Cognitive Dimensions of Notation. From the early 1970s to late 1980s, several British psychologists had collaborated on studies applying psychology to programming: Thomas Green, Simon Davies, David Gilmore, Max Sime, and others. This prolific group published dozens of papers on programming and cognition (including several of the studies criticized by Sheil [31]). But this body of work largely existed as a set of independent experiments, with no coherent framework to incorporate each result.

Therefore in 1989, Thomas Green published "Cognitive Dimensions of Notation" [10], which coalesced these low-level results into a set of high-level "dimensions" describing common usability problems with information systems. For example, the dimension of "viscosity" describes whether a change to an object affects only other objects close to the change (good, local, viscous) or far away from the change (bad, global, not viscous). The dimension of "premature commitment" describes the extent

to which a system requires users to commit to decisions before they have sufficient information to make that decision. The specific choice of dimensions was not particularly systematic, but likely informed by Green's deep experience working in the various Applied Psychology Units of British universities.

In a 1996 paper titled "Delivering Cognitive Psychology to HCI" [52], Green and others articulated why they believed this style of research could be useful:

> Our contention is that person-system research cannot be achieved merely by collating person-based research with artifact-based research. It is only feasible if a common language can be developed, in which relevant aspects of both the person and the system can be expressed. [...]
>
> The framework of cognitive dimensions is far from complete, but the aim is apparent: to supply a vocabulary of discourse in which non-specialists can find terse descriptions of important aspects, reminders not to overlook other aspects, and some hint of the trade-offs between different aspects; and which can be indexed into specialist research in each field. [This constitutes] a 'delivery' of cognitive psychology by making sure that the vocabulary includes terms for design considerations which have been shown relevant by cognitive psychology.

History has arguably supported Green's claims. Their ideas have explicitly influenced notable research systems like the Whyline debugger [53] and the Protovis visualization toolkit [54]. The Cognitive Dimensions of Notation papers have collectively accrued thousands of citations from across the cognitive and computer sciences. Researchers seem to have found the dimensions both easy enough to incorporate into their design process, but also informative enough to generate useful design insights.

2.1.5 The Field Moves On

Since the turn of the millennium, research on the human factors of programming has dispersed throughout academia. Researchers in HCI, software engineering, and computing education continue to conduct user studies, build prototype programming

systems, and otherwise still pursue the ideal of usable and learnable programming. However, cognitive psychology has largely fallen by the wayside in favor of other human-centered design methods: contextual inquiry, grounded theory, surveys, and so on. For example, the 2016 article "Programmers Are Users Too" [55] by Myers et al. offered ten different methodologies for studying programmers, but none of them involved the use of theory (cognitive or otherwise).

Arguably the closest analogy today to the 1970s-90s programming/psychology research is the recent push towards neurological investigations of programming. Siegmund et al. [56] explain the motivations for neuroimaging in their 2020 article:

> "Research in program comprehension has been a cycle of booms and busts. In the early 1970s and 1980s, the first wave of researchers were psychologists, using methods, such as memory recall, to probe how programmers represent and process code in their mind. As a result, various theories and mechanisms were proposed, such as programming plans and bottom-up comprehension, but no clear consensus emerged. [...]
>
> Programming research has entered the Neuroage. Neuroimaging offers a unique opportunity to understand, build, and test theories of program comprehension like never before. [...] For example, rather than stating that programmers tend to process loops faster than recursive structures, we would like to quantify how and to what extent the use of loops or recursion influences task completion time and which cognitive processes are responsible for this difference."

Neurological research on programming has thus far taken the following form. fMRI data provides coarse information about whether a particular location in the brain is active during a task. Neuroscientists have done hundreds of studies to associate each brain location with a high-level cognitive task, like language processing. Then programming researchers will have participants read or write programs within an fMRI machine and correlate brain activations with previous results.

For example, Peitek et al. [57] used fMRI while participants traced short programs, finding that "during bottom-up program comprehension, Brodmann areas 6, 21, 40,

44, and 47 are activated." In discussion, they say that "[Brodmann areas] 21, 44, and 47 are related to different facets of language processing," leading them to conclude: "our results [...] imply that, during learning programming, training [...] language skills might also be essential for programming skills."

However, time will tell whether neurological methods provide enough information for researchers to construct robust and useful theories of cognition in programming. For instance, CS education researchers have struggled to effectively transfer even well-established theories like cognitive load theory [58, 59]. Creating design-relevant theory has always been a challenging aspect of importing psychological methods into computer science. If neuroimaging should serve as the foundation of a new theory of how people program, then more work needs to be done in linking the theoretical results to the efficacy of its proposed interventions.

2.2 A Renewed Focus on Working Memory

Overall, programming/psychology research has struggled to find a balance between being precise, correct, general, and useful. The Cognitive Dimensions of Notation represent the best attempt yet to strike that balance, but I believe they still only embody a fraction of the potential benefit that cognitive psychology can bring to programming.

Previous research focused on using cognitive psychology to understand expertise, e.g. using recall to investigate chunking which distinguishes novice from expert knowledge. In that style of research, cognitive resources like working memory are noisy channels through which expertise can be distilled. However, an alternative style of research is to focus on how the limitations of these cognitive resources affect human performance on cognitive tasks.

For example, research on information visualization has shown how the nature of perception can guide the design of visualizations. Say a person wants to visualize a categorical variable against a quantitative variable. They want to make it easy to compare two quantities, and they are considering either a bar chart or a pie chart. Vision science can tell us which kinds of visual objects people can compare quickly

(i.e. preattentively). Namely, aligned bars are easy to compare for heights, while pie slices are more difficult to compare for sizes, and therefore the bar chart more cognitively efficient to a sighted reader than a pie chart. Any modern infovis text such as *Information Visualization: Perception for Design* [5] is replete with such design principles that are deeply informed by cognitive psychology.

Akin to how perceptual limitations influence information visualizations, one would similarly expect that working memory limitations influence programming (as described in Section 1.1). However, there is surprisingly little research on this subject, at least within computer science proper. Therefore I set out to investigate: can we find clear evidence demonstrating the influence of working memory on programming?

2.2.1 Program Tracing

A challenge in psychological research on programming is selecting the right task to study. "Programming" is easily too broad. "Debugging" and "comprehension" are complex — they can span hours, days, or even weeks. A cognitive psychologist once remarked to me that the ideal task should take five seconds or less! I was put off by this remark for a long time, but I have since come to see the wisdom in it.

The simplest programming task I could imagine (that would still be useful to study) was *program tracing*. Program tracing is the task of mentally simulating a program on concrete inputs to produce a concrete output. To trace a program, a person follows a procedure step-by-step as specified in the source code. Go here, add this, store that, repeat — a literal human information processor. Program tracing encompasses questions such as: for some function f what is the concrete value of $f(1)$? By contrast, program *comprehension* involves tasks about abstract relationships, e.g. what is the symbolic value of $f(x)$ in terms of any $x \in \mathbb{N}$?

Tracing may seem like an absurd task — the entire point of a computer is to *automatically* trace programs, and have the human operate at a higher level of abstraction. However, cognitive psychologists have repeatedly demonstrated the close relationship between action and understanding in comprehension of language [60] and

gestures [61]. By extrapolation, program tracing, the ability to *execute* a computational process, is likely a component skill of program comprehension, the ability to *understand* that same process.

This claim has support from research in computing education and software engineering. Students who could correctly trace programs performed better on program comprehension questions [62, 63]. Students' comprehension errors can often be attributed to a flawed mental model (*notional machine*) of how a computer would trace individual instructions [64]. Expert programmers use tracing to understand code when its structure is not similar to a known schema [41, 65].

Prior qualitative studies have, in fact, shown that students encounter working memory difficulties while tracing. Vainio and Sajaniemi [66] conducted a qualitative analysis of students attempting to trace programs on paper. They found that some students would use a strategy called "Single Value Tracing". Namely, that students would remember "at most one value for all the variables in the program." One value! That observation is certainly quite suggestive of a working memory limitation.

While program tracing itself has not been the subject of much research, many related tasks have been investigated by both computer scientist and psychologists through the lens of working memory. I review the evidence about those tasks in the remainder of this section.

2.2.2 Arithmetic

Mental arithmetic is a well-studied task within the context of working memory [67]. It is also a "program" that people trace every day — adding multi-digit numbers is a procedure many people are taught to memorize and mentally simulate from a young age. For example, computing $47 + 18$ is similar to tracing the program in Figure 2.1 (left).

Graham Hitch [68] first found in 1978 that a working memory analysis of mental arithmetic could explain many observed calculation errors. While calculating, a person must maintain a number of *intermediates* in working memory, such as all previously computed digits and a carry bit. Based on a mathematical model fit to

```
def add(n1, n2):
  output, carry = [], 0
  for d1, d2 in reversed(list(zip(n1, n2))):
    intermediate = d1 + d2 + carry
    output.insert(0, intermediate % 10)
    carry = 1 if intermediate >= 10 else 0
  if carry > 0:
    output.insert(0, carry)
  return output

assert add([4, 7], [1, 8]) == [6, 5]
```

```
def square_twodigit(n):
  cn = min(n % 10, 10 - (n % 10))
  nmt = n + cn
  otn = n - cn
  p1 = nmt * (otn // 10 * 10)
  p2 = nmt * (otn % 10)
  sm = p1 + p2
  cn2 = cn * cn
  return sm + cn2

assert square_twodigit(23) == 529
```

Figure 2.1: Mental procedures for addition (left) and squaring a two-digit number (right). Procedures are represented as Python programs to emphasize the similarity of mental arithmetic to program tracing.

experimental data, Hitch's theory was that a person's probability of forgetting an intermediate increased exponentially with the number of calculations since computing the intermediate. For example, in mentally computing $138 + 326$, if a person first computes $6 + 8 = 14$, then they will forget the ones-place digit 4 with exponentially increasing probability after computing each subsequent digit.

Within the domain of mental computation, prior cognitive psychology research has focused on *memorized* procedures like addition, as opposed to tracing unfamiliar procedures presented through an external medium. Zhang and Norman [69] use a representational analysis to justify why Arabic numerals are an ideal number representation for performing mental multiplication. Campbell and Charness [70] ran an experiment where participants memorized an eight-step procedure for squaring two-digit numbers, shown in Figure 2.1 (right). They found that the majority of errors occurred within 5 stages of the procedure, and most errors could be explained as working memory errors (not calculational errors) where one intermediate was substituted for another.

2.2.3 Variables

To understand the influence of working memory on variables in program tracing, we can extrapolate the concept of an intermediate in mental arithmetic. For example, consider tracing this program:

```
1 x = 12 + 9 - 2
2 print(x + 3)
```

This trace involves three kinds of intermediates. First, computing $12 + 9$ involves the intermediates of multi-digit addition, e.g. a carry bit. Second, the expression $12 + 9 - 2$ requires remembering the intermediate 21 while computing $21 - 2$. Finally, the variable assignment $x = 19$ must be remembered while tracing the second line. The key idea is that each intermediate must be stored in working memory by association to its role in the program. That association could be behavioral (carry bit), positional (left hand side of an expression), or symbolic (the letter `x`). In this view, variables in program tracing are intermediates with association to strings. A person remembers `x = 19`, then retrieves the value `19` when cued with the variable `x`.

In the vocabulary of cognitive science, remembering a variable/value pair is *paired-associate learning*, and retrieving the value from memory given a variable is *cued recall*. Cued recall for paired-associate learning has been studied in a variety of domains. The simplest form of study is *memory span*: how many pairs can a person remember at once with no distractions? Multiple experiments on cued recall for pairs of common English nouns found that participants could remember on average 5 out of 10 pairs (when presented for 2s/pair) [71, 72].

Program tracing is not a pure memorization task, however — a person must interleave the storage of variables in working memory with the calculation of expressions. This raises the question: would calculation interfere with working memory for variables? While this question has not been studied directly, its inverse has prior work: working memory used by mental computation can be interfered with by certain competing tasks. In one experiment, participants had to mentally add three-digit numbers while concurrently performing a dual task. Participants were significantly more likely to forget the carry bit under a dual task which involves central executive

working memory (Trails task, speaking aloud an alternating sequence e.g. "A-1-B-2-. . .") versus a dual task which involves phonological working memory (articulatory suppression, repeatedly saying the word "the" aloud) [73].

2.2.4 Computation order

An important distinction between program tracing and mental arithmetic is that program tracing is accompanied by the program source code, while mental arithmetic uses a memorized procedure. The source code acts an external memory of the process's instructions, but also potentially for program state. For example, try tracing this program:

```
1 x = 8
2 q = 3 + x
3 r = 6
4 print(x - r + q)
```

A "linear" strategy, the most similar to how a computer actually executes a program, would trace from lines $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$, committing each variable/value pair to memory along the way. But you might have used an "on-demand" strategy: skim to line 4, then look up the value of each variable as needed, e.g. by going $4 \rightarrow 1 \rightarrow 3 \rightarrow 2$. Either strategy produces the correct answer, although on-demand strategies nominally become more challenging in the presence of side effects (e.g. writing to a file) because out-of-order tracing may not produce the same ordering of effects as linear tracing.

Vainio and Sajaniemi [66] observed that students tracing programs would adopt a strategy they called "single value tracing." In our terminology, students would trace on-demand with respect to variables assigned to constants. In the example above, a student might ignore line 1 (a "trivial" value in their words) and skip to line 2, looking up the value of `x` on demand. However, we do not know generally how often people will adopt a particular strategy, or how strategy relates to working memory.

For more complex programs (e.g. with control flow, indentation, methods, etc.) eye-tracking studies of programmers have shown that programmers will read programs non-linearly. Busjahn et al. [74] found that the gaze path of expert programmers was more similar to following control flow from top-to-bottom than to reading line-by-line

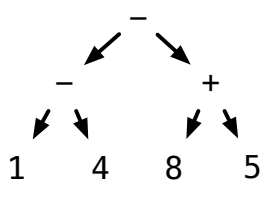

Figure 2.2: Three equivalent representations of the arithmetic expression $(1-4)-(8+5)$. Left is an nameless/tree representation. Center is a topological sort of the tree into a program using variables. Right is an arbitrary ordering of the nodes into functions.

like a book. A replication of Busjahn et al. by Peitek et al. [75] further found that more linear programs (e.g. with fewer function calls) correlated to less vertical eye movement.

2.2.5 Control flow

When tracing a program, a person needs to know what instruction to execute next, i.e. how to follow the program's control flow. If a person adopts a linear tracing strategy in a straight-line program, they only need to know the "instruction pointer", or the current position in the program. A person can trivially determine whether a line has been previously executed (above the pointer), or still needs to be executed (below the pointer).

In what situations, then, does a person need to remember more about control flow than the instruction pointer? Consider the three equivalent representations of an arithmetic expression shown in Figure 2.2. A trace through these programs is akin to a walk around the expression tree. For example, linearly tracing the variable program in Figure 2.2 starts from the leaves: observing 1 and 4, then computing $1-4$, and so on. By contrast, an on-demand tracing strategy for the variable program starts at the root: observing that the goal is `u - s`, then computing `u = v + j`, and so on. A standard tracing strategy for the function program should look similar to the on-demand tracing strategy for the variable program: start with `x()`, observe that `x`

depends on `u() - s()`, then compute `u()` and so on.

Because the variable program must be a topological sort of the tree, tracing linearly ensures that an expression's dependencies have always been computed before computing the expression. However, this property is lost if the person adopts a non-linear tracing strategy, or if the program itself is non-linear. When tracing the variable program on-demand or tracing the function program, a person must maintain a set of previously visited program positions (i.e. a *call stack*). After visiting `x()` → `u()` → `v()`, the tracer must remember that `v()` was contained in the definition of `u`. This information must either be retrieved from the program text or stored in working memory.

In the abstract, the on-demand tracing strategy has a task structure where a goal (e.g. compute `x = u - s`) and its state (`u = 13`) become temporarily replaced by a sub-goal (compute `s`) with separate state. Within cognitive load theory, this hierarchical subgoal-within-goal task structure has been consistently shown to induce working memory errors:

- Students solving basic Lisp programming problems make more errors attributable to working memory when the problem uses a more deeply nested expression [76].
- Students solving multi-step geometry problems (e.g. "to compute angle X, that's 180 - Y, so now I compute Y") make errors most commonly within subgoal stages where working memory load is highest [77].
- Students mentally distributing multiplication over algebraic expressions (e.g. $-3(-4-5x)-2(-3x-4)$) make errors most commonly when expanding the the second parenthetical, because the state of the first expansion must be held in working memory while accomplishing the secondary goal [78].

Hence, we should expect to people make increased working memory errors when tracing a program with nested control flow.

Chapter 3

Working Memory and Program Tracing

As described in Section 2.2, working memory limitations likely influence a person's ability to trace a program. However, beyond qualitative observations like that of Vainio and Sajaniemi [66], no further research has explicitly linked working memory to program tracing. Therefore I ran a series of experiments to test the predictions of working memory theory in the context of tracing. Specifically, I investigated two high-level questions:

1. How much program state can a person hold in working memory?
2. How do working memory limitations for program state influence tracing strategies?

Working memory theory suggests a few coarse answers: (1) people can't remember much state, and (2) they will forget relevant state while tracing and have to look up that information on the fly. But I wanted to dive deeper into the nuances of each question: how much program state can a person remember in pure recall vs. interleaved with mental arithmetic? Do people linearly trace in the same fashion as a computer, or do they optimize their strategies for working memory? To that end, I ran four controlled experiments to elucidate the influence of working memory for

program state on tracing. In each experiment, I carefully restrict the participants' view on the program, for example by seeing a single line/function at a time, or blurring lines of code until hovered. Then I use the participants' responses and their mouse movements to derive quantitative measures of working memory influence.

This chapter describes the design and results of each experiment, as well as an interpretation of their implications for theory and design. We ran four experiments to explore the role of working memory for tracing with two kinds of program state: data (variable/value pairs) and control flow (tracking data dependencies).

3.1 Variable Recall

First, we sought to measure the total capacity of working memory for variable/value pairs without any interference. This provides a baseline for later comparing how other tracing tasks (e.g. arithmetic) affect working memory capacity. Specifically, we asked two questions:

1. How many variable/value pairs can a person keep in working memory?
2. How does the kind of variable name affect its memorability?

In a realistic program, both of these questions are inevitably confounded by the semantic relationship between a variable and its value. For example, `greeting = "Hello"` is likely easier to remember than `foo = "Hello"` because "Hello" is a kind of greeting. To avoid this confounder, we will consider the case where a variable has no particular relationship to its value. We only consider numeric values, and we consider variable names that are either common English nouns or single letters.

For the first question, our hypothesis is that participants should hold about 5 variable/value pairs in working memory, consistent with prior work on cued recall in paired-associate learning [71, 72]. For the second question, we hypothesize that English words should be more memorable than single letters due to having *some* meaning if not a *relevant* meaning, and therefore should correspond to more pairs remembered on average.

3.1.1 Methodology

In this experiment, we tested working memory capacity via cued recall. Participants were presented with several expressions of the form `variable = value` for 2s each. They were then prompted to answer `variable = ?` for every observed pair. The specific methodology and parameters are similar to those used in Nobel and Shiffrin's paired-associate experiments [71]. Within a given trial, we randomly generated 10 variable/value pairs. Each value is a digit from 0 to 9. We have two different conditions for variable names: single-letters (A to Z), and common English nouns (e.g. cave, tax, cherries). The participant was presented with one pair at a time for 2s per pair. The presentation used programming syntax, e.g. `x = 4`. After the final pair, the participant was prompted to input the corresponding value of all variables, e.g. `x = ?; f = ?`. The order of prompts was randomized with respect to the order of initial presentation. (The randomization was used to defeat mnemonic strategies that simply memorized variables and values as two separate streams of serial data which we observed in a pilot.) We then measured the participant's accuracy as the number of values correctly recalled.

Unlike some prior work on cued recall (e.g. Nobel and Shiffrin [71]), we did not add a distractor task (e.g. mental arithmetic) inbetween the presentation and prompt phases. In that setting, the goal is to measure long-term memory capacity, while our goal is to measure working memory capacity. As a result, we may expect to see a larger number of recalled pairs than in their experiment.

Each participant completed 4 trials per condition. The order of trials was counterbalanced, and this experiment had 15 participants. In this and all other experiments, we recruited participants with the following criteria and instructions:

- **Participant pool:** Participants were recruited from Amazon Mechanical Turk within the United States and Canada. They were compensated a fixed amount, estimated by timing the average duration of a pilot study and paid a corresponding amount prorated at $15/hr.

- **Required expertise:** We required that participants have a basic competency with tracing Python programs, which we ensured with a short pre-test before

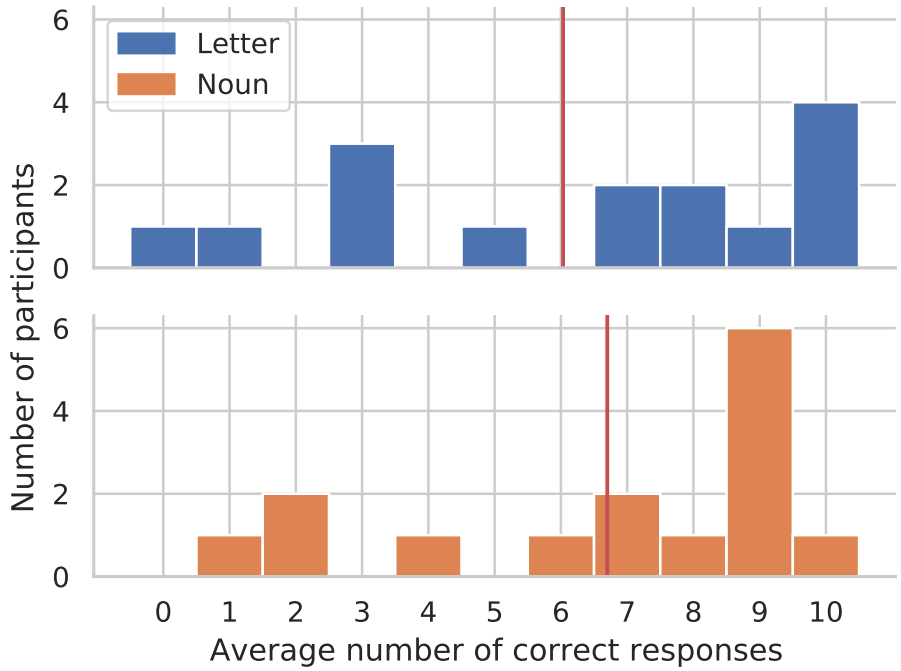

Figure 3.1: Number of values correctly recalled when cued with the paired variable, averaged by participant. The red line indicates the mean value of each distribution.

each experiment. Participants also needed to correctly complete a sample trial before doing an experiment.

- **No memory aids:** Participants were instructed to not use pen/paper or any other external media as a memory aid.
- **No distractions:** Participants were asked to complete the experiment without any visual or audio distractions, and to complete the experiment in one sitting.

3.1.2 Results

The distribution of average accuracy by participant in each condition is shown in Figure 3.1. On average across both conditions, participants were able to recall a mean/standard deviation of 6.5 ($\sigma = 3.7$) for letters and 7.2 ($\sigma = 3.4$) for nouns, and a median of 7.5 and 9.0, respectively. Notably, 5 out of 15 participants achieved an average of 9 or better in both conditions. Using a Wilcoxon signed-ranks test,

the difference in accuracy between the conditions was not statistically significant ($T = 40.5, p = 0.45$).

3.1.3 Discussion

For semantically-unrelated variable/value pairs, these results suggest that a person can remember on average about 7 pairs in a pure memorization setting. This number is higher than hypothesized, likely because our experiment did not include a distractor tasks like the related work. The notion of average capacity may also be somewhat tenuous if memory capacity varies so widely across individuals. Participants could not remember significantly more pairs when random nouns were used as variable names. This result suggests that nouns are not more memorable, or name memorability does not matter if the semantic relationship of variable and value is arbitrary.

3.2 Variable Recall with Arithmetic

Program tracing involves the maintenance of program state while performing operations like arithmetic. So next, we consider: how does working memory capacity for variable/value pairs change when interleaved with mental arithmetic? Within the multi-component model of working memory [79], prior work has found that three-digit mental arithmetic interferes with simultaneous tasks that use central executive working memory, but not phonological working memory [73]. Single-digit mental arithmetic problems have also been repeatedly confirmed to use central executive working memory [67]. The cognitive action of updating working memory in a running memory task is coordinated by central executive working memory [80], therefore we hypothesize that participants should have a lower effective working memory capacity for variable/value pairs than in the prior experiment.

3.2.1 Methodology

In this experiment, participants traced a straight-line program of variables assigned to arithmetic expressions, one line at a time and without the ability to look back. We

measured working memory capacity by the point at which the participant provides an incorrect response. Specifically, we randomly generated programs of the form:

```
1 x = 3 + 4
2 t = x - 1
3 b = t + x
4 z = x - b
5 # ...and so on
```

Each line assigns a new variable to an arithmetic expression that involves randomly selected variables from the previous lines (or a constant for lines 1-2). Variables names are lower-case single letters and all constants are between 0 and 9. The only arithmetic operations are addition and subtraction to keep the mental effort of any individual operation relatively small. The program is 11 lines long, such that the participant would have to remember 10 variables by the last line.

Within a given trial, we present the participant one line at a time. In the above example, the participant would start by seeing: "`x = 3 + 4`. What is the value of `x`?" After entering 7, that line disappears and the participant sees: "`t = x - 1`. What is the value of `t`?" This way, the participant must commit the variable/value pairs to working memory, instead of looking them up later. After entering a value, the software waits for two seconds before proceeding to the next line. This process repeats until the participant responds incorrectly.

An incorrect response cannot necessarily be attributed to a working memory failure. The participant could simply add two numbers incorrectly (which could itself be a working memory failure, but for now we ignore that possibility). Using the methodology of Campbell and Charness two-digit squaring experiment [70], we classify the incorrect response as either a substitution error or calculation error. Substitution means the participant entered an a number that could have been computed by mixing up two previously seen variables. For example, if the current state is `x = 7; t = 6; b = 13` and for `x - b` the participant enters `1`, then we classify that as a substitution of `t` for `b`, a failure of working memory. All other incorrect numbers are considered calculation errors.

Each participant completed 10 trials, and this experiment had 15 participants

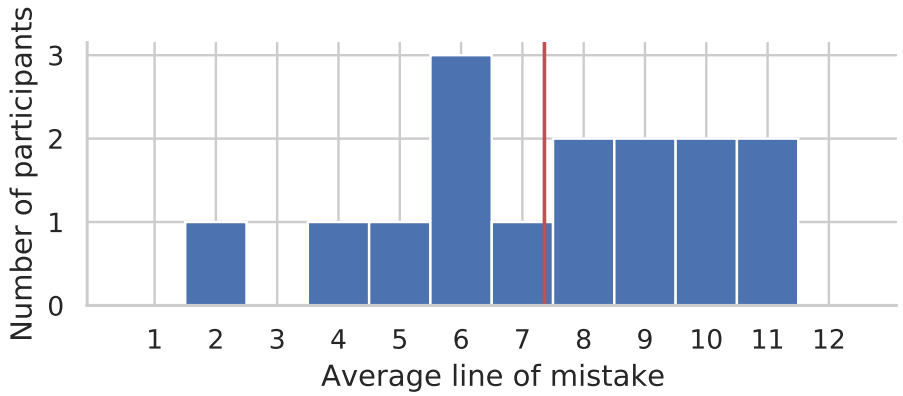

Figure 3.2: Histogram of the average line of code at which a participant made a mistake in Experiment 2.

(one participant was excluded for on average failing at the first line, leaving 14 participants).

3.2.2 Results

Each participant finished the experiment at a particular line, either by providing an incorrect response (final line 1-11) or completing the task without error (final line 12). The histogram of participants' average final line is shown in Figure 3.2. Across all participants, the average final line was 7.9 ($\sigma = 3.7$) and median was 8.0.

To compare these results to Experiment 1, first we have to model working memory capacity in terms of the line of failure. If participants failed at line 8, then they either calculated incorrectly or forgot at least one of 7 preceding variables, suggesting a working memory capacity of 6 variables. Hence, we model effective working memory capacity as line number $-\,2$, indicating the average working memory capacity for this experiment was 5.9. To test the hypothesis of whether mental calculations reduced working memory capacity, we compared the distribution of working memory capacity in Experiment 2 vs. the distribution of working memory capacity measured in Experiment 1 in the letter condition. Using a Kruskal-Wallis test, the difference was not significant ($H(1) = 1.51, p = 0.21$).

For each trial where the participant gave an incorrect response, we classified the

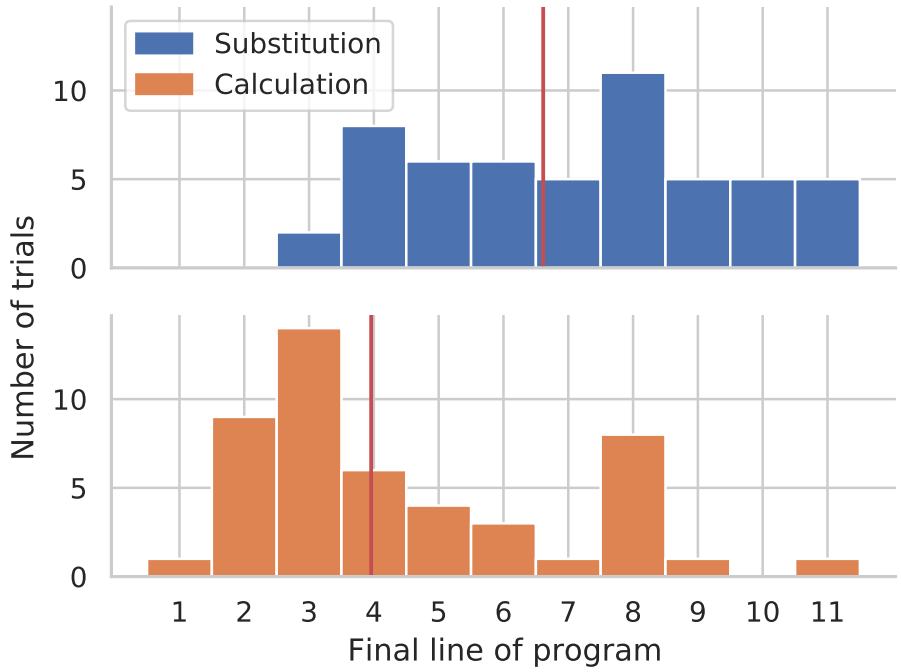

Figure 3.3: Histogram of error types at each line in Experiment 2. Participants made 70% more substitution errors than calculation errors.

error as substitution or calculation. The distribution of errors by type is shown in Figure 3.3. Of the 101 total errors, 53 were substitution and 48 were calculation. The median calculation error happened at line 3.5, while the median substitution error happened at line 7.0. Using a Kruskal-Wallis test, the difference was significant ($H(1) = 25.57, p < 0.001$).

3.2.3 Discussion

These results do not support the hypothesis that performing mental arithmetic reduces working memory capacity. Participants in Experiment 2 on average remembered fewer variables than Experiment 1, but the different is not statistically significant. However, the experimental designs are somewhat different (e.g. continually prompting for information vs. asking for it all at once) and the test is between-subjects, so future work can control for these factors. Consistent with Campbell and

Charness [70], we found that over half of mistakes can be attributed to substitution in working memory. Specifically, errors could be explained by accidentally swapping the association between variables in working memory, as opposed to forgetting them entirely.

3.3 Straight-line tracing strategy

Next, we consider how working memory influences tracing for straight-line programs when the complete program is available at all times. We investigate two questions:

1. Do participants demonstrate tracing strategies that diverge from linear execution?

2. Where in the program are participants most likely to have working memory errors?

As described in Sections 2.3 and 2.4, we expect there to be two basic tracing strategies: linear and on-demand. Prior work has shown that people will trace somewhat out of linear order [66], but we do not know precisely to what extent or how often. Hence our first goal for this experiment was to design a code viewing interface such that we can deduce the participants' strategies from their viewing patterns. Then once we have identified which strategy a participant is using, we can hypothesize where errors are likely to occur:

- For linear tracing, the primary source of working memory load should be remembering variable/value bindings. Based on Hitch's model of errors in mental arithmetic [68], a person is more likely to forget values computed earlier in the program than later. Then participants should forget computations in the top lines of the program (leaf nodes) more frequently than later lines (inner nodes).

- For on-demand tracing, the primary source of working memory load should be tracking the current path in the expression tree. Based on the predictions of cognitive load theory for hierarchical task structures [76, 77, 78], participants

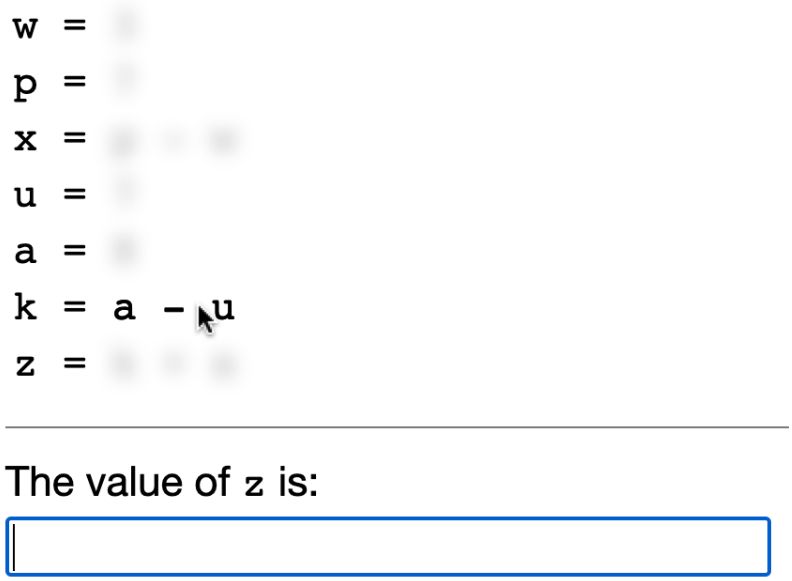

Figure 3.4: Interface for tracking the participant's attention during tracing. Each expression is blurred unless the particpant's cursor hovers over it.

should make the most working memory errors at the deepest sub-goal (i.e. leaf nodes). A working memory error would cause a participant to forget their path to the leaf (i.e. inner nodes), so participants should revisit inner nodes more frequently than leaf nodes.

3.3.1 Methodology

In this experiment, we track where a participant's attention goes as they trace through a straight-line program. Given the participant's attention over time, we classify their tracing strategy as linear or on-demand. Then we identify working memory errors in their traces based on re-visits to lines of the program.

To track attention during tracing, we created a code viewing interface where each expression is blurred by default, i.e. a restricted focus viewer [81] (we did not use an eye-tracker so the experiment could be performed in a browser on Mechanical Turk). The participant can move their mouse over an individual line to bring it into focus. This way, the mouse acts like a foveal region in an eye tracking experiment — it

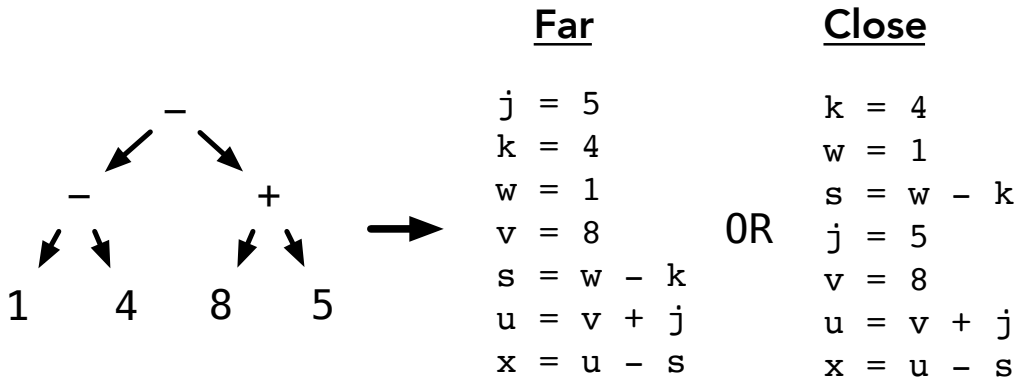

Figure 3.5: Process of turning an expression tree into a program. Far has the highest average distance between variable definition and use, while Close has the lowest.

represents the user's current focus. W Figure 3.4 shows the interface. Because the mouse may briefly hover over other lines while moving to a destination, we discard any hover actions that occurred for less than 300ms. This number was selected by inspecting the noise in the data, but in the spirit of multiverse analysis [82] we confirmed that all significant results still hold with $p < 0.05$ for a threshold of 100ms and 500ms.

To generate straight-line programs, we start by randomly generating an arithmetic tree of size 7 with constants at the leaves and binary operators at the inner nodes. Every node is given a random single-letter variable name. To map a tree to a straight-line program, we have to pick an ordering that respects the dependencies in the tree, i.e. any topological sort. The main difference between sorts is the distance from a variable definition to its usage. For this experiment, we consider two conditions: sorts with the highest average distance ("Far"), and lowest average distance ("Close"). Figure 3.5 shows an example translation.

Each participant completed 5 trials in each condition (Far/Close), and this experiment had 15 participants. 3 participants were removed due to exceptionally poor performance (accuracy less than 20%), leaving 12 participants.

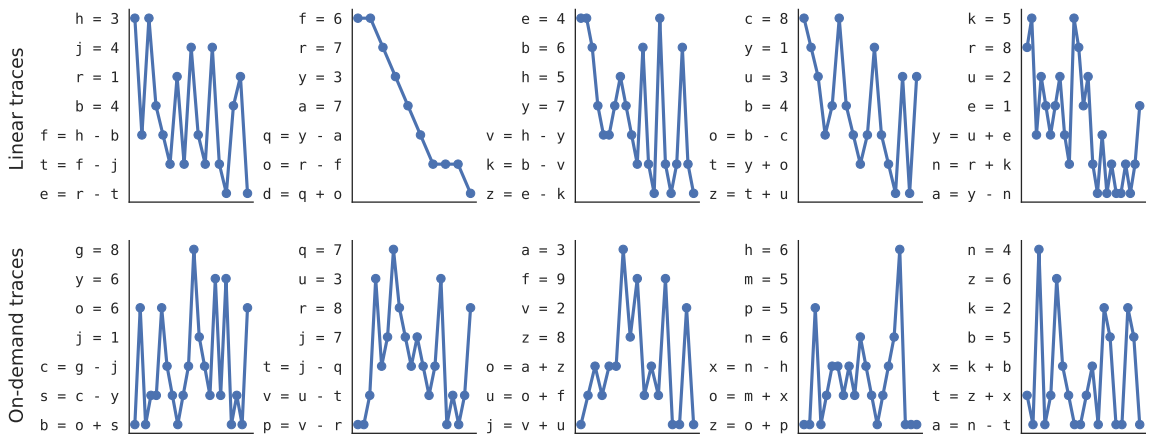

Figure 3.6: Example traces of straight-line programs categorized by strategy. The x-axis is time, and the y-axis is the lines of the program. Each dot is a hover event.

3.3.2 Results

First, for each trial, we classified the participant's mouse movements as indicating an on-demand or linear strategy. As a simple heuristic, we classify a trial as linear if the first line is first visited before the last line, and on-demand otherwise. Across all trials, 55% were classified as on-demand and 45% as linear. Most participants appeared to individually prefer one strategy over another — 9 out of 12 participants adopted a single strategy at least 70% of the time. Figure 3.6 shows a sample of actual traces classified as linear (top) and on-demand (bottom) in the Far condition. In these examples, we can already observe a few working memory influences:

- The top-left and top-center participants with the linear strategy traced linearly in the set of binary operations, but looked up constant values on-demand (consistent with Vainio and Sajaniemi [66]). Others like the top-middle-left participant traced linearly in both variables and binary operations.

- Some actions are clearly attributable to strategy vs. working memory error. For example, the bottom-left participant goes from `b = o + s` to `o = 6`, indicating an intentional on-demand shift in attention, but then returns to `b` before moving to `s = c - y`. This indicates a working memory error of forgetting the second operand to `b`. By contrast, the bottom-middle-left participant goes straight

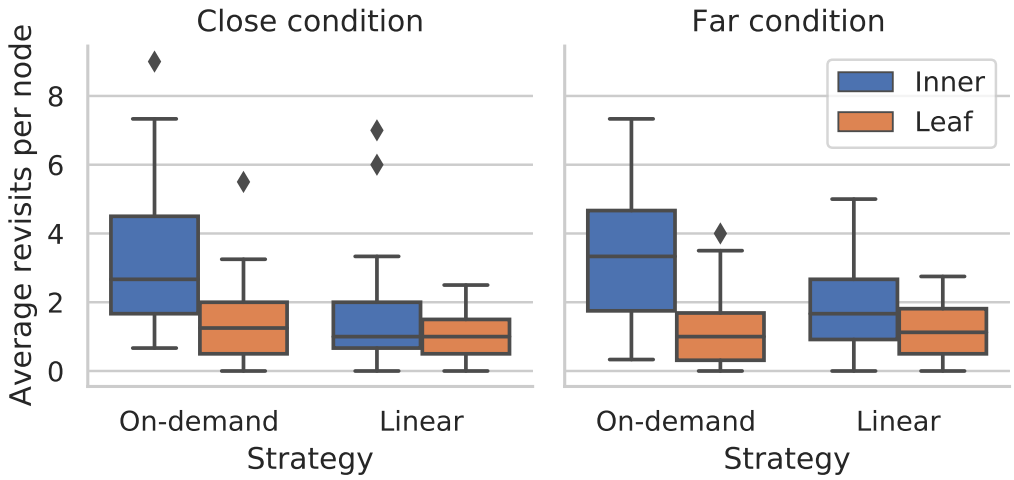

Figure 3.7: Distribution of average revisits to expression tree nodes, separated by node type, tracing strategy, and program type. The difference between visits to inner nodes and leaves was greater in the on-demand strategy than the linear strategy.

from `u` to `t` after observing `v = u - t`, indicating no comparable working memory error.

To quantitatively analyze forgetting, we counted the number of times a participant re-visited each line of the program. We then categorized the lines as either leaves of the expression tree (constant variables) or inner nodes (binary operations). Figure 3.7 shows the distribution of average re-visits separated by participant's strategy and the program sorting condition. To test for differences between distributions, we fit a linear mixed model with a three-way interaction of strategy, node type, and variable distance as the fixed effects and participant as the random effect. The number of revisits was the dependent variable. A one-way ANOVA showed a significant difference between revisits in the interaction of node type and strategy ($p < 0.001$), but not variable distance. We ran six post-hoc T-tests on all contrasts between pairs of the interaction, adjusted by the Tukey method. We found:

- Participants on average revisited nodes 1.37 more times per node in the on-demand strategy than the linear strategy ($T(200) = 8.66, p < 0.001$).
- Participants revisited inner nodes more than leaf nodes on average 2.1 more

times in the on-demand strategy ($T(200) = 9.95, p < 0.001$), and 0.63 times in the linear strategy ($T(200) = 2.69, p = 0.038$).

- Participants revisited inner nodes an average of 1.11 more times in the the on-demand strategy than the linear strategy ($T(216) = 4.23, p < 0.001$).

3.3.3 Discussion

We have found evidence that participants will adopt both on-demand and linear strategies for tracing straight-line code. The results also support our hypothesis that working memory theory predicts where a participant will make errors given a strategy: at leaves for linear, and at inner nodes for on-demand. Moreover, we also found evidence that the linear strategy causes fewer working memory errors than on-demand. That suggests that the cognitive load caused by tracking paths through the tree is greater than storing variable/value pairs in working memory.

3.4 Function tracing strategy

Finally, we extend the methodology of the prior experiment to programs with functions. Similarly, we want to understand: what strategies will people use to trace programs with functions, and where are they most likely to make errors? Recall from Section 2.4 that tracing a function program starting from the top-level function call is equivalent to an on-demand tracing strategy in a straight-line program. When projected onto the abstract expression tree, both traces start at the root and proceed to the leaves.

However, now a linear strategy should be harder than before, because a person has to mentally reconstruct the dependency graph and topological sort before tracing linearly. Hence we hypothesize that more people should adopt an on-demand strategy than in the prior experiment. For a given strategy, though, we hypothesize that working memory errors should occur most frequently as predicted in the prior experiment: on-demand traces forget the inner nodes, and linear traces forget the leaves.

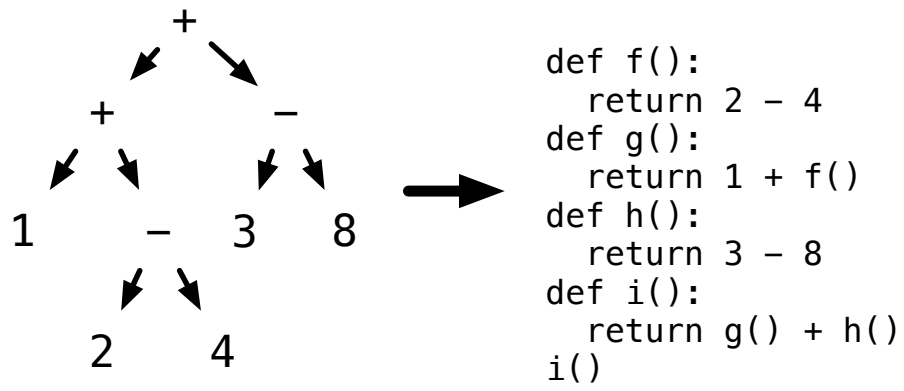

Figure 3.8: Transformation of an expression tree into a sequence of functions. We only convert binary operations into functions, because turning every node into a function took too much per-trial time for participants in a pilot study.

3.4.1 Methodology

In this experiment, participants trace through a program with functions while we track which function has their attention. Like the prior experiment, we randomly generate an expression tree and convert it into a program with functions as shown in Figure 3.8. Then to track the participant's attention, we created a code viewing environment where one function is displayed at a time, shown in Figure 3.9. The participant is given a bank of buttons with the name of each function, and clicking on the button goes to the function's source, similar to an IDE. A participant's trace through the expression tree corresponds to their sequence of button clicks. The order of buttons is randomized, so it bears no relationship with respect to the functions' order of usage in the program.

Each participant completed 10 trials, and this experiment has 15 participants (1 was removed for poor performance, leaving 14 participants).

3.4.2 Results

First, we classified each trace as an on-demand or linear strategy. We used a simple heuristic similar to before: if the participant visited the root function node first, they used an on-demand strategy, and they used a linear strategy for visiting any other

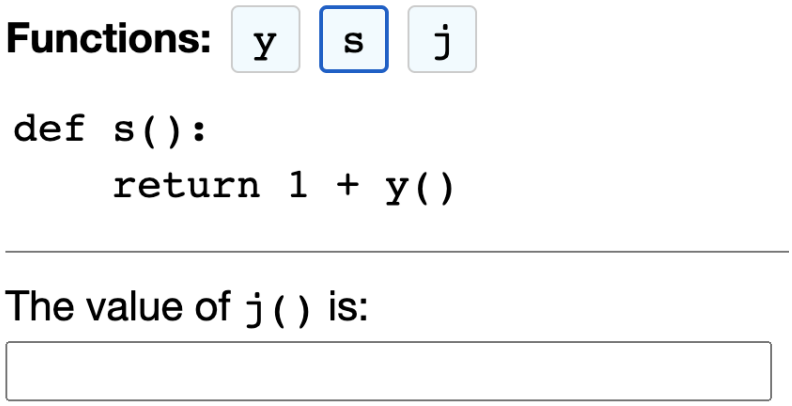

Figure 3.9: Experiment interface for showing participants one function at a time.

node first. Figure 3.11 shows several examples of traces around the expression tree within each strategy. Out of 140 trials, 54% used a linear strategy and 46% used an on-demand strategy. Like the prior experiment, 11/14 participants consistently picked one strategy at least 70% of the time.

Next, we computed the average number of revisits to inner and leaf nodes in each trace. The distribution of revisits by strategy and node type is shown in Figure 3.10. As in the previous experiment, we fit a linear mixed model with strategy and node type as fixed effects and the participant as a random effect, and the number of revisits as the dependent variable. Using a one-way ANOVA, the only statistically significant difference in revisits is the node type ($F(263) = 41.06, p < 0.001$).

3.4.3 Discussion

These results provide mixed support for our first hypothesis. It was not true that most participants preferred an on-demand strategy. In fact, we found the preponderance of linear traces somewhat bewildering — participants appeared to just click through each function, remembering what they could and then synthesizing the information into an answer at the end. However, the results do suggest that the linear strategy had increased working memory load for function programs compared to straight-line programs. In Experiment 4, there was not a statistically significant difference between working memory errors in the on-demand and linear strategies, whereas on-demand

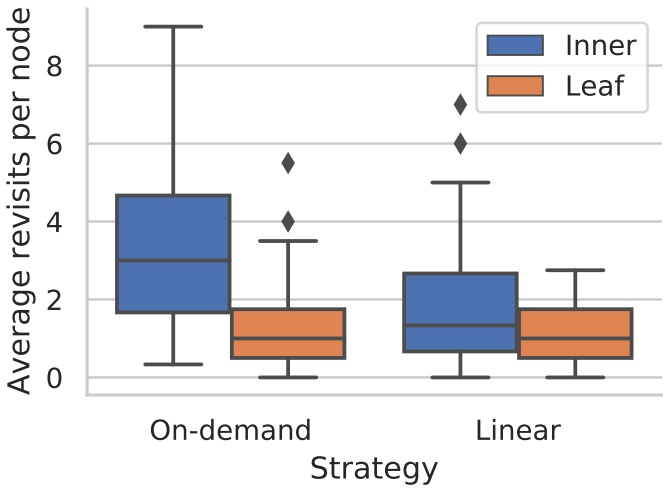

Figure 3.10: Distribution of revisits to nodes by node type and strategy. Participants visited more inner nodes than leaf nodes in the on-demand strategy, but not in the linear strategy.

traces contained more errors in Experiment 3.

The results do not support our second hypothesis, that participants revisit nodes as predicted by working memory theory. The relative revisits to inner vs. leaf nodes was not different between strategies. One possible explanation is that the notion of a linear strategy is different for the straight-line interface than the function interface. The function interface does not expose the call graph, so a participant cannot easily walk from the leaves to the root as they can when scanning top-to-bottom in the straight-line code.

3.5 General Discussion

In summary, we made six findings from our experiments:

- A person can hold about 7 variable/value pairs in working memory, and can remember variables named as random nouns equally well as random letters.
- Single-digit mental arithmetic does not reduce WM capacity for variable/value pairs.

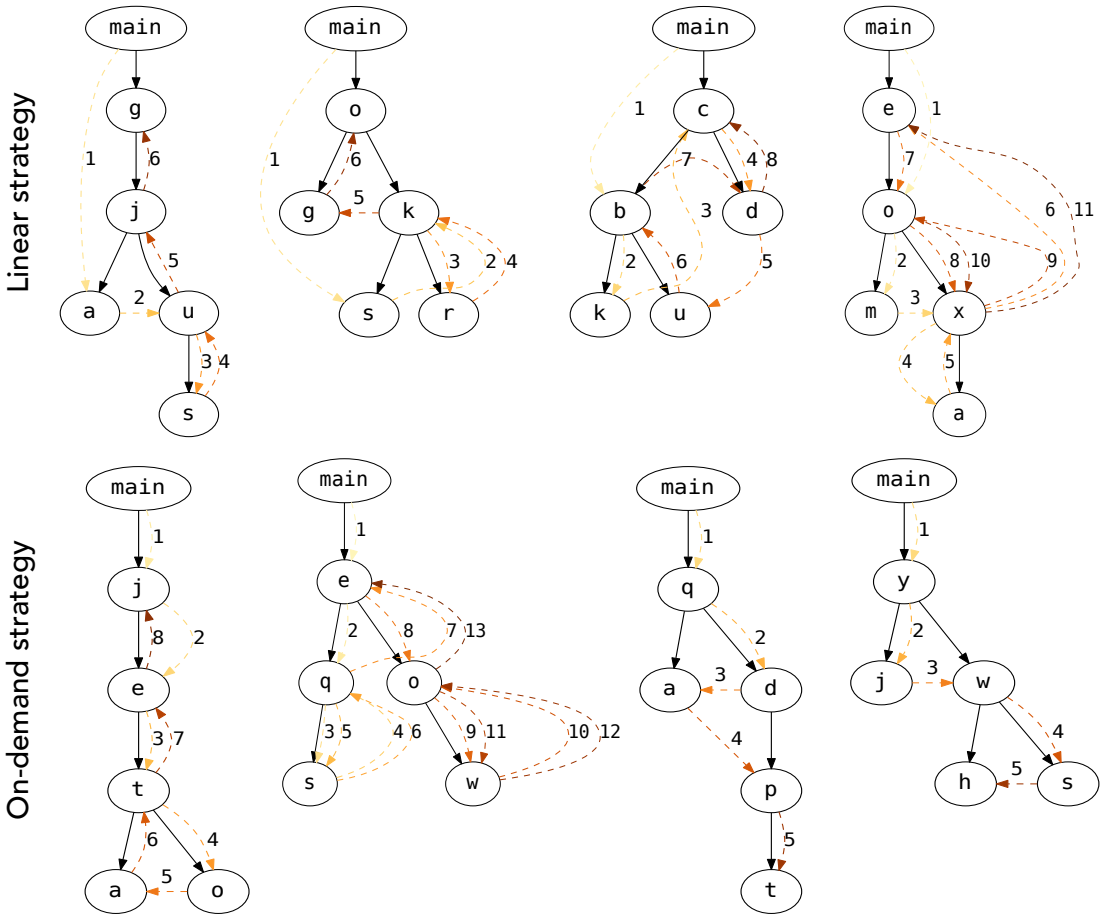

Figure 3.11: A sample of traces from the participants, grouped by strategy. Each tree represents a dependency graph of function calls e.g. `j` → `e` means the definition of `j` calls `e()`.

- When recalling values for mental arithmetic, a majority of errors could be explained by recalling a value bound to a different variable held in WM.
- People will trace a straight-line program both linearly and on-demand, and a given person is likely to stick with one strategy.
- People will make WM errors in different parts of the program based on their tracing strategy, and on-demand strategy causes more WM errors than the linear strategy for a straight-line program.

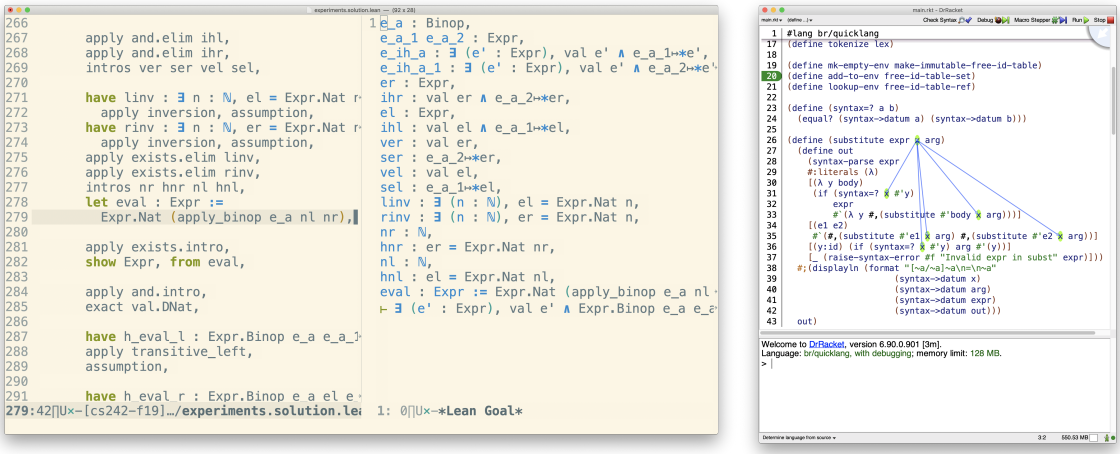

Figure 3.12: Left: the Lean theorem proving language inside its Emacs IDE. The right pane shows all variables in scope on the current line in the left pane. Right: the DrRacket IDE shows all usages of a variable on mouse hover.

- People will not necessarily trace a function in order of execution.

These findings occurred within a controlled experimental setting and within a specific application domain. As with all laboratory / cognitivist research, this naturally raises the question — do these results generalize to realistic programs? And how can we practicably apply these results to high-level design?

3.5.1 Limitations

Our findings are not universal statements about e.g. all variable/value pairs. For example, if a person tries to remember `x = [1, 5, 7]`, the list is a more complex object to be stored in WM than a single digit. A person likely cannot remember as many variable/list pairs as variable/number pairs. More generally, several factors limit the generality of our current results and point to interesting future work.

- **Data types:** we focus on integers in this work. But practical programs have booleans, strings, structs, classes, and so on. Variables assigned to values of different data types might be less likely to be swapped in WM.

- **Mental operations:** we focus on single-digit mental arithmetic with addition and subtraction. Harder mental operations may shift the balance e.g. in Experiment 2 and interfere with WM for program state.
- **Mutability**: we only test programs that assign to variables once, i.e. immutable state. Imperative programs often rely on in-place updates to databases, files, or so on. Mutating variables means updating an existing association between a variable/value in WM rather than adding a new one, which may be subject to different kinds of interference. Mutation also potentially increases the difficulty of on-demand tracing.
- **Control flow constructs:** we used functions due to their close relationship to on-demand tracing. Standard structured programming constructs like if-statements and for-loops should also be investigated for their WM influence.
- **Schematic structure:** our programs do not solve any common problem, e.g. sorting a list or computing the quadratic formula. A person's tracing strategies and WM encodings of program state would likely differ for programs with a recognizable structure, i.e. pieces that fit into a known schema.

3.5.2 Implications for theory

One goal of this work is to contribute towards a broader theory of program comprehension: how do people understand programs, and how do aspects of cognition influence that process? We focused on program tracing because it is a likely component skill of comprehension. We have shown that WM both limits how much state a person can remember, and causes different kinds of errors based on tracing strategies. However, most popular models of program comprehension focus largely on schemata/plans as a mechanism of comprehension, e.g. the seven models in the survey of von Mayrhauser and Vans [83]. We think that tracing deserves a closer look to understand its role in comprehension.

One major finding of our work is that different people will adopt substantially different tracing strategies for the same program. We demonstrated that both the

linear and on-demand strategies are common, and that individuals seem to prefer one strategy. But we do not yet know: what factors made a person pick one strategy over the other? And in what situations is a particular strategy better than the other? We found that linear tracing caused fewer WM errors than on-demand tracing for straight-line programs, but not for function programs. Future work should identify other factors that influence this trade-off.

Additionally, tracing as a task likely lies on a continuum of abstraction. We focused on tracing with concrete inputs and outputs, but one can imagine tracing over symbolic values, akin to the method of abstract interpretation in static analysis [84]. In this setting, variables are represented not by *values* but by *properties*. For example, if `x > 0` and `y = x + 1`, then we know `y > 1`. Abstract tracing likely introduces even more WM load than concrete tracing. For example, while tracing "`if (x != NULL) { .. }`", the property "`x != NULL`" must be kept in working memory while reasoning about the body of the if-statement. A WM account of tracing should consider tracing at all levels of abstraction.

3.5.3 Implications for design

Another goal of our research is to generate design principles for reducing WM load that can be applied to IDEs, refactoring tools, style guides, and so on. We suggest a few guidelines that extend from our findings. As a general framing, it's important to observe that every experiment had significant between-subjects variability in memory capacity, time to completion, accuracy, strategy, and so on. Accessible programming tools need to account for working memory differences, e.g. tool designers should not assume that their users will be able to remember as much as themselves.

Reduce variable scope.

If a program has many overlapping variables, it will be hard for a person to keep the variables' values in working memory as shown in Experiments 1 and 2. Therefore, programs should avoid having many variables within a given scope where possible. One implementation of this suggestion is to put a variable's definition as close as

possible to its usage. This advice is consistent with most modern style guides, for example the Google C++ Style Guide [85]. Our work confirms that this style has genuine cognitive benefits, as opposed to being a matter of preference.

This advice could also be mechanized as a complexity metric. The compiler technique of liveness analysis can identify when the live ranges (point from definition of variable to its last use) of variables will overlap. To test the technique, we applied it to every function in the Python standard library. We identified that some functions have up to 18 overlapping variables! We also found that functions with many parameters had a high complexity score, since they effectively are declaring dozens of variables at the top of a block rather than close to their usage, a de facto violation of the style rule.

Visualize variable context.

As shown in Experiment 3, people with different tracing strategies will make different kinds of WM errors, and so will need different kinds of tools to augment WM. A person tracing linearly needs to keep track of previously seen variables, and so a programming environment can visualize information about all the variables in scope up until a particular line. For example, the Lean interactive theorem prover [86] in Emacs (Figure 3.12, left) will show all the variables in scope at the current line based on the editor's mouse position. Lean only shows the name and type of the variable, but a visualization could also show the last line of code modifying that variable, or other semantic information.

By contrast, a person tracing on-demand needs to keep track of their path through the variable dependency graph so they can follow it back. If a person forgets a part of their path, a programming environment can visualize information about where a variable is used to help refresh their memory. For example, the DrRacket IDE [87] (Figure 3.12, right) will show all references to a variable on mouse hover. The point is that these ideas already exist in some form within (admittedly niche) programming tools, but *every* IDE should support *both* of these visualizations to reduce WM load during program tracing.

Externalize program state.

Many of our experiments would be significantly different if the participants had been tracing on paper with a pen, able to write down program state such as variable values or flow markers. Providing programmers the ability to externalize their tracing process could significantly reduce working memory load.

However, nearly all interfaces for displaying code outside of an IDE are read-only. For example, GitHub has a specialized interface for pull requests that propose a change to a code base. The entire point of this interface is for a reviewer to understand a change, but the interface provides no "margin notes" or other features for throwaway annotations. Consistent with the theory of distributed cognition [88], code comprehension tools should consider how to enable programmers to externalize their thought process to the environment without needing to print out code.

Chapter 4

Program Slicing: Applying Cognitive Theory to Design

As established in Chapter 2, there is a large body of theories about cognition and programming. Chapter 3 further contributes to this body concerning the role of a specific facet of cognition (working memory) in a specific task (program tracing). Theories like these are *descriptive*: given a task, we can predict that a particular kind of person will think a certain way. But my ultimate goal is to make these theories *prescriptive*: to guide the design of new programming tools.

One approach is to generate *cognitive design principles*, as described by Tversky et al. [89] for the case of visualizations:

> "Our approach combines research in cognitive and computer science in three iterating steps:
>
> 1. Revealing the mental representations people have for a given domain and the visual devices they use to convey it, yielding domain cognitive design principles.
> 2. Developing algorithms that create effective visualizations based on cognitive design principles.
> 3. Testing the visualizations to insure that they adequately convey the desired information."

For example, Tversky et al. show that cognitive design principles can inform tools for automatically generating visualizations like route maps. Based on psychological experiments that elicited participants' mental models of geography, they determined that "people don't accurately apprehend or represent distances or angles." This observation lead to the cognitive design principle that "geometric information can be simplified to increase emphasis on the turning points." This design principle was applied to create an algorithm for automatically generating maps that are easier for people to follow than more realistic maps [90].

Inspired by this approach, I set out to apply it to programming by leveraging the experimental results in Chapter 3. I showed that a person's working memory significantly limits their ability to mentally maintain even small quantities of program information at a time. The question then becomes: what higher-level mental strategies will programmers adopt to overcome this cognitive limitation?

One implication of working memory limitations for many cognitive tasks is the *reduction of working context*. When solving a complex problem, a person cannot hold all possible factors in their head at once, and therefore must focus on smaller subproblems that eventually lead to the final answer. For example, when a problem involves a search space, people will only consider a single solution at a time and explore the space depth-first, as shown in chess [91] and abstract logical reasoning [92].

IDE extensions targeted at reducing working context have consistently shown to improve programmer productivity. For example, in the Code Bubbles IDE [93], if two functions are nearby in the call-graph, then they are placed adjacent to each other in the IDE canvas, which helps programmers focus on just the code relevant to a particular code path. The Mylar plugin for Eclipse [94] integrates a task list into the IDE which tracks which files are relevant to a given task, which helps programmers who are switching between multiple tasks.

One challenge with both of these approaches is they still require significant user effort to learn and use — they are quite far from the ambient visualizations described in Section 1.2. So I started thinking: are there automatic program analyses that could help programmers reduce their working context? After going back through the old programming/psychology literature, I saw a clear connection to one specific

technique: program slicing.

4.1 The Cognitive Basis for Slicing

Program slicing is unique in the history of programming tools for having a tight connection between cognitive psychology and PL theory. Slicing was first described by Mark Weiser[1] in his 1979 Ph.D. thesis "Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method" [95]. The work is best known from his two later papers: "Programmers Use Slices When Debugging" [96] and "Program Slicing" [97].

In the first paper [96], Weiser had read of qualitative evidence for a variety of ways that programmers debug programs:

> Gould [98] reports that many programmers start debugging by carefully reading the faulty program from top to bottom, without ever bothering to look closely at the erroneous program output. Dijkstra [99] and others have proposed that debugging time could be shortened by rigorous reasoning about a program's correctness. However, perhaps the most basic method of debugging is to start at the point in the program where an error first becomes manifest, and then proceed to reason about the sequence of events (as verified by the program text) that could have led to that error. [...] Gould [98] and Lukey [100] report instances of programmers working backwards from an error's appearance.

To test the hypothesis that programmers worked backward from a bug, Weiser devised an experiment using the memory-recall methodology described in Section 2.1.2. Programmers were given a buggy Algol-W program and asked to debug it. Afterwards, programmers were shown program fragments (from the original program or distractors), and asked to rate their certainty of whether the fragments appeared

[1] Weiser would later go on to become a pioneer in HCI, developing the concept of *ubiquitous computing* as a manager at the famous Xerox PARC research lab. As someone who also happens to be writing a Ph.D. thesis about cognition and programming, if you need someone to manage your research lab, let me know!

in the original program. Weiser compared the accuracy of ratings across different plausible fragments such as "relevant slice" (slice of the bug), "relevant contiguous" (contiguous code overlapping the relevant slice), and others.

The results showed that the relevant slice was indeed recognized more frequently than an irrelevant slice or a random set of code, although not more frequently than the relevant contiguous code. Weiser interpreted this result as evidence that programmers mentally formed slices when debugging. Later experiments would also show that programmers who focused on a bug's slice (vs. other code) would be more likely to correctly fix a bug [101]. Therefore as a cognitive design principle, a tool that automatically computes slices would theoretically support the natural mental processes of programmers in tasks like debugging.

4.2 Slicing Definitions and Techniques

In his subsequent paper [97], Weiser laid out a formal definition of a program slice, and he also described the first algorithm for computing slices in imperative languages. Slicing research continues to use much of Weiser's problem definition and terminology, so it's worth understanding his original formulation.

4.2.1 Weiser's Technique

Weiser's goal was to capture the concept of a "projection" of a program's behavior. For example, say a person only cared about some specific program element, like the value of a particular variable x. Then the x-projection of the program is the smallest subset of code needed just to compute x exactly the same as in the original program. Variables take on different values at different lines of code, so the projection must also be specified at a particular program location.

Therefore Weiser defined a slice of a program P in terms of a *slicing criterion*: a statement i and a set of variables $V = \{\overline{x}\}$. Given a criterion $C = \langle i, V \rangle$, then the slice S of P on C is the subset of code such that executing S produces exactly the same execution trace as P for the variables V at statement i.

```
1  fn main() {
2    let mut sum = 0;
3    let mut i = 1;
4    while i < 11 {
5      a(&mut sum, &mut i);
6    }
7  }
8
9  fn add(a: &mut i32, b: i32) {
10   *a = *a + b;
11 }
12 fn a(x: &mut i32, y: &mut i32) {
13   add(x, *y);
14   increment(y);
15 }
16
17 fn increment(z: &mut i32) {
18   add(z, 1);
19 }
```

Figure 4.1: Example of the difficulty of whole-program slicing used by Horwitz et al. [102]. The code snippets have been adapted from Algol-W to Rust.

For a simple imperative language (imagine C without pointers), statically computing S within a single function is relatively straightforward. Starting at the slicing criterion, the slicer works backwards following data and control dependencies. For example, if $C = \langle x := y + z, \{x\}\rangle$, then slicer would add y and z to the set of relevant variables, and transitively include all statements that affect y and z.

To handle programs with multiple functions, Weiser took a simple approach: imagine computing the slice of a variable y inside a function $f(x)\{\ldots\}$ where y depends on x. Then all calls to f (and their dependencies) are added to the slice.

4.2.2 Context-Sensitive Slicing

One subsequent line of work recognized that Weiser's simple approach to function-calls could be imprecise, i.e. code that should not be part of the slice was included by the slicer. For example, Horwitz et al. [102] showed that when computing the slice of `z` in `increment` in Figure 4.1, that Weiser's algorithm would include every line in the slice. However, the call to `add(x, *y)` on line 13 is irrelevant as it has no influence on any possible value of `z`.

This issue is what Horowitz et al. called "the calling-context problem", and what today is more concisely named *context-sensitivity*. The behavior of a function depends upon how it is called (the context), and so a whole-program analysis must be sensitive

to these contexts to produce a precise answer. Context-sensitivity affects any whole-program analysis, in particular pointer analysis (determining which memory locations a pointer can point-to at runtime).

The design of context-sensitive algorithms has occupied the static analysis research field for over three decades and counting. In short, the challenge is that the set of possible contexts can be quite large. Whaley and Lam [103] reported that some real-world applications have over 10^{14} contexts. Therefore a context-sensitive analysis must trade-off precision (how many contexts are considered) and performance (execution time and memory usage).

Horwitz et al. [102] provided the first algorithm for context-sensitive program slicing by construction of a *system dependence graph* (SDG). This data structure combined a function-call graph with a dataflow graph to encoding calling contexts with greater precision. Horwitz et al. did not provide any experimental results at the time to justify the precision claim, but it was supported by later findings [104].

Many more techniques have been invented for handling context-sensitivity, especially for tasks beyond slicing. For example, context-sensitive analysis has been reduced to problems like graph reachability [105] and inference on binary decision diagrams [103]. But SDGs seem to be the most common technique used in program slicers, e.g. the recent "dg" slicer for LLVM [106] uses SDGs.

4.2.3 Slicing Variations

Since Weiser's original formulation, researchers have created a number of slicing variations to address both human-centered and analysis-centered issues. Korel and Laski [107] introduced *dynamic* program slices (as distinct from Weiser's static slices) with the following motivation:

> "In debugging practice, however, we typically deal with a particular incorrect execution and, consequently, are interested in locating the cause of incorrectness (programming fault) of that execution. For this reason we are interested in a slice that preserves the program's behavior for a specific input, rather than that for the set of all inputs for which the program

terminates."

Korel and Laski believed that dynamic slices could be substantially smaller than static slices in some situations and therefore more useful for debugging. Later work continued to introduce yet more variations of program slicing:

- *Conditioned slicing* generalized dynamic slicing by allowing a slicing criterion to specify an arbitrary predicate on the sliced values, rather than an equality constraint [108].
- *Amorphous slicing* relaxed the syntactic constraint on slices to allow for simpler programs that are behaviorally equivalent but not a syntactic subset of the original code [109].
- *Thin slicing* relaxed the semantic constraint on slices to contain a fully executable program, allowing slices to contain only immediate (vs. transitive) influences on a particular sliced value [110].

The above list is naturally non-exhaustive, but it still represents the breadth of approaches to modifying Weiser's formulation of slicing to fit different programmer needs.

4.2.4 Studies on the Human Factors of Slicing

Despite decades of research into program slicing, and despite the origins of the field stemming from cognitive psychology, very little work has tried to analyze the human factors of slicing. One review found that "only 3 out of 111 papers on slicing based debugging techniques have considered issues with the use of the techniques in practice" [111]. Only two of those papers actually involve the use of a program slicer, so we focus on those two here.

The first study of slicing was conducted by Weiser and Lyle [12], which compared debugging times within a bespoke graphical slicing environment called Focus. They said:

> "We were unable to show that having a slicing tool helped reduce debugging time. If anything the trend was the other way: slicer users took a little longer to debug, possibly because they were playing with their new command.
>
> This experimental result was disappointing because it had seemed to follow from the mental use of slices that a slicing aid would be useful. Perhaps more learning time, or larger programs would have given a different result."

The only other study that I could find was conducted by Kusumoto et al. [13]. They had 34 undergraduate students debug six Pascal programs (about 40 LOC each), separated into one group with a slicer and one without. They found that students with the slicer took 41 minutes to find all the bugs, while students without the slicer took 49 minutes, and the different was statistically significant. However, Kusumoto et al. do not provide any details on how students used slices to accomplish the debugging task.

Another relevant success story is the Whyline tool developed by Ko and Myers [53]. Whyline used dynamic program slicing as a component of a debugging interface designed for the Alice game engine (and later for Java GUIs). In a user study, they found that users spent 8× less time on debugging with Whyline than a traditional interface. This striking result demonstrates that a slicing-based debugging tool can be very effective in certain contexts. However, Whyline combines slicing with many other ideas such as the explicit enumeration and presentation of "why"/"why not" questions, and so it is difficult to pinpoint the extent to which slicing specifically helped in their interface.

Ultimately, there is only light cognitive and empirical evidence that slices would be directly useful to programmers. Moreover, this evidence was generated on unrealistic tasks — programmers today do not work with small 40-line Pascal programs, but rather with sprawling codebases written in complex programming languages. As Parnin and Orso [111] claim, the conventional wisdom is that slicing doesn't work in these settings because "the sets of relevant statements identified are often still fairly

large."

Yet this explanation is not wholly satisfying. Binkley et al. [112] found in an empirical analysis of slice sizes in realistic codebases that the average slice is 1/3 the size of the entire program. Based on this result, they write:

> There are many programs that have sets of very small slices, making these programs highly amenable to slice-based approaches to comprehension. Even for programs with larger slices, any reduction in code size is going to be beneficial to comprehension, since it avoids the human wasting time on code not relevant to the comprehension question at hand. The fact that the study found very few extremely large slices indicates that optimism is not misplaced.

Notably, their results were generated by GrammaTech CodeSurfer, one of the few industrial-grade program slicers in existence. GrammaTech never wrote anything publicly about customers' experiences using CodeSurfer, so we do not know the extent to which programmers have found slicing useful (or not) in realistic settings.

4.3 The Seed of a New Slicer

After realizing how little evidence existed about the utility of slicing, my first instinct was to run a user study. I wanted to find an program slicer that programmers in a given language could easily adopt, meaning subject to the following constraints:

1. **Applicable to common language features:** the language being analyzed should support widely used features like pointers and in-place mutation.

2. **Zero configuration to run on existing code:** the analyzer must integrate with an existing language and existing unannotated programs. It must not require users to adopt a new language designed for information flow.

3. **No dynamic analysis:** to reduce integration challenges and costs, the analyzer must be purely static — no modifications to runtimes or binaries are needed.

4. **Modular over dependencies:** programs may not have source available for dependencies. The analyzer must have reasonable precision without whole-program analysis.

However, these constraints turned out to be quite demanding. There are a few static program slicers such as the `db` slicer for LLVM [106] and the Frama-C slicer for C [113]. But these slicers require whole-program analysis to handle function calls, which violates the fourth requirement of being modular over dependencies. For example, consider computing the slice of the return value in this C++ function:

```
1 // Copy elements 0 to max into a new vector
2 vector<int> copy_to(vector<int>& v, size_t max) {
3   vector<int> v2; size_t i = 0;
4   for (auto x(v.begin()); x != v.end(); ++x) {
5     if (i == max) { break; }
6     v2.push_back(*x); ++i;
7   }
8   return v2;
9 }
```

Here, a key part of the slice is that `v2` is influenced by `v`: (1) `push_back` mutates `v2` with `*x` as input, and (2) `x` points to data within `v`. But how could an analyzer statically deduce these facts? For C++, the answer is *by looking at function implementations*. The implementation of `push_back` mutates `v2`, and the implementation of `begin` returns a pointer to data in `v`.

Analyzing such implementations violates the modularity requirement, since these functions may only have their type signature available. In C++, given only a function's type signature, not much can be inferred about its behavior. But this observation raises an intriguing question: could a more sophisticated type system be leveraged to assist in a static analysis like program slicing?

4.3.1 Information Flow ≈ Slicing

After two decades of intense interest following the publication of Weiser's paper [97], research on program slicing started to decline. But at the same time, a new research topic started to gain steam: *information flow*. First described by Denning [114] in 1976, information flow describes whether one value can affect another. Information flow is commonly used in security research to determine whether a secure value (like a password) can affect an insecure value (like a string printed to the console). The detection of these leaks is called information flow control (IFC).

Information flow nominally seems somewhat different from program slicing, but both problems reduce to the same underlying question: does one part of a program depend on another? This reduction was formalized by Abadi et al. [16] who demonstrated that both information flow analysis and program slicing could be embedded within a *dependency calculus* that tracked which parts of a program a given variable depended on.

Program slicing is principally viewed as a program analysis problem: take a given language, and try to deduce some property of it. But research in IFC has gone a step beyond to design the language *around* the analysis. Languages like JFlow [115] and FlowCaml [116] added IFC to Java and OCaml, respectively, by changing the syntax and type system of the underlying language. For instance, in FlowCaml every type is annotated with a security level ℓ (e.g. int^{ℓ}) that represents the set of permissions associated with a value of that type.

Since these languages have information flow built-in to the type system, it should be relatively easily to implement a program slicer for these languages. However, IFC-oriented languages like JFlow and FlowCaml are research prototypes, and so there are very few users of these languages for me to study. Therefore I needed to look elsewhere to find a suitable language that could play host to a new slicer.

4.3.2 Ownership Types

Recall from the C++ example at the beginning of this section that the core challenges in practical program slicing are modularly analyzing pointers and mutation. One

of the main insights of this dissertation is these are the same challenges faced by a *memory safety* analysis. That is, a tool to statically identify memory safety violations must understand how a program uses memory, which entails understanding pointers into memory and mutations (including frees) of objects in memory.

Ownership types are a mechanism for analyzing memory safety in the type system that have been recently popularized in Rust. Ownership emerged from several intersecting lines of research on linear logic [117], class-based alias management [118], and region-based memory management [119]. The fundamental law of ownership is that data cannot be simultaneously aliased and mutated. Ownership-based type systems enforce this law by tracking which entities own which data, allowing ownership to be transferred between entities, and flagging ownership violations like mutating immutably-borrowed data.

To see the connection between ownership types and slicing, consider computing the slice of the return value in this Rust implementation of the same `copy_to` function:

```
1 fn copy_to(v: &Vec<i32>, max: usize) -> Vec<i32> {
2   let mut v2 = Vec::new();
3   for (i, x) in v.iter().enumerate() {
4     if i == max { break; }
5     v2.push(*x);
6   }
7   return v2;
8 }
```

Focus on the two methods `push` and `iter`. For a `Vec<i32>`, these methods have the following type signatures:

```
1 fn push(&mut self, value: i32);
2 fn iter<'a>(&'a self) -> Iter<'a, i32>;
```

To determine that `push` mutates `v2`, we leverage *mutability modifiers*. All references in Rust are either immutable (i.e. the type is `&T`) or mutable (the type is `&mut T`). Therefore `iter` does not mutate `v` because it takes `&self` as input (excepting interior mutability, discussed in Section 5.3.3), while `push` may mutate `v2` because it takes `&mut self` as input.

To determine that `x` points to `v`, we leverage *lifetimes*. All references in Rust are annotated with a lifetime, either explicitly (such as `'a`) or implicitly. Shared lifetimes indicate aliasing: because `&self` in `iter` has lifetime `'a`, and because the returned `Iter` structure shares that lifetime, then we can determine that `Iter` may contain pointers to `self`.

In sum, both the pointer and mutation problems can be approximated by using ownership types. This solution is consistent with all the criteria: Rust satisfies the criteria for a practical language. By using the type system, the analysis is static and modular. Finally, the annotations needed for this analysis must already be provided by the programmer to satisfy Rust's memory safety checks.

Based on this insight, I set out to design the infrastructure that would underlie a Rust program slicer, namely an engine for information flow analysis. As a part of the design, I formalized the theory of this insight so as to argue for the formal correctness of this approach.

Chapter 5

Modular Information Flow through Ownership

Based on the insight described in Section 4.3.2, I developed a theoretical foundation for analyzing information flow in ownership-based languages. I applied this theory to the design of Flowistry, a system for analyzing information flow in Rust programs. This chapter describes the theory, implementation, and evaluation of Flowistry.

5.1 Analysis

Inspired by the dependency calculus of Abadi et al. [16], our analysis represents information flow as a set of dependencies for each variable in a given function. The analysis is flow-sensitive, computing a different dependency set at each program location, and field-sensitive, distinguishing between dependencies for fields of a data structure.

While the analysis is implemented in and for Rust, our goal here is to provide a description of it that is both concise (for clarity of communication) and precise (for amenability to proof). We therefore base our description on Oxide [120], a formal model of Rust. At a high level, Oxide provides three ingredients:

1. A syntax of Rust-like programs with expressions e and types τ.

2. A type-checker, expressed with the judgment $\Sigma; \Delta; \Gamma \vdash e : \tau \Rightarrow \Gamma'$ using the contexts Γ for types and lifetimes, Δ for type variables, and Σ for global functions.

3. An interpreter, expressed by a small-step operational semantics with the judgment $\Sigma \vdash (\sigma; e) \rightarrow (\sigma'; e')$ using σ for a runtime stack.

We extend this model by assuming that each expression in a program is automatically labeled with a unique location ℓ. Then for a given expression e, our analysis computes the set of dependencies $\kappa ::= \{\overline{\ell}\}$. Because expressions have effects on persistent memory, we further compute a *dependency context* $\Theta ::= \{\overline{p \mapsto \kappa}\}$ from memory locations p to dependencies κ. The computation of information flow is intertwined with type-checking, represented as a modified type-checking judgment (additions highlighted in red):

$$\Sigma; \Delta; \Gamma; \Theta \vdash e_\ell : \tau \ \bullet \ \kappa \Rightarrow \Gamma'; \Theta'$$

This judgment is read as, "with type contexts Σ, Δ, Γ and dependency context Θ, e at location ℓ has type τ and dependencies κ, producing a new dependency context Θ'."

Oxide is a large language, so rather than covering every rule at once, we first focus on a few key rules that demonstrate the novel aspects of our system. We first lay the foundations for dealing with variables and mutation (Section 5.1.1), and then describe how we modularly analyze references (Section 5.1.2) and function calls (Section 5.1.3). Finally, the remaining rules are provided for completeness in Section 5.1.4.

5.1.1 Variables and Mutation

The core of Oxide is an imperative calculus with constants and variables. The abstract syntax for these features is below:

$$\begin{aligned}
&\text{Variable } x \quad \text{Number } n \\
\text{Path } q &::= \varepsilon \mid n.q \\
\text{Place } \pi &::= x.q \\
\text{Constant } c &::= () \mid n \mid \mathsf{true} \mid \mathsf{false} \\
\text{Base Type } \tau^{\mathrm{B}} &::= \mathsf{unit} \mid \mathsf{u32} \mid \mathsf{bool} \\
\text{Sized Type } \tau^{\mathrm{SI}} &::= \tau^{\mathrm{B}} \mid (\tau_1^{\mathrm{SI}}, \ldots, \tau_n^{\mathrm{SI}}) \mid \ldots \\
\text{Expression } e &::= c \mid \pi \mid \mathbf{let}\ x : \tau_a^{\mathrm{SI}} \ = \ e_1;\ e_2 \mid \\
&\qquad \pi := e \mid e_1;\ e_2 \mid \ldots
\end{aligned}$$

Constants are Oxide's atomic values and also the base-case for information flow. A constant's dependency is simply itself, expressed through the T-U32 rule:

T-U32

$$\frac{}{\Sigma; \Delta; \Gamma; \Theta \vdash n_\ell : \mathsf{u32} \bullet \{\ell\} \Rightarrow \Gamma; \Theta}$$

Variables and mutation are introduced through let-bindings and assignment expressions, respectively. For example, this (location-annotated) program mutates a field of a tuple:

$$\mathbf{let}\ t : (\mathsf{u32}, \mathsf{u32}) \ = \ (1_{\ell_1}, 2_{\ell_2});\ t.1 := 3_{\ell_3}$$

Here, t is a variable and $t.1$ is a *place*, or a description of a specific region in memory. For information flow, the key idea is that let-bindings introduce a set of places into Θ, and then assignment expressions change a place's dependencies within Θ. In the above example, after binding t, then Θ is:

$$\Theta = \{t, t.0, t.1 \mapsto \{\ell_1, \ell_2\}\}$$

After checking "$t.1 := 3$", then ℓ_3 is added to $\Theta(t)$ and $\Theta(t.1)$, but not $\Theta(t.0)$. This is because the values of t and $t.1$ have changed, but the value of $t.0$ has not. Formally, the let-binding rule is:

T-Let

$$\frac{\begin{array}{c}\Sigma; \Delta; \Gamma; \Theta \vdash e_1 : \tau_1^{\text{SI}} \bullet \kappa_1 \Rightarrow \Gamma_1; \Theta_1 \\ \Gamma; \Delta_1 \vdash \tau_1^{\text{SI}} \lesssim \tau_a^{\text{SI}} \Rightarrow \Gamma_1' \qquad \Theta_1' = \Theta_1[\forall \pi^{\square}[x] \,.\, \pi \mapsto \kappa_1] \\ \Sigma; \Delta; \text{gc-loans}(\Gamma_1', x : \tau_a^{\text{SI}}); \Theta_1' \vdash e_2 : \tau_2^{\text{SI}} \bullet \kappa_2 \Rightarrow \Gamma_2, x : \tau^{\text{SD}}; \Theta_2\end{array}}{\Sigma; \Delta; \Gamma; \Theta \vdash \textbf{let } x : \tau_a^{\text{SI}} = e_1;\ e_2 : \tau_2^{\text{SI}} \bullet \kappa_2 \Rightarrow \Gamma_2; \Theta_2}$$

Again, this rule (and many others) contain aspects of Oxide that are not essential for understanding information flow such as the subtyping judgment $\tau_1 \lesssim \tau_2$ or the metafunction gc-loans. For brevity we will not cover these aspects here, and instead refer the interested reader to Weiss et al. [120]. We have deemphasized (in grey) the judgments which are not important to understanding our information flow additions.

The key concept is the formula $\Theta_1[\forall \pi^{\square}[x] \,.\, \pi \mapsto \kappa_1]$. This introduces two shorthands: first, $\pi^{\square}[x]$ means "a place π with root variable x in a context $\pi^{\square}$", used to decompose a place. In T-Let, the update to Θ_1 happens for all places with a root variable x. Second, $\Theta_1[\pi \mapsto \kappa_1]$ means "set π to κ_1 in Θ_1". So this rule specifies that when checking e_2, all places within x are initialized to the dependencies κ_1 of e_1.

Next, the assignment expression rule is defined as updating all the *conflicts* of a place π:

T-Assign

$$\frac{\begin{array}{c}\Sigma; \Delta; \Gamma; \Theta \vdash e : \tau^{\text{SI}} \bullet \kappa \Rightarrow \Gamma_1; \Theta_1 \\ \Gamma_1(\pi) = \tau^{\text{SX}} \qquad (\tau^{\text{SX}} = \tau^{\text{SD}} \vee \Delta; \Gamma_1 \vdash_{\mathsf{uniq}} \pi \Rightarrow \{\ ^{\mathsf{uniq}}\pi\}) \\ \Delta; \Gamma_1 \vdash \tau^{\text{SI}} \lesssim \tau^{\text{SX}} \Rightarrow \Gamma' \qquad \Theta_2 = \Theta_1[\mathsf{update\text{-}conflicts}\,(\Theta_1, \pi, \kappa)]\end{array}}{\Sigma; \Delta; \Gamma; \Theta \vdash \pi := e : \mathsf{unit} \bullet \varnothing \Rightarrow \Gamma'[\pi \mapsto \tau^{\text{SI}}] \rhd \pi; \Theta_2}$$

If you conceptualize a type as a tree and a path as a node in that tree, then a node's conflicts are its ancestors and descendants (but not siblings). Semantically, conflicts are the set of places whose value change if a given place is mutated. Recall

from the previous example that $t.1$ conflicts with t and $t.1$, but not $t.0$. Formally, we say two places are disjoint ($\#$) or conflict ($\sqcap$) when:

$$x_1.q_1 \# x_2.q_2 \stackrel{\text{def}}{=} x_1 \neq x_2 \vee (\quad (q_1 \text{ is not a prefix of } q_2) \wedge (q_2 \text{ is not a prefix of } q_1))$$

$$\pi_1 \sqcap \pi_2 \stackrel{\text{def}}{=} \neg(\pi_1 \# \pi_2)$$

Then to update a place's conflicts in Θ, we define the metafunction $\mathsf{update\text{-}conflicts}$ to add κ to all conflicting places p'. (Note that this rule is actually defined over place *expressions* p, which are explained in the next subsection.)

$$\begin{aligned}&\mathsf{update\text{-}conflicts}\,(\Theta, p, \kappa) \stackrel{\text{def}}{=}\\ &\quad \forall p' \mapsto \kappa_{p'} \in \Theta_{\mathsf{cfl}} \;.\; p' \mapsto \kappa_{p'} \cup \kappa\\ &\quad \text{where } \Theta_{\mathsf{cfl}} = \{p' \mapsto \kappa_{p'} \in \Theta \mid p \sqcap p'\}\end{aligned}$$

Finally, the rule for reading places is simply to look up the place's dependencies in Θ:

T-Move

$$\frac{\Delta; \Gamma \vdash_{\mathsf{uniq}} \pi \Rightarrow \{\,^{\mathsf{uniq}}\pi\} \qquad \Gamma(\pi) = \tau^{\text{SI}} \quad \mathsf{noncopyable}_{\Sigma}\ \tau^{\text{SI}}}{\Sigma; \Delta; \Gamma; \Theta \vdash \pi : \tau^{\text{SI}} \;\bullet\; \Theta(\pi) \Rightarrow \Gamma[\pi \mapsto \tau^{\text{SI}^\dagger}]; \Theta}$$

5.1.2 References

Beyond concrete places in memory, Oxide also contains references that point to places. As in Rust, these references have both a lifetime (called a "provenance") and a mutability qualifier (called an "ownership qualifier"). Their syntax is:

$$
\begin{aligned}
&\text{Concrete Provenance } r \quad \text{Abstract Provenance } \varrho \\
&\text{Place Expression } p ::= x \mid *p \mid p.n \\
&\text{Provenance } \rho ::= \varrho \mid r \\
&\text{Ownership Qual. } \omega ::= \mathsf{shrd} \mid \mathsf{uniq} \\
&\text{Sized Type } \tau^{\text{SI}} ::= \ldots \mid \&\rho\,\omega\,\tau^{\text{XI}} \\
&\text{Expression } e ::= \ldots \mid \&r\,\omega\,p \mid p := e \mid \mathbf{letprov}\langle r \rangle\ e
\end{aligned}
$$

Provenances are created via a **letprov** expression, and references are created via a borrow expression $\&r\,\omega\,p$ that has an initial concrete provenance r (abstract provenances are just used for types of function parameters). References are used in conjunction with place expressions p that are places whose paths contain dereferences. For example, this program creates, reborrows, and mutates a reference:

$$
\begin{aligned}
&\mathbf{letprov}\langle r_1, r_2, r_3, r_4 \rangle \\
&\mathbf{let}\ x : (\mathsf{u32}, \mathsf{u32}) = (0, 0); \\
&\mathbf{let}\ y : \&r_2\,\mathsf{uniq}\,(\mathsf{u32}, \mathsf{u32}) = \&r_1\,\mathsf{uniq}\,x; \\
&\mathbf{let}\ z : \&r_4\,\mathsf{uniq}\,\mathsf{u32} = \&r_3\,\mathsf{uniq}\,(*y).1; \\
&*z := 1_\ell
\end{aligned}
$$

Consider the information flow induced by $*z := 1_\ell$. We need to compute all places that z could point-to, in this case $x.1$, so ℓ can be added to the conflicts of $x.1$. Essentially, we must perform a *pointer analysis* [121].

The key idea is that Oxide already does a pointer analysis! Performing one is an essential task in ensuring ownership-safety. All we have to do is extract the relevant information with Oxide's existing judgments. This is represented by the information

flow extension to the reference-mutation rule:

$$\frac{\Sigma;\Delta;\Gamma;\Theta \vdash e : \tau_n^{\text{SI}} \bullet \kappa \Rightarrow \Gamma_1;\Theta_1 \qquad \Delta;\Gamma_1 \vdash_{\mathsf{uniq}} p : \tau_o^{\text{SI}} \qquad \Delta;\Gamma_1 \vdash_{\mathsf{uniq}} p \Rightarrow \{\bar{l}\} \qquad \Delta;\Gamma_1 \vdash \tau_n^{\text{SI}} \lesssim \tau_o^{\text{SI}} \Rightarrow \Gamma' \qquad \Theta_2 = \Theta_1[\forall\, {}^{\omega}p' \in \{\bar{l}\} \,.\, \mathsf{update\text{-}conflicts}\,(\Theta_1, p', \kappa)]}{\Sigma;\Delta;\Gamma;\Theta \vdash p := e : \mathsf{unit} \bullet \varnothing \Rightarrow \Gamma' \rhd p;\Theta_2}$$

T-AssignDeref

Here, the important concept is Oxide's ownership safety judgment: $\Delta;\Gamma \vdash_{\omega} p \Rightarrow \{\bar{l}\}$, read as "in the contexts Δ and Γ, p can be used ω-ly and points to a loan in $\{\bar{l}\}$." A loan $l ::= {}^{\omega}p$ is a place expression with an ownership-qualifier. In Oxide, this judgment is used to ensure that a place is used safely at a given level of mutability. For instance, in the example at the top of this column, if $*z := 1$ was replaced with $x.1 := 1$, then this would violate ownership-safety because x is already borrowed by y and z.

In the example as written, the ownership-safety judgment for $*z$ would compute the loan set:

$$\{\bar{l}\} = \{\, {}^{\mathsf{uniq}}(*z), {}^{\mathsf{uniq}}(*y).1, {}^{\mathsf{uniq}}x.1\}$$

Note that $x.1$ is in the loan set of $*z$. That suggests the loan set can be used as a pointer analysis. The complete details of computing the loan set can be found in Weiss et al. [120, p. 12], but the summary for this example is:

1. Checking the borrow expression "$\&r_1\,\mathsf{uniq}\,x$" gets the loan set for x, which is just $\{{}^{\mathsf{uniq}}x\}$, and so sets $\Gamma(r_1) = \{{}^{\mathsf{uniq}}x\}$.

2. Checking the assignment "$y = \&r_1\,\mathsf{uniq}\,x$" requires that $\&r_1\,\mathsf{uniq}\,(\mathsf{u32},\mathsf{u32})$ is a subtype of $\&r_2\,\mathsf{uniq}\,(\mathsf{u32},\mathsf{u32})$, which requires that r_1 "outlives" r_2, denoted $r_1 :> r_2$.

3. The constraint $r_1 :> r_2$ adds $\Gamma(r_1)$ to $\Gamma(r_2)$, so $\Gamma(r_2) = \{{}^{\mathsf{uniq}}x\}$.

4. Checking "$\&r_3\,\mathsf{uniq}\,(*y).1$" gets the loan set for $(*y).1$, which is:

$$\{{}^{\mathsf{uniq}}p.1 \mid {}^{\mathsf{uniq}}p \in \Gamma(r_2)\} \cup \{{}^{\mathsf{uniq}}(*y).1\} = \{{}^{\mathsf{uniq}}x.1, {}^{\mathsf{uniq}}(*y).1\}$$

That is, the loans for r_2 are looked up in Γ (to get $\{x\}$), and then the additional projection _.1 is added on-top of each loan (to get $\{x.1\}$).

5. Then $\Gamma(r_4) = \Gamma(r_3)$ because $r_3 :> r_4$.

6. Finally, the loan set for $*z$ is:

$$\Gamma(r_4) \cup \{^{\mathsf{uniq}}(*z)\} = \{^{\mathsf{uniq}}x.1, {}^{\mathsf{uniq}}(*y).1, {}^{\mathsf{uniq}}(*z)\}$$

Applying this concept to the T-AssignDeref rule, we compute information flow for reference-mutation as: when mutating p with loans $\{\bar{l}\}$, add κ_e to all the conflicts for every loan $^{\mathsf{uniq}}p' \in \{\bar{l}\}$.

5.1.3 Function Calls

Finally, we examine how to modularly compute information flow through function calls, starting with syntax:

$$\begin{aligned} &\qquad\qquad \mathsf{Type\ Var}\ \alpha \quad \mathsf{Frame\ Var}\ \varphi \\ \mathsf{Expression}\ e &::= \ldots \mid f\langle\overline{\Phi}, \overline{\rho}, \overline{\tau}\rangle(\pi) \\ \mathsf{Global\ Entry}\ \varepsilon &::= \mathbf{fn}\ f\langle\overline{\varphi}, \overline{\varrho}, \overline{\alpha}, \overline{\varrho_1 : \varrho_2}\rangle(x : \tau_a^{\mathrm{SI}}) \to \tau_r^{\mathrm{SI}}\ \{\ e\ \} \\ \mathsf{Global\ Env.}\ \Sigma &::= \bullet \mid \Sigma, \varepsilon \end{aligned}$$

Oxide functions are parameterized by frame variables φ (for closures), abstract provenances ϱ (for provenance polymorphism), and type variables α (for type polymorphism). Unlike Oxide, we restrict to functions with one argument for simplicity in the formalism. Calling a function f requires an argument π and any type-level parameters Φ, ρ and τ.

The key question is: without inspecting its definition, what is the *most precise* assumption we can make about a function's information flow while still being sound? By "precise" we mean "if the analysis says there is a flow, then the flow actually exists", and by "sound" we mean "if a flow actually exists, then the analysis says

that flow exists." For example consider this program:

$$
\begin{aligned}
&\texttt{fn}\ \mathsf{f}\langle \varrho_1, \varrho_2\rangle(x : (\&\varrho_1\ \mathsf{uniq}\ \mathsf{u32}, \&\varrho_2\ \mathsf{shrd}\ \mathsf{u32}))\{\ ???\ \}\\
&\texttt{let}\ x : \mathsf{u32} = 1_{\ell_1};\ \texttt{let}\ y : \mathsf{u32} = 2_{\ell_2};\\
&\texttt{letprov}\langle r_1, r_2\rangle\ \texttt{let}\ t : (\&r_1\ \mathsf{uniq}\ \mathsf{u32}, \&r_2\ \mathsf{shrd}\ \mathsf{u32})\\
&\quad = (\&r_1\ \mathsf{uniq}\ x, \&r_2\ \mathsf{shrd}\ y);\\
&\mathsf{f}\langle r_1, r_2\rangle(t)
\end{aligned}
$$

First, what can $\mathsf{f}(t)$ mutate? Any data behind a shared reference is immutable, so only $*t.0$ could possibly be mutated, not $*t.1$. More generally, the argument's *transitive mutable references* must be assumed to be mutated.

Second, what are the inputs to the mutation of $*t.0$? This could theoretically be any possible value in the input, so both $*t.0$ and $*t.1$. More generally, every *transitively readable place* from the argument must be assumed to be to be an input to the mutation. So in this example, a modular analysis of the information flow from calling cp would add $\{\ell_1, \ell_2\}$ to $\Theta(x)$ but not $\Theta(y)$.

To formalize these concepts, we first need to describe the transitive references of a place. The $\omega\text{-}\mathsf{refs}(p, \tau)$ metafunction computes a place expression for every reference accessible from p. If $\omega = \mathsf{uniq}$ then this just includes unique references, otherwise it includes unique and shared ones.

$$
\begin{aligned}
\omega\text{-}\mathsf{refs}(p, \tau^{\mathrm{B}}) &= \varnothing\\
\omega\text{-}\mathsf{refs}(p, (\tau_1^{\mathrm{SI}}, \ldots, \tau_n^{\mathrm{SI}})) &= \textstyle\bigcup_i \omega\text{-}\mathsf{refs}(p.i, \tau_i^{\mathrm{SI}})\\
\omega\text{-}\mathsf{refs}(p, \&\rho\, \omega'\, \tau^{\mathrm{XI}}) &= \begin{cases} \{*p\} \cup \omega\text{-}\mathsf{refs}(*p, \tau^{\mathrm{XI}}) & \text{if } \omega \lesssim \omega' \\ \varnothing & \text{otherwise} \end{cases}
\end{aligned}
$$

Here, $\omega \lesssim \omega'$ means "a loan at ω can be used as a loan at ω'", defined as $\mathsf{uniq} \not\lesssim \mathsf{shrd}$ and $\omega \lesssim \omega'$ otherwise. Then $\omega\text{-}\mathsf{loans}(p, \tau, \Delta, \Gamma)$ can be defined as the set of concrete

places accessible from those transitive references:

$$\omega\text{-loans}(p, \tau, \Delta, \Gamma) \stackrel{\text{def}}{=} \bigcup_{p_1 \in \omega\text{-refs}(p,\tau)} \{p_2 \mid {}^{\omega}p_2 \in \{\bar{l}\}\} \qquad \text{where } \Delta; \Gamma \vdash_{\omega} p_1 \Rightarrow \{\bar{l}\}$$

Finally, the function application rule can be revised to include information flow as follows:

T-App

$$\frac{\begin{array}{c}\overline{\Sigma; \Delta; \Gamma \vdash \Phi} \qquad \overline{\Delta; \Gamma \vdash \rho} \qquad \overline{\Sigma; \Delta; \Gamma \vdash \tau^{\text{SI}}} \\ \Sigma(f) = \textbf{fn}\ f\langle\overline{\varphi}, \overline{\varrho}, \overline{\alpha}, \overline{\varrho_1 : \varrho_2}\rangle(x : \tau_a^{\text{SI}}) \to \tau_r^{\text{SI}}\ \{\ e\ \} \\ \Sigma; \Delta; \Gamma; \Theta \vdash \pi : \tau_a^{\text{SI}}[\overline{\Phi/\varphi}][\overline{\rho/\varrho}][\overline{\tau^{\text{SI}}/\alpha}] \bullet \kappa \Rightarrow \Gamma_1; \Theta \\ \Delta; \Gamma_1 \vdash \overline{\varrho_2[\rho/\varrho] :> \varrho_1[\rho/\varrho]} \Rightarrow \Gamma_2 \qquad \kappa_{\text{arg}} = \kappa \cup \bigcup_{p \in \text{shrd-loans}(\pi, \tau_a^{\text{SI}}, \Delta, \Gamma_2)} \Theta(p)\end{array}}{\begin{array}{c}\Theta' = \Theta[\forall p \in \text{uniq-loans}(\pi, \tau_a^{\text{SI}}, \Delta, \Gamma_2)\ . \\ \text{update-conflicts}\,(\Theta, p, \kappa_{\text{arg}})] \\ \hline \Sigma; \Delta; \Gamma; \Theta \vdash f\langle\overline{\Phi}, \overline{\rho}, \overline{\tau^{\text{SI}}}\rangle(\pi) : \tau_r^{\text{SI}}[\overline{\Phi/\varphi}][\overline{\rho/\varrho}][\overline{\tau^{\text{SI}}/\alpha}] \bullet \kappa_{\text{arg}} \Rightarrow \Gamma_2; \Theta'\end{array}}$$

The collective dependencies of the input π are collected into κ_{arg}, and then every unique reference is updated with κ_{arg}. Additionally, the function's return value is assumed to be influenced by any input, and so has dependencies κ_{arg}.

Note that this rule does not depend on the body e of the function f, only its type signature in Σ. This is the key to the modular approximation. Additionally, it means that this analysis can trivially handle higher-order functions. If f were a parameter to the function being analyzed, then no control-flow analysis is needed to guess its definition.

5.1.4 Additional rules

In Figure 5.1, we provide the remaining information flow rules for the expression forms not covered in the sections above. T-Tuple, T-Seq, T-LetProv are relatively

T-Tuple

$$\frac{\forall i \;.\; \Sigma; \Delta; \Gamma_{i-1}; \Theta_{i-1} \vdash \hat{e}_i : \tau_i^{\text{SI}} \;\bullet\; \kappa_i \Rightarrow \Gamma_i; \Theta_i \qquad \kappa = \{\ell\} \cup \bigcup_i \kappa_i}{\Sigma; \Delta; \Gamma_0; \Theta_0 \vdash (\hat{e}_1, \ldots, \hat{e}_n)_\ell : (\tau_1^{\text{SI}}, \ldots, \tau_n^{\text{SI}}) \;\bullet\; \kappa \Rightarrow \Gamma_n; \Theta_n}$$

T-Seq

$$\frac{\Sigma; \Delta; \Gamma; \Theta \vdash e_1 : \tau_1^{\text{SI}} \;\bullet\; \_ \Rightarrow \Gamma_1; \Theta_1 \qquad \Sigma; \Delta; \text{gc-loans}(\Gamma_1); \Theta_1 \vdash e_2 : \tau_2^{\text{SI}} \;\bullet\; \kappa_2 \Rightarrow \Gamma_2; \Theta_2}{\Sigma; \Delta; \Gamma; \Theta \vdash e_1; e_2 : \tau_2^{\text{SI}} \;\bullet\; \kappa_2 \Rightarrow \Gamma; \Theta_2}$$

T-LetProv

$$\frac{\Sigma; \Delta; \Gamma, r \mapsto \varnothing; \Theta \vdash e : \tau^{\text{SI}} \;\bullet\; \kappa \Rightarrow \Gamma', r \mapsto \{\bar{l}\}; \Theta'}{\Sigma; \Delta; \Gamma; \Theta \vdash \texttt{letprov}\langle r \rangle\ e : \tau^{\text{SI}} \;\bullet\; \kappa \Rightarrow \Gamma'; \Theta'}$$

T-Borrow

$$\frac{\Gamma(r) = \varnothing \qquad \Delta; \Gamma \vdash_\omega p \Rightarrow \{\bar{l}\} \qquad \Delta; \Gamma \vdash_\omega p : \tau^{\text{XI}} \qquad \kappa = \{\ell\} \cup \bigcup_{\omega p' \in \{\bar{l}\}} \Theta(p')}{\Sigma; \Delta; \Gamma; \Theta \vdash (\&r\,\omega\,p)_\ell : \&r\,\omega\,\tau^{\text{XI}} \;\bullet\; \kappa \Rightarrow \Gamma[r \mapsto \{\bar{l}\}]; \Theta}$$

T-Copy

$$\frac{\Delta; \Gamma \vdash_{\mathsf{shrd}} p \Rightarrow \{\bar{l}\} \qquad \Delta; \Gamma \vdash_{\mathsf{shrd}} p : \tau^{\text{SI}} \qquad \mathsf{copyable}_\Sigma \tau^{\text{SI}} \qquad \kappa = \bigcup_{\omega p' \in \{\bar{l}\}} \Theta(p')}{\Sigma; \Delta; \Gamma; \Theta \vdash p : \tau^{\text{SI}} \;\bullet\; \kappa \Rightarrow \Gamma; \Theta}$$

T-Branch

$$\frac{\begin{array}{c}\Sigma; \Delta; \Gamma; \Theta \vdash e_1 : \mathsf{bool} \;\bullet\; \kappa_1 \Rightarrow \Gamma_1; \Theta_1 \qquad \Sigma; \Delta; \Gamma_1; \Theta_1 \vdash e_2 : \tau_2^{\text{SI}} \;\bullet\; \kappa_2 \Rightarrow \Gamma_2; \Theta_2 \\ \Sigma; \Delta; \Gamma_1; \Theta_1 \vdash e_3 : \tau_3^{\text{SI}} \;\bullet\; \kappa_3 \Rightarrow \Gamma_3; \Theta_3 \qquad \tau^{\text{SI}} = \tau_2^{\text{SI}} \vee \tau^{\text{SI}} = \tau_3^{\text{SI}} \\ \Delta; \Gamma_2 \vdash \tau_2^{\text{SI}} \lesssim \tau^{\text{SI}} \Rightarrow \Gamma_2' \qquad \Delta; \Gamma_3 \vdash \tau_3^{\text{SI}} \lesssim \tau^{\text{SI}} \Rightarrow \Gamma_3' \\ \Gamma_2' \Cup \Gamma_3' = \Gamma' \qquad \Theta_2 \Cup \Theta_3 = \Theta' \qquad \Theta'' = \Theta'[\forall p \in \Theta' \setminus \Theta_1 \;.\; p \mapsto \Theta'(p) \cup \kappa_1]\end{array}}{\Sigma; \Delta; \Gamma; \Theta \vdash \mathsf{if}\ e_1\ \{\ e_2\ \}\ \mathsf{else}\ \{\ e_3\ \} : \tau^{\text{SI}} \;\bullet\; \kappa_1 \cup \kappa_2 \cup \kappa_3 \Rightarrow \Gamma'; \Theta''}$$

Figure 5.1: Additional information flow inference rules.

straightforward, so we focus on the remaining three rules.

First, T-Borrow: a borrow is unique in that its runtime value is a *place*, i.e. $\mathsf{ptr}\ \pi$. A borrow $\&\pi$ will always have value $\mathsf{ptr}\ \pi$, so it has no dependencies. But the borrow is a reborrow, i.e. of the form $\&*p$, then we need to include the dependencies of all places that p could point-to. Therefore we include $\Theta(p')$ for each loan p' in the loan set of p.

Next, T-Copy shows a read from an arbitrary place expression p. Unlike T-Move, we have to account for p referring to many possible memory locations. Again this is captured by the ownership-safety judgment $\vdash_{\mathsf{shrd}} p \Rightarrow \{\bar{l}\}$. Therefore the dependencies of p are the dependencies of any possibly read place, i.e. $\bigcup_{\pi'} \Theta(\pi')$.

Finally, T-Branch shows how to handle conditional execution. The κ is simple, as the value of an if-expression could depend on either branch, $\kappa_2 \cup \kappa_3$, along with the condition, κ_1. The more complex aspect is handling effects in Θ. The core idea is that if a place p could be mutated in either e_2 or e_3, then that place has a control dependency on e_1, and κ_1 should be part of the dependencies of p.

To encode this idea, we introduce two new metafunctions. First, $\Theta_1 \Cup \Theta_3$ represents the union distributed over like entries:

$$\Theta_2 \Cup \Theta_3 \stackrel{\text{def}}{=} \{p \mapsto \Theta_2(p) \cup \Theta_3(p) \mid p \in \Theta_2 \vee p \in \Theta_3\}$$

Next, we represent "could be mutated in e_2 or e_3" via $\Theta' \setminus \Theta_1$, similarly distributed over like entries:

$$\begin{aligned}\Theta' \setminus \Theta_1 \stackrel{\text{def}}{=} \{&p \mapsto \Theta'(p) \setminus \Theta_1(p) \\ &\mid (p \in \Theta' \vee p \in \Theta_1) \wedge \Theta'(p) \setminus \Theta_1(p) \neq \varnothing\}\end{aligned}$$

Then the rule says: after independently computing the contexts Θ_2 and Θ_3, compute a unioned context Θ'. Then for all places p that had new dependencies generated in e_2 or e_3, i.e. $p \in \Theta' \setminus \Theta_1$, add κ_1 to the dependencies of p.

5.2 Soundness

To characterize the correctness of our analysis, we seek to prove its *soundness*: if a true information flow exists in a program, then the analysis computes that flow. The standard soundness theorem for information flow systems is *noninterference* [122]. At a high level, noninterference states that for a given program and its dependencies, and for any two execution contexts, if the dependencies are equal between contexts, then the program will execute with the same observable behavior in both cases. For our analysis, we focus just on values produced by the program, instead of other behaviors like termination or timing.

To formally define noninterference within Oxide, we first need to explore Oxide's operational semantics. Oxide programs are executed in the context of a stack of frames that map variables to values:

$$\begin{aligned} \mathsf{Stack}\ \sigma &::= \bullet \mid \sigma \natural \varsigma \\ \mathsf{Stack\ Frame}\ \varsigma &::= \bullet \mid \varsigma, x \mapsto v \end{aligned}$$

For example, in the empty stack $\bullet$, the expression "$\texttt{let}\ x : \mathsf{u32} = 1;\ x := 2$" would first add $x \mapsto 1$ to the stack. Then executing $x := 2$ would update $\sigma(x) = 2$. More generally, we use the shorthand $\sigma(p)$ to mean "reduce p to a concrete location π, then look up the value of π in σ."

The key ingredient for noninterference is the equivalence of dependencies between stacks. That is, for two stacks σ_1 and σ_2 and a set of dependencies κ in a context Θ, we say those stacks are *the same up to* κ if all p with $\Theta(p) \subseteq \kappa$ are the same between stacks. Formally, the dependencies of κ and equivalence of heaps are defined as:

$$\begin{aligned} \mathsf{deps}(\Theta, \kappa) &\stackrel{\text{def}}{=} \{p \mid p \mapsto \kappa_p \in \Theta \wedge \kappa_p \subseteq \kappa\} \\ \sigma_1 \sim_P \sigma_2 &\stackrel{\text{def}}{=} \forall p \in P\ .\ \sigma_1(p) = \sigma_2(p) \\ \sigma_1 \sim_\kappa^\Theta \sigma_2 &\stackrel{\text{def}}{=} \sigma_1 \sim_{\mathsf{deps}(\Theta,\kappa)} \sigma_2 \end{aligned}$$

Then we define noninterference as follows:

Theorem 5.2.1 (Noninterference)**.** *Let e such that:*

$$\Sigma; \bullet; \Gamma; \Theta \vdash e : \tau \ \bullet \ \kappa \Rightarrow \Gamma'; \Theta'$$

For $i \in \{1, 2\}$, let σ_i such that:

$$\Sigma \vdash \sigma_i : \Gamma \quad \textit{and} \quad \Sigma \vdash (\sigma_i; e) \xrightarrow{*} (\sigma_i'; v_i)$$

Then:

(a) $\sigma_1 \sim_{\kappa}^{\Theta} \sigma_2 \implies v_1 = v_2$

(b) $\forall p \mapsto \kappa_p \in \Theta' \ . \ \sigma_1 \sim_{\kappa_p}^{\Theta} \sigma_2 \implies \sigma_1'(p) = \sigma_2'(p)$

This theorem states that given a well-typed expression e and corresponding stacks σ_i, then its output v_i should be equal if the expression's dependencies κ are initially equal. Moreover, for any place expression p, if its dependencies in the output context Θ' are initially equal then the stack value will be the same after execution.

Note that the context Δ is required to be empty because an expression e can only evaluate if it does not contain abstract type or provenance variables. The judgment $\Sigma \vdash \sigma_i : \Gamma$ means "the stack σ_i is well-typed under Σ and Γ". That is, for all places π in Γ, then $\pi \in \sigma$ and $\sigma(\pi)$ has type $\Gamma(\pi)$.

5.2.1 Proofs

The proof of non-interference relies on a few key lemmas about the semantics of Oxide. We start by defining and proving these lemmas (Section 5.2.1), and then proceed to prove noninterference (Section 5.2.1).

Lemmas

Lemma 1 (Mutating a place also mutates its conflicts)**.** *Let:*

- $\pi_{\mathsf{mut}} = \pi_{\mathsf{mut}}^{\square}[x]$, σ *where* $\sigma \vdash \pi_{\mathsf{mut}}^{\square} \times x \Downarrow \mathcal{V}$
- v *be a value and* $\sigma' = \sigma[x \mapsto \mathcal{V}[v]]$

- $\pi_{\mathsf{any}} \in \sigma$

Then $\sigma(\pi_{\mathsf{any}}) \neq \sigma'(\pi_{\mathsf{any}}) \implies \pi_{\mathsf{mut}} \sqcap \pi_{\mathsf{any}}$.

Proof.

1. Assume $\sigma(\pi_{\mathsf{any}}) \neq \sigma'(\pi_{\mathsf{any}})$. Want to show $\pi_{\mathsf{mut}} \sqcap \pi_{\mathsf{any}}$.
2. Because only x is assigned, then $\pi_{\mathsf{any}} = \pi^{\square}_{\mathsf{any}}[x]$.
3. Assume for sake of contradiction that $\pi_{\mathsf{mut}} \# \pi_{\mathsf{any}}$. Let q be the shared path and $n_{\mathsf{mut}}, n_{\mathsf{any}}$ be the split, i.e. $\pi_{\mathsf{mut}} = x.q.n_{\mathsf{mut}}.q_{\mathsf{mut}}$ and $\pi_{\mathsf{any}} = x.q.n_{\mathsf{any}}.q_{\mathsf{any}}$.
4. By ER-Projection, then

$$\sigma \vdash x.q.n_{\mathsf{mut}} \Downarrow \mathcal{V}[(v_0, \ldots, v_{n_{\mathsf{any}}}, \ldots, \square_{n_{\mathsf{mut}}}, \ldots, v_n)]$$

5. $v_{n_{\mathsf{any}}} = \sigma(x.q.n_{\mathsf{any}})$ by induction on the derivation of $\mathcal{V}$.
6. Therefore $\sigma'(x.q.n_{\mathsf{mut}}) = v_{n_{\mathsf{any}}} = \sigma(x.q.n_{\mathsf{mut}})$.
7. This is a contradiction with $\sigma(\pi_{\mathsf{any}}) \neq \sigma'(\pi_{\mathsf{any}})$, therefore $\pi_{\mathsf{mut}} \sqcap \pi_{\mathsf{any}}$.

□

Lemma 2 (A place expression's loan set contains the place it points-to at runtime.)**.** *Let:*

- σ, Σ, Γ *where* $\Sigma \vdash \sigma : \Gamma$
- p_{mut} *where* $\bullet; \Gamma \vdash_{\mathsf{uniq}} p_{\mathsf{mut}} \Rightarrow \{\bar{l}\}$ *and* $\sigma \vdash p_{\mathsf{mut}} \Downarrow \pi_{\mathsf{mut}}$
- p_{any} *where* $\sigma \vdash p_{\mathsf{any}} \Downarrow \pi_{\mathsf{any}}$

Then $\pi_{\mathsf{any}} \sqcap \pi_{\mathsf{mut}} \implies \exists\, ^{\mathsf{uniq}} p_{\mathsf{loan}} \in \{\bar{l}\} \;.\; p_{\mathsf{any}} \sqcap p_{\mathsf{loan}}$

Proof. Proof by induction on the derivation of
$\bullet; \Gamma \vdash_{\mathsf{uniq}} p_{\mathsf{mut}} \Rightarrow \{\bar{l}\}$

(a) O-SafePlace:

1. If $p_{\mathsf{any}} \neq \pi_{\mathsf{any}}$ then $\exists r \;.\; \pi_{\mathsf{any}} \in \Gamma(r)$. But the first premise of O-SafePlace, it must be the case that $\pi_{\mathsf{any}} \# \pi_{\mathsf{mut}}$, a contradiction. Therefore $p_{\mathsf{any}} = \pi_{\mathsf{any}}$.
2. Then by the conclusion of O-SafePlace, $\{\bar{l}\} = \{\, {}^{\mathsf{uniq}}\pi_{\mathsf{mut}} \,\}$. Then the theorem holds for $p_{\mathsf{loan}} = \pi_{\mathsf{mut}}$.

(b) O-Deref:

1. Let $p_{\mathsf{mut}} = p^{\square}_{\mathsf{mut}}[*\pi_{\mathsf{ptr}}]$ and $\Gamma(\pi_{\mathsf{ptr}}) = \&r\,\mathsf{uniq}\,\tau$.
2. By extension of Lemma E.6 [2019, p. 47],

$$\exists\, {}^{\mathsf{uniq}} p_i \in \Gamma(r) \;.\; \sigma \vdash p^{\square}_{\mathsf{mut}}[p_i] \Downarrow \pi_{\mathsf{mut}}$$

3. By the inductive hypothesis on

$$\bullet; \Gamma \vdash_{\mathsf{uniq}} p^{\square}_{\mathsf{mut}}[p_i] \Rightarrow \{\overline{{}^{\mathsf{uniq}} p'_i}\}$$

   then:

$$\exists\, {}^{\mathsf{uniq}} p_{\mathsf{loan}} \in \{\overline{{}^{\mathsf{uniq}} p'_i}\} \;.\; p_{\mathsf{any}} \sqcap p_{\mathsf{loan}}$$

4. Because $\{\overline{{}^{\mathsf{uniq}} p'_i}\} \subseteq \{\bar{l}\}$, then the theorem holds.

(c) O-DerefAbs: does not apply since $\Delta = \bullet$.

□

Lemma 3 (A function only mutates unique references in its argument)**.** *Let:*

- $\Gamma, \pi_{\mathsf{arg}}, \sigma$ *where* $\Gamma(\pi_{\mathsf{arg}}) = \tau^{\mathrm{SI}}$ *and* $\Sigma \vdash \sigma : \Gamma$
- f *where* $\Sigma \vdash (\sigma; f(\pi_{\mathsf{arg}})) \xrightarrow{*} (\sigma'; \_)$
- $\sigma'' = \sigma'[\forall p_{\mathsf{loan}} \in \mathsf{uniq\text{-}loans}(\pi_{\mathsf{arg}}, \tau^{\mathrm{SI}}, \bullet, \Gamma) \;.\; p_{\mathsf{loan}} \mapsto \sigma(p_{\mathsf{loan}})]$

Then $\sigma = \sigma''$.

Proof.

1. Let:
$$\Sigma(f) = \textsf{fn}\ f\langle\overline{\varphi}, \overline{\varrho}, \overline{\alpha}, \overline{\varrho_1 : \varrho_2}\rangle(x : \tau_a^{\text{SI}}) \to \tau_r^{\text{SI}}\ \{\ e\ \}$$

2. By E-App, E-EvalCtx, and E-Framed:
$$\Sigma \vdash (\sigma; f(\pi)) \to (\sigma \natural x \mapsto \sigma(\pi);\ \textsf{framed}\ e) \xrightarrow{*} (\sigma' \natural \varsigma;\ \textsf{framed}\ v) \to (\sigma';\ v)$$

3. By inspection of the operational semantics, the only rule that could modify σ (as apart from ς) is E-Assign. Assume that $p_{\text{mut}} := e'$ is executed under stack $\sigma_{\text{mut}} \natural \varsigma$ during $f(\pi_{\text{arg}})$ where $\sigma_{\text{mut}} \vdash p_{\text{mut}} \Downarrow \pi_{\text{mut}}$.

4. By inspection of the operational semantics, the only way to create a pointer is E-Borrow on a place p. The only places in σ_{mut} that are accessible from e are $\textsf{shrd-loans}(\pi_{\text{arg}}, \tau^{\text{SI}}, \bullet, \Gamma)$, so it must be that $p_{\text{mut}} = p^{\square}[p_{\text{loan}}]$ and $p_{\text{loan}} \in \textsf{shrd-loans}(\pi_{\text{arg}}, \tau^{\text{SI}}, \bullet, \Gamma)$.

5. Because e is well-typed under $x : \tau_a^{\text{SI}}$, then any shared references in x cannot be mutated: T-AssignDeref requires $\vdash_{\textsf{uniq}} p_{\text{mut}} \Rightarrow \{\bar{l}\}$, which by O-Deref requires that the reference under p_{mut} has type $\&r\ \textsf{uniq}\ \tau$. Therefore $p_{\text{loan}} \in \textsf{uniq-loans}(\pi_{\text{arg}}, \tau^{\text{SI}}, \bullet, \Gamma)$.

6. Hence, all mutated places in σ' are projections of a p_{loan}. Therefore $\sigma'[\forall p_{\text{loan}} \in \textsf{uniq-loans}(\pi_{\text{arg}}, \tau^{\text{SI}}, \bullet, \Gamma)\ .\ p_{\text{loan}} \mapsto \sigma(p_{\text{loan}})]$ reverts all possible mutations, and $\sigma = \sigma''$.

□

Lemma 4 (A function's effects are only influenced by its argument.)**.** *Let:*

- Γ, π, σ_i *where* $\Gamma(\pi_{\text{arg}}) = \tau^{\text{SI}}$ *and* $i \in \{1, 2\}$ *and* $\Sigma \vdash \sigma_i : \Gamma$
- f *where* $\Sigma \vdash (\sigma_i; f(\pi_{\text{arg}})) \xrightarrow{*} (\sigma_i'; v_i)$
- $P = \{\pi_{\text{arg}}\} \cup \textsf{shrd-loans}(\pi_{\text{arg}}, \tau^{\text{SI}}, \bullet, \Gamma)$

$\sigma_1 \sim_P \sigma_2 \implies \sigma_1' \sim_P \sigma_2' \wedge v_1 = v_2$

Proof.

1. As with the proof for Lemma 3, have $(\sigma_i, f(\pi_{\mathsf{arg}})) \xrightarrow{*} (\sigma_i'; v_i)$.
2. By inspection of the operational semantics, the only rule that could read σ is E-Copy via the premise $(\sigma_{\mathsf{read}}; p_{\mathsf{read}}) \to (\sigma_{\mathsf{read}}; v)$ where $\sigma_{\mathsf{read}} \vdash p_{\mathsf{read}} \Downarrow \pi_{\mathsf{read}} \mapsto v$.
3. As with the proof for Lemma 3, the only possible value of $p_{\mathsf{read}} = p^{\square}[p_{\mathsf{loan}}]$ where $p_{\mathsf{loan}} \in \mathsf{shrd\text{-}loans}(\pi_{\mathsf{arg}}, \tau^{\mathrm{SI}}, \bullet, \Gamma)$.
4. Because $p_{\mathsf{loan}} \in P$ and $\sigma_1 \sim_P \sigma_2$ then $\sigma_1(p_{\mathsf{loan}}) = \sigma_2(p_{\mathsf{loan}})$.
5. Similarly because $\pi_{\mathsf{arg}} \in P$ then $\sigma_1(\pi_{\mathsf{arg}}) = \sigma_2(\pi_{\mathsf{arg}})$.
6. Therefore every readable input for f is equal, and for the same function body e, the evaluation should result in the same effects on σ and same output v. Hence, $\sigma_1' \sim_P \sigma_2'$ and $v_1 = v_2$.

□

Noninterference

Theorem 5.2.1 (Noninterference)**.** *Let e such that:*

$$\Sigma; \bullet; \Gamma; \Theta \vdash e : \tau \ \bullet \ \kappa \Rightarrow \Gamma'; \Theta'$$

For $i \in \{1, 2\}$, let σ_i such that:

$$\Sigma \vdash \sigma_i : \Gamma \quad \textit{and} \quad \Sigma \vdash (\sigma_i; e) \xrightarrow{*} (\sigma_i'; v_i)$$

Then:

(a) $\sigma_1 \sim_{\kappa}^{\Theta} \sigma_2 \implies v_1 = v_2$

(b) $\forall p \mapsto \kappa_p \in \Theta' \ . \ \sigma_1 \sim_{\kappa_p}^{\Theta} \sigma_2 \implies \sigma_1'(p) = \sigma_2'(p)$

Proof. Proof by induction over the derivation of $e : \tau$.

- T-u32: $e = n_\ell$ where $\kappa = \{\ell\}$. (a) is trivial because n always evaluates to n, and (b) is trivial because n has no effects.

- T-Move: $e = \pi$ where $\kappa = \Theta(\pi)$. (b) is trivial because π has no effects, so we focus on (a): $\sigma_1 \sim^{\Theta}_{\Theta(\pi)} \sigma_2 \implies v_1 = v_2$

  1. By the definition of equivalence of stacks, $\Theta(\pi) \subseteq \Theta(\pi) \implies \sigma_1(\pi) = \sigma_2(\pi)$.
  2. By E-Move, $v_i = \sigma_i(\pi)$.
  3. Therefore $v_1 = v_2$.

- T-Copy: the proof is the same as for T-Move.

- T-Seq: $e = e_1; e_2$ where $\kappa = \kappa_2$.

  (a) $\sigma_1 \sim^{\Theta}_{\kappa_2} \sigma_2 \implies v_1 = v_2$

  1. By E-EvalCtx, $\Sigma \vdash (\sigma_i; e_1) \xrightarrow{*} (\sigma_i^{e_1}; v_i^{e_1})$.
  2. By E-Seq, $\Sigma \vdash (\sigma_i^{e_1}; (v_i^{e_1}; e_2)) \to (\sigma_i^{e_1}; e_2) \xrightarrow{*} (\sigma_i^{e_2}; v_i^{e_2})$. WTS $v_1^{e_2} = v_2^{e_2}$.
  3. By the IH for e_2, $v_1^{e_2} = v_2^{e_2}$ if $\sigma_1^{e_1} \sim^{\Theta_1}_{\kappa_2} \sigma_2^{e_1}$.
  4. Let $\pi \mapsto \kappa_\pi \in \Theta_1$. WTS $\kappa_\pi \subseteq \kappa_2 \implies \sigma_1^{e_1}(\pi) = \sigma_2^{e_1}(\pi)$.
  5. By the IH for e_1, $\sigma_1^{e_1}(\pi) = \sigma_2^{e_1}(\pi)$ if $\sigma_1 \sim^{\Theta}_{\kappa_\pi} \sigma_2$.
  6. Let $\pi' \mapsto \kappa_{\pi'} \in \Theta_1$ such that $\kappa_{\pi'} \subseteq \kappa_\pi$. WTS $\sigma_1(\pi') = \sigma_2(\pi')$.
  7. By assumption, because $\kappa_\pi \subseteq \kappa_2 \wedge \kappa_{\pi'} \subseteq \kappa_\pi$, then $\kappa_{\pi'} \subseteq \kappa_2$.
  8. By assumption, $\sigma_1(\pi') = \sigma_2(\pi')$.

  (b) $\forall \pi \mapsto \kappa_\pi \in \Theta_2 \,.\, \sigma_1 \sim^{\Theta}_{\kappa_\pi} \sigma_2 \implies \sigma_1'(\pi) = \sigma_2'(\pi)$

  1. By the IH for e_2, $\sigma_1'(\pi) = \sigma_2'(\pi)$ if $\sigma_1^{e_1} \sim^{\Theta_1}_{\kappa_\pi} \sigma_2^{e_1}$.
  2. The proof that $\sigma_1^{e_1} \sim^{\Theta_1}_{\kappa_\pi} \sigma_2^{e_1}$ follows similarly as above.

- T-Let: $e =$ "**let** $x : \tau_a^{\text{SI}} = e_1;\ e_2$". The proof of (b) follows from the proof for T-Seq. We focus on (a): $\sigma_1 \sim^{\Theta}_{\kappa_2} \sigma_2 \implies v_1 = v_2$

  1. By E-EvalCtx, E-Let, E-EvalCtx, E-Shift:
     $\Sigma \vdash (\sigma_i; \mathsf{let}\ x : \tau_a^{\text{SI}} = e_1; e_2) \xrightarrow{*} (\sigma_i^{e_1}; \mathsf{let}\ x : \tau_a^{\text{SI}} = v_{e_1}^i;\ e_2) \to (\sigma_i^{e_1}, x \mapsto v_{e_1}^i;\ \mathsf{shift}\ e_2) \xrightarrow{*}$
     $(\sigma_i', x \mapsto \_;\ \mathsf{shift}\ v_i^{e_2}) \to (\sigma_i';\ v_i^{e_2})$.

2. Let $\sigma_i^x = \sigma_i^{e_1}, x \mapsto v_i^{e_1}$. By IH (a) for e_2, if $\sigma_1^x \sim_{\kappa_2}^{\Theta_1'} \sigma_2^x$ then $v_1^{e_2} = v_2^{e_2}$.
3. Let π such that $\Theta_1'(\pi) \subseteq \kappa_2$. WTS $\sigma_1^x(\pi) = \sigma_2^x(\pi)$.
4. If $\Theta_1'(\pi) \subseteq \Theta(\pi)$, then $\sigma_1^x(\pi) = \sigma_2^x(\pi)$ is true by assumption.
5. Otherwise must be $\pi = \pi^{\square}[x]$ and $\kappa_q \subseteq \Theta_1'(\pi)$.
6. By IH (a) for e_1, if $\sigma_1 \sim_{\kappa_1}^{\Theta} \sigma_2$ then $v_1^{e_1} = v_2^{e_1}$.
7. Because $\sigma_1 \sim_{\kappa_2}^{\Theta} \sigma_2$ and $\kappa_1 \subseteq \kappa_2$ by (3), then $\sigma_1 \sim_{\kappa_1}^{\Theta} \sigma_2$ and $v_1^{e_1} = v_2^{e_1}$.
8. Therefore $\sigma_1^x(\pi) = \sigma_2^x(\pi)$.

- T-Assign: $e =$ "$\pi_{\mathsf{mut}} := e$". Because $v_i = ()$ then (a) is trivial, so we focus on (b): $\forall \pi_{\mathsf{any}} \mapsto \kappa_{\mathsf{any}} \in \Theta_1' \;.\; \sigma_1 \sim_{\kappa_{\mathsf{any}}}^{\Theta} \sigma_2 \implies \sigma_1'(\pi_{\mathsf{any}}) = \sigma_2'(\pi_{\mathsf{any}})$

  1. By E-EvalCtx, $\Sigma \vdash (\sigma_i; e) \xrightarrow{*} (\sigma_i^e; v_i^e)$. By IH (b) on e, then $\sigma_1^e(\pi_{\mathsf{any}}) = \sigma_2^e(\pi_{\mathsf{any}})$.
  2. By E-Assign, $\Sigma \vdash (\sigma_i^e;\ \pi_{\mathsf{mut}} := v_i^e)$ $\to (\sigma_i^e[x \mapsto \mathcal{V}_i[v_i^e]];\ ())$ where $\pi_{\mathsf{mut}} = \pi_{\mathsf{mut}}^{\square}[x]$ and $\sigma_i^e \vdash \pi_{\mathsf{mut}} \Downarrow \mathcal{V}_i$.
  3. By Lemma 1, if $\sigma_i^e(\pi_{\mathsf{any}}) \neq \sigma_i'(\pi_{\mathsf{any}})$ then $\pi_{\mathsf{any}} \sqcap \pi_{\mathsf{mut}}$.
  4. By T-Assign, because $\pi_{\mathsf{any}} \sqcap \pi_{\mathsf{mut}}$ then $\Theta_1'(\pi_{\mathsf{any}}) = \Theta_1(\pi_{\mathsf{any}}) \cup \kappa_e$.
  5. Because $\kappa_e \subseteq \Theta_1'(\pi_{\mathsf{any}})$, then $\sigma_1 \sim_{\kappa_e}^{\Theta} \sigma_2$.
  6. By IH (a) on e, then $v_1^e = v_2^e$.
  7. Therefore $\sigma_1'(\pi_{\mathsf{any}}) = \sigma_2'(\pi_{\mathsf{any}})$.

- T-AssignDeref: $e =$ "$p_{\mathsf{mut}} := e$". Like T-Assign, we focus on (b): $\forall \pi_{\mathsf{any}} \mapsto \kappa_{\mathsf{any}} \in \Theta_1' \;.\; \sigma_1 \sim_{\kappa_{\mathsf{any}}}^{\Theta} \sigma_2 \implies \sigma_1'(\pi_{\mathsf{any}}) = \sigma_2'(\pi_{\mathsf{any}})$

  1. By E-EvalCtx and E-Assign, $\Sigma \vdash (\sigma_i; p_{\mathsf{mut}} := e) \xrightarrow{*} (\sigma_i^e[x_i \mapsto \mathcal{V}_i[v_i^e]]; ())$ where $\sigma_i^e \vdash p_{\mathsf{mut}} \Downarrow \mathcal{V}_i \times \pi_{\mathsf{mut},i}^{\square}[x_i]$.
  2. By IH (b) on e, $\sigma_1^e(\pi_{\mathsf{any}}) = \sigma_2^e(\pi_{\mathsf{any}})$.
  3. By Lemma 1, only consider the case where $\pi_{\mathsf{mut},i} \sqcap \pi_{\mathsf{any}}$.

4. By Lemma 2, because $\pi_{\mathsf{mut},i} \sqcap \pi_{\mathsf{any}}$ then
   $\exists^{\mathsf{uniq}} p_{\mathsf{loan}} \in \{\bar{l}\} \;.\; \pi_{\mathsf{any}} \sqcap p_{\mathsf{loan}}$.
5. By T-AssignDeref, then $\kappa \subseteq \Theta'_1(\pi_{\mathsf{any}})$, and $v_1^e = v_2^e$, and $\sigma'_1(\pi_{\mathsf{any}}) = \sigma'_2(\pi_{\mathsf{any}})$ as in the proof for T-Assign.

- T-App: $e =$ "$f(\pi_{\mathsf{arg}})$" where $\kappa = \kappa_{\mathsf{arg}}$.

  (a) $\sigma_1 \sim^{\Theta}_{\kappa_{\mathsf{arg}}} \sigma_2 \implies v_1 = v_2$

  1. Let $p \in \{\pi_{\mathsf{arg}}\} \cup \mathsf{shrd\text{-}loans}(\pi_{\mathsf{arg}}, \tau_a^{\mathrm{SI}}, \bullet, \Gamma_2)$.
     By Lemma 4, if $\sigma_1(p) = \sigma_2(p)$, then $v_1 = v_2$.
  2. By T-App, $\Theta(p) \subseteq \kappa_{\mathsf{arg}}$. Therefore $\sigma_1(p) = \sigma_2(p)$ and hence $v_1 = v_2$.

  (b) $\forall \pi_{\mathsf{any}} \mapsto \kappa_{\mathsf{any}} \in \Theta_2 \;.\; \sigma_1 \sim^{\Theta}_{\kappa_{\mathsf{any}}} \sigma_2 \implies \sigma'_1(\pi_{\mathsf{any}}) = \sigma'_2(\pi_{\mathsf{any}})$

  1. By Lemma 3 and Lemma 1, if $\sigma_i(\pi_{\mathsf{any}}) \neq \sigma'_i(\pi_{\mathsf{any}})$ then:
     $\exists p_{\mathsf{loan}} \in \mathsf{uniq\text{-}loans}(\pi_{\mathsf{arg}}, \tau^{\mathrm{SI}}, \bullet, \Gamma_2) \;.\; (\sigma_i \vdash p_{\mathsf{loan}} \Downarrow \pi_{\mathsf{loan},i}) \wedge \pi_{\mathsf{any}} \sqcap \pi_{\mathsf{loan},i}$.
  2. By T-App, then $\kappa_{\mathsf{arg}} \subseteq \kappa_{\mathsf{any}}$.
  3. By Lemma 4, then $\sigma'_1(\pi_{\mathsf{any}}) = \sigma'_2(\pi_{\mathsf{any}})$.

□

5.3 Implementation

Our formal model provides a sound theoretical basis for analyzing information flow in Oxide. However, Rust is a more complex language than Oxide, and the Rust compiler uses many intermediate representations beyond its surface syntax. Therefore in this section, we describe the key details of how our system, Flowistry, bridges theory to practice. Specifically:

1. Rust computes lifetime-related information on a control-flow graph (CFG) program representation, not the high-level AST. So we translate our analysis to work for CFGs (Section 5.3.1).
2. Rust does not compute the loan set for lifetimes directly like in Oxide. So we must reconstruct the loan sets given the information exported by Rust (Section 5.3.2).

```
fn get_count(
  h: &mut HashMap<String, u32>,
  k: String
) -> u32 {
  if !h.contains_key(&k) {
    h.insert(k, 0); 0
  } else {
    *h.get(&k).unwrap()
  }
}
```

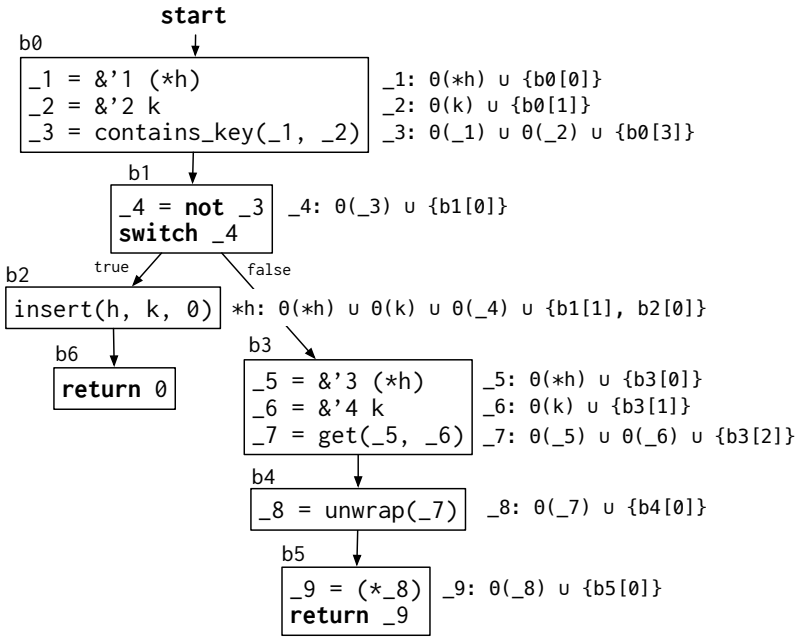

Figure 5.2: Example of how Flowistry computes information flow. On the top is a Rust function `get_count` that finds a value in a hash map for a given key, and inserts 0 if none exists. On the bottom `get_count` is lowered into Rust's MIR control-flow graph, annotated with information flow. Each rectangle is a basic block, named at the top. Arrows indicate control flow (panics omitted).
Beside each instruction is the result of the information flow analysis, which maps place expressions to locations in the CFG (akin to Θ in Section 5.1). For example, the `insert` function call adds dependencies to `*h` because it is assumed to be mutated, since it is a mutable reference. Additionally, the `switch` instructions and `_4` variable are added as dependencies to `h` because the call to `insert` is control-dependent on the switch.

3. Rust contains escape hatches for ownership-unsafe code that cannot be analyzed using our analysis. So we describe the situations in which our analysis is unsound for Rust programs (Section 5.3.3).

5.3.1 Analyzing Control-Flow Graphs

The Rust compiler lowers programs into a "mid-level representation", or MIR, that represents programs as a control-flow graph. Essentially, expressions are flattened into sequences of instructions (basic blocks) which terminate in instructions that can jump to other blocks, like a branch or function call. Figure 5.2 shows an example CFG and its information flow.

To implement the modular information flow analysis for MIR, we reused standard static analysis techniques for CFGs, i.e., a flow-sensitive, forward dataflow analysis pass where:

- At each instruction, we maintain a mapping from place expressions to a set of locations in the CFG on which the place is dependent, comparable to Θ in Section 5.1.
- A transfer function updates Θ for each instruction, e.g. $p := e$ follows the same rules as in T-AssignDeref by adding the dependencies of e to all conflicts of aliases of p.
- The input Θ^{in} to a basic block is the join of each of the output Θ_i^{out} for each incoming edge, i.e. $\Theta^{\mathsf{in}} = \bigvee_i \Theta_i^{\mathsf{out}}$. The join operation is key-wise set union, or more precisely:

$$\Theta_1 \vee \Theta_2 \stackrel{\text{def}}{=} \{x \mapsto \Theta_1(x) \cup \Theta_2(x) \mid x \in \Theta_1 \vee x \in \Theta_2\}$$

- We iterate this analysis to a fixpoint, which we are guaranteed to reach because $\langle \Theta, \vee \rangle$ forms a join-semilattice.

To handle indirect information flows via control flow, such as the dependence of `h` on `contains_key` in Figure 5.2, we compute the control-dependence between instructions. We define control-dependence following Ferrante et al. [123]: an instruction Y

is control-dependent on X if there exists a directed path P from X to Y such that any Z in P is post-dominated by Y, and X is not post-dominated by Y. An instruction X is post-dominated by Y if Y is on every path from X to a `return` node. We compute control-dependencies by generating the post-dominator tree and frontier of the CFG using the algorithms of Cooper et al. [124] and Cytron et al. [125], respectively.

Besides a return, the only other control-flow path out of a function in Rust is a panic. For example, each function call in Figure 5.2 actually has an implicit edge to a panic node (not depicted). Unlike exceptions in other languages, panics are designed to indicate unrecoverable failure. Therefore we exclude panics from our control-dependence analysis.

5.3.2 Computing Loan Sets from Lifetimes

To verify ownership-safety (perform "borrow-checking"), the Rust compiler does not explicitly build the loan sets of lifetimes (or provenances in Oxide terminology). The borrow checking algorithm performs a sort of flow-sensitive dataflow analysis that determines the range of code during which a lifetime is valid, and then checks for conflicts e.g. in overlapping lifetimes (see the non-lexical lifetimes RFC [126]).

However, Rust's borrow checker relies on the same fundamental language feature as Oxide to verify ownership-safety: outlives-constraints. For a given Rust function, Rust can output the set of outlives-constraints between all lifetimes in the function. These lifetimes are generated in the same manner as in Oxide, such as from inferred subtyping requirements or user-provided outlives-constraints. Then given these constraints, we compute loan sets via a process similar to the ownership-safety judgment described in Section 5.1.2. In short, for all instances of borrow expressions $\&r\,\omega\,p$ in the MIR program, we initialize $\Gamma(r) = \{p\}$. Then we propagate loans via $\Gamma(r) = \bigcup_{r' :> r} \Gamma(r')$ until Γ reaches a fixpoint.

5.3.3 Handling Ownership-Unsafe Code

Rust has a concept of *raw pointers* whose behavior is comparable to pointers in C. For a type `T`, an immutable reference has type `&T`, while an immutable raw pointer

has type `*const T`. Raw pointers are not subject to ownership restrictions, and they can only be used in blocks of code demarcated as `unsafe`. They are primarily used to interoperate with other languages like C, and to implement primitives that cannot be proved as ownership-safe via Rust's rules.

Our pointer and mutation analysis fundamentally relies on ownership-safety for soundness. We do not try to analyze information flowing directly through unsafe code, as it would be subject to the same difficulties of analyzing C++ described in Section 4.3. While this limits the applicability of our analysis, empirical studies have shown that most Rust code does not (directly) use unsafe blocks [127, 128].

5.4 Evaluation

Section 5.2 established that our analysis is *sound*. The next question is whether it is *precise*: how many spurious flows are included by our analysis? We evaluate two directions:

1. What if the analysis had *more* information? If we could analyze the definitions of called functions, how much more precise are whole-program flows vs. modular flows?

2. What if the analysis had *less* information? If Rust's type system was more like C++, i.e. lacking ownership, then how much less precise do the modular flows become?

To answer these questions, we created three modifications to Flowistry:

- WHOLE-PROGRAM: the analysis recursively analyzes information flow within the definitions of called functions. For example, if calling a function `f(&mut x, y)` where `f` does not actually modify `x`, then the WHOLE-PROGRAM analysis will not register a flow from `y` to `x`.

- MUT-BLIND: the analysis does not distinguish between mutable and immutable references. For example, if calling a function `f(&x)`, then the analysis assumes that `x` can be modified.

- REF-BLIND: the analysis does not use lifetimes to reason about references, and rather assumes all references of the same type can alias. For example, if a function takes as input `f(x: &mut i32, y: &mut i32)` then `x` and `y` are assumed to be aliases.

The WHOLE-PROGRAM modification represents the most precise information flow analysis we can feasibly implement. The MUT-BLIND and REF-BLIND modifications represent an ablation of the precision provided by ownership types. Each modification can be combined with the others, representing $2^3 = 8$ possible conditions for evaluation.

To better understand WHOLE-PROGRAM, say we are analyzing the information flow for an expression `f(&mut x, y)` where `f` is defined as `f(a, b) { (*a).1 = b; }`. After analyzing the implementation of `f`, we translate flows to parameters of `f` into flows on arguments of the call to `f`. So the flow `b` $\rightarrow$ `(*a).1` is translated into `y` $\rightarrow$ `x.1`. Additionally, if the definition of `f` is not available, then we fall back to the modular analysis. Importantly, due to the architecture of the Rust compiler, the only available definitions are those *within the package being analyzed*. Therefore even with WHOLE-PROGRAM, we cannot recurse into e.g. the standard library.

With these three modifications, we compare the number of flows computed from a dataset of Rust projects (Section 5.4.1) to quantitatively (Section 5.4.2) and qualitatively (Section 5.4.3) evaluate the precision of our analysis.

5.4.1 Dataset

To empirically compare these modifications, we curated a dataset of Rust packages (or "crates") to analyze. We had two selection criteria:

1. To mitigate the single-crate limitation of WHOLE-PROGRAM, we preferred large crates so as to see a greater impact from the WHOLE-PROGRAM modification. We only considered crates with over 10,000 lines of code as measured by the `cloc` utility [129].

2. To control for code styles specific to individual applications, we wanted crates from a wide range of domains.

Table 5.1: Dataset of crates used to evaluate information flow precision, ordered in increasing number of variables analyzed. Each project often contains many crates, so a sub-crate is specified where applicable, and the root crate is analyzed otherwise. Metrics displayed are LOC (lines of code), number of variables, number of functions, and the AIF (average [MIR] instructions per function).

Project	**Crate**	**Purpose**	**LOC**	**# Vars**	**# Funcs**	**AIF**
rayon		Data parallelism library	15,524	10,607	1,079	16.6
Rocket	core/lib	Web backend framework	10,688	12,040	741	25.5
rustls	rustls	TLS implementation	16,866	23,407	868	42.4
sccache		Distributed build cache	23,202	23,987	643	62.1
nalgebra		Numerics library	31,951	35,886	1,785	26.7
image		Image processing library	20,722	39,077	1,096	56.8
hyper		HTTP server	15,082	44,900	790	82.9
rg3d		3D game engine	54,426	59,590	3,448	25.7
rav1e		Video encoder	50,294	76,749	931	115.4
RustPython	vm	Python interpreter	47,927	97,637	3,315	51.0
		Total:	286,682	435,979	14,696	

After a manual review of large crates in the Rust ecosystem, we selected 10 crates, shown in Table 5.1. We built each crate with as many feature flags enabled as would work on our Ubuntu 16.04 machine. Details like the specific flags and commit hashes can be found in the appendix.

For each crate, we ran the information flow analysis on every function in the crate, repeated under each of the 8 conditions. Within a function, for each local variable x, we compute the size of $\Theta(x)$ at the exit of the CFG — in terms of program slicing, we compute the size of the variable's backward slice at the function's return instructions. The resulting dataset then has four independent variables (crate, function, condition, variable name) and one dependent variable (size of dependency set) for a total of 3,487,832 data points.

Our main goal in this evaluation is to analyze precision, not performance. Our baseline implementation is reasonably optimized — the median per-function execution time was 370.24μs. But Whole-program is designed to be as precise as possible, so its naive recursion is sometimes extremely slow. For example, when analyzing the `GameEngine::render` function of the `rg3d` crate (with thousands of functions in its call graph), the modular analysis takes 0.13s while the recursive analysis takes

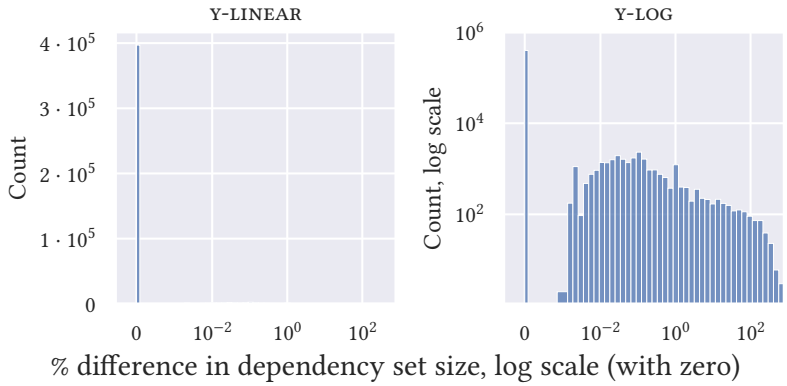

Figure 5.3: Distribution in differences of dependency set size between Whole-program and Modular analyses. The x-axis is a log-scale with 0 added for comparison. Most sets are the same, so 0 dominates (left). A log-scale (right) shows the tail more clearly.

23.18s, a 178× slowdown. Future work could compare our modular analysis to whole-program analyses across the precision/performance spectrum, such as in the extensive literature on context-sensitivity [121].

5.4.2 Quantitative Results

We observed no meaningful patterns from the interaction of modifications — for example, in a linear regression of the interaction of Mut-blind and Ref-blind against the size of the dependency set, each condition is individually statistically significant ($p < 0.001$) while their interaction is not ($p = 0.337$). So to simplify our presentation, we focus only on four conditions: three for each modification individually active with the rest disabled, and one for all modifications disabled, referred to as Modular.

Whole-program

For Whole-program, we compare against Modular to answer our first evaluation question: how much more precise is a whole-program analysis than a modular one? To quantify precision, we compare the *percentage increase in size* of dependency sets for a given variable between two conditions. For instance, if Whole-program computes

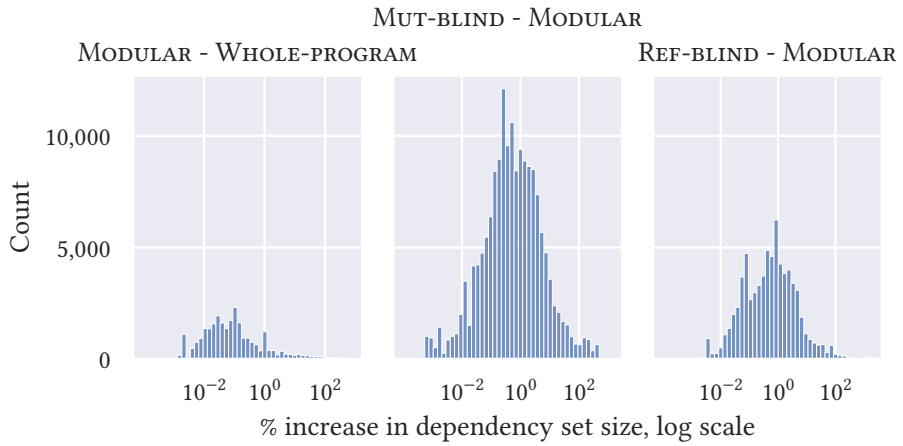

Figure 5.4: Distribution in differences between Modular and each alternative condition, with zeros excluded to highlight the shape of each distribution. Mut-blind and Ref-blind both reduce the precision more often and more severely than Modular does vs. Whole-program.

$|\Theta(x)| = 2$ and Modular computes $|\Theta(x)| = 5$ for some x, then the difference is $(5-2)/2 = 1.5 = 150\%$.

Figure 5.3 shows a histogram of the differences between Whole-program and Modular for all variables. In 94% of all cases, the Whole-program and Modular conditions produce the same result and hence have a difference of 0. In the remaining 6% of cases with a non-zero difference, visually enhanced with a log-scale in Figure 5.3-right, the metric follows a right-tailed log-normal distribution. We can summarize the log-normal by computing its median, which is 7%. This means that within the 6% of non-zero cases, the median difference is an increase in size by 7%. Thus, the modular approximation does not significantly increase the size of dependency sets in the vast majority of cases.

Mut-blind and Ref-blind

Next, we address our second evaluation question: how much less precise is an analysis with weaker assumptions about the program than the Modular analysis? For this question, we compare the size of dependency sets between the Mut-blind and Ref-blind conditions versus Modular. Figure 5.4 shows the corresponding histograms

of differences, with the Whole-program vs. Modular histogram included for comparison.

First, the Mut-blind and Ref-blind modifications reduce the precision of the analysis more often and with a greater magnitude than Modular does vs. Whole-program. 39% of Mut-blind cases and 17% of Ref-blind cases have a non-zero difference. Of those cases, the median difference in size is 50% for Mut-blind and 56% for Ref-blind.

Therefore, the information from ownership types is valuable in increasing the precision of our information flow analysis. Dependency sets are often larger without access to information about mutability or lifetimes.

5.4.3 Qualitative Results

The statistics convey a sense of how often each condition influences precision. But it is equally valuable to understand the kind of code that leads to such differences. For each condition, we manually inspected a sample of cases with non-zero differences vs. Modular.

Modularity.

One common source of imprecision in modular flows is when functions take a mutable reference as input for the purposes of passing the mutable permission off to an element of the input.

```
fn crop<I: GenericImageView>(
  image: &mut I, x: u32, y: u32,
  width: u32, height: u32
) -> SubImage<&mut I> {
  let (x, y, width, height) =
    crop_dimms(image, x, y, width, height);
  SubImage::new(image, x, y, width, height)
}
```

For example, the function `crop` from the `image` crate returns a mutable view on an image. No data is mutated, only the mutable permission is passed from whole image

to sub-image. However, a modular analysis on the `image` input would assume that `image` is mutated by `crop`.

Another common case is when a value only depends on a subset of a function's inputs. The modular approximation assumes all inputs are relevant to all possible mutations, but this is naturally not always the case.

```
fn solve_lower_triangular_with_diag_mut<R2,C2,S2>(
  &self, b: &mut Matrix<N, R2, C2, S2>, diag: N,
) -> bool {
  if diag.is_zero() { return false; }
  // logic mutating b...
  true
}
```

For example, this function from `nalgebra` returns a boolean whose value solely depends on the argument `diag`. However, a modular analysis of a call to this function would assume that **`self`** and `b` is relevant to the return value as well.

Mutability

The reason Mut-blind is less precise than Modular is quite simple — many functions take immutable references as inputs, and so many more mutations have to be assumed.

```
fn read_until<R, F>(io: &mut R, func: F)
  -> io::Result<Vec<u8>>
  where R: Read, F: Fn(&[u8]) -> bool
{
  let mut buf = vec![0; 8192]; let mut pos = 0;
  loop {
    let n = io.read(&mut buf[pos..])?; pos += n;
    if func(&buf[..pos]) { break; }
    // ...
  }
}
```

For instance, this function from `hyper` repeatedly calls an input function `func`

with segments of an input buffer. Without a control-flow analysis, it is impossible to know what functions `read_until` will be called with. And so MUT-BLIND must always assume that `func` could mutate `buf`. However, MODULAR can rely on the immutability of shared references and deduce that `func` could not mutate `buf`.

Lifetimes

Without lifetimes, our analysis has to make more conservative assumptions about objects that could possibly alias. We observed many cases in the REF-BLIND condition where two references shared different lifetimes but the same type, and so had to be classified as aliases.

```
fn link_child_with_parent_component(
  parent: &mut FbxComponent,
  child: &mut FbxComponent,
  child_handle: Handle<FbxComponent>,
) { match parent {
  FbxComponent::Model(model) => {
    model.geoms.push(child_handle),
  },
  // ..
}}
```

For example, the `link_child_with_parent_component` function in `rg3d` takes mutable references to a `parent` and `child`. These references are guaranteed not to alias by the rules of ownership, but a naive pointer analysis must assume they could, so modifying `parent` could modify `child`.

5.4.4 Threats to Validity

Finally, we address the issue: how meaningful are the results above? How likely would they generalize to arbitrary code rather than just our selected dataset? We discuss a few threats to validity below.

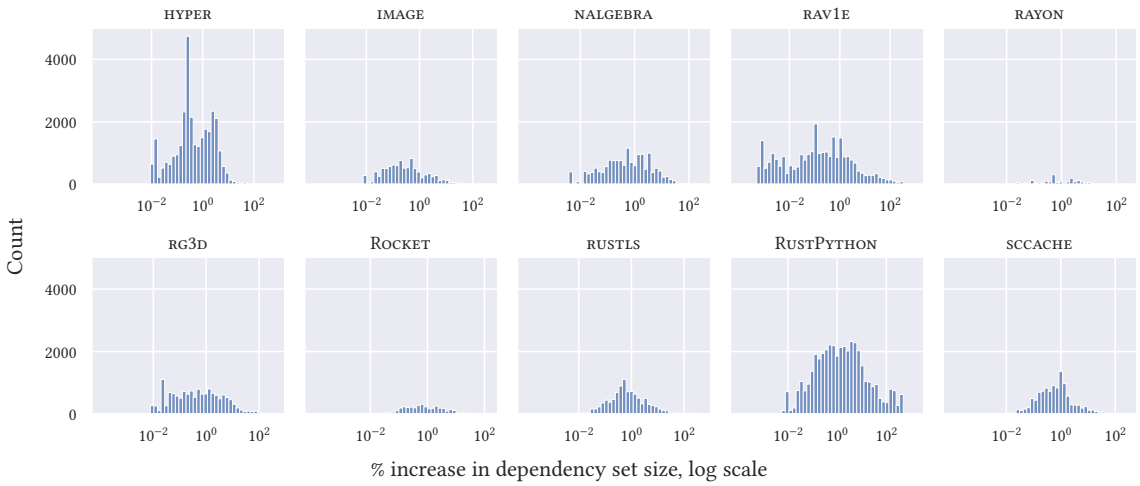

Figure 5.5: Distribution of non-zero differences between MODULAR and MUT-BLIND, broken down by crate.

Are the results due to only a few crates? If differences between techniques only arose in a small number of situations that happen to be in our dataset, then our technique would not be as generally applicable. To determine the variation between crates, we generated a histogram of non-zero differences for the MODULAR vs. MUT-BLIND comparison, broken down by crate in Figure 5.5.

As expected, the larger code bases (e.g. rav1e and RustPython) have more non-zero differences than smaller codebases — in general the correlation between non-zero differences and total number of variables analyzed is strong, $R^2 = 0.79$. However variation also exists for crates with roughly the same number of variables like `image` and `hyper`. MUT-BLIND reduces precision for variables in `hyper` more often than `image`. A qualitative inspection of the respective codebases suggests this may be because `hyper` simply makes greater use of immutable references in its API.

These findings suggest that the impact of ownership types and the modular approximation likely do vary with code style, but a broader trend is still observable across all code.

Would Whole-program be more precise with access to dependencies? A limitation of our whole-program analysis is our inability to access function definitions

outside the current crate. Without this limitation, it may be that the Modular analysis would be significantly worse than Whole-program. So for each variable analyzed by Whole-program, we additionally computed whether the information flow for that variable involved a function call across a crate boundary.

Overall 96% of cases reached at least one crate boundary, suggesting that this limitation does occur quite often in practice. However, the impact of the limitation is less clear. Of the 96% of cases that hit a crate boundary, 6.6% had a non-zero difference between Modular and Whole-program. Of the 4% that did not hit a crate boundary, 0.6% had a non-zero difference. One would expect that Whole-program would be the most precise when the whole program is available (no boundary), but instead it was much closer to Modular.

Ultimately it is not clear how much more precise Whole-program would be given access to all a crate's dependencies, but it would not necessarily be a significant improvement over the benchmark presented.

Is ownership actually important for precision? The finding that Ref-blind only makes a difference in 17% of cases may seem surprisingly small. For instance, Shapiro and Horwitz [130] found in a empirical study of slices on C programs that "using a pointer analysis with an average points-to set size twice as large as a more precise pointer analysis led to an increase of about 70% in the size of [slices]."

A limitation of our ablation is that the analyzed programs were written to satisfy Rust's ownership safety rules. Disabling lifetimes does not change the structure of the programs to become more C-like — Rust generally encourages a code style with fewer aliases to avoid dealing with lifetimes. A fairer comparison would be to implement an application idiomatically in both Rust and Rust-but-without-feature-X, but such an evaluation is not practical. It is therefore likely that our results understate the true impact of ownership types on precision given this limitation.

Chapter 6

Evaluating the Utility of Slicing When Debugging

After implementing Flowistry as described in Chapter 5, I returned to the original goal of this line of work: evaluating how program slicing can be useful to programmers in finding relevant code. The main design challenge was the user interface: how should a programmer ask for a slice, and how should the slice be presented? These questions need to be answered in terms of the context of use: how would a programmer use the slice to accomplish a task?

The answer suggested by prior work is *relevance*. For example, as described by Weiser and Lyle [12]:

> One aid to understanding is to reduce the amount of detail a programmer sees by extracting only relevant information. An application of data-flow analysis, program slicing, can be used to transform a large program into a smaller one containing only those statements relevant to the computation of a given output.

Nearly every paper on program slicing since Weiser's has used the term "relevance" in the same way: a slice is the set of code "relevant" to a slicing criterion. Therefore I set out to design a slicing UI, Focus Mode, that would help programmers find code relevant to their task (Section 6.1), and to evaluate whether this design holds

up in preliminary user testing (Section 6.2).

6.1 Design

As described in Section 1.2, the natural place for a programmer's cognitive support tool is the IDE. Therefore I designed the slicer UI, Focus Mode, as an extension to the Visual Studio Code IDE. At a high level, a user can click on a variable, and the extension will visualize the (modular) slice of that variable. In this section, I will describe the details of computing and visualizing a slice.

6.1.1 Computing a Slice

The interface provided by Flowistry is essentially: given a Rust function f, then for each location $\ell \in f$, Flowistry produces a dependency context Θ_ℓ. When the user asks to compute a slice on a given variable x, Focus Mode finds the function f containing x, finds the location ℓ of x, and the computes Θ_ℓ.

Given Θ_ℓ, computing the slice is relatively simple. The backward and forward slices of a variable (or more generally a place expression p) are:

$$\mathsf{backward}_\Theta(p) = \Theta(p)$$
$$\mathsf{forward}_\Theta(p) = \{\ell \mid \exists p' \in \mathsf{inputs}(e_\ell) \;.\; \Theta(p) \subseteq \Theta(p')\}$$

A place's backward slice is exactly its set of dependencies. A place's forward slice is the set of things that depend on it, i.e. an input p' to the expression depends on p which can be determined by $\Theta(p') \subseteq \Theta(p)$.

Notably this analysis produces the *modular* slice of the user's selection. Focus Mode only operates on a single function at a time, rather than performing a whole-program analysis every time the user moves their cursor.

```
1  fn average_and_median_of_evens(ns: impl Iterator<Item = usize>) -> (usize,
       usize) {
2    println!("Starting computation!");
3    let start = Instant::now();
4    let mut v = ns.collect::<Vec<_>>();
5    v.sort();
6    let v2 = v.iter().copied().filter(|x| *x % 2 == 0).collect::<Vec<_>>();
7    assert!(v2.len() > 0);
8    let median = v2[v2.len() / 2];
9    let avg = v2.iter().copied().sum::<usize>() / v.len();
10   println!("Elapsed: {}", start.elapsed().as_secs_f64());
11   (avg, median)
12 }
```

Figure 6.1: A buggy Rust program that computes the average and median of the even numbers in the input. The denominator on line 9 is supposed to be `v2.len()`.

6.1.2 Visualizing a Slice

To effectively design an interface for program slices, we first need to understand the relationship between slices and relevance: what makes information cognitively relevant to a particular programming task? And how does a slice represent (or not) such relevant information? For a programmer performing a particular task (e.g. debugging, refactoring, etc.), we say information is *relevant* to the programmer if mentally processing that information would influence the programmer's approach to the task, e.g. by helping them complete task more quickly.

While information flow is a precise theory of relevance as a formal construct, we cannot provide as clean a theory of when information is *cognitively* relevant — that determination is heavily contingent on both the task and the programmer. We can, at least, make a few conjectures grounded in our current understanding of programmer practice and cognitive psychology. As a running example, consider the buggy Rust program in Figure 6.1. This function attempts to compute the average and median of even numbers in a vector, except the average's denominator incorrectly refers to the wrong vector (`v` instead of `v2`). Suppose a programmer observes a failing unit tests that shows an incorrect output for the average. Then what information is relevant to fixing this bug?

1. **The code affecting the buggy value is relevant:** the code which affects the runtime value of `avg` (lines 4, 6, and 9) are relevant to an operational understanding of how `avg` is produced. A programmer would likely mentally trace these lines to simulate how the bug occurs [131].

2. **Parallel code structures may be relevant:** the median calculation on line 8 also involves dividing by a vector length, and a programmer may notice that line 8 uses `v2.len()` while line 9 uses `v.len()`, deducing the root cause. Programmers frequently identify patterns (or schema) in code and use these patterns to make such inferences [44].

3. **Direct influence is more relevant than transitive influence:** when trying to understand what affects `avg`, a programmer is more likely to be first interested in the direct influence on `avg` (e.g. the vector `v2`) than the transitive influence (e.g. the input `ns`). Programmers have very limited working memory, and so must judiciously decide which code to read and in what order to avoid overloading their memory (Chapter 3).

4. **The set of relevant code will rapidly shift:** upon observing that `v2` affects `avg`, a programmer may ask "what is the role of `v2` in the program?" The code relevant to this question is entirely different, e.g. lines 7 and 8 indicate the role of `v2`. Programmers often swap between bottom-up ("why is this here?") and top-down ("how does this work?") forms of reasoning [132].

A key takeaway from these observations is that a program slice does clearly contain relevant information (as per principle 1), but a slice misses important nuances of cognitive relevance. Important information such as parallel code may be outside a slice. Within a slice, not all information should be weighted equally. And a programmer will likely want multiple slices to understand a given program.

Based on these principles, we designed Focus Mode, a novel interface for visualizing program slices that integrates into the Visual Studio Code IDE, shown in Figure 6.2. To use the tool, a programmer enters Focus Mode with a keyboard shortcut, and then places their cursor on the variable or expression they want to

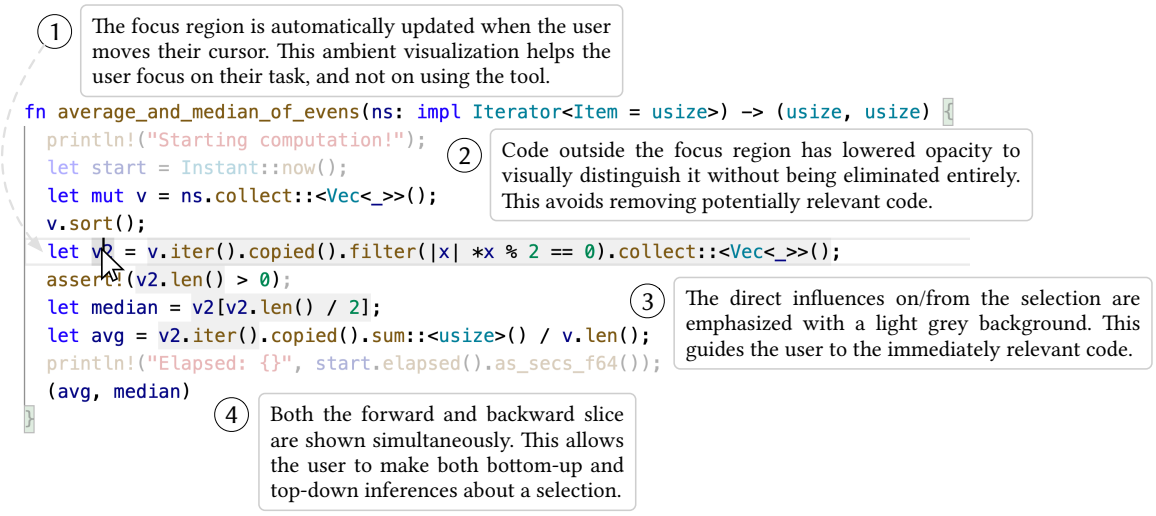

Figure 6.2: A diagrammed screenshot of the Focus Mode interface in Visual Studio Code, showing the focus regions after selecting the variable `v2`. Each point highlights how a cognitive principle informed the UI design.

focus on. After selecting code with a corresponding place p at a location ℓ, Focus Mode computes and visualizes the "focus region" of p as the union of its forward and backward slices at the location ℓ, i.e. $\mathsf{forward}_{\Theta_\ell}(p) \cup \mathsf{backward}_{\Theta_\ell}(p)$. Figure 6.2 explains how the UI design of Focus Mode draws on the principles above.

In line with our goal to make a practical slicing tool, Focus Mode is a free and open-source VSCode extension that is currently available to the public. As of August 2022, Focus Mode has been downloaded 1,800 times from the VSCode extension marketplace. Focus Mode is available for download here: https://github.com/willcrichton/flowistry

6.2 User study

With a slicing interface in hand, we next returned to the core question of this work: do program slices (mediated through our interface) actually help programmers find relevant code? To gather preliminary evidence about this question, we conducted a study of $N = 18$ Rust developers using Focus Mode to debug small Rust programs. Although we expect slices to be useful for other programming tasks, we specifically

used a debugging task because it (a) requires programmers to read and understand code, and (b) has been the primary application of slicing in the past. Our two main research questions were:

1. Could developers successfully incorporate Focus Mode into their debugging process?

2. For which scenarios can Focus Mode help programmers find relevant code?

6.2.1 Methodology

We designed three small Rust programs that contained a bug and a corresponding broken unit test: Url, Median, and Command-line. See Figure 6.3 for a description and excerpt of each program. The programs were designed to exhibit a range of domains, difficulties, and code styles so as to test Focus Mode in many settings. Additionally, each program had more code than was relevant to the bug (as would be true in any real-world setting) so Focus Mode would be potentially useful for distinguishing relevant code. The full source of each program is available in the supplementary materials.

The overall structure of the experiment was: first, participants were given a 5 minute tutorial on how to use Focus Mode, and then given 10 minutes to use Focus Mode to debug a sample program. Then for each task, participants were instructed to find and fix the bug in the given program. Participants had a maximum of 15 minutes per task. Participants used their own Rust development setup, except they were required to use Visual Studio Code to integrate with Focus Mode. Participants were allowed to use all normal debugging tools (printing, debuggers, Google, etc.) in addition to Focus Mode. After the tasks were completed, we elicited feedback from participants about their experience in a semi-structured interview.

Additionally, we wanted to compare the experiences of developers both using and not using Focus Mode on each task, so as to better highlight where Focus Mode may be useful. Therefore we had participants use Focus Mode for two of the tasks (randomly selected), and not use Focus Mode for the remaining task. The order of

```
let mut path = Vec::new();
loop {
  let part = chars.take_while(|c| *c != '/').collect::<String>();
  if part.len() > 0 { path.push(part); }
  else              { break; }
}
let query = match path.last() {
  Some(part) if part.contains('?') => {
    let page = path.pop().unwrap();
    let (_, query) = page.split_once('?').unwrap();
    Some(query.to_string())
  }
  _ => None,
};
Some(Url { scheme, hostname, tld, port, path, query })
```

(a) URL (61 LOC function, 129 LOC total): a string parsing program. Given a string representing a URL, the program parses the string into its constituent components. Bug: the last `path` component is removed for processing on line 9, but not pushed back after removing the query suffix.

```
for (continent, cont_countries) in continents.iter_mut() {
  cont_countries.sort_by_key(|c| c.population);
  let n = cont_countries.len();
  let median_countries = if n % 2 == 1 {
    vec![&cont_countries[(n - 1) / 2]]
  } else {
    vec![&cont_countries[n / 2 - 1], &cont_countries[n / 2]]
  };
  let names = median_countries
    .iter().map(|c| c.name.to_string()).collect::<BTreeSet<_>>();
  let pop_total = median_countries.iter().map(|c| c.population).sum::<usize>();
  let median = pop_total / cont_countries.len();
  medians.insert(continent.to_string(), (names, median));
}
```

(b) MEDIAN (36 LOC function, 108 LOC total): a tabular data analytics program. Given a list of countries, the program computes the median population and names of countries at the median, grouped by continent. Bug: the calculation of `median` on line 12 incorrectly uses `cont_countries.len()` instead of `median_countries.len()`.

```
let mut cli = self.clone();
let mut args = VecDeque::from(args);
let mut parsed = HashMap::new();
let build_error = |err: ErrorType| err.to_string(self);
```

(c) COMMAND-LINE (several functions, 256 LOC total): a command-line interface (CLI) API. Allows users to describe a CLI, parse a list of strings into arguments, and build a helpful error. Bug: the `build_error` closure incorrectly captures the original `self` and not `cli`, which is mutated during parsing to indicate which arguments have already been accepted.

Figure 6.3: Excerpts of the programs for each debugging task demonstrating the core bug.

tasks was also randomized for each participant to reduce the influence of cross-task effects.

All 18 participants were recruited through social media, specifically the first author's Twitter and the Rust community on Reddit (/r/rust). Participants were required to have at least one month of experience with Rust, and were compensated $20 for their participation. Participants had a median 6.5 years of total programming experience, and 15/18 reported using Rust either for their jobs or on a regular basis. This study was approved by our university's Institutional Review Board. To protect the privacy of the participants, we cannot provide access to the collected data.

Our focus in this study was not to solely perform a quantitative analysis of time-on-task. Such a study is better suited to confirmatory testing of well-worn tasks and tools, as opposed to a user study of new tools. Moreover, our results are inherently prone to high variance given a diverse yet small sample. Participants had between 1 and 40 years of total programming experience, and ranged from undergraduate students to professional Rust developers to Rust compiler contributors to university professors.

Therefore we used a constructivist grounded theory methodology [133], comparable to the methodology used by Lubin and Chasins [134] in their study of functional programmers. That is, to analyze the study data, we engaged in an iterative process of annotating notable events (generating a hypothesis, remarking about an inference generated by use of Focus Mode, etc.) in the debugging sessions and post-debugging interviews, and used those annotations to find similar events in other participants' data. Then we clustered the events into thematic groups that identified larger trends in the study.

6.2.2 Task results

First, we will report on participants' experiences within each debugging task. The high-level takeaways are summarized in Table 6.1, and overall statistics are shown in Figure 6.4.

Task	Helpful?	Takeaways
URL	Yes	FOCUS MODE helped participants both eliminate irrelevant code and draw attention to relevant code they otherwise might have missed.
MEDIAN	No	When code outside a slice is relevant, FOCUS MODE's visualization does not facilitate understanding that code.
COMMAND-LINE	Somewhat	Participants did successfully use FOCUS MODE to ignore irrelvant code, but still encountered issues comparable to the MEDIAN task.

Table 6.1: Summary of takeaways from observing participants' during each debugging task.

Url task

The bug in this task is that when a URL contains a query string, the query-processing code incorrectly pops the last string off the `path` without restoring it after processing (lines 9-10 of Figure 6.3a). This task provided a clear situation where FOCUS MODE helped programmers focus on relevant code. The time-on-task for participants using FOCUS MODE was statistically significantly lower ($p = 0.029$) when controlling for participant experience. We controlled for experience by regressing time-on-task against the interaction of whether the participant used FOCUS MODE and their reported years of coding experience, and by analyzing the estimated regression coefficient of the "using-FOCUS MODE" term.

Here, FOCUS MODE provided two main functions: first, it helped de-emphasize code for decoding other parts of the URL that was not relevant to the bug. Second, it helped emphasize that code which affects `path` existed not just in the main `loop` that defines the variable, but also in the query processing code. The two participants that failed to solve the task without FOCUS MODE never read to the query processing code, and therefore never found the bug. By contrast, participants with FOCUS MODE frequently had experiences like this:

- P1 (while solving URL): "So let's have a look at path. What influences our path? Cause that's the thing that we've got wrong. [*clicks on* `path` *in the return value*

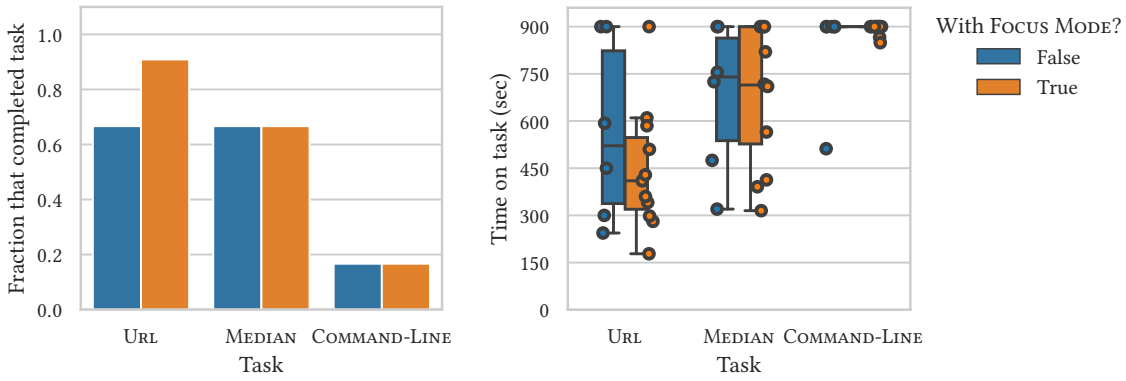

Figure 6.4: Statistics about each task, split by whether the participants were using FOCUS MODE. Left: fraction of participants that completed a given task. Right: distribution of time-on-task, showing both the individual datapoints and a boxplot summary.

and scans the focus region] I'm a bit wary of these `path.pop()`s because we might be popping things and then not saving them anywhere."

- P11 (in interview): "The addon helped me determine that `path` was used in the query portion more than I thought it was initially because I saw it here [*points to* `path.last()`] and then the addon helped me notice it was here too [*points to* `path.pop()`]."

To explore this phenomenon, we went through each participant's URL session and marked the first moment at which they explored the query processing code as a possible bug location (c.f. some participants initially glanced at the code and assumed nothing relevant was happening). From this data, we determined that participants solving URL with FOCUS MODE on average identified this code as potentially buggy 3.5 minutes earlier than participants without FOCUS MODE. This result is also statistically significant ($p = 0.014$) when controlling for programmer experience (using the same methodology as for overall time-on-task).

Median task

The bug in this task is that the denominator of a median calculation (line 12 of Figure 6.3b) refers to the wrong vector. Focus Mode was not ultimately effective at helping participants debug this issue. The difference in completion rates (Figure 6.4-left) and time-on-task (Figure 6.4-right) was not statistically significant, nor did we qualitatively observe participants gaining much insight from using Focus Mode.

The main insight we hypothesized that a participant may get from Focus Mode is to observe that `median_countries` was not relevant to the denominator, especially in contrast to `pop_total` to which it is relevant. However, in practice participants either did not click on the denominator to see its influence, or if they did they observed it for a moment and then moved on without realizing anything was amiss. In the former case, participants using Focus Mode would click on `median` to see what influences `median` (which included `median_countries` via `pop_total`), but then not click on the respective components of its calculation. In the latter case, participants would simply ignore or not notice that `median_countries` was missing from the slice. Two participants even explicitly mentioned this fact:

- P3 (after completing Median): "Focus Mode does show me that it is inconsistent, `cont_countries` is not shown here, so we should know those two are different. I looked at the suffix in the two variables "countries" and thought they were the same. Maybe if I got used to Focus Mode it could help."
- P10 (in interview): "If I clicked on the right place in the population calculation, I should have noticed that `median_countries` wasn't highlighted. [...] but I didn't observe that while I was doing it. That could be because I'm not used to Focus Mode, and so it doesn't register in my head that this isn't lit up."

This outcome is concordant with the theory that code with no information flow to a selection can still be relevant to that selection. It suggests that either users need more training with Focus Mode to detect these subtler cases (as both participants mentioned), or that fading out code may not be an appropriate visualization for code with no information flow to the selection, and alternative visualizations should be explored.

Command-line task

The bug in this task is that the code which builds an error for a command-line parser incorrectly references a version of the parser that does not register which fields have already been parsed, leading to an incorrect error message. This task was both the largest and subtlest of the three, and subsequently only 3 out of 18 participants managed to complete it (2/12 with Focus Mode, 1/6 without). Completion times were not statistically significantly different between the two groups.

This task provides mixed evidence for the utility of Focus Mode. One one hand, participants with Focus Mode were able to effectively ignore some irrelevant code in the task. Specifically, the function `ErrorType::to_string` contains code defining both the cause of the CLI error and the context of the CLI error, where only the latter is relevant to the bug. Two participants not using Focus Mode spent 90 seconds inspecting the irrelevant 36 LOC about the error cause, while all participants using Focus Mode spent a maximum of 10 seconds inspecting this code.

However, the root cause in the `Cli::parse` function went largely unnoticed. We observed that participants generally assumed that the bug would be in the "core logic" of the function, where arguments are actually parsed, and not in the "header" of the function, where variables used in the function (including the `build_error` closure) are initialized. When participants did e.g. focus on `build_error`, they did not closely inspect its focus region and notice a lack of connection to `cli`, comparable to the issue in Median. In such situations, techniques such as fault localization [135] may complement Focus Mode in guiding the user to focus on buggy lines.

6.2.3 Interview results

Finally, we report on a few recurring themes that arose in post-debugging interviews, summarized in Table 6.2.

Focus Mode as a flow visualization

Several participants reacted positively to Focus Mode because it visualized a dataflow-oriented representation of a function rather than the traditional control-flow order:

Theme	Representative quote
Participants want a flow visualization	"'Being able to see [flows] in the code without having to look at error messages and having that kind of instant feedback is really nice, because quite frankly that's the purpose of Rust."
Participants found a trust/learning curve	"Initially it's very hard to trust the extension, you feel like it might be leaving something out. You maybe double guess the extension."
Participants quickly acclimated to Focus Mode	(*on a task without* Focus Mode) "Oh my god, I'm clicking things as if I were using Focus Mode. Ok, back to basics."
Participants disagree on the best use case	"I think it would be very helpful if you have a new-to-you codebase" / "I can imagine it being more useful if I had just written the code, whereas looking at unfamiliar code I am still trying to look at every single line"

Table 6.2: Summary of takeaways from participants' post-debugging interviews.

- P0: "The main thing is that it's helping you construct a flow graph of the function by letting you look at slices of that flow graph."
- P1: "It's useful to see all those [dataflow] interactions. Without that, you just have to in your head keep track of all that stuff."
- P2: "It's nice to be able to see dataflow, because quite frankly that's the purpose of Rust. Not THE purpose, but the compiler tries so hard to internalize this notion of ownership and borrowing, so being able to visually see it in the code without having to look at error messages and having that kind of instant feedback is really nice."

However, participants disagreed about exactly what kind of information they preferred to have visualized. For example, P14: "it highlights the transitive dependencies of the thing you're clicking on. For me, it seems like the thing I would reach for more is direct dependencies." Participants also differed on whether/how they wanted to distinguish forward and backward slices:

- P1: "I definitely like the fact that it handles forward and backward dependencies depending on where you've highlighted it."

- P9: "Being able to control what influences me versus what do I influence, both before and after, might be kind of useful."
- P13: "It might be beneficial to highlight things before and after in different colors. If code influences the thing you're clicking on, then it's highlighted in orange, and otherwise in blue or something."

Focus Mode's learning curve

Several participants, especially those with fewer years of programming experience, felt that Focus Mode was either confusing or not trustworthy.

- P3: "I found it kind of confusing to use at first."
- P4: "Initially it's very hard to trust the extension, because you feel like it might be leaving something out. You maybe double guess the extension."
- P13 - "I found it a little confusing before you explained it at the start what is exactly highlighted. I clicked some things, and sometimes everything is highlighted, or something is highlighted that I don't expect."

These participants in particular called out the location-sensitivity of the analysis as confusing. Most IDE tools visualize location-insensitive program analyses, such as type checking, so clicking on the same variable at different points in a function will always show the same result. But Focus Mode gives different answers based on the information flow at the point of selection, which surprised these participants.

Subconsciously acclimating to Focus Mode

No participants reported that Focus Mode was overly intrusive or hard to read visually. Conversely, several participants reported that Focus Mode faded into the background. For example, P5: "The funny thing is that I didn't really realize I was using it. I think I was still relying on it when it was on. You don't really notice it once it's on. I guess it's quite nice in that sense. The simple fact that you just see the relevant code, and the rest is mixed into the background."

Additionally, for participants whose first two tasks involved using Focus Mode and third task did not, they frequently reported during the third task a surprising attempt to use Focus Mode despite it being disabled:

- P1 (in interview): "In the back of my head, I'm already waiting for this to highlight stuff. Amazing how quickly your brain will get lazy."
- P2 (solving Median): "You see, I had an instinct to click [the return type] here to see where we are returning from."
- P11 (in interview): "I found myself clicking on `median` and looking to see if it was connected to `median_countries`. And the addon wasn't there."
- P15 (solving Median): "It looks like we're returning the wrong value there. [*selects the return value and waits*] Oh my god, I'm clicking things as if I were using Focus Mode. Ok, back to basics."
- P16 (solving Url): "[*selects* `chars` *and waits*] Ah, now I see that I got a little bit used to having the relevant parts highlighted."

These responses suggest that even after a short time, programmers can quickly and even subconsciously acclimate to incorporating Focus Mode into their debugging process.

Hypothesized use cases for Focus Mode

Several participants speculated about scenarios in their daily practice where Focus Mode could help. However, participants often disagreed on e.g. whether Focus Mode is more or less useful after you understand the high-level structure of a codebase:

- P6: "I think it would be very helpful if you have a new-to-you codebase, and you're trying to figure out how something is built up from a series of mutations."
- P9: "I can imagine it being more useful if I had just written the code, whereas looking at unfamiliar code I am still trying to look at every single line and get a rough idea of what's going on. And so it's not as helpful in that situation."

- P15: "If you're going into a codebase and already have the background semantic knowledge, then I saw tangible benefits in bootstrapping in where we left off."

This disagreement points towards a direction for future work: examining the role of slicing in codebases with different levels of familiarity to the programmer.

6.2.4 Discussion

Our two main research questions were:

1. Could developers successfully incorporate Focus Mode into their debugging process?

2. For which scenarios can Focus Mode help programmers find relevant code?

For the first question, this study provides compelling evidence that developers could quite easily incorporate Focus Mode into their debugging process. As discussed in Section 6.2.3, many participants reported (unprompted!) that they had unexpectedly trying to use Focus Mode on a task without the tool. Participants with less programming experience would likely need more practice to fully understand how to use Focus Mode, as discussed in Section 6.2.3.

For the second question, the two main scenarios where Focus Mode helped programmers were (1) ignoring large blocks of irrelevant code, as with the irrelevant URL components in Url and the error building in Command-line, and (2) drawing attention to unexpected mutations, as with `path` in Url. The former scenario is likely to occur in many programs that have multiple concerns mixed in to a single function. The latter scenario is especially pernicious in imperative programs where pointers, closures, and other forms of indirection conspire to make mutations hard to spot.

However, Focus Mode was not effective when relevant code was outside the slice of an object of inquiry, as in Median and parts of Command-line. This suggests that the foundational framing of a slice as "relevant code", stemming all the way to Weiser's original work [1984], is misleading as it conflates a PL-theoretic concept (runtime relevance) with a cognitive concept (mental relevance). The expected usage

of a program slicer should not be to simply ignore code outside of a slice, and future work should investigate slice visualizations that facilitate comparisons of code both within and outside of a slice.

6.3 Dataset analysis

The user study demonstrates that Focus Mode holds promise in helping programmers find relevant code, but that promise is contingent on the style of code being analyzed. Some participants explicitly echoed that concern:

- P2: "This is very different from code that I would normally write. This is a much more imperative style, and my instinct is that Focus Mode would be better on this imperative-style code."
- P15: "One thing I worry about is in practice, how many of the functions I write have a lot of splitting control flow, so returning things that aren't depending on one another. I tried using Focus Mode a little bit on some Rust code I'm working on now. Because it's more recursive in nature, it didn't seem to cut out as much of the code as I'd hoped."

These concerns are comparable to those raised by Parnin and Orso [137] that programs slices may be too big to be useful. Therefore we tried to answer the question: if you randomly selected a variable in any function in an arbitrary Rust codebase, how much code would you expect to be filtered out by Focus Mode?

6.3.1 Methodology

Our methodology for answering this question is relatively simple: compute many slices in existing codebases, and analyze how much code is eliminated in each slice. For a dataset of Rust codebases, we reuse the same dataset from the evaluation of Flowistry as it represents a large and diverse set of code styles and applications written in Rust. See Table 5.1 again for the specific crates used and statistics about them.

For each codebase, for each function in the codebase, and for each selectable program object (variable, expression, function parameter, etc.) in the function, we compute the forward, backward, and combined (both directions) slice of that object. Note that an instruction can be in both a variable's forward and backward slice in the event of loops, so the size of a combined slice may be less than the sum of the size of its parts. In total, we generated 284,107 slices, multiplied by three directions to make 852,321 data points.

Unlike Section 5.4, where we measured the size of code in number of IR instructions, our goal is to approximate how much code a person would see faded out if they actually opened the codebase in an IDE. Therefore we measure the size of a focus region and its containing function in *significant lines of code*, i.e. lines of code that contain at least one token (not whitespace or comments). To compare slice sizes across functions with different sizes, we represent slice sizes as a fraction of the total SLOC in the containing function. For example, a slice containing 5 LOC in a function with 20 LOC has a fractional size 0.25.

6.3.2 Results

The distribution of slice sizes is shown in Figure 6.5-top-left. Because the distributions are not normal, we will summarize them with their median and interquartile range (25th percentile to 75th percentile). Overall, the slice sizes were: backward 32% (8%-70%), forward 50% (15%-93%), and combined 84% (50%-100%).

We would expect smaller functions to have fewer independent concerns, and therefore see larger slices in smaller functions. To test this hypothesis, we partitioned the data into small and large functions based on the median function size of 55 LOC. Figure 6.5-top-right shows the distribution of slice sizes for small and large functions. Indeed, median slice sizes were smaller for large functions. Backward slices had a size distribution of 36% (11%-78%) vs. 15% (3%-42%) for small functions and large functions, respectively. Forward slices were 60% (20%-100%) vs. 29% (5%-71%), and combined slices 89% (58%-100%) vs. 63% (24%-91%). The difference in medians is statistically significant ($p < 0.001$) for all three directions by a Kruskal-Wallis H-test.

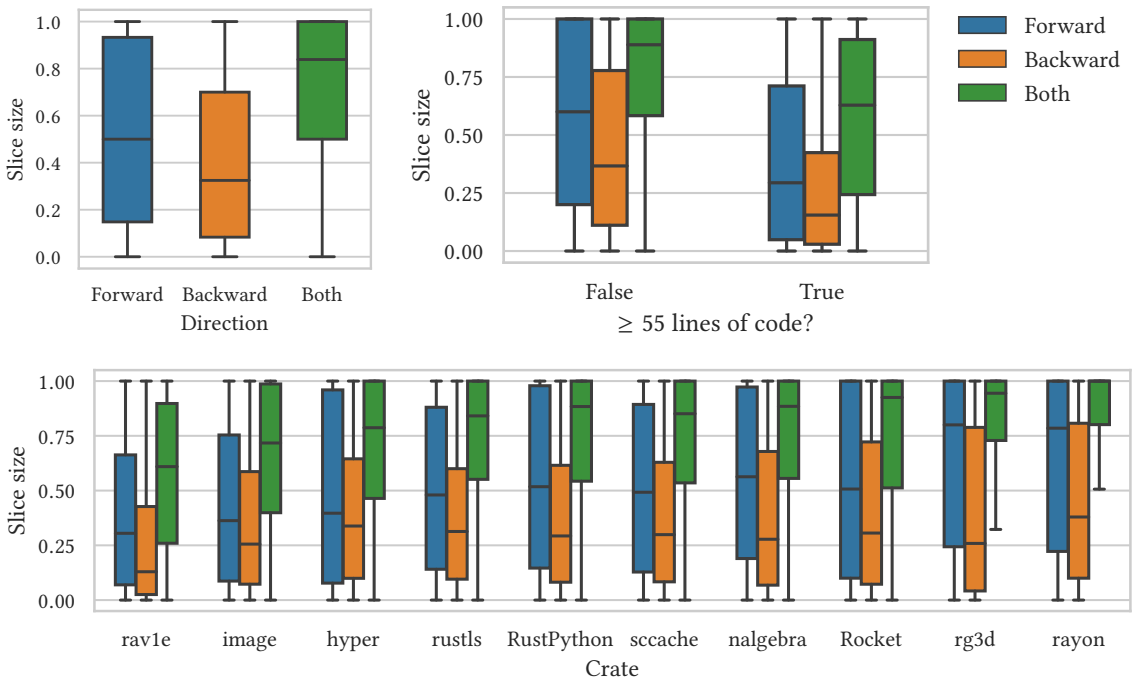

Figure 6.5: The distribution of slice sizes as a fraction of the containing function, faceted by direction (top-left) and further faceted by function size (top-right) and crate (bottom). Lower is better — for example, the forward slice of the return value and the backward slice of a function parameter includes 0% of code.

To estimate the effect of crate-level code style on slice size, we analyzed the distribution of slice sizes per crate, shown in Figure 6.5-bottom. Consistent with the crate-level analysis in Section 5.4.4, there is indeed some notable inter-crate variance. Slices in `rayon` are, on average, larger than slices in `rav1e`, likely due to the fact that `rayon` has on average much smaller functions. But slices still excluded a considerable amount of code in most crates.

6.3.3 Discussion

These results show that modular slices can be quite effective at eliminating code. The median backward slice in a large function was only 15% of the total LOC in that function — a very substantial reduction! Backward slices are on average smaller than forward slices, which suggests that the information flow graph of most computations will branch outward (e.g. a tree with a root branching to many leaves) more than inward (e.g. many leaves reducing to a single root).

The distributions of slice sizes exhibit significant variance, regardless of crate or function size. A finer granularity analysis of code styles or information flow topologies might explain this variance – is the variance due to straight-line vs. branching computation? Sequential vs. iterative control flow? Future work should explore this question to help programmers better understand when they would expect a tool like Focus Mode to be of greatest help.

A concern about these results is whether program slicing is only useful for messy code. One cause of large functions is when many responsibilities are lumped into a single location, whereas a better code style might factor these responsibilities into several smaller functions. One possible direction would be to make Focus Mode work interprocedurally, i.e. compute and visualize slices across function boundaries. But working with code fragmented across many functions is a known challenge for programmers [93], so an alternative could be to combine modular slicing with a partial evaluation tool that inline and simplify code from called functions [138].

Chapter 7

Conclusions and Future Work

In this dissertation, I have shown that program slicing is both technically feasible and empirically useful in helping programmers find relevant code. I reinforced the cognitive basis for slicing with the experiments in Chapter 3, showing that working memory limitations influence core programming tasks like tracing. I provided a new theoretical foundation for slicing in ownership-based languages through the information flow analysis in Chapter 5. And finally, I showed in Chapter 6 that a Rust slicer can help programmers find relevant code while debugging, with some notable caveats.

This dissertation further supports the argument that cognitive psychology and programming language theory can only *together* provide a principled foundation for the design of programming tools. As shown in Chapter 2, pure cognitive psychology enables us to uncover facts about how people think when programming, but these facts are difficult to apply in isolation. As shown in Chapter 4, pure programming language theory enables us to design an infinite array of mechanisms for analyzing program properties, but these mechanisms won't be useful to programmers without tailoring them to cognition.

To that end, my research points to several directions for future work that span both disciplines.

7.1 Cognitive Design Principles for Programming

Working memory is just one of many possible lenses for understanding the practice of programming. Future work should explore how every relevant aspect of cognition influences programming, from micro-tasks like tracing a line of code to macro-tasks like implementing a complex high-performance system. Particular focus should be given on translating cognitive insights into principles that can meaningfully inform the design of practical tools. Here are two areas I think are particularly promising:

Perception and information visualization. Despite decades of research on software visualization, most programmers use very few visualizations today beyond the IDE features outlined in Figure 1.1 [139, 140]. Yet, the need for visualization is still present: a modern codebase contains an overwhelming amount of information for a person to comprehend. Moreover, researchers in information visualization have successfully applied theories of perception to generate design principles for all manner of graphics: data visualizations [5], route maps [141], assembly instructions [142], and more.

For example, many emerging programming languages have complex type systems. These type systems operate by combining explicit information in a program's syntax with inferred facts and constraints, and checking for violations of various safety properties. A common experience in these languages is a "fight" with the compiler, i.e. iteratively writing a program, being rejected by the compiler, and rewriting to try and satisfy its rules.

I believe that a core challenge in this setting is that much of the relevant information may be invisible to the programmer — they must deduce it and hold in working memory. Very little work has explored how visualizations of these constraints could facilitate the high-level inferences that a programmer wants to make: why is my program broken, and how do I fix it? Identifying principles for visualizations of statically inferred program properties would be an invaluable contribution toward improving the accessibility of the next generation of programming languages to all programmers.

Abstraction and API design. Abstraction is both a cognitive and technical phenomenon: programmers mentally build abstractions to think about the segments of a problem domain or semantics of a programming language. Programmers also build technical abstractions, usually in the form of APIs: functions, data structures, libraries, and frameworks that encapsulate a way of thinking about a domain.

The gap between between cognitive and technical abstractions is a known usability challenge with APIs, as shown by Robillard and Deline [143]:

> "When professional developers like the ones in our studies learn a new API, they struggle not so much in the mechanics of using the API, but in understanding how the API relates *upwards* towards its problem domain and *downwards* towards its implementation."

Conversely, the proliferation of APIs advertised as being "for humans" shows the value that programmers place on technical abstractions that mirror the way they think about a given problem.

Cognitive psychology has a long history of research about the formation and application of concepts, categories, analogies, and other cognitive abstractions. These ideas have been linked to programming-relevant domains — for example, psychologists Lakoff and Núñez argue in *Where Mathematics Comes From* [144] that many advances in mathematics can be traced back to the evolution of metaphors that people used to conceptualize mathematical objects like numbers and sets.

Therefore I believe that psychological research may provide significant insight into the relationship between cognitive and technical abstractions in programming. For example, is it possible to argue precisely why one API is a "better" model of a domain than another in purely cognitive terms? Are there principles for how API designers can document their APIs so as to facilitate the formation of mental abstractions that mirror the technical ones? Programmers stand to benefit greatly from answers to these questions.

```
6   fn grep(path: &Path, query: &str) -> Vec<String> {
7     let mut p: Command = Command::new(program: "grep");
8     let mut add_arg: |&str| -> () = |s: &str| {
9       p.arg(s);
10    };
11
12    let abs_prefix: &String = &fs::canonicalize(path: path.parent().unwrap())
13      .unwrap()
14      .display()
15      .to_string();
16    let strip_prefix: |&str| -> String = |s: &str| -> String {
17      s.strip_prefix(abs_prefix)
18        .unwrap_or_else(|| panic!("…"))
19        .to_string()
20    };
21
22    log::debug!("Adding arguments to process");
23    let should_recurse: bool = path.is_file();
24    if should_recurse {
25      add_arg("-r");
26    }
27
28    add_arg(query);
29    add_arg(&path.display().to_string());
30
31    log::debug!("Executing grep");
32    let result: Output = p.output().unwrap();
33    assert!(result.stderr.is_empty());
34
35    return String::from_utf8(vec: result.stdout)
36      .unwrap()
37      .split("\n")
38      .map(strip_prefix)
39      .map(|s: String| s.to_string())
40      .collect::<Vec<_>>();
41  } fn grep
```

Figure 7.1: A prototype IDE extension that combines information flow with graph community analysis to identify clusters of code. Each cluster is highlighted in a different color. This clustering shows, for example, that lines 12-20 are a distinct logical chunk from lines 7-10 and 23-33, even though they are in a sequential block.

7.2 Information Flow Beyond Security

I have demonstrated that my system for ownership-based information flow, Flowistry, has clear and immediate implications for building a program slicer, and could certainly be adapted for the traditional application of information flow control. But I am equally excited by the potential applications of information flow analysis beyond security, such as in compiler optimizations or IDE refactoring. Here are two particular applications I've considered:

Interactive code decomposition. Codebases evolve over time, grow in scope, and churn through maintainers. In the process, functions often acquire many different responsibilities until being refactored into smaller functions. Currently, identifying when and how to decompose a function is an expert skill simply gained through experience.

One potential application of information flow analysis is to automatically identify the functional sub-units of a given program. The hypothesis is that code which is more related to each other has more internal dependencies. Then a tool can statistically analyze the average connectivity of nodes within the information flow graph to distinguish clusters of functionality.

I have implemented a prototype of this idea, shown in Section 7.2. The prototype uses modularity maximization to identify communities within the information flow graph, and then highlights the communities in the IDE. In a more complete prototype, a user would be able to interactively sweep through different granularities of decomposition, adjust individual blocks, and then refactor each block into a function with its own name. I believe tools like this would help maintainers of legacy codebases in sorting through complex intra-function dependency graphs.

Compiler optimizations. For languages with simple type systems like C and C++, optimizing compilers must make conservative assumptions about the behavior of code. For example, if a C function has a type signature `void f(int* x, int* y)` then `x` and `y` may possibly point to the same data, and so reads and writes cannot be freely reordered. However, if a Rust function has a type signature

`fn f(x: &mut i32, y: &mut i32)`, then `x` cannot be an alias of `y` by the rules of ownership, and so a compiler can more aggressively optimize instructions involving one but not the other (see Jung et al. [145] for the details).

Information flow analysis is essentially about constructing the dependency graph of the instructions in a function. Therefore a compiler could theoretically use a tool like Flowistry to reorder instructions, automatically parallelize blocks of code, and eliminate dead code with greater precision than before. Future work can investigate the magnitude of potential performance improvements to be gained with information-flow-guided optimizations.

7.3 Slicing at Scale

Finally, the program slicer itself should also the subject of future work. The study in Chapter 6 only focused on programmers reading small, self-contained program. This style of experiment is good for finding patterns of experience, but it reduces the ecological validity of the findings since most programmers do not work on small, self-contained programs. Therefore one next step is to study professional Rust developers using Focus Mode in their day-to-day work.

Based on the interviews with user study participants in Section 6.2.3, I can anticipate a number of possible refinements to Focus Mode that would help users in large codebases. In codebases where functionality is factored into small units, then the set of code relevant to a programmer's task may cross dozens of functions and files. A modular slicer will probably not help a programmer in this setting. One potentially useful mechanism may be progressively enlarging a slice as a user jumps from one definition to the next.

Part of this study should also focus on building a theory of relevance in programming. For example, the codebase is just one of many information sources that a professional developer uses — they also have design documents, reference documentation, customer communications, and more. The slicers of the future would ideally capture the entire vertical of information: everything a programmer needs to accomplish their task.

Bibliography

[1] Roberto Minelli, Andrea Mocci, and Michele Lanza. I know what you did last summer - an investigation of how developers spend their time. In *2015 IEEE 23rd International Conference on Program Comprehension*, pages 25–35, 2015. doi: 10.1109/ICPC.2015.12.

[2] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering*, 44(10):951–976, 2018. doi: 10.1109/TSE.2017.2734091.

[3] Consortium for Information & Software Quality. The cost of poor software quality in the us: A 2020 report. https://www.it-cisq.org/pdf/CPSQ-2020-report.pdf, 2020.

[4] Christopher D Wickens, William S Helton, Justin G Hollands, and Simon Banbury. *Engineering psychology and human performance*. Routledge, 2021.

[5] Colin Ware. *Information visualization: perception for design*. Morgan Kaufmann, 2019.

[6] Stuart K. Card, Thomas P. Moran, and Allen Newell. *The Psychology of Human-Computer Interaction*. L. Erlbaum Associates Inc., USA, 1983. ISBN 0898592437.

[7] Alan Baddeley. Working memory. *Current Biology*, 20(4):R136–R140, 2010. ISSN 0960-9822. doi: https://doi.org/10.1016/j.cub.2009.12.014. URL https://www.sciencedirect.com/science/article/pii/S0960982209021332.

[8] Advait Sarkar. The impact of syntax colouring on program comprehension. In *Proceedings of the 26th Annual Conference of the Psychology of Programming Interest Group (PPIG)*, pages 49–58, 2015.

[9] Amy J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on Software Engineering*, 32(12):971–987, 2006. doi: 10.1109/TSE.2006.116.

[10] T. R. G. Green. Cognitive dimensions of notations. In *Proceedings of the Fifth Conference of the British Computer Society, Human-Computer Interaction Specialist Group on People and Computers V*, page 443–460, USA, 1990. Cambridge University Press. ISBN 0521384303.

[11] Mark Weiser. Programmers use slices when debugging. *Commun. ACM*, 25 (7):446–452, jul 1982. ISSN 0001-0782. doi: 10.1145/358557.358577. URL https://doi.org/10.1145/358557.358577.

[12] Mark Weiser and Jim Lyle. Experiments on slicing-based debugging aids. In *Papers Presented at the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers*, page 187–197, USA, 1986. Ablex Publishing Corp. ISBN 089391388X.

[13] Shinji Kusumoto, Akira Nishimatsu, Keisuke Nishie, and Katsuro Inoue. Experimental evaluation of program slicing for fault localization. *Empirical Software Engineering*, 7(1):49–76, 2002. doi: 10.1023/a:1014823126938. URL https://doi.org/10.1023/a:1014823126938.

[14] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Resource aware ml. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, page 781–786, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 9783642314230. doi: 10.1007/978-3-642-31424-7_64. URL https://doi.org/10.1007/978-3-642-31424-7_64.

[15] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1), apr 2016. ISSN 0360-0300. doi: 10.1145/2873052. URL https://doi.org/10.1145/2873052.

[16] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, page 147–160, New York, NY, USA, 1999. Association for Computing Machinery. ISBN 1581130953. doi: 10.1145/292540.292555. URL https://doi.org/10.1145/292540.292555.

[17] Thomas RG Green. Cognitive dimensions of notations. *People and computers V*, 5(1):443–460, 1989.

[18] Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. Silq: A high-level quantum language with safe uncomputation and intuitive semantics. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 286–300, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450376136. doi: 10.1145/3385412.3386007. URL https://doi.org/10.1145/3385412.3386007.

[19] Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. Monotasks: Architecting for performance clarity in data analytics frameworks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 184–200, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350853. doi: 10.1145/3132747.3132766. URL https://doi.org/10.1145/3132747.3132766.

[20] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, page 305–320, USA, 2014. USENIX Association. ISBN 9781931971102.

[21] Gracy Murray Hopper. *Keynote Address*, page 7–20. Association for Computing Machinery, New York, NY, USA, 1978. ISBN 0127450408. URL https://doi.org/10.1145/800025.1198341.

[22] Edsger W. Dijkstra. Letters to the editor: Go to statement considered harmful. *Commun. ACM*, 11(3):147–148, mar 1968. ISSN 0001-0782. doi: 10.1145/362929.362947. URL https://doi.org/10.1145/362929.362947.

[23] Donald E. Knuth. An empirical study of FORTRAN programs. *Software: Practice and Experience*, 1(2):105–133, April 1971. ISSN 00380644, 1097024X. doi: 10.1002/spe.4380010203. URL http://doi.wiley.com/10.1002/spe.4380010203.

[24] C. A. R. Hoare. Hints on programming language design. Technical report, Stanford, CA, USA, 1973.

[25] M.E. Sime, T.R.G. Green, and D.J. Guest. Psychological evaluation of two conditional constructions used in computer languages. *International Journal of Man-Machine Studies*, 5(1):105–113, 1973. ISSN 0020-7373. doi: https://doi.org/10.1016/S0020-7373(73)80011-2. URL https://www.sciencedirect.com/science/article/pii/S0020737373800112.

[26] Max E. Sime, Thomas R. G. Green, and DJ Guest. Scope marking in computer conditionals—a psychological evaluation. *International Journal of Man-Machine Studies*, 9(1):107–118, 1977.

[27] T.R.G. Green. Conditional program statements and their comprehensibility to professional programmers. *Journal of Occupational Psychology*, 50(2):93–109, 1977. doi: https://doi.org/10.1111/j.2044-8325.1977.tb00363.x. URL https://bpspsychub.onlinelibrary.wiley.com/doi/abs/10.1111/j.2044-8325.1977.tb00363.x.

[28] J. D. Gannon. An experimental evaluation of data type conventions. *Commun. ACM*, 20(8):584–595, aug 1977. ISSN 0001-0782. doi: 10.1145/359763.359800. URL https://doi.org/10.1145/359763.359800.

[29] Ben Shneiderman, Richard Mayer, Don McKay, and Peter Heller. Experimental investigations of the utility of detailed flowcharts in programming. *Commun. ACM*, 20(6):373–381, jun 1977. ISSN 0001-0782. doi: 10.1145/359605.359610. URL https://doi.org/10.1145/359605.359610.

[30] Lawrence Thomas Love. *Relating individual differences in computer programming performance to human information processing abilities.* PhD thesis, University of Washington, 1977.

[31] B. A. Sheil. The psychological study of programming. *ACM Comput. Surv.*, 13(1):101–120, mar 1981. ISSN 0360-0300. doi: 10.1145/356835.356840. URL https://doi.org/10.1145/356835.356840.

[32] George A Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological review*, 63(2):81, 1956.

[33] William G. Chase and K. Anders Ericsson. Skill and working memory. In Gordon H. Bower, editor, *Psychology of Learning and Motivation*, volume 16, pages 1–58. Academic Press, 1982. doi: https://doi.org/10.1016/S0079-7421(08)60546-0. URL https://www.sciencedirect.com/science/article/pii/S0079742108605460.

[34] William G. Chase and Herbert A. Simon. Perception in chess. *Cognitive Psychology*, 4(1):55–81, 1973. ISSN 0010-0285. doi: https://doi.org/10.1016/0010-0285(73)90004-2. URL https://www.sciencedirect.com/science/article/pii/0010028573900042.

[35] Beth Adelson. Problem solving and the development of abstract categories in programming languages. *Memory & Cognition*, 9(4):422–433, July 1981. ISSN 0090-502X, 1532-5946. doi: 10.3758/BF03197568. URL http://www.springerlink.com/index/10.3758/BF03197568.

[36] Katherine B. McKeithen, Judith S. Reitman, Henry H. Rueter, and Stephen C. Hirtle. Knowledge organization and skill differences in computer programmers. *Cognitive Psychology*, 13(3):307–325, July 1981. ISSN 00100285.

doi: 10.1016/0010-0285(81)90012-8. URL https://linkinghub.elsevier.com/retrieve/pii/0010028581900128.

[37] David Rumelhart. Schemata: The building blocks of cognition. In *Theoretical Issues in Reading Comprehension*. Routledge, 1980.

[38] P. C. Wason. Reasoning about a rule. *Quarterly Journal of Experimental Psychology*, 20(3):273–281, 1968. doi: 10.1080/14640746808400161. URL https://doi.org/10.1080/14640746808400161.

[39] Richard A. Griggs and James R. Cox. The elusive thematic-materials effect in wason's selection task. *British Journal of Psychology*, 73 (3):407–420, 1982. doi: https://doi.org/10.1111/j.2044-8295.1982.tb01823.x. URL https://bpspsychub.onlinelibrary.wiley.com/doi/abs/10.1111/j.2044-8295.1982.tb01823.x.

[40] J.S.B.T. Evans. *Hypothetical Thinking*. Essays in Cognitive Psychology. Routledge, London, England, May 2007.

[41] Stanley Letovsky. Cognitive processes in program comprehension. *Journal of Systems and software*, 7(4):325–339, 1987.

[42] John R. Anderson, Robert Farrell, and Ron Sauers. Learning to program in lisp. *Cognitive Science*, 8(2):87–129, 1984. ISSN 0364-0213. doi: https://doi.org/10.1016/S0364-0213(84)80013-0. URL https://www.sciencedirect.com/science/article/pii/S0364021384800130.

[43] James C. Spohrer, Elliot Soloway, and Edgar Pope. A goal/plan analysis of buggy pascal programs. *Human–Computer Interaction*, 1(2):163–207, 1985. doi: 10.1207/s15327051hci0102_4. URL https://doi.org/10.1207/s15327051hci0102_4.

[44] E. Soloway. Learning to program = learning to construct mechanisms and explanations. *Commun. ACM*, 29(9):850–858, sep 1986. ISSN 0001-0782. doi: 10.1145/6592.6594. URL https://doi.org/10.1145/6592.6594.

[45] Kathi Fisler, Shriram Krishnamurthi, and Janet Siegmund. Modernizing Plan-Composition Studies. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education - SIGCSE '16*, pages 211–216, Memphis, Tennessee, USA, 2016. ACM Press. ISBN 978-1-4503-3685-7. doi: 10.1145/2839509.2844556. URL http://dl.acm.org/citation.cfm?doid=2839509.2844556.

[46] Rodrigo Duran, Juha Sorva, and Sofia Leite. Towards an Analysis of Program Complexity From a Cognitive Perspective. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*, pages 21–30, Espoo Finland, August 2018. ACM. ISBN 978-1-4503-5628-2. doi: 10.1145/3230977.3230986. URL https://dl.acm.org/doi/10.1145/3230977.3230986.

[47] Kathryn Cunningham. Purpose-first programming: A programming learning approach for learners who care most about what code achieves. In *Proceedings of the 2020 ACM Conference on International Computing Education Research*, ICER '20, page 348–349, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370929. doi: 10.1145/3372782.3407102. URL https://doi.org/10.1145/3372782.3407102.

[48] J.-M Hoc, T.R.G. Green, R. Samurçay, and D.J. Gilmore, editors. *Psychology of Programming*. Elsevier, 1990. doi: 10.1016/c2009-0-21596-0. URL https://doi.org/10.1016/c2009-0-21596-0.

[49] Thomas K. Landauer. Let's Get Real: A Position Paper on the Role of Cognitive Psychology in the Design of Humanly Useful and Usable Systems. In *Readings in Human–Computer Interaction*, pages 659–665. Elsevier, 1995. ISBN 978-0-08-051574-8. doi: 10.1016/B978-0-08-051574-8.50067-4. URL https://linkinghub.elsevier.com/retrieve/pii/B9780080515748500674.

[50] Felienne Hermans. *The Programmer's Brain: What every programmer needs to know about cognition*. Manning Publications, New York, NY, December 2021.

[51] L. Prechelt, B. Unger-Lamprecht, M. Philippsen, and W.F. Tichy. Two controlled experiments assessing the usefulness of design pattern documentation

in program maintenance. *IEEE Transactions on Software Engineering*, 28(6): 595–606, 2002. doi: 10.1109/TSE.2002.1010061.

[52] T.R.G. Green, S.P. Davies, and D.J. Gilmore. Delivering cognitive psychology to HCI: the problems of common language and of knowledge transfer. *Interacting with Computers*, 8(1):89–111, 03 1996. ISSN 0953-5438. doi: 10.1016/0953-5438(95)01020-3. URL https://doi.org/10.1016/0953-5438(95)01020-3.

[53] Amy J. Ko and Brad A. Myers. Designing the whyline: A debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '04, page 151–158, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581137028. doi: 10.1145/985692.985712. URL https://doi.org/10.1145/985692.985712.

[54] Michael Bostock and Jeffrey Heer. Protovis: A graphical toolkit for visualization. *IEEE Transactions on Visualization and Computer Graphics*, 15(6): 1121–1128, 2009. doi: 10.1109/TVCG.2009.174.

[55] Brad A. Myers, Amy J. Ko, Thomas D. LaToza, and YoungSeok Yoon. Programmers are users too: Human-centered methods for improving programming tools. *Computer*, 49(7):44–52, 2016. doi: 10.1109/MC.2016.200.

[56] Janet Siegmund, Norman Peitek, André Brechmann, Chris Parnin, and Sven Apel. Studying programming in the neuroage: Just a crazy idea? *Commun. ACM*, 63(6):30–34, may 2020. ISSN 0001-0782. doi: 10.1145/3347093. URL https://doi.org/10.1145/3347093.

[57] Norman Peitek, Janet Siegmund, Sven Apel, Christian Kästner, Chris Parnin, Anja Bethmann, Thomas Leich, Gunter Saake, and André Brechmann. A look into programmers' heads. *IEEE Transactions on Software Engineering*, 46(4): 442–462, 2020. doi: 10.1109/TSE.2018.2863303.

[58] Briana B. Morrison, Lauren E. Margulieux, and Mark Guzdial. Subgoals, context, and worked examples in learning computing problem solving. In

Proceedings of the Eleventh Annual International Conference on International Computing Education Research, ICER '15, page 21–29, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336307. doi: 10.1145/2787622.2787733. URL https://doi.org/10.1145/2787622.2787733.

[59] Greg L. Nelson and Amy J. Ko. On use of theory in computing education research. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*, ICER '18, page 31–39, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356282. doi: 10.1145/3230977.3230992. URL https://doi.org/10.1145/3230977.3230992.

[60] Martin J Pickering and Simon Garrod. Do people use language production to make predictions during comprehension? *Trends in cognitive sciences*, 11(3): 105–110, 2007.

[61] Margaret Wilson and Günther Knoblich. The case for motor involvement in perceiving conspecifics. *Psychological bulletin*, 131(3):460, 2005.

[62] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. Relationships between reading, tracing and writing skills in introductory programming. In *Proceedings of the fourth international workshop on computing education research*, pages 101–112, Sydney, Australia, 2008. ACM.

[63] Raymond Lister, Colin Fidge, and Donna Teague. Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. *Acm sigcse bulletin*, 41(3):161–165, 2009.

[64] Juha Sorva. Notional machines and introductory programming education. *Trans. Comput. Educ*, 13(2):1–31, 2007.

[65] Françoise Détienne and Elliot Soloway. An empirically-derived control structure for the process of program understanding. *International Journal of Man-Machine Studies*, 33(3):323–342, 1990.

[66] Vesa Vainio and Jorma Sajaniemi. Factors in novice programmers' poor tracing skills. *ACM SIGCSE Bulletin*, 39(3):236–240, 2007.

[67] Diana DeStefano and Jo-Anne LeFevre. The role of working memory in mental arithmetic. *European Journal of Cognitive Psychology*, 16(3):353–386, 2004.

[68] Graham J Hitch. The role of short-term working memory in mental arithmetic. *Cognitive Psychology*, 10(3):302–323, 1978.

[69] Jiajie Zhang and Donald A Norman. A representational analysis of numeration systems. *Cognition*, 57(3):271–295, 1995.

[70] Jamie ID Campbell and Neil Charness. Age-related declines in working-memory skills: Evidence from a complex calculation task. *Developmental Psychology*, 26(6):879, 1990.

[71] Peter A Nobel and Richard M Shiffrin. Retrieval processes in recognition and cued recall. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 27(2):384, 2001.

[72] Nash Unsworth. Examining variation in working memory capacity and retrieval in cued recall. *Memory*, 17(4):386–396, 2009.

[73] Ansgar J FÜrst and Graham J Hitch. Separate roles for executive and phonological components of working memory in mental arithmetic. *Memory & cognition*, 28(5):774–782, 2000.

[74] Teresa Busjahn, Roman Bednarik, Andrew Begel, Martha Crosby, James H Paterson, Carsten Schulte, Bonita Sharif, and Sascha Tamm. Eye movements in code reading: Relaxing the linear order. In *2015 IEEE 23rd International Conference on Program Comprehension*, pages 255–265, Florence, Italy, 2015. IEEE, IEEE.

[75] Norman Peitek, Janet Siegmund, and Sven Apel. What drives the reading order of programmers? an eye tracking study. In *Proceedings of the 28th International*

Conference on Program Comprehension, pages 342–353, Seoul, South Korea, 2020. IEEE/ACM.

[76] John R Anderson and Robin Jeffries. Novice lisp errors: Undetected losses of information from working memory. *Human–Computer Interaction*, 1(2):107–131, 1985.

[77] Paul Ayres and John Sweller. Locus of difficulty in multistage mathematics problems. *The American Journal of Psychology*, 103(2):167–193, 1990.

[78] Paul L Ayres. Systematic mathematical errors and cognitive load. *Contemporary Educational Psychology*, 26(2):227–248, 2001.

[79] Alan Baddeley. Working memory. *Current biology*, 20(4):R136–R140, 2010.

[80] Neil Morris and Dylan M Jones. Memory updating in working memory: The role of the central executive. *British journal of psychology*, 81(2):111–121, 1990.

[81] Anthony R Jansen, Alan F Blackwell, and Kim Marriott. A tool for tracking visual attention: The restricted focus viewer. *Behavior research methods, instruments, & computers*, 35(1):57–69, 2003.

[82] Sara Steegen, Francis Tuerlinckx, Andrew Gelman, and Wolf Vanpaemel. Increasing transparency through a multiverse analysis. *Perspectives on Psychological Science*, 11(5):702–712, 2016.

[83] Anneliese Von Mayrhauser and A Marie Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.

[84] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, Los Angeles, CA, 1977. ACM.

[85] Google. Google c++ style guide, 2020. URL https://google.github.io/styleguide/cppguide.html.

[86] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In *International Conference on Automated Deduction*, pages 378–388, Berlin, Germany, 2015. Springer, CADE.

[87] Robert Bruce Findler. Drracket: The racket programming environment, 2020. URL https://docs.racket-lang.org/drracket/.

[88] Edwin Hutchins. *Cognition in the Wild*. MIT press, Cambridge, MA, 1995.

[89] Barbara Tversky, Maneesh Agrawala, Julie Heiser, Paul Lee, Pat Hanrahan, Phan Doantam, Chris Stolte, and Marie-Paule Daniel. Cognitive design principles for automated generation of visualizations. In *Applied Spatial Cognition*, pages 53–74. Psychology Press, 2020.

[90] Maneesh Agrawala and Chris Stolte. Rendering effective route maps: Improving usability through generalization. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, page 241–249, New York, NY, USA, 2001. Association for Computing Machinery. ISBN 158113374X. doi: 10.1145/383259.383286. URL https://doi.org/10.1145/383259.383286.

[91] Adriaan D. de Groot. *Thought and Choice in Chess*. The Hague: Mouton, 1965. ISBN 9789053569986.

[92] Jonathan St. B. T. Evans. The heuristic-analytic theory of reasoning: Extension and evaluation. *Psychonomic Bulletin & Review*, 13(3):378–395, June 2006. doi: 10.3758/bf03193858. URL https://doi.org/10.3758/bf03193858.

[93] Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola. Code bubbles: A working set-based interface for code understanding

and maintenance. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, page 2503–2512, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605589299. doi: 10.1145/1753326.1753706. URL https://doi.org/10.1145/1753326.1753706.

[94] Mik Kersten and Gail C. Murphy. Using task context to improve programmer productivity. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, page 1–11, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595934685. doi: 10.1145/1181775.1181777. URL https://doi.org/10.1145/1181775.1181777.

[95] Mark Weiser. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, University of Michigan, 1979.

[96] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, 1982.

[97] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, page 439–449. IEEE Press, 1981. ISBN 0897911466.

[98] John D. Gould. Some psychological evidence on how people debug computer programs. *International Journal of Man-Machine Studies*, 7(2):151–182, 1975. ISSN 0020-7373. doi: https://doi.org/10.1016/S0020-7373(75)80005-8. URL https://www.sciencedirect.com/science/article/pii/S0020737375800058.

[99] Edsger W. Dijkstra. *Correctness Concerns and, among Other Things, Why They Are Resented*, pages 80–88. Springer New York, New York, NY, 1978. ISBN 978-1-4612-6315-9. doi: 10.1007/978-1-4612-6315-9_8. URL https://doi.org/10.1007/978-1-4612-6315-9_8.

[100] F.J. Lukey. Understanding and debugging programs. *International Journal of Man-Machine Studies*, 12(2):189–202, 1980. ISSN 0020-7373. doi: https://doi.org/10.1016/S0020-7373(80)80017-4. URL https://www.sciencedirect.com/science/article/pii/S0020737380800174.

[101] Margaret Ann Francel and Spencer Rugaber. The value of slicing while debugging. *Science of Computer Programming*, 40(2):151–169, 2001. ISSN 0167-6423. doi: https://doi.org/10.1016/S0167-6423(01)00013-2. URL https://www.sciencedirect.com/science/article/pii/S0167642301000132. Special Issue on Program Comprehension.

[102] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 35–46, 1988.

[103] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. *SIGPLAN Not.*, 39(6):131–144, jun 2004. ISSN 0362-1340. doi: 10.1145/996893.996859. URL https://doi.org/10.1145/996893.996859.

[104] Gagan Agrawal and Liang Guo. Evaluating explicitly context-sensitive program slicing. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, page 6–12, New York, NY, USA, 2001. Association for Computing Machinery. ISBN 1581134134. doi: 10.1145/379605.379630. URL https://doi.org/10.1145/379605.379630.

[105] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, page 49–61, New York, NY, USA, 1995. Association for Computing Machinery. ISBN 0897916921. doi: 10.1145/199448.199462. URL https://doi.org/10.1145/199448.199462.

[106] Marek Chalupa. Slicing of llvm bitcode. Master's thesis, Masaryk University, 2016.

[107] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988. ISSN 0020-0190. doi: https://doi.org/10.1016/0020-0190(88)90054-3. URL https://www.sciencedirect.com/science/article/pii/0020019088900543.

[108] A. De Lucia, A.R. Fasolino, and M. Munro. Understanding function behaviors through program slicing. In *WPC '96. 4th Workshop on Program Comprehension*, pages 9–18, 1996. doi: 10.1109/WPC.1996.501116.

[109] M. Harman and S. Danicic. Amorphous program slicing. In *Proceedings Fifth International Workshop on Program Comprehension. IWPC'97*, pages 70–79, 1997. doi: 10.1109/WPC.1997.601266.

[110] Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. Thin slicing. *SIGPLAN Not.*, 42(6):112–122, jun 2007. ISSN 0362-1340. doi: 10.1145/1273442.1250748. URL https://doi.org/10.1145/1273442.1250748.

[111] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 international symposium on software testing and analysis*, pages 199–209, 2011.

[112] David Binkley, Nicolas Gold, and Mark Harman. An empirical study of static program slice size. *ACM Trans. Softw. Eng. Methodol.*, 16(2):8–es, apr 2007. ISSN 1049-331X. doi: 10.1145/1217295.1217297. URL https://doi.org/10.1145/1217295.1217297.

[113] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c. In *International conference on software engineering and formal methods*, pages 233–247. Springer, 2012.

[114] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, may 1976. ISSN 0001-0782. doi: 10.1145/360051.360056. URL https://doi.org/10.1145/360051.360056.

[115] Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, page 228–241, New York, NY, USA, 1999. Association for Computing Machinery. ISBN 1581130953. doi: 10.1145/292540.292561. URL https://doi.org/10.1145/292540.292561.

[116] François Pottier and Vincent Simonet. Information flow inference for ml. *ACM Trans. Program. Lang. Syst.*, 25(1):117–158, jan 2003. ISSN 0164-0925. doi: 10.1145/596980.596983. URL https://doi.org/10.1145/596980.596983.

[117] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, 1987. doi: 10.1016/0304-3975(87)90045-4. URL https://doi.org/10.1016/0304-3975(87)90045-4.

[118] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '98, page 48–64, New York, NY, USA, 1998. Association for Computing Machinery. ISBN 1581130058. doi: 10.1145/286936.286947. URL https://doi.org/10.1145/286936.286947.

[119] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, page 282–293, New York, NY, USA, 2002. Association for Computing Machinery. ISBN 1581134630. doi: 10.1145/512529.512563. URL https://doi.org/10.1145/512529.512563.

[120] Aaron Weiss, Olek Gierczak, Daniel Patterson, and Amal Ahmed. Oxide: The essence of rust, 2019.

[121] Yannis Smaragdakis and George Balatsouras. Pointer analysis. *Foundations and Trends in Programming Languages*, 2(1):1–69, 2015.

[122] J. A. Goguen and J. Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy*. IEEE, April 1982. doi: 10.1109/sp.1982.10014. URL https://doi.org/10.1109/sp.1982.10014.

[123] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, jul 1987. ISSN 0164-0925. doi: 10.1145/24039.24041. URL https://doi.org/10.1145/24039.24041.

[124] Keith D Cooper, Timothy J Harvey, and Ken Kennedy. A simple, fast dominance algorithm. *Software Practice & Experience*, 4(1-10):1–8, 2001.

[125] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, page 25–35, New York, NY, USA, 1989. Association for Computing Machinery. ISBN 0897912942. doi: 10.1145/75277.75280. URL https://doi.org/10.1145/75277.75280.

[126] Niko Matsakis. Non-lexical lifetimes, 2017. URL https://rust-lang.github.io/rfcs/2094-nll.html.

[127] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. How do programmers use unsafe rust? *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020. doi: 10.1145/3428204. URL https://doi.org/10.1145/3428204.

[128] Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. Is rust used safely by software developers? In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 246–257, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371216. doi: 10.1145/3377811.3380413. URL https://doi.org/10.1145/3377811.3380413.

[129] Al Danial. cloc: Count lines of code, 2021. URL https://github.com/AlDanial/cloc.

[130] Marc Shapiro and Susan Horwitz. The effects of the precision of pointer analysis. In *Proceedings of the 4th International Symposium on Static Analysis*, SAS '97, page 16–34, Berlin, Heidelberg, 1997. Springer-Verlag. ISBN 3540634681.

[131] Françoise Détienne and Elliot Soloway. An empirically-derived control structure for the process of program understanding. *International Journal of Man-Machine Studies*, 33(3):323–342, September 1990. doi: 10.1016/s0020-7373(05)80122-1. URL https://doi.org/10.1016/s0020-7373(05)80122-1.

[132] Stanley Letovsky. Cognitive processes in program comprehension. *Journal of Systems and Software*, 7(4):325–339, 1987. ISSN 0164-1212. doi: https://doi.org/10.1016/0164-1212(87)90032-X. URL https://www.sciencedirect.com/science/article/pii/016412128790032X.

[133] Kathy Charmaz. *Constructing grounded theory: A practical guide through qualitative analysis*. SAGE Publishing Inc., 2006.

[134] Justin Lubin and Sarah E. Chasins. How statically-typed functional programmers write code. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021. doi: 10.1145/3485532. URL https://doi.org/10.1145/3485532.

[135] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016. doi: 10.1109/TSE.2016.2521368.

[136] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, page 439–449. IEEE Press, 1984. ISBN 0897911466.

[137] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, page 199–209, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450305624. doi: 10.1145/2001420.2001445. URL https://doi.org/10.1145/2001420.2001445.

[138] Sandrine Blazy and Philippe Facon. Partial evaluation for program comprehension. *ACM Comput. Surv.*, 30(3es):17–es, sep 1998. ISSN 0360-0300. doi: 10.1145/289121.289138. URL https://doi.org/10.1145/289121.289138.

[139] S.P. Reiss. The Paradox of Software Visualization. In *3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 1–5, Budapest, Hungary, 2005. IEEE. ISBN 978-0-7803-9540-4. doi: 10.1109/VISSOF.2005.1684306. URL http://ieeexplore.ieee.org/document/1684306/.

[140] Leonel Merino, Mohammad Ghafari, and Oscar Nierstrasz. Towards Actionable Visualisation in Software Development. In *2016 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 61–70, Raleigh, NC, USA, October 2016. IEEE. ISBN 978-1-5090-3850-3. doi: 10.1109/VISSOFT.2016.10. URL http://ieeexplore.ieee.org/document/7780158/.

[141] Gagan Agrawal and Liang Guo. Evaluating explicitly context-sensitive program slicing. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 6–12, 2001.

[142] Julie Heiser, Doantam Phan, Maneesh Agrawala, Barbara Tversky, and Pat Hanrahan. Identification and validation of cognitive design principles for automated generation of assembly instructions. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, AVI '04, page 311–319, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581138679. doi: 10.1145/989863.989917. URL https://doi.org/10.1145/989863.989917.

[143] Martin P. Robillard and Robert Deline. A field study of api learning obstacles. *Empirical Softw. Engg.*, 16(6):703–732, dec 2011. ISSN 1382-3256. doi: 10.1007/s10664-010-9150-8. URL https://doi.org/10.1007/s10664-010-9150-8.

[144] George Lakoff and Rafael Nunez. *Where mathematics come from*. Basic Books, London, England, July 2001.

[145] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. Stacked borrows: An aliasing model for rust. *Proc. ACM Program. Lang.*, 4(POPL), dec 2019. doi: 10.1145/3371109. URL https://doi.org/10.1145/3371109.