IT UNIVERSITY OF COPENHAGEN

Department of Computer Science
Data, Systems, and Robotics

PhD Thesis

Rogers: Writing Verified Programs for Computational Storage

Author:
Morten Clausen

Supervisors:
Jesper Bengtson
Pınar Tözün
Philippe Bonnet

Submitted on
October 31st, 2025

```
Summarizing:  as a slow-witted human being I have a very
small head and I had better learn to live with it and to
respect my limitations and give them full credit, rather
than to try to ignore them, for the latter vain effort will
be punished by failure.
```

- Edsger W. Dijkstra, Notes on Structed Programming, January 1972

Abstract

Computational storage is the idea of pushing computations to the storage device containing the data of interest rather than loading the data onto a host device and then performing the computation. This approach aims to minimize data movement and increase performance by allowing computations to make better use of the hardware found on the storage device.

This approach to data modification is still in its infancy, and as such there are many open research questions regarding the architecture and implementation of such a setup. Some of those questions have been posed by Barbalace & Do [10], of which this thesis focuses on: How to program applications? How to manage shared resources? How to ensure application are safe to execute on the storage device?

This thesis takes a step towards answering the above questions through the following contributions.

- A programming paradigm which separates data management and computations into two distinct domains. This allows storage providers to control what kind of computations are executed on their devices, while still providing users with the ability to configure the computations to their desire.
- A domain-specific language (DSL) for expressing data placements and invoking computations. This is accompanied by Rogers, a type checker that validates applications written in the DSL. The DSL combines its type system with region-based memory management, enabling Rogers to validate that no buffer overflows occur. The user is thus freed from worrying about the correctness of low-level memory details.
- A resource manager to provide shared memory access to applications on computational storage devices. Concurrency with respect to SSDs is an often overlooked topic, so this contribution serves to take a first step toward addressing the topic. The manager is co-designed alongside the DSL and utilizes a deterministic queue system to guarantee the absence of deadlocks and eventual access for applications.

The contributions result in storage vendors being able to market their computational storage devices as more trustworthy, service providers being able to offer computational services without worries of security, and application developers having an easy way to interact with computational storage for increased performance.

Resumé

Computational Storage (Beregningsmæssigt Lager) er et koncept om at overføre beregninger til det lagringsmedie som opbevarer den relevante data, fremfor at hente dataen op til værtsenheden og først derefter udføre beregningerne. Denne tilgang har som mål at minimere dataoverførsler og forøge hastigheden af beregningerne gennem bedre udnyttelse af den tilgængelige hardware på lagringsmediet.

Denne tilgang indenfor databehandling er stadig forholdsvis ny, og der er derfor stadig mange ubesvarede forskningsspørgsmål angående arkitekturen og implementeringen af sådan en løsning. Nogle af disse spørgsmål er blevet fremsat af Barbalace & Do [10], hvoraf jeg fokuserer på: Hvordan programmerer man applikationer til computational storage? Hvordan håndterer man deling af ressourcer mellem applikationer? Hvordan sikrer man at det er sikkert at eksekverer applikationerne på lagringsmediet?

Denne afhandling tager et skridt mod at besvare ovenstående spørgsmål ved hjælp af følgende bidrag.

- Et programmeringsparadigme der adskiller data administrering og beregninger i to separate domæner. Dette tillader lagerudbydere at kontrollere hvilken slags beregninger der kan udføres på deres enheder, samtidigt med at brugere stadig har evnen til at konfigurere beregningerne efter behov.
- Et domain specific language (DSL, domæne-specifikt sprog) til at udtrykke data-placeringer. Sammen med dette viser jeg Rogers, et program der validerer applikationer skrevet i førnævnte DSL. Sproget kombinerer et type system med regionsbaseret behandling af maskinhukommelse, hvilket tillader Rogers at afgøre om der sker buffer overflows.
- En ressource-administrator der tildeler adgang til ressourcer delt mellem applikationer. Samtidig programeksekvering på SSD'er er et ofte overset emne, så dette bidrag yder at tage første skridt mod adressering af emnet. Administratoren er designet sideløbende med DSL'en og anvender et deterministisk kø-system for at garanterer at der ikke opstår dead-locks, og at applikationer altid opnår adgang.

Bidragene resulterer i at lagerudbyder kan markedsføre deres beregningsmæssige lagerenheder som pålidelige, service-udbydere kan tilbyde beregningsmæssige services uden frygt for sikkerhedsbrud, og applikationsudviklere får en let indgangsmulighed til at inddrage beregningsmæssigt lager i deres applikationer for øget ydeevne.

Acknowledgements

I would like to thank the people in DSAR for creating a sociable environment, with special regards to my fellow PhD friends: Kasper, who, even though he was contractually obliged to offer his friendship, still managed to look like he enjoyed it somewhat. Ties, sharing not only his birthday with me, but also his thoughts and invaluable support. Dovile, who is excellent at both brewing and sharing tea. The three of you made the PhD journey enjoyable straight from the beginning.

Outside of DSAR, I would like to also thank Holger Borum and Peter Sestoft for allowing me to work as a research assistant before my PhD, giving me a headstart on doing research. Through them, I got acquainted with the Square group and met Adrian, with his special interests in Volapük and military history, and Laura, who would later go on to bring about the creation of the PhD club at ITU, for which I am very grateful.

Thank you to 김지응 교수님, for inviting me to Korea, being a research partner and essentially an additional supervisor. Your advice about my research and academia as a whole has helped me a lot. To 김종학, 박찬준, 이준용, 송해인, 김소연, 남주현 and 문순원, thank you for helping me adjust while I was in Korea.
한국 문화를 공유해줘서 정말 고맙습니다!

Lastly, I would like to thank my supervisors for listening to my needs, guiding me on how to create the best possible project, and taking a genuine interest in my work.

Contents

Abstract **iii**

Resumé **v**

Acknowledgements **vii**

Contents **xi**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Roadmap . . . 3

2 Background **5**
2.1 The Inner Workings of the SSD . . . 5
2.2 Modifiable SSDs . . . 6
2.3 Code Offload . . . 8
2.4 Program Verification . . . 9
2.5 Design Decisions . . . 11
2.6 Setup & Terminology . . . 14

3 The Rogers DSL **17**
3.1 Memory Management . . . 17
3.2 Syntax . . . 19
3.3 Type System . . . 24
3.3.1 Dependent Types . . . 25
3.3.2 Typing Rules . . . 26
3.3.3 Validation Through Types . . . 32
3.4 Implementation . . . 43
3.4.1 Emitting Code . . . 47
3.5 Discussion . . . 50
3.5.1 Multiple Write Regions . . . 50
3.5.2 Write Regions of Indeterminate Length . . . 51
3.5.3 Reusing Shared Regions . . . 51
3.5.4 Estimating Space Usage . . . 52
3.6 Conclusion . . . 53

4 Registered Functions **55**
4.1 Requirements . . . 56

4.2 Specification & Verification . . . 57
4.2.1 Sum . . . 57
4.2.2 Conversion . . . 61
4.2.3 Filter . . . 66
4.3 Discussion . . . 71
4.3.1 Axiomatic and Inductive Definitions . . . 71
4.3.2 Wrapping Registered Functions . . . 71
4.3.3 Interfacing Verifications . . . 73
4.3.4 Hardware Accelerators . . . 74
4.3.5 Type Signatures of Registered Functions . . . 75
4.4 Conclusion . . . 75

5 Coordinating Resources **77**
5.1 Requirements . . . 78
5.2 Requesting Regions . . . 79
5.3 Conflict Analysis . . . 82
5.4 Safety & Liveness . . . 86
5.5 Discussion . . . 87
5.5.1 Freeing Memory Regions . . . 87
5.5.2 Resizing Shared Regions . . . 88
5.5.3 Multiple Users . . . 88
5.6 Conclusion . . . 89

6 Evaluation **91**
6.1 Experimental Setup . . . 91
6.2 Measuring the Split Between Data and Computation . . . 92
6.3 Verifiability versus Performance . . . 95
6.4 Overhead of the Resource Manager . . . 95
6.5 Composing Functions . . . 97
6.6 Conclusion . . . 98

7 Conclusion & Future Work **99**
7.1 Applications for Compositional Devices . . . 100
7.2 Mechanizing the Type System Proofs . . . 100
7.3 Additional Language Features . . . 100

Bibliography **104**

Appendices **113**

A Registered Functions with ACSL Specification **115**
A.1 Sum . . . 115
A.2 Conversion . . . 117
A.3 Filter . . . 120

B Coq Proof of Unchanged Occurrences Lemma **123**

C Rogers Implementation **127**

Chapter 1

Introduction

1.1 Motivation

System designers are on a never-ending quest to increase the performance of computer systems. The approach has for a long time been to increase the number of resources available in the system, whether that is increasing the number of transistors on a chip, increasing the number of CPUs in a computer, or by joining multiple computers together in a server performing parallel work [7]. However, scaling performance through increased resources has diminishing returns [78]. Transistors cannot be infinitely small, hardware components need to dissipate heat, and the energy costs of multiple computers are prohibitively expensive. These limitations occasionally necessitate a rethinking of how computations are carried out. This is often done by specializing hardware to be more efficient at certain tasks. Examples of this are FPGAs, which can perform computations at a very high rate and low power consumption [46, 77], and GPUs, which offer an unprecedented number of concurrent threads for numerical computations [23, 52, 68, 82, 6, 34, 63, 79]. However, FPGAs lack modularity and GPUs are not suited for tasks with control flow.

One of the newer approaches to efficient computation is near-data processing within storage media [27, 59]. In other words, storage devices equipped with a processor, allowing the execution of programs separately from host devices. The main factor enabling near-data processing is the advancement of solid-state drives (SSDs). SSDs require firmware, known as the flash transition layer (FTL), to operate a flash memory controller for reading and writing flash memory [21, 48]. The FTL manages many aspects of interfacing with flash storage, such as performing address translations, garbage collection, and error correction. This means that SSDs necessarily possess processing capabilities for modifying data. As SSD technology has improved, an opportunity has arisen to use the processing for queries (data retrieval operations) other than reading and writing. The benefit of processing queries internally on the SSD-side is that the bandwidth within the SSD is typically greater than the bandwidth between the host and the SSD [29]. This means that any data query that produces output smaller than the input could, in theory, benefit from being executed on the storage side rather than the host side.

The Storage Networking Industry Association (SNIA) has taken to calling storage architectures that offer near-data processing for **computational storage**, as part

of their efforts to standardize compute resources for storage [1]. Some projects have already attempted to utilize computational storage, by offloading parts of data queries to SSD-hardware to decrease the latency of the query. In 2013, Do et al. investigated the possibility and benefits of running Microsoft SQL queries on a SmartSSD, showing speedups of up to 3x while cutting energy usage by more than half when running queries with small output sizes [29]. When the output was the same size as the input, execution on the smartSSD was 43% slower than regular SSD read and host execution, which highlights the importance of only offloading data-reducing queries. Do went on to prototype a programmable SSD as part of the SoftFlash project [28], to facilitate the hardware necessary for near-data processing in data centers. Other projects, such as the Biscuit framework [38], focused on the usability aspect of near-data processing by enabling programmers to define applications through a data flow programming model and offload them to SSDs with processing capabilities. More recently, companies such as Amazon and ScaleFlux have attempted to commercialize computational storage in their products [12, 73].

While previous work has shown the benefits of computational storage and tried to facilitate its use in data-centric applications, the technology currently does not see widespread adoption. Recent literature reviews propose a list of unanswered issues that are limiting the technology [75, 10]. One of the challenging aspects is the choice of programming model for computational storage. As shown by Do et al. [29], executing computations that do not reduce the amount of transferred data is detrimental to performance, so programmers or a compiler are required to analyze computations before offloading them. Singh et al. also discuss the need for the algorithms used in these computations to be aware of the architecture they are running on, in order to realize the greatest performance gains [75].

A potential avenue for addressing some of the unresolved issues with computational storage lies within the field of formal verification. Formal verification of software consists of proving that a software system matches a certain specification or property, expressed through a logic system. Barbalace & Do mention such properties when discussing the use of a programming language to define storage computations, stating: "[...] a language that provides some sort of formal properties, such as termination and memory safety, is preferred."[10] Relevant examples of formal verification that could be beneficial for computational storage include the use of a type system to estimate the memory usage of programs [24], an annotation language to express the behavior and result of programs [18], and programming language constructs that guarantee memory safety [65].

Based on the above, we identify the following research question:

How can software developers create programs for computational storage in a way that guarantees **(a)** *termination of programs,* **(b)** *memory safety, and* **(c)** *expected output?*

To address these concerns and unlock more potential of computational storage, this thesis proposes and investigates the use of program verification to aid in creating a robust framework for offloading computations to storage that can supply the following properties to users:

⋆ Absence of buffer overflows

⋆ Non-fragmentation of user-placed data

† Functional correctness of the computations

† Size bounds on the amount of data produced by the computations

▲ Deadlock-free access to shared resources

▲ Isolated execution in a concurrent environment

The framework consists of three main contributions: a domain-specific language (DSL) for writing computational storage programs, a methodology for writing function specifications, and a resource manager to orchestrate access to shared resources.

The DSL uses region-based memory management to provide the properties marked by (⋆). Combined with a type system, we show Rogers, a type checker for the DSL that can ensure any well-typed programs produce non-fragmented data. Rogers can additionally analyze accesses to shared memory and emit run-time checks to assert that data well not extend past the boundary of a region.

The function specification supplies the properties marked by (†). The specfication, which is written and verified using Frama-Cs WP plugin[1] for the examples in this thesis, can be used to indicate which areas of memory a function will write to, thus providing bounds on the amount of data written. The specification can also be used to express characteristics of the output using a logic notation, providing a caller with trusted information regarding the result.

The resource manager provides the properties marked by (▲). Using conservative two-phase locking and a queuing system, the resource manager can analyze programs on the computational storage side for concurrency conflicts. If a conflict is detected, the resource manager deterministically delays programs to resolve the conflict.

1.2 Roadmap

- Chapter 2 discusses the state of the art of computational storage, program verification, and how they can interact.
- Chapter 3 defines a DSL to express queries, using a subset of the C language. Additionally, the chapter introduces Rogers, a type checking program that validates whether a program adheres to the typing rules of the DSL.
- Chapter 4 introduces a case study of writing specifications of statically installed functions. It contains examples of how formal verification tools, in this case Frama-C [26], can be used to formally specify function behavior.
- Chapter 5 outlines the design of a resource manager that coordinates access to shared memory between computational storage programs while automatically resolving concurrency conflicts.

[1] https://www.frama-c.com/fc-plugins/wp.html

- Chapter 6 evaluates the performance of programs written in the Rogers DSL and compares them to programs from a previous computational storage project, Delilah [39].
- Chapter 7 summarizes the findings of earlier chapters and provide proposals for future research directions.

Chapter 2

Background

2.1 The Inner Workings of the SSD

Storage has traditionally been a black box for application developers when it comes to workloads that process data. Seeing storage as a component for storing and retrieving data helps with abstraction, but recent research aims to demystify the black box in order to adapt its behavior to fit the workloads.

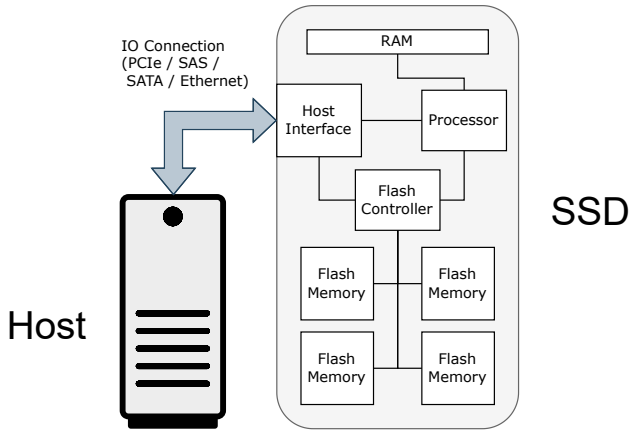


FIGURE 2.1: Traditional SSD setup. A host sends I/O requests to the SSD via the host interface. The flash controller retrieves or stores data from flash memory based on the request.

At the lowest level of an SSD, bits are stored in flash memory cells. These cells are strung together to form a wordline, with multiple wordlines making up a page. The fundamental factor affecting the design of SSDs is the physical constraints of flash memory. Reading and writing can be done at page granularity, but writing to a cell can affect nearby cells, so writing can only be done to blank pages. Overwriting a previously written page requires that the page be erased first, but pages cannot be erased individually. Instead, overwriting old data requires full erasure of all pages within the same block [48]. This difference in write-/erase granularity shapes most of the design choices regarding SSD implementation.

Figure 2.1 shows a simplified overview of the SSD components considered relevant for this thesis. When a host issues an I/O request to the SSD, the request is handled by the flash controller (sometimes also called storage controller or flash memory controller). The controller carries out the request according to the many policies of the storage device, as dictated by the firmware. Collectively, these policies are called the flash transition layer (FTL) and deal with the following non-exhaustive list of tasks:

- *Address Translation*: Requests refer to data through the use of *logical block addresses* (LBA), but the physical location of the requested data is pages within blocks of flash memory. The controller thus maintains a mapping from LBAs to physical locations, allowing for a persistent association of data to LBAs.
- *Garbage Collection*: As mentioned, flash memory cannot be overwritten in the traditional sense, only erased at the block level and then written at the page level. When no blocks are free, the storage controller must evict data from an existing block, by moving the active pages to another block, before erasing the entire block. This process also requires an update to the mapping of LBAs and blocks.
- *Wear Leveling*: Flash memory can only support a limited number of writes before it deteriorates. To increase the longevity of the SSD, some devices utilize block remapping to move data from oft-written blocks to lesser-used blocks, updating the LBA to block map in the process.
- *Error Correction*: Although being fast to use, flash memory has a higher chance of data corruption compared to magnetic storage. To compensate, SSDs employ the use of error correction code, reducing the error rate to an acceptable margin.

SSDs can optionally be equipped with RAM to increase performance speed. The RAM can act as a cache for stored data or provide the controller with fast access to metadata relevant to the above-mentioned policies. Conforming to these policies necessarily introduces some processing overhead. Storage vendors have until now been the party responsible for choosing and implementing these policies, leading to performance goals aimed at general-purpose functionality. However, adapting hardware to specific workloads yields performance gains for those particular workloads, motivating the co-design of storage devices.

2.2 Modifiable SSDs

Adapting the behavior of storage to match workload characteristics can be done in a variety of ways. These approaches will be grouped into classes based on the amount of control they give a user over the behavior of the storage device. Figure 2.2 ranks these classes by the degree of freedom they offer in terms of allowable changes.

Adaptable

As described in Section 2.1, the storage controller of an SSD adheres to a set of policies chosen by the manufacturer. Adaptable SSDs allow for swapping out one policy for another, enabling users to custom-tailor the set of policies based on the

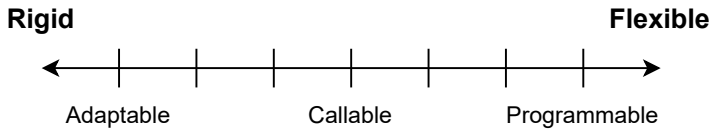


FIGURE 2.2: Classifications of computational storage units ranked by freedom of choice

type of workload [58]. Video streaming can typically tolerate a few incorrect pixel values, so changing the error correction code to a faster low-guarantee model could be a worthwhile trade-off. If most data for workloads are only used once, disabling the cache on the storage side and freeing up the RAM for the storage controller to store more meta-data can improve response times.

Plenty of research has gone into designing and evaluating different processing policies. Kang et al. proposed the usage of multi-stream SSDs, allowing the host to group data into separate streams depending on the expected lifetime of the data [50]. This minimized the amount of garbage collection required, resulting in increased I/O performance and SSD longevity. Yang et al. expanded upon this work by introducing AutoStream, a collection of stream assignment algorithms that analyze workloads at run-time and automatically assign them to a stream, freeing applications from manually managing the streams [81]. Li et al. designed an approximate cache strategy dubbed ASCache, which reduces cache misses at the expense of a higher error rate, intended for use in error-tolerant applications [61].

Other works focus on the construction of FTLs itself. The HIL framework by Choi et al. aims to support the construction of FTLs through compositional logs, promising fast crash recovery and enough generality to support multiple variants of FTLs [25]. Wu et al. proposed a component-based view of FTL design, supporting a modular approach to FTL construction and enabling developers to dynamically swap device policies [80].

Callable

Callable SSDs extend the interface exposed to the host with additional functionality beyond reading and writing. An example of this is the key-value SSD (KV-SSD), which allows the host to treat the SSD as a key-value store, mimicking the same key-value pairs used by a host application. KAML is one such KV-SSD [47], which supports the creation and deletion of namespaces by exposing `CreateNamespace` and `DeleteNamespace` operations in the host interface. These namespaces can be used to store or retrieve values by calling the operations `Put` or `Get`.

The GraphSSD is optimized for dealing with graph data and, like the KV-SSD, exposes a specialized interface to the host [64]. This interface allows for adding, updating, or deleting vertices and edges as part of a graph structure. Using this graph-aware storage method, Matam et al. showed a 29%-60% speedup of conventional graph algorithms.

Programmable

Programmable SSDs extend callable SSDs with the option to expand the host interface with user-defined functionality. An early example of this concept is the Willow SSD [74]. Seshadri et al. define modules for dealing with device IO and allow users to define their own programs in C, naming them SSD Apps. The SSD Apps, located on the Willow SSD, can be activated using remote procedure calls (RPC) between the host and the SSD, and program parameters and results are transferred between host and SSD using direct memory accesses (DMA).

Seshadri et al. noted that applications that rely on data-intensive computations would, in general, be able to exploit programmable SSDs, but that the Willow SSD was not designed with this goal in mind. To address this gap, Gu et al. created the Biscuit framework to define data-intensive queries that can run on storage systems [38]. Similarly, the Insider storage system provides in-storage computing using field-programmable gate arrays (FPGA) for increased performance during data-intensive workloads [72].

2.3 Code Offload

Particularly for programmable SSDs, a choice must be made for an execution environment of user-defined programs. As part of their standardization efforts, SNIA has identified four different environments to execute programs on computational storage devices: operating system images, container platforms, eBPF, and FPGA bit streams [1]. In their terminology, these programs are called *computational storage functions* and can either be installed by a manufacturer at the time of production, or downloaded onto the device by a user. Hedam et at. name these two types of functions *statically installed* and *downloaded*, respectively [40]. Statically installed functions can also be invoked from user-defined programs (downloaded functions), unlike the functions in callable SSDs, which can only be accessed on the host side through the host interface (recall Figure 2.1).

Examples of projects using the different environments are BlueDBM, DockerSSD, Delilah, and Newport. BlueDBM uses a reconfigurable FPGA to offload parts of data queries [49]. DockerSSD uses a minimal operating system capable of running Docker containers without the need to modify storage interfaces [37]. Delilah is an SSD equipped with a virtual machine capable of executing eBPF bytecode programs [40]. Newport is a family of computational storage devices with a full Linux stack installed, capable of running Linux binaries [42].

In SNIA's programming model, a programmer first goes through a discovery step, where they query a computational storage device for what functions are available to execute. These functions are located in a resource repository that is stored in either flash memory or RAM on the storage device. If functions other than those available are needed, the user defines such a function and downloads it to the device, adding it to the resource repository.

2.4 Program Verification

The goal of program verification is to provide a level of guarantee that a program matches the behavior intended by the programmer. At its most basic, software testing is an aspect of program verification. Programmers and/or testers write tests to supply the program with fixed input and compare the results with the expected output. Once the tests are deemed comprehensive enough and no more bugs are found, the program is considered safe to use. However, determining when tests have covered enough cases is not a simple task.

To navigate this challenge, different strategies for writing tests are used, with one of the newer approaches being property-based testing [36]. In this testing strategy, testers write testable properties that the output of a program must fulfill, given certain conditions on the input. As an example, a property for an addition operator might be: "for all integers X, X plus 0 equals X", which is then encoded in an executable format. Testing the property consists of randomly generating a set of input data and seeing if the property holds for all the output data. The properties of a program make up what is called the formal specification of the program.

Where program verification differs from testing is that it does not concern itself with concrete input / output, but rather proving the formal specification of a program logically. This approach translates the properties in the formal specification into proof obligations (also called contracts), which must be fulfilled in a machine-verifiable manner. The process of fulfilling the obligations depends on the verification system being used, but generally consists of generating a proof by stepping through the structure of the program. This stepwise proof creation can be attempted either automatically using provers such as Alt-Ergo and Z3 [2, 3], or manually using interactive theorem provers such as Roqc and Isabelle [16, 4]. Some projects build on top of these provers by supplying a language for both writing programs as well as specifications, complete with a way to convert the specifications to proof obligations which are discharged to a prover for verification. Examples include Frama-C, which uses ACSL as a specification language for C programs [26], Spark, which uses a modified version of Ada for both programming and writing specifications [5], and the Why3 platform, which uses its own WhyML language [20].

The existing research on computational storage deals with either performance or usability, but very little touches on the aspect of verification. The HIL framework was designed to provide provably correct crash recovery, and Choi et al. provide a proof sketch for the correctness of the recovery mechanism [25]. However, such proofs only provide guarantees about the singular component under consideration, and as stated by Barbalace & Do, there is an open issue regarding the trust of computational storage programs [10].

The following subsections will look at how program verification can be used to address the research gap on trusted computational storage programs.

Existing Verifiers

When it comes to defining the specification for a program, in the typical case, whoever is writing the program is free to choose which properties are of interest for the formal specification. There are, however, also cases where the properties are requirements imposed by another party. The person responsible for implementing the software must then prove that these requirements are met before the other party is willing to accept and execute the software. Often, this process is carried out automatically by a helper program called a *verifier*, that attempts to check if an input program satisfies a predefined set of criteria. This is done, for example, in the Linux kernel [71], where a verifier tests kernel extensions written in eBPF, before the extensions are allowed to execute. These verifiers perform a static analysis of the programs rather than generating and proving verification conditions. The analysis itself can be formally proven to only allow programs that satisfy the relevant criteria, such as is the case with the PREVAIL verifier [35].

This process of verification by analysis differs from the formal verification of user-defined specifications, as the user has a lot less flexibility in guiding the analysis compared to constructing a proof on their own. To differentiate the two processes, the rest of this thesis will use the term *validation* to refer to the automatic check of programs against a fixed specification, and *validators* to mean the tools that carry out the validation.

Section 2.3 mentioned that it is possible to express and execute downloadable computations in eBPF, and validators already exist for eBPF programs. This makes it an option to use such a verifier to validate downloadable functions.

Verified Environments

As described in Section 2.3, different environments for executing code on a CSD exist. Outside the field of computational storage, some of these environments have variants that have been verified. The seL4 microkernel is an operating system kernel that has been formally verified to follow a specification defined by Klein et al. [51] They show that their kernel does not crash or perform operations outside of those described in the specification. The seL4 microkernel is thus an option for a verified environment on CSDs that rely on operating systems for program execution. For CSDs that use containers as their execution environment, the NOVA virtualization architecture implements a virtualization layer for hosting guest operating systems [76]. The layer is designed with an emphasis on minimizing the trusted coding base and decomposing the layer into small components that are independently verifiable.

FPGAs can have their design verified using hardware description languages, such as SystemVerilog or VHDL, to model their behavior [44, 45]. The model is then simulated to verify a specification match, which again differs from the proof construction and static analysis approaches discussed earlier in this section. eBPF is either compiled to native machine code or interpreted, as done in the uBPF VM [54]. As of writing, no verified compiler or virtual machine exists for eBPF code, and environments using eBPF rely only on validators to analyze the contents of eBPF programs.

Proof-carrying Code

Proof-carrying code is an idea formulated by Necula in the late 90s, with the goal of being able to prove that code sent by untrusted sources adheres to some specified policy or contract [67]. Code sent by an untrusted source would be embedded with a proof that the code adheres to the contract. This proof would be checked by a validator installed at the destination, before allowing the code to be executed. By having a validator on the destination device, the device would then only have to trust that its own mechanism correctly validates the proof.

In the context of computational storage, there are cases where the host and the storage device belong to separate parties. In those cases, proof-carrying code sounds like an excellent setup for the host owner to send computations and proofs to the provider of the computational storage device. However, there are reasons why this approach is still not seeing widespread use today. For one, checking whether a proof is valid is simple, but constructing a proof is highly non-trivial. As shown by Turing and his famous halting problem, no general algorithm exists which can determine the behavior of an arbitrary program. This means that the responsibility of proof construction falls to a human. Most often, this would be the creator of the program, but unfortunately not every programmer enjoys formal logics. One of the many challenges of computational storage is usability [10], and requiring every programmer who intends to use computational storage to also take a course in semantics and logics runs counter to the goal of usability.

2.5 Design Decisions

Section 2.4 has shown that there are different options for creating a computational storage architecture built on verified components. However, no current research touches on verification of the programs offloaded to CSDs. There are many tools for verification of general programs [18, 20, 5, 16, 4], but as covered in Section 1.1, programs for computational storage do not follow general programming patterns.

The design goal for the following chapters is to bring program verification techniques to a new class of computational storage devices that are located between callable and programmable devices on the flexibility spectrum (Figure 2.2). While it is theoretically possible to support fully programmable devices with verification through proof-carrying code, it would place greater burden on the user, who already has to analyze whether their application even benefits from offloading to computational storage. Alternatively, a validator could be installed on programmable devices to analyze offloaded code, but existing validators conservatively limit what programs the user is allowed to offload [35].

Rather, the goal in this thesis is to move the proof obligations from the user to the storage manufacturer or provider. The concept of fixed functions utilized in some of the projects, such as the GraphSSD and KV-SSD [64, 47], provides an excellent target for programs that need only be verified once. A case study of the verification process will be covered in Chapter 4. To still allow some flexibility to users, Chapter 3 defines a DSL which supports the composition of statically installed functions and low-level

data management. The proposed name for devices that would utilize such a DSL is *compositional* devices.

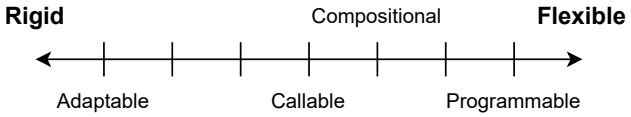


FIGURE 2.3: Flexibility spectrum with compositional devices placed between callable and programmable

As Figure 2.3 shows, this class of devices is placed slightly to the right of callable devices on the flexibility spectrum, as they allow users to build their own functions by composing existing functions, without allowing the implementation of functionality that does not already exist on the device.

In compositional devices, statically installed functions handle computations, while user-defined functions handle only data placements and function composition. Splitting up the computational and data management parts of computational storage provides a clearer separation of domains. This simplifies verification, but it also allows storage vendors to advertise the selection of functions installed on their devices. To further strengthen the trust of a device, formally verified functions act as a certificate for a potential user. Imagine a scenario in which a storage vendor is selling CSDs to a cloud provider who intends to offer a data querying service. The service provider would likely want functionality specific to their service, in which case it would be highly beneficial to have a verified foundation to build any extensions on top of. In addition, they would be able to extend the specification in order to provide guarantees to consumers that might use their service. Figure 2.4 illustrates this relationship between the storage vendor, the service provider, and the user.

The remaining part of this section will cover the assumptions of the computational storage architecture. Recall that Figure 2.1 shows an optional RAM component as part of the SSD. In this thesis, RAM will be assumed to always exist on the storage device and will make up the base of the memory model for computational storage functions (CSFs). Additionally, we assume a theory of operation similar to the Eid-Hermes project by Eideticom [13]. In this project, some of the device memory is configured to act as program slots and data slots. The program slots contain the instructions of CSFs to be executed, so the host must load a CSF from a resource repository into a program slot. The data slots store both input parameters, intermediate results, and output data from a program. The host supplies parameters and retrieves results through direct memory access to the relevant part of the data slot. To execute, a program slot containing a CSF and a data slot must be associated to a processor core, which will then perform the instructions as specified by the program slot, while having exclusive write access to the associated data slot.

In his Ph.D. thesis, Hedam expands this memory model to also include a portion of shared memory, allowing programs to communicate shared results [39]. In this expanded model, programs are able to access shared memory through raw pointers,

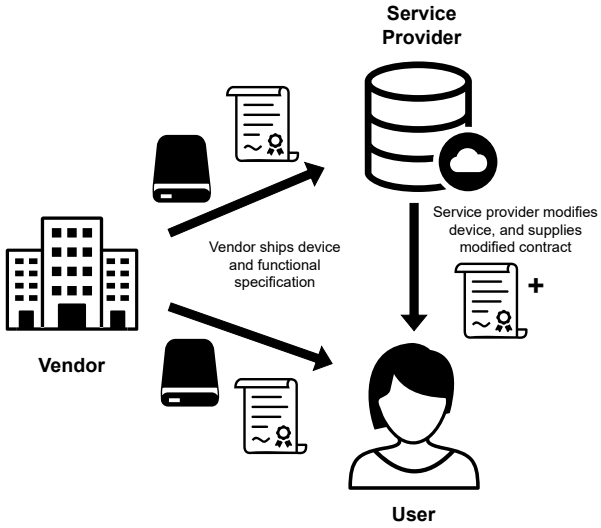


FIGURE 2.4: A storage vendor produces devices with a formal guarantee of behavior. The behavior and guarantee can be extended by a third party.

but this thesis proposes and implements a resource manager to coordinate access to shared memory, in order to guarantee defined behavior. This resource manager is presented in Chapter 5. An overview of the final memory model is illustrated in Figure 2.5.

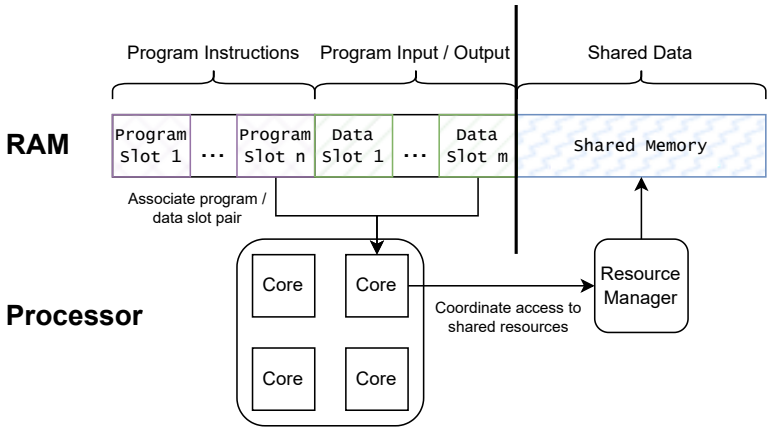


FIGURE 2.5: Separation of RAM into program slots, local data slots and shared memory.

2.6 Setup & Terminology

As this is quite a new area of research, it is worth spending some time defining a clear terminology. The use or naming of some terms may differ from other research on computational storage. This section will clarify the meaning of certain terms. The terms cover both objects and actions and are all covered in Figure 2.6. The following workflow description introduces the terms using natural examples. A bolded term in the workflow description has a corresponding entry in the glossary at the end of the section. The glossary is sorted according to the appearance of the terms in the workflow description.

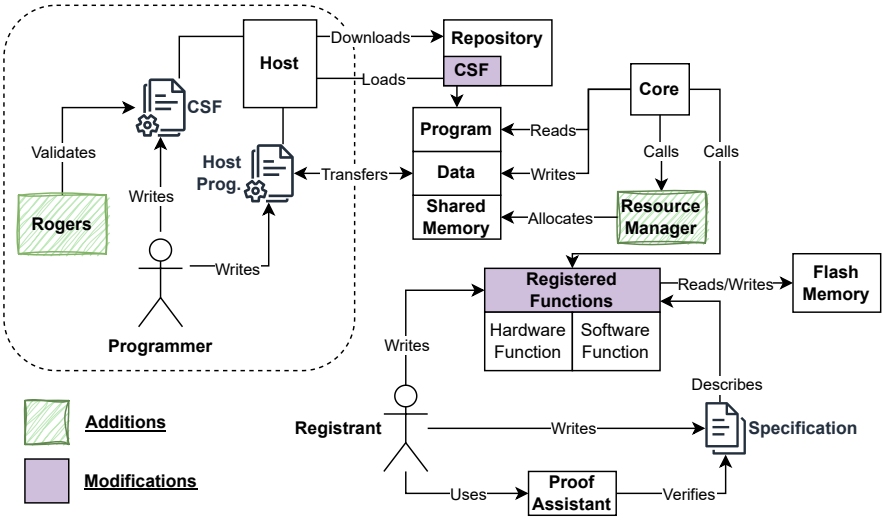


FIGURE 2.6: Components and actors interacting with parts of computational storage

Workflow Description

A **programmer** queries a **computational storage device** (CSD) for which **registered functions** are on the device. The query returns a list of function **specifications**. The registered functions have been preinstalled on the CSD by a **registrant**. This registrant has both implemented the registered functions and written their specification. The specifications have been **verified** using a **proof assistant**, and both the specification and the proof are included on the device along with the function. After querying the device, the programmer uses the **Rogers** DSL to write a **computational storage function** (CSF) which can call the registered functions on the CSD. After **validating** the CSF with Rogers, the programmer **downloads** the CSF onto the CSD, which transfers the CSF to a **resource repository** on the device. Later, the programmer can write a **host program**, which can load any CSF, found in the resource repository on the CSD, into a **program slot** on the device. The host program can configure the CSF by transferring parameters to a **data slot**, and then associate the program and data slot pair to the same **core**. The core requests regions of **shared memory** from a **resource manager** and invokes registered functions as

specified by the CSF. The host program can then transfer data produced by the CSF back to the host.

Glossary

- **Programmer**: The actor who writes CSFs to execute on a CSD. Contrasted with *registrant*.
- **Computational Storage Device** (CSD): A solid-state drive equipped with memory and a processor.
- **Registered Function**: Statically installed functions that have been verified. These functions carry out the main part of computations and are invoked by CSFs. While they are called functions, they can be implemented in any programming language or even hardware.
- **Specification**: The post- and pre-conditions for a function, written in first-order logic.
- **Registrant**: The actor responsible for implementing and verifying registered functions. In most cases, this will be either a storage vendor or a service provider, but it can also be the programmer themselves.
- **Verification**: The act of proving the behavior of some program follows a certain specification. In this thesis, the programs subject to verification are solely registered functions. Others may use the term 'verification' to refer to the act of checking that a program follows a set of pre-defined rules, which this thesis calls *validation*.
- **Proof Assistant**: Software that automatically or interactively proves a set of proof obligations.
- **Rogers**: The name of the tool which must validate CSFs before they are downloaded to a CSD.
- **Computational Storage Function** (CSF): Program indicating data placements and registered function invocations on a CSD. Others may call this a kernel or a program. This thesis differentiates CSFs from programs by referring to the software running on the host as a *host program*, while the parts of a program that runs on the CSD are CSFs. CSF as a term in this thesis also only ever covers the code that is dynamically offloaded to the CSD, and not the registered functions.
- **Validation**: The analysis of a program (CSF in this case) and checking that it does not perform any illegal action and performs all required actions.
- **Downloading**: The act of transferring a CSF to the resource repository of a CSD.
- **Resource Repository**: A collection of callable CSFs located on a CSD.

- **Host Program**: Application running on the host side. Host programs are responsible for loading CSFs, configuring input, and retrieving results after execution.
- **Program Slot**: Section of RAM on the storage device dedicated to storing program instructions of CSFs.
- **Data Slot**: Section of RAM on the storage device dedicated to storing program input, intermediate results, and program output.
- **Core**: A single threaded process on a CSD, capable of executing a CSF. Multiple cores are often present on a CSD, allowing some degree of concurrency.
- **Shared Memory**: Section of RAM on the storage device dedicated to sharing state between CSFs. Memory regions in this section can only be allocated by the resource manager.
- **Resource Manager**: The central component responsible for allocating shared memory to cores and arbitrating concurrency conflicts.

Chapter 3

The Rogers DSL

This chapter introduces a DSL that we use to write CSFs for compositional devices (recall Figure 2.3). The DSL aims to support programmers of CSFs in expressing data placements and orchestrating function calls while allowing the validator (Rogers) to easily analyze CSFs for guarantees of memory safety. The guarantees will be defined more concretely in Section 3.1. We have chosen to base the DSL on a subset of C. C has the benefit of being low-level enough to feature manual memory management while still being high-level enough to allow type checking of the programs. Additionally, both the LLVM and GCC compilers support compiling C code to eBPF, meaning CSFs written in the Rogers DSL can be used with three of the four proposed execution environments covered in Section 2.3, i.e. eBPF, OS images, and container platforms. FPGAs are programmed using bit streams derived from a hardware description language and are thus not supported by the Rogers DSL. Lastly, as a non-functional benefit, basing the DSL on a language that is familiar to most system programmers lowers the barrier of entry.

3.1 Memory Management

Figure 2.5 covers the different types of memory introduced as part of the Eid-Hermes protocol [13] and later modified by Hedam et al. [39]. To clarify what requirements memory management entails, let us review the characteristics of the three memory types.

Program slots are the simplest to manage. The host application can request a CSF to be loaded into a program slot and a core will read the instructions stored in the slot, but no modifications are made to the program slot during execution. The main requirement for program slots is that CSFs may not obtain pointers into program slots so that they do not alter their own behavior during execution.

Data slots contain the local memory of each CSF. When a CSF is launched on the CSD, its only argument is a pointer to a data slot. Access to that data slot is assumed to be exclusive, so a CSF can read/write from/to its own data slot without interference but cannot obtain access to other data slots. Any arguments to a CSF must be placed in its associated data slot. Since the CSD uses an address space separate from the host application, the position of the arguments cannot be communicated via pointers. Instead, the position of all parameters need to be described using relative offsets into

the data slot. The lifetime of data in data slots is equivalent to the lifetime of the associated CSF. Once the CSF has finished execution and the host application has fetched the relevant results, the data slot can be reused for another CSF.

Shared memory is a section of device memory dedicated to storing data communicated between CSFs. In the original memory model used by Delilah [39], a pointer to shared memory was passed to all CSFs as an extra argument. As this is a raw pointer, all CSFs can access any part of shared memory through pointer arithmetic on this shared pointer. It is the responsibility of the programmer to coordinate accesses to shared memory between CSFs in such a way that the CSFs would not overwrite each others work or read outside of bounds.

From this analysis of the different memory types, we arrive at three notable aspects:

- For a CSF to be well behaved, it must stay within the bounds of its own data slot. That is, any data it produces must either be written locally or placed in shared memory.
- Data passed from and to the host are placed relative to the start of the data slots. As data is written in the data slot, the CSF must keep track of which offsets store the results relevant for the host program, and communicate the offsets back to the host, so the host program knows which section of the data slot to read the results from.
- The burden of managing concurrency is currently placed on the programmer. They must mentally keep track of which parts of shared memory contain intermediate results as they perform code offload in the host program.

Additionally, Hedam notes that the use of static-size data slots negatively impacts the performance of the CSD. In their evaluation, they fixed the size of the data slots to 1GB for ease of integration. However, they concluded from the performance benchmarks that the resulting fragmentation of device memory stemming from fixed-size data slots is a problem that needs to be solved using dynamically sized data slots instead.

Using these aspects as a basis, the main goal of the DSL is to enable safe and simple memory management for CSF programmers. We achieve this by using region-based memory management and create a type system to help keep track of the different regions used in a CSF. The existing usage of offsets already supports a region-based approach, as the offsets can easily be converted to describe lengths of memory regions. The DSL can thus promote these offsets to first-class citizens and type them accordingly. We add a component for managing shared memory (described further in Chapter 5), creating a layer of indirection between CSFs and shared memory. The manager assigns IDs to shared regions, and CSFs can request access to a region through the management component using the relevant ID. As the manager is responsible for translating IDs to shared regions, the programmer no longer needs to worry about overwriting parts of shared memory used by other CSFs after access has been granted. With the help of verification tools discussed further in Chapter 4, the DSL is able to estimate the size of the regions that a CSF will need. This estimation can be used by the host to dynamically allocate data slots of fitting sizes.

3.2 Syntax

Listing 3.1 shows the template that all CSFs expressed in the Rogers DSL must follow. The terms on lines 5 and 13 indicate the parts of the program written by the programmer, which must follow the grammar defined in Figure 3.2. A term of the form $\overrightarrow{Term}$ indicates 0 or more repetitions of valid terms. Lines 1-3 contain fixed definitions necessary for dealing with memory regions. These definitions consist of `mem_range`, representing the aforementioned memory regions, `permission` to indicate allowable actions on shared data, as well as a request-`struct` for accessing shared data through IDs.

Listing 3.1: The frame outlining the shape of every Rogers program.

```
1  struct mem_range { char* ptr; int length; };
2  enum permission { Read, Write};
3  struct request { int id; enum permission p; };
4  struct layout {
5      Field⃗;
6      char dynamic[];
7  };
8  int prog(void *ctx) {
9      struct layout *info = (struct layout *)ctx;
10     struct request reqs[Reqs(layout)];
11     // automatic initialization of requests (sec. 3.4.1)
12     struct mem_range *resources = obtain(reqs, Reqs(layout));
13     Statement⃗;
14     return 0;
15 }
```

As CSFs are only passed a single parameter in the form of a pointer to its data slot, the DSL requires CSFs to define a layout-`struct` on lines 4-7 to describe the type of input data that is expected to be placed in the data slot at the start of execution. Some CSFs may take inputs of variable lengths, e.g. strings. If we wish to write a CSF that takes a file name as a parameter, we must take into account the fact that file names can have different lengths. To address this fact, we require all layout structs to declare a flexible array member, which we call `dynamic`. Input data of variable length can then be placed in this `dynamic` array and can be accessed through offset fields defined in the layout. If some data B are placed after some other data A in `dynamic`, the declaration of the offset field containing the length of A must precede the declaration of the offset field of B. The total length of the array will then be the sum of all offsets. This is visualized in Figure 3.1.

As indicated by the grammar regarding *Field* in Figure 3.2, we differentiate different kinds of fields on the layout `struct` based on the name of the field. We use n to range over regular names (any legal variable name in C). Names of offset fields must end in `_offset`, and names of shared region IDs must end in `_req` (short for *request*). Any field that does not end in `_offset` or `_req` is considered a standard value.

The main body of a CSF is described in lines 8-15 of Listing 3.1. The first line of any CSF must be a cast from the supplied data slot pointer (`ctx`) to a layout pointer, in order to interpret the data located in the slot as specified in the layout.

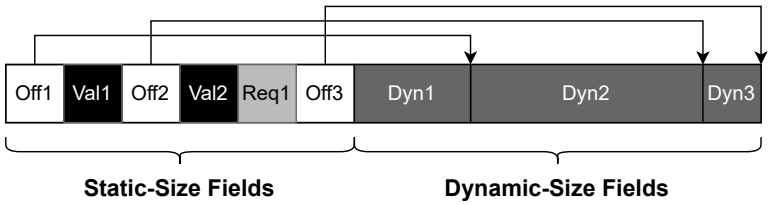


FIGURE 3.1: Layout of input structure showcasing the different types of fields: offsets, values, requests and dynamically-sized data

For a CSF to access shared data, the host has to set one of the request fields in the layout-`struct` to the ID of the desired memory region. IDs are created by the resource manager after a call to `allocate_shared`, one of two registered functions dealing with shared memory, which we assume to always be present on the CSD. The other function is `obtain`, used in line 12, responsible for acquiring access to the shared regions referenced by the IDs on the layout. *Reqs*(`layout`) is the number of request fields defined for a given layout, which is used on lines 10 and 12. Requests have to be combined with a permission level (read or write) before requests can be passed to `obtain`. To free the programmer from having to manually set permissions, we will use the typing done by Rogers to automatically infer the necessary permissions and insert the construction of requests into the code. Section 3.4.1 covers how this is done in more detail. Once requests have been passed to `obtain`, the CSF stalls execution until the call returns with the location and size of the requested regions. The programmer can then access the shared regions by indexing into the `resources` array, with valid indices between 0 and *Reqs*(`layout`) − 1. The regions are ordered in the same manner as the dynamic data, i.e. the region whose ID is defined first in layout will be placed at index 0 and so on.

Statements in the language largely consist of either registered function calls, value declarations, and constructing memory regions. Memory regions are constructed using offset expressions which represent either the address or the length of a memory region. Initially, the only valid offset expressions are the offset fields of the layout `struct`. Lengths and addresses can be summed together to obtain new addresses, and new lengths are added to the program when registered functions write to a region.

Registered functions do not return a value per se, but rather write their output to a provided memory region. For this reason, we do not classify registered function calls as expressions. An exception to this is the registered function `allocate_shared`, which creates a new shared memory region and returns the corresponding ID, which has to be stored in a field on the layout `struct`.

This grammar may look unusually restrictive. One may wonder how to write a meaningful program without conditionals or loops. As discussed in Section 2.6, a CSD will provide a collection of registered functions for use in computations. The compositional device proposal consists of separating *all* computations into registered functions and only using the CSF as a way to organize the data and invoke the relevant functions. Branching control flow is therefore unnecessary for CSFs. Restricting CSFs

Field ::=	`int` n`_offset`	Declaration of region length field
	| `int` n`_req`	Declaration of request ID field
	| *PrimType* n	Declaration of regular C field
PrimType ::=	`char` | `int` | `float`	Standard C types
Statement ::=	*FCall*	Function call
	| *PrimType* n	Variable declaration
	| *PrimType* n = *Expr*	Variable declaration with initialization
	| `struct mem_range` n	Local memory region declaration
	| `info` $\rightarrow n$ = *Expr*	Assignment to the layout struct
	| n`.ptr` = *OffsetExpr*	Assignment of address of memory region
	| n`.length` = *OffsetExpr*	Assignment of length of memory region
FCall ::=	$n(\overrightarrow{Expr})$	Call to verified function
	| `info` $\rightarrow n$ = `allocate_shared`$(Expr, n)$	Specialized allocation call
Expr ::=	n	Variable
	| `&`n	Reference to variable
	| *Const*	Constant value
	| *Expr BinOp Expr*	Binary operation
	| *OffsetExpr*	Offset Expression
	| `info` $\rightarrow n$	Field access of input
	| `resources`$[i]$	Access shared resource
	| `&resources`$[i]$	Reference to shared resource
BinOp ::=	`+` | `-` | `*` | `/`	Arithmetic operations
OffsetExpr ::=	n`.length`	Memory region length
	| n`.ptr`	Memory region address
	| `info` $\rightarrow n$`_offset`	Layout access
	| `info` $\rightarrow$ `dynamic`	Flexible array access
	| *OffsetExpr* `+` *OffsetExpr*	Pointer arithmetic
Const ::=	c | i | f	Numeric literals (char, int, float)

FIGURE 3.2: Grammar of the Rogers DSL.

to a simple sequential flow also guarantees termination, a concern stated in prior research [10]. Termination requires all calls to the resource manager and registered functions return, so we will show that to be the case in the Chapters 4 and 5.

To both get a feel for the DSL and showcase how CSFs may interact with each other, let us consider an example. Assume that we have a CSD with two files of interest: Events.txt and Dates.txt. Each line in Events.txt contains the name of an event, and each line in Dates.txt contains the date where an event occurred. The contents of the two files are implicitly linked by position, meaning that the first event in Events.txt happened on the first date in Dates.txt and so on. Assume further that we have two goals: finding all events that happened in 1997 and also separately finding all events that happened after 1998.

We could write a single CSF that takes a filtering predicate as argument and answers both queries by executing that CSF twice (with different arguments). However, it can be more performant to split the execution over two CSFs: One that reads the files (which are used in both queries) from storage to shared memory and another that reads from shared memory and performs a filter-based selection.

We show how to write such CSFs, which we name 'Program 0' and 'Program 1' in Listings 3.2 and 3.3. Program 0 takes the names of two files as well as the size of the files. Since filenames have variable length, they will be stored in the `dynamic` array, and thus the CSF also needs the length of the names as input. When the CSF is executed, it will allocate two regions in shared memory, in which it will store the contents of the two read files. The IDs of these regions are written in the layout `struct`, and the host can then transfer these IDs as parameters to Program 1.

Listing 3.2: Program placing two files to shared memory

```
// -- Program 0 --
// [Fixed definitions omitted]
struct layout {
    int event_fname_offset;
    int event_fsize;
    int event_id;
    int date_fname_offset;
    int date_fsize;
    int date_id;
    char dynamic[];
};
int main (void *ctx) {
    struct layout *info = (struct layout *)ctx;
    struct request reqs[0];
    struct mem_range *resources = obtain (reqs, 0);
    // Construct local region containing name of event file
    struct mem_range event_fname;
    event_fname.ptr = info->dynamic;
    event_fname.length = info->event_fname_offset;
    // Allocate a shared region to store contents of event file
    struct mem_range event_placement;
    info->event_id = allocate_shared(info->event_fsize, &event_placement);
    read_file(event_fname, &event_placement);
    // Construct local region containing name of date file
    struct mem_range date_fname;
    date_fname.ptr = info->dynamic + info->event_fname_offset;
    date_fname.length = info->date_fname_offset;
    // Allocate a shared region to store contents of date file
    struct mem_range date_placement;
    info->date_id = allocate_shared(info->date_fsize, &date_placement);
    read_file(date_fname, &date_placement);

    return 0;
}
```

Listing 3.3: Program doing filter based selection

```
// -- Program 1 --
// [Fixed definitions omitted]
struct layout {
    int event_file_req;
    int date_file_req;
    int comp_op;
    int date;
    int res_start;
    int res_length;
    char dynamic[];
};
int main (void *context) {
    struct layout *info = (struct layout *)context;
    struct request reqs[2];
    // Automatic fill-in of requests
    struct mem_range *resources = obtain(reqs, 2);
    // Perform filter and store results in local region 0
    struct mem_range filter_res;
    filter_res.ptr = info->dynamic;
    filter(resources[1], &filter_res, info->comp_op, info->date);
    // Perform selection and store results in local region 1
    struct mem_range select_res;
    select_res.ptr = info->dynamic + filter_res.length;
    select(resources[0], filter_res, &select_res);
    // Inform host of location of local region 1
    info->res_start = filter_res.length;
    info->res_length = select_res.length;
    return 0;
}
```

Program 1 takes two IDs, a comparison operator (less than, equal to, greater than) and a date to compare against. It will request access to the two shared regions indicated by the IDs and then perform a filter and a selection in local memory. As the selection is the final result of the CSF, it communicates the start and end offset of the local region with the results back to the host. Looking through the body of Program 1, notice that the length of the memory regions `filter_res` and `select_res` is not set by the program. This is because the called function has to update the `struct` with how much data it has written.

3.3 Type System

The main goal of any type system is to reduce programming errors by restricting operations to those that are *well-typed*, with the definition of well-typed depending on the system. Based on the review of memory in Section 3.1, we want well-typed CSFs to respect the bounds of data slots by interacting with memory through regions. A well-typed CSF should not overlap these regions or access memory outside a region. To be space-efficient, we also want local regions to be packed tightly together to avoid fragmenting the data slot.

3.3.1 Dependent Types

By deriving types on expressions, a type system can enforce certain restrictions on valid programs, such as disallowing division of pointers or addition of strings with integers. The simplest type systems work by treating types as labels for sets, with elements within the set considered to have that type. As an example, `uint32` could be considered the label for the set $\{x \mid 0 \leq x \leq 2^{32} - 1\}$ and `bool` the label for the set $\{0, 1\}$ or $\{\texttt{false}, \texttt{true}\}$. Most modern programming languages go a step above these simpler types, by allowing *polymorphic* types, that is, types that depend on other types. A commonly encountered polymorphic type is `List<Type>`, a type constructor that represents generic lists of some type of elements. The constructor can be seen as a function that takes a type as input and produces a type as output, i.e. `List` : `Type` $\rightarrow$ `Type`. *Dependent* types, like polymorphic types, act as functions, but rather than relying on an input type, they depend on a term (or value or expression). Consider a vector, a list of elements with a specified length. Some vector operations, such as the cross product, require that vectors have a specific length. Dependent types allow the length to be expressed as part of the type, by creating a type constructor `Vector` : `integer` $\rightarrow$ `Type`. This could allow a type system to, e.g. invalidate a program that tries to calculate the cross product of two-dimensional vectors with the type `Vector<2>`.

The type system of the Rogers DSL uses dependent types in two places for the purpose of managing memory. The first instance is the construction of local memory regions. Recall the discussion surrounding Figure 3.1. Input to CSFs may contain dynamically sized data, which have to be reconstructed based on the offset values provided by the layout struct. To ensure the reconstruction is done correctly, we will use two dependent types: `address` : `integer` $\rightarrow$ `Type` and `length` : `integer` $\rightarrow$ `Type`. By indexing the local memory regions based on their relative position in the data slot, Rogers can type the pointers and offsets of the regions using the index of the region. The first region, which has index 0, then needs a pointer of type `address 0` and an offset of type `length 0` to be a validly constructed region.

The second instance of dependent types comes from function calls. As some functions will write output to memory regions, we must ensure that the memory region is large enough to contain the output. Not all functions produce outputs of a size that is statically determinable, so we use a dependent type `write` : `Cexpr` $\rightarrow$ `Type` for memory regions to which a function intends to write. The type depends on a C-expression that defines an upper bound on the amount of output that the function will write to the specific region. The expression must be typeable as an integer so that we can then compare the bound with the concrete length of a region. It is not trivial to infer the bound, so we will discuss in greater detail how to derive this C-expression in section 4.3.5.

The expressions used to bound the amount of data written will often refer to the input of the function. We express these references by the position of the function arguments. This means that an expression of the form $\#i$ refers to the ith argument of the function. As an example, in Listing 3.3 we showed an example of a CSF calling a filter function, which took an input region as its first argument and an output region

as its second. Since the output region needs to be at least as big as the read region (as it could be that no element gets filtered out), the type of the second argument is `write #0.length`, meaning that the length of the write region must be at least the length of the first argument (which is the read region).

With dependent types covered, we give the full collection of types used by Rogers in Figure 3.3.

RogersType ::=	`char` | `int` | `float`	Standard C types
	| *RogersType*`*`	Pointer to a Rogers type
	| `id`	IDs of shared regions
	| `address` $i, i \in \mathbb{N}$	Region pointers
	| `length` $i, i \in \mathbb{N}$	Region lengths
	| `local` | `shared`	Region locations
	| `read` | `write` $e, e \in$ *CExpr*	Access permissions
CExpr ::=	[Any integer-typable expression allowed in C99]	

FIGURE 3.3: Types of the Rogers DSL.

3.3.2 Typing Rules

The main purpose of the typing rules is to keep track of memory regions, their position and size. Memory regions in the shared space have to be created through the resource manager, while regions in the local space must be placed in a non-overlapping manner. For the purpose of typing expressions, we use a typing context σ : `Name` $\times$ `Kind` $\rightarrow$ `RogersType` to store the types of variables as they appear throughout the program. We tag variables with a `Kind` to make it easier to distinguish different kinds of variables during typing and proving. A `Kind` is one of either `Prim`, `Info`, `Resource`, `Pointer`, or `Size`. We start by showing how to type offset expressions in Figure 3.4.

As the `dynamic` array is expected to always be the starting point for the placement of local regions, rule E-Dyn states that `dynamic` is always typable as `address` 0. E-Ptr and E-Length state that typing the address and length of a memory region corresponds to looking up the region in the typing context with kind `Pointer` or `Size`, respectively. E-Info states that typing an offset from the layout `struct` (named `info`) is typed the same ways as other lengths, but using kind `Info`. Lastly, E-Addr states that adding two offset expressions can be typed as `address` $(i + 1)$, only if the two expressions can be typed as the address and length belonging to the region located at `address` i. This rule of adding matching lengths and addresses to obtain the next address up is key to ensuring correct placement of local regions.

Now that the typing of offset expressions have been covered, we take a look at the typing rules for standard expressions in Figure 3.5.

The rules E-Byte, E-Int and E-Float simply state that C literals can always be typed to its corresponding C-type.

$\boxed{\sigma \vDash \mathit{OffsetExpr} : \tau}$

$$\frac{}{\sigma \vDash \texttt{info} \rightarrow \texttt{dynamic} : \texttt{address}\ 0}\ \text{(E-Dyn)}$$

$$\frac{\sigma(n, \texttt{Pointer}) = \texttt{address}\ i}{\sigma \vDash n.\texttt{ptr} : \texttt{address}\ i}\ \text{(E-Ptr)} \qquad \frac{\sigma(n, \texttt{Size}) = \texttt{length}\ i}{\sigma \vDash n.\texttt{length} : \texttt{length}\ i}\ \text{(E-Length)}$$

$$\frac{\sigma(n\texttt{_offset}, \texttt{Info}) = \texttt{length}\ i}{\sigma \vDash \texttt{info} \rightarrow n\texttt{_offset} : \texttt{length}\ i}\ \text{(E-Info)}$$

$$\frac{\sigma \vDash e_1 : \texttt{address}\ i \qquad \sigma \vDash e_2 : \texttt{length}\ i}{\sigma \vDash e_1 + e_2 : \texttt{address}\ (i+1)}\ \text{(E-Addr)}$$

FIGURE 3.4: Typing of offset expressions.

$\boxed{\sigma \vdash \mathit{Expr} : \tau}$

$$\frac{}{\sigma \vdash c : \texttt{char}}\ \text{(E-Byte)} \qquad \frac{}{\sigma \vdash i : \texttt{int}}\ \text{(E-Int)} \qquad \frac{}{\sigma \vdash f : \texttt{float}}\ \text{(E-Float)}$$

$$\frac{\sigma(n, \texttt{Prim}) = \tau}{\sigma \vdash n : \tau}\ \text{(E-Var)} \qquad \frac{\sigma(\mathit{field}, \texttt{Info}) = \tau}{\sigma \vdash \texttt{info} \rightarrow \mathit{field} : \tau}\ \text{(E-Field)}$$

$$\frac{\sigma \vdash e_1 : \tau \qquad \sigma \vdash e_2 : \tau \qquad \tau \in \{\texttt{byte}, \texttt{int}, \texttt{float}\}}{\sigma \vdash e_1\ \mathit{BinOp}\ e_2 : \tau}\ \text{(E-Binop)}$$

$$\frac{\sigma \vdash e : \tau}{\sigma \vdash \&e : *\tau}\ \text{(E-Ref)} \qquad \frac{\sigma \vDash oe : \tau}{\sigma \vdash oe : \tau}\ \text{(E-Off)} \qquad \frac{\sigma \vDash oe : \texttt{length}\ i}{\sigma \vdash oe : \texttt{int}}\ \text{(E-LSub)}$$

$$\frac{\sigma(res_i, \texttt{Resource}) = \texttt{shared}}{\sigma \vdash \texttt{resources}[i] : \texttt{shared}}\ \text{(E-ShrA)} \qquad \frac{\sigma \vdash e : \texttt{shared}}{\sigma \vdash e : \texttt{read}}\ \text{(E-ShrR)}$$

$$\frac{\sigma \vdash e : \texttt{shared} \qquad \mathit{bound} \leq e.\texttt{length}}{\sigma \vdash e : \texttt{write}\ \mathit{bound}}\ \text{(E-ShrW)}$$

$$\frac{\sigma \vdash n : \texttt{local} \qquad \sigma \vdash n.\texttt{ptr} : \texttt{address}\ i \qquad \sigma \vdash n.\texttt{length} : \texttt{length}\ i}{\sigma \vdash n : \texttt{read}}\ \text{(E-LocR)}$$

$$\frac{\sigma \vdash n : \texttt{local} \qquad \sigma \vdash n.\texttt{ptr} : \texttt{address}\ (\texttt{max_i}(\sigma) + 1)}{\sigma \vdash n : \texttt{write}\ \mathit{bound}}\ \text{(E-LocW)}$$

FIGURE 3.5: Typing of expressions

E-Var states that typing of variables is done by looking up their type in the typing context as a primitive. E-Field shows field accesses on the layout `struct` (named `info`) are typed the same way, but using the `Info Kind`. E-Binop states that binary operations just require the two operands to be of matching numeral types. E-Ref states that references to expressions can be typed as a pointer of the type of the expression. E-Off states that when a regular expression is an offset expression, we must use the typing rules for offset expressions. To allow the length of regions to be used in calculations or assigned to `info` E-LSub allows any length to be typed as a regular C integer.

The remaining rules regarding expressions deal with typing memory regions. E-ShrA states that all indices of `resources` can be typed as shared memory, as long as a binding for the index exists in the typing context. Since we will show how to analyze the necessary permissions of shared memory in Section 3.4.1, we relax the typing requirements on shared memory. E-ShrR and E-ShrW thus state that it is always possible to type shared memory as read or write as necessary. E-ShrW has the extra criterion of showing that the size of a shared region is sufficient to contain all the possible data that can be written. This inequality is often not provable during typing, so Section 3.4.1 will also show how to perform runtime checks to provide the guarantee. Lastly, E-LocR and E-LocW shows how to type local regions as either read or write. For a local region to be valid read memory, the index of the address and the length must match. For a local region to be writeable, it cannot have been written to before, so the address of the region must be the next available address in the data slot. Any local region containing data will have an associated length, so we must use the smallest index for which no length exists. To express this, we make use of the function `max_i` found in Definition 3.3.1.

Definition 3.3.1. $\texttt{max_i}(\sigma)$ returns the greatest i for which σ contains a binding to `length` i, or 0 if no such binding exists.

$$\texttt{max_i}(\sigma) \triangleq \texttt{max}\left(\{i \mid \texttt{length}\ i \in \texttt{image}(\sigma)\} \cup \{0\}\right)$$

Corollary 3.3.2.

$$\forall \sigma, e, i.\ \sigma \vDash e : \texttt{length}\ i \implies i \leq \texttt{max_i}(\sigma)$$

Proof. By the cases of $\sigma \vDash e : \texttt{length}\ i$ and the definition of `max_i`. □

Constructing pointers to memory regions can be done in different ways, which can complicate reasoning about pointers. As an example, a pointer of type `address` 2 can either be constructed by adding two expressions with types `address` 1 and `length` 1, or three expressions with types `address` 0, `length` 0, and `length` 1. However, the value of the pointer should be the same in both cases. In fact, for any pointer of type `address` i, the value of the pointer should be equivalent to adding the sum of all lengths with type `length` i' for $0 \leq i' \leq i$ to an expression of type `address` 0, i.e. `info` $\rightarrow$ `dynamic`.

To apply this fact in the type-checking of statements, we make use of another construct in addition to the typing context. We use a map $\omega : \texttt{OffsetExpr} \to \texttt{OffsetExpr}$ of *canonical offsets*. As offsets are added together to create new pointers, ω will keep track of the sub-terms used to construct the pointers. An offset expression e can be converted to its canonical offset $\texttt{co}(\omega, e)$ in the following manner:

Definition 3.3.3 (Canonical Offset)**.**

$$\texttt{co}(\omega, e) \triangleq \begin{cases} \texttt{co}(\omega, e_1) + \texttt{co}(\omega, e_2) & \text{if } e = e_1 + e_2 \\ \omega(e) & \text{if } e \in \omega \\ e & \text{otherwise} \end{cases}$$

Corollary 3.3.4. $\forall e.\texttt{co}(\texttt{nil}, e) = e$

Proof. By the definition of `co`. □

We define a function `initial` in Definition 3.3.5, which constructs a map from all fields defined in the layout to their relevant type, with values mapping to their C type, offset having type `length` i numbered by declaration order, and *Reqs* having type `id`. All these fields will be tagged as `Info`. In addition, `initial(layout)` will contain a binding $(\mathit{res}_r, \texttt{Resource}) \mapsto \texttt{shared}$ for all $0 \leq r < \mathit{Reqs}(\texttt{layout})$.

Definition 3.3.5.

$$\texttt{aux}(i, r, \texttt{nil}) = \texttt{nil}$$

$$\texttt{aux}(i, r, f :: \mathit{fs}) = \begin{cases} (\texttt{aux}(i+1, r, \mathit{fs}))[(n_\texttt{offset}, \texttt{Info}) \mapsto \texttt{length } i] & \text{if } f = n_\texttt{offset} \\ (\texttt{aux}(i, r+1, \mathit{fs}))[(\mathit{res}_r, \texttt{Resource}) \mapsto \texttt{shared}] & \text{if } f = n_\texttt{req} \\ (\texttt{aux}(i, r, \mathit{fs}))[(n, \texttt{Info}) \mapsto \mathit{PrimType}] & \text{if } f = \mathit{PrimType}\ n \end{cases}$$

$$\texttt{initial}(\mathit{fields}) = \texttt{aux}(0, 0, \mathit{fields})$$

Since the collection of registered functions available is device-specific, the execution of any CSF heavily depends on the device it will be executed on. Thus, we assume a map $\phi : \texttt{Name} \to \texttt{List}\langle\texttt{RogersType}\rangle$ from function names to function signatures, supplied by the device for which the program is intended to run on. Normally, the return type of a function would also be included, but as mentioned earlier, functions should not return values, so we only care about the types of the parameters.

We define the type checking of statements using the judgment $\sigma, \omega \vdash_\phi S \Downarrow \sigma', \omega'$, which should be read as: "Given a typing context σ, canonical offsets ω, and a device specification ϕ, type checking the statement S produces a new typing context σ' and canonical offsets ω'". The rules for type checking statements are given in Figure 3.6. S-Decl, S-InitDecl and S-MemDecl all handle updating the typing context following a declaration of a variable. If a variable is initialized during declaration, the initializing expression must match the declared type (S-InitDecl). If a memory region is declared, it must not share the name with a previous variable. S-AAssign and S-LAssign deals with constructing local memory regions. As shared memory regions are not to be altered, both rules require the region to be local. The assignment of a region pointer

$\boxed{\sigma, \omega \vdash_\phi \mathit{Stmt} \Downarrow \sigma', \omega'}$

$$\frac{}{\sigma, \omega \vdash_\phi \mathit{PrimType}\ n \Downarrow \sigma[(n, \texttt{Prim}) \mapsto \mathit{PrimType}], \omega} \text{ (S-Decl)}$$

$$\frac{\sigma \vdash e : \mathit{PrimType}}{\sigma, \omega \vdash_\phi \mathit{PrimType}\ n = e \Downarrow \sigma[(n, \texttt{Prim}) \mapsto \mathit{PrimType}], \omega} \text{ (S-InitDecl)}$$

$$\frac{n \notin \texttt{dom}(\sigma)}{\sigma, \omega \vdash_\phi \texttt{struct mem_range}\ n \Downarrow \sigma[(n, \texttt{Prim}) \mapsto \texttt{local}], \omega} \text{ (S-MemDecl)}$$

$$\frac{\sigma \vdash n : \texttt{local} \qquad \sigma \vdash e : \texttt{address}\ i}{\sigma, \omega \vdash_\phi n.\texttt{ptr} = e \Downarrow \sigma[(n, \texttt{Pointer}) \mapsto \texttt{address}\ i], \omega[n.\texttt{ptr} \mapsto\ \texttt{co}(\omega, e)]} \text{ (S-AAssign)}$$

$$\frac{\sigma \vdash n : \texttt{local} \qquad \sigma \vdash e : \texttt{length}\ i \qquad (n, \texttt{Size}) \notin \texttt{dom}(\sigma)}{\sigma, \omega \vdash_\phi n.\texttt{length} = e \Downarrow \sigma[(n, \texttt{Size}) \mapsto \texttt{length}\ i], \omega[n.\texttt{length} \mapsto\ \texttt{co}(\omega, e)]} \text{ (S-LAssign)}$$

$$\frac{\sigma(n, \texttt{Info}) = \tau \qquad \sigma \vdash e : \tau}{\sigma, \omega \vdash_\phi \texttt{info} \rightarrow n = e \Downarrow \sigma, \omega} \text{ (S-SAssign)}$$

$$\frac{\sigma, \omega \vdash_\phi s_1 \Downarrow \sigma'', \omega'' \qquad \sigma'', \omega'' \vdash_\phi s_2 \Downarrow \sigma', \omega'}{\sigma, \omega \vdash_\phi s_1; s_2 \Downarrow \sigma', \omega'} \text{ (S-Seq)}$$

$$\frac{\sigma(n_{id}, \texttt{info}) = \texttt{int} \qquad \sigma \vdash e : \texttt{int} \qquad \sigma \vdash n_r : \texttt{local}}{\sigma, \omega \vdash_\phi \texttt{info} \rightarrow n_{id} = \texttt{allocate_shared}(e, \&n_r) \Downarrow \sigma', \omega} \text{ (S-AllocAssign)}$$

$$\text{when } (n, \texttt{Size}) \notin \texttt{dom}(\sigma) \text{ and where}$$
$$\sigma' = \sigma[(n_{id}, \texttt{Info}) \mapsto \texttt{id}][(n_r, \texttt{Prim}) \mapsto \texttt{shared}] - (n_r, \texttt{Pointer})$$

$$\frac{\phi(n) = \overrightarrow{\tau} \qquad \sigma, \omega \Vdash \overrightarrow{e}, \texttt{insert_refs}(\overrightarrow{e}, \overrightarrow{\tau}) \Downarrow \sigma', \omega'}{\sigma, \omega \vdash_\phi n(\overrightarrow{e}) \Downarrow \sigma', \omega'} \text{ (S-FCall)}$$

FIGURE 3.6: Typing of statements

requires the expression to be an address, and assignment of the size requires the expression to be a length. In both cases, ω is updated with a new canonical offset. S-SAssign requires expressions that match the type of fields in the layout `struct` when assigning to those fields. S-Seq shows how σ and ω must be threaded through the type checking when sequencing multiple statements.

S-AllocAssign is a bit more tricky. The call to `allocate_shared` is expected to take as argument an integer denoting the number of bytes to allocate and a memory region in which to store information about the shared region. The resulting ID is stored in a field of `info`. The rule requires both the field and the number of bytes to be integers, while the region must be a local region, as we do not wish to overwrite another shared region. Once the allocation has been performed, we must take care to update the typing context to reflect the change of types. The field in `info` is now considered an `id`, a privileged class of data that should not be used in computations. The memory region is changed to a shared memory region, but we must also remove any potential type bindings of the pointer and length of the affected region.

Lastly, for function calls, S-FCall states that we need to look up the type signature of a function in ϕ. As mentioned in Section 3.3.1, type signatures of functions can contain expressions which refer to the other function arguments by their position. If $\overrightarrow{e}$ are the arguments of a function call, then `insert_refs` replaces any references of the form $\#i$ with the ith expression of $\overrightarrow{e}$. Definition 3.3.6 shows a more precise definition of `insert_refs`.

Definition 3.3.6.

$$\texttt{subst}(\overrightarrow{e}, e_\tau) = \begin{cases} \overrightarrow{e}[i] & \text{if } e_\tau = \texttt{\#}i \\ \texttt{map subst}(\overrightarrow{e}, \cdot)\ e_\tau & \text{if } \exists e'_\tau.\ e'_\tau \in e_\tau \\ e_\tau & \text{otherwise} \end{cases}$$

$$\texttt{insert_refs}(\overrightarrow{e}, \overrightarrow{\tau}) = \begin{cases} \texttt{subst}(\overrightarrow{e}, e_\tau) :: \texttt{insert_refs}(\overrightarrow{e}, \overrightarrow{\tau_r}) & \text{if } \overrightarrow{\tau} = \texttt{*(write } e_\tau) :: \overrightarrow{\tau_r} \\ \tau_0 :: \texttt{insert_refs}(\overrightarrow{e}, \overrightarrow{\tau}) & \text{if } \overrightarrow{\tau} = \tau_0 :: \overrightarrow{\tau_r} \\ \texttt{nil} & \text{otherwise} \end{cases}$$

To match arguments with types, we make use of a separate class of rules that recurse through the arguments one by one. The rules for type matching arguments in functions calls are given in Figure 3.7.

The rules deal with both matching argument types and updating the typing context and canonical offsets. F-Base states that nothing is done when there are no expressions and types left to check. Since F-NoWrite and F-Write match a single argument from each list, F-Base essentially also ensures that the number of provided function arguments matches the number expected from the function signature. F-NoWrite checks if an argument matches the expected type and makes no changes to the context or offsets, but it can only be used if either the expression is not local memory, or the expected type is not write memory with some bound. If the argument is local memory and the type is write memory, F-Write must be used. The rule states that if the argument can be typed as both local and write memory, the type checking can continue, but the type of the length of the local write region must be updated to the

$$\boxed{\sigma, \omega \Vdash \overrightarrow{e}, \overrightarrow{\tau} \Downarrow \sigma', \omega'}$$

$$\frac{}{\sigma, \omega \Vdash \texttt{nil}, \texttt{nil} \Downarrow \sigma, \omega}\ \text{(F-Base)} \qquad \texttt{next}(\sigma) \triangleq \texttt{length}(\texttt{max_i}(\sigma) + 1)$$

$$\frac{\sigma \vdash e : \tau \qquad \mathcal{T} \qquad \sigma, \omega \Vdash e', \tau' \Downarrow \sigma', \omega'}{\sigma, \omega \Vdash e :: e', \tau :: \tau' \Downarrow \sigma', \omega'}\ \text{(F-NoWrite)}$$

$$\text{Where } \mathcal{T} = (\sigma \vdash e : \texttt{local}) \implies (\nexists expr.\tau = *\texttt{write}\ expr)$$

$$\frac{\sigma \vdash n : \texttt{write}\ e \qquad \sigma \vdash n : \texttt{local} \qquad \mathcal{F}}{\sigma, \omega \Vdash \&n :: e', (*\texttt{write}\ e) :: \tau' \Downarrow \sigma', (\omega' - n.\texttt{length})}\ \text{(F-Write)}$$

$$\text{Where } \mathcal{F} = \sigma[(n, \texttt{Size}) \mapsto \texttt{next}(\sigma)], \omega \Vdash e', \tau' \Downarrow \sigma', \omega'$$

FIGURE 3.7: Rules for matching types of function parameters.

next available length type. Afterwards, we reset the canonical offset of the length by removing the binding from ω.

3.3.3 Validation Through Types

The goal of the type system is to ensure that all local regions are placed in sequence after each other without overlaps or gaps. This section introduces definitions and proves what guarantees the type system provides.

Definition 3.3.7 (Well-Typedness)**.** The initial typing context with no canonical offsets is well-typed. Any context and offsets, produced by a type-check starting from a well-typed context and offsets, is itself well-typed.

$$\forall \texttt{layout}.\ \mathsf{WT}(\mathsf{initial}(\texttt{layout}), \texttt{nil})$$
$$\forall P, \sigma, \sigma', \omega, \omega'.\ \mathsf{WT}(\sigma, \omega) \implies \sigma, \omega \vdash_\phi P \Downarrow \sigma', \omega' \implies \mathsf{WT}(\sigma', \omega')$$

Lemma 3.3.8 (Monotonic `max_i`)**.** When type checking a program, the value of `max_i` can only increase.

$$\forall \sigma, \sigma', \omega, \omega', P, \phi.\ \left(\sigma, \omega \vdash_\phi P \Downarrow \sigma', \omega'\right) \implies \texttt{max_i}(\sigma) \leq \texttt{max_i}(\sigma')$$

Proof. Induction on the derivation $\sigma, \omega \vdash_\phi P \Downarrow \sigma', \omega'$. The value of `max_i` depends on bindings to `length` i types in σ, and the only rules that change `length` bindings is S-LAssign and S-FCall. S-LAssign requires there to be no preexisting binding in σ, so no `length` binding is lost by the use of S-LAssign. S-FCall only adds bindings to $\texttt{next}(\sigma)$, which happens when it uses F-Write, and this is guaranteed to increase the value of `max_i`. □

Lemma 3.3.9 (Max Address)**.** The pointer of any region in a well-typed context σ cannot have an index greater than $\texttt{max_i}(\sigma) + 1$.

$$\forall \sigma, \omega, n, i.\ \mathsf{WT}(\sigma, \omega) \implies \sigma \vDash n.\texttt{ptr} : \texttt{address}\ i \implies i \leq \texttt{max_i}(\sigma) + 1.$$

Proof. Assume a typing context σ, canonical offsets ω, variable n, integer i, proof $\mathcal{W}$ showing $\mathsf{WT}(\sigma, \omega)$, and proof $\mathcal{E}$ showing $\sigma \vDash n.\texttt{ptr} : \texttt{address}\ i$. We know that $\sigma(n, \texttt{Pointer}) = \texttt{address}\ i$ since $\mathcal{E}$ must have used E-Ptr. We proceed by induction on $\mathcal{W}$.

In the base case, we have $\sigma = \mathsf{initial}(\texttt{layout})$ for some `layout`. But then we have $\mathsf{initial}(\texttt{layout})(n, \texttt{Pointer}) = \texttt{address}\ i$, and since $\mathsf{initial}$ does not create any bindings of the `Pointer` kind we have a contradiction and the case is trivially true.

In the inductive case, we must have a context σ' and canonical offsets ω' along with a proof $\mathcal{W}'$ showing $\mathsf{WT}(\sigma', \omega')$, and a proof $\mathcal{S}$ showing $\sigma', \omega' \vdash_\phi P \Downarrow \sigma, \omega$ for some program P and device specification ϕ. the induction hypothesis shows $\forall n', i'.\ \sigma' \vDash n'.\texttt{ptr} : \texttt{address}\ i' \implies i' \leq \texttt{max_i}(\sigma') + 1$.

We proceed by induction on $\mathcal{S}$. From Lemma 3.3.8 on $\mathcal{S}$ we get $\texttt{max_i}(\sigma') \leq \texttt{max_i}(\sigma)$, and it is therefore sufficient to show $i \leq \texttt{max_i}(\sigma') + 1$. For any case where we have $\sigma(n, \texttt{Pointer}) = \sigma'(n, \texttt{Pointer}) = \texttt{address}\ i$ we can prove our goal by using E-Ptr and the outer IH. Thus we only need to prove cases where $\mathcal{S}$ may have updated σ' with a binding of kind `Pointer`. S-AllocAssign only deletes a binding of kind `Pointer`, and since we already know $\sigma(n, \texttt{Pointer}) = \texttt{address}\ i$, n cannot have been the deleted pointer, so in this case we also have $\sigma(n, \texttt{Pointer}) = \sigma'(n, \texttt{Pointer})$. From the type checking rules for function calls in Figure 3.7, we know that S-FCall only updates with bindings of kind `Size`, so this case is also covered. This only leaves S-Seq and S-AAssign.

- **Case S-Seq**

  $$\mathcal{S} = \dfrac{\overset{\mathcal{S}_0}{\sigma', \omega' \vdash_\phi s_1 \Downarrow \sigma'', \omega''} \qquad \overset{\mathcal{S}_1}{\sigma'', \omega'' \vdash_\phi s_2 \Downarrow \sigma, \omega}}{\sigma', \omega' \vdash_\phi s_1; s_2 \Downarrow \sigma, \omega}$$

  From $\mathcal{S}_0$ and the definition of WT, we get $\mathsf{WT}(\sigma'', \omega'')$. Then by the inner IH on $\mathcal{S}_1$ we prove our goal.

- **Case S-AAssign**

  $$\mathcal{S} = \dfrac{\overset{\mathcal{E}}{\sigma' \vdash n : \texttt{local} \qquad \sigma' \vdash e : \texttt{address}\ i}}{\sigma', \omega' \vdash_\phi n.\texttt{ptr} = e \Downarrow \sigma'[(n, \texttt{Pointer}) \mapsto \texttt{address}\ i], \omega'[n.\texttt{ptr} \mapsto\ \texttt{co}(\omega', e)]}$$

  $\mathcal{E}$ must have used E-Off, so we have a proof $\mathcal{E}'$ showing $\sigma' \vDash e : \texttt{address}\ i$. We proceed by splitting $\mathcal{E}'$ into cases.

  - **Case E-Dyn**

    $$\mathcal{E}' = \dfrac{}{\sigma' \vDash \texttt{info} \to \texttt{dynamic} : \texttt{address}\ 0}$$

    We have $i = 0$ and clearly $0 \leq \texttt{max_i}(\sigma) + 1$.

– **Case E-Ptr**

$$\mathcal{E}' = \frac{\sigma'(n', \texttt{Pointer}) = \texttt{address}\ i}{\sigma' \vDash n'.\texttt{ptr} : \texttt{address}\ i}$$

We can use the outermost IH on $\mathcal{W}'$ and $\mathcal{E}'$ to show $i \leq \texttt{max_i}(\sigma') + 1$.

– **Case E-Addr**

$$\mathcal{E}' = \frac{\sigma' \vDash e_1 : \texttt{address}\ i' \qquad \overset{\mathcal{E}_2}{\sigma' \vDash e_2 : \texttt{length}\ i'}}{\sigma' \vDash e_1 + e_2 : \texttt{address}\ (i' + 1)}$$

From Corollary 3.3.2 and $\mathcal{E}_2$ we can conclude $i' \leq \texttt{max_i}(\sigma')$. Since $i = i' + 1$ we also get $i' + 1 = i \leq \texttt{max_i}(\sigma') + 1$.

□

Lemma 3.3.10 (Generic Max Address)**.**

$$\forall \sigma, \omega, e, i.\ \mathsf{WT}(\sigma, \omega) \implies \sigma \vDash e : \texttt{address}\ i \implies i \leq \texttt{max_i}(\sigma) + 1.$$

Proof. Assume a typing context σ, canonical offsets ω, expression e, integer i, proof $\mathcal{W}$ showing $\mathsf{WT}(\sigma, \omega)$, and proof $\mathcal{E}$ showing $\sigma \vDash e : \texttt{address}\ i$. We proceed by considering the cases of $\mathcal{E}$.

- **Case E-Dyn**

$$\mathcal{E} = \frac{}{\sigma \vDash \texttt{info} \to \texttt{dynamic} : \texttt{address}\ 0}$$

We have $i = 0$ and clearly $0 \leq \texttt{max_i}(\sigma) + 1$.

- **Case E-Ptr**

$$\mathcal{E} = \frac{\sigma(n, \texttt{Pointer}) = \texttt{address}\ i}{\sigma \vDash n.\texttt{ptr} : \texttt{address}\ i}$$

We can use Lemma 3.3.9 with $\mathcal{W}$ and $\mathcal{E}$ to show $i \leq \texttt{max_i}(\sigma) + 1$.

- **Case E-Addr**

$$\mathcal{E} = \frac{\sigma' \vDash e_1 : \texttt{address}\ i' \qquad \overset{\mathcal{E}_2}{\sigma' \vDash e_2 : \texttt{length}\ i'}}{\sigma' \vDash e_1 + e_2 : \texttt{address}\ (i' + 1)}$$

Here $i = i' + 1$. From Corollary 3.3.2 on $\mathcal{E}_2$ we get $i' \leq \texttt{max_i}(\sigma)$ and thus we have $i = i' + 1 \leq \texttt{max_i}(\sigma) + 1$.

□

Lemma 3.3.11 (Function Writes to Local Regions Must Have Max Address)**.**

$$\forall \sigma, \omega, n, \overrightarrow{e}, \overrightarrow{\tau}.$$
$$\sigma, \omega \Vdash \overrightarrow{e}, \overrightarrow{\tau} \Downarrow \sigma[(n, \texttt{Size}) \mapsto \texttt{next}(\sigma)], \omega - n.\texttt{length} \implies$$
$$\sigma \vDash n.\texttt{ptr} : \texttt{address}(\texttt{max_i}(\sigma) + 1)$$

Proof. Assume a typing context σ, canonical offsets ω, a variable n, an expression list $\overrightarrow{e}$, a type list $\overrightarrow{\tau}$, and a proof $\mathcal{F}$ showing $\sigma, \omega \Vdash \overrightarrow{e}, \overrightarrow{\tau} \Downarrow \sigma[(n, \texttt{Size}) \mapsto \texttt{next}(\sigma)], \omega - n.\texttt{length}$. Proceed by induction on $\mathcal{F}$. F-Nil cannot have been used, which leaves F-NoWrite and F-Write.

- **Case F-NoWrite**

$$\mathcal{F} = \frac{\sigma \vdash e : \tau \qquad \mathcal{N} \qquad \overset{\mathcal{F}'}{\sigma, \omega \Vdash e', \tau' \Downarrow \sigma[(n, \texttt{Size}) \mapsto \texttt{next}(\sigma)], \omega - n.\texttt{length}}}{\sigma, \omega \Vdash e :: e', \tau :: \tau' \Downarrow \sigma[(n, \texttt{Size}) \mapsto \texttt{next}(\sigma)], \omega - n.\texttt{length}}$$

  Induction hypothesis on $\mathcal{F}'$ proves our goal.

- **Case F-Write**

$$\mathcal{F} = \frac{\overset{\mathcal{E}}{\sigma \vdash n : \texttt{write}\ e} \qquad \sigma \vdash n : \texttt{local} \qquad \mathcal{F}'}{\sigma, \omega \Vdash \&n :: e', (*\texttt{write}\ e) :: \tau' \Downarrow \sigma[(n, \texttt{Size}) \mapsto \texttt{next}(\sigma)], (\omega - n.\texttt{length})}$$

  Since $\sigma \vdash n : \texttt{local}$, $\mathcal{E}$ must have used E-LocW. Then we must have a sub-proof of $\sigma \vdash n.\texttt{ptr} : \texttt{address}\ (\texttt{max_i}(\sigma) + 1)$, which must have used E-Off containing a proof of our goal.

□

Lemma 3.3.12 (Single Write Region)**.** A function call in a well-typed context and canonical offsets will either leave the context and offsets untouched, or update the length of exactly one region and remove the canonical offset for the length of that region.

$$\forall \sigma, \sigma', \omega, \omega', \overrightarrow{e}, \overrightarrow{\tau}.\ \mathsf{WT}(\sigma, \omega) \implies \sigma, \omega \Vdash \overrightarrow{e}, \overrightarrow{\tau} \Downarrow \sigma', \omega' \implies$$
$$(\sigma' = \sigma \wedge \omega' = \omega) \vee (\exists n. \sigma' = \sigma[(n, \texttt{Size}) \mapsto \texttt{next}(\sigma)] \wedge \omega' = \omega - n.\texttt{length})$$

Proof. Assume we have typing contexts σ, σ', canonical offsets ω, ω', list of expressions $\overrightarrow{e}$, list of types $\overrightarrow{\tau}$, proof $\mathcal{W}$ showing $\mathsf{WT}(\sigma, \omega)$, and proof $\mathcal{F}$ showing $\sigma, \omega \Vdash \overrightarrow{e}, \overrightarrow{\tau} \Downarrow \sigma', \omega'$. We proceed by induction on $\mathcal{F}$.

- **Case F-Base**

$$\mathcal{F} = \frac{}{\sigma, \omega \Vdash \texttt{nil}, \texttt{nil} \Downarrow \sigma, \omega}$$

  We immediately have $\sigma = \sigma'$ and $\omega = \omega'$, so we are done.

- **Case F-NoWrite**

$$\mathcal{F} = \frac{\sigma \vdash e : \tau \qquad \mathcal{N} \qquad \overset{\mathcal{F}'}{\sigma, \omega \Vdash e', \tau' \Downarrow \sigma', \omega'}}{\sigma, \omega \Vdash e :: e', \tau :: \tau' \Downarrow \sigma', \omega'}$$

  Induction hypothesis on $\mathcal{F}'$ proves our goal.

- **Case F-Write**

$$\mathcal{F} = \frac{\overset{\mathcal{E}}{\sigma \vdash n : \texttt{write}\ e} \qquad \sigma \vdash n : \texttt{local} \qquad \mathcal{F}'}{\sigma, \omega \Vdash \&n :: e', (*\texttt{write}\ e) :: \tau' \Downarrow \sigma'', (\omega'' - n.\texttt{length})}$$

Where $\mathcal{F}' = \sigma[(n, \texttt{Size}) \mapsto \texttt{next}(\sigma)], \omega \Vdash e', \tau' \Downarrow \sigma'', \omega''$.

Using the IH on $\mathcal{F}'$, we get $\sigma'' = \sigma[(n, \texttt{Size}) \mapsto \texttt{next}(\sigma)] \wedge \omega'' = \omega$ or $\exists n'.\sigma'' = \sigma[(n, \texttt{Size}) \mapsto \texttt{next}(\sigma)][(n', \texttt{Size}) \mapsto \texttt{next}(\sigma')] \wedge \omega'' = \omega - n'.\texttt{length}$.

- $\sigma'' = \sigma[(n, \texttt{Size}) \mapsto \texttt{next}(\sigma)] \wedge \omega'' = \omega$

  Since we have $\sigma' = \sigma''$ and $\omega' = \omega''$, we can prove the right side of the disjunction by choosing n as our witness. Rewriting the equations we get $\sigma' = \sigma'' = \sigma[(n, \texttt{Size}) \mapsto \texttt{next}(\sigma)]$ and $\omega' = \omega'' = \omega - n.\texttt{length}$.

- $\exists n'.\sigma'' = \sigma_n[(n', \texttt{Size}) \mapsto \texttt{next}(\sigma_n)] \wedge \omega'' = \omega - n'.\texttt{length}$
  where $\sigma_n = \sigma[(n, \texttt{Size}) \mapsto \texttt{next}(\sigma)]$.

  By definition of `next` and `max_i` we must have $\texttt{max_i}(\sigma_n) = \texttt{max_i}(\sigma) + 1$. By obtaining the witness n' we can rewrite σ'' and ω'' in $\mathcal{F}'$ and use Lemma 3.3.11 to construct a proof $\mathcal{E}_n$ showing $\sigma_n \vDash n'.\texttt{ptr} : \texttt{address}\ (\texttt{max_i}(\sigma_n) + 1)$. $\mathcal{E}_n$ must have used E-Ptr, and since bindings of kind `Size` does not affect lookups of kind `Pointer`, we must be able to construct a proof $\mathcal{E}_{con}$ showing $\sigma \vDash n'.\texttt{ptr} : \texttt{address}\ (\texttt{max_i}(\sigma_n) + 1)$. But then we can use Lemma 3.3.10 with $\mathcal{W}$ and $\mathcal{E}_{con}$ to show $\texttt{max_i}(\sigma_n) + 1 = (\texttt{max_i}((\sigma) + 1) + 1) \leq \texttt{max_i}(\sigma) + 1$, which is a contradiction.

□

For proving lemmas about changes to the canonical offsets, we define a `replace` function that substitutes all occurrences of an offset expression with another. We then show that in the context of converting offset expressions to their canonical form, adding a new canonical offset to an expression is equivalent to replacing the expression with the canonical offset.

Definition 3.3.13 (Replace Offset Expression)**.** Replace e_1 with e_2 in e.

$$\texttt{replace}(e_1, e_2, e) \triangleq \begin{cases} e_2 & \text{if } e = e_1 \\ \texttt{replace}(e_1, e_2, e') + \texttt{replace}(e_1, e_2, e'') & \text{if } e = e' + e'' \\ e & \text{otherwise} \end{cases}$$

Corollary 3.3.14 (Replace Equivalent to New Canonical Offset)**.**

$$\forall \omega, e, e_r, e_s.\ \texttt{co}\,(\omega[e_r \mapsto \texttt{co}(\omega, e_s)], e) = \texttt{co}(\omega, \texttt{replace}(e_r, e_s, e))$$

Proof. By induction on e followed by definition of `co` and `replace`. □

Lemma 3.3.15 (Replace Preserves Type)**.**

$$\begin{gathered}\forall e, e_0, e_1, \sigma, \tau, \tau'. \\ \sigma \vDash e : \tau \implies \\ \sigma \vDash e_0 : \tau' \implies \\ \sigma \vDash e_1 : \tau' \implies \\ \sigma \vDash \texttt{replace}(e_0, e_1, e) : \tau\end{gathered}$$

Proof. By induction on $\sigma \vDash e : \tau$. □

Lemma 3.3.16.

$$\begin{gathered}\forall \sigma, n, e, e'\tau, \tau'. \\ \sigma[(n, \texttt{Pointer}) \mapsto \tau] \vDash e' : \tau' \implies \\ \sigma \vDash e : \tau \implies \\ \sigma \vDash \texttt{replace}(n.\texttt{ptr}, e, e') : \tau'\end{gathered}$$

Proof. By induction on $\sigma[(n, \texttt{Pointer}) \mapsto \tau] \vDash e' : \tau'$ □

Corollary 3.3.17.

$$\begin{gathered}\forall \sigma, n, e, e'\tau, \tau'. \\ \sigma[(n, \texttt{Size}) \mapsto \tau] \vDash e' : \tau' \implies \\ \sigma \vDash e : \tau \implies \\ \sigma \vDash \texttt{replace}(n.\texttt{length}, e, e') : \tau'\end{gathered}$$

Proof. By induction on $\sigma[(n, \texttt{Size}) \mapsto \tau] \vDash e' : \tau'$ □

Definition 3.3.18 (Sub-expression)**.** An offset expression oe_s is a sub-expression of another oe if either the two expressions are equal, or oe is an addition of two other expressions oe' and oe'', where oe_s is a sub-expression of either.

$$oe_s \subseteq oe \triangleq (oe_s = oe) \vee (\exists oe', oe''. oe = oe' + oe'' \wedge (oe_s \subseteq oe' \vee oe_s \subseteq oe''))$$

Corollary 3.3.19.

$$\forall \sigma, e_0, e_1, i.\ e_0 \subseteq e_1 \implies \sigma \vDash e_1 : \texttt{length}\ i \implies e_0 = e_1$$

Proof. The proof for $\sigma \vDash e_1 : \texttt{length}\ i$ can only have used E-Length or E-Info. This means that e_1 cannot be an addition-expression, so the goal then follows from definition of $\subseteq$. □

Definition 3.3.20 (Not Sub-expression)**.**

$$oe_s \subsetneq oe \triangleq \neg(oe_s \subseteq oe)$$

Lemma 3.3.21 (Deterministic Offset Typing)**.**

$$\forall \sigma, e, \tau, \tau'.\ \sigma \vDash e : \tau \implies \sigma \vDash e : \tau' \implies \tau = \tau'$$

Proof. Assume a typing context σ, offset expression e, types τ,τ' and proofs $\mathcal{E}, \mathcal{E}'$ showing $\sigma \vDash e : \tau$ and $\sigma \vDash e : \tau'$, respectively. Proceed by induction on $\mathcal{E}$. If $\mathcal{E}$ used E-Dyn, $e = \texttt{info} \to \texttt{dynamic}$ so $\mathcal{E}'$ must also have used E-Dyn. Then $\tau = \texttt{address 0} = \tau'$. For the cases where $\mathcal{E}$ used E-Ptr, E-Length or E-Info, $\mathcal{E}'$ must have used the same rule, so τ and τ' both comes from looking up the same binding in σ, which implies $\tau = \tau'$. For the case of E-Addr, we have $e = e_0 + e_1$ for some e_0, e_1, as well as $\tau = \texttt{address}\ i{+}1$ and $\tau' = \texttt{address}\ i'{+}1$ for some i and i'. We must also have proofs $\mathcal{E}_0$ and $\mathcal{E}_0'$ showing $\sigma \vDash e_0 : \texttt{address}\ i$ and $\sigma \vDash e_0 : \texttt{address}\ i'$, respectively. Then by the induction hypothesis on $\mathcal{E}_0$ and $\mathcal{E}_0'$ we get $\texttt{address}\ i = \texttt{address}\ i'$, and therefore $\tau = \texttt{address}\ i + 1 = \texttt{address}\ i' + 1 = \tau'$. □

Definition 3.3.22 (Index of Offset Type)**.**

$$\texttt{index}(\tau) = \begin{cases} i & \text{if } \tau = \texttt{address}\ i \\ i & \text{if } \tau = \texttt{length}\ i \\ \texttt{undefined} & \text{otherwise} \end{cases}$$

Lemma 3.3.23 (Sub-expression Implies Lower Index)**.**

$$\forall \sigma, e_0, e_1, \tau_0, \tau_1.$$
$$e_0 \subseteq e_1 \implies \sigma \vDash e_0 : \tau_0 \implies \sigma \vDash e_1 : \tau_1 \implies \texttt{index}(\tau_0) \leq \texttt{index}(\tau_1)$$

Proof. By induction on $e_0 \subseteq e_1$.

- If $e_0 = e_1$, Lemma 3.3.21 shows $\tau_0 = \tau_1$, and clearly $\texttt{index}(\tau_0) \leq \texttt{index}(\tau_1)$.
- If $e_1 = e_1' + e_1''$ for some e_1', e_1'' and $e_0 \subseteq e_1' \vee e_0 \subseteq e_1''$, then the proof of $\sigma \vDash e_1 : \tau_1$ must have used E-Addr, so $\tau_1 = \texttt{address}\ i + 1$ for some i, and we have proofs showing $\sigma \vDash e_1' : \texttt{address}\ i$ and $\sigma \vDash e_1'' : \texttt{length}\ i$.
  - $e_0 \subseteq e_1'$

    The IH shows $\texttt{index}(\tau_0) \leq \texttt{index}(\texttt{address i}) \leq \texttt{index}(\texttt{address}\ i + 1) = \texttt{index}(\tau_1)$.
  - $e_0 \subseteq e_1''$.

    The IH shows $\texttt{index}(\tau_0) \leq \texttt{index}(\texttt{length i}) \leq \texttt{index}(\texttt{address}\ i + 1) = \texttt{index}(\tau_1)$.

□

Corollary 3.3.24.

$$\forall \sigma, e_0, e_1, i_0, i_1.$$
$$e_0 \subseteq e_1 \implies \sigma \vDash e_0 : \texttt{length}\ i_0 \implies \sigma \vDash e_1 : \texttt{address}\ i_1 \implies i_0 < i_1$$

From $e_0 \subseteq e_1$ we either have $e_0 = e_1$ or $e_1 = e_1' + e_1''$ for some e_1', e_1''. If $e_0 = e_1$, then Lemma 3.3.21 shows $\texttt{length}\ i_0 = \texttt{address}\ i_1$ which is a contradiction. Instead we must have $e_1 = e_1' + e_1''$ and $\sigma \vDash e_1 : \texttt{address}\ i_1$ must have used E-Addr. This gives us a proof $\mathcal{E}_1'$ showing $\sigma \vDash e_1' : \texttt{address}\ i_1'$ and $\mathcal{E}_1''$ showing $\sigma \vDash e_1'' : \texttt{length}\ i_1'$ with $i_1 = i_1' + 1$. We know that either $e_0 \subseteq e_1'$ or $e_0 \subseteq e_1''$. In each case, using Lemma 3.3.23 with the relevant sub-proofs shows $\texttt{index}(\texttt{length}\ i_0) = i_0 \leq i_1' < i_1' + 1 = i_1$.

Definition 3.3.25 (Canonical Equality)**.** Two offset expressions are canonically equal in ω if their canonical offsets in ω are equal.

$$\omega[e_1 = e_2] \triangleq \texttt{co}(\omega, e_1) = \texttt{co}(\omega, e_2)$$

Lemma 3.3.26 (Unique Canonical Offset)**.** If two offset expressions have the same type in a well-typed context, their canonical offsets must be equal.

$$\forall \sigma, \omega.\ \mathsf{WT}(\sigma, \omega) \implies \forall e_0, e_1, \tau.\ \sigma \vDash e_0 : \tau \implies \sigma \vDash e_1 : \tau \implies \omega[e_0 = e_1]$$

Proof. Assume a context σ, canonical offsets ω, a proof $\mathcal{W}$ showing $\mathsf{WT}(\sigma, \omega)$, offset expressions e_0, e_1, a type τ and proofs $\mathcal{E}_0, \mathcal{E}_1$ showing $\sigma \vDash e_0 : \tau$ and $\sigma \vDash e_1 : \tau$, respectively. We show that $\omega[e_0 = e_1]$ by induction of $\mathcal{W}$.

In the base case, we know that $\sigma = \texttt{initial}(\texttt{layout})$ for some layout and $\omega = \texttt{nil}$. From Definition 3.3.25 and Corollary 3.3.4, we know that our goal $\texttt{nil}[e_0 = e_1]$ reduces to showing $e_0 = e_1$. We proceed to show this by induction on $\mathcal{E}_0$. We know from the definition of `initial` that all bindings in the type context are tagged either `Info` or `Resources`, so rules E-Ptr or E-Length cannot have been used to construct $\mathcal{E}_0$.

- **Case E-Dyn**

  $$\mathcal{E}_0 = \frac{}{\texttt{initial}(\texttt{layout}) \vDash \texttt{info} \to \texttt{dynamic} : \texttt{address}\ 0}$$

  Since $\tau = \texttt{address}\ 0$ in this case, we know that $\mathcal{E}_1$ must have used either rule E-Dyn, E-Ptr or E-Addr. Like before, we know that any initial type context cannot contain bindings tagged as `Pointer`, so E-Ptr cannot have been used. E-Addr would require a proof that some expression has type $\texttt{address}(-1)$, but addresses cannot have a negative index as part of their type, so this rule cannot have been used either. This leaves us with $\mathcal{E}_1$ being constructed from E-Dyn, which implies $e_1 = \texttt{info} \to \texttt{dynamic}$, so $e_0 = e_1$.

- **Case E-Ptr**

  $$\mathcal{E}_0 = \frac{\sigma(\texttt{info} \to n_0_\texttt{offset}, \texttt{Info}) = \texttt{length}\ i}{\texttt{initial}(\texttt{layout}) \vDash \texttt{info} \to n_0_\texttt{offset} : \texttt{length}\ i}$$

We have $\tau = \texttt{length}\ i$ for some i. Again, E-Length cannot have been used to construct $\mathcal{E}_1$, so the only option must be E-Info, meaning $e_1 = \texttt{info} \rightarrow n_1\texttt{_offset}$. The construction of `initial` assigns unique `length` types to each offset field on any layout, so this can only be the case if $n_0 = n_1$, which means $e_0 = e_1$.

- **Case E-Addr**

$$\mathcal{E}_0 = \frac{\mathcal{E}_0' \qquad \mathcal{E}_0''}{\texttt{initial(layout)} \vDash e_0' : \texttt{address}\ i \qquad \texttt{initial(layout)} \vDash e_0'' : \texttt{length}\ i}{\texttt{initial(layout)} \vDash e_0' + e_0'' : \texttt{address}\ (i+1)}$$

With $\tau = \texttt{address}\ (i+1)$ for some i, $\mathcal{E}_1$ must also have used rule E-Addr with similar reasoning as the previous cases. This means we have $e_1 = e_1' + e_1''$ and $\mathcal{E}_1'$ and $\mathcal{E}_1''$ showing $\texttt{initial(layout)} \vDash e_1' : \texttt{address}\ i$ and $\texttt{initial(layout)} \vDash e_1'' : \texttt{length}\ i$ respectively. Using the inner induction hypothesis on $\mathcal{E}_0'$ and $\mathcal{E}_1'$ we obtain an equality of $e_0' = e_1'$. We can obtain the equality of $e_0'' = e_1''$ by using the IH on $\mathcal{E}_0''$ and $\mathcal{E}_1''$. Combining these equalities shows $e_0' + e_0'' = e_1' + e_1''$ which implies $e_0 = e_1$.

With the base case of $\mathcal{W}$ covered, we move on to the inductive case. In this case, we must have a context σ' and canonical offsets ω' along with a proof $\mathcal{W}'$ showing $\mathsf{WT}(\sigma', \omega')$ and a proof $\mathcal{S}$ showing $\sigma', \omega' \vdash_\phi P \Downarrow \sigma, \omega$ for some program P and device specification ϕ. We proceed by induction on $\mathcal{S}$. The cases where $\mathcal{S}$ uses S-Decl, S-InitDecl or S-MemDecl are all proven the same way. Since $\omega = \omega'$, by the outer induction hypothesis we just need to show that any expression typable in σ retains its type in σ'. Since $\sigma = \sigma'[(n, \texttt{Prim}) \mapsto \tau']$ for some n and τ' and typings of offset expressions in Figure 3.4 are unaffected by bindings tagged as `Prim`, this is trivial to show. For S-SAssign we have $\sigma = \sigma'$ and $\omega = \omega'$, so this case is also trivial.

- **Case S-AAssign**

$$\mathcal{S} = \frac{\mathcal{E}}{\sigma' \vdash n : \texttt{local} \qquad \sigma' \vdash e : \texttt{address}\ i}{\sigma', \omega' \vdash_\phi n\texttt{.ptr} = e \Downarrow \sigma'[(n, \texttt{Pointer}) \mapsto \texttt{address}\ i], \omega'[n\texttt{.ptr} \mapsto\ \texttt{co}(\omega', e)]}$$

We must show $\omega'[n\texttt{.ptr} \mapsto \texttt{co}(\omega', e)][e_0 = e_1]$. By Corollary 3.3.14 it is sufficient to show $\texttt{co}(\omega', \texttt{replace}(n\texttt{.ptr}, e, e_0)) = \texttt{co}(\omega', \texttt{replace}(n\texttt{.ptr}, e, e_1))$. $\mathcal{E}$ must have used rule E-Off, so we have a proof $\mathcal{E}'$ showing $\sigma' \vDash e : \texttt{address}\ i$. Using Lemma 3.3.16 on $\mathcal{E}_0$ and $\mathcal{E}'$ we get $\mathcal{E}_0'$ showing $\sigma' \vDash \texttt{replace}(n\texttt{.ptr}, e, e_0) : \tau$. We repeat the process to get a similar proof $\mathcal{E}_1'$ for e_1. Using the outer IH on $\mathcal{W}'$, $\mathcal{E}_0'$ and $\mathcal{E}_1'$ we get $\texttt{co}(\omega', \texttt{replace}(n\texttt{.ptr}, e, e_0)) = \texttt{co}(\omega', \texttt{replace}(n\texttt{.ptr}, e, e_1))$ and are done.

- **Case S-LAAssign**

$$\mathcal{S} = \frac{\mathcal{E}}{\sigma' \vdash n : \texttt{local} \qquad \sigma' \vdash e : \texttt{length}\ i}{\sigma', \omega' \vdash_\phi n\texttt{.length} = e \Downarrow \sigma'[(n, \texttt{Size}) \mapsto \texttt{length}\ i], \omega'[n\texttt{.length} \mapsto\ \texttt{co}(\omega', e)]}$$

We must show $\omega'[n.\texttt{length} \mapsto \texttt{co}(\omega', e)][e_0 = e_1]$. By Corollary 3.3.14 it is sufficient to show $\texttt{co}(\omega', \texttt{replace}(n.\texttt{length}, e, e_0)) = \texttt{co}(\omega', \texttt{replace}(n.\texttt{length}, e, e_1))$. $\mathcal{E}$ must have used rule E-Off, so we have a proof $\mathcal{E}'$ showing $\sigma' \vDash e : \texttt{length}\ i$. Using Corollary 3.3.17 on $\mathcal{E}_0$ and $\mathcal{E}'$ we get $\mathcal{E}'_0$ showing $\sigma' \vDash \texttt{replace}(n.\texttt{length}, e, e_0) : \tau$. We repeat the process to get a similar proof $\mathcal{E}'_1$ for e_1. Using the outer IH on $\mathcal{W}'$, $\mathcal{E}'_0$ and $\mathcal{E}'_1$ we get $\texttt{co}(\omega', \texttt{replace}(n.\texttt{length}, e, e_0)) = \texttt{co}(\omega', \texttt{replace}(n.\texttt{length}, e, e_1))$ and are done.

- **Case S-Seq**

$$\mathcal{S} = \frac{\overset{\mathcal{S}_0}{\sigma', \omega' \vdash_\phi s_1 \Downarrow \sigma'', \omega''} \qquad \overset{\mathcal{S}_1}{\sigma'', \omega'' \vdash_\phi s_2 \Downarrow \sigma, \omega}}{\sigma', \omega' \vdash_\phi s_1; s_2 \Downarrow \sigma, \omega}$$

Since $\mathsf{WT}(\sigma', \omega')$ and we have $\mathcal{S}_0$, we know by definition that $\mathsf{WT}(\sigma'', \omega'')$. Then, by the inner induction hypothesis on $\mathcal{S}_1$, we get $\omega[e_0 = e_1]$.

- **Case S-AllocAssign**

$$\mathcal{S} = \frac{\sigma' \vdash \texttt{info} \to n_{id} : \texttt{int} \qquad \sigma' \vdash e : \texttt{int} \qquad \sigma' \vdash n_r : \texttt{local}}{\sigma', \omega \vdash_\phi \texttt{info} \to n_{id} = \texttt{allocate_shared}(e, \&n_r) \Downarrow \sigma, \omega}$$

where $\sigma = \sigma'[(n_{id}, \texttt{Info}) \mapsto \texttt{id}][(n_r, \texttt{Prim}) \mapsto \texttt{shared}] - (n_r, \texttt{Size}) - (n_r, \texttt{Pointer})$

Since the typing rules for offset expressions do not rely on any `Prim` kinds and n_{id} is not an offset field, we can conclude from $\mathcal{E}_0$ and $\mathcal{E}_1$ that e_0 and e_1 have type τ in $\sigma' - (n_r, \texttt{Size}) - (n_r, \texttt{Pointer})$. If e_0 and e_1 have type τ without the bindings of $(n_r, \texttt{Size})$ and $(n_r, \texttt{Pointer})$ present in σ', they must clearly also have type τ with the bindings present, which means that we have $\sigma' \vDash e_0 : \tau$ and $\sigma' \vDash e_1 : \tau$. Then by the outer IH we have $\omega[e_0 = e_1]$.

- **Case S-FCall**

$$\mathcal{S} = \frac{\phi(n_{fun}) = \overrightarrow{\tau} \qquad \overset{\mathcal{F}}{\sigma', \omega' \Vdash \overrightarrow{e}, \texttt{insert_refs}(\overrightarrow{e}, \overrightarrow{\tau}) \Downarrow \sigma, \omega}}{\sigma', \omega' \vdash_\phi n_{fun}(\overrightarrow{e}) \Downarrow \sigma, \omega}$$

By using Lemma 3.3.12 on $\mathcal{W}'$ and $\mathcal{F}$, we get either $\sigma' = \sigma \wedge \omega' = \omega$ or $\exists n.\sigma = \sigma'[(n, \texttt{Size}) \mapsto \texttt{next}(\sigma')] \wedge \omega = \omega' - n.\texttt{length}$. In the first case we are immediately done by the outer induction hypothesis. In the other case, we know that for some n, $\mathcal{E}_0$ show $\sigma'[(n, \texttt{Size}) \mapsto \texttt{next}(\sigma')] \vdash e_0 : \tau$ and similar for $\mathcal{E}_1$. We must show $(\omega' - n.\texttt{length})[e_0 = e_1]$. We consider the cases of whether or not $n.\texttt{length}$ is a sub-expression of e_0 and e_1.

  - $n.\texttt{length} \subseteq e_0 \wedge n.\texttt{length} \subseteq e_1$:

    If we consider $n.\texttt{length} \subseteq e_0$, we either have $e_0 = n.\texttt{length}$ or $e_0 = e'_0 + e''_0$ for which $n.\texttt{length} \subseteq e'_0 \vee n.\texttt{length} \subseteq e''_0$. If $e_0 = n.\texttt{length}$, then $\mathcal{E}_0$ must have used E-Length, meaning $\tau = \texttt{length}\ i$ for some i. Then Corollary 3.3.19 with $\mathcal{E}_1$ shows $e_1 = n.\texttt{length}$. Then we have $e_0 = n.\texttt{length} = e_1$ and we must also have $\omega[e_0 = e_1]$.

If we instead have $e_0 = e_0' + e_0''$ and $n.\texttt{length} \subseteq e_0' \vee n.\texttt{length} \subseteq e_0''$, we know $\mathcal{E}_0$ must have used E-Addr, and $\tau = \texttt{address}\ i{+}1$ for some i. We must also have proofs $\mathcal{E}_0'$ and $\mathcal{E}_0''$ showing $\sigma \vDash e_0' : \texttt{address}\ i$ and $\sigma \vDash e_0'' : \texttt{length}\ i$. Applying Lemma 3.3.10 on $\mathcal{W}$ and $\mathcal{E}_0$, we get $i + 1 \leq \texttt{max_i}(\sigma) + 1$ and thereby $i \leq \texttt{max_i}(\sigma)$. From this we can conclude $n.\texttt{length} \subsetneq e_0'$, because if we have $n.\texttt{length} \subseteq e_0'$, we can show $\sigma \vDash n.\texttt{length} : \texttt{length}\ \texttt{max_i}(\sigma)$ and use Corollary 3.3.24 on $\mathcal{E}_0'$ to get $\texttt{max_i}(\sigma) < i \leq \texttt{max_i}(\sigma)$, which is a contradiction. We must then have $n.\texttt{length} \subseteq e_0''$ instead. By Corollary 3.3.19 we then have $e_0'' = n.\texttt{length}$.

Following the same logic, we can show $e_1 = e_1' + n.\texttt{length}$, with $n.\texttt{length} \subsetneq e_1'$ and a proof $\mathcal{E}_1'$ showing $\sigma \vDash e_1' : \texttt{address}\ i$. Our goal is to show $\omega[e_0 = e_1]$ which is equivalent to showing $\omega[e_0' + n.\texttt{length} = e_1' + n.\texttt{length}]$. This simplifies to showing $\texttt{co}(\omega, e_0') = \texttt{co}(\omega, e_1')$. Since $\omega = \omega' - n.\texttt{length}$ and we have $n.\texttt{length} \subsetneq e_0'$ and $n.\texttt{length} \subsetneq e_1'$, it is sufficient to show $\texttt{co}(\omega', e_0') = \texttt{co}(\omega', e_1')$. We can also obtain proofs of $\sigma' \vDash e_0' : \texttt{address}\ i$ and $\sigma' \vDash e_1' : \texttt{address}\ i$ from $\mathcal{E}_0'$ and $\mathcal{E}_1'$, and then we are done by the outer IH.

– $n.\texttt{length} \subseteq e_0 \wedge n.\texttt{length} \subsetneq e_1$:

We must either have $\tau = \texttt{length}\ i$ for some i or $\tau = \texttt{address}\ i$ for some i. If $\tau = \texttt{length}\ i$, then Corollary 3.3.19 shows $e_0 = n.\texttt{length}$ and $\mathcal{E}_0$ must have used E-Length, giving us $\tau = \texttt{next}(\sigma') = \texttt{length}\ i$ and therefore $i = \texttt{max_i}(\sigma') + 1$. Since $n.\texttt{length} \subsetneq e_1$, we can conclude $\sigma' \vDash e_1 : \texttt{length}\ i$ from $\mathcal{E}_1$, but then by Corollary 3.3.2 we get $i = \texttt{max_i}(\sigma') + 1 \leq \texttt{max_i}(\sigma')$, which is a contradiction.

If $\tau = \texttt{address}\ i$, we can use E-Length to construct a proof $\sigma \vDash n.\texttt{length} : \texttt{next}(\sigma')$, and then from Corollary 3.3.24 we get $\texttt{index}(\texttt{next}(\sigma')) = \texttt{max_i}(\sigma') + 1 < i = \texttt{index}(\texttt{address}\ i)$. Since we have $\sigma' \vDash e_1 : \texttt{address}\ i$, we can use Lemma 3.3.10 together with $\mathcal{W}$ to show $\texttt{index}(\texttt{address}\ i) = i \leq \texttt{max_i}(\sigma') + 1$, but then we get $\texttt{max_i}(\sigma') + 1 < i \leq \texttt{max_i}(\sigma') + 1$, which is a contradiction.

– $n.\texttt{length} \subsetneq e_0 \wedge n.\texttt{length} \subseteq e_1$:

Symmetric case of $n.\texttt{length} \subseteq e_0 \wedge n.\texttt{length} \subsetneq e_1$.

– $n.\texttt{length} \subsetneq e_0 \wedge n.\texttt{length} \subsetneq e_1$:

If $n.\texttt{length}$ is not a sub-expression in either e_0 or e_1, it is sufficient to show $\omega'[e_0 = e_1]$. From $\mathcal{E}_0$ and $\mathcal{E}_1$ we can obtain proofs $\mathcal{E}_0'$ and $\mathcal{E}_1'$ showing $\sigma' \vDash e_0 : \tau$ and $\sigma' \vDash e_1 : \tau$, and our proof is finished by the outer IH.

□

Definition 3.3.27 (Region Validity)**.** A region at index i is valid if the pointer and length of the region both include i in their type.

$$\mathsf{ValidLocal}(\sigma, r, i) \triangleq (\sigma \vDash r.\texttt{ptr} : \texttt{address}\ i \wedge \sigma \vDash r.\texttt{length} : \texttt{length}\ i)$$

Definition 3.3.28 (Contiguity)**.** Two regions are contiguous, if adding the length of the first region to its pointer produces an expression that is canonically equivalent to the pointer of the second region.

$$\mathsf{contiguous}(\omega, r, s) \triangleq \omega[r.\texttt{ptr} + r.\texttt{length} = s.\texttt{ptr}]$$

Definition 3.3.29 (Non-Fragmentation)**.** A typing context with canonical offsets is considered non-fragmented if all valid regions are contiguous with the region at the next index.

$$\begin{gathered}\mathsf{NF}(\sigma, \omega) \triangleq \forall r, s, i.\mathsf{ValidLocal}(\sigma, r, i) \implies \\ \sigma \vDash s.\texttt{ptr} : \texttt{address}(i+1) \implies \\ \mathsf{contiguous}(\omega, r, s)\end{gathered}$$

Lemma 3.3.30 (Well-typed implies Non-Fragmentation)**.**

$$\forall \sigma, \omega.\ \mathsf{WT}(\sigma, \omega) \implies \mathsf{NF}(\sigma, \omega)$$

Proof. Assume a typing context σ, canonical offsets ω and a proof $\mathcal{W}$ showing $\mathsf{WT}(\sigma, \omega)$. Assume further variables r and s and index i such that we have proofs $\mathcal{E}_r$ showing $\mathsf{ValidLocal}(\sigma, r, i)$ and $\mathcal{E}_s$ showing $\sigma \vDash s.\texttt{ptr} : \texttt{address}\ (i+1)$. By definition of $\mathsf{ValidLocal}$, we have proofs $\mathcal{E}'_r$ and $\mathcal{E}''_r$ showing $\sigma \vDash r.\texttt{ptr} : \texttt{address}\ i$ and $\sigma \vDash r.\texttt{length} : \texttt{length}\ i$. Using rule E-Addr with $\mathcal{E}'_r$ and $\mathcal{E}''_r$ we obtain a proof $\mathcal{E}_{r+1}$ showing $\sigma \vDash r.\texttt{ptr} + r.\texttt{length} : \texttt{address}\ (i+1)$. Using Lemma 3.3.26 with $\mathcal{W}$ on $\mathcal{E}_{r+1}$ and $\mathcal{E}_s$ we obtain $\omega[r.\texttt{ptr} + r.\texttt{length} = s.\texttt{ptr}]$. □

Theorem 3.3.31. All type-checked programs P with layout L are non-fragmented.

$$\forall P, L, \sigma, \omega.\ (\mathsf{initial}(L), \texttt{nil} \vdash_\phi P \Downarrow \sigma, \omega) \implies \mathsf{NF}(\sigma, \omega)$$

Proof. Since $\mathsf{WT}(\texttt{initial}(L), \texttt{nil})$ we know by definition that $\mathsf{WT}(\sigma, \omega)$. Lemma 3.3.30 then shows our goal and the theorem is proved. □

3.4 Implementation

This section covers the implementation of Rogers, the validity checker for CSFs written in the Rogers DSL. The source code can be found in Appendix C or at `https://github.com/mortentc/Rogers`.

As the DSL is a strict subset of C, it is possible to use a premade C-parser to generate an abstract syntax tree (AST) for analysis. With this in mind, we choose to implement Rogers in Haskell, relying on the `language-c`[1] Haskell package for parsing. The parser is based on the "Scrap Your Boilerplate" design pattern [53], which simplifies the traversal of the generated AST, but also requires a little explanation.

[1] `https://hackage.haskell.org/package/language-c`

The essence of the design pattern consists of a safe type-casting function and a generic `Data` type class.[2] The type-casting function has the following type:

```
1 cast :: (Typeable a, Typeable b) => a -> Maybe b
```

What it does is take an argument `x` of type `a`, and if `x` can be cast to type `b` at runtime, the `cast` returns `Just x` and `Nothing` otherwise. This rather simple function allows for defining functions whose implementation only cares about input of a specific type, while typing the function to accept any input.

The `Data` type class is then used for data structures that define a generic traversal `gfoldl` of the structure. Combining the type class and the `cast` function, the design pattern makes it possible to define *generic queries* and *generic transformations* that work for any `Data`.

To give an example of what this means for the analysis of our C-AST, assume that we want to count the number of statements that occur in the AST. For this purpose, we would need a query that returns 1 if a node in the AST is a statement, and 0 otherwise.

```
1 statementCounter :: CStat -> Int
2 statementCounter _ = 1
3
4 statementCounterQuery :: GenericQ Int
5 statementCounterQuery = mkQ 0 statementCounter
```

The `mkQ` transforms our type-specific `statementCounter` into a generic query, which returns 1 if the input can be cast to `CStat`, or our user-supplied default value of 0 otherwise. To apply the query to an entire data structure, we can use the `everything` function:

```
1 statementsInAst :: GenericQ Int
2 statementsInAst = everything (+) statementCounterQuery
```

With this we now have a query that can count the number of statements in any `Data` structure (even those that cannot possibly contain C-statements), without having to define the traversal of the structure.

With the dependencies covered, let us dive into the implementation. Rather than going through all the code, we can look at the overall design idea. First, in Listing 3.4, we have the types mentioned in the validation section. Line 2 defines the simple types that correspond to C types, with the addition of a `Unit` type to cover expressions that should not have a Rogers type (namely, function calls). Line 3 contains the types used for the address and length of a memory region. Line 4 corresponds to the `mem_range`, `layout`, and `request struct`, with `mem_range` being either `local` or `shared`. Lastly, the `Access` type is used for typing memory as either `read` or `write`. In the case of `write`, the last line shows that we include a `CExpr` in the type, to act as the representation of the lower bound of a write region.

[2] https://hackage-content.haskell.org/package/base-4.19.2.0/docs/Data-Data.html

Listing 3.4: Rogers Types

```
data RogersType =
    Unit | Byte | Integer | Float | Id | Pointer RogersType |
    Address Int | Length Int |
    Memory Placement | Layout | Request |
    Access Permission
    deriving Show
data Placement = Local | Shared deriving Show
data Permission = Read | Write CExpr deriving Show
```

To carry out the validation, we also define a state in Listing 3.5 that stores necessary information to check if the CSF follows the syntax in Section 3.2 and the type system in Section 3.3. The number of *Off* fields and *Reqs* fields defined in the layout is stored in `lengthCount` and `reqCount`, respectively. The map ϕ, which contains the signatures of the device-specific functions, is stored in `funcs`. The types of variables, corresponding to the typing context σ, are stored in `vars`. Recall that in Section 3.2 we mentioned that shared regions would be analyzed to determine their required permissions. Indices in the `resources` array, for which the region at the index must be a write region, are stored in the set `writes`. Alterations to the CSF are stored as program statements in `modifiedBody`. These altered statements include setting the necessary permissions for regions, and run-time checks to ensure that writes stay within bounds. The alteration of the statements is covered more in-depth in Section 3.4.1. Lastly, `errs` is a list of errors used in debugging.

Listing 3.5: Information used in validity checking

```
data AbstractState = AbstractState {
  lengthCount :: Int,
  reqCount :: Int,
  funcs :: Device,
  vars :: Variables,
  writes :: Set Integer,
  modifiedBody :: [CBlockItem],
  errs :: [String]
}
type Device = Map String [RogersType]
type Variables = Map String RogersType
```

In this implementation, we use a combined error and state monad for most of our analysis, with the `Maybe` type used to indicate errors and the `AbstractState` from earlier as our state. The definition of the monad is shown in Listing 3.6.

Listing 3.6: Definition of the Verification Monad.

```
newtype ValidationM a = Valid {
  runValid :: AbstractState -> (Maybe a, AbstractState)
}
instance Monad ValidationM where
  Valid comp >>= f = Valid $ \s ->
      let (res, s') = comp s in
      case res of
        Nothing -> (Nothing, s')
        Just r -> runValid (f r) s'
```

Using this monadic definition, it is easy to create a mechanism for logging errors and indicating when a boolean property encountered during the analysis does not hold (Listing 3.7).

Listing 3.7: Functions for error-logging and validating properties.

```
logErrM :: String -> ValidationM a
logErrM err = Valid $ \s -> (Nothing, s { errs = err : errs s })

validateM :: Bool -> String -> ValidationM ()
validateM check errMsg
  | check = return ()
  | otherwise = logErrM errMsg
```

Most of the implementation then boils down to defining boolean checks at the different syntax nodes of the parsed AST, corresponding to the rules in Section 3.3.2. We show how this looks in Listing 3.8 using the rule for addition of two *OffsetExpr*. This simplifies much of the implementation of the analysis to translating the rules from the validation section into boolean properties.

Listing 3.8: Analysis of an addition expression, with different rules depending on if the type of the operands are offset expressions or basic numerics.

```
indexMatch (Address ia) (Length is) = ia == is
indexMatch _ _ = False

typeExprM :: CExpr -> ValidationM RogersType
typeExprM (CBinary CAddOp e1 e2 info) = do
  t1 <- typeExprM e1
  t2 <- typeExprM e2
  case t1 of
    Address i -> do
      validateM (indexMatch t1 t2) "mismatching types in addition"
      return (Address $ i+1)
    Length i -> do
      validateM (indexMatch t1 t2) "mismatching types in addition"
      return (Address $ i+1)
    _ -> do
      validateM (t1 == t2) "mismatching types in addition"
      return t1
```

3.4.1 Emitting Code

In Section 3.2 and again in Section 3.3.2 we made certain assumptions regarding memory regions. One was that shared regions would automatically receive the required permissions based on analysis of the CSF. The other was a run-time check on the sizes of shared regions being written to. This section will show how to guarantee these assumptions. The requirements that the implementation must check for are summarized in Table 3.1.

TABLE 3.1: Read and write requirements for local and shared regions.

	Shared	**Local**
Read	No extra requirements	Matching address and length types
Write	Exclusive access and check size	Must be located at the lowest available address

To meet the requirements, we define a function `canBeTyped` to be used when we need to validate whether a region can be typed as required by a function signature and update the abstract state accordingly. The checks correspond to the typing rules E-ShrR, E-ShrW, E-LocR and E-LocW defined in Figure 3.5. Listing 3.9 shows the checks performed to type read regions. As just talked about, reading shared regions imposes no extra requirements, so line 5 just returns the check with no issue or modifications. If the region is local, lines 6-11 try to type the pointer and length of the region, and if successful, validate whether the indices in the types match.

LISTING 3.9: Fulfilling requirements on read regions.

```
1  canBeTyped :: CExpr -> RogersType -> ValidationM [CStat]
2  canBeTyped expr (Access Read) = do
3    typ <- typeExprM expr
4    case typ of
5      Memory Shared -> return []
6      Memory Local -> do
7        n <- justOrErrM nameFromVar expr "non-var local mem_range"
8        ptrTyp <- typeOfVar $ printf "%s.ptr" n
9        lenTyp <- typeOfVar $ printf "%s.length" n
10       validateM (indexMatch ptrTyp lenTyp) $
11         mkErr ("Mismatch of pointer and length in mem_range "++n) expr
12       return []
13     _ -> do
14       validateM False "Mismatch on types in function call"
15       return []
```

For the write requirements, Listing 3.10 shows the checks performed. The case for typing shared memory as write is covered on lines 4-6. Line 5 makes a C assertion of the form `assert`(*bound* $\leq$ *expr*`.length`). Line 6 tries to retrieve the index used in the expression, assuming that the expression is of the form `resources[`i`]`. If the expression is indexing into `resources`, the index is recorded as an index for which write permission is required. If not, the expression must be a region obtained from a call to `allocate_shared`, in which case it already has write permission, and no further requirements are imposed. Lines 7-14 handle the remaining case of typing a local region as a write region. Only the pointer of the region needs to be typed, but it must be typed as an address with index one greater than any other region in the

abstract state. Lastly, as the region is being written to, the length of the region is being updated with the appropriate type.

Listing 3.10: Fulfilling requirements on write regions.

```
canBeTyped (CUnary CAdrOp expr _) (Pointer (Access (Write bound))) = do
  typ <- typeExprM expr
  case typ of
    Memory Shared ->
      (:) (mkAssert expr bound) <$>
      maybe (return []) trackAsWrite (getIndexExpr expr >>= getNum)
    Memory Local -> do
      n <- justOrErrM nameFromVar expr "non-var local mem_range"
      ptrTyp <- typeOfVar $ printf "%s.ptr" n
      maxI <- getFromStateM lengthMax
      validateM (ptrTyp == Address (maxI + 1))
        (n++".ptr does not point to a writable region")
      updateEnv (printf "%s.length" n) (Length (maxI + 1))
      return []
    _ -> do
      validateM False "Mismatch on types in function call"
      return []
```

With these checks performed, Rogers can request the necessary permissions at program start for shared regions, as first mentioned in Section 3.2, and insert run-time assertions on shared regions before function calls, as required by Section 3.3.2. Using this approach, running Rogers on example programs 0 and 1 from Listing 3.2 and 3.3, we end up with the modified programs in Listings 3.11 and 3.12, respectively. Naturally, for Rogers to execute correctly, we need a device specification containing the signatures of relevant registered functions. For illustration purposes, we have hard-coded a map with the signatures below. For the `read_file` signature, there is no simple expression for the required size of the write region, so we set it to 0 and rely on the fields `event_fsize` and `date_fsize` to correspond to the actual file sizes. We discuss this situation further in Section 3.5.2.

$$\phi(\texttt{read_file}) = [\texttt{read}, \texttt{write}\ (0)\ \texttt{pointer}]$$
$$\phi(\texttt{filter}) = [\texttt{read}, \texttt{write}\ (\#0.\texttt{length})\ \texttt{pointer}, \texttt{integer}, \texttt{integer}]$$
$$\phi(\texttt{select}) = [\texttt{read}, \texttt{read}, \texttt{write}\ (\#1.\texttt{length})\ \texttt{pointer}]$$

Listing 3.11: Modifed function body from running Rogers on program 0 from 3.2

```
1  int main(void * context)
2  {
3      struct layout * info = (struct layout *) context;
4      struct request reqs[0];
5      struct mem_range * resources = obtain(reqs, 0);
6      struct mem_range event_fname;
7      event_fname.ptr = info->dynamic;
8      event_fname.length = info->event_fname_offset;
9      struct mem_range event_placement;
10     info->event_id = allocate_shared(info->event_fsize, &event_placement);
11     assert(0 <= event_placement.length);
12     read_file(event_fname, &event_placement);
13
14     struct mem_range date_fname;
15     date_fname.ptr = info->dynamic + info->event_fname_offset;
16     date_fname.length = info->date_fname_offset;
17     struct mem_range date_placement;
18     info->date_id = allocate_shared(info->date_fsize, &date_placement);
19     assert(0 <= date_placement.length);
20     read_file(date_fname, &date_placement);
21     return 0;
22 }
```

With program 0 allocating new shared regions rather than requesting existing regions, the only modifications to the program are on lines 11 and 19, containing the assertions regarding the write memory for `read_file`. As per the above discussion, these assertions are emitted just before the relevant function calls are made on lines 12 and 20.

Listing 3.12: Modifed function body from running Rogers on program 1 from Listing 3.3

```
1  int main(void * context)
2  {
3      struct layout * info = (struct layout *) context;
4      struct request reqs[2];
5      reqs[0].id = info->event_file_req;
6      reqs[0].p = Read;
7      reqs[1].id = info->date_file_req;
8      reqs[1].p = Read;
9      struct mem_range * resources = obtain(reqs, 2);
10     struct mem_range filter_res;
11     filter_res.ptr = info->dynamic;
12     filter(resources[1], &filter_res, info->comp_op, info->date);
13     struct mem_range select_res;
14     select_res.ptr = info->dynamic + filter_res.length;
15     select(resources[0], filter_res, &select_res);
16     info->res_start = filter_res.length;
17     info->res_length = select_res.length;
18     return 0;
19 }
```

Program 1, however, does request existing regions (the one allocated by program 0), and as such, lines 5-8 fill in the IDs and permissions before the `obtain` call on line 9. The calls to `filter` and `select` on lines 12 and 15 do not generate

any assertions, since the memory we write to is local memory. This can only be seen from lines 6 and 8, as the shared regions are only read from. If we were to look at the final typing context σ for this program, we would see the bindings `filter_res.ptr` $\mapsto$ `Address 0`, `filter_res.length` $\mapsto$ `Length 0` as well as `select_res.ptr` $\mapsto$ `Address 1`, `select_res.length` $\mapsto$ `Length 1`, indicating that the results of the `filter` function are located immediately after the layout struct (same place as `dynamic[]`), followed by the results of the `selection` function.

For the sake of example, we could decide that we want to reuse one of the shared regions to store the results of the `select` function. Changing line 15 to "`select(resources[0], filter_res, &resources[1]);`", Rogers would change line 8 to "`reqs[1].p = Write;`" and we would end up with an assertion of the form: "`assert(filter_res.length <= resources[1].length);`". Additionally, we would get an error as `select_res.length` is now untyped, but we still try to access it on line 17. This makes sense as we would no longer place the final result of the CSF in local memory, so we should not communicate a local placement back to the host. Rather, the lines 16+17 should be removed, and the ID corresponding to the shared region we wrote to should serve as the indicator for the location of the results.

3.5 Discussion

3.5.1 Multiple Write Regions

As apparent from Lemma 3.3.12, it is only possible to have one local write region at any given time. This is not an issue if a function does not write output to more than one region at a time, but it may be desirable for some functions to have two output regions. As an example, a function that partitions data on some predicate could benefit from writing the resulting partitions into two different regions. Using the current type rules of Rogers, it is possible to call such a function, but only one of the write regions can be local. The rest must be in shared memory.

If it becomes necessary to support multiple local write regions, the typing of function calls and address pointers could be updated to allow this change. Currently, for a local region to be a write region, its pointer must have the greatest `address` type of any local region, ordered by the indices in the type. To allow multiple local write regions, the typing of a write region can be changed to also require the length of the region to be equal to the size bound of the function signature. Multiple write regions can then exist as long as their pointers are offset according to the bounds in the signature. However, this would mean that the type system can no longer guarantee non-fragmentation, as regions would be placed according to estimated sizes rather than actual sizes, which naturally can leave some gaps between regions. The decision between multiple local writes or non-fragmentation should be made on a per setup basis, based on whether maximizing locality of minimizing fragmentation is preferred.

3.5.2 Write Regions of Indeterminate Length

Depending on the function, it does not always make sense to define an upper bound on a write region. An example of this is the `read_file` function used in Listing 3.2. Clearly, any write region used with this function must be large enough to contain the contents of the file to read, but there is no simple C-expression that can state this bound. A possible solution to this could be to define a special `file_size`-function that returns the number of bytes in a specified file and use that as our bound, but then we end up with calling one function just to check whether another can safely execute. We may also include this check in the implementation of `read_file` instead. As an additional parameter, the function could take a flag to indicate whether the write region is a shared region, and if it is, check that the length of the region is sufficient to store the file about to be read. If it is insufficient, the function sets the length to 0 and exits to indicate that there are no valid data in the region.

If the responsibility of checking sizes is moved into the body of some registered functions, it is necessary to make the bound on write regions optional. If the dependent type for write regions in function signatures instead contains an optional expression, the rule E-ShrW in Figure 3.5 can be slightly modified to only require a bounds check when a concrete bound exists.

It is not only for I/O functions where skipping the bound makes sense, but also for functions where the write region serves as both input and output. Let us say that we have a function that turns strings of ASCII characters to lower-case by overwriting the supplied input string. In this case, since the length of the write region is equivalent to how much data are written, assertions on the length of the region become redundant.

3.5.3 Reusing Shared Regions

In Section 3.4.1 we stated that runtime checks of shared regions are emitted just before they are used in function calls. As a reminder, all the checks deal with the minimum required size of shared regions, so obviously any checks must be performed *before* passing a shared region as argument to a function. But even still, we have two choices: (a) checking immediately after the shared regions have been acquired from the `obtain`-call, or (b) immediately before the function call where the region is used as a parameter. The difference between the two approaches may not be immediately apparent. In fact, if a shared region is not used after it has been written to, the two approaches are equivalent. The difference appears if a shared region is part of an assertion of a function call that happens after the region has been written to. Due to the length of a write region being updated during a function call, the result of the assertion can change depending on if approach (a) or (b) is used. As mentioned in Section 3.2, registered functions are required to update the length of any write region it modifies, in order to reflect how much data have been written to it. This is mainly necessary for regions of local memory, but since functions are intended to be agnostic towards the placement of regions, this update also affects shared memory regions. It is important to note that the function does not actually modify the true length of the shared memory region, but only the representation (a.k.a. the `mem_range struct`) local to the program.

The question in general is thus whether to make any assertions on the true / max length of the region, or the length according to the local representation. We will take a look at an example to discuss when the two approaches make the most sense. Assume that we have a shared region capable of holding n bytes of data. A file is read into the region, filling in $0.8n$ bytes of data. If we then want to filter the data, it would make most sense to assert that the region to store the output of the filtering is at least $0.8n$ bytes long, rather than n, as it is impossible for the filter function to produce more than $0.8n$ bytes of output in this instance. This creates a strong argument for using the local representation of the length in the assertions. However, if we want to reuse the region to store a full n bytes of data, relying on the local representation would prevent that, even though we know that the region is able to store the entire file. In this thesis, we chose to use the local representation, as the scenario that benefits from using the true length requires storing and then overwriting data in a shared region. This would mean that temporary data had been stored in exclusive shared memory, with no intention of communicating that data to other CSFs, which indicates that shared memory should not have been used for the data in the first place. The exception to this is if shared memory is used for the reason discussed in Section 3.5.1.

The compromise on the issue would be to change the approach based on the context. Recall that all assertions are of the form $bound \leq n.\texttt{length}$. If the region length is used on the right-hand side of the inequality, it makes sense to test on the max length of the region, as we are checking how much data the region is able to store. If it is on the left-hand side, it is part of the bound and thus is an indication of how much data can potentially be generated by the function, in which case it makes most sense to test on how much actual data the region contains. However, testing on different representations of the length requires making the assertions at different program points, which can cause conflicts if both sides of the assertion contain a shared region. To circumvent this, it would be necessary to store the original lengths of shared regions in meta-variables at the beginning of the program, such that we have access to both the max length and the current length for assertions.

3.5.4 Estimating Space Usage

In Section 3.1 we covered how earlier research had concluded that static-sized data slots were detrimental to the performance of the CSD [39]. When we showed the type system in Section 3.3, we did not make explicit assumptions about the size of data slots, other than assuming that we would always be able to fit local data into a data slot. Depending on how data slots are sized, Rogers can be modified to do away with the assumption. If data slots have a fixed size, we can use the write bound in function signatures to make run-time assertions on local regions, similar to what we did in Section 3.4.1 for shared regions. If we construct a pointer to the end boundary of the data slot, the length of a local write region is then just the difference between its own pointer and the boundary pointer.

The more interesting case is when data slots can be dynamically allocated by the host application. For efficient space usage, the host application should not allocate data slots larger than necessary, and Rogers can be used to estimate a data slot size that

is guaranteed to be able to contain the local regions created by a CSF. Naturally, a data slot needs to be large enough to contain the layout-`struct` of a CSF as well as the possible dynamic fields on the layout. The size of local regions created by the CSF can be estimated using the bounds stored in the function signatures of any registered functions called by the CSF. The total estimated data slot size needed by a CSF is then `sizeof(layout)`, plus the sum of the offset fields on the `layout` and the sum of all bounds of registered functions that write to a local region. As discussed in Section 3.5.2, it may not be possible to express a sensible bound for all functions, so this approach is not a silver bullet for size estimation, but provides a solid foundation for many scenarios.

3.6 Conclusion

In this chapter, we have covered the Rogers DSL, a subset of C that can be used to write computational storage functions for compositional storage devices. The main feature of the DSL is the use of offset expressions to construct local memory regions, which can be type checked by the accompanying validator: Rogers. Rogers is the first attempt at using a type system for memory management in the context of computational storage. Typing rules for offset expressions and run-time checks prevent misaligned accesses and buffer overflows, a guarantee that has been formally proven to hold.

Chapter 4

Registered Functions

This chapter will serve as both a case study on creating registered functions for CSDs and a tutorial for verifying these functions. Hedam et al. evaluated the performance of using a programmable CSD, named Delilah, to offload parts of database queries [39]. The evaluation focused on query 3.3 from the star schema benchmark suite [69], shown in Listing 4.1. In the evaluation setup, the CSD stored the table data and a host application written in C++ executed the query. From Listing 4.1, Hedam et al. identified three operations as candidates for computational storage programs:

- The **sum (aggregation)** on line 1
- An implicit **conversion** of dates from strings to integers on line 8
- The **filter**ing in the where-clause on lines 3-8

These three operations will be the basis for the registered functions covered in this chapter.

Listing 4.1: Star Schema Benchmark for evaluating the performance of database systems.

```
1  select c_city, s_city, d_year, sum(lo_revenue) as revenue
2  from customer, lineorder, supplier, date
3  where lo_custkey = c_custkey
4    and lo_suppkey = s_suppkey
5    and lo_orderdate = d_datekey
6    and (c_city='UNITED KI1' or c_city='UNITED KI5')
7    and (s_city='UNITED KI1' or s_city='UNITED KI5')
8    and d_year >= 1992 and d_year <= 1997
9  group by c_city, s_city, d_year
10 order by d_year asc, revenue desc;
```

The Delilah CSD used eBPF as the offload mechanism, so the offloaded programs were written in C and compiled to eBPF before being downloaded to the CSD. To keep both the translation and comparison between the original Delilah programs and the registered functions simpler, the registered functions are also written in C.

When it comes to verification of C code, there are multiple options. Why3 and Frama-C support annotating C code and generating verification conditions from the annotations [20, 26], and Verifiable C can parse C code into an abstract syntax tree that is representable in Rocq's Gallina language, enabling Rocq to prove properties

about the code [9]. It is our belief that verification tools focusing on automation has a greater chance of being adopted by the storage community, so we will not be doing the entire verification using Rocq, though we can still rely on it if necessary, as Section 4.2.3 will show. Why3 only supports a subset of C, so our tool of choice will be Frama-C with the WP plug-in [26, 18]. Frama-C + WP allows us to write a specification via annotations in the source code by using the ANSI/ISO C Specification Language (ACSL) [15].

A key aspect of the three functions under consideration is that not only do they produce outputs smaller than their inputs, which is a necessary prerequisite for computational storage to be efficient, but they do so in three distinctly different ways. The sum operation produces a result of constant size that can fit within a register, independent of the size of the table column. The conversion operation produces a 4-byte integer for each 8-byte string passed to it, so it will always produce output half the size of the input. The size of the output of the conversion can thus be expressed as a product of the number of entries and the size of an integer. The filter operation is a less clear case, as the output size depends on the contents of the input and the filtering predicate. This makes it impossible to determine a concrete output size, and instead a bound has to be used. We will take a look at how the different output sizes can be expressed in the specifications.

4.1 Requirements

While registered functions can be implemented in almost any way a registrant deems fitting, we do have some constraints on the function signatures, in order for the functions to match the behavior of the DSL, defined in Chapter 3. For one, the compositional device programming paradigm dictates that CSFs are the components responsible for orchestrating data placements. As such, registered functions should never allocate memory themselves but only interact with the memory explicitly passed to them as parameters. When it comes to verification, this restriction serves as a boon, since dynamic allocation complicates reasoning about memory. In fact, Frama-C does not yet support annotations regarding allocation [14].

Secondly, as the DSL deals with data through the use of memory regions, it is necessary for registered functions to use a similar construct when it comes to reading and writing data. We already require memory to be passed to registered functions as parameters, but this memory must at the very least be represented using a location and a length. For these examples written in C, we can simply use pointers and integers to denote these attributes, but some implementation methods might require different language structures, such as lists or arrays.

Lastly, the DSL requires all registered functions to update the length of any region they write to, in order to determine exactly how many bytes have been written to a region. These requirements will likely force us to rewrite parts of the examples, but one of the goals is still to have the registered functions match the originals as closely as possible. Part of the case study is to investigate how feasible it is to verify functions used in computational storage without forcing registrants to change the way they write code. This is why we have chosen to base the functions on real-life

computational storage programs, and as such we will try to minimize any rewriting of the functions.

4.2 Specification & Verification

4.2.1 Sum

Verifying that a function sums an array is a rather trivial task, depending on the verification framework. In fact, the Why3 platform already contains modules for reasoning about array sums, which Frama-C can import and make use of. In the interest of covering a slightly more interesting verification challenge and showcasing more aspects of Frama-C, we will look at a variant of summing an array based on indices. This variant was also used in the original Star Schema Benchmark on computational storage, as the team evaluated different strategies for processing database entries. The function is shown in Listing 4.2.

Listing 4.2: Function to sum an array based on supplied indices.

```
int aggregate_by_pos(
  int *data, uint32_t data_size,
  uint32_t *indices, uint32_t index_count
) {
  int result = 0;
  uint32_t *indices_end = indices + index_count;
  for (; indices != indices_end; ++indices) {
    result += data[*indices];
  }
  return result;
}
```

Before writing a specification for the function, it is necessary to define what a sum is (or import an existing definition, such as the one in Why3's `ArraySum` module[1]). For this function, we will use the definition found in Listing 4.3, which is written in ACSL. The definition states: "Given an array and a length, if the length is 0 or less, the sum of the array is 0. If the length is greater than 0, the sum of the array is equal to the last element plus the sum of the preceding elements".

Listing 4.3: Axiomatic definition of the sum of an array.

```
/*@
  axiomatic Sum_array{
    logic integer sum(int* array, integer len) reads array[0 .. (len-1)];

    axiom empty:
      \forall int* a, integer l; 0 >= l ==> sum(a,l) == 0;

    axiom range:
      \forall int* a, integer l; 0 < l ==> sum(a,l) == sum(a,l-1)+a[l-1];
  }
*/
```

[1] https://www.why3.org/stdlib/array.html

This definition is quite standard as far as regular sums go, but the function under consideration does not sum an array from start to finish. Rather, it looks up the elements to be summed based on the indices stored in a second array. The behavior of the function can be thought of as a two-step process:

1. Create a new `terms` array by mapping all indices in the `indices` array to elements in the `data` array.
2. Sum the elements of the `terms` array.

Although the function does not use this two-step process, it is still possible to write the specification of the function based on this process and then show that the function is behaviorally equivalent to the two-step process. Frama-C supports this using a feature called *ghost code*. Ghost code is program code that purely exists as part of the annotations and thus is not executed and has no effect on the behavior of the function. This makes it possible to be explicit about the logic of the function without altering it, such as declaring the `terms` array and populating it with elements from `data`. A possible specification using this approach is given in Listing 4.4.

Listing 4.4: Specification for an index-based sum function.

```
1  /*@
2    requires \valid_read(indices+(0..index_count-1));
3    requires \valid_read(data+(0..data_size-1));
4    requires \valid(terms+(0..index_count-1));
5
6    requires \separated(
7      indices+(0..index_count-1),
8      data+(0..data_size-1),
9      terms+(0..index_count-1)
10   );
11
12   requires IN_BOUNDS:
13     \forall integer idx;
14     0 <= idx < index_count ==> 0 <= indices[idx] < data_size;
15
16   assigns terms[0..index_count-1];
17
18   ensures TOTAL:
19     \forall integer idx;
20     0 <= idx < index_count ==> terms[idx] == data[indices[idx]];
21
22   ensures AGGREGATE: \result == sum(terms, index_count);
23 */
24 int aggregate_by_pos(
25   int *data, uint32_t data_size,
26   uint32_t *indices, uint32_t index_count
27 ) /*@ ghost (int terms[])*/ {
```

Lines 2-4 simply state that it must be possible to read the addresses spanned over by the arrays. Lines 6-10 require that the arrays do not overlap. The last requirement on lines 12-14 is that all the indices in `indices` fall within the bounds of `data`. Line 16 denotes what areas of memory the function will write to. Even though the function does not write to any address, the ghost code is pretending to update the `terms` array, so this has to be declared in the specification. The two cases labeled `TOTAL` and `AGGREGATE` on lines 18-22 are the expected properties of the function output.

`TOTAL` states that, after execution, `terms` will indeed contain all the elements in `data` as specified by `indices`. Finally, `AGGREGATE` states that the return value of the function is equal to the sum of `terms`. On line 27, `terms` is declared as a ghost parameter of the function, as it must be present in the function signature in order for the specification to be valid.

Writing ghost code allows for more than adding extra code to a function. It is possible to define another function written entirely in ghost code. For this example, it makes sense to have a `select` auxiliary function that looks up the correct elements of `data` and places them in `terms`. This ghost function also needs a specification, which largely matches the specification of the sum function. Both the implementation and specification of `select` is given in Listing 4.5.

Listing 4.5: Specification and implementation of a ghost function in ACSL.

```
1  /*@ ghost
2    /@ // [Low-level pointer requirements omitted]
3      requires IN_BOUNDS:
4        \forall integer idx;
5        0 <= idx < idx_len ==>
6        0 <= indices[idx] < src_len;
7
8      assigns dest[0..idx_len-1];
9
10     ensures TOTAL:
11       \forall integer idx;
12       0 <= idx < idx_len ==>
13       dest[idx] == src[indices[idx]];
14   @/
15   void select(
16     int *src, uint32_t src_len,
17     uint32_t *indices, uint32_t idx_len,
18     int \ghost * dest
19   ){/@
20       loop invariant COMPLETE:
21         \forall integer idx;
22         0 <= idx < i ==>
23         dest[idx] == src[indices[idx]];
24
25       loop assigns i, dest[0..idx_len-1];
26       loop variant idx_len - i; @/
27     for(uint32_t i = 0; i<idx_len; i++) dest[i] = src[indices[i]];
28 } */
```

To understand how to prove that `select` matches its specification, it is necessary to briefly discuss *loop invariants*. The standard approach to matching code to specification consists of analyzing the statements of the code one-by-one and determining how the program state will update during execution. Loops, however, generally execute an indeterminate amount of times, which complicates reasoning about the program state after execution. Loop invariants remedy this, as they are logical assertions that must be true before the loop starts and true after any arbitrary iteration of the loop. By using loop invariants it is possible to reason about the program state after the loop has executed, independent of the number of loop iterations. The approach mimics that of inductive reasoning, where a proof considers a base case (is the property true before loop execution?) and an inductive case (if the property is true before the nth

iteration, will it be true after the nth iteration?), but finding a usable loop invariant can be a sizable task.

Lines 23-26 in Listing 4.5 show a provable loop invariant for `select`. Using the loop variable `i` to denote the ith iteration of the loop, the invariant states that all indices of `dest` smaller than `i` contain the element of `src` as dictated by `indices`. The property clearly holds before the start of the loop since `i = 0` and we do not consider negative indices. After any loop iteration, another element of `src` is placed in `dest` and `i` is incremented by one, causing the property to hold again after any iteration as well. The loop stops when `i = idx_len` which means the loop invariant at that point exactly corresponds to the `TOTAL` property on lines 12-15, thus proving the specification of `select`.

Listing 4.6: Annotation of the body of the sum by indices function.

```
1  int result = 0;
2  uint32_t *indices_end = indices + index_count;
3  //@ ghost select(data, data_size, indices, index_count, terms);
4  //@ ghost uint32_t ghost_count = 0;
5  /*@
6    loop invariant indices + index_count - ghost_count == indices_end;
7    loop invariant 0 <= ghost_count <= index_count;
8
9    loop invariant SUM: result == sum(terms, ghost_count);
10
11   loop assigns result, indices, ghost_count;
12   loop variant index_count - ghost_count;
13 */
14 for (; indices != indices_end; ++indices) {
15   result += data[*indices];
16   //@ ghost ghost_count++;
17 }
18 return result;
```

Using the `select` function it is now possible to annotate the body of the sum function, as done in Listing 4.6. On line 3 the ghost code "calls" the `select` function to populate `terms` with the relevant elements of `data`. Since the loop on line 14 iterates by incrementing the `indices` pointer, line 4 declares a ghost counter to number the current loop iteration. The two invariants on lines 6 and 7 relate the three variables `indices`, `index_count` and `ghost_count` to help the prover determine that the loop in fact terminates since the `indices` pointer monotonically increases until it reaches `indices_end`. Normally, the prover can infer this by its own, but as this loop iterates over pointers rather than integers, this extra information is necessary to guide the prover. The invariant related to the specification is labeled `SUM` on line 9. This states that for any iteration, the value of `result` is equal to the sum of the first `ghost_count` elements of `terms`. The specification of `select` already asserts that `terms` contains the same elements in the same order as the ones `result` accumulates, so the proof goes through without requiring additional assertions, and the entire specification is verified.

4.2.2 Conversion

The original implementers of the conversion function left comments in the source code regarding assumptions they made on the functions behavior [39]. As can be seen in the comment on lines 7+8 of the `convert_dates` function in Listing 4.7, all strings to be converted are passed as one long continuous `char` array, and every 8 bytes are assumed to represent a YYYY-MM-DD formatted date.

Listing 4.7: Function to convert ACSII strings to C integers.

```
1  typedef enum { TO_YEAR, TO_MONTH, TO_DAY } CONVERSION_TYPE;
2
3  void convert_dates(
4    char *date_strings, uint32_t bytes, CONVERSION_TYPE conv, uint32_t* results
5  ){
6    // We assume a file layout of 8 chars to represent a date,
7    // e.g. 19960505, i.e. YYYY-MM-DD
8    const uint32_t TUPLE_WIDTH = 8;
9    char *data_ptr_end = date_strings + bytes;
10
11   // Assuming that branching is expensive,
12   // we only branch in the beginning and manually roll out everything else.
13   uint32_t val = 0;
14   if (conv == TO_YEAR) {
15     for (; date_strings != data_ptr_end; date_strings += TUPLE_WIDTH) {
16       val ^= val;
17       val += (date_strings[0] - 0x30) * 1000;
18       val += (date_strings[1] - 0x30) * 100;
19       val += (date_strings[2] - 0x30) * 10;
20       val += (date_strings[3] - 0x30);
21       *results = val;
22       ++results;
23     }
24   }
25   // [Similar cases for TO_MONTH and TO_DAY]
26 }
```

The assumptions on the input can be made explicit through the specification, although creating a logical assertion that a string represents a valid date of the Gregorian calendar is quite a mouthful. Instead, we simply create a predicate asserting that all characters of a string are ASCII digits, as the code still functions independently of the validity of the supplied dates. The predicate, along with a definition of string-to-int conversion, is given in Listing 4.8. Somewhat similar to the definition of sum in Listing 4.3 the base case on lines 9-11 of the conversion states that an empty string gets converted to 0. The recursive case on lines 13-17 states that, given a conversion of a string of a certain length, the conversion can be extended with one character by multiplying the conversion result by 10 and adding the next digit subtracted by the ASCII value of '0'.

With the predicate and the definition declared, it is now possible to write a specification for the conversion function. As the comment on lines 11+12 of Listing 4.7 explains, the original implementation splits the computation into three cases based on the value of the `conv` parameter. This scenario is easily expressed using WP's `behavior` keyword. This allows declaring that parts of a specification are only true when some precondition is met. In our case, the precondition is the value of `conv`, and

Listing 4.8: Predicate for checking if a string consists of ASCII digits and an definition of what it means to convert a string to integers.

```
1  /*@
2    predicate is_digits(char *string, integer length) =
3      \forall integer i; 0 <= i < length ==> '0' <= string[i] <= '9';
4
5    axiomatic Conversion{
6      logic uint32_t int_of_string(char *string, integer length);
7
8      axiom no_digit:
9        \forall char *string; int_of_string(string, 0) == 0;
10
11     axiom multi_digit:
12       \forall char *string, integer length;
13       0 <= length ==>
14       int_of_string(string, length+1) ==
15       int_of_string(string, length) * 10 + string[length] - '0';
16   }
17 */
```

Listing 4.9 shows a specification of the three different behaviors of `convert_dates`. Lines 4+5 make explicit the assumptions on the input, that all characters must fall within the ASCII range of digits, and the number of characters must be a multiple of 8 (called `TUPLE_WIDTH` in the code). The three cases on lines 9-25 describe almost the same property, that the `results` array will contain an integer conversion of the supplied strings, with the only difference among the three being the offset and length of the conversion, which corresponds to the placement of requested value in the YYYY-MM-DD format. Since this function intends to write output to memory, line 7 declares which memory addresses will be affected. A requirement from the DSL is that registered functions that write to memory must also declare the number of bytes written as an output parameter, so we have extended the function with a `res_length` variable to store the output size.

Unfortunately, even though the branching cases can be covered, the manual loop unrolling provides a challenge for the automatic solvers. Since the conversion is calculated by manually multiplying and adding the digits together, there is no place to provide a loop invariant, and without a loop invariant it becomes harder for the solvers to determine if the code follows the recursive definition of conversion. To make the verification easier, we choose to roll the loop back up and extract it to an auxiliary function. A reasonable question is whether it would not be better to change the definition of conversion to a non-recursive definition. Such a definition would require using the index of a digit to determine the magnitude of the digit's value (such as $\texttt{digit} * 10^{\texttt{index}-\texttt{length}}$), and since C has no built-in operator for exponents, we would end up relying on a loop to calculate powers anyway. Instead, we use the `parse_string` function in Listing 4.10. The specification cleanly states that for a string of non-negative length consisting of only digits, the result of the function will be the integer represented by the string.

With the necessary definitions and implementations in place, let us consider the `TO_YEAR` case of the conversion function. The case contains a single for-loop, so we can attempt to prove the specification using the loop invariants shown in Listing

Listing 4.9: Requirements and behaviors of `convert_dates`.

```
/*@
  // [Low-level pointer requirements omitted]

  requires is_digits(date_strings, bytes);
  requires bytes % TUPLE_WIDTH == 0;

  assigns *res_length, results[0..bytes/TUPLE_WIDTH-1];

  behavior to_year:
    assumes conv == TO_YEAR;
    ensures \forall integer idx;
      0 <= idx < bytes / TUPLE_WIDTH ==>
      results[idx] == int_of_string(date_strings+idx*TUPLE_WIDTH, 4);

  behavior to_month:
    assumes conv == TO_MONTH;
    ensures \forall integer idx;
      0 <= idx < bytes / TUPLE_WIDTH ==>
      results[idx] == int_of_string(date_strings+idx*TUPLE_WIDTH+4, 2);

  behavior to_day:
    assumes conv == TO_DAY;
    ensures \forall integer idx;
      0 <= idx < bytes / TUPLE_WIDTH ==>
      results[idx] == int_of_string(date_strings+idx*TUPLE_WIDTH+6, 2);
*/
void convert_dates(
  char *date_strings, uint32_t bytes,
  CONVERSION_TYPE conv,
  uint32_t* results, uint32_t *res_length
)
```

4.11. Since we require the function to update the size of the memory region it writes to, we add a result counter on line 1. To help in proving termination and keeping track of pointer positions, lines 3+4 define two ghost variables equal to the number of dates in the input. These variables are used in the invariants on lines 7-9, which are necessary for the solvers to determine the loop terminates. The invariant which proves the specification for the case is given on lines 11-14. It states that all previous loop iterations have stored the value of converting the first 4 digits of each string encountered so far in the `results` array. It should be noted that line 14 makes use of the `\at` operator of ACSL, to refer to the value of the `date_strings` pointer at the start of the function (denoted using the label `Pre`). Since the loop updates the value of `date_strings` during iteration, use of `\at` is necessary to make explicit that the invariant should be proved with respect to the original value of `date_strings` and not the current one.

Since the loop body now consists of calling the auxiliary `parse_string` function, whose specification matches the loop invariant, the solvers easily prove the invariant. However, our auxiliary function requires that any strings passed to it consist only of digits, and the solvers get stuck trying to verify if the condition is met. We know from the preconditions we imposed on `convert_dates` that `date_strings[0..bytes-1]` are all digits, but as the loop moves the `date_strings` pointer, the assertion is lost to the solvers. We can add it back in by stating the loop invariant on line 16, but the

Listing 4.10: Auxiliary function and specification for parsing a string

```
/*@
  requires 0 <= len;
  requires \valid_read(string+(0..len-1));
  requires is_digits(string, len);

  assigns \nothing;

  ensures \result == int_of_string(string, len);
*/
inline uint32_t parse_string(char *string, int len){
  uint32_t res = 0;
  /*@
    loop invariant res == int_of_string(string, i);
    loop invariant 0 <= i <= len;
    loop assigns i, res;
    loop variant len - i;
  */
  for(int i = 0; i<len; i++) res = string[i] - '0' + res * 10;
  return res;
}
```

solvers cannot verify this invariant either. To help them, we state the lemma shown in Listing 4.12.

The lemma states that any suffix of a digit-only string is itself digit-only. With this, the solvers are able to prove all of the loop invariants as well as the specification of the entire function. The verification is not done, though, as the solvers simply believe whatever is stated in the lemma to be a fact. As they lack the ability to prove the lemma on their own, the lemma is treated as an axiom. The logical assertions interspersed in the C code are suitable for automatic verification, as the solvers are guided by the semantics of the code being annotated. Lemmas on the other hand do not deal with code, but are purely logical constructs that exist based on our axioms and definitions. Instead, to prove a lemma, we must rely on interactive program verification. Fortunately, the WP plug-in has a built-in method for creating proof scripts, in order to prove more complex logical statements. This method provides a set of *tactics*, which are approaches that can be used to try and prove a property. The tactics will generate new proof obligations which must then be proved using another tactic or, in some cases when the obligation is simple, discharged to a solver. Figure 4.1 shows how this looks in practice. As the lemma is on the simpler side, the proof can be completed by highlighting the proof obligation and using the `Definition` tactic, which unfolds the definition of `is_digits` and makes it possible for the solvers to deduce that the predicate also holds for a sub-range of indices.

All of these properties together make for a verified specification of `convert_dates`. Since it did require rewriting the function to make use of our auxiliary function, it will give rise to an interesting performance evaluation regarding the trade-off between the efficiency and verifiability of code. This evaluation is carried out in Section 6.3.

Listing 4.11: The loop extracting the years from a date string, along with loop invariants.

```
1   uint32_t num_results = 0;
2   char *data_ptr_end = date_strings + bytes;
3   //@ ghost int entries = bytes / TUPLE_WIDTH;
4   //@ ghost int remaining = bytes / TUPLE_WIDTH;
5   if (conv == TO_YEAR) {
6     /*@
7       loop invariant remaining >= 0;
8       loop invariant num_results + remaining == entries;
9       loop invariant date_strings + remaining * TUPLE_WIDTH == data_ptr_end;
10
11      loop invariant
12        \forall integer i;
13        0 <= i < num_results ==>
14        results[i] == int_of_string(\at(date_strings, Pre)+i*TUPLE_WIDTH, 4);
15
16      loop invariant is_digits(date_strings, remaining * TUPLE_WIDTH);
17
18      loop assigns remaining, results[0..entries-1], num_results, date_strings;
19
20      loop variant remaining;
21    */
22    for (; date_strings != data_ptr_end; date_strings += TUPLE_WIDTH) {
23      results[num_results++] = parse_string(date_strings, 4);
24      //@ ghost remaining--;
25    }
26  } else if (conv == TO_MONTH) {
```

Listing 4.12: Lemma stating that if a string contains only digits, any suffix of the string is also only digits.

```
1  /*@
2    lemma substring_digits:
3      \forall char* string, integer length;
4      is_digits(string, length) ==>
5      \forall integer i; 0 <= i < length ==>
6      is_digits(string+i, length-i);
7  */
```

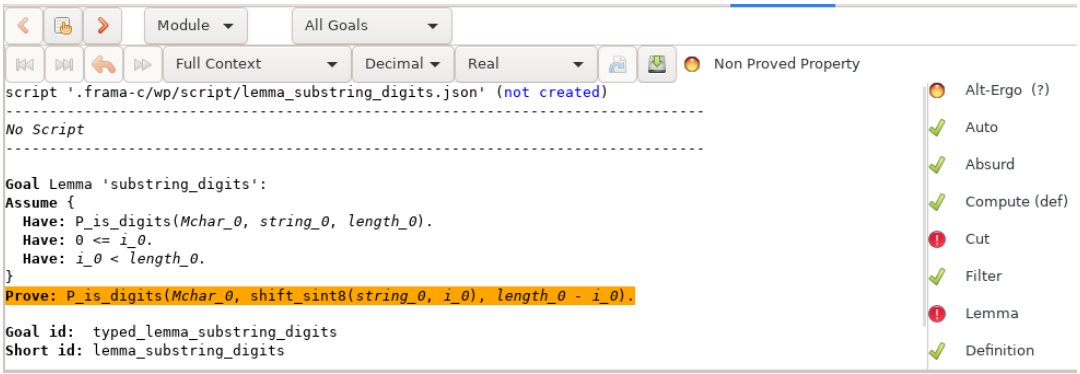


Figure 4.1: WP's script generator working on the `substring_digits` lemma. The proof obligation is highlighted in orange, and available tactics are listed on the right.

4.2.3 Filter

The filtering function shown in Listing 4.13 is the last example that we will verify. Given a lower and upper bound, the function will filter out all elements of `data` that do not fall within the bound.

Listing 4.13: Filter function implemented in C for use in the Delilah research project.

```
1  uint32_t delilah_functions_tsl_filter_sequential(
2    uint32_t* data, size_t len,
3    uint32_t* out,
4    uint32_t _c_pred1, uint32_t _c_pred2
5  ) {
6    size_t curr_idx = 0;
7    for (size_t i = 0; i < len; i++) {
8      const uint32_t inc = (data[i] >= _c_pred1) && (data[i] <= _c_pred2);
9      out[curr_idx] = i;
10     curr_idx += inc;
11   }
12   return curr_idx;
13 }
```

To define the specification of the filter function, it is necessary to express some properties about arrays. Stating that elements fall within a certain range is simply enough, but we need to express that all elements in the `out` array come from the `data` array. To achieve this, we axiomatically define a property to count the number of occurrences of a specific element in an array in Listing 4.14. Any value occurs 0 times in an empty array, as stated in lines 6+7. If the last element in an array is the value being counted, the occurrence count is 1 plus the number of occurrences in the rest of the array, per lines 9-13. Lastly, lines 15-19 state that if the last element in an array is *not* equal to the value being counted, the number of occurrences is the same if we exclude the last element.

Listing 4.14: Axiomatic definition of the number of occurrences in an array.

```
1  /*@
2    axiomatic Occurrences{
3      logic integer occurrences_in{L1}
4          (uint32_t* arr, integer len, uint32_t value) reads arr[0..len-1];
5
6      axiom empty: \forall uint32_t* arr, uint32_t v, integer len;
7        len <= 0 ==> occurrences_in(arr, len, v) == 0;
8
9      axiom in_nonempty:
10       \forall uint32_t* arr, integer len, uint32_t v; len > 0 ==>
11         (arr[len-1] == v <==>
12         occurrences_in(arr, len, v) ==
13         1 + occurrences_in(arr, len-1, v));
14
15     axiom notin_nonempty:
16       \forall uint32_t* arr, integer len, uint32_t v;
17       len > 0 ==>
18         (arr[len-1] != v <==>
19         occurrences_in(arr, len, v) == occurrences_in(arr, len-1, v));
20   } */
```

Using this new definition, the specification of the filter function is declared in Listing 4.15. Like in the previous example, we have to express some practical requirements on the pointers and add a parameter to describe the length of the output array. The two properties regarding the output of the filter function are the ones on lines 8-16 labeled `Correct` and `Complete`:

- `Correct`: All values in the output array are within the specified bounds.
- `Complete`: For all values within the bounds, exactly the same number occurs in both the data array and the output array.

Listing 4.15: Pre- and Post-conditions for the filter function.

```
1  /*@
2    requires \valid(out+(0..len-1)) && \valid(written);
3    requires \valid_read(data + (0..len-1));
4    requires \separated(written, data+(0..len-1), out+(0..len-1));
5
6    assigns *written, out[0..len-1];
7
8    ensures Correct:
9      \forall size_t j;
10     0 <= j < *written ==> _c_pred1 <= out[j] <= _c_pred2;
11
12   ensures Complete:
13     \forall uint32_t value;
14     _c_pred1 <= value <= _c_pred2 ==>
15       occurrences_in(data, len, value) ==
16       occurrences_in(out, *written, value);
17 */
18 void
19 delilah_functions_tsl_filter_sequential(uint32_t* data, size_t len,
20                                         uint32_t* out, size_t* written,
21                                         uint32_t _c_pred1, uint32_t _c_pred2)
```

To prove that the loop of the filter function produces the required output, some extra definitions are required in order to establish suitable loop invariants. The loop functions by looking at elements one by one and doing one of two things: If the element is within the requested range, it stores it in the output array and increments the current index. If not, it still stores the element in the array, but does not increment the index. This implementation leads to situations where a loop iteration might overwrite the work of a previous iteration, which makes it difficult to prove a loop invariant with respect to all prior indices. To navigate this challenge, we add the predicate and lemma shown in Listing 4.16. The `unchanged` predicate on lines 2+3 states that an array is unchanged between two different program points `L1` and `L2` if all elements in the array have the same value at both program points. We use this predicate to state on lines 5-8 that an unchanged array will have the same number of occurrences of any value at both program points.

As with the conversion example, trying to prove the lemma using the automatic solvers Qed, Alt-Ergo and Z3 results in them timing out with no proof generated. The lemma is too complex for solvers to prove without assistance, so we again have to switch to an interactive proving method. As we already covered WP's interactive proof script generation in the previous example, this example will showcase another

Listing 4.16: Array lemma written in ACSL, to be used in the verification of the filter function.

```
/*@
  predicate unchanged{L1, L2}(uint32_t* array, integer len) =
    \forall integer i; 0 <= i < len ==> \at(array[i], L1) == \at(array[i], L2);

  lemma unchanged_occurrences{L1, L2}:
    \forall uint32_t *arr, value, integer len;
    unchanged{L1, L2}(arr, len) ==>
    occurrences_in{L1}(arr, len, value) == occurrences_in{L2}(arr, len, value);
*/
```

approach. The WP plug-in supports the dispatching of proofs to proof assistants such as Rocq [16], by using the Why3 verification platform [20]. For each property that we wish to prove interactively, this will generate a Rocq file with extra definitions and constraints to deal with C-types. Listing 4.17 shows the obligation we have to prove in Rocq in order for `unchanged_occurrences` to be verified.

Listing 4.17: Type of the proof required to show `unchanged_occurrences`

```
Theorem wp_goal :
  forall (L1:addr -> Numbers.BinNums.Z) (L2:addr -> Numbers.BinNums.Z)
    (arr:addr) (len:Numbers.BinNums.Z) (val:Numbers.BinNums.Z),
  is_uint32 val -> P_unchanged L1 L2 arr len ->
  ((L_occurrences_in L2 arr len val) = (L_occurrences_in L1 arr len val)).
```

We will not look at the entire proof script, which can be found in Appendix B, but instead sketch out how to prove this property. At a high level, the lemma shows that a predicate (`unchanged`) on an array implies another (`occurrences_in`). We can prove this by considering a specific value and proceeding with induction on the length of the array and applying the axioms of Listing 4.14.

- In the base case where the length is 0, we can use the `empty` axiom to conclude that there are 0 occurrences of any value at both program points.
- In the inductive case, we can assume that the length is positive. Technically, as we are doing induction on integers, there is also an inductive case for negative lengths, but that case is handled the same way as the base case. Given the specific value, we know that the number of occurrences of the value between index 0 and `len-1` is the same at both program points `L1` and `L2`. We must now show that the occurrences are the same when the array has length `len+1`. We do this by splitting the proof further into two cases: one where `arr[len] == val` and one where `arr[len] != val`. We know from the `unchanged` predicate that `arr[len]` has the same value in both `L1` and `L2`.
  - `arr[len]==val`: In this case, we can use the axiom `in_nonempty` to conclude that at program point `L1` the occurrences in the array with `len+1` elements must be the same as 1 plus the occurrences when there is `len` elements. The same can be done for program point `L2`. Since both sides of the equality contains a +1, we just have to show that the occurrences

in the array with length `len` is the same at both `L1` and `L2`, which is the induction hypothesis.

– `arr[len]!=value`: In this case, the `notin_nonempty` axiom allows us to rewrite the occurrences from using `len+1` to using `len`. The induction hypothesis then shows that the occurrences are equal at both program points.

While the proof is conceptually straightforward, proving it in Rocq is another matter. The extra constraints generated by Why3 require us to prove that the values in the array can be expressed using unsigned 32-bit integers. Since we deal with an array of type `uint32_t*`, this should be evident, but our choice in how we define `occurrences_in` prevents us from using this fact. Since `occurrences_in` is defined through axioms, it is not an actual function and is thus never run. As it is never run, we have no prior knowledge that the counted elements can be expressed as 32-bit integers. We can of course add another axiom stating that if `occurrences_in` produces a result, then all elements in the counted array must be of size `uint32_t` (Listing 4.18).

Listing 4.18: Axiom stating that all elements in a counted array fall within the range of an unsigned 32-bit integer.

```
1 /*@
2     axiom occurrence_size:
3       \forall uint32_t* arr, integer len, uint32_t v; len > 0 ==>
4       (\exists integer k; k == occurrences_in(arr, len, v)) ==>
5       \forall integer i; 0 <= i < len ==> 0 <= arr[i] <= UINT32_MAX;
6 */
```

With the lemma taken care of, we can move on to defining loop invariants for the body of the filter function. This is done in Listing 4.19. The loop is changed to rely on the `written` pointer to keep track of the current index, since the function needs to update `written` with the total amount of data written anyway. By bounding the value stored in `written` using the loop invariants on lines 3+4, we can put a bound on the locations written to using the `assigns` clause on line 15. Even though the loop might not update any values in the `write` array, the `assigns` clause cares about a tight upper bound. The maximum amount of data the loop will write happens if all elements in the `data` array lie within the specified range, in which case the locations `out[0..len-1]` will be written to, so this becomes the upper bound. Lastly, the two invariants on lines 6-8 and 10-13 are written to closely match the `Correct` and `Complete` properties of the function specification.

While these invariants are provable and match our post-conditions, the automatic solvers still struggle to prove them. Earlier in this section, we described the loop as doing one of two things, depending on whether the current element is within the range or not. Just as it made intuitive sense to explain the loop body based on whether the current element is within the range or not, the verification becomes simpler by considering those two cases. We can split the verification into cases by writing a conditional in ghost code. In these cases, we can add manual assertions to help guide the solvers. The goal is to add enough properties for the solvers to

Listing 4.19: Loop invariants for the implementation of filter.

```
1  *written = 0;
2  /*@
3    loop invariant 0 <= i <= len;
4    loop invariant 0 <= *written <= i;
5    loop invariant
6      \forall size_t j;
7      0 <= j < *written ==> _c_pred1 <= out[j] <= _c_pred2;
8
9    loop invariant
10     \forall uint32_t value;
11     _c_pred1 <= value <= _c_pred2 ==>
12     occurrences_in(data, i, value) == occurrences_in(out, *written, value);
13
14   loop assigns i, *written, out[0..len-1];
15
16   loop variant len-i;
17 */
18 for (size_t i = 0; i < len; i++) {
19   const uint32_t inc = (data[i] >= _c_pred1) && (data[i] <= _c_pred2);
20   out[*written] = data[i];
21   *written += inc;
22   // assertions written in Listing 4.20
23 }
```

make use of the lemma we just defined earlier. This approach of guiding solvers into triggering lemmas is known as auto-active verification, a portmanteau of automatic and interactive verification [19].

Listing 4.20: Assertions used to prove the loop invariants of the filter function.

```
1    //@ assert unchanged{Here, LoopCurrent}(data, i);
2    /*@ ghost
3      if((data[i] >= _c_pred1) && (data[i] <= _c_pred2)){
4        //@ assert unchanged{Here, LoopCurrent}(out, *written-1);
5        /@ assert \forall uint32_t value; _c_pred1 <= value <= _c_pred2 ==>
6            occurrences_in(data, i, value) ==
7            occurrences_in(out, *written-1, value); @/
8
9      } else {
10       //@ assert unchanged{Here, LoopCurrent}(out, *written);
11       /@ assert \forall uint32_t value; _c_pred1 <= value <= _c_pred2 ==>
12           occurrences_in(data, i, value) ==
13           occurrences_in(out, *written, value); @/
14     } */
```

Listing 4.20 shows how we achieve this goal. As our loop invariant pertains to the occurrences of values in `data` and `out`, we achieve the greatest chance of triggering the `unchanged_occurrences` lemma by asserting that parts of `data` and `out` remain unchanged. Line 1 asserts this about `data`, and lines 4+10 make the assertion about `out`, depending on which branch of the ghost code we are in. After these assertions, on lines 5-7 and 11-13 we ask the solvers to assert properties matching our loop invariant. Since we have asserted that both `data` and `out` are unchanged up to a point, and we have a lemma implying equality of occurrences between unchanged arrays, the

solvers easily verify that the invariant holds, which concludes the verification of the filter function.

4.3 Discussion

4.3.1 Axiomatic and Inductive Definitions

In the filtering example, we added an axiom to deal with the fact that logic functions do not have knowledge about the word size of the integers they describe. Our other option was to define `occurrences_in` as an inductive property rather than a logic function. In Listing 4.18 the three axioms regarding occurrences have been converted into cases for an inductive property. With this approach, once we have a proof about the occurrences of a value in an array, we also get a guarantee that the values of the array can be expressed as 32-bit integers.

Listing 4.21: Inductive definition of occurrences.

```
/*@
inductive occurrences(uint32_t* arr, integer len,
                      uint32_t value, integer count){
  case empty:
  \forall uint32_t* arr, integer start, integer len, uint32_t value;
    start >= end ==> occurrences(arr, len, value, 0);

  case in_nonempty:
  \forall uint32_t* arr, integer len, uint32_t value, integer count;
    occurrences(arr, len, value, count) ==>
    arr[end] == value ==>
    occurrences(arr, len+1, value, count+1);

  case notin_nonempty:
  \forall uint32_t* arr, integer len, uint32_t value, integer count;
    occurrences(arr, len, value, count) ==>
    arr[end] != value ==>
    occurrences(arr, len+1, value, count);
}*/
```

Using an inductive property also makes the manual proving in Rocq easier, but changing the specification to rely on a property instead of a "counting"-function challenges the automatic solvers ability to prove the specification. It is possible to combine both approaches by stating a lemma showing that all values produced by the axiomatic definition also satisfy the inductive property, and then define the specification in terms of the axiomatic definition while proving the lemmas using the inductive definition. In our example, we relied on the purely axiomatic approach to lessen the workload, but adding axioms carries the risk of introducing unsound assumptions. In some scenarios it may be better to minimize the usage of axioms, in order to achieve higher levels of trust in the proof.

4.3.2 Wrapping Registered Functions

As covered in the start of this chapter, registered functions need to make explicit the amount of data they write, potentially requiring extra size parameters in the function signature. In the Rogers DSL (Chapter 3) regions of data are represented

using a `mem_range` C-`struct`, consisting of a byte pointer and length variable denoting the number of bytes of data. Another option for clarifying data sizes in registered functions is to make use of the same `mem_range` abstraction as the DSL, as shown in Listing 4.22.

Listing 4.22: Two possibilities for a modified function signature for filtering.

```
1 void filter_sequential_1(uint32_t* read, size_t read_size,
2                          uint32_t* write, size_t* write_size,
3                          uint32_t _c_pred1, uint32_t _c_pred2)
4
5 struct mem_range {char* ptr; uint32_t length; };
6 void filter_sequential_2(struct mem_range read,
7                          struct mem_range *write,
8                          uint32_t _c_pred1, uint32_t _c_pred2)
```

The second choice makes it clearer what pointers and lengths belong together, but it also has the unfortunate consequence of requiring casts to match the semantics of the original function. Memory regions need to be able to describe any region of data, so the pointer and length describe individual bytes. Most functions, however, work with specifically sized data, such as the sum and filter functions that work with 32-bit integers. Functions could be generalized to work with arbitrarily sized elements, similar to `qsort()` in the standard library of C.[2] Although not conceptually difficult, it makes both writing and verifying a specification quite messy.

The choice in this chapter was to extend functions with size parameters where necessary. This makes the specification more straight-forward, but also requires a wrapper function to act as an interface between the byte-based regions used by the DSL and the potentially word-based function implementations. We show an example of such a wrapper-function in Listing 4.23.

Listing 4.23: A Possible wrapper function to interface memory regions with function implementations.

```
1  void wrapper(struct mem_range read,
2               struct mem_range *write,
3               uint32_t _c_pred1, uint32_t _c_pred2)
4  {
5      write->length = write->length / sizeof(uint32_t);
6      filter_sequential_1((uint32_t *)read.ptr, read.length / sizeof(uint32_t),
7                          (uint32_t *)write->ptr, &(write->length),
8                          _c_pred1, _c_pred2);
9      write->length = write->length * sizeof(uint32_t);
10 }
```

These wrapper functions are not formally verified, but the code is always limited to casting points between bytes and element types, and scaling the region lengths accordingly. This leads to a rather small trusted computing base that is easily verifiable for a human.

[2] https://cplusplus.com/reference/cstdlib/qsort/

4.3.3 Interfacing Verifications

This chapter showed a verification process for functions written in C. Other C-functions can be verified using the same method, but not all functions on a CSD will necessarily be written in C. Many other implementation methods also have a possible verification method, but a question arises as to whether it makes sense to combine different types of verification. Is a user expected to be familiar with an entire toolbox of verification tools in order to understand the specification of the functions on a device? This may be something we can expect from a registrant, but an ordinary programmer will likely not have the necessary verification experience to understand all the possible ways to express functional behavior.

Rather, we need a unifying interface to reduce the amount of material a programmer has to learn. This could also serve to minimize the trusted computing base, as we do not need to trust all the individual verification tools. This verification interface already exists in the form of intermediate verification languages such as Boogie, WhyML, and Viper [57, 33, 66]. To understand what role intermediate verification languages play in program verification, consider Figure 4.2. We have already discussed in Section 2.4 how specification languages can be used to express specifications of programs. Intermediate verification languages (IVLs) serve as an interface between verification tools and SMT (satisfiability modulo theories) solvers. A verification tool can convert its specification language to an IVL and then rely on solvers supported by that IVL. Solvers often rely on an input language different from the IVL, so the IVL must again be converted to a language supported by the specific solver. As a concrete example, for the functions verified in this chapter, we have used C as the programming language and annotated the functions using ACSL. The annotations have been converted to WhyML by Frama-C, which in turn has been converted by Why3 to the SMT-LIB language used by the Alt-Ergo solver.

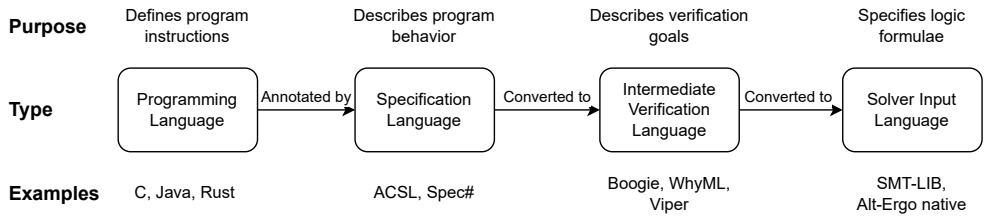


FIGURE 4.2: The different types of languages used in formal verification of software.

If an IVL is chosen to serve as a bridge between a programmer and the different specification language, it would mean that a programmer would only need to familiarize themselves with the particular IVL in order to understand the specifications of registered functions on a given CSD. The limitation of this approach lies in which front-ends the different IVLs support. There is no "universal" IVL, so relying on a single IVL limits what implementation methods are available for registered functions. The Boogie IVL was developed alongside Spec# [11], a compiler for implementing and specifying C# programs, but other programming languages such as Dafny [56] and Move [17] also support exporting verification conditions to the Boogie language.

As shown, Why3 and WhyML support C programs through Frama-C, but also Java through Krakatoa and a variant of Ada through Spark2014 [32, 5]. Lastly, Viper supports Java, Rust, Python, and Go [31]. The languages mentioned here are not an exhaustive list of supported languages for each IVL, but serve to show that each IVL focuses on different sets of languages to support. Ameri & Furia described a translation of Boogie into WhyML [8], so it is also possible to unify the front-ends supported by different IVLs by defining a conversion between IVLs. This carries the drawback of increasing the size of the trusted computing base (TCB), but the results from Ameri & Furia showed that Why3 was still able to re-verify 83% of the translated Boogie programs, indicating that it may be possible to combine multiple IVLs without expanding the TCB.

As an alternative to interfacing verification tools, it may be suitable to simply "interface the specifications" by defining a common human-readable specification language. This specification language would not need to be converted to actual IVL, but rather serve as documentation for a programmer to convey the properties of a CSF. This is not an unheard of approach, as the researchers developing Wasm SpecTec [22] have similarly proposed to automatically generate formal specifications in LaTeXbased on the implementation of their DSL. It should be noted that the suggestion in the context of computational storage is not to automatically generate a specification based on implementation, but rather to define a target specification language (expressed in LaTeXterms) into which other handwritten specifications can be translated.

4.3.4 Hardware Accelerators

The verification of registered functions covered in this chapter deal with operations implemented in software, but recall that Figure 2.6 from the introduction also mentions hardware implemented functions. Such functions would be hardware accelerators like FPGAs that deal with, e.g., compression or encryption, as shown by [55]. For this purpose, it would be necessary to use a *hardware description language* (HDL) such as Verilog or its extension SystemVerilog [44]. The discussion around HDLs largely mirrors the points brought forward in the prior section (4.3.3). The hardware parallel to IVLs is the Property Specification Language [43], which is a temporal logic compatible with some HDLs such as SystemVerilog. However, rather than serving as an intermediate language, PSL acts as a higher-level language which builds on top of one of the compatible HDLs. Properties written in PSL can then either be tested through simulation, or a subset of PSL can be compiled to a temporal logic formula and delegated to a theorem prover.

As hardware specifications are written using temporal logic to account for hardware cycles, and software specifications usually are written in nontemporal first-order logic, it is not clear how one would go about interfacing specifications between hardware and software. The choice would most likely to have one type of verification interface for software (à la Boogie or WhyML), and another for hardware (PSL), with no interfacing between the two types. This would leave programmers with only two types of specification to familiarize themselves with.

4.3.5 Type Signatures of Registered Functions

Section 3.3.1 highlighted the need for a method of determining bounds on the amount of data a function will write, in order to correctly type the function for the Rogers DSL. As the examples found in this chapter shows, verification tools provide such a method. For Frama-C in particular, we are able to annotate functions with the locations the function may update.

The idea is that given these annotations, we can convert them into an enriched type signature of the function and store it in on the CSD alongside the specification. This gives us the function mapping ϕ used for the type checking discussed in Section 3.3.2. The annotations can be used to type arguments as follows:

`requires \valid_read(n.ptr + (0 .. n.length-1))` $\Rightarrow n :$ `read`
`requires \valid(n->ptr + (0 ..` e`-1))` $\Rightarrow n :$ `write replace`(e)

A region that requires all data between its start and length to be readable must be a `read` region. A region that requires data between its start and some expression e to be writable must be a `write` e region. As stated in Section 3.3.1, the expressions stored in a `write` e type must use absolute references to the position of function arguments, so variable references in the expression must be replaced as part of the signature enrichment.

If the verification is performed on functions using memory regions directly, this enrichment of the type signature can be done automatically. Of course, as mentioned in Section 4.3.2, we may end up using wrapper functions instead, in which case the annotations must be converted manually.

The annotations shown are specific to Frama-C, but annotating writable data locations is possible in many other verification tools. At the beginning of this chapter, Why3 and Verifiable C were discussed as alternative tools for the case study. Why3's micro-C supports using arrays as parameters, and annotations can make assertions regarding the length of those arrays. Verifiable C has a `data_at` predicate to assert that some location contains a value of a specific type and permission. The type can be a `tarray` t n, indicating an array of n elements of type t. The permission can be used to infer whether the array is read-only or writable. Outside of verification of C programs, languages such as Spark, WhyML and Dafny all support reasoning about arrays, their lengths, and their modifiability.

4.4 Conclusion

In this chapter, we have verified existing computational storage functions, thereby creating the first registered functions for use in compositional storage devices. The functions, written in C, are verified by annotating them with ACSL [15], Frama-C's [26] annotation language, and then using the WP plug-in [14] of Frama-C to convert the annotations to WhyML [33]. The annotations are then formally proven using a mix of automatic verification through the SMT solver Alt-Ergo [2], and manual verification in the form of the Rocq theorem prover [16].

Verification of the functions shows not only how creators of computational storage devices can supply trustworthy computations, but also how verification tools can be used to estimate the space usage of computations, a key aspect in determining the efficiency of computational storage. This space estimation can be used together with the Rogers DSL from the previous chapter to provide enriched type signatures for functions, enabling the Rogers validator to provide its guarantees on memory safety.

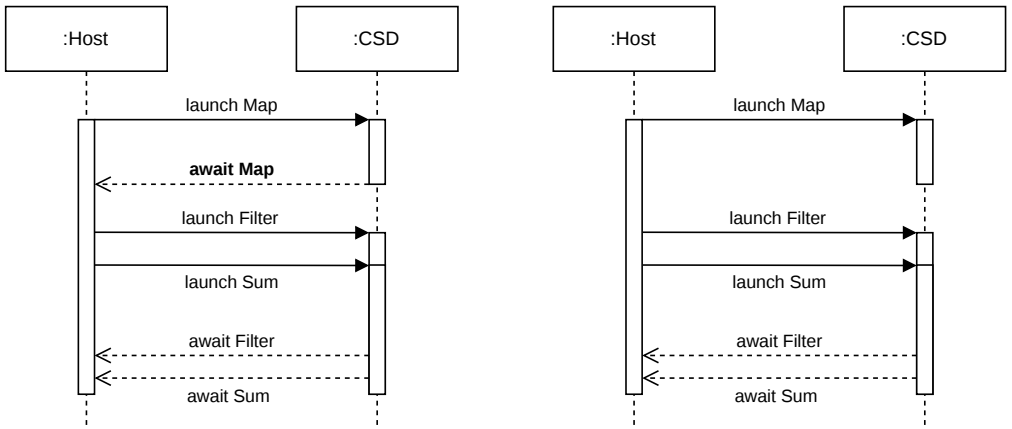
Chapter 5

Coordinating Resources

The purpose of the resource manager is to help programmers manage concurrent dependencies between their CSFs. When talking about CSFs dependencies, we consider two types: *explicit* and *implicit* dependencies. Explicit dependencies occur when the results of a CSF is passed as parameters to another CSF. Such dependencies are clear to see from the host side, as it does not make sense to start a CSF before the input parameters for the CSF have been produced. Implicit dependencies are when multiple CSFs access the same part of shared memory. Such dependencies are harder to spot on the host side. The current approach requires the programmer to manually keep track of these dependencies and insert synchronization barriers where appropriate.

Consider the following scenario: We want to map some data from one data type to another, followed by summing up the result. At the same time, we also want to filter the mapped data. Rather than performing the mapping twice, we want to reuse the mapped results by storing them in shared memory, and then launching two CSFs with read access to the mapped data. In an architecture without the resource manager, the programmer must define a blocking synchronization point in the host program to await the results of the mapping CSF before launching the sum and filter CSF. These interactions between the host and the CSD are shown as a sequence diagram in Figure 5.1a with the synchronization point for the implicit dependency highlighted in bold.

If the programmer forgets to insert the correct synchronization barriers, debugging the host program becomes very challenging, as the results of the CSFs will be affected by race conditions on the CSD. By associating regions of shared memory with IDs, in combination with the typing from the Rogers DSL, a resource manager can automatically detect implicit dependencies between CSFs. By detecting dependencies between CSFs and queuing their requests to shared regions, we can achieve the sequence of Figure 5.1b, while still avoiding common concurrency conflicts. We define these conflicts more precisely in the next section.



(A) Existing sequence for handling implicit dependencies between CSFs.

(B) New sequence achievable by having a resource manager queue CSFs.

FIGURE 5.1: Interactions between the host program and the CSD when executing CSFs that share resources.

5.1 Requirements

As it is possible for multiple CSFs to run concurrently, all operating on the same shared memory space, we need some way to ensure that program executions will not interfere with each other. To this end, CSFs are restricted to only work on the regions of shared memory to which they have been granted explicit access. Regions in shared memory are defined the same way as in local memory. A single shared region is a contiguous array of bytes in shared memory defined by the address of the first byte as well as the number of total bytes (the region length).

In order for the CSF programmer to focus on how a CSF should transform data, and not have to worry about concurrency, we want to provide the following guarantees:

1. A CSF is guaranteed to eventually gain access to any region of shared memory used in the CSFs computation.
2. Once a CSF has obtained access to a region of shared memory, the contents of that region will not change outside operations performed by the CSF itself.
3. A CSF that has started execution will not be aborted due to concurrency conflicts.

guarantees (1) and (3) also mean that we promise no deadlocks, as this would require aborting a CSF to guarantee access.

To achieve these guarantees, we utilize a queue system to impose a total ordering on requests. With the request ordered, the resource manager can scan the queue for any conflicting requests and either grant the request or delay the execution of the requesting CSF. From the program analysis in Section 3.4.1 we can deduce the necessary permissions for a CSF before execution. We name these permissions the request-set of a CSF. Knowing the request set in advance makes it so that each CSF can request all necessary regions at once before executing, known as Conservative

Table 5.1: Overview of the number of available cores in prior computational storage projects.

Project	Compute Resources
GraphSSD [64]	2 x ARM Cortex-A9 processor @ 1GHz
Delilah [39]	4 x ARM Cortex-A53 @ 1.5GHz
Willow [74]	8 x RISC microprocessor @ 125MHz
Biscuit [38]	2 x ARM Cortex-R7 @ 750 MHz

Two-Phase Locking (or C2PL). Combining C2PL and ordered locking is enough to provide us with the above-mentioned guarantees. We will analyze this claim further in Section 5.4.

Using a centralized queue to manage multiple CSFs may sound like it will lead to a bottleneck in execution, but CSDs are not highly concurrent environments. Section 2.5 discussed how the Eid-Hermes protocol associates a single CSF per core. Table 5.1 summarizes the number of cores different computational storage projects had available, showing that in most cases only a handful of CSFs are able to execute at once on the same CSD. Other projects such as Biscuit use a different approach to associating cores and CSFs. In Biscuit, a core can perform cooperative multi-threading to execute multiple SSDlets (Biscuits term for CSFs) concurrently. However, a host program only gets assigned one core to ensure that all SSDlets from a program are executed on the same core. This in turn again limits how many CSFs the host program is able to launch concurrently.

Using queues is not a new approach when it comes to computational storage. All projects listed in Table 5.1 also utilize queues for enqueuing commands between the host and the CSD. However, these queues are used to support asynchronous calls by the host program and not to handle concurrency conflicts between calls. The approach with the most similarities is the GraphSSD, which in addition to queues also employs a cache manager. The result of calls to the GraphSSD is stored in a host-side cache, so if the host program tries to access the same graph data multiple times, the cached data are returned instead of repeating the original computation. The difference between the GraphSSD's cache manager and what we aim to achieve is giving the programmer control over what is placed in the cache (shared memory). The novelty of our approach lies in analyzing computations while they are in the queue. This gives the programmer the power to decide which data to cache without needing to manually synchronize between the implicit dependencies that arise when accessing cached data.

5.2 Requesting Regions

Shared regions are only ever created by the resource manager, and regions are distinguished by assigning each a unique ID. We will not discuss a specific allocation strategy as the specific implementation for dynamic allocation does not affect how the resource manager operates. The only thing that matters for allocation is that the

resource manager also associates an address and a length with the region ID after allocation.

As laid out by the syntax of the Rogers DSL in Section 3.2, a request for a single region consists of a tuple (id, p), which requests access to region id with permission p (either read or write). To *obtain* a region is to acquire the address and length values of the region, which is information that only the resource manager provides.

Since we utilize C2PL, all requests must be made at the same time. A CSF that needs multiple regions will then send a set of request-tuples to the resource manager through the `obtain` call described in Listing 3.1. We refer to this set as the *request set* of a CSF. In addition to the request set, the resource manager also requires two pointers from each `obtain` call: one to signify where to write the address and length values of the requested regions and a second to signal to the calling CSF when the acquisition is completed. We will refer to these two pointers as *res* and *flag*, respectively. The request set *reqs* along with the two pointers make up the obtain triplet $(reqs, res, flag)$ of a CSF.

Once the resource manager starts processing an `obtain` call, it will assign a logical timestamp to the obtain triplet. It will then compare the requested IDs + permissions to its own view of which other requests are already trying to access the regions associated with the IDs. For this purpose, the resource manager maintains a queue of ongoing / waiting requests for each ID, sorted by the timestamp of the request. These lists are named *future lists*, as they identify the future actions of each region. An incoming request is always added to the end of the future list for each region it wants to access. At some point in the future, when the request has been granted and is ready to release its regions, the request is removed from the future list again. Read requests can be granted if the future list for the region does not contain a pending write request with a lower timestamp. Write requests can be granted if no request has a lower timestamp in the future list.

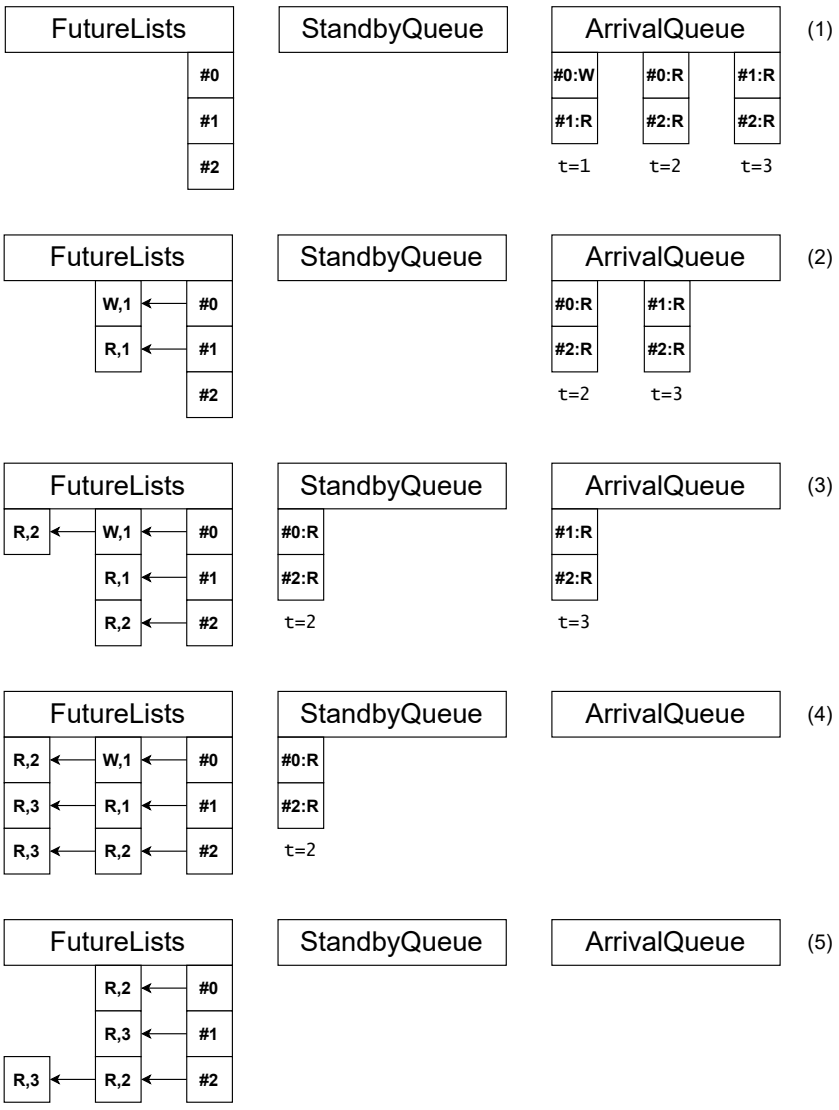


FIGURE 5.2: 5 steps showcasing how requests are processed

Figure 5.2 shows an example of how this would work in practice. For this example, we assume only three accessible regions with IDs 0, 1, and 2. For simplicity, we ignore the pointers in the example and only show the request sets and timestamps. Starting at step (1), we have three obtain triplets in the arrival queue:

- Request for write access to ID 0 and read access to ID 1 at timestamp 1
- Request for read access to ID 0 and 2 at timestamp 2
- Request for read access to ID 1 and 2 at timestamp 3

The resource manager starts by de-queuing the triplet with timestamp 1. Since all relevant future lists are empty, the triplet can immediately be granted. The future lists of IDs 0 and 1 are still updated.

In step (2), the triplet with timestamp 2 cannot be granted, since the future list of ID 0 contains a write access with a lower timestamp ($1 < 2$). The triplet is instead moved to the standby queue, and the future lists of IDs 0 and 2 are updated.

In step (3), the triplet with timestamp 3 can be granted immediately since the future lists of ID 1 and 2 contain no writes with a lower timestamp. Again, the future lists of 1 and 2 are updated.

In step (4), the triplet with timestamp 1 is finished and releases the regions that it had acquired. This is done simply by removing the first element of the future list for each ID in the request set of the triplet. The standby queue is then iterated to see if any waiting requests can be granted due to the change in future lists.

Finally, step (5) shows the resulting state following steps 1-4. With the removal of the first element in future lists 0 and 1, nothing conflicts with the triplet in the standby queue, and thus the triplet is granted access and removed from the queue.

5.3 Conflict Analysis

As the example in Figure 5.2 shows, the resource manager needs to maintain an arrival queue *ArrQue*, a standby queue *StbQue*, a key-value table of IDs and *FutureLists*, and a *Time* counter. Furthermore, the resource manager also needs a *Releases* queue for released regions so CSFs can asynchronously release acquired regions. Finally, as mentioned earlier, the resource manager must maintain a key-value table of *Regions*, in order to correlate region IDs with addresses and lengths. On startup, all collections will simply be empty, and *Time* will be set to 1.

When CSFs want to obtain or release regions, they add the necessary information to the relevant queue, and let the resource manager handle the event when it is ready. For obtaining regions, the information is the obtain triplet, while releasing regions only requires the request set. Since CSFs do not need to wait for a response from the resource manager when releasing regions, pointers to provide return values are unnecessary for the release process.

```
procedure OnRequest(reqs, res, flag)
    Enqueue(ArrQueue, (reqs, res, flag))
end procedure
```

```
procedure OnRelease(reqs)
    Enqueue(Releases, reqs)
end procedure
```

One thing to note is that since multiple CSFs can try to enqueue requests or releases at the same time, the *ArrQueue* and *Releases* queues must be thread-safe. The same is true for the mappings *Regions* and *FutureLists*, since multiple CSFs can try to allocate new regions in shared memory at the same time. The other structures as well as the individual future lists are only accessed by the main loop of the resource manager, which runs single-threaded and, therefore, do not need to be thread-safe.

The queuing can be skipped when allocating new regions in shared memory, as no other process other than the calling one knows about the new section. We show this

in alg. 1. The calling process can immediately be granted write permission to the newly allocated regions. To reflect this in the future list of the new region, we insert a write-access with timestamp 0, which is guaranteed to be lower than any other timestamp, as the global time starts at 1. In addition to the $\{(Write, 0)\}$ singleton list, future lists also maintain a number corresponding to the lowest timestamp of any write in the list. This is done to make the check for conflicts much faster. We refer to the lowest timestamp of a write as *nextWrite*.

Algorithm 1 Creating new regions

```
1: procedure ALLOCATE(length)
2:     id ← FRESHNAME()
3:     addr ← MALLOC(length)
4:     Regions[id] ← (addr, length)
5:     FutureLists[id] ← (0, {(Write, 0)})
6:     return (id, (addr, length))
7: end procedure
```

To look at how the *nextWrite* value makes conflict checking faster, we next show the `CanBeGranted` function (alg. 2). Given a request set and a timestamp, it will return **true** if there is no conflict amongst the requirement set and the related future lists, and **false** otherwise. When a request tuple is requesting read permission, it is necessary to check whether any write requests should be granted before the read. This is easy to check using the *nextWrite* value (lines 5-6). When the request is for write permission, it is necessary to check that no other request should be granted before the write (lines 8-9). This can be seen from the timestamp of the first element in the future list, as the list is in ascending order according to the timestamps. This makes the time complexity of checking for conflicts linear in the size of the request set *reqs*.

Algorithm 2 Checking for conflicts

```
1: function CANBEGRANTED(reqs, localTime)
2:     for all (id, perm) ∈ reqs do
3:         nextConflict := 0
4:         nextWrite, future ← FutureLists[id]
5:         if perm == Read then
6:             nextConflict ← nextWrite
7:         else
8:             _, t ← HEAD(future)
9:             nextConflict ← t
10:        end if
11:        if localTime > nextConflict then
12:            return false
13:        end if
14:    end for
15:    return true
16: end function
```

Algorithm 3 Updating future lists

```
1: procedure UNLOCK(reqs)
2:     for all (id, perm) ∈ reqs do
3:         nextWrite, future ← FutureLists[id]
4:         POP(future)
5:         if perm == Write then
6:             nextWrite ← ∞
7:             for all (type, time) ∈ future do
8:                 if type == Write then
9:                     nextWrite ← time
10:                    break
11:                end if
12:            end for
13:            FutureLists[id] ← (nextWrite, future)
14:        end if
15:    end for
16: end procedure
```

When it is time for the resource manager to unlock regions, it is necessary to do some bookkeeping to keep a valid *nextWrite* value, which we show in alg. 3. The resource manager removes the first element of the future lists of each region in the unlocked request set. If the region was unlocked after a write request, a new *nextWrite*

value has to be found. This is simply done by going through the list and taking the timestamp of the first encountered write request, or ∞ if there are no requests.

It may look like the time complexity of the `Unlock` procedure is proportional to the size of the request set times the length of the future lists. While that is indeed the worst case complexity, the amortized time is just linear in the size of the request set. Since the loop on lines 7-12 stops at the first write, only read requests will extend the execution of the loop. But for each subsequent read request iterated through, we are guaranteed that the next unlock involving the same id will not enter the true-branch on line 5, as this is only done for writes. Put plainly, if the inner loop runs 5 times on the same list, the front of the list must be 4 reads followed by a write, and it will take 5 unlock calls involving that list before bookkeeping is necessary again.

The last procedure to describe, before going into the main loop of the resource manager, is how the manager returns granted requests to the calling CSF (both alg. 4). In `Acquire`, all the IDs in the request set are converted into their corresponding address and size and written to the pointer specified by the calling CSF. Once that is done, the manager sets the value pointed to by $flag$ to 1, to indicate a successful obtain-call. The `Main` loop of the resource manager (alg. 5) then simply alternates calling `HandleArrival` (alg. 6) and `HandleRelease` (alg. 7).

Algorithm 4 Acquiring regions

```
1: procedure ACQUIRE(reqs, res, flag)
2:     for all (id, _) ∈ reqs do
3:         APPEND(res, Regions[id])
4:     end for
5:     flag ← 1
6: end procedure
```

Algorithm 5 Main loop

```
1: procedure MAIN
2:     while true do
3:         HANDLEARRIVAL
4:         HANDLERELEASE
5:     end while
6: end procedure
```

Starting with the `HandleArrival`, we remove the first element of the queue if possible (lines 2-3). If any of the IDs in the request set do not correspond to a region, we set *flag* to -1 and stop processing the obtain triplet (lines 5-7). If they are all valid, we add each request tuple to their respective future list (lines 11-17), along with the current logical time of the manager. If the permission of a request tuple is write, we may also need to update the *nextWrite* value of the future list (lines 14-16). This only happens if there are no writes in the list (in which case $nextWrite = \infty$), so we just take the minimum of the two timestamps. If the obtain triplet can be granted immediately, it acquires the requested regions, otherwise it is put into the *StbQue* (lines 19-23).

Algorithm 6 Processing new requests

```
 1: procedure HandleArrival
 2:     if NotEmpty(ArrQueue) then
 3:         reqs, res, flag ← Pop(ArrQueue)
 4:         for all (id, _) ∈ reqs do
 5:             if id ∉ Regions then
 6:                 flag ← −1
 7:                 break
 8:             end if
 9:         end for
10:         if flag ≠ −1 then
11:             for all (id, perm) ∈ reqs do
12:                 nextWrite, future ← FutureLists[id]
13:                 Append(future, (perm, Time))
14:                 if perm == Write then
15:                     nextWrite ← min(Time, nextWrite)
16:                 end if
17:                 FutureLists[id] ← (nextWrite, future)
18:             end for
19:             if CanBeGranted(reqs, Time) then
20:                 Acquire(reqs, res, flag)
21:             else
22:                 Enqueue(StbQue, (Time, req, res, flag))
23:             end if
24:             Time ← Time + 1
25:         end if
26:     end if
27: end procedure
```

Next in `HandleRelease`, when there are regions ready to be released, we unlock all of the released regions before we go through *StbQue*, so we do not have to check the queue multiple times at this step (lines 3-6). Then, each element in *StbQue* is checked for conflict (line 9) and allowed to acquire regions if there is no conflict (line 10), before being removed from the queue (line 11). Acquiring regions during the iteration does not introduce conflicts that were not already present, as `CanBeGranted` uses *FutureLists* to check for conflicts, and all triplets have already been added to *FutureLists* before they were added to *StbQue*.

Algorithm 7 Releasing regions

```
 1: procedure HandleRelease
 2:     WasReleased := NotEmpty(Releases)
 3:     while NotEmpty(Releases) do
 4:         r ← Pop(Releases)
 5:         Unlock(r)
 6:     end while
 7:     if WasReleased then
 8:         for all (time, reqs, res, flag) ∈ StbQue do
 9:             if CanBeGranted(reqs, time) then
10:                 Acquire(reqs, res, flag)
11:                 Remove(time, StbQue)
12:             end if
13:         end for
14:     end if
15: end procedure
```

Since `HandleRelease` iterates through all elements in *StbQue* and calls `CanBeGranted` for each, the time complexity becomes the sum of the size of the request sets of the elements in *StbQue*. This could be quite the bottleneck, but as we saw in Table 5.1, the CSDs used in related projects do not possess more than 8 cores, and we assume a protocol of one CSF per core. This will keep *StbQue* rather small and limit the running time of `HandleRelease`.

5.4 Safety & Liveness

We will now sketch out proofs of the claims laid out in the beginning of Section 5.1. As the proofs rely on the future lists being sorted, we start by proving this property.

Corollary 5.4.1. All future lists are in ascending sorted order according to the timestamps of the elements.

Proof Sketch. Inductively, we have the base case of when a future list is first created. This is during the `Allocate` call, in which case the future list is the singleton $\{(Write, 0)\}$, which must necessarily be sorted. In the inductive case, we assume all current future lists are in sorted order. We can modify a future list either by appending a new element during `HandleArrival` or by removing the first element during `Unlock`. Since removing the first element of a sorted list still results in a sorted list, the `Unlock` case is trivial. In `HandleArrival`, elements are assigned a timestamp on line 13. As this value is only changed by incrementing on line 24 of the same procedure, we get that the value is strictly increasing. This means that any elements in a future list cannot have a higher timestamp than the one we are currently inserting. Thus, inserting a greatest element at the end of a sorted list maintains the sortedness. □

Lemma 5.4.2. A CSF is guaranteed to eventually gain access to any region of shared memory used in the CSFs computation.

Proof Sketch. If the currently processed CSF is not granted immediate access to the requested regions on line 20 of `HandleArrival`, it must be because it conflicts with at least one element in the future list of one of the requested regions. If the time assigned to the CSF is t_c, and the lowest timestamp of any element amongst the future lists is t_i, then we know that $t_i < t_c$ from corollary 5.4.1. Since the future lists are sorted, there are at most $t_c - t_i$ elements ahead of an element with timestamp t_c. This means that after at most $t_c - t_i$ calls to `Unlock`, all elements preceding elements with timestamp t_c have been removed, in which case no conflicts remain, and access can be granted. □

Lemma 5.4.3. Once a CSF has obtained access to a shared memory region, the contents of that region will not change outside of the operations performed by the CSF itself.

Proof Sketch. There are only two points where a CSF can acquire access: line 20 of `HandleArrival` and line 10 of `HandleRelease`. Both lines are nested within a check of `CanBeGranted`. Therefore, if a CSF C has acquired access to a region, there can be no write operation ahead of it in the future list, based on the comparison on line 11 in `CanBeGranted`. Any writes appearing later in the future list will also not be granted access while C is still present in the list. This means that only writes performed by C can change the accessed region. □

Lemma 5.4.4. A CSF that has started execution will not be aborted due to concurrency conflicts.

Proof Sketch. As all request sets have to go through a queue on arrival, their order in the queue imposes a total order on all request sets. Access to regions in a request set is either granted all at once or not at all, so deadlocks cannot occur. There is therefore no need to abort a request set to break a deadlock. Additionally, since lemma 5.4.3 shows that accesses do not interfere with each other, a CSF can write changes directly into accessed regions, without needing to validate changes in a commit-style protocol. This means that once a CSF is done executing, it can simply release its regions and terminate. □

5.5 Discussion

5.5.1 Freeing Memory Regions

Section 3.2 regarding the DSL syntax, Section 3.3.3 on type checking and Section 5.2 on the resource manager all deal with how to allocate and access shared memory, but not how to de-allocate it. Freeing memory is something that should be done by the host program, and thus it is not necessary for a lot of support for de-allocation on the CSD side. What is needed is just an operation in the host/device-interface for supplying IDs of the regions to de-allocate, and then the resource manager removes the freed regions from its map of regions.

The reason for placing allocation of shared memory at the CSF-level but de-allocation at the host-level is that different information is required for allocation and de-allocation. For allocation, what is important is to know how much memory to allocate. Although the host program could allocate shared regions, the CSFs have more information available to reason about the required size. If, for example, the result of a filtering operation should be stored in shared memory, the CSF performing the filter knows exactly how large the result is and can allocate accordingly, while the host must allocate a region that is guaranteed to be large enough. Freeing shared memory is, however, not a question of *how much*, but a question of *when*. The overall goal of CSDs is to support host programs in processing data, so deciding when data stored in shared memory are no longer relevant is a task better suited for the host program.

5.5.2 Resizing Shared Regions

As discussed in Section 3.4, there are two different types of lengths when it comes to shared regions, denoting either the concrete contents or the maximum capacity of the region. The maximum capacity is information maintained by the resource manager, and the content length is a value local to a program instance. However, it may make sense to facilitate programs informing the manager of the content length of a shared region. If a file is read into a region bigger than the file size, and another program intends to read the file from that region, the manager ends up misleading the program by providing the max capacity as the length of the region. The original program responsible for reading the file should either have a way of resizing the length of the region to match the file size, or the manager should be modified to keep track of both values for shared regions. This however slightly breaks the abstraction of the `mem_range struct` discussed in Section 3.2, as it would need a separate field only for shared regions.

If the resizing approach is taken, the most simple solution would be for all CSFs to send the final value for the local length of any shared regions for which it has write access. It would require a small change in the `OnRelease` procedure, to not only accept requests to be released, but also potential size updates. The `Unlock` procedure (algorithm 3) would then have to update the region with a new size once it enters the true-branch on line 5.

5.5.3 Multiple Users

The resource manager is designed with the assumption of a single-user system. This means that the design does not address issues of security or privacy, which can be important if computational storage is to be used by service providers in e.g cloud solutions. Currently, any program can get access to any shared memory region, as long as the program provides a valid ID. To facilitate privacy, a possible extension of the design is to also assign a hard-to-guess token to shared regions, which programs must prove they know before the resource manager will handle their requests. One remedy for this would be to employ virtualization [62], by sectioning shared memory and assigning each user to a section. The resource manager can then be replicated

and configured to only work within a specific section, preventing users from accessing regions allocated by a resource manager other than their own.

5.6 Conclusion

This chapter has covered the design of a resource manager that utilizes a novel queue system to minimize the amount of communication a host and a CSD need to prevent concurrency conflicts. While other computational storage projects use queues to enable asynchronous execution, the resource manager in this chapter is designed to work in tandem with Rogers, the CSF validator of Chapter 3, to automatically detect dependencies between CSFs. This frees the host application from having to insert synchronization barriers between CSFs with implicit dependencies. Additionally, the chapter concludes with informal proof outlines, showing that the resource manager avoids common concurrency issues in the form of deadlocks, resource starvation, and non-serializability.

Chapter 6

Evaluation

In this chapter, we evaluate how the decisions regarding compositional devices and the covered implementations affect the performance of CSFs. More precisely, we want to find answers to the following questions:

- When comparing the programmable and compositional paradigm of Figure 2.3, the compositional paradigm splits computational storage programs into two parts: computations and data management. Does this split affect the throughput of CSFs? This question is answered in Section 6.2.
- The concept of registered functions may require rewriting computations to ease verification, as seen in the conversion example of Section 4.2.2. Will the goal of writing verified code conflict with writing performant code? This question is answered in Section 6.3.
- The resource manager of Chapter 5 removes some of the responsibility of concurrency control from the programmer by queuing and delaying CSFs to avoid conflicts. Is the overhead introduced by this queue system small enough to have the benefits outweigh the performance loss? This question is answered in Section 6.4.
- When composing registered functions where one function consumes the output of the other, the CSF will manifest potentially unnecessary regions to store the intermediate output. How big of a drawback does this manifestation pose? This question is answered in Section 6.5.

6.1 Experimental Setup

The evaluation methods of previous research commonly use both a host device and a CSD [64, 47, 40, 74, 38, 72]. The evaluations consist of measuring the performance of a host program invoking functionality of the CSD. However, our goal is not to re-establish the overall performance gain of programs utilizing CSDs for data-processing, since we consider this a fact proven by earlier research. Rather, our objective is to evaluate the performance between existing CSFs and ones written in the Rogers DSL. Thus, we focus our evaluation on running micro-benchmarks on the CSD-side.

To ensure that our experiments are run with similar resource constraints as the projects shown in Table 5.1, we evaluate the CSFs on a Jetson Nano[1] with 4 x ARM Cortex-A57 cores at a clock frequency of 1.5 MHz. For all of the benchmarks of this chapter, we perform 10 warmup runs and then average the execution time of the following 100 runs. We time the execution using the `clock()` function from the `time.h` library in C.

For our choice of execution environment, we utilize the UBPF VM [54], a virtual machine for executing eBPF programs. This environment was also used to evaluate Delilah [40], the project on which we based the registered functions of Chapter 4. We also choose the CSFs of the Delilah project as our comparison points, and reusing the original execution environment ensures a more fair comparison. The experiments can be found at `https://github.com/mortentc/rogers_experiment`, but the Delilah CSFs are stored in a private submodule of the repository.

6.2 Measuring the Split Between Data and Computation

The UBPF VM [54] is able to execute eBPF programs either by interpreting the bytecode or by compiling the code further to native instructions. The main idea behind using eBPF for code offload in computational storage is to provide a method of writing programs that are agnostic to the underlying instruction set architecture. Using the UBPF VM to execute eBPF programs avoids recompiling the programs to machine code, which is why the Delilah project used it in their evaluation. Additionally, the UBBF VM supports linking pre-compiled functions, e.g. the registered functions of Chapter 4, to the VM, allowing eBPF programs to invoke the functions.

Figure 6.1 shows the result of different execution methods for a CSF performing a sum. We evaluated two different CSFs: one written in the Rogers DSL that calls the sum defined in Section 4.2.1, another CSF from the Delilah project which in-lines the sum computation. Both CSFs operate on 1e6 32-bit integers, with half as many indices to sum, resulting in 6 MB of data.

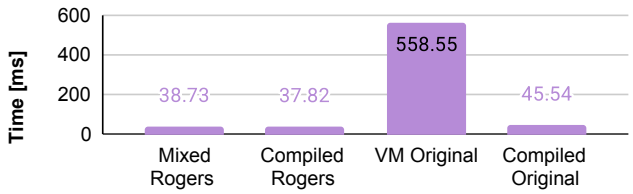


FIGURE 6.1: Comparison of different execution methods for a sum CSF.

The first evaluation method, labeled *Mixed Rogers*, consists of compiling the Rogers CSF to eBPF and interpreting the CSF in the UBPF VM. In *Compiled Rogers*, the second method, we compile the Rogers CSF to machine code and execute it directly. As can be seen from the results, only 0.01 ms separates the two methods, accounting for less than 0.1% of the execution time. As the majority of the computation is

[1] `https://developer.nvidia.com/embedded/jetson-nano`

performed by the registered sum function, which is compiled in both cases, performing the data placement aspect as interpreted eBPF code imposes no significant overhead.

The third method is *VM Original* and consists of the UBPF VM interpreting the in-line sum CSF from the Delilah project. With the sum computation being part of the CSF and thus translated to eBPF as well, the CSF experiences a massive slowdown. This result corroborates the earlier result from the Delilah project [39], where the authors found that fully interpreted computational storage programs run ∼10× slower than their compiled counterparts. We arrive at the same conclusion when comparing the results of the final execution method, *Compiled Original*, in which we fully compile the Delilah CSF. The difference in execution time between the interpreted and the compiled Delilah CSF is a factor of 12.4×. This fact further motivates the use of registered functions and compositional devices, given how much benefit there is to implementing computations specific to the device on which they will be executed.

We can reason about the impact of splitting data and computations into CSFs and registered functions by comparing the mixed approach to the fully compiled approach. Even with the VM interpreting eBPF at an order of magnitude slower than running native code, performing the data placements as interpreted eBPF code has a negligible impact on performance. From this we can conclude that execution time is dominated by the registered functions. This is further cemented when comparing the compiled Delilah CSF with the Rogers CSF. Had the two CSFs performed exactly the same computation, the Delilah CSF should have been slightly faster by virtue of not performing function calls. In its current state, the Delilah CSF is 17% slower than the Rogers. The only difference between the two computations is the fact that the Delilah CSF stores the result of the sum in memory rather than in a register, which means that the Delilah CSF performs an extra memory access each time the result is incremented.

To further show that the implementation of the registered functions is a deciding factor for a CSFs performance, we compared Rogers CSFs (*Mixed Rogers*) calling the convert and filter functions from Sections 4.2.2 and 4.2.3 to their in-lined Delilah counter-parts (*Compiled Original*). The results are shown in Figure 6.2. The convert CSFs ran on 4 MB of strings (corresponding to 500,000 dates) and the filters ran on 4MB of 32-bit integers with a selectivity of 10%.

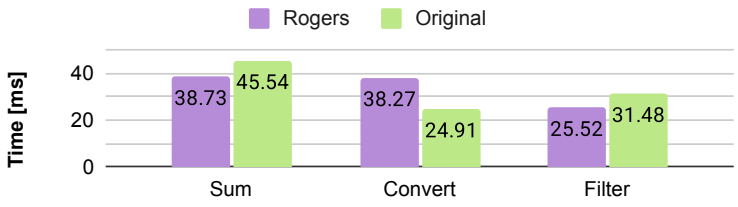


FIGURE 6.2: Running times for Rogers CSF calling the different registered functions compared to their original Delilah counterpart.

As we have already covered in Section 4.2.2, we reversed the decision to unroll the inner loop of the `convert_dates` function in order to provide a loop invariant for the

verification of the function. The decision has resulted in a 54% slowdown compared to the unrolled Delilah CSF. We discuss the impact of this result further in Section 6.3.

An important note regarding the evaluation of the filter CSF is that we encountered a limitation to the number of arguments eBPF supports in function calls. There is a limit of 5 arguments per call and, as the reader may recall, we added a sixth parameter to the filter function in Section 4.2.3, in order to comply with the requirement of updating region sizes during function calls. This means that for evaluating the filter CSF, we fully compiled the CSF to native code, rather than compiling it to eBPF and using the VM to interpret.

Like the sum CSFs, the only difference between the filter CSFs is how they store the results. As required by the type system of the Rogers DSL, the Rogers CSF stores the result of the filter in the part of local memory that immediately follows the input data. The Delilah CSF stores the result *before* the input data. As the results in Figure 6.2 show, this small change makes the Rogers CSF perform 23% faster. To understand how this placement affects performance, consider Figure 6.3.

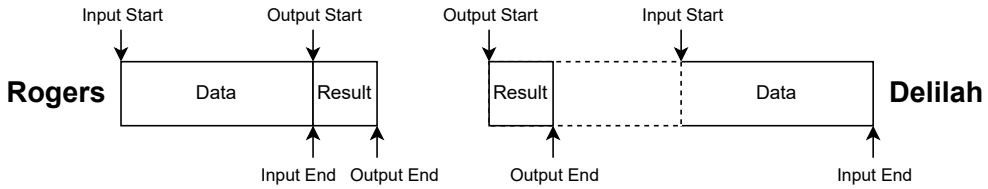


FIGURE 6.3: Data placement of the Rogers and Delilah CSFs performing filtering.

In both CSFs, the start of the input is separated from the start of the output by a distance equal to the length of the input (so 4 MB in this case). As elements get filtered out during execution, the distance of the two pointers changes. For the Rogers CSF, since the output is located next to the end of the input, the distance between the two pointers shrinks. For Delilah, the distance increases. For a filter with 10% selectivity, the end result is a pointer distance of 0.4 MB for Rogers and 7.6 MB for Delilah. The size of the L2 cache on the Jetson Nano is 2 MB [30], so the Rogers CSF is able to benefit more from sequential locality once the pointer distance gets below a certain threshold. For the Delilah CSF, this threshold is never reached.

The choice for placing the output before the input data is likely to minimize the amount of direct memory accesses necessary to transfer results between host and CSD. Using the Delilah approach, the output is placed next to the CSF parameters / layout `struct` in memory, so a single DMA can be used to transfer both the parameters and filter results back to the host. By placing the results after the input, two separate DMAs are needed to transfer both parameters and results back.

Although placing the results next to the parameters reduces the required DMA transfers from two to one, the singular transfer has to include more data than necessary. The host cannot know exactly how much data to transfer, as the output size of the

filter is initially unknown, so the host must transfer the entire region between the layout and the input data to guarantee that all the results are fetched.

6.3 Verifiability versus Performance

As we already saw from Figure 6.2, the choice to roll up the loop of the convert function for the sake of verifiability has resulted in 54% slower performance. Figure 6.4 shows that if we execute the same function and convert the dates to month or day, which only performs two loop iterations per string, the penalty becomes a little less severe, at around 41% slowdown.

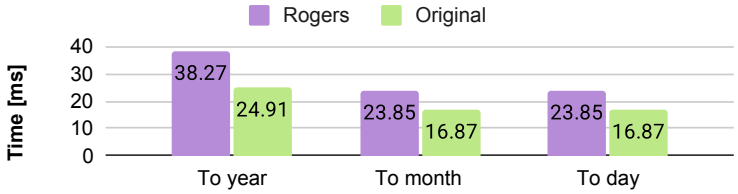


FIGURE 6.4: Running times of CSFs extracting different information from dates.

Even with the reduction in performance loss when converting to months or days, a 41% slowdown is in most cases unacceptable. This does not necessarily mean that verification has to get in the way of performance. A common strategy to handle hard-to-verify program optimizations is to implement two variants of the program under consideration: one with optimizations and one that is easier to verify. Once the easier of the two programs have been verified correct with regard to a specification, the other program can be verified by proving that the two programs are equivalent [70], i.e. produce the same outputs for all inputs.

Program equivalence is often used in proving the correctness of program transformations, such as optimizing compilers [60], and in our case it can be used to prove that optimized registered functions match the same specification as the verified registered functions. Of course, we are not guaranteed to have an easy way of proving the equivalence between two registered functions. However, depending on how strict we want the verification to be, we can deem other methods sufficient in determining equivalence. Testing the outputs of the functions on a sample of randomly generated input, similar to property based testing (recall Section 2.4), is one such method. All of this is to say, although the convert function verified in this thesis is 41-54% slower than its optimized counterpart, we are not left without the possibility of having our cake and eating it too.

6.4 Overhead of the Resource Manager

There are three possible interactions between CSFs and the resource manager: obtaining regions, releasing regions, and allocating regions. To estimate the overhead introduced by the resource manager, we have implemented a simple version of the data structure and procedures described in Section 5.3 in C. For the mappings from

IDs to regions and future lists, we use hash-tables with chaining, and for the queues, we use linked lists. The execution times for the three actions are summarized in Table 6.1.

Table 6.1: Latency of each interaction between a CSF and the resource manager.

Operation	Obtain	Release	Allocate
Time taken	$1.94\mu s$	$1.57\mu s$	$2.33\mu s$

For the evaluation, we started with empty queues and no mappings and then performed 100 individual region allocations, averaging the time of each allocation. Afterwards, we created a CSF with the only action of obtaining read permission to a single region. To include the time it took to initialize the request array (discussed in Section 3.2), we also created an empty CSF performing no actions and measured the time difference between the empty CSF and the CSF obtaining the region, rather than just measuring the time of the `obtain` call. It took $1.59\mu s$ to run the empty CSF and $3.53\mu s$ for the region-obtaining CSF, giving the difference of $1.94\mu s$ shown in Table 6.1. After each run of the CSF that obtains a region, we measured the time it took to release the same region.

As can be seen from the results, the overheads of the individual actions are in the order of microseconds. As we saw in Sections 6.2 and 6.3, CSFs working on megabytes of data are measured in tens of milliseconds. This means that the latency of interacting with the resource manager is 4 orders of magnitude smaller than the running time of the CSFs evaluated so far. This comes with the caveat that we have evaluated the resource manager with only one active CSF at a time, even though it is designed to arbitrate conflicts between multiple CSFs.

Intuitively, it would make sense to evaluate the manager with more CSFs running concurrently, but we reach a limitation with our testing setup. Having multiple CSFs running is relevant when a CSF is waiting on another CSF to finish. When a CSF releases regions, the resource manager has to update the future lists of the relevant regions before it can process the obtain call of another CSF. Measuring the time between the release call of the first CSF and the completion of the obtain call of the second CSF would evaluate the latency introduced by the manager, but we are missing a value to compare this latency to. Recall Figure 5.1 to understand that in a scenario without a resource manager, the waiting CSF cannot start execution immediately following the termination of the previous CSF. The CSD first needs to inform the host that the first CSF has finished executing, after which the host can send an activation request for the second CSF. This message passing also introduces latency and should be the one to which we compare the updating of future lists. Depending on the fabric between the CSD and the host, updating lists locally may even be faster than performing a round-trip, but without a more complete setup of host, CSD, and I/O connection between the two, we cannot investigate this claim further.

6.5 Composing Functions

One of the key aspects of the Rogers DSL has been to support the composition of registered function calls when writing CSFs. Our last evaluation takes a CSF that composes a call to convert followed by a call to filter and compares its performance to a CSF from the Delilah project which performs the same computation of converting and filtering. The Delilah CSF converts and filters each element one at a time, whereas the Rogers CSF converts all dates first and only then performs the filter. This means the Rogers CSF creates an extra region half the size of the strings to store the converted and unfiltered dates, while the Delilah CSF saves this space. Using concrete numbers, with a selectivity of 10% for the filter, for every 1 MB of date strings, the Rogers CSF will store 500 KB of converted dates and 50 KB of filtered dates. The Delilah CSF, on the other hand, will only store 50 KB of filtered dates.

Although we are interested in measuring the impact of this extra space usage, we also saw in Sections 6.2 and 6.3 that the Rogers and Delilah CSFs differ in more than one aspect, making it difficult to isolate the impact of just the extra stored values. We know that the conversion performed by the registered function is 54% slower than the one done by the Delilah CSF, while the filtering is 23% faster. Given that the input for the conversion is the largest data region, it would be reasonable to assume the Rogers CSF to be outperformed. As Figure 6.5 shows, this is also the case when using 1 MB of strings as input, where the Rogers CSF is 19% slower than the Delilah CSF.

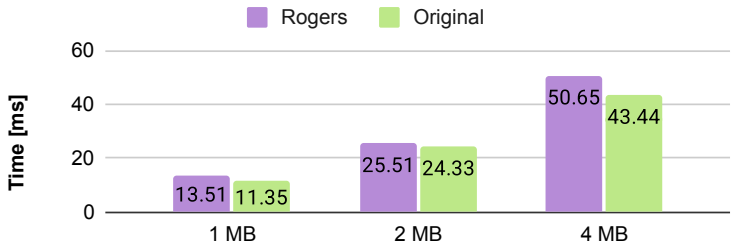


FIGURE 6.5: Scalability of CSFs performing a composed conversion and filtering.

However, something interesting happens when the input size is increased to 2 MB. As mentioned earlier, the size of the L2 cache on the Jetson Nano is 2 MB, so when the input data was just 1 MB, both CSF could fit all their data entirely in cache. Since the Delilah CSFs place data further apart, as we discussed in Section 6.2, when we approach the size limit of the cache, the better use of sequential locality by the Rogers CSF results in only 4.8% difference in performance. Once we increase the size of the data further beyond the limit of the cache, the benefit of locality lessens, and the performance difference goes back up to 17%. What we can conclude from this is that avoiding creating redundant data is less important than placing it optimally.

6.6 Conclusion

From the evaluations, we have learned that there are two main factors that impact the performance of CSFs:

- The implementation of the computations / registered functions.
- The choice of data placements.

We saw in Section 6.2 how even small differences in the sum and conversion computations could either speed up a CSF by 17% or slow it down by 54%. Additionally, we saw how the exact same filter computation performed 23% faster when placing input and output closer together. Section 6.5 showed that even when relying on a slower registered function, placing data to make better use of the processor cache can negate the loss in performance, implying that proper data placement is just as important as optimizing computations.

With the execution time being dominated by the two factors mentioned above, we conclude that it is possible to introduce small overheads, such as interpreting Rogers CSF or interacting with the resource manager, with no significant impact on the overall performance. For the resource manager, its main goal is reducing the amount of communication necessary between host and CSD, so further testing is required to investigate the performance benefits of this reduced communication.

Chapter 7

Conclusion & Future Work

This thesis expands the current research on computational storage by proposing the use of *compositional devices*, a new type of storage device with statically installed functions which can be composed together to form data-processing queries, or *computational storage functions* (CSF). In the beginning of the thesis we pose the question: "How can software developers create programs for computational storage in a way that guarantees termination of programs, memory safety, and expected output?". We have answered this question in the following manner.

The Rogers DSL introduced in Chapter 3 defines a programming language to create CSFs for compositional devices and uses region-based memory management to ensure computations stay within predefined bounds. The memory management is supported by the type system of the DSL, which is accompanied by proofs showing that any type checked program produces non-fragmented data regions. The evaluation in Chapter 6 highlights the positive performance impact of non-fragmented regions, showing cases of 23% faster computation for specific functions. We also make clear under which conditions the performance improvements occur.

By relying entirely on statically installed functions for processing data, Chapter 4 shows how program verification tools can give rise to *registered functions*, statically installed functions that have been verified to match a specification and designed to comply with the requirements of the Rogers DSL. The verified specifications can be used to provide an enriched type signature for the functions, providing information about region sizes to Rogers.

CSFs can share results between each other through the use of shared memory. To resolve potential concurrency conflicts between CSFs, Chapter 5 covers the design of a resource manager. The manager uses a queue system to detect and delay conflicting CSFs, which guarantees the absence of dead-locks while at the same time guaranteeing eventual access. Evaluations in Chapter 6 show that the overhead imposed by the resource manager constitutes a fraction of the running time of CSFs.

In summary, the Rogers framework for compositional devices provides a way for application programmers to interact with computational storage in a predictable manner, a methodology for device manufacturers to create trustworthy specifications for compositional devices, and a foundation for data system researchers to build future research

upon. With the findings of the thesis summarized, we next propose future research directions for compositional devices.

7.1 Applications for Compositional Devices

With the creation of a DSL for compositional CSFs and a framework for creating registered functions, an obvious next step is looking at which domains contain further candidates for registered functions. In Chapter 4 we show why some operations in database queries are suitable for translation into registered functions. Additionally, most spreadsheet applications, such as Microsoft Excel, also contain functions for operating on columns of data[1], which could be converted to registered functions for use in the domain of statistics. Similarly, some projects such as Daphne [27] deal with the creation of pipelines for data analysis. For that purpose, they define operations on matrices which, if translated to registered functions, could expand the pipelines to also cover computational storage.

7.2 Mechanizing the Type System Proofs

The type rules defined in Section 3.3 along with the proofs of non-fragmentation are created with an emphasis on being easy to export to a proof assistant. As part of sanity testing the rules during the proof process, the definitions and type rules were implemented in Gallina, the specification language of Rocq [16], with some of the lemmas partially verified. Using the proofs in Section 3.3.3 as an outline, using a proof assistant to create mechanized proofs of the type system properties is a rather low-hanging fruit. In addition to strengthening the belief in the correctness of the type system, a collection of mechanized lemmas would serve as a foundation for proving properties about any future additions to the DSL, such as the ones we cover in the next section.

7.3 Additional Language Features

With the design of the Rogers DSL in Chapter 3, we have provided a way for programmers to define CSFs for compositional storage devices, with the ability to invoke registered functions, organize data in a contiguous manner, and share results between CSFs. Using the Rogers DSL and validator as a base, the language can be extended with more features to provide increased performance or usability. In this section, we suggest a handful of such features.

Releasing Locks Early

As part of preventing concurrency conflicts, we mentioned in Section 5.1 that CSFs employ conservative two-phase locking. Although this is true, the implementation shown more specifically uses both conservative and strict two-phase locking (S2PL), since locks are only released once a CSF terminates. Using S2PL is not a requirement to guarantee the properties defined in Section 5.1, but rather a consequence of being

[1] https://support.microsoft.com/en-us/office/excel-functions-by-category

simpler to implement. However, this means that CSFs can end up holding access to a region for longer than necessary, thereby delaying CSFs that depend on the same region. We have already shown that Rogers can be used to infer the necessary permissions for region accesses in Section 3.4.1. This analysis can be expanded to detect when a CSF accesses a region for the last time and then emit a release call to the resource manager immediately after the last access. This would slightly increase the running time of individual CSFs, as they would have to interact with the resource manager multiple times, but could massively speed up the overall execution of a host program by allowing dependent CSFs to start execution as early as possible.

Automated Region Construction

To prove the non-fragmentation of local regions in the DSL in Section 3.3.3, we utilize a mapping ω from offset expressions to their canonical offsets. The argument is that all pointers to local memory must naturally be equivalent to adding the length of preceding memory regions to the address of the `dynamic` array defined on every layout `struct`. Following this argument further, it should be possible for Rogers to automatically construct pointers to any local region. Additionally, from the typing context σ, Rogers is able to search for a variable that stores the length of a local region. Combining these facts we can see that Rogers has the ability to automatically initialize the length and address of local memory regions if they are typeable in the given program state. We can leverage this ability to free programmers from having to manually set values for each region. Instead, a programmer could declare a memory region of type `region` i, with Rogers automatically constructing a pointer of type `address` i and a length of type `length` i. Such a declaration is not naturally supported by the C99 standard, so the implementation of Rogers would either have to move away from the current `language-c` parser, or define some existing syntax pattern in the DSL to have the special meaning of region declarations.

Destructive Updates

Section 6.2 compares the performance of a filter CSF placing output after input, to another filter CSF placing output *before* input, and saw a 23% speedup in the first case. This speedup would likely increase further if the CSF overwrites the input with the results, as this is the closest results can be placed to input. However, the Rogers DSL does not support writing to existing local regions. Changing the DSL to allow destructive updates to regions would require extensive changes to the type system, and most certainly break the proofs of non-fragmentation. The same is not true for shared regions, though. If the specification of a registered function does not require a read region and a write region to be separated, the same shared region could be used as both. The reason this does not work for local regions is that the type system does not allow a local region to be typed as both read and write at the same time.

An alternative to supporting destructive updates to local regions is fusing registered functions. Other languages that support composition of computations, such as Futhark [41], also support the fusion of two computations into a single equivalent computation. In Section 6.5 we evaluated a CSF calling registered functions performing a conversion and a filtering. The filter results are placed after the conversion results, but if a fused `convert+filter` function had been used, the end result would

be a single region of results. This end state is equivalent to first creating the conversion results and then overwriting them with the results of the filter. To enable this, a compositional device can contain a selection of fused registered functions for commonly composed functions, and Rogers can be extended with instructions to replace the composed function calls with their fused variant.

Bibliography

[1] Aug. 2022. URL: https://www.snia.org/sites/default/files/technical-work/computational/release/SNIA-Computational-Storage-Architecture-and-Programming-Model-1.0.pdf.

[2] URL: https://alt-ergo.ocamlpro.com/. Accessed: 2025-08-25.

[3] URL: https://www.microsoft.com/en-us/research/project/z3-3/. Accessed: 2025-08-25.

[4] URL: https://isabelle.in.tum.de/. Accessed: 2025-08-25.

[5] URL: https://www.adacore.com/about-spark. Accessed: 2025-08-25.

[6] A. Afroozeh, L. Felius, and P. A. Boncz. Accelerating GPU Data Processing using FastLanes Compression. In C. Binnig and N. Tatbul, editors, *Proceedings of the 20th International Workshop on Data Management on New Hardware, DaMoN 2024, Santiago, Chile, 10 June 2024*, 8:1–8:11. ACM, 2024. DOI: 10.1145/3662010.3663450. URL: https://doi.org/10.1145/3662010.3663450.

[7] A. Ailamaki, E. Liarou, P. Tözün, D. Porobic, and I. Psaroudakis. How to stop under-utilization and love multicores. In *2015 IEEE 31st International Conference on Data Engineering*, pages 1530–1533, 2015. DOI: 10.1109/ICDE.2015.7113419.

[8] M. Ameri and C. A. Furia. Why just boogie? translating between intermediate verification languages. *CoRR*, abs/1601.00516, 2016. arXiv: 1601.00516. URL: http://arxiv.org/abs/1601.00516.

[9] L. B. Andrew W. Appel and Q. Cao. *Verifiable C*. B. C. Pierce, editor, volume 5 of *Software Foundations*. Electronic textbook, 2023. URL: http://softwarefoundations.cis.upenn.edu. Version 1.2.2.

[10] A. Barbalace and J. Do. Computational storage: where are we today? In *Conference on Innovative Data Systems Research*, 2021. URL: https://api.semanticscholar.org/CorpusID:232328691.

[11] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. *Boogie: a modular reusable verifier for object-oriented programs*. In *Formal methods for components and objects*. F. S. de Boer, M. M. Bonsangue, S. Graf, W.-P. de Roever, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, and G. Weikum, editors. Volume 4111. Lecture notes in computer science. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pages 364–387. ISBN: 978-3-540-36749-9. DOI: 10.1007/11804192_17. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2005/01/krml160.pdf.

[12] J. Barr. AQUA (Advanced Query Accelerator) – A Speed Boost for Your Amazon Redshift Queries. White Paper, Amazon Web Services, 2021. URL: https:

//aws.amazon.com/blogs/aws/new-aqua-advanced-query-accelerator-for-amazon-redshift/.

[13] S. Bates and M. Ichilevici de Oliverira. Eid-hermes/specs/eid-hermes-theory-of-operation.md at master · eideticom/eid-hermes, Jan. 2021. URL: https://github.com/Eideticom/eid-hermes/blob/master/specs/eid-hermes-theory-of-operation.md.

[14] P. Baudin, F. Bobot, L. Correnson, Z. Dargaye, and A. Blanchard. *WP Plug-in Manual*. URL: https://frama-c.com/download/frama-c-wp-manual.pdf.

[15] P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI/ISO C Specification Language*. URL: http://frama-c.com/download/acsl.pdf.

[16] Y. Bertot and P. Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013. URL: https://coq.inria.fr/documentation.

[17] S. Blackshear, E. Cheng, D. L. Dill, V. Gao, B. Maurer, T. Nowacki, A. Pott, S. Qadeer, D. Rain, S. Russi, T. Sezer, R. Zakian, and Zhou. Move: a language with programmable resources. In 2019. URL: https://api.semanticscholar.org/CorpusID:201681125.

[18] A. Blanchard, F. Bobot, P. Baudin, and L. Correnson. *Formally verifying that a program does what it should: the wpwp (frama-c plug-in) plug-in*. In *Guide to Software Verification with Frama-C: Core Components, Usages, and Applications*. N. Kosmatov, V. Prevosto, and J. Signoles, editors. Springer International Publishing, Cham, 2024, pages 187–261. ISBN: 978-3-031-55608-1. DOI: 10.1007/978-3-031-55608-1_4. URL: https://doi.org/10.1007/978-3-031-55608-1_4.

[19] A. Blanchard, F. Loulergue, and N. Kosmatov. Towards full proof automation in frama-c using auto-active verification. In *NASA Formal Methods*, 2019. URL: https://api.semanticscholar.org/CorpusID:167221771.

[20] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd Your Herd of Provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wroclaw, Poland, 2011. URL: https://inria.hal.science/hal-00790310.

[21] J. Boukhobza and P. Olivier. 7 - flash translation layer. In J. Boukhobza and P. Olivier, editors, *Flash Memory Integration*, pages 129–147. Elsevier, 2017. ISBN: 978-1-78548-124-6. DOI: https://doi.org/10.1016/B978-1-78548-124-6.50010-4. URL: https://www.sciencedirect.com/science/article/pii/B9781785481246500104.

[22] J. Breitner, P. Gardner, J. Lee, S. Lindley, M. Pretnar, X. Rao, A. Rossberg, S. Ryu, W. Shin, C. Watt, and D. Youn. Wasm spectec: engineering a formal language standard, 2023. arXiv: 2311.07223 [cs.PL]. URL: https://arxiv.org/abs/2311.07223.

[23] J. Cao, R. Sen, M. Interlandi, J. Arulraj, and H. Kim. Gpu database systems characterization and optimization. *Proc. VLDB Endow.*, 17(3):441–454, Nov. 2023. ISSN: 2150-8097. DOI: 10.14778/3632093.3632107. URL: https://doi.org/10.14778/3632093.3632107.

[24] W.-N. Chin, H. H. Nguyen, S. Qin, and M. Rinard. Memory usage verification for oo programs. In *Proceedings of the 12th International Conference on Static Analysis*, SAS'05, pages 70–86, London, UK. Springer-Verlag, 2005. ISBN: 3540285849. DOI: 10.1007/11547662_7. URL: https://doi.org/10.1007/11547662_7.

[25] J.-Y. Choi, E. H. Nam, Y. J. Seong, J. H. Yoon, S. Lee, H. S. Kim, J. Park, Y.-J. Woo, S. Lee, and S. L. Min. Hil: a framework for compositional ftl development and provably-correct crash recovery. *ACM Trans. Storage*, 14(4), Dec. 2018. ISSN: 1553-3077. DOI: 10.1145/3281030. URL: https://doi.org/10.1145/3281030.

[26] L. Correnson, P. Cuoq, F. Kirchner, A. Maroneze, V. Prevosto, A. Puccetti, J. Signoles, and B. Yakobowski. *Frama-C User Manual*. URL: http://frama-c.com/download/frama-c-user-manual.pdf.

[27] P. Damme, M. Birkenbach, C. Bitsakos, M. Boehm, P. Bonnet, F. Ciorba, M. Dokter, P. Dowgiallo, A. Eleliemy, C. Faerber, G. Goumas, D. Habich, N. Hedam, M. Hofer, W. Huang, K. Innerebner, V. Karakostas, R. Kern, T. Kosar, A. Krause, D. Krems, A. Laber, W. Lehner, E. Mier, M. Paradies, B. Peischl, G. Poerwawinata, S. Psomadakis, T. Rabl, P. Ratuszniak, P. Silva, N. Skuppin, A. Starzacher, B. Steinwender, I. Tolovski, P. Tözün, W. Ulatowski, I. Wrosz, A. Zamuda, C. Zhang, and X. Zhu. Daphne: an open and extensible system infrastructure for integrated data analysis pipelines. English. In *12th Conference on Innovative Data Systems Research, CIDR 2022*, 2022. DBLP License: DBLP's bibliographic metadata records provided through http://dblp.org/ are distributed under a Creative Commons CC0 1.0 Universal Public Domain Dedication. Although the bibliographic metadata records are provided consistent with CC0 1.0 Dedication, the content described by the metadata records is not. Content may be subject to copyright, rights of privacy, rights of publicity and other restrictions.

[28] J. Do. SoftFlash: Software-Defined Flash for Programmable Storage. Presentation at SNIA SDC 2017, Sept. 2017. URL: https://www.snia.org/educational-library/softflash-software-defined-flash-programmable-storage-2017.

[29] J. Do, Y.-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt. Query processing on smart ssds: opportunities and challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1221–1230, New York, New York, USA. Association for Computing Machinery, 2013. ISBN: 9781450320375. DOI: 10.1145/2463676.2465295. URL: https://doi.org/10.1145/2463676.2465295.

[30] Documentation x2013; Arm Developer — developer.arm.com. https://developer.arm.com/documentation/ddi0488/h/level-2-memory-system/about-the-l2-memory-system. [Accessed 22-10-2025].

[31] M. Eilers, M. Schwerhoff, A. J. Summers, and P. Müller. Fifteen years of viper. In R. Piskac and Z. Rakamarić, editors, *Computer Aided Verification (CAV)*, pages 107–123, Cham. Springer Nature Switzerland, 2025. DOI: 10.1007/978-3-031-98668-0_5. URL: https://link.springer.com/chapter/10.1007/978-3-031-98668-0_5.

[32] J. C. Filliâtre and C. Marché. The why/krakatoa/caduceus platform for deductive program verification. In *Proceedings of the 19th International Conference on Computer Aided Verification*, CAV'07, pages 173–177, Berlin, Germany. Springer-Verlag, 2007. ISBN: 9783540733676.

[33] J.-C. Filliâtre and A. Paskevich. Why3 – Where Programs Meet Provers. In *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Proceedings of the 22nd European Symposium on Programming*, Rome, Italy. Springer, Mar. 2013. URL: https://inria.hal.science/hal-00789533.

[34] H. Funke, S. Breß, S. Noll, V. Markl, and J. Teubner. Pipelined query processing in coprocessor environments. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1603–1618, Houston, TX, USA. Association for Computing Machinery, 2018. ISBN: 9781450347037. DOI: 10.1145/3183713.3183734. URL: https://doi.org/10.1145/3183713.3183734.

[35] E. Gershuni, N. Amit, A. Gurfinkel, N. Narodytska, J. A. Navas, N. Rinetzky, L. Ryzhyk, and M. Sagiv. Simple and precise static analysis of untrusted linux kernel extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 1069–1084, Phoenix, AZ, USA. Association for Computing Machinery, 2019. ISBN: 9781450367127. DOI: 10.1145/3314221.3314590. URL: https://doi.org/10.1145/3314221.3314590.

[36] H. Goldstein, J. W. Cutler, D. Dickstein, B. C. Pierce, and A. Head. Property-based testing in practice. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, Lisbon, Portugal. Association for Computing Machinery, 2024. ISBN: 9798400702174. DOI: 10.1145/3597503.3639581. URL: https://doi.org/10.1145/3597503.3639581.

[37] D. Gouk, M. Kwon, H. Bae, and M. Jung. Containerized in-storage processing model and hardware acceleration for fully-flexible computational ssds. *IEEE Computer Architecture Letters*:1–4, 2023. DOI: 10.1109/LCA.2023.3289828.

[38] B. Gu, A. S. Yoon, D.-H. Bae, I. Jo, J. Lee, J. Yoon, J.-U. Kang, M. Kwon, C. Yoon, S. Cho, J. Jeong, and D. Chang. Biscuit: a framework for near-data processing of big data workloads. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 153–165, 2016. DOI: 10.1109/ISCA.2016.23.

[39] N. Hedam. *Delilah: Efficient eBPF Offload for Integrated Data Pipelines on Computational Storage*. English. PhD thesis, IT-Universitetet i København, Denmark, Oct. 2024. ISBN: 978-87-7949-525-8.

[40] N. Hedam, M. Tychsen Clausen, P. Bonnet, S. Lee, and K. Friis Larsen. Delilah: ebpf-offload on computational storage. In *Proceedings of the 19th International Workshop on Data Management on New Hardware*, DaMoN '23, pages 70–76, Seattle, WA, USA. Association for Computing Machinery, 2023. ISBN: 9798400701917. DOI: 10.1145/3592980.3595319. URL: https://doi.org/10.1145/3592980.3595319.

[41] T. Henriksen. *Design and Implementation of the Futhark Programming Language*. PhD thesis, University of Copenhagen, Universitetsparken 5, 2100 KÃ¸benhavn, Nov. 2017.

[42] A. HeydariGorji, S. Rezaei, M. Torabzadehkashi, H. Bobarshad, V. Alves, and P. H. Chou. Leveraging computational storage for power-efficient distributed data analytics. *ACM Trans. Embed. Comput. Syst.*, 21(6), Oct. 2022. ISSN: 1539-9087. DOI: 10.1145/3528577. URL: https://doi.org/10.1145/3528577.

[43] Ieee standard for property specification language (psl) - redline. *IEEE Std 1850-2010 (Revision of IEEE Std1850-2005) - Redline*:1–188, Apr. 2010. DOI: 10.1109/IEEESTD.2010.5948310.

[44] Ieee standard for systemverilog–unified hardware design, specification, and verification language. *IEEE Std 1800-2023 (Revision of IEEE Std 1800-2017)*:1–1354, Feb. 2024. DOI: 10.1109/IEEESTD.2024.10458102.

[45] Ieee standard for vhdl language reference manual. *IEEE Std 1076-2019*:1–673, 2019. DOI: 10.1109/IEEESTD.2019.8938196.

[46] Z. István, D. Sidler, and G. Alonso. Caribou: intelligent distributed storage. *Proc. VLDB Endow.*, 10(11):1202–1213, Aug. 2017. ISSN: 2150-8097. DOI: 10.14778/3137628.3137632. URL: https://doi.org/10.14778/3137628.3137632.

[47] Y. Jin, H.-W. Tseng, Y. Papakonstantinou, and S. Swanson. Kaml: a flexible, high-performance key-value ssd. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 373–384, 2017. DOI: 10.1109/HPCA.2017.15.

[48] Y. Jin and B. Lee. Chapter one - a comprehensive survey of issues in solid state drives. In A. R. Hurson, editor. Volume 114, Advances in Computers, pages 1–69. Elsevier, 2019. DOI: https://doi.org/10.1016/bs.adcom.2019.02.001. URL: https://www.sciencedirect.com/science/article/pii/S0065245819300117.

[49] S.-W. Jun, M. Liu, S. Lee, J. Hicks, J. Ankcorn, M. King, S. Xu, and Arvind. Bluedbm: an appliance for big data analytics. *SIGARCH Comput. Archit. News*, 43(3S):1–13, June 2015. ISSN: 0163-5964. DOI: 10.1145/2872887.2750412. URL: https://doi.org/10.1145/2872887.2750412.

[50] J.-U. Kang, J. Hyun, H. Maeng, and S. Cho. The multi-streamed Solid-State drive. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*, Philadelphia, PA. USENIX Association, June 2014. URL: https://www.usenix.org/conference/hotstorage14/workshop-program/presentation/kang.

[51] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. Sel4: formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, Big Sky, Montana, USA. Association for Computing Machinery, 2009. ISBN: 9781605587523. DOI: 10.1145/1629575.1629596. URL: https://doi.org/10.1145/1629575.1629596.

[52] A. Kroviakov, P. Kurapov, C. Anneser, and J. Giceva. Heterogeneous Intra-Pipeline Device-Parallel Aggregations. In *Proceedings of the 20th International Workshop on Data Management on New Hardware*, DaMoN '24, Santiago, AA, Chile. Association for Computing Machinery, 2024. ISBN: 9798400706677. DOI:

10.1145/3662010.3663441. URL: https://doi.org/10.1145/3662010.3663441.

[53] R. Lämmel and S. P. Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '03, pages 26–37, New Orleans, Louisiana, USA. Association for Computing Machinery, 2003. ISBN: 1581136498. DOI: 10.1145/604174.604179. URL: https://doi.org/10.1145/604174.604179.

[54] R. Lane. Userspace ebpf vm. https://github.com/iovisor/ubpf, 2015.

[55] J. Lee. Hardware accelerated zfs using computational storage. 2022. URL: https://www.snia.org/educational-library/hardware-accelerated-zfs-using-computational-storage-2022.

[56] K. R. M. Leino. Dafny: an automatic program verifier for functional correctness. In E. M. Clarke and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg. Springer Berlin Heidelberg, 2010. ISBN: 978-3-642-17511-4.

[57] K. R. M. Leino. This is boogie 2. June 2008. URL: https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/.

[58] A. Lerner and P. Bonnet. Not your grandpa's ssd: the era of co-designed storage devices. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, pages 2852–2858, Virtual Event, China. Association for Computing Machinery, 2021. ISBN: 9781450383431. DOI: 10.1145/3448016.3457540. URL: https://doi.org/10.1145/3448016.3457540.

[59] A. Lerner and P. Bonnet. *Principles of Database and Solid-State Drive Co-Design*. Springer, 2025.

[60] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009. ISSN: 0001-0782. DOI: 10.1145/1538788.1538814. URL: https://doi.org/10.1145/1538788.1538814.

[61] F. Li, Y. Lu, Z. Wu, and J. Shu. Ascache: an approximate ssd cache for error-tolerant applications. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2019.

[62] F. Lombardi and R. Di Pietro. Secure virtualization for cloud computing. *Journal of Network and Computer Applications*, 34(4):1113–1122, 2011. ISSN: 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2010.06.008. URL: https://www.sciencedirect.com/science/article/pii/S1084804510001062. Advanced Topics in Cloud Computing.

[63] T. Maltenberger, I. Ilic, I. Tolovski, and T. Rabl. Evaluating multi-gpu sorting with modern interconnects. In Z. G. Ives, A. Bonifati, and A. E. Abbadi, editors, *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 1795–1809. ACM, 2022. DOI: 10.1145/3514221.3517842. URL: https://doi.org/10.1145/3514221.3517842.

[64] K. K. Matam, G. Koo, H. Zha, H.-W. Tseng, and M. Annavaram. Graphssd: graph semantics aware ssd. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pages 116–128, Phoenix, Arizona. Association for Computing Machinery, 2019. ISBN: 9781450366694. DOI: 10.1145/3307650.3322275. URL: https://doi.org/10.1145/3307650.3322275.

[65] A. E. Michael, A. Gollamudi, J. Bosamiya, E. Johnson, A. Denlinger, C. Disselkoen, C. Watt, B. Parno, M. Patrignani, M. Vassena, and D. Stefan. Mswasm: soundly enforcing memory-safe execution of unsafe code. *Proc. ACM Program. Lang.*, 7(POPL), Jan. 2023. DOI: 10.1145/3571208. URL: https://doi.org/10.1145/3571208.

[66] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: a verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 41–62, Berlin, Heidelberg. Springer Berlin Heidelberg, 2016. ISBN: 978-3-662-49122-5.

[67] G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 106–119, Paris, France. Association for Computing Machinery, 1997. ISBN: 0897918533. DOI: 10.1145/263699.263712. URL: https://doi.org/10.1145/263699.263712.

[68] H. Nicholson, A. Raza, P. Chrysogelos, and A. Ailamaki. HetCache: Synergising NVMe Storage and GPU acceleration for Memory-Efficient Analytics. In *Proceedings of the 13th Annual Conference on Innovative Data Systems Research*, CIDR '23, 2023.

[69] P. O'Neil, E. O'Neil, X. Chen, and S. Revilak. The star schema benchmark. *University of Massachusetts, Boston, Tech. Rep*, 2007.

[70] B. C. Pierce, A. Azevedo de Amorim, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjöberg, A. Tolmach, and B. Yorgey. *Program equivalence*. In *Software Foundations*. URL: https://softwarefoundations.cis.upenn.edu/plf-current/Equiv.html.

[71] D. Reimerink, S. C. Cuesta, and A. Terzolo. Verifier, Dec. 2024. URL: https://docs.ebpf.io/linux/concepts/verifier/.

[72] Z. Ruan, T. He, and J. Cong. INSIDER: designing In-Storage computing system for emerging High-Performance drive. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 379–394, Renton, WA. USENIX Association, July 2019. ISBN: 978-1-939133-03-8. URL: https://www.usenix.org/conference/atc19/presentation/ruan.

[73] ScaleFlux. Guide 2024: What is Computational Storage? White Paper, ScaleFlux, 2024. URL: https://go.pardot.com/l/970113/2024-03-13/4cp9t/970113/1710375276NYw9uVgj/Guide_2024___What_is_Computational_Storage___ScaleFlux.pdf.

[74] S. Seshadri, M. Gahagan, S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson. Willow: a user-programmable ssd. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 67–80, Broomfield, CO. USENIX Association, 2014. ISBN: 9781931971164.

[75] G. Singh, L. Chelini, S. Corda, A. Javed Awan, S. Stuijk, R. Jordans, H. Corporaal, and A.-J. Boonstra. A review of near-memory computing architectures: opportunities and challenges. In *2018 21st Euromicro Conference on Digital System Design (DSD)*, pages 608–617, 2018. DOI: 10.1109/DSD.2018.00106.

[76] U. Steinberg and B. Kauer. Nova: a microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European Conference on Computer*

Systems, EuroSys '10, pages 209–222, Paris, France. Association for Computing Machinery, 2010. ISBN: 9781605585772. DOI: 10.1145/1755913.1755935. URL: https://doi.org/10.1145/1755913.1755935.

[77] J. Teubner and R. Mueller. How soccer players would do stream joins. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 625–636, Athens, Greece. Association for Computing Machinery, 2011. ISBN: 9781450306614. DOI: 10.1145/1989323.1989389. URL: https://doi.org/10.1145/1989323.1989389.

[78] N. C. Thompson and S. Spanuth. The decline of computers as a general purpose technology. *Commun. ACM*, 64(3):64–72, Feb. 2021. ISSN: 0001-0782. DOI: 10.1145/3430936. URL: https://doi.org/10.1145/3430936.

[79] L. Thostrup, G. Doci, N. Boeschen, M. Luthra, and C. Binnig. Distributed gpu joins on fast rdma-capable networks. *Proc. ACM Manag. Data*, 1(1), May 2023. DOI: 10.1145/3588709. URL: https://doi.org/10.1145/3588709.

[80] C.-H. Wu, D.-Y. Wu, H.-M. Chou, and C.-A. Cheng. Rethink the design of flash translation layers in a component-based view. *IEEE Access*, 5:12895–12912, 2017. DOI: 10.1109/ACCESS.2017.2718559.

[81] J. Yang, R. Pandurangan, C. Choi, and V. Balakrishnan. Autostream: automatic stream management for multi-streamed ssds. In *Proceedings of the 10th ACM International Systems and Storage Conference*, SYSTOR '17, Haifa, Israel. Association for Computing Machinery, 2017. ISBN: 9781450350358. DOI: 10.1145/3078468.3078469. URL: https://doi.org/10.1145/3078468.3078469.

[82] B. Yogatama, W. Gong, and X. Yu. Scaling your Hybrid CPU-GPU DBMS to Multiple GPUs. *Proc. VLDB Endow.*, 17(13):4709–4722, Feb. 2025. ISSN: 2150-8097. DOI: 10.14778/3704965.3704977. URL: https://doi.org/10.14778/3704965.3704977.

Appendices

Appendix A

Registered Functions with ACSL Specification

A.1 Sum

```
#include <stdint.h>
#include "registered_functions.h"
/*@
  axiomatic Sum_array{
    logic integer sum(int* array, integer len) reads array[0 .. (len-1)];

    axiom empty:
      \forall int* a, integer l; 0 >= l ==> sum(a,l) == 0;

    axiom range:
      \forall int* a, integer l; 0 < l ==> sum(a,l) == sum(a,l-1)+a[l-1];
  }
*/

/*@ ghost
  /@
    requires \valid_read(src+(0..src_len-1));
    requires \valid_read(indices+(0..idx_len-1));
    requires \valid(dest+(0..idx_len-1));

    requires \separated(
      src+(0..src_len-1),
      indices+(0..idx_len-1),
      dest+(0..idx_len-1)
    );

    requires IN_BOUNDS:
      \forall integer idx;
      0 <= idx < idx_len ==> 0 <= indices[idx] < src_len;

    assigns dest[0..idx_len-1];

    ensures TOTAL:
      \forall integer idx;
      0 <= idx < idx_len ==> dest[idx] == src[indices[idx]];
  @/
```

```
  void select(
    int *src, uint32_t src_len,
    uint32_t *indices, uint32_t idx_len,
    int \ghost * dest
  ){
    /@
      loop invariant COMPLETE:
        \forall integer idx;
        0 <= idx < i ==> dest[idx] == src[indices[idx]];

      loop assigns i, dest[0..idx_len-1];

      loop variant idx_len - i;
    @/
    for(uint32_t i = 0; i<idx_len; i++) dest[i] = src[indices[i]];
} */

/*@
  requires \valid_read(indices+(0..index_count-1));
  requires \valid_read(data+(0..data_size-1));
  requires \valid(terms+(0..index_count-1));

  requires \separated(
    indices+(0..index_count-1),
    data+(0..data_size-1),
    terms+(0..index_count-1)
  );

  requires IN_BOUNDS:
    \forall integer idx;
    0 <= idx < index_count ==> 0 <= indices[idx] < data_size;

  assigns terms[0..index_count-1];

  ensures TOTAL:
    \forall integer idx;
    0 <= idx < index_count ==> terms[idx] == data[indices[idx]];

  ensures AGGREGATE: \result == sum(terms, index_count);
*/
int aggregate_by_pos(
  int *data, uint32_t data_size,
  uint32_t *indices, uint32_t index_count
) /*@ ghost (int terms[])*/ {
  int result = 0;
  uint32_t *indices_end = indices + index_count;
  //@ ghost select(data, data_size, indices, index_count, terms);
  //@ ghost uint32_t ghost_count = 0;
  /*@
    loop invariant indices + index_count - ghost_count == indices_end;
    loop invariant 0 <= ghost_count <= index_count;

    loop invariant SUM: result == sum(terms, ghost_count);

    loop assigns result, indices, ghost_count;
    loop variant index_count - ghost_count;
  */
```

```
  for (; indices != indices_end; ++indices) {
    result += data[*indices];
    //@ ghost ghost_count++;
  }
  return result;
}
```

A.2 Conversion

```
#include <stdint.h>

#define TUPLE_WIDTH 8

typedef enum {
  TO_YEAR,
  TO_MONTH,
  TO_DAY } CONVERSION_TYPE;

/*@
  predicate is_digits(char *string, integer length) =
    \forall integer i; 0 <= i < length ==> '0' <= string[i] <= '9';

  axiomatic Conversion{
    logic uint32_t int_of_string(char *string, integer length);

    axiom no_digit:
      \forall char *string; int_of_string(string, 0) == 0;

    axiom multi_digit:
      \forall char *string, integer length;
      0 <= length ==>
      int_of_string(string, length+1) ==
      int_of_string(string, length) * 10 + string[length] - '0';
  }
*/

/*@
  requires 0 <= len;
  requires \valid_read(string+(0..len-1));
  requires is_digits(string, len);

  assigns \nothing;

  ensures \result == int_of_string(string, len);
*/
inline uint32_t parse_string(char *string, int len){
  uint32_t res = 0;
  /*@
    loop invariant res == int_of_string(string, i);
    loop invariant 0 <= i <= len;
    loop assigns i, res;
    loop variant len - i;
  */
```

```
  for(int i = 0; i<len; i++) res = string[i] - '0' + res * 10;
  return res;
}

/*@
  lemma substring_digits:
    \forall char* string, integer length;
    is_digits(string, length) ==>
    \forall integer i; 0 <= i < length ==>
    is_digits(string+i, length-i);
*/

// SPECIFICATION
/*@
  requires \separated(
    date_strings+(0..bytes-1),
    results+(0..bytes/TUPLE_WIDTH-1),
    res_length
  );
  requires \valid(results+(0..bytes/TUPLE_WIDTH-1)) && \valid(res_length);
  requires \valid_read(date_strings+(0..bytes-1));

  requires is_digits(date_strings, bytes);

  requires bytes % TUPLE_WIDTH == 0;

  assigns *res_length, results[0..bytes/TUPLE_WIDTH-1];

  behavior to_year:
    assumes conv == TO_YEAR;
    ensures \forall integer idx;
      0 <= idx < bytes / TUPLE_WIDTH ==>
      results[idx] == int_of_string(date_strings+idx*TUPLE_WIDTH, 4);

  behavior to_month:
    assumes conv == TO_MONTH;
    ensures \forall integer idx;
      0 <= idx < bytes / TUPLE_WIDTH ==>
      results[idx] == int_of_string(date_strings+idx*TUPLE_WIDTH+4, 2);

  behavior to_day:
    assumes conv == TO_DAY;
    ensures \forall integer idx;
      0 <= idx < bytes / TUPLE_WIDTH ==>
      results[idx] == int_of_string(date_strings+idx*TUPLE_WIDTH+6, 2);
*/
void convert_dates(
  char *date_strings, uint32_t bytes,
  CONVERSION_TYPE conv,
  uint32_t* results, uint32_t *res_length
){
  uint32_t num_results = 0;
  char *data_ptr_end = date_strings + bytes;
  //@ ghost int entries = bytes / TUPLE_WIDTH;
  //@ ghost int remaining = bytes / TUPLE_WIDTH;
  if (conv == TO_YEAR) {
    /*@
```

```
    loop invariant remaining >= 0;
    loop invariant num_results + remaining == entries;
    loop invariant date_strings + remaining * TUPLE_WIDTH == data_ptr_end;

    loop invariant
      \forall integer i;
      0 <= i < num_results ==>
      results[i] == int_of_string(\at(date_strings, Pre)+i*TUPLE_WIDTH, 4);

    loop invariant is_digits(date_strings, remaining * TUPLE_WIDTH);

    loop assigns remaining, results[0..entries-1], num_results, date_strings;

    loop variant remaining;
  */
  for (; date_strings != data_ptr_end; date_strings += TUPLE_WIDTH) {
    results[num_results++] = parse_string(date_strings, 4);
    //@ ghost remaining--;
  }
} else if (conv == TO_MONTH) {
  /*@
    loop invariant remaining >= 0;
    loop invariant num_results + remaining == entries;
    loop invariant date_strings + remaining * TUPLE_WIDTH == data_ptr_end;

    loop invariant
      \forall integer i;
      0 <= i < num_results ==>
      results[i] == int_of_string(\at(date_strings, Pre)+i*TUPLE_WIDTH+4, 2);

    loop invariant is_digits(date_strings, remaining * TUPLE_WIDTH);

    loop assigns remaining, results[0..entries-1], num_results, date_strings;

    loop variant remaining;
  */
  for (; date_strings != data_ptr_end; date_strings += TUPLE_WIDTH) {
    results[num_results++] = parse_string(date_strings+4, 2);
    //@ ghost remaining--;
  }
} else if (conv == TO_DAY) {
  /*@
    loop invariant remaining >= 0;
    loop invariant num_results + remaining == entries;
    loop invariant date_strings + remaining * TUPLE_WIDTH == data_ptr_end;

    loop invariant
      \forall integer i;
      0 <= i < num_results ==>
      results[i] == int_of_string(\at(date_strings, Pre)+i*TUPLE_WIDTH+6, 2);

    loop invariant is_digits(date_strings, remaining * TUPLE_WIDTH);

    loop assigns remaining, results[0..entries-1], num_results, date_strings;

    loop variant remaining;
  */
```

```
    for (; date_strings != data_ptr_end; date_strings += TUPLE_WIDTH) {
      results[num_results++] = parse_string(date_strings+6, 2);
      //@ ghost remaining--;
    }
  }
  *res_length = num_results * TUPLE_WIDTH;
}
```

A.3 Filter

```
#include <stdint.h>
#include <stdlib.h>

/*@
  axiomatic Occurrences{
    logic integer occurrences_in{L1}(uint32_t* arr, integer len, uint32_t value)
      reads arr[0..len-1];

    axiom empty: \forall uint32_t* arr, uint32_t v, integer len;
      len <= 0 ==> occurrences_in(arr, len, v) == 0;

    axiom in_nonempty:
      \forall uint32_t* arr, integer len, uint32_t v; len > 0 ==>
        (arr[len-1] == v <==>
        occurrences_in(arr, len, v) ==
        1 + occurrences_in(arr, len-1, v));

    axiom notin_nonempty:
      \forall uint32_t* arr, integer len, uint32_t v;
      len > 0 ==>
        (arr[len-1] != v <==>
        occurrences_in(arr, len, v) == occurrences_in(arr, len-1, v));

    axiom occurrence_size:
      \forall uint32_t* arr, integer len, uint32_t v; len > 0 ==>
      (\exists integer k; k == occurrences_in(arr, len, v)) ==>
      \forall integer i; 0 <= i < len ==> 0 <= arr[i] <= UINT32_MAX;
  }
*/

/*@
  predicate unchanged{L1, L2}(uint32_t* array, integer len) =
    \forall integer i ; 0 <= i < len ==> \at(array[i], L1) == \at(array[i], L2);

  lemma unchanged_occurrences{L1, L2}:
    \forall uint32_t *arr, value, integer len;
    unchanged{L1, L2}(arr, len) ==>
    occurrences_in{L1}(arr, len, value) == occurrences_in{L2}(arr, len, value);
*/

/*@
  requires \valid(out+(0..len-1)) && \valid(written);
  requires \valid_read(data + (0..len-1));
```

```
  requires \separated(written, data+(0..len-1), out+(0..len-1));

  assigns *written, out[0..len-1];

  ensures Correct:
    \forall size_t j;
    0 <= j < *written ==> _c_pred1 <= out[j] <= _c_pred2;

  ensures Complete:
    \forall uint32_t value;
    _c_pred1 <= value <= _c_pred2 ==>
      occurrences_in(data, len, value) ==
      occurrences_in(out, *written, value);
*/
void
delilah_functions_tsl_filter_sequential(uint32_t* data, size_t len,
                                        uint32_t* out, size_t* written,
                                        uint32_t _c_pred1, uint32_t _c_pred2)
{
  *written = 0;
  /*@
    loop invariant 0 <= i <= len;
    loop invariant 0 <= *written <= i;
    loop invariant
      \forall size_t j;
      0 <= j < *written ==> _c_pred1 <= out[j] <= _c_pred2;

    loop invariant
      \forall uint32_t value;
      _c_pred1 <= value <= _c_pred2 ==>
      occurrences_in(data, i, value) == occurrences_in(out, *written, value);

    loop assigns i, *written, out[0..len-1];

    loop variant len-i;
  */
  for (size_t i = 0; i < len; i++) {
    const uint32_t inc = (data[i] >= _c_pred1) && (data[i] <= _c_pred2);
    out[*written] = data[i];
    *written += inc;
    //@ assert unchanged{Here, LoopCurrent}(data, i);
    /*@ ghost
      if((data[i] >= _c_pred1) && (data[i] <= _c_pred2)){
        //@ assert unchanged{Here, LoopCurrent}(out, *written-1);
        /@ assert \forall uint32_t value; _c_pred1 <= value <= _c_pred2 ==>
            occurrences_in(data, i, value) ==
            occurrences_in(out, *written-1, value); @/

      } else {
        //@ assert unchanged{Here, LoopCurrent}(out, *written);
        /@ assert \forall uint32_t value; _c_pred1 <= value <= _c_pred2 ==>
            occurrences_in(data, i, value) ==
            occurrences_in(out, *written, value); @/
      } */
  }
}
```

Appendix B

Coq Proof of Unchanged Occurrences Lemma

```
(* Why3 assumption *)
Definition P_unchanged (Mint:addr -> Numbers.BinNums.Z)
    (Mint1:addr -> Numbers.BinNums.Z) (array:addr) (len:Numbers.BinNums.Z) :
    Prop :=
  forall (i:Numbers.BinNums.Z),
  let a := shift array i in
  (0%Z <= i)%Z -> (i < len)%Z -> ((Mint1 a) = (Mint a)).

Parameter L_occurrences_in:
  (addr -> Numbers.BinNums.Z) -> addr -> Numbers.BinNums.Z ->
  Numbers.BinNums.Z -> Numbers.BinNums.Z.

Axiom Q_empty :
  forall (Mint:addr -> Numbers.BinNums.Z) (arr:addr) (v:Numbers.BinNums.Z)
    (len:Numbers.BinNums.Z),
  (len <= 0%Z)%Z -> is_uint32 v -> ((L_occurrences_in Mint arr len v) = 0%Z).

Axiom Q_in_nonempty :
  forall (Mint:addr -> Numbers.BinNums.Z) (arr:addr) (len:Numbers.BinNums.Z)
    (v:Numbers.BinNums.Z),
  let x := ((-1%Z)%Z + len)%Z in
  let x1 := Mint (shift arr x) in
  (0%Z < len)%Z -> is_uint32 v -> is_uint32 x1 ->
  ((1%Z + (L_occurrences_in Mint arr x v))%Z =
   (L_occurrences_in Mint arr len v)) <->
  (x1 = v).

Axiom Q_notin_nonempty :
  forall (Mint:addr -> Numbers.BinNums.Z) (arr:addr) (len:Numbers.BinNums.Z)
    (v:Numbers.BinNums.Z),
  let x := ((-1%Z)%Z + len)%Z in
  let x1 := Mint (shift arr x) in
  (0%Z < len)%Z -> is_uint32 v -> is_uint32 x1 ->
  ((L_occurrences_in Mint arr x v) = (L_occurrences_in Mint arr len v)) <->
  ~ (x1 = v).

Axiom Q_occurrence_size :
  forall (Mint:addr -> Numbers.BinNums.Z) (arr:addr) (len:Numbers.BinNums.Z),
```

```
542   (0%Z < len)%Z -> forall (i:Numbers.BinNums.Z),
543   let x := Mint (shift arr i) in
544   (0%Z <= i)%Z -> (i < len)%Z -> (0%Z <= x)%Z /\ (x <= 4294967295%Z)%Z.
545
546 (* Why3 goal *)
547 Theorem wp_goal :
548   forall (t:addr -> Numbers.BinNums.Z) (t1:addr -> Numbers.BinNums.Z)
549     (a:addr) (i:Numbers.BinNums.Z) (i1:Numbers.BinNums.Z),
550   is_uint32 i1 -> P_unchanged t t1 a i ->
551   ((L_occurrences_in t1 a i i1) = (L_occurrences_in t a i i1)).
552 (* Why3 intros t t1 a i i1 h1 h2. *)
553 Proof.
554 Require Import Lia.
555 intros t t1 arr len val h1 h2.
556 destruct len.
557 - repeat (rewrite Q_empty; auto). lia. lia.
558 - rename p into len.
559   assert (forall i, 0 <= i < (Z.pos len) ->
560           is_uint32 (t1 (shift arr i)))%Z as Hlen.
561   intros. unfold is_uint32. assert (0 <= t1 (shift arr i) <= 4294967295)%Z.
562   apply Q_occurrence_size with (len := Z.pos len); lia. lia.
563   induction len using Pind.
564   + unfold P_unchanged in h2.
565     assert (t1 (shift arr (-1+1)%Z) = val \/
566             t1 (shift arr (-1+1)%Z) <> val) by lia.
567     destruct H.
568     * assert (H0 := H). assert (H1 := H). rewrite h2 in H0 by lia.
569       rewrite <- Q_in_nonempty in H.
570       rewrite <- Q_in_nonempty in H0.
571       rewrite <- H0. rewrite <- H.
572       rewrite Q_empty. rewrite Q_empty.
573       all: auto.
574       all: try lia.
575       simpl in *. rewrite H0. auto.
576       simpl in *. rewrite H1. auto.
577     * assert (H0 := H). assert (H1 := H). rewrite h2 in H0 by lia.
578       rewrite <- Q_notin_nonempty in H.
579       rewrite <- Q_notin_nonempty in H0.
580       rewrite <- H0. rewrite <- H.
581       rewrite Q_empty. rewrite Q_empty.
582       all: auto.
583       all: try lia.
584       rewrite <- h2.
585       all: simpl.
586       all: try lia.
587       all: apply Hlen.
588       all: lia.
589   + assert (t1 (shift arr (-1+(Z.pos (Pos.succ len)))%Z) = val \/
590             t1 (shift arr (-1+(Z.pos (Pos.succ len)))%Z) <> val) by lia.
591     destruct H.
592     * assert (H0 := H). assert (H1 := H). rewrite h2 in H0 by lia.
593       rewrite <- Q_in_nonempty in H.
594       rewrite <- Q_in_nonempty in H0.
595       rewrite <- H0. rewrite <- H.
596       f_equal.
597       replace (- (1) + (Z.pos (Pos.succ len)))%Z with (Z.pos len) by lia.
598       apply IHlen.
```

```
      unfold P_unchanged. intros. apply h2; auto; try lia.
      intros. apply Hlen. lia. all: try lia. all: auto.
      all: replace (- (1) + (Z.pos (Pos.succ len)))%Z with (Z.pos len) by lia.
      rewrite <- h2 by lia.
      all: apply Hlen; lia.
    * assert (H0 := H). assert (H1 := H). rewrite h2 in H0 by lia.
      rewrite <- Q_notin_nonempty in H.
      rewrite <- Q_notin_nonempty in H0.
      rewrite <- H0. rewrite <- H.
      replace (- (1) + (Z.pos (Pos.succ len)))%Z with (Z.pos len) by lia.
      apply IHlen.
      unfold P_unchanged. intros. apply h2; auto; try lia.
      intros. apply Hlen. lia. all: try lia. all: auto.
      all: replace (- (1) + (Z.pos (Pos.succ len)))%Z with (Z.pos len) by lia.
      rewrite <- h2 by lia.
      all: apply Hlen; lia.
- repeat rewrite Q_empty; try lia; auto.
Qed.
```

Appendix C

Rogers Implementation

```
{-# LANGUAGE RankNTypes #-}
{-# OPTIONS_GHC -Wno-unrecognised-pragmas #-}
{-# HLINT ignore "Use tuple-section" #-}
{-# LANGUAGE FlexibleInstances #-}
module Main where

import Language.C
import Language.C.System.GCC
import Data.Data
import Data.Generics
import Data.List (isSuffixOf, find, maximumBy)
import Data.Set (Set)
import qualified Data.Set as Set
import Data.Map (Map)
import qualified Data.Map as Map
import Data.Maybe
import Language.C.Data.Ident
import Text.Printf
import System.Environment (getArgs, lookupEnv)
import Foreign (Ptr)
import Control.Concurrent (addMVarFinalizer)
import Control.Arrow ((>>>))
import Control.Monad.State
import Control.Monad.Writer.Lazy
import Control.Monad
import qualified Data.Ord as Map
import Control.Exception (assert)

filterSignature = ("filter", [
  Access Read,
  Pointer $ Access $ Write $
    CMember
    (CVar (mkIdent nopos "#0" $ head newNameSupply) undefNode)
    (mkIdent nopos "length" $ head newNameSupply) False undefNode,
  Integer,
  Integer
  ])

selectSignature = ("select",
  [Access Read,
  Access Read,
```

```
  Pointer $ Access $ Write $
    CMember
    (CVar (mkIdent nopos "#1" $ head newNameSupply) undefNode)
    (mkIdent nopos "length" $ head newNameSupply) False undefNode
  ])

readSignature = ("read_file", [
  Access Read,
  Pointer $ Access $ Write $ CConst (CIntConst (cInteger 0) undefNode)
  ])

main :: IO ()
main = do
  args <- getArgs
  let file = fromMaybe "CSFs/valid/prog0.c" $ listToMaybe args
  stream <- readInputStream file
  let ast = parseC stream $ initPos file
  case ast of
    Left err -> print err
    Right c -> do
      case findS c of
        Just struct -> do
          let dummyDevice =
                Map.fromList [
                  filterSignature,
                  selectSignature,
                  readSignature
                ]
          let body = fromJust $ findProg c
              main = findFunDefQ "main" c
              (_, state'') =
                runValid (
                  maybe (logErrM "no main function declared") validateMainM main
                ) (initialState dummyDevice struct)
          mapM_ print $ reverse $ errs state''
          let reqs = map nameFromIdent $ listify isIdIdent struct
          let body'' = reverse $ modifiedBody state''
          let body' =
                take 2 body ++
                genRequests reqs (writes state'') ++
                drop 2 body'' ++
                [last body]
          writeFile "modified.c" $
            (++) "#include <assert.h>\n" $
            show $ pretty $ changeProg body' c
        Nothing ->
          print "No layout structure defined. It must be named 'layout'"
      writeFile "simple_out_new.txt" $ show c

indexReqsExpr :: Int -> CExpression NodeInfo
indexReqsExpr i =
  CIndex
    (varExprFromString "reqs")
    (CConst (CIntConst (cInteger $ toInteger i) undefNode))
    undefNode

accessExpr expr pointer field =
```

```
  CMember expr (identFromString field) pointer undefNode

identFromString n = mkIdent nopos n $ head newNameSupply

varExprFromString n = CVar (identFromString n) undefNode

assignStmt leftExp rightExp =
  CBlockStmt (
    CExpr (Just (CAssign CAssignOp leftExp rightExp undefNode))
    undefNode)

genRequests reqNames reqSet =
  concat [
    [assignStmt
      (accessExpr (indexReqsExpr i) False "id")
      (accessExpr (varExprFromString "info") True $ reqNames !! i),
    assignStmt
      (accessExpr (indexReqsExpr i) False "p")
      (varExprFromString (if toInteger i `elem` reqSet then "Write" else "Read"))
    ] | i <- [0 .. length reqNames-1]
  ]

data RogersType =
  Unit | Byte | Integer | Float | Id | Pointer RogersType |
  Address Int | Length Int |
  Memory Placement | Layout | Request |
  Access Permission
  deriving Show
data Placement = Local | Shared deriving Show
data Permission = Read | Write CExpr deriving Show

instance Eq RogersType where
  (==) Unit Unit = True
  (==) Byte Byte = True
  (==) Integer Integer = True
  (==) Float Float = True
  (==) Id Id = True
  (==) (Address {}) (Address {}) = True
  (==) (Length {}) (Length {}) = True
  (==) (Pointer t1) (Pointer t2) = t1 == t2
  (==) (Memory {}) (Memory {}) = True
  (==) (Access {}) (Access {}) = True
  (==) Layout Layout = True
  (==) Request Request = True
  (==) _ _ = False

data AbstractState = AbstractState {
  lengthCount :: Int,
  reqCount :: Int,
  funcs :: Device,
  vars :: Variables,
  writes :: Set Integer,
  modifiedBody :: [CBlockItem],
  errs :: [String]
}
type Device = Map String [RogersType]
type Variables = Map String RogersType
```

```
type Validation = AbstractState -> AbstractState
type ErrorOr a = Either String a

newtype ValidationM a = Valid {
  runValid :: AbstractState -> (Maybe a, AbstractState)
}

instance Functor ValidationM where
  fmap f (Valid comp)= Valid $ \s ->
    let (res, s') = comp s in (fmap f res, s')

instance Applicative ValidationM where
  pure a = Valid $ \s -> (Just a, s)
  Valid trans <*> Valid comp =
    Valid $ \s ->
      let (trs, s') = trans s
          (res, s'') = comp s'
      in (trs <*> res, s'')
  Valid a *> Valid b = Valid $ \s -> let (_, s') = a s in b s'
  Valid a <* Valid b = Valid $ \s -> let (a_res, s') = a s in (a_res, snd $ b s')

instance Monad ValidationM where
  Valid comp >>= f = Valid $ \s ->
      let (res, s') = comp s in
      case res of
        Nothing -> (Nothing, s')
        Just r -> runValid (f r) s'

fromCType :: CTypeSpec -> ErrorOr RogersType
fromCType ctyp =
  case ctyp of
    CCharType _ -> Right Byte
    CIntType _ -> Right Integer
    CFloatType _ -> Right Float
    CSUType (CStruct CStructTag (Just ident) _ _ _) _ ->
      case nameFromIdent ident of
        "mem_range" -> Right $ Memory Local
        "layout" -> Right Layout
        "request" -> Right Request
        other ->
          Left $ mkErr (other ++ " is an unknown struct") ctyp
    other ->
      Left $ mkErr ((show . pretty) other ++ " is not a simple type") ctyp

addVarToContext v t s = s {vars = Map.insert v t $ vars s}

nameToType :: RogersType -> [Char] -> AbstractState -> AbstractState
nameToType Integer name state
  | isIdName name =
    let r = reqCount state in
    (addVarToContext ("info." ++ name) Id state) {reqCount = r +1 }
  | isSizeName name =
    let s = lengthCount state in
      (addVarToContext ("info." ++ name) (Length s) state) {lengthCount = s+1}
  | otherwise = addVarToContext ("info." ++ name) Integer state
```

```
nameToType t name s = addVarToContext ("info." ++ name) t s

type DeclTypingRule = String -> AbstractState -> AbstractState
type InitTriple = (Maybe CDeclr, Maybe CInit, Maybe CExpr)
attachTypeToDeclG :: DeclTypingRule -> InitTriple -> Validation
attachTypeToDeclG rule (Just cdeclr, _, _) =
  justOrErr findIdent cdeclr "no declaration found" rule

assignable c r =
  case (c, r) of
    (Integer, Id) -> True
    (Integer, Address _) -> True
    (Integer, Length {}) -> True
    (a, b) -> a == b

attachTypeToLayoutDecl :: RogersType -> InitTriple -> Validation
attachTypeToLayoutDecl typ =
  attachTypeToDeclG (
    \declr ->
      everything (.) (id `mkQ` validateContextDecl) declr . nameToType typ declr
  )

updateEnv k v = promote $ \s -> s { vars = Map.insert k v (vars s)}
removeVar k = promote $ \s -> s {vars = Map.delete k (vars s)}

attachTypeToBodyDeclM :: RogersType -> InitTriple -> ValidationM ()
attachTypeToBodyDeclM typ (Just c, Just (CInitExpr e _), _) = do
    inferredTyp <- typeExprM e
    validateM (assignable typ inferredTyp) $
      mkErr (
        printf "Declared type (%s) does not match inferred type (%s)"
        (show typ) (show inferredTyp)
      ) e
    name <- justOrErrM findIdent c "no declaration found"
    updateEnv name inferredTyp

attachTypeToBodyDeclM typ (Just c, Just i, _) = do
    logErrM $ mkErr "this type of initialization is not allowed" i

attachTypeToBodyDeclM typ (Just c, _, _) = do
  name <- justOrErrM findIdent c "no declaration found"
  updateEnv name typ

indirectionsToTypeConstructor (CPtrDeclr _ _) = return Pointer
indirectionsToTypeConstructor other =
  logErrM $ mkErr "Functions and arrays are not allowed as declarations" other

validateDeclarationM :: CDecl -> ValidationM ()
validateDeclarationM decl@(CDecl specs triples info) = do
  typ <- fromCTypeQM specs
  indirections <- findIndirectionsQM decl
  fullType <- foldM (
    \c ind -> indirectionsToTypeConstructor ind >>= \con -> return $ c . con
    ) id indirections
  mapM_ (attachTypeToBodyDeclM $ fullType typ) triples

validateDeclarationM asrt = logErrM $ mkErr "Assertations are not allowed" asrt
```

```
lookupFunc :: String -> ValidationM [RogersType]
lookupFunc name =
  Valid $ \s -> (
    Map.lookup name (funcs s),
    validate (Map.member name $ funcs s) "Attempted to call unknown function" s
  )

lengthIndex (Length i) = Just i
lengthIndex _ = Nothing

lengthMax :: AbstractState -> Int
lengthMax = maximum . (-1 :) . mapMaybe lengthIndex . Map.elems . vars

indexMatch (Address ia) (Length is) = ia == is
indexMatch (Length is) (Address ia) = ia == is
indexMatch _ _ = False

getIndexExpr (CIndex _ idx _) = Just idx
getIndexExpr _ = Nothing

trackAsWrite :: Integer -> ValidationM [a]
trackAsWrite i = Valid $ \s -> (Just [], s {writes =  Set.insert i $ writes s})

mkAssert memRange bound =
  CExpr (
    Just $
    CCall
      (varExprFromString "assert")
      [CBinary CLeqOp bound (accessExpr memRange False "length") undefNode]
      undefNode)
  undefNode

canBeTyped :: CExpr -> RogersType -> ValidationM [CStat]
canBeTyped expr (Access Read) = do
  typ <- typeExprM expr
  case typ of
    Memory Shared -> return []
    Memory Local -> do
      n <- justOrErrM nameFromVar expr "non-var local mem_range"
      ptrTyp <- typeOfVar $ printf "%s.ptr" n
      lenTyp <- typeOfVar $ printf "%s.length" n
      validateM (indexMatch ptrTyp lenTyp) $
        mkErr ("Mismatch of pointer and length in mem_range "++n) expr
      return []
    _ -> do
      validateM False "Mismatch on types in function call"
      return []

canBeTyped (CUnary CAdrOp expr _) (Pointer (Access (Write bound))) = do
  typ <- typeExprM expr
  case typ of
    Memory Shared ->
      (:) (mkAssert expr bound) <$>
      maybe (return []) trackAsWrite (getIndexExpr expr >>= getNum)
    Memory Local -> do
      n <- justOrErrM nameFromVar expr "non-var local mem_range"
```

```
      ptrTyp <- typeOfVar $ printf "%s.ptr" n
      maxI <- getFromStateM lengthMax
      validateM (ptrTyp == Address (maxI + 1))
        (n++".ptr does not point to a writable region")
      updateEnv (printf "%s.length" n) (Length (maxI + 1))
      return []
    _ -> do
      validateM False "Mismatch on types in function call"
      return []

canBeTyped e t = do
  et <- typeExprM e
  validateM (et == t) $
    mkErr (
      printf "Mismatch on types in function call, %s does not have type %s"
      (show $ pretty e) (show t)
    ) e
  return []

typeExprM :: CExpr -> ValidationM RogersType
typeExprM (CConst c) =
  case c of
    CIntConst _ _ -> return Integer
    CCharConst _ _ -> return Byte
    CFloatConst _ _ -> return Float
    CStrConst _ _ -> logErrM $ mkErr "strings are not allowed" c

typeExprM (CUnary CAdrOp expr info) =  Pointer <$> typeExprM expr

typeExprM (CIndex eArr eIdx info) = do
  name <- justOrErrM nameFromVar eArr "Only 'resources' can be accessed"
  validateM (name == "resources")
    ("Only 'resources' can be accessed, not " ++ name)
  idx <- justOrErrM getNum eIdx "Only constants can be used to access resources"
  typeOfVar $ printf "%s[%d]" name idx

typeExprM (CVar ident info) = typeOfVar $ nameFromIdent ident

typeExprM (CCast decl (CVar ident _) info) = do
  validateM (nameFromIdent ident == "context") "Only context can be cast from"
  cast_type <- justOrErrM findCTypeQ decl "Cast with no type"
  indirs <- justOrErrM findIndirectionsQ decl "Not a standard declaration"
  validateM (isSimplePtr indirs) $
    mkErr "Only casting to layout pointers are allowed" info
  rogers_type <- fromCTypeQM cast_type
  case rogers_type of
    Layout -> return $ Pointer Layout
    other -> logErrM $ mkErr "Non-layout cast" info

typeExprM (CMember expr ident deRef info) = do
  struct_name <-
    justOrErrM nameFromVar expr "Attempted access on non-var expression"
  typeOfVar (makeMemberName struct_name $ nameFromIdent ident)

typeExprM (CBinary CAddOp e1 e2 info) = do
  t1 <- typeExprM e1
  t2 <- typeExprM e2
```

```
  case t1 of
    Address i -> do
      validateM (indexMatch t1 t2) "mismatching types in addition"
      return (Address $ i+1)
    Length i -> do
      validateM (indexMatch t1 t2) "mismatching types in addition"
      return (Address $ i+1)
    _ -> do
      validateM (t1 == t2) "mismatching types in addition"
      return t1

typeExprM (CBinary op e1 e2 info) =
  if op `elem` [CMulOp, CDivOp, CSubOp]
  then do
      t1 <- typeExprM e1
      t2 <- typeExprM e2
      validateM (t1 == t2) $
        mkErr "Binary operation on two different types" info
      return t2
  else logErrM $ mkErr "Invalid operation" info

typeExprM e@(CCall (CVar ident _) args info) =
  let funname = nameFromIdent ident in
  if funname == "allocate_shared"
  then do
    validateM (length args == 2) "Wrong number of arguments to allocate shared"
    lengthTyp <- typeExprM (head args)
    validateM (lengthTyp == Integer) "Only integer lengths can be allocated"
    memTyp <- typeExprM (last args)
    validateM (memTyp == Pointer (Memory Local))
      "Second arg must be a reference to a non-shared memory region"
    field <- justOrErrM findIdent (head args) ""
    memName <- justOrErrM findIdent (last args) ""
    updateStateM [(memName, Memory Shared), (memName++".length", Integer)]
    removeVar $ memName++".ptr"
    return Id
  else do
    signature <- lookupFunc funname
    asserts <-  concat <$> zipWithM canBeTyped args signature
    mapM_ (emit . CBlockStmt) $ replaceRefs args asserts
    return Unit

typeExprM other =
  logErrM $ mkErr ("unrecognized expression " ++ show (pretty other)) other

substRef args expr@(CVar ident _) =
  if head (nameFromIdent ident) == '#'
  then args !! read (tail $ nameFromIdent ident)
  else expr
substRef _ e = e

replaceRefs :: [CExpr] -> [CStat] -> [CStat]
replaceRefs args = everywhere (mkT $ substRef args)

emit item = promote $ \s -> s { modifiedBody = item : modifiedBody s}

type Typing = AbstractState -> ErrorOr RogersType
```

```
typeExpr :: CExpr -> Typing
typeExpr (CConst c) _ =
  case c of
    CIntConst _ _ -> Right Integer
    CCharConst _ _ -> Right Byte
    CFloatConst _ _ -> Right Float
    CStrConst _ _ -> Left $ mkErr "strings are not allowed" c

typeExpr other _ =
  Left $ mkErr ("unrecognized expression " ++ show (pretty other)) other

findCTypeQ :: Data a => a -> Maybe CTypeSpec
findCTypeQ = something (mkQ Nothing Just)

fromCTypeQ :: GenericQ (Maybe (ErrorOr RogersType))
fromCTypeQ = something $ Nothing `mkQ` (Just . fromCType)

attempt comp a msg = do
  res <- justOrErrM comp a msg
  case res of
    Left err -> logErrM err
    Right v -> return v

fromCTypeQM a = attempt fromCTypeQ a "no type detected"

typeFromDecls :: CDecl -> Validation
typeFromDecls decl@(CDecl declSpecs [
  (Just
    (CDeclr (Just (Ident "dynamic" _ _) )[CArrDeclr [] (CNoArrSize _) _] _ _ _),
    Nothing,
    Nothing)
  ] _)
  state =  updateStateT (Right [("info.dynamic", Address 0)]) state

typeFromDecls decl@(CDecl declSpecs initDecls _) state =
    case fromCType <$> findCTypeQ decl of
      Nothing -> logErr state $ mkErr "Nothing found" decl
      Just errOrTyp ->
        case errOrTyp of
          Left err -> logErr state err
          Right typ ->
            foldl (\f d -> attachTypeToLayoutDecl typ d . f) id initDecls state

updateStateT :: ErrorOr [(String, RogersType)] -> Validation
updateStateT =
  either
    (\err s -> s {errs = err : errs s})
    (flip $ foldl (\s' (v,t) -> s' { vars = Map.insert v t (vars s') }))

reverseDecs :: [CDecl] -> [CDecl]
reverseDecs = reverse

initialState :: Data a => Device -> CStructureUnion a -> AbstractState
initialState devFuncs struct@(CStruct _ (Just ident) _ _ _) =
  let struct' = everywhere (mkT reverseDecs) struct in
  everything (.) (id `mkQ` typeFromDecls) struct' $
```

```
    AbstractState 0 0 devFuncs Map.empty Set.empty [] []

logErr :: AbstractState -> String -> AbstractState
state `logErr` err = state {errs = err:errs state}

validate :: Bool -> String -> Validation
validate check errMsg state
  | check = state
  | otherwise = state `logErr` errMsg

validateContextDecl :: CDerivedDeclr -> Validation
validateContextDecl (CPtrDeclr _ info) =
  validate False (mkErr "Pointers are not allowed as part of the layout" info)
validateContextDecl (CFunDeclr _ _ info) =
  validate False (mkErr "Function are not allowed as part of the layout" info)
validateContextDecl (CArrDeclr _ _ info) =
  validate False (mkErr "Arrays are not allowed as part of the layout" info)

makeMemberName struct field = struct ++ "." ++ field

promote :: (AbstractState -> AbstractState) -> ValidationM ()
promote v = Valid $ \s -> (return (), v s)

validateBlockItemM :: CBlockItem -> ValidationM ()
validateBlockItemM (CNestedFunDef fdef) =
  logErrM $ mkErr "Nested function definitions are not allowed" fdef
validateBlockItemM b@(CBlockDecl decl) = validateDeclarationM decl >> emit b
validateBlockItemM b@(CBlockStmt stmt) = validateStatementM stmt >> emit b

validateAssignmentM :: CExpr -> CExpr -> ValidationM ()
validateAssignmentM (CMember struct field deref info) rexp =
  let fieldName = nameFromIdent field in do
  struct_typ <- typeExprM struct
  struct_name <- justOrErrM findIdent struct "Unnamed struct"
  assigned_typ <- typeExprM rexp
  dest_typ <-
    case struct_typ of
      Pointer Layout -> typeOfVar (makeMemberName "info" fieldName)
      Memory Local ->
        case fieldName of
          "ptr" -> return (Address $ -1)
          "length" -> return (Length $ -2)
      _ ->
        logErrM $
          mkErr ("Assignment to unknown struct " ++ show struct_typ) struct
  case dest_typ of
    Length _ -> do
      v <- getFromStateM vars
      validateM (not $ Map.member (printf "%s.length" struct_name) v) $
        printf "Cannot overwrite existing length of %s" struct_name
    _ -> validateM True ""
  validateM (assignable dest_typ assigned_typ) $
    mkErr (
      printf "mismatch of assigned types %s and %s"
      (show dest_typ) (show assigned_typ)
    ) info
  updateEnv (makeMemberName struct_name fieldName) assigned_typ
```

```
validateStatementM :: CStat -> ValidationM ()
validateStatementM (CExpr (Just (CAssign CAssignOp e1 e2 _)) info) =
  validateAssignmentM e1 e2
validateStatementM (CExpr (Just call@(CCall {})) _) = do
  typeExprM call
  return ()
validateStatementM s = logErrM $
    mkErr ("This type of statement is not allowed ("++ show (pretty s) ++")") s

isSimplePtr [CPtrDeclr _ _] = True
isSimplePtr _ = False

mkErr :: Pos a => String -> a -> String
mkErr msg node = msg ++ ": at line " ++ (show . posRow . posOf) node

nameFromVar (CVar ident _) = Just $ nameFromIdent ident
nameFromVar _ = Nothing

findS :: CTranslUnit -> Maybe CStructUnion
findS = something $ mkQ Nothing findCS

findCS :: CStructUnion -> Maybe CStructUnion
findCS s@(CStruct CStructTag (Just id@(Ident "layout" _ _)) (Just _) _ _) =
  Just s
findCS _ = Nothing

findProg :: Data a => a -> Maybe [CBlockItem]
findProg = something $ mkQ Nothing findBlock

findBlock :: CStat -> Maybe [CBlockItem]
findBlock p@(CCompound _ stmts _) = Just stmts
findBlock _ = Nothing

changeProg :: Data a1 => [CBlockItem] -> a1 -> a1
changeProg newProg = everywhere (mkT $ changeBlock newProg)

changeBlock newprog p@(CCompound ids _ info) = CCompound ids newprog info
changeBlock _ other = other

findIndirectionsQ :: Data a => a -> Maybe [CDerivedDeclr]
findIndirectionsQ = something $ mkQ Nothing Just

findIndirectionsQM :: (Pos a, Data a) => a -> ValidationM [CDerivedDeclr]
findIndirectionsQM decl =
  justOrErrM findIndirectionsQ decl "Not a standard declaration"

validateCastM :: CBlockItem -> ValidationM ()
validateCastM cstat@(CBlockDecl decl) = do
    validateDeclarationM decl
    name <- justOrErrM (search findDeclaredIdent) decl
      "No name found as part of declaration"
    validateM (name == "info") $
      mkErr ("layout pointer must be named 'info', but is named "++name) decl
    info_typ <- typeOfVar name
    validateM (info_typ == Pointer Layout) "first statement must be layout cast"
```

```
typeOfVar :: String -> ValidationM RogersType
typeOfVar v =
  Valid $ \s -> (
    Map.lookup v (vars s),
    validate (Map.member v $ vars s) ("Attempted to lookup unknown var " ++ v) s
  )

validateM :: Bool -> String -> ValidationM ()
validateM check errMsg
  | check = return ()
  | otherwise = logErrM errMsg

isReturn (CBlockStmt (CReturn _ info)) = True
isReturn _ = False

getFromStateM :: (AbstractState -> a) -> ValidationM a
getFromStateM f = Valid $ \s -> (Just $ f s, s)

validateMainM :: CFunDef -> ValidationM ()
validateMainM (CFunDef cdeclSpecs decl _ stmt info) =
      do
        indirections <- justOrErrM findIndirectionsQ decl "impossible"
        validateM (length indirections == 1) "not a simple function"
        case head indirections of
          CFunDeclr (Right ([decl], _)) _ _ ->
            do
              typ <- justOrErrM findCTypeQ decl "missing pointer type"
              indirs <- justOrErrM findIndirectionsQ decl "incorrect pointer type"
              validateM (isSimplePtr indirs) "not a simple pointer"
              validateM (case typ of {CVoidType _ -> True; _ -> False})
                "argument is not void"
              prog <- justOrErrM findProg stmt "no program body"
              validateM (length prog >= 4)
                "program does not contain all the required statements"
              validateCastM $ head prog
              validateRequestM $ head $ tail prog
              validateObtainM $ head $ tail $ tail prog
              mapM_ emit $ take 3 prog
              validateM (isReturn $ last prog) "last statement must be a return"
              mapM_ validateBlockItemM $ tail $ tail $ tail $ init prog
          other ->
            validateM False $
              mkErr "Main must take exactly one void-pointer argument" other

getFArgs :: CExpr -> Maybe (String, [CExpr])
getFArgs (CCall (CVar fname _) args info) = Just (nameFromIdent fname, args)
getFArgs _ = Nothing

validateObtainM :: CBlockItem -> ValidationM ()
validateObtainM resCall = do
  name <- justOrErrM (search findDeclaredIdent) resCall
    "No name found as part of declaration"
  validateM (name == "resources") "Shared resources must be named 'resources'"
  indirs <- findIndirectionsQM resCall
  validateM (length indirs == 1) "not a simple pointer"
  res_typ <- justOrErrM findCTypeQ resCall "Declaration with no type"
  res_rogers_typ <- fromCTypeQM res_typ
```

```
  validateM (res_rogers_typ == Memory Local)
    "The result of obtain must be stored as a mem_range pointer"
  (fname, params) <- justOrErrM  (something $ mkQ Nothing getFArgs) resCall
    "Obtain call expected"
  validateM (fname == "obtain") "not a call to obtain"
  validateM (length params == 2) "Obtain expected 2 arguments"
  let [reqs, nums] = params
  reqsName <- justOrErrM nameFromVar reqs "ill-formatted call"
  validateM (reqsName == "reqs") "ill-formatted call"
  req_nums <- justOrErrM getNum nums "ill-formatted call"
  layout_nums <- getFromStateM reqCount
  validateM (fromInteger req_nums == layout_nums)
    "Number in obtain call must match number of requests defined in layout"
  updateStateM
    [(printf "resources[%d]" i, Memory Shared) | i <- [0..layout_nums-1]]

updateStateM = promote . updateStateT . Right

findArraySize :: CArraySize NodeInfo -> Maybe CExpr
findArraySize (CArrSize _ exp) = Just exp
findArraySize _ = Nothing

getNum :: CExpr -> Maybe Integer
getNum (CConst c) =
  case c of
    CIntConst i _ -> Just $ getCInteger i
    CCharConst c _ -> Just $ getCCharAsInt c
    _ -> Nothing

validateRequestM :: CBlockItem -> ValidationM ()
validateRequestM reqDef = do
    name <- justOrErrM (search findDeclaredIdent) reqDef
      "No name found as part of declaration"
    validateM (name == "reqs") $
      mkErr ("request array must be named 'reqs', but is named "++name) reqDef
    indirs <- findIndirectionsQM reqDef
    validateM (length indirs == 1) "not a simple array"
    reqSize <- justOrErrM (something $ mkQ Nothing findArraySize) reqDef
      "reqs must be an array"
    numReqs <- justOrErrM getNum reqSize
      "The declared size of reqs must be a constant"
    reqsInLayout <- getFromStateM reqCount
    validateM (fromInteger numReqs == reqsInLayout)
      "Reqs size does not correspond to number of reqs defined in the layout"
    req_typ <- justOrErrM findCTypeQ reqDef
      "Declaration with no type"
    req_rogers_typ <- fromCTypeQM req_typ
    validateM (req_rogers_typ == Request)
      "second statement must be request declaration"


justOrErr :: Pos a =>
  (a -> Maybe b) -> a -> String -> (b -> Validation) -> Validation
justOrErr operation item errMsg continuation =
  case operation item of
    Just res -> continuation res
    Nothing -> \s -> logErr s $ mkErr errMsg item
```

```
logErrM :: String -> ValidationM a
logErrM err = Valid $ \s -> (Nothing, s { errs = err : errs s })

justOrErrM :: Pos a => (a -> Maybe b) -> a -> String -> ValidationM b
justOrErrM operation item errMsg =
  case operation item of
    Just res -> return res
    Nothing -> logErrM $ mkErr errMsg item

findFunDefQ :: Data a => String -> a -> Maybe CFunDef
findFunDefQ name =
  something $ mkQ Nothing (
    \def@(CFunDef _ declr _ _ _) -> (
      \ident -> if name == ident then Just def else Nothing
    ) =<< findIdent declr
  )

nameFromIdent (Ident name _ _) = name

findIdent :: Data a => a -> Maybe String
findIdent = something (Nothing `mkQ` (Just . nameFromIdent))

findDeclaredIdent :: CDeclr -> String
findDeclaredIdent (CDeclr (Just (Ident name _ _)) _ _ _ _) = name

search :: Typeable b => (b -> u) -> GenericQ (Maybe u)
search match = something $ mkQ Nothing (Just . match)

isSizeName = isSuffixOf "_offset"
isIdName = isSuffixOf "_req"
isIdIdent (Ident name _ _) = "_req" `isSuffixOf` name
```