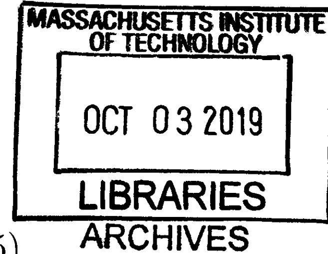
Verification of Correctness Properties of Programs that Read Input Files


MASSACHUSETTS INSTITUTE OF TECHNOLOGY
OCT 03 2019
LIBRARIES
ARCHIVES

by

Deokhwan Kim

B.S., Seoul National University (2005)
M.S., Seoul National University (2007)
S.M., Massachusetts Institute of Technology (2011)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2019

© Massachusetts Institute of Technology 2019. All rights reserved.

Signature redacted

Author ....
Department of Electrical Engineering and Computer Science
August 30, 2019

Signature redacted

Certified by...... ....
Martin C. Rinard
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Signature redacted

Accepted by........ ....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Verification of Correctness Properties of Programs that Read Input Files

by

Deokhwan Kim

Submitted to the Department of Electrical Engineering and Computer Science
on August 30, 2019, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering

Abstract

This thesis presents new techniques for verifying correctness properties of programs that process input files. These techniques apply to programs written in standard programming languages such as C and focus on relationships that must hold between program execution points, the current location of file position indicator of the open input file, and the contents of the input file. The thesis presents a specification language that developers can use to express these relationships and insert them into the program as assertions involving the file position indicator and file contents at different program points. It also presents a program verification system that verifies, for all possible input files and all possible input file contents, that the assertions hold in all program executions. The soundness of the verification system has been proved, based on the formal definition of the syntax and semantics of the specification language. The system synthesized verification conditions from the specifications for a PNG image viewer and a JPEG image converter, and successfully verified all of them.

Thesis Supervisor: Martin C. Rinard
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank my advisor, Martin Rinard. He provided me with tremendous support and encouragement throughout my extended PhD studies. I was really lucky to study under his great supervision for a long time. His insight, understanding and patience were all there whenever I needed them. It was also a really great opportunity and pleasure to be able to closely observe his bravery and skill in persuading others of his opinion on research and life, especially when it was contrary to popular belief.

I received a lot of help and advice from my thesis committee members, Armando Solar-Lezama and Arvind. Starting with a very early version of the thesis, they read a number of drafts carefully and provided valuable feedback on them. That helped me to improve my thesis in many ways.

I am so grateful to my former advisor, Kwangkeun Yi, for guiding me into this interesting field of computer science. He also had a great effect on my way of thinking and my attitude to research.

I want to say thank you to all the former and current members of Martin's group. Thanks to them, I had a good time at MIT. It was a blessing that I could work and hang out with them. In particular, I have spent most of my time at MIT in our office, Room 32-G730. The quality of both of my research and life heavily depended on my office mates. For the first half of my MIT years, my office mates were Michael Carbin, Sasa Misailovic, and Fan Long. Michael showed me how to collaborate with others in research and effectively distribute tasks to achieve best results. Sasa was not only highly motivated all the time but also continually motivated me. Fan's expertise on LLVM was a great help to me many times. I was lucky to spend the second half of my degree course with Sara Achour and Jiasi Shen. Sara let me see what it is like to fully concentrate on a chosen task. Jiasi kindly and patiently answered all my questions and helped me a lot in preparing for my thesis defense. Besides those office mates, Stelios Sidiroglou-Douskos gave me a lot of advice on troubles that I underwent as a graduate student, and his jokes were a tonic for my life. Michael Gordon showed me how to collaborate well with colleagues in the industrial world as well as the academic

world. Phillip Stanley-Marbell patiently introduced me to his research topics, so that I could understand them even though I was a layman in the topics. Jeff Perkins readily made himself available whenever I needed his help. Eric Lahtinen always treated me in a friendly way, showing his great smile. I am also grateful to the other members for making the enjoyable working environment: José Cambronero, Jürgen Cito, Thurston Dang, Anthony Eden, Jordan Eikenberry, Austin Gadient, Vijay Ganesh, Shivam Handa, Fereshte Khani, Youry Khmelevsky, Christopher Musco, Paolo Piselli, Zichao Qi, Julia Rubin, Malavika Samak, Yichen Yang, Adam Yedidia, Karen Zee, Damien Zufferey, and others. In retrospect, all the members of our group have certainly been a positive influence on me. I hope I exerted a favorable influence on them, too.

I sincerely appreciate the assistance of all administrative staff of MIT. In particular, I would like to thank Mary McDavitt for making every effort for me to just concentrate on my studies. Thankfully, Janet Fischer managed my last-minute submission of documents with patience.

Finally, I would like to express my heartfelt thanks to my parents, Indong Kim and Choonja Kim, and siblings, Hunhae Kim and Kyounghui Kim. It is their endless love and support that makes all things possible.

Contents

1 Introduction **13**

1.1 Motivating Example 15

1.1.1 Program 15

1.1.2 Correctness Properties 17

1.1.3 Verification 20

1.2 Parser Generators 22

1.3 Contributions 23

1.4 Structure 24

2 Portable Network Graphics **25**

2.1 The PNG Format 25

2.2 A PNG Image Viewer 26

2.3 Input Specification 32

2.4 Verification 34

3 Core Language **51**

3.1 Syntax 51

3.2 Dynamic Semantics 52

3.2.1 Semantic Domains 52

3.2.2 Semantics of Expressions 53

3.2.3 Semantics of Statements 53

3.2.4 Semantics of Programs 62

4 Specification Language 65

4.1 Syntax 65

4.2 Semantics 66

5 Predicate Transformer Semantics 71

5.1 Predicate Transformer Semantics 71

5.2 Properties 84

5.3 Alias Axioms 90

5.3.1 Alias Analysis 90

5.3.2 Alias Axioms 91

6 JPEG File Interchange Format 95

6.1 The JFIF Format 95

6.2 A Resilient JPEG Image Converter 97

6.3 Input Specification 105

6.4 Verification 107

7 Implementation 109

8 Related Work 111

9 Conclusion 115

List of Figures

1-1 An example file format. 15
1-2 An example program. 16
1-3 The structure of the verification system. 20

2-1 The structure of a typical PNG image file. 26
2-2 The common structure of PNG chunks. 26
2-3 A (simplified) code snippet from an image viewer for PNG files (1/2). 27
2-4 A (simplified) code snippet from an image viewer for PNG files (2/2). 29

3-1 The syntax of an imperative language. 52
3-2 The semantic domains. 53
3-3 The semantics of arithmetic expressions. 54
3-4 The semantics of Boolean expressions. 54
3-5 The semantics of a sequence of statements. 54
3-6 The semantics of environment/memory-related statements. 56
3-7 The semantics of file-related statements. 58
3-8 The semantics of an `if` statement. 60
3-9 The semantics of a `while` statement. 61

4-1 The syntax of a specification language. 66
4-2 The semantics of Term. 67
4-3 The semantics of Mem_s. 68
4-4 The semantics of FilePos_s. 68
4-5 The semantics of FileCont_s. 68

4-6 The semantics of Formula. 69

5-1 The MustAlias axioms. 91
5-2 The NoAlias axioms. 93

6-1 The layout of a typical JFIF image file. 96
6-2 The common structure of JFIF marker segments. 97
6-3 A (simplified) code snippet for an image converter from the JFIF format to the PGM format (1/2). 98
6-4 A (simplified) code snippet for an image converter from the JFIF format to the PGM format (2/2). 99

List of Tables

5.1 Predicate transformer semantics (1/2). 72
5.2 Predicate transformer semantics (2/2). 73

7.1 Verification Times. 110

Chapter 1

Introduction

Essentially all programs process inputs. Many programs read inputs from files and must process the contents of the files properly to produce the correct output. Despite the ubiquity of input file processing code, there is currently no standard, widely accepted methodology for developing or structuring this kind of code. Indeed, most input processing code is manually developed for the specific input files at hand. Unsurprisingly, given the complexity of the input files that modern software systems must process, the input file processing code is a prominent and increasingly troublesome source of errors and security vulnerabilities in modern software systems [31, 40, 42].

Program verification provides one longstanding answer to the problem of incorrect programs. Program verification has been the focus of intensive research for multiple decades [24, 45]. The field has now progressed to the point where researchers are able to verify programs that implement significant functionality. For example, the verified CompCert compiler for C is one prominent and visible milestone in the development of the field [29].

Interestingly enough, however, despite the importance of file processing code and despite the sustained resources and research effort invested into program verification, there has been very little research into verifying programs that process input files. Indeed, the verified CompCert compiler mentioned above had an *unverified* parser — the verified part of the compiler only started *after* the program text had been converted to internal data structures. And interestingly enough, some errors have

been reported in the unverified front-end part of CompCert [46]. These facts highlight the lack of research in this area and the potential for software errors in file reading code to undermine the integrity of an otherwise fully verified system.

This thesis presents new techniques for verifying correctness properties of programs that process input files. These techniques apply to programs written in standard programming languages such as C and focus on relationships that must hold between program execution points, the current location of file position indicator of the open input file, and the contents of the input file. The thesis presents a specification language that developers can use to express these relationships and insert them into the program as assertions involving the file position indicator and file contents at different program points. It also presents a program verification system that verifies, for all possible input files and all possible input file contents, that the assertions hold in all program executions.

I note that many input files have complicated formats and that the distinction between valid files that conform to the format and corrupted files that do not conform to the format is important for many developers. I emphasize that the presented techniques do not explicitly take any file format information into account and do not explicitly differentiate between valid and corrupt files. They instead verify that the assertions hold for arbitrary file contents, whether valid or not. This property supports the verification of programs that apply sophisticated recovery techniques when encountering corrupted files. Indeed, such recovery techniques, which often attempt to maximize the amount of data in the file that the program processes, can be quite complex, difficult to test, and difficult to get right, and therefore the source of many obscure errors and security vulnerabilities [33]. I therefore see the presented techniques as particularly appropriate for ensuring important correctness properties of these programs, which attempt to process the input file as resiliently as possible.

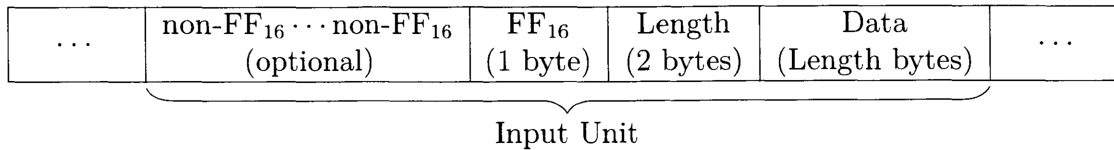


Figure 1-1: An example file format.

1.1 Motivating Example

I next present a motivating example that highlights several key characteristics of the approach and the properties that it aims to verify. The example program reads a file with format summarized in Figure 1-1. Specifically, the file comprises a sequence of *input units.* Each input unit consists of an optional sequence of non-FF_{16} bytes followed by a FF_{16} byte. The next two bytes are a length field that specify the length of a data field, which immediately follows the length field. This format contains two features that are designed to enable the program to navigate the input file:

- **Delimiters:** The FF_{16} marker is a *delimiter* that enables the program to locate the start of the input unit.
- **Lengths:** The length field specifies the length of the data field and enables the program to be sure that it reads the entire data field.

Because they are so useful, delimiters and length fields are both common features of input files. They can be especially important for helping programs resiliently process partially corrupted files — the FF_{16} delimiter can help a program find the start of the next input unit and successfully process that input unit even when a previous input unit is corrupted; the length field can enable the program to skip an appropriate number of bytes if it encounters an error in the middle of processing a data field.

1.1.1 Program

Figure 1-2 on page 16 presents an example program that reads files in this format. The program is written in C and uses the following two file operations:

```
1   uint8_t ff;
2   uint16_t length;
3   uint8_t *data;

4   ssize_t nbytes;

5   for (;;) {
6     for (;;) {
7       nbytes = read(fd, &ff, sizeof(ff));
8       if (nbytes != sizeof(ff)) goto EXIT;
9       if (ff == 0xFF) break;
10    }

11  START:
12    ASSERT(GET_CONTENT(fd, GET_POS(fd, "CURRENT") - 1) == 0xFF);

13    nbytes = read(fd, &length, sizeof(length));
14    if (nbytes != sizeof(length)) goto EXIT;
15    length = ntohs(length);

16    if ((data = malloc(length)) != NULL) {
17      nbytes = read(fd, data, length);
18      if (nbytes == length) {
19        // process data.
20        free(data);
21      } else if (0 <= nbytes && nbytes < length) {
22        free(data);
23        if (lseek(fd, length - nbytes, SEEK_CUR) == -1) goto EXIT;
24      } else {  // nbytes == -1
25        free(data);
26        goto EXIT;
27      }
28    } else {
29      if (lseek(fd, length - nbytes, SEEK_CUR) == -1) goto EXIT;
30    }

31    ASSERT(GET_POS(fd, "CURRENT) == GET_POS(fd, "START") + 2 +
32           (GET_CONTENT(fd, GET_POS(fd, "START")) * 256 +
33            GET_CONTENT(fd, GET_POS(fd, "START") + 1)));
34  }

35  EXIT:
36  ...
```

Figure 1-2: An example program.

- **Read:** The `read(fd, buf, len)` operation attempts to read `len` bytes from the open file `fd` into the buffer `buf`. It returns the number of bytes actually read or 0 if `fd` is at the end of the file. The file position indicator is increased by the number of bytes actually read. If an error occurs during reading, -1 is returned.

- **Lseek:** The `lseek(fd, len, SEEK_CUR)` operation sets the file position indicator to the current file position indicator plus `len`. If successful, it returns the new file position indicator. Otherwise it returns -1.

The outer loop (lines 5–34) iterates over the input units. This is a common coding pattern in many input file processing programs [33] — even though the file is ostensibly a stream of bytes, the program interprets the stream of bytes as a sequence of input units and contains an outer loop that iterates over the input units.

The first inner loop (lines 6–10) skips the non-FF_{16} bytes at the start of the next input unit, leaving the file position indicator referencing the first byte of the length field.

The program then reads the length field (line 13) and invokes `ntohs` to convert the input bytes in the length field into a number in the program (line 15). The following block of code (lines 16–30) uses the `read` system call to read the bytes in the data field into the `data` variable in the program.

In general, the program attempts to process the input file resiliently by skipping over problematic input units and proceeding on to the next iteration of the loop to process the next input unit when possible. If the program encounters an error from which it cannot recover, it exits the outer loop and proceeds along with the computation. Examples of errors from which it cannot recover include an inability to read potential delimiter bytes (line 8), an inability to read the length field (line 14), or an inability to skip over the appropriate number of data field bytes if something goes wrong when attempting to read the data field (lines 23 and 29).

1.1.2 Correctness Properties

I identify two correctness properties: 1) at the program point before the program reads the length field (line 12), the byte just before the current file position indicator has

value FF_{16} and 2) the difference between the file position indicators before the program reads the length field (line 12) and after attempting to read the data field (line 31) is two bytes (the size of the length field) plus the number of bytes specified by the length field.

The program expresses both of these correctness properties as assertions embedded in the program. The first `ASSERT` statement (line 12) ensures that the program correctly skips any non-FF_{16} bytes before the FF_{16} delimiter, leaving the file position indicator correctly referencing the the length field starting with the next byte in the file after the FF_{16} delimiter.

```
12   ASSERT(GET_CONTENT(fd, GET_POS(fd, "CURRENT") - 1) == 0xFF);
```

The second `ASSERT` statement (lines 31–33) ensures that the current file position indicator equals the file position indicator from line 12 (labeled with `START`) plus 2 (the number of bytes in the length field) plus the length of the data field.

```
31   ASSERT(GET_POS(fd, "CURRENT) == GET_POS(fd, "START") + 2 +
32           (GET_CONTENT(fd, GET_POS(fd, "START")) * 256 +
33            GET_CONTENT(fd, GET_POS(fd, "START") + 1)));
```

The assertions use the `GET_POS(fd, label)` construct to reference the file position indicator for the open file `fd` at program point `label`. Here label `CURRENT` references the current program point. In general, assertions may be executed multiple times, in which case the referenced file position indicator is the value from the most recently executed program point with the corresponding label. The assertions also use the `GET_CONTENT(fd, offset)` construct, which denotes the byte content of the open file `fd` at offset `offset`.

These two assertions capture the kind of correctness properties that often occur in programs that process input files and the kind of properties that the techniques presented in this thesis are designed to verify — the first involves the contents of the input file at a point in the file defined relative to the file position indicator at a

specific program point; the second involves a relationship between the file position indicators at two program points as defined, in part, by the contents of the input file (again at a point in the file defined relative to the file position indicator at a specific program point).

Verifying these properties is especially important for ensuring that programs process corrupted files properly or properly handle rare events such as the file read operation returning fewer bytes than requested. While testing can often help validate that the program correctly processes common case input files that conform to the file format, programs often contain defects that are triggered by overlooked cases or corrupt input files. One class of defects can be caused, for example, by a developer simply assuming that a delimiter occurs at the current file position instead of writing code to search for the delimiter. Verifying the first assertion can ensures that the example program does not have this defect.

Another common class of defects can be caused by a failure to advance the file position indicator to the end of an input unit when the program encounters an unexpected condition while reading an input unit. In this case the program can become *desynchronized* with the file and may start to process the next input unit from the wrong location in the file. An examination of the code in lines 16–30 highlights how complex handling correctly every case can become. Even though the code only reads the data field, it must consider and correctly handle multiple uncommon error cases:

- **Failed Memory Allocation.** If the data field is too large, the memory allocation on line 16 may fail. In this case the program uses `lseek` to skip the data field before executing the next loop iteration to process the next (possibly shorter) input unit (lines 28–29).

- **Partial Read.** The `read` operation on line 17 may read only part of the data field. In this case the program skips the remaining bytes in the data field before executing the next loop iteration to process the next input unit (lines 21–23).

- **Failed Read.** The `read` operation on line 17 may fail outright. In this case the program aborts processing the file and does not attempt to recover (lines 24–26).

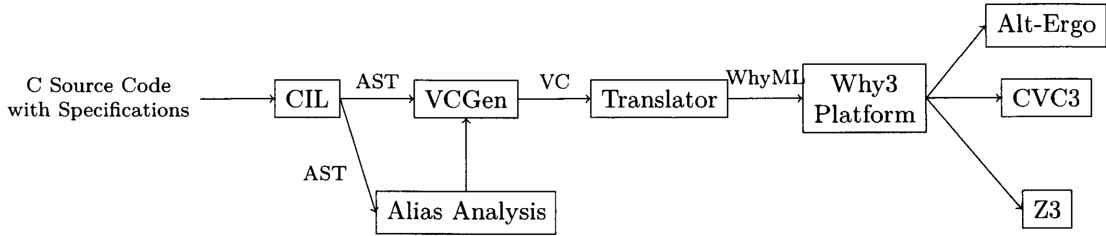


Figure 1-3: The structure of the verification system.

Overlooking any one of these cases or failing to correctly code the recovery code for one of these cases can cause the program to become desynchronized with the input file and process subsequent input units incorrectly.

Note that in the example the loop that finds the delimiter at the start of the next input unit does *not* eliminate the problem of the program becoming desynchronized with the input file — if the `read` operation leaves the file position indicator in the middle of the data field, the program may incorrectly mistake an FF_{16} byte in the data field as the delimiter indicating the start of the next input unit. Verifying the second assertion ensures that the example program does not have this kind of defect because it correctly reads or skips all bytes in the data field, leaving the file position indicator referencing the byte after the last byte in the input unit.

1.1.3 Verification

Figure 1-3 presents a block diagram of the structure of the presented verification system. The system works with C code augmented with specifications inserted inline at appropriate program points. Specifications take the form of C macros, which expand into CIL attributes [38]. The system uses the CIL front end to obtain an abstract syntax tree (AST) for the program. Because the specifications are expressed as CIL attributes, they are parsed by the CIL front end and inserted into the abstract syntax tree that the CIL front end generates. The system next runs an extended version of the GOLF alias analysis algorithm [16] to derive loop invariants that characterize potential aliasing relationships in the program. The verification condition generator takes the abstract syntax tree (including specifications) and the aliasing information.

It then uses a precondition analysis to propagate the specifications against the flow of control to obtain appropriate verification conditions, which it discharges using the Why3 platform [18], which deploys multiple reasoning systems and decision procedures to discharge the verification conditions.

A key aspect of the system is how it reasons about the file contents and the file position indicator. To support the kind of accurate reasoning required to verify the target specifications, the system represents the file contents symbolically and accurately, including taking information from conditionals into account when accumulating information about the file contents. For example, a common pattern in many programs that read files is to check the contents of certain file bytes against flag values identified in the file format specification (for example, line 9 in Figure 1-2 on page 16). Precisely tracking the values of these file bytes across the program, as established by the conditionals that read these bytes, is necessary to verify specifications (such as line 12 in Figure 1-2 on page 16) that check specific values of specific file bytes.

The expressions in the conditions identify the positions of specific bytes within a given file as offsets from the file position indicator at an identified program point. This value of this file position indicator is the value from the last time the program executed the identified program point. The expressions that denote offsets can use constants, program variables, and the contents of specific bytes within the file, and sums over these quantities.

To support scalable verification, the system separates out two specific file read cases. When the file read operation reads a compile-time constant number of bytes, the system generates expressions that separately represent the value of each read byte. Such file read operations typically read flags or delimiters, which often contain specific values that the program checks. Representing the bytes separately facilitates the kind of precise reasoning required to verify that the checks appropriately satisfy the specification.

When a file read operation reads a number of bytes that cannot be statically determined, the system generates a single expression that summarizes the values of all read bytes. The rationale is that such reads typically correspond to loading data

fields (as opposed to flags, delimiters, or other formatting information) whose specific values are not relevant to the specification. This approach precisely but efficiently handles this case.

1.2 Parser Generators

Correctly processing input files (specifically, parsing computer programs) was identified as a problem decades ago, soon after programming language designers started producing the first high-level languages [26, 28, 41]. Indeed, parsing research was a focal point of computer science for several decades [3, 22] and is still an active area of research [20, 27, 30, 39]. This research has produced a range of sophisticated techniques for correctly parsing complex inputs. Packaging these techniques into parser generators automates their application and makes it possible to automatically convert a specification of the input file format into a parser that can correctly process all files that conform to the file format [1, 9, 12].

Given the decades of research that have gone into this effort and the many parser generators that this research has produced, one may reasonably ask why developers still overwhelmingly write input processing code by hand given the widespread availability of parser generators and the significant advantages they claim to deliver. I identify several reasons why this may be the case.

First, parser generators take a specification of what a valid input looks like and generate a program that processes valid inputs. Because the question of how to handle invalid inputs typically lies outside the scope of the specification language, the parser generator usually applies a default error handling strategy. The most common error handling strategy is to simply reject the input and abandon the parsing effort.

But many input processing programs must apply much more sophisticated strategies that involve using domain-specific information to recover from corrupted or unexpected inputs. Parser generators provide no satisfactory way to incorporate this domain-specific information and therefore do not satisfy the needs of developers building software for processing files. In this situation developers fall back onto writing file

processing software in standard programming languages, which enable developers to apply custom domain-specific error recovery strategies that attempt to resiliently process their inputs.

Second, parser generators complicate the build process and environment. They inject a new language and language processor (the parser generator) into the build process. Because parser generators are much less widely used than standard programming language implementations, they tend to have more problems, are not as well documented or supported, and it can be difficult to find developers who can use the parser generator effectively. All of these issues argue against the use of parser generators in large software systems, once again leaving developers writing input processing code in standard programming languages.

The significant practical drawbacks that the history of parser generators has surfaced, the demonstrated unwillingness of developers to use parser generators, and the known problems associated with developing input processing code in standard programming languages all motivate the need for techniques (such as those presented in this thesis) that help developers produce correct input processing code written in standard programming languages.

1.3 Contributions

This thesis makes the following contributions:

- **Specification Language.** The thesis presents a specification language that enables developers to specify the following correctness properties for programs that process files:
  - A program has consumed the correct number of bytes between two different program points.
  - A program has checked that file contents have correct values, such as file signatures and delimiters, at the corresponding program points.
- **Verification System.** The thesis presents a program verification system that

verifies, for all possible input files, that those correctness properties hold in all possible program executions.

1.4 Structure

The remainder of the thesis is structured as follows. Chapter 2 presents a detailed example that illustrates how to specify correctness properties for a PNG image viewer using the proposed specification language and how the verification system checks that they hold for all input files and program executions. Chapter 3 defines an imperative programming language that supports file I/O operations and byte-wise operations such as endianness conversions. Based on the programming language, Chapter 4 formally defines the syntax and semantics of the proposed specification language. Chapter 5 explains how the verification system works in terms of predicate transformer semantics and proves the soundness of the verification process of input specifications. Chapter 6 gives another example that shows how the proposed system can be used to verify programs that process delimiter-based file formats, such as the JPEG File Interchange Format (JFIF), as well as length-based file formats. Chapter 7 discusses about the implementation of the proposed system and evaluates its practical aspects. Chapter 8 presents related work and the thesis concludes with Chapter 9.

Chapter 2

Portable Network Graphics

This chapter presents a detailed example that illustrates how the proposed programming system can help a developer when he or she wants to write an image viewer for PNG files in an error-resilient way. The example shows what challenges the proposed system has come across in the process of verifying that the developer's implementation meets its input specification and how it could cope with them.

2.1 The PNG Format

PNG (Portable Network Graphics) [2] is a file format for raster graphics images that supports lossless data compression. In 2013, the PNG format was reported to be the most widely-used lossless image file format used in the Internet [21].

Figure 2-1 on page 26 shows the structure of a typical PNG image file. The file format starts with an 8-byte signature, which is used for a file to identify itself as the PNG image file format. The signature is followed by a series of blocks, which the PNG standard refers to as *chunks*. The first chunk is called IHDR, which specifies the dimensions, bit depth, and color type of the contained image. The PLTE chunk contains the colormap information for the image data and is present only if the image's color type is palette. The IDAT chunks contain the actual (compressed) image data and must appear consecutively. A PNG file ends with an IEND chunk.

All the chunks have a common structure shown in Figure 2-2 on page 26. Each

PNG Signature	IHDR Chunk	PLTE Chunk	IDAT Chunk 1	···	IDAT Chunk n	IEND Chunk

Figure 2-1: The structure of a typical PNG image file.

Length (4 bytes)	Type (4 bytes)	Data (Length bytes)	CRC (4 bytes)

Figure 2-2: The common structure of PNG chunks.

chunk consists of 4 fields: Length (4 bytes), Type (4 bytes), Data (Length bytes), and CRC (4 bytes). The Length field is a 4-byte unsigned integer in network byte order and specifies the size of the chunk's Data field in bytes. The Type field determines the type of the chunk and is a sequence of 4 ASCII letters, such as "IHDR", "PLTE", "IDAT", and "IEND". The Data field contains information proper to the type of the chunk. When the Length field of a chunk is zero (for example, the IEND chunk), its Data field is omitted. Every chunk ends with a 4-byte CRC (Cyclic Redundancy Check) value that has been calculated over the Type and Data fields.

2.2 A PNG Image Viewer

Figures 2-3 and 2-4 on pages 27 and 29 present code snippets from an image viewer for PNG files that was written as resiliently as possible. Figure 2-3 shows the first part of the viewer, which attempts to read 8 bytes and checks if they match the PNG signature in order to confirm that the input file is a PNG-formatted image (lines 10–18).

```
10  if ((nbytes = read(fd, signature, sizeof(signature))) != sizeof(signature)) {
11    exit(1);  // I/O ERROR OR NOT WELL-FORMED
12  }
13  if (signature[0] != '\x89' || signature[1] != '\x50' ||
14      signature[2] != '\x4E' || signature[3] != '\x47' ||
15      signature[4] != '\x0D' || signature[5] != '\x0A' ||
16      signature[6] != '\x1A' || signature[7] != '\x0A') {
17    exit(1);  // NOT WELL-FORMED
18  }
```

```
1  uint8_t signature[8];

2  uint32_t length;  // the Length field (4 bytes)
3  uint8_t type[4];  // the Type field (4 bytes)
4  uint8_t *data;  // the Data field (Length bytes)
5  uint32_t crc;  // the CRC field (4 bytes)

6  ssize_t nbytes;
7  ...

8  FILE_START:
9  ...

10 if ((nbytes = read(fd, signature, sizeof(signature))) != sizeof(signature)) {
11   exit(1);  // I/O ERROR OR NOT WELL-FORMED
12 }
13 if (signature[0] != '\x89' || signature[1] != '\x50' ||
14     signature[2] != '\x4E' || signature[3] != '\x47' ||
15     signature[4] != '\x0D' || signature[5] != '\x0A' ||
16     signature[6] != '\x1A' || signature[7] != '\x0A') {
17   exit(1);  // NOT WELL-FORMED
18 }
19 ...
20 ASSERT(GET_POS(fd, "CURRENT") == GET_POS(fd, "FILE_START") + 8 &&
21        GET_CONTENT(fd, GET_POS(fd, "CURRENT") - 8) == 0x89 &&
22        GET_CONTENT(fd, GET_POS(fd, "CURRENT") - 7) == 0x50 &&
23        GET_CONTENT(fd, GET_POS(fd, "CURRENT") - 6) == 0x4E &&
24        GET_CONTENT(fd, GET_POS(fd, "CURRENT") - 5) == 0x47 &&
25        GET_CONTENT(fd, GET_POS(fd, "CURRENT") - 4) == 0x0D &&
26        GET_CONTENT(fd, GET_POS(fd, "CURRENT") - 3) == 0x0A &&
27        GET_CONTENT(fd, GET_POS(fd, "CURRENT") - 2) == 0x1A &&
28        GET_CONTENT(fd, GET_POS(fd, "CURRENT") - 1) == 0x0A);
```

Figure 2-3: A (simplified) code snippet from an image viewer for PNG files (1/2).

The POSIX `read(`*fd*`,` *buf*`,` *n*`)` system call tries to read n number of bytes from the file referenced by the file descriptor *fd* into the buffer *buf*, and returns the number of bytes that it has actually read or -1 to indicate that an error has occurred during its execution. More specifically, there are three cases where the `read()` call here cannot read as many bytes as specified in the argument:

- The file descriptor is not associated with an open file or an I/O error (e.g., hardware failure) has occurred, where the `read` function returns -1.

- The file descriptor has already reached the end of the image file, where the `read` function returns 0. This indicates that the file is empty and not well-formed.

- The file descriptor has not reached the end of the image file yet but the number of bytes left is less than the requested amount. This indicates that the image file ends prematurely and therefore is not well-formed. The `read` function reads all the bytes left and returns the count, which is greater than zero and less than the number of bytes specified in the argument.

For all these cases, the viewer terminates with a failure exit code, displaying some error messages (not shown here) as appropriate.

Once the viewer has read the PNG signature successfully, it processes each chunk in a loop because all the chunks share the same structure (Figure 2-4 on page 29).

Each iteration of the loop starts by making an attempt to read the Length field of 4 bytes into the `length` variable. As in the PNG signature, the `read` call may not be able to read as many bytes as specified in the argument. However, the image file is not always ill-formed when the `read` function returns 0. Rather, if the previous iteration has handled the IEND chunk, the return value 0 indicates that all parts of the image file have been successfully consumed. In all cases where the `read` function could not read the specified number of bytes, the viewer breaks out of the loop, setting some

```
1  for (;;) {
2  CHUNK_START:;

3      // the Length field (4 bytes)
4      if ((nbytes = read(fd, &length, sizeof(length))) != sizeof(length)) {
5          if (nbytes == 0) break;  // EOF
6          break;  // I/O ERROR OR NOT WELL-FORMED
7      }
8      length = ntohl(length);

9      // the Type field (4 bytes)
10     if ((nbytes = read(fd, type, sizeof(type))) != sizeof(type))
11         break;  // I/O ERROR OR NOT WELL-FORMED

12     // the Data field (Length bytes)
13     if ((data = malloc(length)) == NULL) {
14         off_t offset = lseek(fd, length + 4, SEEK_CUR);
15         if (offset == -1) break;  // I/O ERROR
16         ASSERT(GET_POS(fd, "CURRENT") ==
17                GET_POS(fd, "CHUNK_START") + 4 + 4 +
18                (GET_CONTENT(fd, GET_POS(fd, "CHUNK_START")) * 16777216 +
19                 GET_CONTENT(fd, GET_POS(fd, "CHUNK_START") + 1) * 65536 +
20                 GET_CONTENT(fd, GET_POS(fd, "CHUNK_START") + 2) * 256 +
21                 GET_CONTENT(fd, GET_POS(fd, "CHUNK_START") + 3)) + 4);
22         continue;  // TOLERATE NOT ENOUGH MEMORY
23     }
24     if ((nbytes = read(fd, data, length)) != length) {
25         free(data); break;  // I/O ERROR OR NOT WELL-FORMED
26     }

27     // the CRC field (4 bytes)
28     if ((nbytes = read(fd, &crc, sizeof(crc))) != sizeof(crc)) {
29         free(data); break;  // I/O ERROR OR NOT WELL-FORMED
30     }
31     crc = ntohl(crc);

32     ASSERT(GET_POS(fd, "CURRENT") ==
33            GET_POS(fd, "CHUNK_START") + 4 + 4 +
34            (GET_CONTENT(fd, GET_POS(fd, "CHUNK_START")) * 16777216 +
35             GET_CONTENT(fd, GET_POS(fd, "CHUNK_START") + 1) * 65536 +
36             GET_CONTENT(fd, GET_POS(fd, "CHUNK_START") + 2) * 256 +
37             GET_CONTENT(fd, GET_POS(fd, "CHUNK_START") + 3)) + 4);

38     // process data.
39     ...
40 }
```

Figure 2-4: A (simplified) code snippet from an image viewer for PNG files (2/2).

flags (not shown here) as appropriate (lines 4–7).

```
4       if ((nbytes = read(fd, &length, sizeof(length))) != sizeof(length)) {
5           if (nbytes == 0) break;   // EOF
6           break;  // I/O ERROR OR NOT WELL-FORMED
7       }
```

On the other hand, when the read call could read the exact number of bytes requested, the viewer invokes the ntohl function to convert the length variable from network byte order to host byte order (line 8).

```
8       length = ntohl(length);
```

Next, the viewer attempts to read 4 bytes for the Type field. Unlike the read call for the Length field, the return value 0 from this read call (or the premature end-of-file) means that the mandatory Type and CRC fields are missing and that the image file is not syntactically well-formed. Therefore, the viewer terminates the execution of the loop in all cases where the read function failed to read the requested number of bytes (lines 10–11).

```
10      if ((nbytes = read(fd, type, sizeof(type))) != sizeof(type))
11          break;  // I/O ERROR OR NOT WELL-FORMED
```

The viewer makes an effort to process the Data field in an error-resilient manner. Because the Data field is of variable length, the viewer dynamically allocates a memory block of size specified in the Length field via the malloc function (line 13).

```
13      if ((data = malloc(length)) == NULL) {
```

Dynamic allocation fails if there is not enough free memory, and then the malloc function returns NULL. In that case, a naïve (and common) approach is to blindly stop the whole process (e.g., by terminating the program with an error message),

wasting all the efforts that have been made so far, even though the previous chunks of the image file have been processed successfully and the following chunks may be shorter than the current one. In contrast, this image viewer skips to the next chunk by adjusting the offset of the file descriptor accordingly, and continues to process the remaining chunks of the image file (lines 14–22): the `lseek(`*fd*`,` *offset*`, SEEK_CUR)` function moves the file descriptor *fd* from its current position by the *offset* amount.

```
14          off_t offset = lseek(fd, length + 4, SEEK_CUR);
15          if (offset == -1) break;  // I/O ERROR
16          ASSERT(GET_POS(fd, "CURRENT") ==
17                 GET_POS(fd, "CHUNK_START") + 4 + 4 +
18                 (GET_CONTENT(fd, GET_POS(fd, "CHUNK_START")) * 16777216 +
19                  GET_CONTENT(fd, GET_POS(fd, "CHUNK_START") + 1) * 65536 +
20                  GET_CONTENT(fd, GET_POS(fd, "CHUNK_START") + 2) * 256 +
21                  GET_CONTENT(fd, GET_POS(fd, "CHUNK_START") + 3)) + 4);
22          continue;  // TOLERATE NOT ENOUGH MEMORY
```

For this strategy to work, a developer must specify the exact number of bytes to skip so that the next iteration of the loop starts at the beginning of the next chunk. Otherwise, the viewer would completely misread the rest of the image file and there would be no good point in continuing the execution of the program. Specifically, the viewer has to skip both of the Data and CRC fields to take the file descriptor to the next chunk of the PNG image file. While the CRC field is a fixed-length field, the size of the Data field is determined by the Length field, which is stored in the file itself. Thus, the amount to skip is not specified by a constant but by an expression involving the `length` variable, which the result of endianness conversion was stored into.

Once the memory allocation is successful, the image viewer processes the Data and succeeding CRC fields similarly to the Type field: it tries to read an appropriate number of bytes and checks how many bytes have actually been read. However, in case of errors, the program also deallocates the memory space for the Data field because it is a local resource to the current iteration.

2.3 Input Specification

If the image viewer has been correctly written according to the PNG specification, it is supposed to have observed the 8-byte PNG signature when the program execution reaches the main loop. Also, the file position indicator should be 8-byte away from the beginning of the input file so that the viewer is ready for reading the first byte of the first chunk. The proposed system provides a specification language that enables a developer to specify 1) how many bytes a program has to consume between two program execution points and 2) what byte values are supposed to be observed at specific offsets in an input file. In the specification language, the above property could be specified as follows, just before entering the main loop (lines 20–28 in Figure 2-3 on page 27):

```
20  ASSERT(GET_POS(fd, "CURRENT") == GET_POS(fd, "FILE_START") + 8 &&
21        GET_CONTENT(fd, GET_POS(fd, "CURRENT") - 8) == 0x89 &&
22        GET_CONTENT(fd, GET_POS(fd, "CURRENT") - 7) == 0x50 &&
23        GET_CONTENT(fd, GET_POS(fd, "CURRENT") - 6) == 0x4E &&
24        GET_CONTENT(fd, GET_POS(fd, "CURRENT") - 5) == 0x47 &&
25        GET_CONTENT(fd, GET_POS(fd, "CURRENT") - 4) == 0x0D &&
26        GET_CONTENT(fd, GET_POS(fd, "CURRENT") - 3) == 0x0A &&
27        GET_CONTENT(fd, GET_POS(fd, "CURRENT") - 2) == 0x1A &&
28        GET_CONTENT(fd, GET_POS(fd, "CURRENT") - 1) == 0x0A);
```

The first line of the assertion specifies that the current file position indicator, denoted by `GET_POS(fd, "CURRENT")`, should be 8-bytes away from the file position indicator at file opening time, denoted by `GET_POS(fd, "FILE_START")`. `FILE_START` is a C label attached to a program point where a file has just been opened (line 8 in Figure 2-3 on page 27). Also, the developer specifies that the previous 8 bytes from the current file position should be identical to the PNG signature (lines 21–28). This requires the image viewer to check whether the image file contains the correct PNG signature or not, as shown at lines 13–16 in Figure 2-3 on page 27. `GET_CONTENT(`*fd*`,` *offset*`)` denotes the byte value at *offset* in the file referenced by the file descriptor *fd*.

When the current chunk is syntactically well-formed, there are two scenarios with the main loop where the image viewer can finish with the current chunk and proceed to the next one: 1) the viewer managed to execute all system calls (or their wrappers), such as `malloc`, without any runtime error, and then the program execution reached the end of the current iteration or 2) the program execution suffered some runtime errors, but the viewer decided to tolerate them and could take appropriate actions such as relocating the file descriptor.

For both of those two cases, it should be guaranteed that the program has consumed all the bytes of the current chunk and is ready for reading the next chunk with the start of the next iteration. Otherwise, the program has been mistakenly written in a way that does not conform to the specification of the file format or take a correct action to recover from errors.

In this example, the developer specified that the image viewer is designed to guarantee that, whenever it manages to reach the end of each iteration, it has read all the bytes of the current chunk (lines 16–21 and 32–37).

```
16        ASSERT(GET_POS(fd, "CURRENT") ==
17               GET_POS(fd, "CHUNK_START") + 4 + 4 +
18               (GET_CONTENT(fd, GET_POS(fd, "CHUNK_START")) * 16777216 +
19                GET_CONTENT(fd, GET_POS(fd, "CHUNK_START") + 1) * 65536 +
20                GET_CONTENT(fd, GET_POS(fd, "CHUNK_START") + 2) * 256 +
21                GET_CONTENT(fd, GET_POS(fd, "CHUNK_START") + 3)) + 4);
```

```
32    ASSERT(GET_POS(fd, "CURRENT") ==
33           GET_POS(fd, "CHUNK_START") + 4 + 4 +
34           (GET_CONTENT(fd, GET_POS(fd, "CHUNK_START")) * 16777216 +
35            GET_CONTENT(fd, GET_POS(fd, "CHUNK_START") + 1) * 65536 +
36            GET_CONTENT(fd, GET_POS(fd, "CHUNK_START") + 2) * 256 +
37            GET_CONTENT(fd, GET_POS(fd, "CHUNK_START") + 3)) + 4);
```

Specifically, the current file position indicators at lines 16 and 32, denoted by `GET_POS(fd, "CURRENT")`, should be different from the file position indicator at the start of the current iteration (labeled as `CHUNK_START` at line 2 in Figure 2-4

on page 29), denoted by `GET_POS(fd, "CHUNK_START")`, by 4 (Length) + 4 (Type) + |Data| + 4 (CRC). |Data| denotes the variable size of the Data field, which is an integer value obtained by interpreting the Length field of the current chunk in big-endian byte order. The Length field is located at the beginning of the current chunk, whose position is expressed as `GET_POS(fd, "CHUNK_START")`. Therefore, the n-th byte of the Length field can be found at `GET_POS(fd, "CHUNK_START")` $+n$, and is multiplied by 2^{24-8n} because the most significant byte is stored first in big-endian format (lines 18–21 and 34–37).

2.4 Verification

This example illustrates three challenges, among others, that the proposed system has to cope with to verify that the developer's implementation meets its input specification.

First, the input specifications are relational in two different dimensions. They stipulate not only how the constituent parts of a program state should be related at a given program point, but also what relationship a program state at a program point should have with program states at different program points.

```
20  ASSERT(GET_POS(fd, "CURRENT") == GET_POS(fd, "FILE_START") + 8 &&
21         GET_CONTENT(fd, GET_POS(fd, "CURRENT") - 8) == 0x89 &&
22         GET_CONTENT(fd, GET_POS(fd, "CURRENT") - 7) == 0x50 &&
23         GET_CONTENT(fd, GET_POS(fd, "CURRENT") - 6) == 0x4E &&
24         GET_CONTENT(fd, GET_POS(fd, "CURRENT") - 5) == 0x47 &&
25         GET_CONTENT(fd, GET_POS(fd, "CURRENT") - 4) == 0x0D &&
26         GET_CONTENT(fd, GET_POS(fd, "CURRENT") - 3) == 0x0A &&
27         GET_CONTENT(fd, GET_POS(fd, "CURRENT") - 2) == 0x1A &&
28         GET_CONTENT(fd, GET_POS(fd, "CURRENT") - 1) == 0x0A);
```

For example, the above specification involves `GET_POS(fd, "CURRENT")` and `GET_POS(fd, "FILE_START")`, which are the states of the file position indicator at two different program execution points `CURRENT` and `FILE_START`. The conventional value analyses, which compute the intervals of each variable's possible values at each program point,

are not suitable for tracking such sophisticated relationships among program states at different program points [13, 15].

Second, the program consists of a big loop that repeats an indefinite number of times. The loop terminates only when the end of the image file has been reached or an irrecoverable error occurs. In other words, the loop does not have any terminating conditions that can impose an upper bound on how many times it will be executed. Thus, it is inappropriate to apply conventional approaches that attempt to reach some fixed points or compute underapproximation by iterating a fixed number of times [32]. On the other hand, if we introduce some coarse approximation to guarantee the termination of our analyses, there is always some chance that we cannot confirm the validity of specifications just because of the granularity of the approximation.

Finally, this kind of program almost always involves byte-wise operations because it deals with a binary file format. In our example, the `read` function reads the contents of a file by units of bytes, and the `ntohl` function converts an integer value from network byte order to host byte order. Previous techniques that treat fixed-width machine integers as mathematical integers usually model the behaviors of those low-level operations approximately. For instance, the return value of a `ntohl` call would be simply modeled as a top element, which denotes any arbitrary integer value, regardless of its input argument. Considering the pervasiveness of those operations in our target applications, such approximation can become too conservative.

To deal with these three challenges, the proposed system starts by transforming an input specification that depends on program states at different program points into one that depends only on the current program state. Specifically, a new ghost variable is introduced for each `GET_POS(`fd`, "`ℓ`")` that appears in the input specification. Then an assignment statement is inserted to initialize the ghost variable to the file position of file descriptor fd at program point ℓ, and the `GET_POS(`fd`, "`ℓ`")` part in the input specification is substituted with a reference to the corresponding ghost variable. For instance, this transformation rewrites the specification on lines 32–37 in Figure 2-4 on page 29 as follows:

```
    ASSERT(GET_POS(fd, "CURRENT") ==
-          GET_POS(fd, "CHUNK_START") + 4 + 4 +
-          (GET_CONTENT(fd, GET_POS(fd, "CHUNK_START")) * 16777216 +
-           GET_CONTENT(fd, GET_POS(fd, "CHUNK_START") + 1) * 65536 +
-           GET_CONTENT(fd, GET_POS(fd, "CHUNK_START") + 2) * 256 +
-           GET_CONTENT(fd, GET_POS(fd, "CHUNK_START") + 3)) + 4);
+          __ghost_GET_POS_fd_CHUNK_START + 4 + 4 +
+          (GET_CONTENT(fd, __ghost_GET_POS_fd_CHUNK_START_) * 16777216 +
+           GET_CONTENT(fd, __ghost_GET_POS_fd_CHUNK_START_ + 1) * 65536 +
+           GET_CONTENT(fd, __ghost_GET_POS_fd_CHUNK_START_ + 2) * 256 +
+           GET_CONTENT(fd, __ghost_GET_POS_fd_CHUNK_START_ + 3)) + 4);
```

and puts an assignment statement `_ghost__GET_POS_fd_CHUNK_START_ = GET_POS(fd, "CURRENT")` below line 2 in Figure 2-4 on page 29.

```
CHUNK_START:
+    _ghost__GET_POS_fd_CHUNK_START_ = GET_POS(fd, "CURRENT");
```

Now the specification just involves the ghost variable, the current file position `GET_POS(fd, "CURRENT")`, and the file contents `GET_CONTENT(fd, ...)`, which all refer to the current program state only.

The proposed system propagates the transformed input specification backwards against the control flow and figures out what condition at each program point should be satisfied to guarantee the specification. When a new condition is propagated into a program point, the system checks if the old condition there implies the new condition. If so, the new condition is not propagated further. Otherwise, the logical conjunction of the old and new conditions is stored at the program point and propagated backwards again. After the system finishes figuring out what condition at each program point should be met, it checks if the resulting condition at the start of the program can be proved to be a tautology, which means that every possible execution of the program correctly meets the specification.

For our example, the analysis translates the (transformed) input specification

above into the following Q_{32} condition that should be satisfied *before* entering line 32:

```
32    ASSERT(GET_POS(fd, "CURRENT") ==
33            __ghost_GET_POS_fd_CHUNK_START + 4 + 4 +
34            (GET_CONTENT(fd, __ghost_GET_POS_fd_CHUNK_START_) * 16777216 +
35             GET_CONTENT(fd, __ghost_GET_POS_fd_CHUNK_START_ + 1) * 65536 +
36             GET_CONTENT(fd, __ghost_GET_POS_fd_CHUNK_START_ + 2) * 256 +
37             GET_CONTENT(fd, __ghost_GET_POS_fd_CHUNK_START_ + 3)) + 4);
```

$$\begin{aligned}\mathcal{P}(\texttt{fd}) = \texttt{_ghost_GET_POS_fd_CHUNK_START_} &+ 4 + 4 + \\ (\mathcal{C}(\texttt{fd}, \texttt{_ghost_GET_POS_fd_CHUNK_START_}) &\times 2^{24} + \\ \mathcal{C}(\texttt{fd}, \texttt{_ghost_GET_POS_fd_CHUNK_START_} + 1) &\times 2^{16} + \\ \mathcal{C}(\texttt{fd}, \texttt{_ghost_GET_POS_fd_CHUNK_START_} + 2) &\times 2^{8} + \\ \mathcal{C}(\texttt{fd}, \texttt{_ghost_GET_POS_fd_CHUNK_START_} + 3)) &+ 4\end{aligned} \qquad (Q_{32})$$

$\mathcal{P}(fd)$ stands for the current file position of the *fd* file descriptor. $\mathcal{C}(fd, offset)$ denotes the byte value at *offset* from the start of a file referenced by the file descriptor *fd*. All together, condition Q_{32} asserts that the current file position indicator of the `fd` file descriptor ($\mathcal{P}(\texttt{fd})$) should be located at 4 + 4 + (the size of the Data field) + 4 away from the `_ghost_GET_POS_fd_CHUNK_START_` ghost variable, which remembers the file position of the `fd` file descriptor at label `CHUNK_START`. The variable size of the Data field is an integer value obtained by interpreting a 4-byte big-endian integer at offset `_ghost_GET_POS_fd_CHUNK_START_` of file *fd*: $\mathcal{C}(\texttt{fd}, \texttt{_ghost_GET_POS_fd_CHUNK_START_}) \times 2^{24} + \mathcal{C}(\texttt{fd}, \texttt{_ghost_GET_POS_fd_CHUNK_START_} +1) \times 2^{16} + \cdots + \mathcal{C}(\texttt{fd}, \texttt{_ghost_GET_POS_fd_CHUNK_START_} + 3)$.

The assignment statement on line 31 does not make any change on condition Q_{32} because the `crc` variable does not occur in condition Q_{32}. Therefore, condition Q_{31} is exactly the same as condition Q_{32}. This reflects the fact that the value of the CRC field is not concerned with the *syntactic* structure of PNG files because the field takes a fixed number of bytes. Even when a checksum error is detected in a chunk, we may still be able to successfully process the remaining chunks of the PNG file and get a

better partial result.

```
31      crc = ntohl(crc);
```

$$\begin{aligned}\mathcal{P}(\texttt{fd}) = {} & \texttt{_ghost_GET_POS_fd_CHUNK_START_} + 4 + 4 + \\ & (\mathcal{C}(\texttt{fd}, \texttt{_ghost_GET_POS_fd_CHUNK_START_}) \times 2^{24} + \\ & \mathcal{C}(\texttt{fd}, \texttt{_ghost_GET_POS_fd_CHUNK_START_} + 1) \times 2^{16} + \\ & \mathcal{C}(\texttt{fd}, \texttt{_ghost_GET_POS_fd_CHUNK_START_} + 2) \times 2^{8} + \\ & \mathcal{C}(\texttt{fd}, \texttt{_ghost_GET_POS_fd_CHUNK_START_} + 3)) + 4\end{aligned} \qquad (Q_{31})$$

Condition Q_{31} is propagated into both the then branch and the implicit else branch of the `if` block on lines 28–30.

```
28      if ((nbytes = read(fd, &crc, sizeof(crc))) != sizeof(crc)) {
29          free(data); break;  // I/O ERROR OR NOT WELL-FORMED
30      }
```

The last statement of the then branch is `break`, which transfers the control to the statement following the enclosing loop. There is no control flow between the previous program point of the `break` statement and its next program point, and the analysis does not propagate condition Q_{31} across the `break` statement. As a result, the condition that should be satisfied at the beginning of the then branch to guarantee the input specification is just "true." On the other hand, condition Q_{31} should be met at the beginning of the implicit else branch, which has an empty body. Line 28 is a C idiom that calls the `read` function to read a specified number of bytes from a file and checks its return value in a single line, which is semantically equivalent to the following two separate statements:

```
nbytes = read(fd, &crc, sizeof(crc));
if (nbytes != sizeof(crc)) {
```

The `crc` variable is declared as a 4-byte unsigned integer, so `sizeof(crc)` in the if condition always returns 4. Therefore, the conditions at the start of the then and else

branches are guarded by the path condition of the `if` statement, $\texttt{nbytes} \neq 4$, and combined conjunctively to guarantee the specification regardless of which branch will be taken:

$$(\texttt{nbytes} \neq 4 \rightarrow \text{true}) \wedge (\neg(\texttt{nbytes} \neq 4) \rightarrow Q_{31}) \qquad (Q_{28a})$$

The `read` call converts condition Q_{28a} into the following Q_{28b} condition:

$$\begin{aligned}
&Q_{28a}[4\,/\,\texttt{nbytes}][\mathcal{P}\{\texttt{fd} \mapsto \mathcal{P}(\texttt{fd})+4\}\,/\,\mathcal{P}]\\
&\quad[\mathcal{M}\{\texttt{crc} \mapsto \mathcal{C}(\texttt{fd},\mathcal{P}(\texttt{fd}))\}\{\texttt{crc}+1 \mapsto \mathcal{C}(\texttt{fd},\mathcal{P}(\texttt{fd})+1)\}\cdots\{\texttt{crc}+3 \mapsto \mathcal{C}(\texttt{fd},\mathcal{P}(\texttt{fd})+3)\}\,/\,\mathcal{M}] \wedge\\
&Q_{28a}[3\,/\,\texttt{nbytes}][\mathcal{P}\{\texttt{fd} \mapsto \mathcal{P}(\texttt{fd})+3\}\,/\,\mathcal{P}]\\
&\quad[\mathcal{M}\{\texttt{crc} \mapsto \mathcal{C}(\texttt{fd},\mathcal{P}(\texttt{fd}))\}\{\texttt{crc}+1 \mapsto \mathcal{C}(\texttt{fd},\mathcal{P}(\texttt{fd})+1)\}\{\texttt{crc}+2 \mapsto \mathcal{C}(\texttt{fd},\mathcal{P}(\texttt{fd})+2)\}\,/\,\mathcal{M}] \wedge\\
&Q_{28a}[2\,/\,\texttt{nbytes}][\mathcal{P}\{\texttt{fd} \mapsto \mathcal{P}(\texttt{fd})+2\}\,/\,\mathcal{P}]\\
&\quad[\mathcal{M}\{\texttt{crc} \mapsto \mathcal{C}(\texttt{fd},\mathcal{P}(\texttt{fd}))\}\{\texttt{crc}+1 \mapsto \mathcal{C}(\texttt{fd},\mathcal{P}(\texttt{fd})+1)\}\}\,/\,\mathcal{M}] \wedge\\
&Q_{28a}[1\,/\,\texttt{nbytes}][\mathcal{P}\{\texttt{fd} \mapsto \mathcal{P}(\texttt{fd})+1\}\,/\,\mathcal{P}][\mathcal{M}\{\texttt{crc} \mapsto \mathcal{C}(\texttt{fd},\mathcal{P}(\texttt{fd}))\}\,/\,\mathcal{M}] \wedge\\
&Q_{28a}[-1\,/\,\texttt{nbytes}][\mathcal{P}\{\texttt{fd} \mapsto \mathcal{P}(\texttt{fd})+4\}\,/\,\mathcal{P}]\\
&\quad[\mathcal{M}\{\texttt{crc} \mapsto \mathcal{C}(\texttt{fd},\mathcal{P}(\texttt{fd}))\}\{\texttt{crc}+1 \mapsto \mathcal{C}(\texttt{fd},\mathcal{P}(\texttt{fd})+1)\}\cdots\{\texttt{crc}+3 \mapsto \mathcal{C}(\texttt{fd},\mathcal{P}(\texttt{fd})+3)\}\,/\,\mathcal{M}] \wedge\\
&Q_{28a}[-1\,/\,\texttt{nbytes}][\mathcal{P}\{\texttt{fd} \mapsto \mathcal{P}(\texttt{fd})+3\}\,/\,\mathcal{P}]\\
&\quad[\mathcal{M}\{\texttt{crc} \mapsto \mathcal{C}(\texttt{fd},\mathcal{P}(\texttt{fd}))\}\{\texttt{crc}+1 \mapsto \mathcal{C}(\texttt{fd},\mathcal{P}(\texttt{fd})+1)\}\{\texttt{crc}+2 \mapsto \mathcal{C}(\texttt{fd},\mathcal{P}(\texttt{fd})+2)\}\,/\,\mathcal{M}] \wedge\\
&Q_{28a}[-1\,/\,\texttt{nbytes}][\mathcal{P}\{\texttt{fd} \mapsto \mathcal{P}(\texttt{fd})+2\}\,/\,\mathcal{P}]\\
&\quad[\mathcal{M}\{\texttt{crc} \mapsto \mathcal{C}(\texttt{fd},\mathcal{P}(\texttt{fd}))\}\{\texttt{crc}+1 \mapsto \mathcal{C}(\texttt{fd},\mathcal{P}(\texttt{fd})+1)\}\,/\,\mathcal{M}] \wedge\\
&Q_{28a}[-1\,/\,\texttt{nbytes}][\mathcal{P}\{\texttt{fd} \mapsto \mathcal{P}(\texttt{fd})+1\}\,/\,\mathcal{P}][\mathcal{M}\{\texttt{crc} \mapsto \mathcal{C}(\texttt{fd},\mathcal{P}(\texttt{fd}))\}\,/\,\mathcal{M}] \wedge\\
&Q_{28a}[0\,/\,\texttt{nbytes}] \wedge Q_{28a}[-1\,/\,\texttt{nbytes}]
\end{aligned} \qquad (Q_{28b})$$

Note that, although the `crc` variable is a non-pointer variable in the source code, it is modeled as a pointer to a 4-byte unsigned integer in the generated condition. We need to consider the address of the `crc` variable because its address is taken at line 28 by the C address-of operator (`&`).

In case of an external I/O error (e.g., hardware failure), the `read` system call may immediately return -1, which will in turn be assigned to the `nbytes` variable, without changing anything. This case is handled by the last line of condition Q_{28b}: $Q_{28a}[-1/\texttt{nbytes}]$, which is a condition obtained by substituting -1 for all `nbytes` occurrences in condition Q_{28a}.

If the current file position is at or after the end-of-file, the `read` call just returns 0, which is assigned to the `nbytes` variable. This case is handled by the last line of

condition Q_{28b}: $Q_{28a}[0/\texttt{nbytes}]$.

Even when the **read** call has read some bytes successfully, it may still read less than the specified `sizeof(crc)` number of bytes (or 4 bytes): there are not enough bytes left in the file or some I/O error has occurred in the middle of reading. To cover all cases where the `read` call has actually consumed some bytes, the condition has a clause for each possible number of bytes read. Note that we can enumerate all possible numbers of bytes read because the specified number of bytes to read is a constant in this `read` call: `sizeof(crc)`. The first seven lines of condition Q_{28b} consider how the `read` call updates a program state when it has consumed i bytes and completes successfully. The `nbytes` variable is set to i, the number of bytes actually read, and the file position associated with the `fd` file descriptor is moved forward by i: $[i/\texttt{nbytes}]$ and $[\mathcal{P}\{\texttt{fd} \mapsto \mathcal{P}(\texttt{fd}) + i\}/\mathcal{P}]$ where $\mathcal{P}\{\texttt{fd} \mapsto \mathcal{P}(\texttt{fd}) + i\}$ denotes the file position state $\mathcal{P}$ with its entry for the `fd` file descriptor increased by i. The file contents are read into the memory region pointed to by the `crc` variable: $[\mathcal{M}\{\texttt{crc} \mapsto \mathcal{C}(\texttt{fd}, \mathcal{P}(\texttt{fd}))\} \cdots \{\texttt{crc} + (i-1) \mapsto \mathcal{C}(\texttt{fd}, \mathcal{P}(\texttt{fd}) + (i-1))\} / \mathcal{M}]$ where $\mathcal{M} \cdots \{\texttt{crc} + j \mapsto \mathcal{C}(\texttt{fd}, \mathcal{P}(\texttt{fd}) + j)\}$ denotes the memory state $\mathcal{M}$ with its memory location $\texttt{crc} + j$ changed to the byte value at offset $\mathcal{P}(\texttt{fd}) + j$ of file `fd`. The eighth to fourteenth lines deal with the case where an I/O error occurs after successfully reading some bytes. The `read` call updates the memory and file position states just as when it completes successfully, but returns -1 regardless of the number of bytes actually read.

For `read` calls that read a fixed number of bytes, I have chosen to have a clause for each possible number of bytes read to handle the involved memory locations individually. Otherwise, the first seven lines of condition Q_{28b} could be expressed more briefly, as follows:

$$\bigwedge_{m=1}^{4} Q_{28a}[m/\texttt{nbytes}][\mathcal{P}\{\texttt{fd} \mapsto \mathcal{P}(\texttt{fd}) + m\}/\mathcal{P}][\mathcal{M}\{\langle \texttt{crc}, m\rangle \mapsto \mathcal{C}(\texttt{fd}, \mathcal{P}(\texttt{fd}), m)\}/\mathcal{M}]$$

$\mathcal{M}\{\langle \texttt{crc}, m\rangle \mapsto \mathcal{C}(\texttt{fd}, \mathcal{P}(\texttt{fd}), m)\}$ denotes a modified memory state of $\mathcal{M}$ whose memory block of size m starting at address `crc`, instead of a single memory location, has been mapped to the same m number of byte values starting at the current file

position $\mathcal{P}(\texttt{fd})$. My observation is that `read` calls that read a fixed number of bytes are usually meant to read the fields of binary file formats that contain metadata information rather than pure data. Such metadata fields tend to be relatively small and are very often the targets of byte-wise processing. As an example, the signature field of PNG files consists of 8 bytes, and, as shown in Figure 2-3 on page 27 (and below), the image viewer checks if the field contains the valid PNG signature at the level of bytes:

```
10 if ((nbytes = read(fd, signature, sizeof(signature))) != sizeof(signature)) {
11   exit(1);  // I/O ERROR OR NOT WELL-FORMED
12 }
13 if (signature[0] != '\x89' || signature[1] != '\x50' ||
14     signature[2] != '\x4E' || signature[3] != '\x47' ||
15     signature[4] != '\x0D' || signature[5] != '\x0A' ||
16     signature[6] != '\x1A' || signature[7] != '\x0A') {
17   exit(1);  // NOT WELL-FORMED
18 }
```

The condition part of the `if` statement are translated into the following logical formula that involves reading individual memory locations:

$$\mathcal{M}(\texttt{signature}) \neq 137 \lor \mathcal{M}(\texttt{signature} + 1) \neq 80 \lor \cdots \lor \mathcal{M}(\texttt{signature} + 7) \neq 10$$

$\mathcal{M}(addr)$ denotes the single byte value at address *addr* of the current memory state $\mathcal{M}$. Such byte-wise operations can be handled more conveniently when we have a separate clause for each number of bytes read rather than the concise representation that collectively updates memory blocks. For instance, the $\mathcal{M}\{\texttt{signature} + 8 \mapsto \mathcal{C}(\texttt{fd}, \mathcal{P}(\texttt{fd}) + 8)\}(\texttt{signature} + 1)$ term can be reduced more easily into $\mathcal{M}(\texttt{signature} + 1)$ because we certainly know that the memory locations $\texttt{signature} + 1$ and $\texttt{signature} + 8$ do not overlap each other. By contrast, we cannot reduce the $\mathcal{M}\{\langle\texttt{signature}, 8\rangle \mapsto \mathcal{C}(\texttt{fd}, \mathcal{P}(\texttt{fd}), 8)\}(\texttt{signature} + 1)$ term further or need more complicated decision procedures that know how and when to expand such terms into the form of having a

clause for each number of bytes read.

Back to our running example, the `if` block on lines 24–26 transforms condition Q_{28b} into condition Q_{24b} below, as did the `if` block on lines 28–30:

```
24      if ((nbytes = read(fd, data, length)) != length) {
25          free(data); break;  // I/O ERROR OR NOT WELL-FORMED
26      }
```

Line 24 is semantically equivalent to the following two statements:

```
nbytes = read(fd, data, length);
if (nbytes != length) {
```

The `if` part, along with the body part (line 25), yields the following condition:

$$\begin{array}{l}(\mathtt{nbytes} \neq \sum_{i=0}^{3}(\mathcal{M}(\mathtt{length}+i) \times (2^8)^i) \rightarrow \mathrm{true}) \wedge \\ (\neg(\mathtt{nbytes} \neq \sum_{i=0}^{3}(\mathcal{M}(\mathtt{length}+i) \times (2^8)^i)) \rightarrow Q_{28b})\end{array} \qquad (Q_{24a})$$

As with the `crc` variable, the `length` variable is modeled as a pointer to a 4-byte unsigned integer in the generated condition because we need to consider both of its address and content. The C address-of operator (`&`) at line 4 takes the address of the `length` variable.

```
4       if ((nbytes = read(fd, &length, sizeof(length))) != sizeof(length)) {
```

In other words, the lookup of the `length` variable in the source code is modeled as pointer dereferencing in the generated condition. Also, for this example, I assume that the target machine is little-endian, so the least significant byte is stored at the lowest memory address and the most significant byte at the highest memory address. Therefore, the lookup of the `length` variable in the source code is expressed as $\sum_{i=0}^{3}(\mathcal{M}(\mathtt{length}+i) \times (2^8)^i)$ in condition Q_{24a}.

This `read(fd, data, length)` call is different from the previous `read(fd, &crc, sizeof(crc))` call in that the exact number of bytes requested and therefore the

possible range of the number of bytes actually read cannot be statically determined. The `length` variable contains the size of the Data field, which is obtained by reading the Length field of the PNG image file during runtime.

My observation is that `read` calls that read an indefinite number of bytes are mainly meant to read the pure data fields of binary file formats. Such fields do not contain information that determines the syntactic structure of binary files, such as the length of other fields and field delimiters, and are redundant for checking if a program meets its input specification. Moreover, they are usually the longest fields of binary file formats. For instance, the maximum length of the Data field of PNG files is $2^{32} - 1$, so it is infeasible to conservatively enumerate all possibilities.

Therefore, I use universal quantification over the number of bytes actually read and a construct $\mathcal{M}\{\langle a, s\rangle \mapsto \mathcal{C}(f, p, s)\}$, which denotes the updating of a memory block of size s starting at address a, instead of a single memory location, to the same number of byte values starting at position p of file f. For instance, $\mathcal{M}\{\langle \texttt{data}, m\rangle \mapsto \mathcal{C}(\texttt{fd}, \mathcal{P}(\texttt{fd}), m)\}$ in condition Q_{24b} below denotes the memory state $\mathcal{M}$ with its memory locations at addresses from `data` to $\texttt{data} + m$ changed to the byte values at offsets from $\mathcal{P}(\texttt{fd})$ to $\mathcal{P}(\texttt{fd}) + m$ of file `fd`.

Specifically, the first two lines of condition Q_{24b} below consider how a program state is updated by the `read` call when it has consumed a positive number of bytes and completes successfully by returning the number of bytes actually read. The whole clause is universally quantified over m, which is the number of bytes actually read. The variable `nbytes` and the file position state $\mathcal{P}$ are updated similarly to the previous `read` calls: $[m \,/\, \texttt{nbytes}]$ and $[\mathcal{P}\{\texttt{fd} \mapsto \mathcal{P}(\texttt{fd}) + m\} \,/\, \mathcal{P}]$. On the other hand, unlike the previous `read` calls, this call records the updating of the memory state $\mathcal{M}$ in granularity of memory blocks rather than individual memory locations: $[\mathcal{M}\{\langle \texttt{data}, m\rangle \mapsto \mathcal{C}(\texttt{fd}, \mathcal{P}(\texttt{fd}), m)\} \,/\, \mathcal{M}]$. The third and fourth lines consider the case where the `read` call returns -1 because an I/O error has occurred after consuming some bytes. The last line of condition Q_{24b} deals with those cases where an end-of-file

condition or an external I/O error occurs immediately.

$$(\forall 1 \leq m \leq \textstyle\sum_{i=0}^{3}(\mathcal{M}(\texttt{length}+i) \times (2^8)^i) :$$
$$Q_{24a}[m \,/\, \texttt{nbytes}][\mathcal{P}\{\texttt{fd} \mapsto \mathcal{P}(\texttt{fd})+m\} \,/\, \mathcal{P}][\mathcal{M}\{\langle \texttt{data}, m\rangle \mapsto \mathcal{C}(\texttt{fd}, \mathcal{P}(\texttt{fd}), m)\} \,/\, \mathcal{M}]) \wedge$$
$$(\forall 1 \leq m \leq \textstyle\sum_{i=0}^{3}(\mathcal{M}(\texttt{length}+i) \times (2^8)^i) :$$
$$Q_{24a}[-1 \,/\, \texttt{nbytes}][\mathcal{P}\{\texttt{fd} \mapsto \mathcal{P}(\texttt{fd})+m\} \,/\, \mathcal{P}][\mathcal{M}\{\langle \texttt{data}, m\rangle \mapsto \mathcal{C}(\texttt{fd}, \mathcal{P}(\texttt{fd}), m)\} \,/\, \mathcal{M}]) \wedge$$
$$Q_{24a}[0 \,/\, \texttt{nbytes}] \wedge Q_{24a}[-1 \,/\, \texttt{nbytes}] \qquad (Q_{24b})$$

Line 13 is a C idiom that calls the `malloc` function to dynamically allocate memory and checks its return value at the same line:

```
13      if ((data = malloc(length)) == NULL) {
```

The single line is semantically equivalent to the following two separate statements:

```
data = malloc(length);
if (data == NULL) {
```

The `if` part, along with the body part (lines 14–23), produces the following condition Q_{13a}:

```
14          off_t offset = lseek(fd, length + 4, SEEK_CUR);
15          if (offset == -1) break;  // I/O ERROR
16          ASSERT(GET_POS(fd, "CURRENT") ==
17                  GET_POS(fd, "CHUNK_START") + 4 + 4 +
18                  (GET_CONTENT(fd, GET_POS(fd, "CHUNK_START")) * 16777216 +
19                   GET_CONTENT(fd, GET_POS(fd, "CHUNK_START") + 1) * 65536 +
20                   GET_CONTENT(fd, GET_POS(fd, "CHUNK_START") + 2) * 256 +
21                   GET_CONTENT(fd, GET_POS(fd, "CHUNK_START") + 3)) + 4);
22          continue;  // TOLERATE NOT ENOUGH MEMORY
23      }
```

$$(\texttt{data} = 0 \rightarrow \text{true}) \wedge (\neg(\texttt{data} = 0) \rightarrow Q_{24b}) \qquad (Q_{13a})$$

The `continue` statement at line 22 passes control to the end of the enclosing loop body. Like the `break` statement, there is no control flow from the previous program

point of the `continue` statement to its next program point. Therefore, condition Q_{24b} is not propagated across the `continue` statement: $\texttt{data} = 0 \rightarrow \text{true}$.

The `malloc` part, in turn, converts condition Q_{13a} into condition Q_{13b} as follows:

$$\forall \texttt{data} : Q_{13a} \qquad (Q_{13b})$$

The `malloc` function cannot allocate the requested bytes of memory and returns `NULL` if there is not enough memory or the requested size is zero.[1] On success, the `malloc` function returns a non-null pointer to the beginning of newly allocated memory block. Condition Q_{13b} asserts that condition Q_{13a} should be satisfied regardless of the allocated memory address, which is stored in the `data` variable. This is an over-approximation because the `malloc` function allocates memory so that the newly allocated memory block does not overlap any previously allocated ones.

The `if` block on lines 10–11 derives condition Q_{10b} from condition Q_{13b} in a similar way to the `if` block on lines 28–30:

```
10      if ((nbytes = read(fd, type, sizeof(type))) != sizeof(type))
11          break;  // I/O ERROR OR NOT WELL-FORMED
```

$$(\texttt{nbytes} \neq 4 \rightarrow \text{true}) \wedge (\neg(\texttt{nbytes} \neq 4) \rightarrow Q_{13b}) \qquad (Q_{10a})$$

[1]Strictly speaking, the behavior of the `malloc` function is implementation-defined when the requested memory size is zero. The C and POSIX standards allow it to return either a null pointer or a special non-null pointer for implementation reasons. Either choice makes no difference, for purpose of verification, so I just presumed that the `malloc` function would return `NULL` for a zero argument.

$$\begin{aligned}
&Q_{10a}[4\,/\,\mathbf{nbytes}][\mathcal{P}\{\mathtt{fd} \mapsto \mathcal{P}(\mathtt{fd}) + 4\}\,/\,\mathcal{P}] \\
&\quad [\mathcal{M}\{\mathtt{type} \mapsto \mathcal{C}(\mathtt{fd}, \mathcal{P}(\mathtt{fd}))\}\{\mathtt{type} + 1 \mapsto \mathcal{C}(\mathtt{fd}, \mathcal{P}(\mathtt{fd}) + 1)\} \cdots \{\mathtt{type} + 3 \mapsto \mathcal{C}(\mathtt{fd}, \mathcal{P}(\mathtt{fd}) + 3)\}\,/\,\mathcal{M}] \wedge \\
&Q_{10a}[3\,/\,\mathbf{nbytes}][\mathcal{P}\{\mathtt{fd} \mapsto \mathcal{P}(\mathtt{fd}) + 3\}\,/\,\mathcal{P}] \\
&\quad [\mathcal{M}\{\mathtt{type} \mapsto \mathcal{C}(\mathtt{fd}, \mathcal{P}(\mathtt{fd}))\}\{\mathtt{type} + 1 \mapsto \mathcal{C}(\mathtt{fd}, \mathcal{P}(\mathtt{fd}) + 1)\}\{\mathtt{type} + 2 \mapsto \mathcal{C}(\mathtt{fd}, \mathcal{P}(\mathtt{fd}) + 2)\}\,/\,\mathcal{M}] \wedge \\
&Q_{10a}[2\,/\,\mathbf{nbytes}][\mathcal{P}\{\mathtt{fd} \mapsto \mathcal{P}(\mathtt{fd}) + 2\}\,/\,\mathcal{P}] \\
&\quad [\mathcal{M}\{\mathtt{type} \mapsto \mathcal{C}(\mathtt{fd}, \mathcal{P}(\mathtt{fd}))\}\{\mathtt{type} + 1 \mapsto \mathcal{C}(\mathtt{fd}, \mathcal{P}(\mathtt{fd}) + 1)\}\}\,/\,\mathcal{M}] \wedge \\
&Q_{10a}[1\,/\,\mathbf{nbytes}][\mathcal{P}\{\mathtt{fd} \mapsto \mathcal{P}(\mathtt{fd}) + 1\}\,/\,\mathcal{P}][\mathcal{M}\{\mathtt{type} \mapsto \mathcal{C}(\mathtt{fd}, \mathcal{P}(\mathtt{fd}))\}\,/\,\mathcal{M}] \wedge \\
&Q_{10a}[-1\,/\,\mathbf{nbytes}][\mathcal{P}\{\mathtt{fd} \mapsto \mathcal{P}(\mathtt{fd}) + 4\}\,/\,\mathcal{P}] \\
&\quad [\mathcal{M}\{\mathtt{type} \mapsto \mathcal{C}(\mathtt{fd}, \mathcal{P}(\mathtt{fd}))\}\{\mathtt{type} + 1 \mapsto \mathcal{C}(\mathtt{fd}, \mathcal{P}(\mathtt{fd}) + 1)\} \cdots \{\mathtt{type} + 3 \mapsto \mathcal{C}(\mathtt{fd}, \mathcal{P}(\mathtt{fd}) + 3)\}\,/\,\mathcal{M}] \wedge \\
&Q_{10a}[-1\,/\,\mathbf{nbytes}][\mathcal{P}\{\mathtt{fd} \mapsto \mathcal{P}(\mathtt{fd}) + 3\}\,/\,\mathcal{P}] \\
&\quad [\mathcal{M}\{\mathtt{type} \mapsto \mathcal{C}(\mathtt{fd}, \mathcal{P}(\mathtt{fd}))\}\{\mathtt{type} + 1 \mapsto \mathcal{C}(\mathtt{fd}, \mathcal{P}(\mathtt{fd}) + 1)\}\{\mathtt{type} + 2 \mapsto \mathcal{C}(\mathtt{fd}, \mathcal{P}(\mathtt{fd}) + 2)\}\,/\,\mathcal{M}] \wedge \\
&Q_{10a}[-1\,/\,\mathbf{nbytes}][\mathcal{P}\{\mathtt{fd} \mapsto \mathcal{P}(\mathtt{fd}) + 2\}\,/\,\mathcal{P}] \\
&\quad [\mathcal{M}\{\mathtt{type} \mapsto \mathcal{C}(\mathtt{fd}, \mathcal{P}(\mathtt{fd}))\}\{\mathtt{type} + 1 \mapsto \mathcal{C}(\mathtt{fd}, \mathcal{P}(\mathtt{fd}) + 1)\}\,/\,\mathcal{M}] \wedge \\
&Q_{10a}[-1\,/\,\mathbf{nbytes}][\mathcal{P}\{\mathtt{fd} \mapsto \mathcal{P}(\mathtt{fd}) + 1\}\,/\,\mathcal{P}][\mathcal{M}\{\mathtt{type} \mapsto \mathcal{C}(\mathtt{fd}, \mathcal{P}(\mathtt{fd}))\}\,/\,\mathcal{M}] \wedge \\
&Q_{10a}[0\,/\,\mathbf{nbytes}] \wedge Q_{10a}[-1\,/\,\mathbf{nbytes}]
\end{aligned} \tag{Q_{10b}}$$

On little-endian machines, the **ntohl** statement on line 8 returns its argument with the byte order reversed and results in the following Q_8 condition [2]:

```
8       length = ntohl(length);
```

$$\begin{aligned}
&Q_{10b}[\mathcal{M}\{\mathtt{length} + 0 \mapsto \mathsf{ntohl}(\textstyle\sum_{i=0}^{3}(\mathcal{M}(\mathtt{length} + i) \times (2^8)^i)) \div (2^8)^0 \bmod 2^8\}/\mathcal{M}] \\
&\quad [\mathcal{M}\{\mathtt{length} + 1 \mapsto \mathsf{ntohl}(\textstyle\sum_{i=0}^{3}(\mathcal{M}(\mathtt{length} + i) \times (2^8)^i)) \div (2^8)^1 \bmod 2^8\}/\mathcal{M}] \\
&\quad [\mathcal{M}\{\mathtt{length} + 2 \mapsto \mathsf{ntohl}(\textstyle\sum_{i=0}^{3}(\mathcal{M}(\mathtt{length} + i) \times (2^8)^i)) \div (2^8)^2 \bmod 2^8\}/\mathcal{M}] \\
&\quad [\mathcal{M}\{\mathtt{length} + 3 \mapsto \mathsf{ntohl}(\textstyle\sum_{i=0}^{3}(\mathcal{M}(\mathtt{length} + i) \times (2^8)^i)) \div (2^8)^3 \bmod 2^8\}/\mathcal{M}]
\end{aligned} \tag{Q_8}$$

The value of the `length` variable is loaded from the memory and then fed into the

[2]If our target machine were big-endian, the **ntohl** function would return its argument as it is, so the statement would be handled as a plain assignment from its argument to a variable that receives its return value:

$$\begin{aligned}
&Q_{10b}[\mathcal{M}\{\mathtt{length} + 0 \mapsto \mathsf{ntohl}(\textstyle\sum_{i=0}^{3}(\mathcal{M}(\mathtt{length} + i) \times (2^8)^{3-i})) \div (2^8)^3 \bmod 2^8\}/\mathcal{M}] \\
&\quad [\mathcal{M}\{\mathtt{length} + 1 \mapsto \mathsf{ntohl}(\textstyle\sum_{i=0}^{3}(\mathcal{M}(\mathtt{length} + i) \times (2^8)^{3-i})) \div (2^8)^2 \bmod 2^8\}/\mathcal{M}] \\
&\quad [\mathcal{M}\{\mathtt{length} + 2 \mapsto \mathsf{ntohl}(\textstyle\sum_{i=0}^{3}(\mathcal{M}(\mathtt{length} + i) \times (2^8)^{3-i})) \div (2^8)^1 \bmod 2^8\}/\mathcal{M}] \\
&\quad [\mathcal{M}\{\mathtt{length} + 3 \mapsto \mathsf{ntohl}(\textstyle\sum_{i=0}^{3}(\mathcal{M}(\mathtt{length} + i) \times (2^8)^{3-i})) \div (2^8)^0 \bmod 2^8\}/\mathcal{M}]
\end{aligned}$$

where

$$\mathsf{ntohl}(x) = x$$

ntohl function: $\mathsf{ntohl}(\sum_{i=0}^{3}(\mathcal{M}(\mathtt{length}+i)\times(2^8)^i))$. The ntohl function models the behavior of the `ntohl` system call in terms of mathematical integer operations and is defined as follows:

$$\mathsf{ntohl}(x)=\sum_{i=0}^{3}(x\div(2^8)^i \bmod 2^8\times(2^8)^{3-i})$$

The least significant byte of the result of the ntohl function is stored first into the memory, with the most significant byte stored last.

The `if` block on line 4–7 receives condition Q_8 and yields the following Q_{4b} condition in the same way as the other `if` blocks:

```
4      if ((nbytes = read(fd, &length, sizeof(length))) != sizeof(length)) {
5          if (nbytes == 0) break;  // EOF
6          break;  // I/O ERROR OR NOT WELL-FORMED
7      }
```

$$(\mathtt{nbytes}\neq 4\rightarrow\mathsf{true})\wedge(\neg(\mathtt{nbytes}\neq 4)\rightarrow Q_8)\qquad(Q_{4a})$$

$$\begin{aligned}
&Q_{4a}[4/\mathtt{nbytes}][\mathcal{P}\{\mathtt{fd}\mapsto\mathcal{P}(\mathtt{fd})+4\}/\mathcal{P}]\\
&\quad[\mathcal{M}\{\mathtt{length}\mapsto\mathcal{C}(\mathtt{fd},\mathcal{P}(\mathtt{fd}))\}\{\mathtt{length}+1\mapsto\mathcal{C}(\mathtt{fd},\mathcal{P}(\mathtt{fd})+1)\}\cdots\{\mathtt{length}+3\mapsto\mathcal{C}(\mathtt{fd},\mathcal{P}(\mathtt{fd})+3)\}/\mathcal{M}]\wedge\\
&Q_{4a}[3/\mathtt{nbytes}][\mathcal{P}\{\mathtt{fd}\mapsto\mathcal{P}(\mathtt{fd})+3\}/\mathcal{P}]\\
&\quad[\mathcal{M}\{\mathtt{length}\mapsto\mathcal{C}(\mathtt{fd},\mathcal{P}(\mathtt{fd}))\}\{\mathtt{length}+1\mapsto\mathcal{C}(\mathtt{fd},\mathcal{P}(\mathtt{fd})+1)\}\{\mathtt{length}+2\mapsto\mathcal{C}(\mathtt{fd},\mathcal{P}(\mathtt{fd})+2)\}/\mathcal{M}]\wedge\\
&Q_{4a}[2/\mathtt{nbytes}][\mathcal{P}\{\mathtt{fd}\mapsto\mathcal{P}(\mathtt{fd})+2\}/\mathcal{P}]\\
&\quad[\mathcal{M}\{\mathtt{length}\mapsto\mathcal{C}(\mathtt{fd},\mathcal{P}(\mathtt{fd}))\}\{\mathtt{length}+1\mapsto\mathcal{C}(\mathtt{fd},\mathcal{P}(\mathtt{fd})+1)\}\}/\mathcal{M}]\wedge\\
&Q_{4a}[1/\mathtt{nbytes}][\mathcal{P}\{\mathtt{fd}\mapsto\mathcal{P}(\mathtt{fd})+1\}/\mathcal{P}][\mathcal{M}\{\mathtt{length}\mapsto\mathcal{C}(\mathtt{fd},\mathcal{P}(\mathtt{fd}))\}/\mathcal{M}]\wedge\\
&Q_{4a}[-1/\mathtt{nbytes}][\mathcal{P}\{\mathtt{fd}\mapsto\mathcal{P}(\mathtt{fd})+4\}/\mathcal{P}]\\
&\quad[\mathcal{M}\{\mathtt{length}\mapsto\mathcal{C}(\mathtt{fd},\mathcal{P}(\mathtt{fd}))\}\{\mathtt{length}+1\mapsto\mathcal{C}(\mathtt{fd},\mathcal{P}(\mathtt{fd})+1)\}\cdots\{\mathtt{length}+3\mapsto\mathcal{C}(\mathtt{fd},\mathcal{P}(\mathtt{fd})+3)\}/\mathcal{M}]\wedge\\
&Q_{4a}[-1/\mathtt{nbytes}][\mathcal{P}\{\mathtt{fd}\mapsto\mathcal{P}(\mathtt{fd})+3\}/\mathcal{P}]\\
&\quad[\mathcal{M}\{\mathtt{length}\mapsto\mathcal{C}(\mathtt{fd},\mathcal{P}(\mathtt{fd}))\}\{\mathtt{length}+1\mapsto\mathcal{C}(\mathtt{fd},\mathcal{P}(\mathtt{fd})+1)\}\{\mathtt{length}+2\mapsto\mathcal{C}(\mathtt{fd},\mathcal{P}(\mathtt{fd})+2)\}/\mathcal{M}]\wedge\\
&Q_{4a}[-1/\mathtt{nbytes}][\mathcal{P}\{\mathtt{fd}\mapsto\mathcal{P}(\mathtt{fd})+2\}/\mathcal{P}]\\
&\quad[\mathcal{M}\{\mathtt{length}\mapsto\mathcal{C}(\mathtt{fd},\mathcal{P}(\mathtt{fd}))\}\{\mathtt{length}+1\mapsto\mathcal{C}(\mathtt{fd},\mathcal{P}(\mathtt{fd})+1)\}/\mathcal{M}]\wedge\\
&Q_{4a}[-1/\mathtt{nbytes}][\mathcal{P}\{\mathtt{fd}\mapsto\mathcal{P}(\mathtt{fd})+1\}/\mathcal{P}][\mathcal{M}\{\mathtt{length}\mapsto\mathcal{C}(\mathtt{fd},\mathcal{P}(\mathtt{fd}))\}/\mathcal{M}]\wedge\\
&Q_{4a}[0/\mathtt{nbytes}]\wedge Q_{4a}[-1/\mathtt{nbytes}]
\end{aligned}\qquad(Q_{4b})$$

Now that we have arrived below line 2, the next statement to deal with is the

`_ghost__GET_POS_fd_CHUNK_START_ = GET_POS(fd, "CURRENT")` assignment statement, which we inserted there at the beginning of the analysis to reduce the input specification involving different program states to one involving a single program state.

```
CHUNK_START:
    _ghost__GET_POS_fd_CHUNK_START_ = GET_POS(fd, "CURRENT");
```

The effect of the assignment statement is to replace the `_ghost__GET_POS_fd_CHUNK_START_` ghost variable with the current file position:

$$Q_{4b}[\mathcal{P}(\mathtt{fd})/\mathtt{_ghost__GET_POS_fd_CHUNK_START_}] \qquad (Q_2)$$

My observation is that, because the specification relates program states at the beginning and the end of each iteration, it has already incorporated into itself enough information to resolve many of its parts when it is propagated up to the beginning of the loop. The missing information is primarily concerned with memory states that are invariantly established at the beginning of each iteration (possibly by operations outside the loop). For example, we need to know that the `length` and `crc` variables do not refer to the same object so that the `read` statement into the `crc` variable (line 28 in Figure 2-4 on page 29) does not overwrite the value stored in the `length` variable.

```
28        if ((nbytes = read(fd, &crc, sizeof(crc))) != sizeof(crc)) {
```

The relationship between the `length` and `crc` variables can be discovered outside the loop, at lines 2–5 in Figure 2-3 on page 27, where they are declared.

```
2  uint32_t length;  // the Length field (4 bytes)
3  uint8_t type[4];  // the Type field (4 bytes)
4  uint8_t *data;  // the Data field (Length bytes)
5  uint32_t crc;  // the CRC field (4 bytes)
```

To obtain such missing information, the system uses an alias analysis rather than always requiring a developer to manually provide a loop invariant or waiting

for the specification being propagated up to the start of the program. For each pair of memory locations appearing in the generated condition, the system inquires of an alias analysis whether they may point to the same object in memory. For instance, $\mathcal{M}\{\texttt{crc} \mapsto \mathcal{C}(\texttt{fd}, \mathcal{P}(\texttt{fd}))\}(\texttt{length})$ is simplified into $\mathcal{M}(\texttt{length})$ when the alias analysis figures out that the memory locations `crc` and `length` never overlap.

After the simplification based on the alias information, the system checks if the old condition stored at the program point, which is "true" at this moment, implies the new simplified condition. To do so, the simplified condition is fed into Why3, a platform for deductive program verification [18]. For this example, Why3 successfully proved the validity of the simplified condition with its theories of integers and maps, using the Z3 SMT solver as its external prover. Therefore, the simplified condition does not need to be propagated further.

Once the system calculates what condition at each program point should be satisfied to guarantee the input specification, it checks the validity of the resulting condition at the start of the program. For this example, the resulting condition is "true" because the input specification has already been resolved before being propagated up to the start of the program.

In this way, the proposed system successfully verified all the specifications in Figures 2-3 and 2-4 on pages 27 and 29, and therefore confirmed that:

- The image viewer has examined whether the image file contains the correct PNG signature, as the PNG standard requires.
- At the end of each iteration of the main loop, the program has consumed all the bytes of the current chunk and is ready to read the next chunk with the start of the next iteration.

Chapter 3

Core Language

This chapter presents the syntax and semantics of a programming language that, in subsequent chapters, I will use to formally define a specification language and show the soundness of the verification process of input specifications written in the specification language. In addition to the standard constructs of imperative languages, the language supports file I/O operations and byte-wise operations such as endianness conversion, which are almost always used in writing parsers for binary-file formats.

3.1 Syntax

Figure 3-1 on page 52 presents the syntax of an imperative language with variables, arithmetic expressions, Boolean expressions, assignments, memory read/write, endianness conversion, file read/seek, conditional branches, loops, and sequential composition. A program is a sequence of statements.

Label. Every program point has a unique label $\ell \in \text{Label}$. For a statement S, $\mathsf{before}(S)$ and $\mathsf{after}(S)$ denote the labels (or program points) before and after the statement S, respectively. For a sequence $Q = S_1 ; \cdots ; S_n$, $\mathsf{before}(Q)$ and $\mathsf{after}(Q)$ are defined as the before label of the first statement S_1 and the after label of the last statement S_n: $\mathsf{before}(Q) = \mathsf{before}(S_1)$ and $\mathsf{after}(Q) = \mathsf{after}(S_n)$. Note that $\mathsf{after}(S_i)$ is the same as $\mathsf{before}(S_{i+1})$ in a sequence $S_1 ; \cdots ; S_n$ because they denote the identical

$$
\begin{array}{rcl}
n \in & \text{Int} & \\
x \in & \text{Var} & \\
\\
E \in \text{AExpr} & ::= & n \mid x \mid \texttt{-}E \mid E_1 \texttt{ + } E_2 \mid E_1 \texttt{ - } E_2 \mid \cdots \\
B \in \text{BExpr} & ::= & E_1 \texttt{ = } E_2 \mid E_1 \texttt{ != } E_2 \mid E_1 \texttt{ < } E_2 \mid E_1 \texttt{ <= } E_2 \mid \cdots \mid \\
& & \texttt{!}B \mid B_1 \texttt{ \&\& } B_2 \mid B_1 \texttt{ || } B_2 \\
S \in \text{Stmt} & ::= & \texttt{skip} \mid x \texttt{ := } E \mid \\
& & x \texttt{ := load(}E_1\texttt{, } E_2\texttt{)} \mid \texttt{store(}E_1\texttt{, } E_2\texttt{)} \mid x \texttt{ := hton(}E\texttt{, } n\texttt{)} \mid \\
& & x \texttt{ := read(}E_1\texttt{, } E_2\texttt{, } E_3\texttt{)} \mid x \texttt{ := seek(}E_1\texttt{, } E_2\texttt{)} \mid \\
& & \texttt{if } B \texttt{ then } Q_1 \texttt{ else } Q_2 \mid \texttt{while (}I\texttt{) } B\ Q \\
Q \in \text{Seq} & ::= & S_1\texttt{;}\ \cdots\texttt{;}S_n \\
\text{Pgm} & ::= & Q
\end{array}
$$

Figure 3-1: The syntax of an imperative language. Nonterminal symbol I represents an optional loop invariant and is defined in Figure 4-1 on page 66.

program point. $\mathsf{labels}(S)$ (resp., $\mathsf{labels}(Q)$) is defined as the set of all labels within a statement S (resp., a sequence Q), inclusive of its **before** and **after** labels.

3.2 Dynamic Semantics

This section provides the formal semantics of the core language. I first define the semantic domains of the language, and then present the meanings of its expressions and statements in terms of the semantic domains.

3.2.1 Semantic Domains

Figure 3-2 on page 53 presents the semantic domains of the core language. A *labeled program state* $\langle \ell, \sigma \rangle \in$ LState is a pair of an instruction pointer ℓ and a program state σ. An *instruction pointer* $\ell \in$ Label is the **before** label of the statement to be executed next. A *program state* $\sigma = \langle \rho, \mu, \phi, \kappa \rangle \in$ State consists of 1) an environment ρ, 2) a memory μ, 3) a file position mapping ϕ, and 4) a file contents mapping κ. An *environment* $\rho \in$ Env is a finite mapping from variables to values. A *memory* $\mu \in$ Mem receives an address as its argument and returns a byte value stored at the address. A *file position mapping* $\phi \in$ FilePos records the current file position associated with each

$$\begin{array}{rcl} & \text{LState} & = \text{Label} \times \text{State} \\ \sigma \in & \text{State} & = \text{Env} \times \text{Mem} \times \text{FilePos} \times \text{FileCont} \\ \rho \in & \text{Env} & = \text{Var} \xrightarrow{\text{fin}} \text{Val} \\ \mu \in & \text{Mem} & = \text{Addr} \to \text{Byte} \\ \phi \in & \text{FilePos} & = \text{FileDesc} \to \text{Pos} \\ \kappa \in & \text{FileCont} & = \text{FileDesc} \to \text{Pos} \to \text{Byte} \end{array}$$

$$o, s, v \in \text{Val} = \text{Int} \quad a \in \text{Addr} = \text{Int} \quad f \in \text{FileDesc} = \text{Int} \quad p \in \text{Pos} = \text{Int} \quad b \in \text{Bool}$$

Figure 3-2: The semantic domains.

file descriptor, and a *file contents mapping* $\kappa \in$ FileCont keeps track of a byte value stored at each position of a file. Values, addresses, file descriptors, and file positions are modeled as integers. In this and the following chapters, I describe all of them by a universal set Int for readability. The real implementation allocates an appropriate number of bytes to each of them as per the target machine architecture, and reflects the exact semantics of byte-level operations in the C programming language.

3.2.2 Semantics of Expressions

Figures 3-3 and 3-4 on page 54 present the semantics of arithmetic and Boolean expressions in the language, respectively. The relation $\rho \vdash E \Rightarrow v$ denotes that evaluating the arithmetic expression E under the environment ρ yields the integer value v, and follows the standard semantics of arithmetic expressions. Similarly, the relation $\rho \vdash B \Rightarrow b$ means that the Boolean expression B, which are composed of the standard comparison operators on integers (e.g. `=`, `!=`, `<`, `<=`, ...) and the standard Boolean operators (e.g. `!`, `&&`, `||`, ...), evaluates to the Boolean value b under the environment ρ.

3.2.3 Semantics of Statements

Statement Sequences. Figure 3-5 on page 54 presents the transition relation between a labeled program state and its resulting labeled program state by a one-step

$$\rho \vdash n \Rightarrow n \qquad \rho \vdash x \Rightarrow \rho(x) \qquad \frac{\rho \vdash E \Rightarrow v}{\rho \vdash \texttt{-}E \Rightarrow -v} \qquad \frac{\rho \vdash E_1 \Rightarrow v_1 \qquad \rho \vdash E_2 \Rightarrow v_2}{\rho \vdash E_1 \texttt{ + } E_2 \Rightarrow v_1 + v_2}$$

$$\frac{\rho \vdash E_1 \Rightarrow v_1 \qquad \rho \vdash E_2 \Rightarrow v_2}{\rho \vdash E_1 \texttt{ - } E_2 \Rightarrow v_1 - v_2}$$

Figure 3-3: The semantics of arithmetic expressions.

$$\frac{\rho \vdash E_1 \Rightarrow v_1 \qquad \rho \vdash E_2 \Rightarrow v_2}{\rho \vdash E_1 \texttt{ = } E_2 \Rightarrow v_1 = v_2} \qquad \frac{\rho \vdash E_1 \Rightarrow v_1 \qquad \rho \vdash E_2 \Rightarrow v_2}{\rho \vdash E_1 \texttt{ != } E_2 \Rightarrow v_1 \neq v_2}$$

$$\frac{\rho \vdash E_1 \Rightarrow v_1 \qquad \rho \vdash E_2 \Rightarrow v_2}{\rho \vdash E_1 \texttt{ < } E_2 \Rightarrow v_1 < v_2} \qquad \frac{\rho \vdash E_1 \Rightarrow v_1 \qquad \rho \vdash E_2 \Rightarrow v_2}{\rho \vdash E_1 \texttt{ <= } E_2 \Rightarrow v_1 \leq v_2} \qquad \frac{\rho \vdash B \Rightarrow b}{\rho \vdash \texttt{!}B \Rightarrow \neg b}$$

$$\frac{\rho \vdash B_1 \Rightarrow b_1 \qquad \rho \vdash B_2 \Rightarrow b_2}{\rho \vdash B_1 \texttt{ \&\& } B_2 \Rightarrow b_1 \wedge b_2} \qquad \frac{\rho \vdash B_1 \Rightarrow b_1 \qquad \rho \vdash B_2 \Rightarrow b_2}{\rho \vdash B_1 \texttt{ || } B_2 \Rightarrow b_1 \vee b_2}$$

Figure 3-4: The semantics of Boolean expressions.

$$\frac{\ell = \mathsf{before}(S_i) \qquad \langle \ell, \sigma \rangle =\!\!\llbracket\, S_i \,\rrbracket\!\!\Rightarrow_{\mathrm{Stmt}} \langle \ell', \sigma' \rangle}{\langle \ell, \sigma \rangle =\!\!\llbracket\, S_1 \texttt{;} \; \cdots \texttt{;} \; S_n \,\rrbracket\!\!\Rightarrow_{\mathsf{Seq}} \langle \ell', \sigma' \rangle}$$

Figure 3-5: The semantics of a sequence of statements.

execution of a program (or a sequence of statements). The transition relation $\langle \ell, \sigma \rangle$ $=\!\![S_1\,;\ \cdots\ ;\ S_n]\!\!\Rightarrow_{\text{Seq}} \langle \ell', \sigma' \rangle$ denotes that the current labeled state $\langle \ell, \sigma \rangle$ is reduced to the new labeled state $\langle \ell', \sigma' \rangle$ after a *one-step* execution of the S_i statement denoted by the current instruction pointer ℓ. Figures 3-6, 3-7, and 3-8 on pages 56 and 60–61 present a one-step execution of each statement.

Environment/Memory-Related Statements. Figure 3-6 on page 56 presents the transition relation of a one-step execution of environment/memory-related statements. In each transition relation, the instruction pointers ℓ and ℓ' are the labels for the program points before and after the statement being considered, respectively.

The rules for `skip` and assignment statements follow the standard semantics of these constructs (SKIP and ASSIGN).

$$\boxed{\text{SKIP}}$$
$$\langle \ell, \sigma \rangle =\!\![\ \texttt{skip}\ ]\!\!\Rightarrow_{\text{Stmt}} \langle \ell', \sigma \rangle$$

$$\boxed{\text{ASSIGN}}$$
$$\frac{\rho \vdash E \Rightarrow v}{\langle \ell, \langle \rho, \mu, \phi, \kappa \rangle \rangle =\!\![\ x\ \texttt{:=}\ E\ ]\!\!\Rightarrow_{\text{Stmt}} \langle \ell', \langle \rho[x \mapsto v], \mu, \phi, \kappa \rangle \rangle}$$

The x `:=` `load(`E_1`,` E_2`)` statement interprets a memory region of E_2 bytes pointed to by E_1 in accordance with the endianness of the machine and assigns the resulting integer value to the variable x. For this presentation, I assume that the target machine is little-endian, so the least significant byte is stored at the lowest memory address and the most significant byte at the highest memory address (LOAD).

$$\boxed{\text{LOAD}}$$
$$\frac{\rho \vdash E_1 \Rightarrow a \qquad \rho \vdash E_2 \Rightarrow s \qquad s > 0 \qquad v = \sum_{i=0}^{s-1} (\mu(a+i) \times (2^8)^i)}{\langle \ell, \langle \rho, \mu, \phi, \kappa \rangle \rangle =\!\![\ x\ \texttt{:=}\ \texttt{load(}E_1\texttt{,}\ E_2\texttt{)}\ ]\!\!\Rightarrow_{\text{Stmt}} \langle \ell', \langle \rho[x \mapsto v], \mu, \phi, \kappa \rangle \rangle}$$

The `load` operation can be understood as the generalization of dereferencing pointers to unsigned integers of variable sizes in the C language.

The `store(`E_1`,` E_2`,` E_3`)` statement stores the lowest E_3 bytes of the nonnegative

In each transition relation, ℓ and ℓ' are the labels for the program points before and after the relevant statement S, respectively: $\ell = \mathsf{before}(S)$ and $\ell' = \mathsf{after}(S)$.

SKIP

$$\langle \ell, \sigma \rangle =\!\![\!\![\ \texttt{skip}\ ]\!\!]\!\!\Rightarrow_{\text{Stmt}} \langle \ell', \sigma \rangle$$

ASSIGN

$$\frac{\rho \vdash E \Rightarrow v}{\langle \ell, \langle \rho, \mu, \phi, \kappa \rangle \rangle =\!\![\!\![\ x\ \texttt{:=}\ E\ ]\!\!]\!\!\Rightarrow_{\text{Stmt}} \langle \ell', \langle \rho[x \mapsto v], \mu, \phi, \kappa \rangle \rangle}$$

LOAD

$$\frac{\rho \vdash E_1 \Rightarrow a \qquad \rho \vdash E_2 \Rightarrow s \qquad s > 0 \qquad v = \sum_{i=0}^{s-1} (\mu(a+i) \times (2^8)^i)}{\langle \ell, \langle \rho, \mu, \phi, \kappa \rangle \rangle =\!\![\!\![\ x\ \texttt{:=}\ \texttt{load(}E_1\texttt{,}\ E_2\texttt{)}\ ]\!\!]\!\!\Rightarrow_{\text{Stmt}} \langle \ell', \langle \rho[x \mapsto v], \mu, \phi, \kappa \rangle \rangle}$$

STORE

$$\frac{\begin{array}{c}\rho \vdash E_1 \Rightarrow v \qquad v \geq 0 \qquad \rho \vdash E_2 \Rightarrow a \qquad \rho \vdash E_3 \Rightarrow s \qquad s > 0 \\ \mu' = \mu[a \mapsto v \bmod 2^8][a+1 \mapsto (v/2^8) \bmod 2^8] \cdots [a + (s-1) \mapsto (v/(2^8)^{s-1}) \bmod 2^8]\end{array}}{\langle \ell, \langle \rho, \mu, \phi, \kappa \rangle \rangle =\!\![\!\![\ \texttt{store(}E_1\texttt{,}\ E_2\texttt{,}\ E_3\texttt{)}\ ]\!\!]\!\!\Rightarrow_{\text{Stmt}} \langle \ell', \langle \rho, \mu', \phi, \kappa \rangle \rangle}$$

HTON

$$\frac{\rho \vdash E \Rightarrow a \qquad n > 0 \qquad v = \sum_{i=0}^{n-1} (\mu(a+i) \times (2^8)^{n-1-i})}{\langle \ell, \langle \rho, \mu, \phi, \kappa \rangle \rangle =\!\![\!\![\ x\ \texttt{:=}\ \texttt{hton(}E\texttt{,}\ n\texttt{)}\ ]\!\!]\!\!\Rightarrow_{\text{Stmt}} \langle \ell', \langle \rho[x \mapsto v], \mu, \phi, \kappa \rangle \rangle}$$

Figure 3-6: The semantics of environment/memory-related statements.

integer value E_1 into a memory region starting at address E_2 in the little-endian format. The i-th least-significant byte of a nonnegative integer value v is obtained by evaluating the "$(v/(2^8)^{i-1}) \bmod 2^8$" expression (STORE).

$$\boxed{\text{STORE}}$$

$$\frac{\rho \vdash E_1 \Rightarrow v \qquad v \geq 0 \qquad \rho \vdash E_2 \Rightarrow a \qquad \rho \vdash E_3 \Rightarrow s \qquad s > 0 \qquad \mu' = \mu[a \mapsto v \bmod 2^8][a+1 \mapsto (v/2^8) \bmod 2^8]\cdots[a+(s-1) \mapsto (v/(2^8)^{s-1}) \bmod 2^8]}{\langle \ell, \langle \rho, \mu, \phi, \kappa \rangle\rangle \;=\!\![\!\![\; \texttt{store(}E_1\texttt{, }E_2\texttt{, }E_3\texttt{)} \;]\!\!]\!\!\Rightarrow_{\text{Stmt}} \langle \ell', \langle \rho, \mu', \phi, \kappa \rangle\rangle}$$

Similar to the `load` statement, the x `:=` `hton(`E`, `n`)` statement interprets a memory region of n bytes pointed to by E and sets the variable x to the resulting integer value. The `hton` statement, however, interprets the memory region in accordance with big-endianness rather than little-endianness (i.e. the endianness of the target machine). In other words, the byte at the lowest memory address is interpreted as the most significant one (HTON).

$$\boxed{\text{HTON}}$$

$$\frac{\rho \vdash E \Rightarrow a \qquad n > 0 \qquad v = \sum_{i=0}^{n-1} (\mu(a+i) \times (2^8)^{n-1-i})}{\langle \ell, \langle \rho, \mu, \phi, \kappa \rangle\rangle \;=\!\![\!\![\; x \texttt{ := hton(}E\texttt{, }n\texttt{)} \;]\!\!]\!\!\Rightarrow_{\text{Stmt}} \langle \ell', \langle \rho[x \mapsto v], \mu, \phi, \kappa \rangle\rangle}$$

The `hton` statement has been introduced so that the POSIX byte order functions, such as `htonl` and `ntohl`, which convert unsigned integer values between host byte order and network byte order, can be simulated. Those functions are commonly used in programs dealing with binary file formats.

File-Related Statements. Figure 3-7 on page 58 presents the transition relation of file `read`/`seek` statements. For each transition relation, the instruction pointers ℓ and ℓ' are the labels for the program points before and after the statement being considered, respectively. These `read` and `seek` functions are carefully modeled on the POSIX `read` and `lseek` functions.

The x `:=` `read(`E_1`, `E_2`, `E_3`)` statement attempts to read E_3 bytes from the file E_1 into the memory pointed to by E_2 and assigns the number of bytes actually read to the variable x. The `read` statement also moves the file position of the file E_1 by the

In each transition relation, ℓ and ℓ' are the labels for the program points before and after the relevant statement S, respectively: $\ell = \mathsf{before}(S)$ and $\ell' = \mathsf{after}(S)$.

READ-1

$$\frac{\rho \vdash E_1 \Rightarrow f \qquad \rho \vdash E_2 \Rightarrow a \qquad \rho \vdash E_3 \Rightarrow s \qquad 0 \leq s' \leq s \qquad \mu' = \mu[a \mapsto \kappa(f)(\phi(f))][a+1 \mapsto \kappa(f)(\phi(f)+1)] \cdots [a+(s'-1) \mapsto \kappa(f)(\phi(f)+(s'-1))]}{\langle \ell, \langle \rho, \mu, \phi, \kappa \rangle\rangle =\!\!\llbracket\ x\ \texttt{:= read(}E_1\texttt{,}\ E_2\texttt{,}\ E_3\texttt{)}\ \rrbracket\!\!\Rightarrow_{\text{Stmt}} \langle \ell', \langle \rho[x \mapsto s'], \mu', \phi[f \mapsto \phi(f)+s'], \kappa \rangle\rangle}$$

READ-2

$$\frac{\rho \vdash E_1 \Rightarrow f \qquad \rho \vdash E_2 \Rightarrow a \qquad \rho \vdash E_3 \Rightarrow s \qquad 0 \leq s' \leq s \qquad \mu' = \mu[a \mapsto \kappa(f)(\phi(f))][a+1 \mapsto \kappa(f)(\phi(f)+1)] \cdots [a+(s'-1) \mapsto \kappa(f)(\phi(f)+(s'-1))]}{\langle \ell, \langle \rho, \mu, \phi, \kappa \rangle\rangle =\!\!\llbracket\ x\ \texttt{:= read(}E_1\texttt{,}\ E_2\texttt{,}\ E_3\texttt{)}\ \rrbracket\!\!\Rightarrow_{\text{Stmt}} \langle \ell', \langle \rho[x \mapsto -1], \mu', \phi[f \mapsto \phi(f)+s'], \kappa \rangle\rangle}$$

SEEK-1

$$\frac{\rho \vdash E_1 \Rightarrow f \qquad \rho \vdash E_2 \Rightarrow o \qquad \phi(f) + o \geq 0}{\langle \ell, \langle \rho, \mu, \phi, \kappa \rangle\rangle =\!\!\llbracket\ x\ \texttt{= seek(}E_1\texttt{,}\ E_2\texttt{)}\ \rrbracket\!\!\Rightarrow_{\text{Stmt}} \langle \ell', \langle \rho[x \mapsto \phi(f)+o], \mu, \phi[f \mapsto \phi(f)+o], \kappa \rangle\rangle}$$

SEEK-2

$$\langle \ell, \langle \rho, \mu, \phi, \kappa \rangle\rangle =\!\!\llbracket\ x\ \texttt{= seek(}E_1\texttt{,}\ E_2\texttt{)}\ \rrbracket\!\!\Rightarrow_{\text{Stmt}} \langle \ell', \langle \rho[x \mapsto -1], \mu, \phi, \kappa \rangle\rangle$$

Figure 3-7: The semantics of file-related statements.

number of bytes read (READ-1).

READ-1

$$\frac{\rho \vdash E_1 \Rightarrow f \qquad \rho \vdash E_2 \Rightarrow a \qquad \rho \vdash E_3 \Rightarrow s \qquad 0 \le s' \le s \\ \mu' = \mu[a \mapsto \kappa(f)(\phi(f))][a+1 \mapsto \kappa(f)(\phi(f)+1)] \cdots [a+(s'-1) \mapsto \kappa(f)(\phi(f)+(s'-1))]}{\langle \ell, \langle \rho, \mu, \phi, \kappa \rangle \rangle =\!\!\llbracket\ x \texttt{ := read(}E_1\texttt{, }E_2\texttt{, }E_3\texttt{)} \rrbracket\!\!\Rightarrow_{\text{Stmt}} \langle \ell', \langle \rho[x \mapsto s'], \mu', \phi[f \mapsto \phi(f)+s'], \kappa \rangle \rangle}$$

The number s' of bytes actually read into the memory can be arbitrary because no assumptions are made about the sizes of files. In particular, the `read` statement can read zero bytes and set the variable x to zero to indicate that the file has already reached the end-of-file.

On the other hand, the `read` statement may fail nondeterministically due to some external errors (e.g. a physical I/O error) and assigns -1 to the variable, even after the `read` statement has read some bytes successfully (READ-2).

READ-2

$$\frac{\rho \vdash E_1 \Rightarrow f \qquad \rho \vdash E_2 \Rightarrow a \qquad \rho \vdash E_3 \Rightarrow s \qquad 0 \le s' \le s \\ \mu' = \mu[a \mapsto \kappa(f)(\phi(f))][a+1 \mapsto \kappa(f)(\phi(f)+1)] \cdots [a+(s'-1) \mapsto \kappa(f)(\phi(f)+(s'-1))]}{\langle \ell, \langle \rho, \mu, \phi, \kappa \rangle \rangle =\!\!\llbracket\ x \texttt{ := read(}E_1\texttt{, }E_2\texttt{, }E_3\texttt{)} \rrbracket\!\!\Rightarrow_{\text{Stmt}} \langle \ell', \langle \rho[x \mapsto -1], \mu', \phi[f \mapsto \phi(f)+s'], \kappa \rangle \rangle}$$

The x `:= seek(`E_1`, `E_2`)` statement advances the file position of the file E_1 by the offset E_2 (SEEK-1).

SEEK-1

$$\frac{\rho \vdash E_1 \Rightarrow f \qquad \rho \vdash E_2 \Rightarrow o \qquad \phi(f)+o \ge 0}{\langle \ell, \langle \rho, \mu, \phi, \kappa \rangle \rangle =\!\!\llbracket\ x \texttt{ = seek(}E_1\texttt{, }E_2\texttt{)} \rrbracket\!\!\Rightarrow_{\text{Stmt}} \langle \ell', \langle \rho[x \mapsto \phi(f)+o], \mu, \phi[f \mapsto \phi(f)+o], \kappa \rangle \rangle}$$

Like the POSIX `lseek` function, the file position is allowed to be set beyond the end of the file but cannot be negative. Note that the `seek` function returns the current file position when the offset E_2 is zero.

If the resulting file position ends up being negative or some other error occurs (e.g. the device associated with the file does not support the seek operation), the variable x is set to -1 with the file position intact (SEEK-2).

SEEK-2

$$\langle \ell, \langle \rho, \mu, \phi, \kappa \rangle \rangle =\!\!\llbracket\ x \texttt{ := seek(}E_1\texttt{, }E_2\texttt{)} \rrbracket\!\!\Rightarrow_{\text{Stmt}} \langle \ell', \langle \rho[x \mapsto -1], \mu, \phi, \kappa \rangle \rangle$$

In each transition relation, S denotes the whole `if` statement in question.

IF-1

$$\frac{\ell = \mathsf{before}(S) \qquad \rho \vdash B \Rightarrow \mathsf{true}}{\langle \ell, \langle \rho, \mu, \phi, \kappa \rangle \rangle =\!\!\llbracket\ \texttt{if}\ B\ \texttt{then}\ Q_1\ \texttt{else}\ Q_2\ \rrbracket\!\!\Rightarrow_{\text{Stmt}} \langle \mathsf{before}(Q_1), \langle \rho, \mu, \phi, \kappa \rangle \rangle}$$

IF-2

$$\frac{\ell = \mathsf{before}(S) \qquad \rho \vdash B \Rightarrow \mathsf{false}}{\langle \ell, \langle \rho, \mu, \phi, \kappa \rangle \rangle =\!\!\llbracket\ \texttt{if}\ B\ \texttt{then}\ Q_1\ \texttt{else}\ Q_2\ \rrbracket\!\!\Rightarrow_{\text{Stmt}} \langle \mathsf{before}(Q_2), \langle \rho, \mu, \phi, \kappa \rangle \rangle}$$

IF-3

$$\frac{\ell \in \mathsf{labels}(Q_i) \setminus \{\mathsf{after}(Q_i)\} \qquad \langle \ell, \sigma \rangle =\!\!\llbracket\ Q_i\ \rrbracket\!\!\Rightarrow_{\text{Seq}} \langle \ell', \sigma' \rangle}{\langle \ell, \sigma \rangle =\!\!\llbracket\ \texttt{if}\ B\ \texttt{then}\ Q_1\ \texttt{else}\ Q_2\ \rrbracket\!\!\Rightarrow_{\text{Stmt}} \langle \ell', \sigma' \rangle}$$

IF-4

$$\frac{\ell = \mathsf{after}(Q_i)}{\langle \ell, \sigma \rangle =\!\!\llbracket\ \texttt{if}\ B\ \texttt{then}\ Q_1\ \texttt{else}\ Q_2\ \rrbracket\!\!\Rightarrow_{\text{Stmt}} \langle \mathsf{after}(S), \sigma \rangle}$$

Figure 3-8: The semantics of an `if` statement.

If Statements. Figure 3-8 presents the transition relation of `if` statements. In each transition relation, the S statement is a shorthand for the whole `if` statement in question. When the current instruction pointer ℓ is at the beginning of an `if` B `then` Q_1 `else` Q_2 statement, it jumps to the statement sequence Q_1 or Q_2, depending on whether the value of the Boolean expression B is true or false (IF-1 and IF-2).

IF-1

$$\frac{\ell = \mathsf{before}(S) \qquad \rho \vdash B \Rightarrow \mathsf{true}}{\langle \ell, \langle \rho, \mu, \phi, \kappa \rangle \rangle =\!\!\llbracket\ \texttt{if}\ B\ \texttt{then}\ Q_1\ \texttt{else}\ Q_2\ \rrbracket\!\!\Rightarrow_{\text{Stmt}} \langle \mathsf{before}(Q_1), \langle \rho, \mu, \phi, \kappa \rangle \rangle}$$

IF-2

$$\frac{\ell = \mathsf{before}(S) \qquad \rho \vdash B \Rightarrow \mathsf{false}}{\langle \ell, \langle \rho, \mu, \phi, \kappa \rangle \rangle =\!\!\llbracket\ \texttt{if}\ B\ \texttt{then}\ Q_1\ \texttt{else}\ Q_2\ \rrbracket\!\!\Rightarrow_{\text{Stmt}} \langle \mathsf{before}(Q_2), \langle \rho, \mu, \phi, \kappa \rangle \rangle}$$

While the current instruction pointer ℓ stays within either branch of an `if` statement, the branch is executed *one-step* at a time until the instruction pointer reaches the

In each transition relation, S denotes the whole `while` statement in question.

WHILE-1

$$\frac{\ell = \mathsf{before}(S) \qquad \rho \vdash B \Rightarrow \text{true}}{\langle \ell, \langle \rho, \mu, \phi, \kappa \rangle \rangle =\!\!\![\![\texttt{while (}I\texttt{)}\ B\ Q\ ]\!]\!\!\Rightarrow_{\text{Stmt}} \langle \mathsf{before}(Q), \langle \rho, \mu, \phi, \kappa \rangle \rangle}$$

WHILE-2

$$\frac{\ell = \mathsf{before}(S) \qquad \rho \vdash B \Rightarrow \text{false}}{\langle \ell, \langle \rho, \mu, \phi, \kappa \rangle \rangle =\!\!\![\![\texttt{while (}I\texttt{)}\ B\ Q\ ]\!]\!\!\Rightarrow_{\text{Stmt}} \langle \mathsf{after}(S), \langle \rho, \mu, \phi, \kappa \rangle \rangle}$$

WHILE-3

$$\frac{\ell \in \mathsf{labels}(Q) \setminus \{\mathsf{after}(Q)\} \qquad \langle \ell, \sigma \rangle =\!\!\![\![Q]\!]\!\!\Rightarrow_{\text{Seq}} \langle \ell', \sigma' \rangle}{\langle \ell, \sigma \rangle =\!\!\![\![\texttt{while (}I\texttt{)}\ B\ Q\ ]\!]\!\!\Rightarrow_{\text{Stmt}} \langle \ell', \sigma' \rangle \rangle}$$

WHILE-4

$$\frac{\ell = \mathsf{after}(Q)}{\langle \ell, \sigma \rangle =\!\!\![\![\texttt{while (}I\texttt{)}\ B\ Q\ ]\!]\!\!\Rightarrow_{\text{Stmt}} \langle \mathsf{before}(S), \sigma \rangle}$$

Figure 3-9: The semantics of a `while` statement.

end of the branch (IF-3).

IF-3

$$\frac{\ell \in \mathsf{labels}(Q_i) \setminus \{\mathsf{after}(Q_i)\} \qquad \langle \ell, \sigma \rangle =\!\!\![\![Q_i]\!]\!\!\Rightarrow_{\text{Seq}} \langle \ell', \sigma' \rangle}{\langle \ell, \sigma \rangle =\!\!\![\![\texttt{if}\ B\ \texttt{then}\ Q_1\ \texttt{else}\ Q_2\ ]\!]\!\!\Rightarrow_{\text{Stmt}} \langle \ell', \sigma' \rangle}$$

After either branch has been executed to the end, the instruction pointer jumps to the program point after the whole `if` statement (IF-4).

IF-4

$$\frac{\ell = \mathsf{after}(Q_i)}{\langle \ell, \sigma \rangle =\!\!\![\![\texttt{if}\ B\ \texttt{then}\ Q_1\ \texttt{else}\ Q_2\ ]\!]\!\!\Rightarrow_{\text{Stmt}} \langle \mathsf{after}(S), \sigma \rangle}$$

While Statements. Figure 3-9 presents the transition relation of `while` statements. In each transition relation, the S statement is a shorthand for the whole `while` statement in question. When the current instruction pointer ℓ is at the beginning of a `while (`I`)` B Q statement, it enters the loop body Q or exits by jumping to the program point after the whole `while` statement, depending on whether the value of

the Boolean expression B is true or false (WHILE-1 and WHILE-2). The optional loop invariant I is used when generating verification conditions in Chapter 5.

$$\boxed{\text{WHILE-1}}\quad \frac{\ell = \mathsf{before}(S) \qquad \rho \vdash B \Rightarrow \text{true}}{\langle \ell, \langle \rho, \mu, \phi, \kappa \rangle\rangle =\!\!\llbracket\ \texttt{while (}I\texttt{)}\ B\ Q\ \rrbracket\!\!\Rightarrow_{\text{Stmt}} \langle \mathsf{before}(Q), \langle \rho, \mu, \phi, \kappa \rangle\rangle}$$

$$\boxed{\text{WHILE-2}}\quad \frac{\ell = \mathsf{before}(S) \qquad \rho \vdash B \Rightarrow \text{false}}{\langle \ell, \langle \rho, \mu, \phi, \kappa \rangle\rangle =\!\!\llbracket\ \texttt{while (}I\texttt{)}\ B\ Q\ \rrbracket\!\!\Rightarrow_{\text{Stmt}} \langle \mathsf{after}(S), \langle \rho, \mu, \phi, \kappa \rangle\rangle}$$

While the current instruction pointer ℓ stays within the loop body Q, the loop body is executed *one-step* at a time until the instruction pointer reaches the end of the loop body (WHILE-3).

$$\boxed{\text{WHILE-3}}\quad \frac{\ell \in \mathsf{labels}(Q) \setminus \{\mathsf{after}(Q)\} \qquad \langle \ell, \sigma \rangle =\!\!\llbracket\ Q\ \rrbracket\!\!\Rightarrow_{\text{Seq}} \langle \ell', \sigma' \rangle}{\langle \ell, \sigma \rangle =\!\!\llbracket\ \texttt{while (}I\texttt{)}\ B\ Q\ \rrbracket\!\!\Rightarrow_{\text{Stmt}} \langle \ell', \sigma' \rangle\rangle}$$

After the loop body has been executed to the end, the instruction pointer jumps back to the beginning of the `while` loop (WHILE-4).

$$\boxed{\text{WHILE-4}}\quad \frac{\ell = \mathsf{after}(Q)}{\langle \ell, \sigma \rangle =\!\!\llbracket\ \texttt{while (}I\texttt{)}\ B\ Q\ \rrbracket\!\!\Rightarrow_{\text{Stmt}} \langle \mathsf{before}(S), \sigma \rangle}$$

3.2.4 Semantics of Programs

The semantics $\llbracket Q \rrbracket \in \mathcal{P}(\text{Trace})$ of a program Q is defined as a set of maximal traces of the program's execution:

$$\begin{aligned}\llbracket Q \rrbracket = \{&\langle \ell_1, \sigma_1 \rangle \cdots \langle \ell_m, \sigma_m \rangle \in \text{LState}^+ \mid \\ &(\forall 1 \le i < m : \langle \ell_i, \sigma_i \rangle =\!\!\llbracket\ Q\ \rrbracket\!\!\Rightarrow_{\text{Seq}} \langle \ell_{i+1}, \sigma_{i+1} \rangle) \wedge \langle \ell_m, \sigma_m \rangle \in \text{Fin}\} \cup \\ \{&\langle \ell_1, \sigma_1 \rangle \cdots \in \text{LState}^\omega \mid \forall i \ge 1 : \langle \ell_i, \sigma_i \rangle =\!\!\llbracket\ Q\ \rrbracket\!\!\Rightarrow_{\text{Seq}} \langle \ell_{i+1}, \sigma_{i+1} \rangle\}\end{aligned}$$

where Fin is a set of final program states:

$$\text{Fin} = \{\langle \ell, \sigma \rangle \mid \neg\exists \langle \ell', \sigma' \rangle \in \text{LState} : \langle \ell, \sigma \rangle =\!\llbracket\, Q \,\rrbracket\!\Rightarrow_{\text{Seq}} \langle \ell', \sigma' \rangle\}$$

and LState^{+} (resp., LState^{ω}) is a non-empty finite (resp., countably infinite) sequence of labeled program states.

Chapter 4

Specification Language

This chapter formally presents the syntax and semantics of a specification language that makes it possible for a developer to specify the relationship between a program and its input file format. By the specification language, a developer can specify 1) how many bytes a program is supposed to consume between two program points and 2) what byte values need to be observed at specific offsets in an input file.

4.1 Syntax

I use a form of first-order logic as a language for specifying the relationship between a program and its input file format, and its syntax is given in Figure 4-1 on page 66. As is usually the case with first-order logic, the terms of the specification language are made up of constants (i.e. n, μ, ϕ, and κ), variables (i.e. x, $\mathcal{M}$, $\mathcal{P}$, and $\mathcal{C}$), and functions applied to them. Note that μ, ϕ, and κ are used here to represent symbols that uniquely identify each element of Mem, FilePos, and FileCont, repectively. The set of predicate symbols consists of the standard comparison operators (i.e. $=$, $\neq$, $<$, $\leq$, $\ldots$). As shorthand for logical connectives, I will also use the following notation:

$$F_1 \vee F_2 \stackrel{\text{def}}{=} \neg(\neg F_1 \wedge \neg F_2)$$

$$F_1 \rightarrow F_2 \stackrel{\text{def}}{=} \neg F_1 \vee F_2$$

$$
\begin{array}{rlcl}
T \in & \text{Term} & ::= & n \mid x \mid -T \mid T_1 + T_2 \mid T_1 - T_2 \mid \cdots \mid \\
 & & & M(T) \mid M(T_1, T_2) \mid P(T) \mid C(T_1, T_2) \\
M \in & \text{Mem}_\text{s} & ::= & \mu \mid \mathcal{M} \mid M\{T_1 \mapsto T_2\} \mid M\{\langle T_1, T_2 \rangle \mapsto T_3\} \mid M\{\langle T_1, T_2 \rangle \mapsto C(T_3, T_4, T_2)\} \\
P \in & \text{FilePos}_\text{s} & ::= & \phi \mid \mathcal{P} \mid P\{T_1 \mapsto T_2\} \\
C \in & \text{FileCont}_\text{s} & ::= & \kappa \mid \mathcal{C} \\
F, I \in & \text{Formula} & ::= & \text{true} \mid \text{false} \mid T_1 = T_2 \mid T_1 < T_2 \mid \cdots \mid \\
 & & & \neg F \mid F_1 \wedge F_2 \mid \forall x : F \mid \forall \mathcal{M} : F \mid \forall \mathcal{P} : F \mid \forall \mathcal{C} : F
\end{array}
$$

Figure 4-1: The syntax of a specification language.

4.2 Semantics

Figure 4-2 on page 67 presents the semantics of terms in the specification language, which is defined over the semantic domains in Section 3.2.1. The denotation $[\![T]\!]_{\text{Term}}$ of a term T is a function mapping a program state $\sigma \in \text{State}$ to an integer. The definitions for an integer constant n, an integer variable x, and unary/binary arithmetic operations follow the standard semantics of those terms. The $M(T)$ term means the single byte value at address T of a memory denoted by M. The $M(T_1, T_2)$ term interprets a memory region of T_2 bytes in little-endian order, starting at address T_1 of a memory denoted by M. The $P(T)$ term obtains the current position in file T from a file position mapping denoted by P. The $C(T_1, T_2)$ term denotes the byte value at offset T_2 from the start of file T_1.

Figure 4-3 on page 68 presents the semantics of memory-related constants, variables, and functions. Given $M \in \text{Mem}_\text{s}$, its denotation $[\![M]\!]_{\text{Mem}}$ is defined as a function from a program state to a memory. Constant μ always denotes a specific memory of the Mem domain identified by μ. On the other hand, the denotation $[\![\mathcal{M}]\!]_{\text{Mem}}$ of variable $\mathcal{M}$ is the memory component of a program state given as an argument. The $M\{T_1 \mapsto T_2\}$ function stands for the same memory as M except that the former has value T_2 at address T_1. The $M\{\langle T_1, T_2 \rangle \mapsto T_3\}$ function is used to express concisely that the

$$[\![\cdot]\!]_{\text{Term}} \in \text{Term} \to \text{State} \to \text{Int}$$

$$\begin{aligned}
[\![n]\!]_{\text{Term}}(\sigma) &= n \\
[\![x]\!]_{\text{Term}}(\langle \rho, \mu, \phi, \kappa \rangle) &= \rho(x) \\
[\![-T]\!]_{\text{Term}}(\sigma) &= -[\![T]\!]_{\text{Term}}(\sigma) \\
[\![T_1 + T_2]\!]_{\text{Term}}(\sigma) &= [\![T_1]\!]_{\text{Term}}(\sigma) + [\![T_2]\!]_{\text{Term}}(\sigma) \\
[\![T_1 - T_2]\!]_{\text{Term}}(\sigma) &= [\![T_1]\!]_{\text{Term}}(\sigma) - [\![T_2]\!]_{\text{Term}}(\sigma) \\
[\![M(T)]\!]_{\text{Term}}(\sigma) &= ([\![M]\!]_{\text{Mem}}(\sigma))([\![T]\!]_{\text{Term}}(\sigma)) \\
[\![M(T_1, T_2)]\!]_{\text{Term}}(\sigma) &= \sum_{i=0}^{[\![T_2]\!]_{\text{Term}}(\sigma)-1} ([\![M]\!]_{\text{Mem}}(\sigma))([\![T_1]\!]_{\text{Term}}(\sigma) + i) \times (2^8)^i \\
[\![P(T)]\!]_{\text{Term}}(\sigma) &= ([\![P]\!]_{\text{Pos}}(\sigma))([\![T]\!]_{\text{Term}}(\sigma)) \\
[\![C(T_1, T_2)]\!]_{\text{Term}}(\sigma) &= ([\![C]\!]_{\text{Cont}})([\![T_1]\!]_{\text{Term}}(\sigma))([\![T_2]\!]_{\text{Term}}(\sigma))
\end{aligned}$$

Figure 4-2: The semantics of Term.

values of a memory region of T_2 bytes starting at address T_1 have been obtained by storing the lowest bytes of value T_3 in the little-endian format. Similarly, the $M\{\langle T_1, T_2\rangle \mapsto \mathcal{C}(T_3, T_4, T_2)\}$ function indicates that the T_2-bytes data from offset T_4 of file T_3 have been read into a memory region pointed to by T_1.

Figures 4-4 and 4-5 on page 68 give the semantics of file position-related and file contents-related constructs, respectively. The denotation $[\![P]\!]_{\text{Pos}}$ of a file position-related construct P is a function that associates a program state to a file position mapping. Constant ϕ stands for an element of FilePos uniquely identified by ϕ itself. In contrast, the $\mathcal{P}$ variable denotes a file position mapping which is a component of a program state given as an argument. The $P\{T_1 \mapsto T_2\}$ function means a file position mapping where the file position indicator of file T_1 has been set to the integer value T_2. The denotation $[\![C]\!]_{\text{Cont}}$ of a file contents-related construct C is defined similarly.

Figure 4-6 on page 69 presents the semantics of formulas in the specification language. The denotation $[\![F]\!]_{\text{Form}}$ of a logical formula F is the set of program states that satisfy the formula F. The semantics $[\![\cdot]\!]_{\text{Form}}$ for formulas that involve the standard comparison operators reuse the semantics $[\![\cdot]\!]_{\text{Term}}$ of terms in Figure 4-2 on page 67. The definitions for the logical connectives, $\neg$ and $\wedge$, and the universal quantifier over variables x, $\mathcal{M}$, $\mathcal{P}$, and $\mathcal{C}$ follow their standard semantics.

$$[\![\cdot]\!]_{\text{Mem}} \in \text{Mem}_{\text{s}} \to \text{State} \to \text{Mem}$$

$$\begin{aligned}
[\![\mu]\!]_{\text{Mem}}(\sigma) &= \mu \\
[\![\mathcal{M}]\!]_{\text{Mem}}(\langle \rho, \mu, \phi, \kappa \rangle) &= \mu \\
[\![M\{T_1 \mapsto T_2\}]\!]_{\text{Mem}}(\sigma) &= ([\![M]\!]_{\text{Mem}}(\sigma))[[\![T_1]\!]_{\text{Term}}(\sigma) \mapsto [\![T_2]\!]_{\text{Term}}(\sigma)]
\end{aligned}$$

$$\begin{aligned}
&[\![M\{\langle T_1, T_2 \rangle \mapsto T_3\}]\!]_{\text{Mem}}(\langle \rho, \mu, \phi, \kappa \rangle \text{ as } \sigma) = \\
&\quad \mu[a \mapsto v \bmod 2^8][a+1 \mapsto (v/2^8) \bmod 2^8] \cdots [a + (s-1) \mapsto (v/(2^8)^{s-1}) \bmod 2^8] \\
&\quad \text{where } a = [\![T_1]\!]_{\text{Term}}(\sigma),\ s = [\![T_2]\!]_{\text{Term}}(\sigma), \text{ and } v = [\![T_3]\!]_{\text{Term}}(\sigma) \\
&[\![M\{\langle T_1, T_2 \rangle \mapsto \mathcal{C}(T_3, T_4, T_2)\}]\!]_{\text{Mem}}(\langle \rho, \mu, \phi, \kappa \rangle \text{ as } \sigma) = \\
&\quad \mu[a \mapsto \kappa(f)(p)][a+1 \mapsto \kappa(f)(p+1)] \cdots [a + (s-1) \mapsto \kappa(f)(p + (s-1))] \\
&\quad \text{where } a = [\![T_1]\!]_{\text{Term}}(\sigma),\ s = [\![T_2]\!]_{\text{Term}}(\sigma),\ f = [\![T_3]\!]_{\text{Term}}(\sigma),\ p = [\![T_4]\!]_{\text{Term}}(\sigma)
\end{aligned}$$

Figure 4-3: The semantics of Mem_{s}.

$$[\![\cdot]\!]_{\text{Pos}} \in \text{FilePos}_{\text{s}} \to \text{State} \to \text{FilePos}$$

$$\begin{aligned}
[\![\phi]\!]_{\text{Pos}}(\sigma) &= \phi \\
[\![\mathcal{P}]\!]_{\text{Pos}}(\langle \rho, \mu, \phi, \kappa \rangle) &= \phi \\
[\![P\{T_1 \mapsto T_2\}]\!]_{\text{Pos}}(\sigma) &= ([\![P]\!]_{\text{Pos}}(\sigma))[[\![T_1]\!]_{\text{Term}}(\sigma) \mapsto [\![T_2]\!]_{\text{Term}}(\sigma)]
\end{aligned}$$

Figure 4-4: The semantics of $\text{FilePos}_{\text{s}}$.

$$[\![\cdot]\!]_{\text{Cont}} \in \text{FileCont}_{\text{s}} \to \text{State} \to \text{FileCont}$$

$$\begin{aligned}
[\![\kappa]\!]_{\text{Pos}}(\sigma) &= \kappa \\
[\![\mathcal{C}]\!]_{\text{Pos}}(\langle \rho, \mu, \phi, \kappa \rangle) &= \kappa
\end{aligned}$$

Figure 4-5: The semantics of $\text{FileCont}_{\text{s}}$.

$$[\![\cdot]\!]_{\text{Form}} \in \text{Formula} \rightarrow \wp(\text{State})$$

$$\begin{aligned}
[\![\text{true}]\!]_{\text{Form}} &= \text{State} \\
[\![\text{false}]\!]_{\text{Form}} &= \emptyset \\
[\![T_1 = T_2]\!]_{\text{Form}} &= \{\sigma \mid [\![T_1]\!]_{\text{Term}}(\sigma) = [\![T_2]\!]_{\text{Term}}(\sigma)\} \\
[\![T_1 < T_2]\!]_{\text{Form}} &= \{\sigma \mid [\![T_1]\!]_{\text{Term}}(\sigma) < [\![T_2]\!]_{\text{Term}}(\sigma)\} \\
[\![\neg F]\!]_{\text{Form}} &= \text{State} \setminus [\![F]\!]_{\text{Form}} \\
[\![F_1 \wedge F_2]\!]_{\text{Form}} &= [\![F_1]\!]_{\text{Form}} \cap [\![F_2]\!]_{\text{Form}} \\
[\![\forall x : F]\!]_{\text{Form}} &= \{\sigma \mid \sigma \in [\![F[v \,/\, x]]\!]_{\text{Form}} \text{ for all } v \in \text{Int}\} \\
[\![\forall \mathcal{M} : F]\!]_{\text{Form}} &= \{\sigma \mid \sigma \in [\![F[\mu \,/\, \mathcal{M}]]\!]_{\text{Form}} \text{ for all } \mu \in \text{Mem}\} \\
[\![\forall \mathcal{P} : F]\!]_{\text{Form}} &= \{\sigma \mid \sigma \in [\![F[\phi \,/\, \mathcal{P}]]\!]_{\text{Form}} \text{ for all } \phi \in \text{FilePos}\} \\
[\![\forall \mathcal{C} : F]\!]_{\text{Form}} &= \{\sigma \mid \sigma \in [\![F[\kappa \,/\, \mathcal{C}]]\!]_{\text{Form}} \text{ for all } \kappa \in \text{FileCont}\}
\end{aligned}$$

Figure 4-6: The semantics of Formula.

Notation. If a program state σ is included in the denotation $[\![F]\!]_{\text{Form}}$ of a logical formula F, the program state σ is a *model* of the formula F, which is designated by $\sigma \models F$:

$$\sigma \models F \equiv \sigma \in [\![F]\!]_{\text{Form}}$$

Chapter 5

Predicate Transformer Semantics

This chapter defines the predicate transformer semantics of program statements in the core language. A predicate transformer is a function mapping a postcondition of a program statement to its corresponding precondition. Most notably, in our predicate transformer semantics, each instance of a `load`/`store`/`read` statement in a program is given a predicate transformer of different levels of granularity, depending on what kinds of arguments it is called with.

5.1 Predicate Transformer Semantics

Tables 5.1 and 5.2 on pages 72–73 present the predicate transformer semantics of the core language. Given a sequence Q of statements, a program point $\ell \in \mathsf{labels}(Q)$, and a logical formula F, the predicate transformer $\mathsf{precond}(Q, \langle \ell, F \rangle)$ returns a set of pairs of the preceding program points of the program point ℓ and their associated conditions that logically imply the formula F at the program point ℓ. More formally, the predicate transformer $\mathsf{precond}$ is a binary relation between 1) a set of tuples of statement sequences, program points, and logical formulas and 2) a power set of a set of pairs of program points and logical formulas:

$$\mathsf{precond} \in (\mathrm{Seq} \times (\mathrm{Label} \times \mathrm{Formula})) \times \mathcal{P}(\mathrm{Label} \times \mathrm{Formula})$$

Table 5.1: Predicate transformer semantics (1/2).

Program Q	Label ℓ	$\mathsf{precond}(Q, \langle \ell, F \rangle)$
`skip`	$\{\mathsf{after}(Q)\}$	$\{\langle \mathsf{before}(Q), F \rangle\}$
x `:=` E	$\{\mathsf{after}(Q)\}$	$\{\langle \mathsf{before}(Q), F[E/x] \rangle\}$
x `:= load(`E`,` n`)`	$\{\mathsf{after}(Q)\}$	$\{\langle \mathsf{before}(Q), F[(\mathcal{M}(E+0) \times (2^8)^0 + \mathcal{M}(E+1) \times (2^8)^1 + \cdots + \mathcal{M}(E+(n-1)) \times (2^8)^{n-1}) \,/\, x] \rangle\}$
x `:= load(`E_1`,` E_2`)`	$\{\mathsf{after}(Q)\}$	$\{\langle \mathsf{before}(Q), F[\mathcal{M}(E_1, E_2) \,/\, x] \rangle\}$
x `:= hton(`E`,` n`)`	$\{\mathsf{after}(Q)\}$	$\{\langle \mathsf{before}(Q), F[(\mathcal{M}(E+0) \times (2^8)^{n-1} + \mathcal{M}(E+1) \times (2^8)^{n-2} + \cdots + \mathcal{M}(E+(n-1)) \times (2^8)^0) \,/\, x] \rangle\}$
`store(`E_1`,`E_2`,`n`)`	$\{\mathsf{after}(Q)\}$	$\{\langle \mathsf{before}(Q),$ $F[\mathcal{M}\{E_2 \mapsto E_1 \bmod 2^8\}\{E_2+1 \mapsto (E_1/2^8) \bmod 2^8\} \cdots \{E_2+(n-1) \mapsto (E_1/(2^8)^{n-1}) \bmod 2^8\} \,/\, \mathcal{M}] \rangle\}$
`store(`E_1`,`E_2`,`E_3`)`	$\{\mathsf{after}(Q)\}$	$\{\langle \mathsf{before}(Q), F[\mathcal{M}\{\langle E_2, E_3 \rangle \mapsto E_1\} \,/\, \mathcal{M}] \rangle\}$
x `:= read(`E_1`,` E_2`,` n`)`	$\{\mathsf{after}(Q)\}$	$\{\langle \mathsf{before}(Q),$ $F[n \,/\, x][\mathcal{P}\{E_1 \mapsto \mathcal{P}(E_1) + n\} \,/\, \mathcal{P}]$ $[\mathcal{M}\{E_2 \mapsto \mathcal{C}(E_1, \mathcal{P}(E_1))\}\{E_2+1 \mapsto \mathcal{C}(E_1, \mathcal{P}(E_1)+1)\} \cdots \{E_2+(n-1) \mapsto \mathcal{C}(E_1, \mathcal{P}(E_1)+(n-1))\} \,/\, \mathcal{M}] \wedge$ $F[-1 \,/\, x][\mathcal{P}\{E_1 \mapsto \mathcal{P}(E_1) + n\} \,/\, \mathcal{P}]$ $[\mathcal{M}\{E_2 \mapsto \mathcal{C}(E_1, \mathcal{P}(E_1))\}\{E_2+1 \mapsto \mathcal{C}(E_1, \mathcal{P}(E_1)+1)\} \cdots \{E_2+(n-1) \mapsto \mathcal{C}(E_1, \mathcal{P}(E_1)+(n-1))\} \,/\, \mathcal{M}] \wedge$ $(\bigwedge_{m=1}^{n-1} F[m \,/\, x][\mathcal{P}\{E_1 \mapsto \mathcal{P}(E_1) + m\} \,/\, \mathcal{P}][\mathcal{M}\{\langle E_2, m \rangle \mapsto \mathcal{C}(E_1, \mathcal{P}(E_1), m)\} \,/\, \mathcal{M}]) \wedge$ $(\bigwedge_{m=1}^{n-1} F[-1 \,/\, x][\mathcal{P}\{E_1 \mapsto \mathcal{P}(E_1) + m\} \,/\, \mathcal{P}][\mathcal{M}\{\langle E_2, m \rangle \mapsto \mathcal{C}(E_1, \mathcal{P}(E_1), m)\} \,/\, \mathcal{M}]) \wedge$ $F[0 \,/\, x] \wedge F[-1 \,/\, x] \rangle\}$
x `:= read(`E_1`,` E_2`,` E_3`)`	$\{\mathsf{after}(Q)\}$	$\{\langle \mathsf{before}(Q),$ $(\forall 1 \le m \le E_3 : F[m \,/\, x][\mathcal{P}\{E_1 \mapsto \mathcal{P}(E_1) + m\} \,/\, \mathcal{P}][\mathcal{M}\{\langle E_2, m \rangle \mapsto \mathcal{C}(E_1, \mathcal{P}(E_1), m)\} \,/\, \mathcal{M}]) \wedge$ $(\forall 1 \le m \le E_3 : F[-1 \,/\, x][\mathcal{P}\{E_1 \mapsto \mathcal{P}(E_1) + m\} \,/\, \mathcal{P}][\mathcal{M}\{\langle E_2, m \rangle \mapsto \mathcal{C}(E_1, \mathcal{P}(E_1), m)\} \,/\, \mathcal{M}]) \wedge$ $F[0 \,/\, x] \wedge F[-1 \,/\, x] \rangle\}$
x `= seek(`E_1`,` E_2`)`	$\{\mathsf{after}(Q)\}$	$\{\langle \mathsf{before}(Q),$ $(\mathcal{P}(E_1) + E_2 \ge 0 \rightarrow F[\mathcal{P}(E_1) + E_2 \,/\, x][\mathcal{P}\{E_1 \mapsto \mathcal{P}(E_1) + E_2\} \,/\, \mathcal{P}]) \wedge F[-1 \,/\, x] \rangle\}$

Table 5.2: Predicate transformer semantics (2/2).

Program Q	Label ℓ	$\mathsf{precond}(Q, \langle \ell, F \rangle)$
`if` B `then` Q_1 `else` Q_2	$\{\mathsf{after}(Q)\}$	$\{\langle \mathsf{after}(Q_1), F \rangle, \langle \mathsf{after}(Q_2), F \rangle\}$
`if` B `then` Q_1 `else` Q_2	$\mathsf{labels}(Q_1)$	$\begin{cases} \{\langle \mathsf{before}(Q), B \rightarrow F \rangle\} & \text{if } \ell = \mathsf{before}(Q_1), \\ \mathsf{precond}(Q_1, \langle \ell, F \rangle) & \text{otherwise} \end{cases}$
`if` B `then` Q_1 `else` Q_2	$\mathsf{labels}(Q_2)$	$\begin{cases} \{\langle \mathsf{before}(Q), \neg B \rightarrow F \rangle\} & \text{if } \ell = \mathsf{before}(Q_2), \\ \mathsf{precond}(Q_2, \langle \ell, F \rangle) & \text{otherwise} \end{cases}$
`while` (I) B Q_1	$\{\mathsf{after}(Q)\}$	$\{\langle \mathsf{before}(Q), I \wedge \forall V : (I \wedge \neg B \rightarrow F) \rangle, \langle \mathsf{after}(Q_1), I \rangle\}$ where V is the set of assigned variables in Q_1 (including $\mathcal{M}$, $\mathcal{P}$, and $\mathcal{C}$)
`while` (I) B Q_1	$\mathsf{labels}(Q_1)$	$\begin{cases} \{\langle \mathsf{before}(Q), I \wedge \forall V : (I \wedge B \rightarrow F) \rangle, \langle \mathsf{after}(Q_1), I \rangle\} \\ \quad \text{where } V \text{ is the set of assigned variables in } Q_1 \text{ (including } \mathcal{M}, \mathcal{P}, \text{ and } \mathcal{C}) & \text{if } \ell = \mathsf{before}(Q_1), \\ \mathsf{precond}(Q_1, \langle \ell, F \rangle) & \text{otherwise} \end{cases}$
$S_1;\ \cdots;\ S_n$	$\{\mathsf{after}(S_i)\}$	$\mathsf{precond}(S_i, \langle \ell, F \rangle)$

I will write $\langle \ell', F' \rangle \in \mathsf{precond}(Q, \langle \ell, F \rangle)$ if there exists a set $LF \in \mathcal{P}(\text{Label} \times \text{Formula})$ such that $\langle\langle Q, \langle \ell, F \rangle\rangle, LF\rangle \in \mathsf{precond}$ and $\langle \ell', F' \rangle \in LF$. Roughly speaking, $\langle \ell', F' \rangle \in \mathsf{precond}(Q, \langle \ell, F \rangle)$ means that, whenever the logical formula F' holds of a program state at the program point ℓ', the logical formula F will hold if the program execution reaches the program point ℓ afterwards.

Skip Statements. If a logical formula F holds before a `skip` statement, the formula F obviously holds after the statement. `skip` statements do not make any change to program states.

$$\mathsf{precond}(\texttt{skip}, \langle \ell, F \rangle) = \{\langle \ell', F \rangle\}$$
$$\text{where } \ell = \mathsf{after}(\texttt{skip}) \text{ and } \ell' = \mathsf{before}(\texttt{skip})$$

Assignment Statements. Any logical formula F that was true for the right-hand side E of an assignment x := E holds for the variable x after the assignment.

$$\mathsf{precond}(x \texttt{ := } E, \langle \ell, F \rangle) = \{\langle \ell', F[E/x] \rangle\}$$
$$\text{where } \ell = \mathsf{after}(x \texttt{ := } E) \text{ and } \ell' = \mathsf{before}(x \texttt{ := } E)$$

Load Statements. For `load` statements that read a fixed number of bytes from memory, the predicate transformer $\mathsf{precond}(x \texttt{ := load(}E\texttt{, }n\texttt{)}, \langle \ell, F \rangle)$ substitutes all occurrences of the variable x in the logical formula F with a term $\mathcal{M}(E+0) \times (2^8)^0 + \mathcal{M}(E+1) \times (2^8)^1 + \cdots + \mathcal{M}(E+(n-1)) \times (2^8)^{n-1}$ that denotes a value obtained by interpreting a memory region of n bytes pointed to by E in the little-endian format. As explained in Section 4.2, the $\mathcal{M}(E+0), \mathcal{M}(E+1), \ldots, \mathcal{M}(E+(n-1))$ terms mean the single byte values at addresses $E+0, E+1, \ldots, E+(n-1)$ of memory $\mathcal{M}$, respectively. Note that the expanded term $\mathcal{M}(E+0) \times (2^8)^0 + \mathcal{M}(E+1) \times (2^8)^1 +$

$\cdots + \mathcal{M}(E + (n-1)) \times (2^8)^{n-1}$ is of fixed length because n is a constant.

$$\begin{aligned}&\mathsf{precond}(x\ \texttt{:=}\ \texttt{load(}E\texttt{,}\ n\texttt{)}, \langle \ell, F\rangle) = \\ &\quad \{\langle \ell', F[(\mathcal{M}(E+0)\times(2^8)^0 + \mathcal{M}(E+1)\times(2^8)^1 + \cdots + \mathcal{M}(E+(n-1))\times(2^8)^{n-1})\ /\ x]\rangle\} \\ &\qquad \text{where } \ell = \mathsf{after}(x\ \texttt{:=}\ \texttt{load(}E\texttt{,}\ n\texttt{)}) \text{ and } \ell' = \mathsf{before}(x\ \texttt{:=}\ \texttt{load(}E\texttt{,}\ n\texttt{)})\end{aligned}$$

For `load` statements that read a variable number of bytes, the predicate transformer $\mathsf{precond}(x\ \texttt{:=}\ \texttt{load(}E_1\texttt{,}\ E_2\texttt{)}, \langle \ell, F\rangle)$ substitutes all occurrences of the variable x in the logical formula F with an abstract term $\mathcal{M}(E_1, E_2)$ that denotes a value obtained by interpreting a memory region of E_2 bytes pointed to by E_1 in the little-endian format. The abstract $\mathcal{M}(E_1, E_2)$ term cannot be expanded into a more fine-grained form that involves the single byte-valued $\mathcal{M}(E_1+0), \mathcal{M}(E_1+1), \ldots$ terms, unlike `load` statements that read a fixed number of bytes. Term E_2 may evaluate to different values during runtime, so the length of the expanded form cannot be statically determined in general.

$$\begin{aligned}&\mathsf{precond}(x\ \texttt{:=}\ \texttt{load(}E_1\texttt{,}\ E_2\texttt{)}, \langle \ell, F\rangle) = \{\langle \ell', F[\mathcal{M}(E_1, E_2)\ /\ x]\rangle\} \\ &\quad \text{where } \ell = \mathsf{after}(x\ \texttt{:=}\ \texttt{load(}E_1\texttt{,}\ E_2\texttt{)}) \text{ and } \ell' = \mathsf{before}(x\ \texttt{:=}\ \texttt{load(}E_1\texttt{,}\ E_2\texttt{)})\end{aligned}$$

HTON Statements. The predicate transformer $\mathsf{precond}(x\ \texttt{:=}\ \texttt{hton(}E\texttt{,}\ n\texttt{)}, \langle \ell, F\rangle)$ substitutes all occurrences of the variable x in the logical formula F with a term $\mathcal{M}(E+0) \times (2^8)^{n-1} + \mathcal{M}(E+1) \times (2^8)^{n-2} + \cdots + \mathcal{M}(E+(n-1)) \times (2^8)^0$ that denotes a value obtained by interpreting a memory region of n bytes pointed to by E in the big-endian format. The definition of $\mathsf{precond}(x\ \texttt{:=}\ \texttt{hton(}E\texttt{,}\ n\texttt{)}, \langle \ell, F\rangle)$ is similar to that of $\mathsf{precond}(x\ \texttt{:=}\ \texttt{load(}E\texttt{,}\ n\texttt{)}, \langle \ell, F\rangle)$ because both statements interpret a memory region of n bytes pointed to by E and set the variable x to the resulting value. However, the `hton` statement interprets the memory region in accordance with big-endianness

while the `load` statement interprets it in accordance with little-endianness.

$$\mathsf{precond}(x \;\texttt{:=}\; \texttt{hton(}E\texttt{,}\; n\texttt{)}, \langle \ell, F\rangle) =$$
$$\{\langle \ell', F[(\mathcal{M}(E+0) \times (2^8)^{n-1} + \mathcal{M}(E+1) \times (2^8)^{n-2} + \cdots + \mathcal{M}(E+(n-1)) \times (2^8)^0) \,/\, x]\rangle\}$$
$$\text{where } \ell = \mathsf{after}(x \;\texttt{:=}\; \texttt{hton(}E\texttt{,}\; n\texttt{)}) \text{ and } \ell' = \mathsf{before}(x \;\texttt{:=}\; \texttt{hton(}E\texttt{,}\; n\texttt{)})$$

Store Statements. For `store` statements that store a fixed number of bytes into memory, the predicate transformer $\mathsf{precond}(\texttt{store(}E_1\texttt{,}\; E_2\texttt{,}\; n\texttt{)}, \langle \ell, F\rangle)$ substitutes all occurrences of memory $\mathcal{M}$ in the logical formula F with a new memory where its memory region starting at address E_2 is updated with the lowest n bytes of the nonnegative integer value E_1. Roughly speaking, if a logical formula F is true for a memory that has a nonnegative integer value E_1 at a memory location E_2, the formula still holds after a $\texttt{store(}E_1\texttt{,}\; E_2\texttt{,}\; n\texttt{)}$ statement.

$$\mathsf{precond}(\texttt{store(}E_1\texttt{,}\; E_2\texttt{,}\; n\texttt{)}, \langle \ell, F\rangle) =$$
$$\{\langle \ell', F[\mathcal{M}\{E_2 + i \mapsto (E_1/(2^8)^i) \bmod 2^8\}_{i=0,\ldots,n-1} \,/\, \mathcal{M}]\rangle\}$$
$$\text{where } \ell = \mathsf{after}(\texttt{store(}E_1\texttt{,}\; E_2\texttt{,}\; n\texttt{)}) \text{ and } \ell' = \mathsf{before}(\texttt{store(}E_1\texttt{,}\; E_2\texttt{,}\; n\texttt{)})$$

The predicate transformer $\mathsf{precond}(\texttt{store(}E_1\texttt{,}\; E_2\texttt{,}\; E_3\texttt{)}, \langle \ell, F\rangle)$, where the `store` statement stores a variable number of bytes into memory, substitutes all occurrences of memory $\mathcal{M}$ in the logical formula F with an abstract term $\mathcal{M}\{\langle E_2, E_3\rangle \mapsto E_1\}$ that means a new memory where its memory region starting at address E_2 is updated with the lowest E_3 bytes of the nonnegative integer value E_1. Roughly speaking, if the logical formula F holds for a memory that stores the lowest E_3 bytes of the nonnegative integer value E_1 at the memory location E_2, the F formula still holds after the $\texttt{store(}E_1\texttt{,}\; E_2\texttt{,}\; E_3\texttt{)}$ statement. $\mathsf{precond}(\texttt{store(}E_1\texttt{,}\; E_2\texttt{,}\; E_3\texttt{)}, \langle \ell, F\rangle)$ is to $\mathsf{precond}(\texttt{store(}E_1\texttt{,}\; E_2\texttt{,}\; n\texttt{)}, \langle \ell, F\rangle)$ what $\mathsf{precond}(x \;\texttt{:=}\; \texttt{load(}E_1\texttt{,}\; E_2\texttt{)}, \langle \ell, F\rangle)$ is

to $\mathsf{precond}(x \texttt{ := load(}E\texttt{, } n\texttt{)}, \langle \ell, F\rangle)$.

$$\mathsf{precond}(\texttt{store(}E_1\texttt{, } E_2\texttt{, } E_3\texttt{)}, \langle \ell, F\rangle) = \{\langle \ell', F[\mathcal{M}\{\langle E_2, E_3\rangle \mapsto E_1\} \,/\, \mathcal{M}]\rangle\}$$
$$\text{where } \ell = \mathsf{after}(\texttt{store(}E_1\texttt{, } E_2\texttt{, } E_3\texttt{)}) \text{ and } \ell' = \mathsf{before}(\texttt{store(}E_1\texttt{, } E_2\texttt{, } E_3\texttt{)})$$

For file-related statements, as we will see below, the **precond** transformer produces more complex formulas than other statements. There is always a chance that file-related statements are partially performed or fail completely by external factors, such physical I/O errors and premature end-of-file. The complexity of the resulting formulas for file-related statements reflects the nondeterminism inherent in those statements.

Read Statements. For `read` statements that attempt to read a fixed number of bytes from a file, the predicate transformer $\mathsf{precond}(x \texttt{ := read(}E_1\texttt{, } E_2\texttt{, } n\texttt{)}, \langle \ell, F\rangle)$ yields a logical conjunction of six clauses, each of which corresponds to one of the possible results of the `read` statement:

1. The `read` function has successfully finished by 1) reading the specified number n of bytes from the file E_1 into the memory pointed to by E_2, 2) moving the file position of the file E_1 appropriately, and 3) returning the number n of bytes actually read. The return value has been assigned to the variable x. The corresponding clause is obtained by substituting the variable x with the return value n, as with assignment statements, and increasing the current file position of the file E_1 by the n bytes actually read. All occurrences of memory $\mathcal{M}$ are also substituted with a new memory where each location of a memory region starting at address E_2 is updated with the corresponding byte value at the current file position of the file E_1.

$$F[n \,/\, x][\mathcal{P}\{E_1 \mapsto \mathcal{P}(E_1) + n\} \,/\, \mathcal{P}][\mathcal{M}\{E_2 + i \mapsto \mathcal{C}(E_1, \mathcal{P}(E_1 + i))\}_{i=0,\ldots,n-1} \,/\, \mathcal{M}]$$

2. The `read` function has read the specified number n of bytes and moved the file position appropriately, but then failed due to an external I/O error. The

occurrence of error is notified to the caller of the `read` function by the return value -1. So the variable x in the logical formula F is substituted by -1 instead of the number of bytes read. Memory $\mathcal{M}$ and file position $\mathcal{P}$ are substituted in the same way as in the preceding case.

$$F[-1\,/\,x][\mathcal{P}\{E_1 \mapsto \mathcal{P}(E_1)+n\}\,/\,\mathcal{P}][\mathcal{M}\{E_2+i \mapsto \mathcal{C}(E_1, \mathcal{P}(E_1)+i)\}_{i=0,\ldots,n-1}\,/\,\mathcal{M}]$$

3. The `read` function did not suffer from any external I/O errors, but has read less than the specified number of bytes because the file turned out to be shorter than expected. In other words, the premature end-of-file error has occurred. For each possible number m of bytes read, the variable x and file position $\mathcal{P}$ in the logical formula F are handled as in the previous cases by substituting all their occurrences with the appropriately updated terms. However, in contrast to the previous cases, the predicate transformer $\mathsf{precond}(x \texttt{ := read(}E_1\texttt{, }E_2\texttt{, }n\texttt{)}, \langle \ell, F \rangle)$ substitutes all occurrences of memory $\mathcal{M}$ with an abstract term $\mathcal{M}\{\langle E_2, m\rangle \mapsto \mathcal{C}(E_1, \mathcal{P}(E_1), m)\}$, which, as explained in Section 4.2, denotes that the m-bytes data from the current file position $\mathcal{P}(E_1)$ of file E_1 have been read into a memory region pointed to by E_2. Note that I could use the expanded form $\mathcal{M}\{E_2 + i \mapsto \mathcal{C}(E_1, \mathcal{P}(E_i)+i)\}_{i=0,\ldots,m-1}$ in place of the abstract form. However, I have observed that, when this case happens, a developer hardly cares about the byte-level contents just read from the file and that the fine granularity of the expanded form is not advantageous. So I decided to employ the more abstract (and therefore more economical) form. If it turns out later that there exist some cases where a developer wants to check if some specific bytes has been observed immediately before the premature end-of-file error, the abstract term can just be replaced with its expanded form.

$$\bigwedge_{m=1}^{n-1} F[m\,/\,x][\mathcal{P}\{E_1 \mapsto \mathcal{P}(E_1) + m\}\,/\,\mathcal{P}][\mathcal{M}\{\langle E_2, m\rangle \mapsto \mathcal{C}(E_1, \mathcal{P}(E_1), m)\}\,/\,\mathcal{M}]$$

4. The `read` function has failed due to an external I/O error after it read some

(but not all) of the specified number of bytes. For this case, the predicate transformer $\textsf{precond}(x \texttt{ := read(}E_1\texttt{, }E_2\texttt{, }n\texttt{)}, \langle \ell, F\rangle$ yields a similar logical clause to that of Case 3 except that the return value, which the variable x is substituted by, is -1.

$$\bigwedge_{m=1}^{n-1} F[-1 \,/\, x][\mathcal{P}\{E_1 \mapsto \mathcal{P}(E_1) + m\} \,/\, \mathcal{P}][\mathcal{M}\{\langle E_2, m\rangle \mapsto \mathcal{C}(E_1, \mathcal{P}(E_1), m)\} \,/\, \mathcal{M}]$$

5. The file has already reached the end-of-file. In this case, the `read` statement immediately returns zero without attempting to read any bytes and is equivalent to just assigning zero to the variable x.

$$F[0 \,/\, x]$$

6. Due to a preexisting external I/O error, the `read` statement has failed without consuming any bytes and just returned -1.

$$F[-1 \,/\, x]$$

Altogether,

$$\textsf{precond}(x \texttt{ := read(}E_1\texttt{, }E_2\texttt{, }n\texttt{)}, \langle \ell, F\rangle) =$$

$$\left\{ \left\langle \ell', \begin{array}{l} F[n \,/\, x][\mathcal{P}\{E_1 \mapsto \mathcal{P}(E_1) + n\} \,/\, \mathcal{P}][\mathcal{M}\{E_2 + i \mapsto \mathcal{C}(E_1, \mathcal{P}(E_1) + i)\}_{i=0,\ldots,n-1} \,/\, \mathcal{M}] \wedge \\ F[-1 \,/\, x][\mathcal{P}\{E_1 \mapsto \mathcal{P}(E_1) + n\} \,/\, \mathcal{P}][\mathcal{M}\{E_2 + i \mapsto \mathcal{C}(E_1, \mathcal{P}(E_1) + i)\}_{i=0,\ldots,n-1} \,/\, \mathcal{M}] \wedge \\ (\bigwedge_{m=1}^{n-1} F[m \,/\, x][\mathcal{P}\{E_1 \mapsto \mathcal{P}(E_1) + m\} \,/\, \mathcal{P}][\mathcal{M}\{\langle E_2, m\rangle \mapsto \mathcal{C}(E_1, \mathcal{P}(E_1), m)\} \,/\, \mathcal{M}]) \wedge \\ (\bigwedge_{m=1}^{n-1} F[-1 \,/\, x][\mathcal{P}\{E_1 \mapsto \mathcal{P}(E_1) + m\} \,/\, \mathcal{P}][\mathcal{M}\{\langle E_2, m\rangle \mapsto \mathcal{C}(E_1, \mathcal{P}(E_1), m)\} \,/\, \mathcal{M}]) \wedge \\ F[0 \,/\, x] \wedge F[-1 \,/\, x] \end{array} \right\rangle \right\}$$

where $\ell = \textsf{after}(\texttt{read(}E_1\texttt{, }E_2\texttt{, }n\texttt{)})$ and $\ell' = \textsf{before}(\texttt{read(}E_1\texttt{, }E_2\texttt{, }n\texttt{)})$

In contrast, the predicate transformer $\textsf{precond}(x \texttt{ := read(}E_1\texttt{, }E_2\texttt{, }E_3\texttt{)}, \langle \ell, F\rangle)$, where E_3 is not a constant, produces a logical conjunction of four clauses instead of six. I have observed that `read` statements whose number of bytes to read is not specified as

a constant are mostly meant to read the pure data fields of binary file formats. Such fields do not contain information that determines the syntactic structure of binary files, such as the length of fields and field delimiters, and the byte-level contents of those fields are redundant for verifying that a program meets its input specification. Therefore, the predicate transformer $\mathsf{precond}(x \texttt{ := read(}E_1\texttt{, }E_2\texttt{, }E_3\texttt{)}, \langle \ell, F \rangle)$ is defined so that it yields fewer clauses than the predicate transformer $\mathsf{precond}(x \texttt{ := read(}E_1\texttt{, }E_2\texttt{, }n\texttt{)}, \langle \ell, F \rangle)$:

1. The `read` function has successfully finished by reading some bytes from the file E_1 into the memory pointed to by E_2 and moving the file position of the file E_1 accordingly. Also, the number m of bytes actually read, which the `read` function returns, has been assigned to the variable x. Note that the `read` statement may read less than the specified number E_3 of bytes because there may be not enough bytes left in the file. This can be understood as the combination of cases 1 and 3 for `read` statements that read a fixed number of bytes. The corresponding clause is obtained by substituting the variable x with the number m of bytes actually read. The current file position of the file E_1 is also increased by m bytes. All occurrences of memory $\mathcal{M}$ are substituted with a new memory that remembers its m-bytes memory region starting at address E_2 has been filled with the m-bytes contents from the current file position $\mathcal{P}(E_1)$ of the file E_1.

$$\forall 1 \leq m \leq E_3 : F[m \,/\, x][\mathcal{P}\{E_1 \mapsto \mathcal{P}(E_1){+}m\} \,/\, \mathcal{P}][\mathcal{M}\{\langle E_2, m\rangle \mapsto \mathcal{C}(E_1, \mathcal{P}(E_1), m)\} \,/\, \mathcal{M}]$$

2. The `read` function has successfully read some bytes and moved the file position accordingly, but then come across some external I/O error. So memory $\mathcal{M}$ and file position $\mathcal{P}$ are substituted in the same way as in the preceding case, while the variable x in the logical formula F is substituted by the error return value -1. This can be understood as the combination of cases 2 and 4 for `read` statements that read a fixed number of bytes.

$$\forall 1 \leq m \leq E_3 : F[-1 \,/\, x][\mathcal{P}\{E_1 \mapsto \mathcal{P}(E_1){+}m\} \,/\, \mathcal{P}][\mathcal{M}\{\langle E_2, m\rangle \mapsto \mathcal{C}(E_1, \mathcal{P}(E_1), m)\} \,/\, \mathcal{M}]$$

3. The `read` function has just returned zero without attempting to read any bytes

because it already reached the end-of-file.

$$F[0\,/\,x]$$

4. The `read` function has not consumed any bytes and just returned the error code -1 because of a preexisting external I/O error.

$$F[-1\,/\,x]$$

Altogether,

$$\mathsf{precond}(x\ \texttt{:=}\ \texttt{read(}E_1\texttt{,}\ E_2\texttt{,}\ E_3\texttt{)}, \langle \ell, F\rangle) =$$

$$\left\{ \left\langle \ell', \begin{array}{l} (\forall 1 \le m \le E_3 : F[m\,/\,x][\mathcal{P}\{E_1 \mapsto \mathcal{P}(E_1) + m\}\,/\,\mathcal{P}][\mathcal{M}\{\langle E_2, m\rangle \mapsto \mathcal{C}(E_1, \mathcal{P}(E_1), m)\}\,/\,\mathcal{M}]) \wedge \\ (\forall 1 \le m \le E_3 : F[-1\,/\,x][\mathcal{P}\{E_1 \mapsto \mathcal{P}(E_1) + m\}\,/\,\mathcal{P}][\mathcal{M}\{\langle E_2, m\rangle \mapsto \mathcal{C}(E_1, \mathcal{P}(E_1), m)\}\,/\,\mathcal{M}]) \wedge \\ F[0\,/\,x] \wedge F[-1\,/\,x] \end{array} \right\rangle \right\}$$

where $\ell = \mathsf{after}(\texttt{read(}E_1\texttt{,}\ E_2\texttt{,}\ E_3\texttt{)})$ and $\ell' = \mathsf{before}(\texttt{read(}E_1\texttt{,}\ E_2\texttt{,}\ E_3\texttt{)})$

Seek Statements. For `seek` statements that move the file position of the file E_1 by the offset E_2, the predicate transformer $\mathsf{precond}(x\ \texttt{:=}\ \texttt{seek(}E_1\texttt{,}\ E_2\texttt{)}, \langle \ell, F\rangle)$ produces a logical conjunction of two clauses, each of which corresponds to one of two possible results of the `seek` statement:

1. The `seek` statement has successfully advanced the file position of the file E_1 by the specified offset E_2. We obtain the corresponding clause by substituting the variable x with the resulting file position $\mathcal{P}(E_1) + E_2$. The current file position of the file E_1 is also increased by the given offset E_2. Note that the resulting file position must be nonnegative, as explained in Section 3.2.3. So the whole clause is conditioned by the resulting file position being nonnegative.

$$\mathcal{P}(E_1) + E_2 \ge 0 \rightarrow F[\mathcal{P}(E_1) + E_2\,/\,x][\mathcal{P}\{E_1 \mapsto \mathcal{P}(E_1) + E_2\}\,/\,\mathcal{P}]$$

2. The `seek` statement has failed and just set the variable x to -1 because it resulted in a negative file position or some external error occurred.

$$F[-1\,/\,x]$$

Altogether,

$$\mathsf{precond}(x \texttt{ := seek(}E_1\texttt{, }E_2\texttt{)}, \langle \ell, F\rangle) =$$
$$\{\langle \ell', (\mathcal{P}(E_1) + E_2 \geq 0 \rightarrow F[\mathcal{P}(E_1) + E_2\,/\,x][\mathcal{P}\{E_1 \mapsto \mathcal{P}(E_1) + E_2\}\,/\,\mathcal{P}]) \wedge F[-1\,/\,x]\rangle\}$$
$$\text{where } \ell = \mathsf{after}(x \texttt{ := seek(}E_1\texttt{, }E_2\texttt{)}) \text{ and } \ell' = \mathsf{before}(x \texttt{ := seek(}E_1\texttt{, }E_2\texttt{)})$$

If Statements. The predicate transformer $\mathsf{precond}(\texttt{if } B \texttt{ then } Q_1 \texttt{ else } Q_2, \langle \ell, F\rangle)$ transforms the logical formula F differently, depending on the current program point ℓ, because an `if` statement contains a multiple number of program points in it. When program execution reaches the after label of the whole `if` statement, the immediately previous program execution point may be either of the after labels of the then and else branches. Therefore, the predicate transformer $\mathsf{precond}(\texttt{if } B \texttt{ then } Q_1 \texttt{ else } Q_2, \langle \ell, F\rangle)$ propagates the logical formula F into both branches Q_1 and Q_2 when the program point ℓ is the after label of the `if` statement.

$$\mathsf{precond}(\texttt{if } B \texttt{ then } Q_1 \texttt{ else } Q_2, \langle \ell, F\rangle) = \{\langle \mathsf{after}(Q_1), F\rangle, \langle \mathsf{after}(Q_2), F\rangle\}$$
$$\text{where } \ell = \mathsf{after}(\texttt{if } B \texttt{ then } Q_1 \texttt{ else } Q_2)$$

When the program point ℓ is within the then branch Q_1 of the `if` statement, the predicate transformer $\mathsf{precond}(\texttt{if } B \texttt{ then } Q_1 \texttt{ else } Q_2, \langle \ell, F\rangle)$ depends on whether the program point ℓ is the before label of the then branch Q_1. If so, the predicate transformer guards the logical formula F with the condition part B of the `if` statement.

Otherwise, the predicate transformer is defined recursively as $\mathsf{precond}(Q_1, \langle \ell, F \rangle)$.

$$\mathsf{precond}(\texttt{if } B \texttt{ then } Q_1 \texttt{ else } Q_2, \langle \ell, F \rangle) = \begin{cases} \{\langle \mathsf{before}(Q), B \to F \rangle\} & \text{if } \ell = \mathsf{before}(Q_1), \\ \mathsf{precond}(Q_1, \langle \ell, F \rangle) & \text{otherwise} \end{cases}$$

where $\ell \in \mathsf{labels}(Q_1)$

For cases where the program point ℓ is within the else branch Q_2 of the `if` statement, the predicate transformer $\mathsf{precond}(\texttt{if } B \texttt{ then } Q_1 \texttt{ else } Q_2, \langle \ell, F \rangle$ is similarly defined, except that the logical formula F is guarded with the negation of the condition part B of the `if` statement.

$$\mathsf{precond}(\texttt{if } B \texttt{ then } Q_1 \texttt{ else } Q_2, \langle \ell, F \rangle) = \begin{cases} \{\langle \mathsf{before}(Q), \neg B \to F \rangle\} & \text{if } \ell = \mathsf{before}(Q_2), \\ \mathsf{precond}(Q_2, \langle \ell, F \rangle) & \text{otherwise} \end{cases}$$

where $\ell \in \mathsf{labels}(Q_2)$

While Statements. The predicate transformer $\mathsf{precond}(\texttt{while (}I\texttt{) } B\ Q_1, \langle \ell, F \rangle)$ transforms the logical formula F differently, depending on whether the current program point ℓ is within the loop body Q_1. Program execution reaches the after label of the whole `while` statement when its condition part B evaluates to false possibly after repeating the loop body. Therefore, the negation of the condition part B along with the loop invariant I has to be able to establish the logical formula F_2 at the beginning of the `while` statement, regardless of the values of variables that are modified in the loop body Q_1. Also, the loop invariant I must hold at the beginning of the loop and be re-established at the end of the loop body Q_1 after each iteration.

$$\mathsf{precond}(\texttt{while (}I\texttt{) } B\ Q_1, \langle \ell, F \rangle) = \{\langle \mathsf{before}(Q), I \wedge \forall V : (I \wedge \neg B \to F) \rangle, \langle \mathsf{after}(Q_1), I \rangle\}$$

where $\ell = \mathsf{after}(\texttt{while (}I\texttt{) } B\ Q_1)$ and

V is the set of assigned variables in Q_1 (including $\mathcal{M}$, $\mathcal{P}$, and $\mathcal{C}$)

When the program point ℓ is within the loop body Q_1 of the `while` statement, the predicate transformer $\mathsf{precond}(\texttt{while } (I)\ B\ Q_1, \langle \ell, F \rangle)$ depends on whether the program point ℓ is the before label of the loop body Q_1. If so, the conjunction of the loop invariant I and the condition part B of the `while` statement must be able to establish the logical formula F at the beginning of the loop. As with the previous case, the loop invariant I has to be established before entering the loop and after executing the loop body. If the program point ℓ is not the before label, the predicate transformer is defined recursively as $\mathsf{precond}(Q_1, \langle \ell, F \rangle)$.

$$\mathsf{precond}(\texttt{while } (I)\ B\ Q_1, \langle \ell, F \rangle) =$$

$$\begin{cases} \{\langle \mathsf{before}(Q), I \wedge \forall V : (I \wedge B \rightarrow F)\rangle, \langle \mathsf{after}(Q_1), I\rangle\} & \text{if } \ell = \mathsf{before}(Q_1), \\ \mathsf{precond}(Q_1, \langle \ell, F \rangle) & \text{otherwise} \end{cases}$$

where $\ell \in \mathsf{labels}(Q_1)$ and

V is the set of assigned variables in Q_1 (including $\mathcal{M}$, $\mathcal{P}$, and $\mathcal{C}$)

Statement Sequences. For a statement sequence $S_1\,;\ \cdots\,;\ S_n$, the predicate transformer $\mathsf{precond}(S_1\,;\ \cdots\,;\ S_n, \langle \ell, F \rangle)$ is defined recursively as the predicate transformer $\mathsf{precond}(S_i, \langle \ell, F \rangle)$, where S_i is a statement that immediately precedes the program point ℓ.

$$\mathsf{precond}(S_1\,;\ \cdots\,;\ S_n, \langle \ell, F \rangle) = \mathsf{precond}(S_i, \langle \ell, F \rangle) \qquad \text{where } \ell = \mathsf{after}(S_i)$$

5.2 Properties

Conceptually, the predicate transformer $\mathsf{precond}$ is *sound* because, when it maps a postcondition F_2 to a precondition F_1 for a statement Q, a program state σ_2 yielded by a one-step execution of the statement Q under a program state σ_1 that satisfies the precondition F_1 is guaranteed to satisfy the postcondition F_2. This concept is formalized as follows:

Theorem 5.2.1 (Soundness). *If* $\langle \ell_1, \sigma_1 \rangle =\!\llbracket\ Q\ \rrbracket\!\Rightarrow_{\mathsf{Seq}} \langle \ell_2, \sigma_2 \rangle$ *and* $\langle \ell_1, F_1 \rangle \in \mathsf{precond}(Q,$

$\langle \ell_2, F_2 \rangle)$*, then* $\sigma_1 \models F_1$ *implies* $\sigma_2 \models F_2$.

Proof. This proof proceeds by induction on the rules of $\langle \ell_1, \sigma_1 \rangle =\!\!\llbracket\, Q \,\rrbracket\!\!\Rightarrow_{\mathsf{Seq}} \langle \ell_2, \sigma_2 \rangle$.

- **Case** $\langle \ell_1, \sigma_1 \rangle =\!\!\llbracket\, \texttt{skip} \,\rrbracket\!\!\Rightarrow_{\mathsf{Seq}} \langle \ell_2, \sigma_2 \rangle$**.** $\sigma_1 = \sigma_2$ by the dynamics semantics of `skip` statements, and $F_1 = F_2$ because $\mathsf{precond}(\texttt{skip}, \langle \ell_2, F_2 \rangle) = \{\langle \ell_1, F_2 \rangle\}$. Therefore, $\sigma_1 \models F_1$ implies $\sigma_2 \models F_2$.

- **Case** $\langle \ell_1, \sigma_1 \rangle =\!\!\llbracket\, x \texttt{ := } E \,\rrbracket\!\!\Rightarrow_{\mathsf{Seq}} \langle \ell_2, \sigma_2 \rangle$**.** Let $\sigma_1 = \langle \rho, \mu, \phi, \kappa \rangle$ and $\rho \vdash E \Rightarrow v$. $\sigma_2 = \langle \rho[x \mapsto v], \mu, \phi, \kappa \rangle$ by the dynamic semantics of assignment statements, and $F_1 = F_2[E \,/\, x]$ because $\mathsf{precond}(x \texttt{ := } E, \langle \ell_2, F_2 \rangle) = \{\langle \ell_1, F_2[E \,/\, x] \rangle\}$. Therefore, $\sigma_1 \models F_1$ implies $\sigma_2 \models F_2$.

- **Case** $\langle \ell_1, \sigma_1 \rangle =\!\!\llbracket\, x \texttt{ := load(}E\texttt{, } n\texttt{)} \,\rrbracket\!\!\Rightarrow_{\mathsf{Seq}} \langle \ell_2, \sigma_2 \rangle$**.** Let $\sigma_1 = \langle \rho, \mu, \phi, \kappa \rangle$ and $\rho \vdash E \Rightarrow a$. $\sigma_2 = \langle \rho[x \mapsto \sum_{i=0}^{n-1} (\mu(a+i) \times (2^8)^i)], \mu, \phi, \kappa \rangle$ by the dynamic semantics of `load` statements, and $F_1 = F_2[\sum_{i=0}^{n-1} (\mathcal{M}(E+i) \times (2^8)^i) \,/\, x]$ because $\mathsf{precond}(x \texttt{ := load(}E\texttt{, } n\texttt{)}, \langle \ell_2, F_2 \rangle) = \{\langle \ell_1, F_2[\sum_{i=0}^{n-1} (\mathcal{M}(E+i) \times (2^8)^i) \,/\, x] \rangle\}$. Therefore, $\sigma_1 \models F_1$ implies $\sigma_2 \models F_2$.

- **Case** $\langle \ell_1, \sigma_1 \rangle =\!\!\llbracket\, x \texttt{ := load(}E_1\texttt{, } E_2\texttt{)} \,\rrbracket\!\!\Rightarrow_{\mathsf{Seq}} \langle \ell_2, \sigma_2 \rangle$**.** Let $\sigma_1 = \langle \rho, \mu, \phi, \kappa \rangle$, $\rho \vdash E_1 \Rightarrow a$, and $\rho \vdash E_2 \Rightarrow s$. $\sigma_2 = \langle \rho[x \mapsto \sum_{i=0}^{s-1} (\mu(a+i) \times (2^8)^i)], \mu, \phi, \kappa \rangle$ by the dynamic semantics of `load` statements, and $F_1 = F_2[\mathcal{M}(E_1, E_2) \,/\, x]$ because $\mathsf{precond}(x \texttt{ := load(}E\texttt{, } n\texttt{)}, \langle \ell_2, F_2 \rangle) = \{\langle \ell_1, F_2[\mathcal{M}(E_1, E_2) \,/\, x] \rangle\}$. Therefore, $\sigma_1 \models F_1$ implies $\sigma_2 \models F_2$.

- **Case** $\langle \ell_1, \sigma_1 \rangle =\!\!\llbracket\, x \texttt{ := hton(}E\texttt{, } n\texttt{)} \,\rrbracket\!\!\Rightarrow_{\mathsf{Seq}} \langle \ell_2, \sigma_2 \rangle$**.** Let $\sigma_1 = \langle \rho, \mu, \phi, \kappa \rangle$ and $\rho \vdash E \Rightarrow a$. $\sigma_2 = \langle \rho[x \mapsto \sum_{i=0}^{n-1} (\mu(a+i) \times (2^8)^{n-1-i})], \mu, \phi, \kappa \rangle$ by the dynamic semantics of `load` statements, and $F_1 = F_2[\sum_{i=0}^{n-1} (\mathcal{M}(E+i) \times (2^8)^{n-1-i}) \,/\, x]$ because $\mathsf{precond}(x \texttt{ := load(}E\texttt{, } n\texttt{)}, \langle \ell_2, F_2 \rangle) = \{\langle \ell_1, F_2[\sum_{i=0}^{n-1} (\mathcal{M}(E+i) \times (2^8)^{n-1-i}) \,/\, x] \rangle\}$. Therefore, $\sigma_1 \models F_1$ implies $\sigma_2 \models F_2$.

- **Case** $\langle \ell_1, \sigma_1 \rangle =\!\!\llbracket\, \texttt{store(}E_1\texttt{, } E_2\texttt{, } n\texttt{)} \,\rrbracket\!\!\Rightarrow_{\mathsf{Seq}} \langle \ell_2, \sigma_2 \rangle$. Let $\sigma_1 = \langle \rho, \mu, \phi, \kappa \rangle$, $\rho \vdash E_1 \Rightarrow v$, and $\rho \vdash E_2 \Rightarrow a$. $\sigma_2 = \langle \rho, \mu[a+i \mapsto (v/(2^8)^i) \bmod 2^8]_{i=0,\ldots,n-1}, \phi, \kappa \rangle$ by the dynamic semantics of `store` statements, and $F_1 = F_2[\mathcal{M}\{E_2 + i \mapsto$

$(E_1/(2^8)^i) \bmod 2^8\}_{i=0,\ldots,n-1} / \mathcal{M}]$ because $\mathsf{precond}(\texttt{store(}E_1\texttt{, }E_2\texttt{, }n\texttt{)}, \langle \ell_2, F_2\rangle) = \{\langle \ell_1, F_2[\mathcal{M}\{E_2 + i \mapsto (E_1/(2^8)^i) \bmod 2^8\}_{i=0,\ldots,n-1} / \mathcal{M}]\rangle\}$. Therefore, $\sigma_1 \models F_1$ implies $\sigma_2 \models F_2$.

- **Case** $\langle \ell_1, \sigma_1\rangle =\!\!\llbracket\ \texttt{store(}E_1\texttt{, }E_2\texttt{, }E_3\texttt{)}\ \rrbracket\!\!\Rightarrow_{\mathsf{Seq}} \langle \ell_2, \sigma_2\rangle$. Let $\sigma_1 = \langle \rho, \mu, \phi, \kappa\rangle$, $\rho \vdash E_1 \Rightarrow v$, $\rho \vdash E_2 \Rightarrow a$, and $\rho \vdash E_3 \Rightarrow s$. $\sigma_2 = \langle \rho, \mu[a + i \mapsto (v/(2^8)^i) \bmod 2^8]_{i=0,\ldots,s-1}, \phi, \kappa\rangle$ by the dynamic semantics of **store** statements, and $F_1 = F_2[\mathcal{M}\{\langle E_2, E_3\rangle \mapsto E_1\} / \mathcal{M}]$ because $\mathsf{precond}(\texttt{store(}E_1\texttt{, }E_2\texttt{, }E_3\texttt{)}, \langle \ell_2, F_2\rangle) = \{\langle \ell_1, F_2[\mathcal{M}\{\langle E_2, E_3\rangle \mapsto E_1\} / \mathcal{M}]\rangle\}$. Therefore, $\sigma_1 \models F_1$ implies $\sigma_2 \models F_2$.

- **Case** $\langle \ell_1, \sigma_1\rangle =\!\!\llbracket\ x\ \texttt{:= read(}E_1\texttt{, }E_2\texttt{, }n\texttt{)}\ \rrbracket\!\!\Rightarrow_{\mathsf{Seq}} \langle \ell_2, \sigma_2\rangle$. Let $\sigma_1 = \langle \rho, \mu, \phi, \kappa\rangle$, $\rho \vdash E_1 \Rightarrow f$, $\rho \vdash E_2 \Rightarrow a$, and $0 \leq s' \leq n$. $\sigma_2 = \langle \rho[x \mapsto s'], \mu[a + i \mapsto \kappa(f)(\phi(f)+i)]_{i=0,\ldots,s'-1}, \phi[f \mapsto \phi(f)+s'], \kappa\rangle$ or $\langle \rho[x \mapsto -1], \mu[a+i \mapsto \kappa(f)(\phi(f)+i)]_{i=0,\ldots,s'-1}, \phi[f \mapsto \phi(f) + s'], \kappa\rangle$ by the dynamic semantics of **read** statements, and

$$F_1 = \begin{pmatrix} F_2[0 / x] \land \\ (\bigwedge_{m=1}^{n-1} F_2[m / x][\mathcal{P}\{E_1 \mapsto \mathcal{P}(E_1) + m\} / \mathcal{P}][\mathcal{M}\{\langle E_2, m\rangle \mapsto \mathcal{C}(E_1, \mathcal{P}(E_1), m)\} / \mathcal{M}]) \land \\ F_2[n / x][\mathcal{P}\{E_1 \mapsto \mathcal{P}(E_1) + n\} / \mathcal{P}][\mathcal{M}\{E_2 + i \mapsto \mathcal{C}(E_1, \mathcal{P}(E_1) + i)\}_{i=0,\ldots,n-1} / \mathcal{M}] \land \\ F_2[-1 / x] \land \\ (\bigwedge_{m=1}^{n-1} F_2[-1 / x][\mathcal{P}\{E_1 \mapsto \mathcal{P}(E_1) + m\} / \mathcal{P}][\mathcal{M}\{\langle E_2, m\rangle \mapsto \mathcal{C}(E_1, \mathcal{P}(E_1), m)\} / \mathcal{M}]) \land \\ F_2[-1 / x][\mathcal{P}\{E_1 \mapsto \mathcal{P}(E_1) + n\} / \mathcal{P}][\mathcal{M}\{E_2 + i \mapsto \mathcal{C}(E_1, \mathcal{P}(E_1) + i)\}_{i=0,\ldots,n-1} / \mathcal{M}] \end{pmatrix}$$

  because

$$\mathsf{precond}(x\ \texttt{:= read(}E_1\texttt{, }E_2\texttt{, }n\texttt{)}, \langle \ell_2, F_2\rangle) =$$
$$\left\{ \left\langle \ell_1, \begin{array}{l} F_2[0 / x] \land \\ (\bigwedge_{m=1}^{n-1} F_2[m / x][\mathcal{P}\{E_1 \mapsto \mathcal{P}(E_1) + m\} / \mathcal{P}][\mathcal{M}\{\langle E_2, m\rangle \mapsto \mathcal{C}(E_1, \mathcal{P}(E_1), m)\} / \mathcal{M}]) \land \\ F_2[n / x][\mathcal{P}\{E_1 \mapsto \mathcal{P}(E_1) + n\} / \mathcal{P}][\mathcal{M}\{E_2 + i \mapsto \mathcal{C}(E_1, \mathcal{P}(E_1) + i)\}_{i=0,\ldots,n-1} / \mathcal{M}] \land \\ F_2[-1 / x] \land \\ (\bigwedge_{m=1}^{n-1} F_2[-1 / x][\mathcal{P}\{E_1 \mapsto \mathcal{P}(E_1) + m\} / \mathcal{P}][\mathcal{M}\{\langle E_2, m\rangle \mapsto \mathcal{C}(E_1, \mathcal{P}(E_1), m)\} / \mathcal{M}]) \land \\ F_2[-1 / x][\mathcal{P}\{E_1 \mapsto \mathcal{P}(E_1) + n\} / \mathcal{P}][\mathcal{M}\{E_2 + i \mapsto \mathcal{C}(E_1, \mathcal{P}(E_1) + i)\}_{i=0,\ldots,n-1} / \mathcal{M}] \end{array} \right\rangle \right\}.$$

  Therefore, $\sigma_1 \models F_1$ implies $\sigma_2 \models F_2$.

- **Case** $\langle \ell_1, \sigma_1\rangle =\!\!\llbracket\ x\ \texttt{:= read(}E_1\texttt{, }E_2\texttt{, }E_3\texttt{)}\ \rrbracket\!\!\Rightarrow_{\mathsf{Seq}} \langle \ell_2, \sigma_2\rangle$. Let $\sigma_1 = \langle \rho, \mu, \phi, \kappa\rangle$,

$\rho \vdash E_1 \Rightarrow f$, $\rho \vdash E_2 \Rightarrow a$, $\rho \vdash E_3 \Rightarrow s$, and $0 \leq s' \leq s$. $\sigma_2 = \langle \rho[x \mapsto s'], \mu[a{+}i \mapsto \kappa(f)(\phi(f){+}i)]_{i=0,\ldots,s'-1}, \phi[f \mapsto \phi(f){+}s'], \kappa\rangle$ or $\langle \rho[x \mapsto -1], \mu[a{+}i \mapsto \kappa(f)(\phi(f) + i)]_{i=0,\ldots,s'-1}, \phi[f \mapsto \phi(f) + s'], \kappa\rangle$ by the dynamic semantics of `read` statements, and

$$F_1 = \begin{pmatrix} F_2[0\,/\,x] \wedge \\ (\forall 1 \leq m \leq E_3 : F_2[m\,/\,x][\mathcal{P}\{E_1 \mapsto \mathcal{P}(E_1) + m\}\,/\,\mathcal{P}][\mathcal{M}\{\langle E_2, m\rangle \mapsto \mathcal{C}(E_1, \mathcal{P}(E_1), m)\}\,/\,\mathcal{M}]) \wedge \\ F_2[-1\,/\,x] \wedge \\ (\forall 1 \leq m \leq E_3 : F_2[-1\,/\,x][\mathcal{P}\{E_1 \mapsto \mathcal{P}(E_1) + m\}\,/\,\mathcal{P}][\mathcal{M}\{\langle E_2, m\rangle \mapsto \mathcal{C}(E_1, \mathcal{P}(E_1), m)\}\,/\,\mathcal{M}]) \end{pmatrix}$$

because

$$\mathsf{precond}(x \texttt{ := read(}E_1\texttt{, }E_2\texttt{, }E_3\texttt{)}, \langle \ell_2, F_2\rangle) =$$
$$\left\{ \left\langle \ell_1, \begin{array}{l} F_2[0\,/\,x] \wedge \\ (\forall 1 \leq m \leq E_3 : F_2[m\,/\,x][\mathcal{P}\{E_1 \mapsto \mathcal{P}(E_1) + m\}\,/\,\mathcal{P}][\mathcal{M}\{\langle E_2, m\rangle \mapsto \mathcal{C}(E_1, \mathcal{P}(E_1), m)\}\,/\,\mathcal{M}]) \wedge \\ F_2[-1\,/\,x] \wedge \\ (\forall 1 \leq m \leq E_3 : F_2[-1\,/\,x][\mathcal{P}\{E_1 \mapsto \mathcal{P}(E_1) + m\}\,/\,\mathcal{P}][\mathcal{M}\{\langle E_2, m\rangle \mapsto \mathcal{C}(E_1, \mathcal{P}(E_1), m)\}\,/\,\mathcal{M}]) \end{array} \right\rangle \right\}.$$

Therefore, $\sigma_1 \models F_1$ implies $\sigma_2 \models F_2$.

- **Case** $\langle \ell_1, \sigma_1\rangle =\!\![\![\; x \texttt{ := } \textbf{seek(}E_1\textbf{, }E_2\textbf{)} \;]\!]\!\!\Rightarrow_{\mathsf{Seq}} \langle \ell_2, \sigma_2\rangle$**.** Let $\sigma_1 = \langle \rho, \mu, \phi, \kappa\rangle$, $\rho \vdash E_1 \Rightarrow f$, and $\rho \vdash E_2 \Rightarrow o$.

$$\sigma_2 = \begin{cases} \langle \rho[x \mapsto \rho(f) + o], \mu, \phi[f \mapsto \phi(f) + o], \kappa\rangle & \text{where } \phi(f) + o \geq 0, \\ \langle \rho[x \mapsto -1], \mu, \phi, \kappa\rangle \end{cases}$$

by the dynamic semantics of `seek` statements, and $F_1 = (\mathcal{P}(E_1) + E_2 \geq 0 \rightarrow F_2[\mathcal{P}(E_1) + E_2\,/\,x][\mathcal{P}\{E_1 \mapsto \mathcal{P}(E_1) + E_2\}\,/\,\mathcal{P}]) \wedge F_2[-1\,/\,x]$ because $\mathsf{precond}(x \texttt{ := seek(}E_1\texttt{, }E_2\texttt{)}, \langle \ell_2, F_2\rangle) = \{\langle \ell_1, (\mathcal{P}(E_1) + E_2 \geq 0 \rightarrow F_2[\mathcal{P}(E_1) + E_2\,/\,x][\mathcal{P}\{E_1 \mapsto \mathcal{P}(E_1) + E_2\}\,/\,\mathcal{P}]) \wedge F_2[-1\,/\,x]\rangle\}$. Therefore, $\sigma_1 \models F_1$ implies $\sigma_2 \models F_2$.

- **Case** $\langle \ell_1, \sigma_1\rangle =\!\![\![\; \textbf{if } B \textbf{ then } Q_1 \textbf{ else } Q_2 \;]\!]\!\!\Rightarrow_{\mathsf{Seq}} \langle \ell_2, \sigma_2\rangle$ **where** $\ell_1 = \mathsf{before}(\textbf{if } B \textbf{ then } Q_1 \textbf{ else } Q_2)$ **and** $\ell_2 = \mathsf{before}(Q_1)$**.** Let $\sigma_1 = \langle \rho, \mu, \phi, \kappa\rangle$. $\rho \vdash B \Rightarrow \text{true}$ and $\sigma_2 = \langle \rho, \mu, \phi, \kappa\rangle$ by the dynamic semantics of `if` statements.

$F_1 = B \to F_2$ because $\mathsf{precond}(\texttt{if } B \texttt{ then } Q_1 \texttt{ else } Q_2, \langle \ell_2, F_2 \rangle) = \{\langle \ell_1, B \to F_2 \rangle\}$. Therefore, $\sigma_1 \models F_1$ implies $\sigma_2 \models F_2$.

- **Case** $\langle \ell_1, \sigma_1 \rangle =\!\![\ \texttt{if } B \texttt{ then } Q_1 \texttt{ else } Q_2\ ]\!\!\Rightarrow_{\mathsf{Seq}} \langle \ell_2, \sigma_2 \rangle$ **where** $\ell_1 = \mathsf{before}(\texttt{if } B \texttt{ then } Q_1 \texttt{ else } Q_2)$ **and** $\ell_2 = \mathsf{before}(Q_2)$. Let $\sigma_1 = \langle \rho, \mu, \phi, \kappa \rangle$. $\rho \vdash B \Rightarrow$ false and $\sigma_2 = \langle \rho, \mu, \phi, \kappa \rangle$ by the dynamic semantics of `if` statements. $F_1 = \neg B \to F_2$ because $\mathsf{precond}(\texttt{if } B \texttt{ then } Q_1 \texttt{ else } Q_2, \langle \ell_2, F_2 \rangle) = \{\langle \ell_1, \neg B \to F_2 \rangle\}$. Therefore, $\sigma_1 \models F_1$ implies $\sigma_2 \models F_2$.

- **Case** $\langle \ell_1, \sigma_1 \rangle =\!\![\ \texttt{if } B \texttt{ then } Q_1 \texttt{ else } Q_2\ ]\!\!\Rightarrow_{\mathsf{Seq}} \langle \ell_2, \sigma_2 \rangle$ **where** $\ell_1 = \mathsf{after}(Q_i)$ **and** $\ell_2 = \mathsf{after}(\texttt{if } B \texttt{ then } Q_1 \texttt{ else } Q_2)$. Let $\sigma_1 = \langle \rho, \mu, \phi, \kappa \rangle$. $\sigma_2 = \langle \rho, \mu, \phi, \kappa \rangle$ by the dynamic semantics of `if` statements, and $F_1 = F_2$ because $\mathsf{precond}(\texttt{if } B \texttt{ then } Q_1 \texttt{ else } Q_2, \langle \ell_2, F_2 \rangle) = \{\langle \ell_1, F_2 \rangle\}$. Therefore, $\sigma_1 \models F_1$ implies $\sigma_2 \models F_2$.

- **Case** $\langle \ell_1, \sigma_1 \rangle =\!\![\ \texttt{if } B \texttt{ then } Q_1 \texttt{ else } Q_2\ ]\!\!\Rightarrow_{\mathsf{Seq}} \langle \ell_2, \sigma_2 \rangle$ **where** $\ell_1 = \mathsf{labels}(Q_i) \setminus \{\mathsf{after}(Q_i)\}$. $\langle \ell_1, \sigma_1 \rangle =\!\![\ Q_i\ ]\!\!\Rightarrow_{\mathsf{Seq}} \langle \ell_2, \sigma_2 \rangle$ by the dynamic semantics of `if` statements, and $F_1 \in \mathsf{precond}(Q_i, \langle \ell_2, F_2 \rangle)$ because $\mathsf{precond}(\texttt{if } B \texttt{ then } Q_1 \texttt{ else } Q_2, \langle \ell_2, F_2 \rangle) = \mathsf{precond}(Q_i, \langle \ell_2, F_2 \rangle)$. By induction hypothesis, $\sigma_1 \models F_1$ implies $\sigma_2 \models F_2$.

- **Case** $\langle \ell_1, \sigma_1 \rangle =\!\![\ \texttt{while (}I\texttt{) } B\ Q_1\ ]\!\!\Rightarrow_{\mathsf{Seq}} \langle \ell_2, \sigma_2 \rangle$ **where** $\ell_1 = \mathsf{before}(\texttt{while (}I\texttt{) } B\ Q_1)$ **and** $\ell_2 = \mathsf{before}(Q_1)$. Let $\sigma_1 = \langle \rho, \mu, \phi, \kappa \rangle$. $\rho \vdash B \Rightarrow$ true and $\sigma_2 = \langle \rho, \mu, \phi, \kappa \rangle$ by the dynamic semantics of `while` statements. $F_1 = I \wedge \forall V : (I \wedge B \to F_2)$ because $\mathsf{precond}(\texttt{while (}I\texttt{) } B\ Q_1, \langle \ell_2, F_2 \rangle) = \{\langle \ell_1, I \wedge \forall V : (I \wedge B \to F_2) \rangle, \langle \mathsf{after}(Q_1), I \rangle\}$. Therefore, $\sigma_1 \models F_1$ implies $\sigma_2 \models F_2$.

- **Case** $\langle \ell_1, \sigma_1 \rangle =\!\![\ \texttt{while (}I\texttt{) } B\ Q_1\ ]\!\!\Rightarrow_{\mathsf{Seq}} \langle \ell_2, \sigma_2 \rangle$ **where** $\ell_1 = \mathsf{before}(\texttt{while (}I\texttt{) } B\ Q_1)$ **and** $\ell_2 = \mathsf{after}(\texttt{while (}I\texttt{) } B\ Q_1)$. Let $\sigma_1 = \langle \rho, \mu, \phi, \kappa \rangle$. $\rho \vdash B \Rightarrow$ false and $\sigma_2 = \langle \rho, \mu, \phi, \kappa \rangle$ by the dynamic semantics of `while` statements. $F_1 = I \wedge \forall V : (I \wedge \neg B \to F_2)$ because $\mathsf{precond}(\texttt{while (}I\texttt{) } B\ Q_1, \langle \ell_2, F_2 \rangle) =$

$\{\langle \ell_1, I \wedge \forall V : (I \wedge \neg B \rightarrow F_2)\rangle, \langle \mathsf{after}(Q_1), I\rangle\}$. Therefore, $\sigma_1 \models F_1$ implies $\sigma_2 \models F_2$.

- **Case** $\langle \ell_1, \sigma_1\rangle \dashv\!\![$ `while (`I`)` $B\ Q_1\ ]\!\!\Rightarrow_{\mathsf{Seq}} \langle \ell_2, \sigma_2\rangle$ **where** $\ell_1 = \mathsf{labels}(Q_1) \setminus \{\mathsf{after}(Q_1)\}$. $\langle \ell_1, \sigma_1\rangle \dashv\!\![\ Q_1\ ]\!\!\Rightarrow_{\mathsf{Seq}} \langle \ell_2, \sigma_2\rangle$ by the dynamic semantics of `while` statements, and $F_1 \in \mathsf{precond}(Q_1, \langle \ell_2, F_2\rangle)$ because $\mathsf{precond}($`while (`I`)` $B\ Q_1, \langle \ell_2, F_2\rangle) = \mathsf{precond}(Q_1, \langle \ell_2, F_2\rangle)$. By induction hypothesis, $\sigma_1 \models F_1$ implies $\sigma_2 \models F_2$.

- **Case** $\langle \ell_1, \sigma_1\rangle \dashv\!\![$ `while (`I`)` $B\ Q_1\ ]\!\!\Rightarrow_{\mathsf{Seq}} \langle \ell_2, \sigma_2\rangle$ **where** $\ell_1 = \mathsf{after}(Q_1)$ **and** $\ell_2 = \mathsf{before}($`while (`$I$`)` $B\ Q_1)$. Vacuously true because $\nexists F_1 : \langle \ell_1, F_1\rangle \in \mathsf{precond}($`while (`$I$`)` $B\ Q_1, \langle \ell_2, F_2\rangle)$.

- **Case** $\langle \ell_1, \sigma_1\rangle \dashv\!\![\ S_1 ;\ \cdots ;\ S_n\ ]\!\!\Rightarrow_{\mathsf{Seq}} \langle \ell_2, \sigma_2\rangle$ **where** $\ell_1 = \mathsf{before}(S_i)$. $\langle \ell_1, \sigma_1\rangle \dashv\!\![\ S_i\ ]\!\!\Rightarrow_{\mathsf{Seq}} \langle \ell_2, \sigma_2\rangle$ by the dynamic semantics of a sequence of statements, and $F_1 = \mathsf{precond}(S_i, \langle \ell_2, F_2\rangle)$ because $\mathsf{precond}(S_1 ;\ \cdots ;\ S_n, \langle \ell_2, F_2\rangle) = \{\langle \ell_1, \mathsf{precond}(S_i, \langle \ell_2, F_2\rangle)\rangle\}$. By induction hypothesis, $\sigma_1 \models F_1$ implies $\sigma_2 \models F_2$.

□

The above soundness concept of the precond predicate transformer over a single step of execution is extended into multiple steps of execution, which makes it possible to check the validity of the specification at the remotely preceding program points as well as the immediate preceding one. Let a binary relation $\mathsf{precond}^{\#}(Q)$ on a set $\mathcal{P}(\text{Label} \times \text{Formula})$ be defined as follows:

$$\mathsf{precond}^{\#}(Q) = \{\langle LF, \bigcup_{\langle \ell, F\rangle \in LF} \mathsf{precond}(Q, \langle \ell, F\rangle)\rangle \mid LF \in \mathcal{P}(\text{Label} \times \text{Formula})\}$$

Assume that a program state σ_1 at a program point ℓ_1 in a program Q evaluates to a program state σ_2 at a program point ℓ_2 after multiple steps of execution. If the original program state σ_1 satisfies *all* preconditions at the program point ℓ_1 included in the reflexive transitive closure $\mathsf{precond}^{*}(Q)$ of the binary relation $\mathsf{precond}^{\#}(Q)$, then

the resulting program state σ_2 is guaranteed to satisfy the postcondition F_2 at the program point ℓ_2. Formally,

Corollary 5.2.2. *If* $\langle \ell_1, \sigma_1 \rangle =\!\![\![\, Q \,]\!]\!\!\stackrel{*}{\Rightarrow}_{\mathsf{Seq}} \langle \ell_2, \sigma_2 \rangle$ *and* $\forall \langle \{\langle \ell_2, F_2 \rangle\}, LF \rangle \in \mathsf{precond}^*(Q) : \forall \langle \ell_1, F_1 \rangle \in LF : \sigma_1 \models F_1$*, then* $\sigma_2 \models F_2$*.*

5.3 Alias Axioms

This section formally presents how to incorporate the information from an alias analysis into verification conditions. The proposed system uses an alias analysis to relieve a developer of manually supplying loop invariants and to accelerate verification by not waiting for verification conditions to be propagated up to the start of a program.

5.3.1 Alias Analysis

For a given program Q and an initial program state σ_0, an alias analysis is assumed to provide a function $\mathsf{alias}_Q^{\sigma_0}$ that can determine whether two memory blocks are aliased at each program point:

$$\mathsf{alias}_Q^{\sigma_0} \in \text{Label} \rightarrow (\text{AExpr} \times \text{AExpr}) \rightarrow (\text{AExpr} \times \text{AExpr}) \rightarrow \{\mathsf{must}, \mathsf{no}, \mathsf{may}\}$$

The $\mathsf{alias}_Q^{\sigma_0}$ takes a program point and two memory blocks, each of which is represented by its base address expression and size expression, and returns must, no, or may as appropriate. The must response means that the two memory blocks are guaranteed to start at the same memory location and be of size 1 at the given program point.

Definition 5.3.1. *If* $\mathsf{alias}_Q^{\sigma_0}(\ell, \langle E_1, E_2 \rangle, \langle E_3, E_4 \rangle) = \mathsf{must}$*, then* $\langle \mathsf{before}(Q), \sigma_0 \rangle = [\![Q]\!]\!\!\stackrel{*}{\Rightarrow}_{\mathsf{Seq}} \langle \ell, \langle \rho, \mu, \phi, \kappa \rangle \rangle \wedge \rho \vdash E_1 \Rightarrow a_1 \wedge \rho \vdash E_2 \Rightarrow s_2 \wedge \rho \vdash E_3 \Rightarrow a_3 \wedge \rho \vdash E_4 \Rightarrow s_4$ *implies* $a_1 = a_3 \wedge s_2 = s_4 = 1$*.*

The no response means that the two memory blocks are non-overlapping memory ranges at the given program point.

$$\frac{\boxed{\text{MUST-1}}\quad \mathsf{alias}_Q^{\sigma_0}(\ell, \langle T_1, 1\rangle, \langle T_3, 1\rangle) = \mathsf{must}}{\vdash^\ell M\{T_1 \mapsto T_2\}(T_3) = T_2}$$

$$\frac{\boxed{\text{MUST-2}}\quad \mathsf{alias}_Q^{\sigma_0}(\ell, \langle T_1, 1\rangle, \langle T_3, T_4\rangle) = \mathsf{must}}{\vdash^\ell M\{T_1 \mapsto T_2\}(T_3, T_4) = T_2}$$

$$\frac{\boxed{\text{MUST-3}}\quad \mathsf{alias}_Q^{\sigma_0}(\ell, \langle T_1, T_2\rangle, \langle T_4, 1\rangle) = \mathsf{must}}{\vdash^\ell M\{\langle T_1, T_2\rangle \mapsto T_3\}(T_4) = T_3 \bmod 2^8}$$

$$\frac{\boxed{\text{MUST-4}}\quad \mathsf{alias}_Q^{\sigma_0}(\ell, \langle T_1, T_2\rangle, \langle T_4, T_5\rangle) = \mathsf{must}}{\vdash^\ell M\{\langle T_1, T_2\rangle \mapsto T_3\}(T_4, T_5) = T_3 \bmod 2^8}$$

$$\frac{\boxed{\text{MUST-5}}\quad \mathsf{alias}_Q^{\sigma_0}(\ell, \langle T_1, T_2\rangle, \langle T_6, 1\rangle) = \mathsf{must}}{\vdash^\ell M\{\langle T_1, T_2\rangle \mapsto \mathcal{C}(T_3, T_4, T_5)\}(T_6) = \mathcal{C}(T_3, T_4)}$$

$$\frac{\boxed{\text{MUST-6}}\quad \mathsf{alias}_Q^{\sigma_0}(\ell, \langle T_1, T_2\rangle, \langle T_6, T_7\rangle) = \mathsf{must}}{\vdash^\ell M\{\langle T_1, T_2\rangle \mapsto \mathcal{C}(T_3, T_4, T_5)\}(T_6, T_7) = \mathcal{C}(T_3, T_4)}$$

Figure 5-1: The MustAlias axioms.

Definition 5.3.2. *If* $\mathsf{alias}_Q^{\sigma_0}(\ell, \langle E_1, E_2\rangle, \langle E_3, E_4\rangle) = \mathsf{no}$, *then* $\langle \mathsf{before}(Q), \sigma_0\rangle \Rrightarrow\!\![Q]\!\!\Rrightarrow^*_{\mathsf{Seq}} \langle \ell, \langle \rho, \mu, \phi, \kappa\rangle\rangle \wedge \rho \vdash E_1 \Rightarrow a_1 \wedge \rho \vdash E_2 \Rightarrow s_2 \wedge \rho \vdash E_3 \Rightarrow a_3 \wedge \rho \vdash E_4 \Rightarrow s_4$ *implies* $s_2 > 0 \wedge s_4 > 0 \wedge (a_1 + s_2 < a_3 \vee a_3 + s_4 < a_1)$.

The may response is returned when the alias analysis cannot determine the relation of the two memory blocks precisely.

5.3.2 Alias Axioms

The aliasing information is used to simplify verification conditions. Figure 5-1 presents how the must response from an alias analysis can be used to simplify memory-related terms. The $M\{T_1 \mapsto T_2\}(T_3)$ and $M\{T_1 \mapsto T_2\}(T_3, T_4)$ terms can be reduced into T_2 when the alias analysis determines that T_1 and T_3 denote the same memory address and that T_4, which denotes the size of the memory block to read, is 1 (MUST-1 and MUST-2). The $M\{\langle T_1, T_2\rangle \mapsto T_3\}(T_4)$ and $M\{\langle T_1, T_2\rangle \mapsto T_3\}(T_4, T_5)$ terms can also be reduced into the lowest byte of T_3, i.e. $T_3 \bmod 2^8$, when the alias analysis establishes that T_1 and T_4 denote the same memory address and that the memory

block $\langle T_4, T_5 \rangle$ is of size 1 (MUST-3 and MUST-4). Note that, as explained in Section 4.2, the $M\{\langle T_1, T_2 \rangle \mapsto T_3\}$ term expresses that the values of a memory region of T_2 bytes starting at address T_1 have been obtained by storing the lowest bytes of value T_3 in the little-endian format. Similarly, the $M\{\langle T_1, T_2 \rangle \mapsto \mathcal{C}(T_3, T_4, T_5)\}(T_6)$ and $M\{\langle T_1, T_2 \rangle \mapsto \mathcal{C}(T_3, T_4, T_5)\}(T_6, T_7)$ terms are able to be reduced into $\mathcal{C}(T_3, T_4)$ when the alias analysis determines that T_1 and T_6 denote the same address and that the memory block $\langle T_6, T_7 \rangle$ is of size 1 (MUST-5 and MUST-6). As explained in Section 4.2, the $M\{\langle T_1, T_2 \rangle \mapsto \mathcal{C}(T_3, T_4, T_5)\}$ term indicates that the T_2-bytes data from the offset T_4 of the file T_3 have been read into a memory region pointed to by T_1, and the $\mathcal{C}(T_3, T_4)$ term denotes the byte value at the offset T_4 from the start of the file T_3.

Figure 5-2 on page 93 presents how the memory-related terms can be reduced when the alias analysis returns no. The $M\{T_1 \mapsto T_2\}(T_3)$ term can be simplified into $M(T_3)$ when the alias analysis determines that those two addresses T_1 and T_3 are guaranteed to be distinct (NO-1). Obviously, the 1-byte memory write $\{T_1 \mapsto T_2\}$ to the address T_1 does not make any effect on the 1-byte memory read from the address T_3 when T_1 and T_3 are different from each other. It is possible to reduce the $M\{T_1 \mapsto T_2\}(T_3, T_4)$ term to $M(T_3, T_4)$ when that the address T_1 is guaranteed not to be included in the memory block $\langle T_3, T_4 \rangle$ (NO-2). Similarly, the $M\{\langle T_1, T_2 \rangle \mapsto T_3\}(T_4)$ term can be simplified into $M(T_4)$ when the memory block $\langle T_1, T_2 \rangle$ does not contain the address T_4 (NO-3). Generally, the $M\{\langle T_1, T_2 \rangle \mapsto T_3\}(T_4, T_5)$ term can be reduced to $M(T_4, T_5)$ when the alias analysis establishes that the memory blocks $\langle T_1, T_2 \rangle$ and $\langle T_4, T_5 \rangle$ do not overlap at all (NO-4). In similar ways, the $M\{\langle T_1, T_2 \rangle \mapsto \mathcal{C}(T_3, T_4, T_5)\}(T_6)$ and $M\{\langle T_1, T_2 \rangle \mapsto \mathcal{C}(T_3, T_4, T_5)\}(T_6, T_7)$ terms can be simplified into $M(T_6)$ and $M(T_6, T_7)$ respectively when the involved memory blocks are guaranteed not to alias (NO-5 and NO-6).

NO-1

$$\frac{\mathsf{alias}_Q^{\sigma_0}(\ell, \langle T_1, 1\rangle, \langle T_3, 1\rangle) = \mathsf{no}}{\vdash^\ell M\{T_1 \mapsto T_2\}(T_3) = M(T_3)}$$

NO-2

$$\frac{\mathsf{alias}_Q^{\sigma_0}(\ell, \langle T_1, 1\rangle, \langle T_3, T_4\rangle) = \mathsf{no}}{\vdash^\ell M\{T_1 \mapsto T_2\}(T_3, T_4) = M(T_3, T_4)}$$

NO-3

$$\frac{\mathsf{alias}_Q^{\sigma_0}(\ell, \langle T_1, T_2\rangle, \langle T_4, 1\rangle) = \mathsf{no}}{\vdash^\ell M\{\langle T_1, T_2\rangle \mapsto T_3\}(T_4) = M(T_4)}$$

NO-4

$$\frac{\mathsf{alias}_Q^{\sigma_0}(\ell, \langle T_1, T_2\rangle, \langle T_4, T_5\rangle) = \mathsf{no}}{\vdash^\ell M\{\langle T_1, T_2\rangle \mapsto T_3\}(T_4, T_5) = M(T_4, T_5)}$$

NO-5

$$\frac{\mathsf{alias}_Q^{\sigma_0}(\ell, \langle T_1, T_2\rangle, \langle T_6, 1\rangle) = \mathsf{no}}{\vdash^\ell M\{\langle T_1, T_2\rangle \mapsto \mathcal{C}(T_3, T_4, T_5)\}(T_6) = M(T_6)}$$

NO-6

$$\frac{\mathsf{alias}_Q^{\sigma_0}(\ell, \langle T_1, T_2\rangle, \langle T_6, T_7\rangle) = \mathsf{no}}{\vdash^\ell M\{\langle T_1, T_2\rangle \mapsto \mathcal{C}(T_3, T_4, T_5)\}(T_6, T_7) = M(T_6, T_7)}$$

Figure 5-2: The NoAlias axioms.

Chapter 6

JPEG File Interchange Format

This chapter presents another example that shows how the proposed programming system can be used to develop an image converter for JPEG image files in an error-resilient way. Each input unit of the JPEG file format starts with a delimiter, so it is possible to handle some invalid input units by discarding bytes until finding the next delimiter.

6.1 The JFIF Format

The JPEG standard provides a commonly used method for lossy compression for digital photographic images. JPEG is an acronym for the Joint Photographic Experts Group, which created the standard. The standard defines how an image is compressed into a stream of bytes and decompressed back, but does not specify a file format to store that stream in.

The JPEG File Interchange Format (JFIF) specifies a file format in which JPEG-compressed images can be stored. Along with the Exchangeable image file format (Exif), JFIF is the most common file format for storing and transmitting photographic images on the Internet [35]. It is what people generally mean when they refer to "a JPEG file," and usually has a filename extension of `.jpg` or `.jpeg`.

A JFIF file is partitioned by *markers*, which consist of 2 bytes. The first byte of each marker always has the value FF_{16}, and the second byte specifies the type of the marker. Some markers stand alone, but most are immediately followed by a

SOI	APP0	DQT	SOF0	DHT	SOS	Scan Data	···	SOS	Scan Data	EOI

Figure 6-1: The layout of a typical JFIF image file.

big-endian 2-byte integer that specifies the length of the following parameter bytes plus the 2 bytes used to represent the length field itself. A marker and its associated set of parameters comprise a *marker segment*.

Figure 6-1 presents the layout of a typical JFIF image file, which starts with the SOI (Start of Image) marker (i.e. FF_{16} $D8_{16}$). The APP0 (Application Data) marker segment immediately follows the SOI marker. The null-terminated string "JFIF" is included in the APP0 marker segment and identifies the file as the JFIF image file format. The JFIF APP0 marker segment also provides information beyond what is specified in the JPEG standard: the JFIF version number, the X and Y pixel densities, the pixel aspect ratio, and an optional thumbnail image. The DQT (Define Quantization Table) marker segment provides the quantization tables, which are arrays of 64 elements, used in the image file. A JPEG image is compressed in *data units*, which are 8×8 sized blocks of samples of one color component. The discrete cosine transform (DCT), which is one of the processes of JPEG encoding, expresses the values of a data unit in terms of an 8×8 array of coefficients of cosine functions oscillating at different frequencies. Then, for better compression, the quantization process converts some of those coefficients to zeros by dividing the array of coefficients by a quantization table element by element. The SOF0 (Start of Frame) marker segment indicates that the image data is encoded in the baseline DCT mode and specifies the dimensions of the image and the sampling factor and quantization table identifier for each color component. The DHT (Define Huffman Table) marker segment defines the Huffman tables used in the image. Entropy coding, the last step of JPEG encoding, encodes the quantized DCT coefficients using Huffman coding while eliminating runs of zero values. The SOS (Start of Scan) marker segment specifies which Huffman tables are used for each color component. It is followed by the compressed scan data. To detect the end of the compressed data, the next marker has to be searched for because the compressed data itself does not contain length information. The JPEG compression

FF_{16} (1 byte)	Marker Type (1 byte)	Length (2 bytes)	Parameters (Length - 2 bytes)

Figure 6-2: The common structure of JFIF marker segments.

algorithm is carefully designed so that it will rarely yield the byte value FF_{16}, which is the first byte of every marker. When the byte value FF_{16} is required in the compressed data, it is suffixed with the byte 00_{16} and encoded as the 2-byte sequence FF_{16} 00_{16}. Because the value 00_{16} does not occur as the second byte of any marker, it is easy to search for the next marker that follows the compressed scan data. A JFIF image file must end with the EOI (End of Image) marker (i.e. FF_{16} $D9_{16}$).

6.2 A Resilient JPEG Image Converter

Figures 6-3 and 6-4 on pages 98–99 present code snippets from an image converter that changes the JFIF image format into the portable anymap format (PNM).

The image converter processes each marker segment in a big loop because all the JFIF marker segments share the common structure shown in Figure 6-2. Every marker consists of 2 bytes: an FF_{16} byte and a marker type byte which is neither 00_{16} nor FF_{16}. The JPEG standard allows any marker to be optionally preceded by any number of fill bytes that have the value FF_{16}.

Each iteration of the loop starts by attempting to read a single byte into the `marker_ff` variable (lines 7–8 in Figure 6-3 on page 98).

```
7            uint8_t marker_ff;
8            nbytes = read(fd, &marker_ff, sizeof(marker_ff));
```

The single-byte `read` call may not be able to read any byte because of a premature end-of-file or an external I/O error. In that case, the converter sets some flags (not shown here) appropriately and breaks out of the outermost loop (lines 9–10).

```
9            if (nbytes != sizeof(marker_ff))
10               goto OUT_OF_LOOP;  // EOF OR I/O ERROR
```

```
1   for (;;) {
2       uint8_t marker_type;

3       // Read until FF not followed by 00 or FF is found.
4       for (;;) {
5           // Read until FF is found.
6           for (;;) {
7               uint8_t marker_ff;
8               nbytes = read(fd, &marker_ff, sizeof(marker_ff));
9               if (nbytes != sizeof(marker_ff))
10                  goto OUT_OF_LOOP;  // EOF OR I/O ERROR
11              if (marker_ff == 0xFF) break;
12              continue;  // TOLERATE NOT WELL-FORMED
13          }

14          nbytes = read(fd, &marker_type, sizeof(marker_type));
15          if (nbytes != sizeof(marker_type))
16              goto OUT_OF_LOOP;  // EOF OR I/O ERROR
17          while (mark_type == 0xFF) {  // FILL BYTE
18              nbytes = read(fd, &marker_type, sizeof(marker_type));
19              if (nbytes != sizeof(marker_type))
20                  goto OUT_OF_LOOP;  // EOF OR I/O ERROR
21          }
22          if (marker_type != 0x00) break;
23          continue;  // TOLERATE NOT WELL-FORMED
24      }

25      ASSERT(GET_CONTENT(fd, GET_POS(fd, "CURRENT") - 2) == 0xFF &&
26             GET_CONTENT(fd, GET_POS(fd, "CURRENT") - 1) != 0x00 &&
27             GET_CONTENT(fd, GET_POS(fd, "CURRENT") - 1) != 0xFF);
```

Figure 6-3: A (simplified) code snippet for an image converter from the JFIF format to the PGM format (1/2).

```
28         switch (marker_type) {
29             case 0xDA: {  // SOS: Start of Scan
30                 uint16_t length; uint8_t *params = NULL;
31                 uint8_t *data = NULL; size_t data_cap = 0, data_size = 0;
32                 ...
33 SOS_LEN_START:
34                 nbytes = read(fd, &length, sizeof(length));
35                 if (nbytes != sizeof(length)) goto SOS_END;
36                 length = ntohs(length);
37                 if (length < sizeof(length)) goto SOS_END;
38                 if ((params = malloc(length - sizeof(length))) == NULL) {
39                     if (lseek(fd, length - sizeof(length), SEEK_CUR) == -1)
40                         goto OUT_OF_LOOP;
41                     ASSERT(GET_POS(fd, "CURRENT") == GET_POS(fd, "SOS_LEN_START") +
42                              GET_CONTENT(fd, GET_POS(fd, "SOS_LEN_START")) * 256 +
43                              GET_CONTENT(fd, GET_POS(fd, "SOS_LEN_START") + 1));
44                     goto SOS_END;
45                 }
46                 nbytes = read(fd, params, length - sizeof(length));
47                 if (nbytes != length - sizeof(length)) goto SOS_END;
48                 ASSERT(GET_POS(fd, "CURRENT") == GET_POS(fd, "SOS_LEN_START") +
49                          GET_CONTENT(fd, GET_POS(fd, "SOS_LEN_START")) * 256 +
50                          GET_CONTENT(fd, GET_POS(fd, "SOS_LEN_START") + 1));

51                 for (;;) {
52                     if (data_cap == data_size) {
53                         data_cap = data_cap == 0 ? 512 : data_cap * 2;
54                         if ((data = realloc(data, data_cap)) == NULL) goto SOS_END;
55                     }
56                     nbytes = read(fd, &byte, sizeof(byte));
57                     if (nbytes != sizeof(byte)) goto SOS_END;
58                     if (byte != 0xFF) *(data + data_size++) = byte;
59                     else {
60                         nbytes = read(fd, &next_byte, sizeof(next_byte));
61                         if (nbytes != sizeof(next_byte)) goto SOS_END;
62                         if (next_byte != 0x00) {
63                             if (lseek(fd, -2, SEEK_CUR) == -1) goto OUT_OF_LOOP;
64                             break;
65                         } else *(data + data_size++) = byte;
66                     }
67                 }
68                 ASSERT(GET_CONTENT(fd, GET_POS(fd, "CURRENT")) == 0xFF &&
69                        GET_CONTENT(fd, GET_POS(fd, "CURRENT") + 1) != 0x00);
70                 ...
71 SOS_END:
72                 free(data); free(params);
73                 break;
74             }
75             ...
76         }
77 }
78 OUT_OF_LOOP: ...
```

Figure 6-4: A (simplified) code snippet for an image converter from the JFIF format to the PGM format (2/2).

If the `read` call has read a non-FF_{16} value, it indicates that the JFIF image file is not well-formed. Rather than breaking out of the loop, the image converter discards the invalid byte and continues to process the rest of the image file (line 12), which enables the converter to successfully handle illegally-formed JFIF image files that are interspersed with some invalid bytes. Only when the FF_{16} byte has been read, the image converter proceeds to read the second byte (line 11).

```
11            if (marker_ff == 0xFF) break;
12            continue;  // TOLERATE NOT WELL-FORMED
```

The converter makes the second call to the `read` function to read the second byte into the `marker_type` variable (line 14).

```
14        nbytes = read(fd, &marker_type, sizeof(marker_type));
```

If the `read` call fails to read any byte, the converter breaks out of the outermost loop, as it did with the previous `read` call (lines 15–16).

```
15        if (nbytes != sizeof(marker_type))
16            goto OUT_OF_LOOP;  // EOF OR I/O ERROR
```

If the value FF_{16} has been read once again, the first FF_{16} byte is a fill byte and the second and current FF_{16} byte may be the start of a marker. Therefore, the image converter keeps reading a single byte in a nested loop until it reads a non-FF_{16} byte (lines 17–21).

```
17        while (mark_type == 0xFF) {  // FILL BYTE
18            nbytes = read(fd, &marker_type, sizeof(marker_type));
19            if (nbytes != sizeof(marker_type))
20                goto OUT_OF_LOOP;  // EOF OR I/O ERROR
21        }
```

On the other hand, if the second byte read is 00_{16}, the JFIF image file is not well-formed.

The JPEG compression method is designed so that the 2-byte sequence FF_{16} 00_{16} may appear only within the compressed scan data that immediately follows the SOS marker segment. The converter discards both bytes of the invalid byte sequence and resiliently restarts by searching again for the first byte (FF_{16}) of a marker (line 23). The next stage of parsing, which depends on each type of marker, happens only after the converter has found a byte FF_{16} that is followed by a marker type byte which is neither FF_{16} nor 00_{16} (line 22).

```
22          if (marker_type != 0x00) break;
23          continue;  // TOLERATE NOT WELL-FORMED
```

The next step of parsing starts by dispatching the control of execution to different parts of code, depending on the marker type byte (line 28 in Figure 6-4 on page 99).

```
28      switch (marker_type) {
```

For the SOS marker segment, the image converter first makes an attempt to read the Length field of 2 bytes into the `length` variable (lines 29–34).

```
29          case 0xDA: {  // SOS: Start of Scan
30              uint16_t length; uint8_t *params = NULL;
31              uint8_t *data = NULL; size_t data_cap = 0, data_size = 0;
32              ...
33  SOS_LEN_START:
34              nbytes = read(fd, &length, sizeof(length));
```

If the `read` function cannot read the specified number of bytes because of a premature end-of-file condition or a physical I/O error, the converter jumps to the end of the current `case` statement, which is labeled `SOS_END`, and will break out of the outermost loop at the next iteration (line 35).

```
35              if (nbytes != sizeof(length)) goto SOS_END;
```

Otherwise, the image converter calls the `ntohs` function to change the `length` variable from network byte order to host byte order (line 36).

```
36              length = ntohs(length);
```

Because the Length field includes the length of itself, the result value in the `length` variable must be greater than or equal to 2 (i.e. `sizeof(length)`) (line 37).

```
37              if (length < sizeof(length)) goto SOS_END;
```

Next, because the Parameters field is of variable length, the converter invokes the `malloc` function to dynamically allocate a memory block of size specified by means of the Length field (line 38). The `malloc` function returns NULL if the requested allocation cannot be fulfilled. In that case, the image converter skips the Parameters field by adjusting the file position indicator accordingly (lines 39–44). Note that the scan data that follows the SOS marker segment will be automatically skipped by the outermost loop because each of its iteration starts by searching for the next marker.

```
38              if ((params = malloc(length - sizeof(length))) == NULL) {
39                  if (lseek(fd, length - sizeof(length), SEEK_CUR) == -1)
40                      goto OUT_OF_LOOP;
41                  ASSERT(GET_POS(fd, "CURRENT") == GET_POS(fd, "SOS_LEN_START") +
42                            GET_CONTENT(fd, GET_POS(fd, "SOS_LEN_START")) * 256 +
43                            GET_CONTENT(fd, GET_POS(fd, "SOS_LEN_START") + 1));
44                  goto SOS_END;
45              }
```

When the `malloc` call is successful, the image converter proceeds to read the Parameters field into the allocated buffer (line 46).

```
46              nbytes = read(fd, params, length - sizeof(length));
```

If the `read` call cannot read the specified number of bytes, the converter jumps to the

end of the current `case` statement, which is labeled `SOS_END`, like the previous `read` call (line 47).

```
47                  if (nbytes != length - sizeof(length)) goto SOS_END;
```

If the `read` call successfully reads the specified number of bytes (and the program has been implemented correctly in conformity with the JFIF specification), it means that the converter has just read all the bytes of the SOS marker segment and is ready for reading the following compressed scan data.

Because the compressed scan data does not have length information within it, the image converter reads bytes one by one in a nested loop and checks if it has come across the beginning of a new marker. For the same reason, the `data` storage, where the scan data is stored into, is expanded as needed (lines 52–55).

```
52                      if (data_cap == data_size) {
53                          data_cap = data_cap == 0 ? 512 : data_cap * 2;
54                          if ((data = realloc(data, data_cap)) == NULL) goto SOS_END;
55                      }
```

The converter attempts to read a single byte after allocating enough memory space (lines 56–57).

```
56                      nbytes = read(fd, &byte, sizeof(byte));
57                      if (nbytes != sizeof(byte)) goto SOS_END;
```

If the byte read is not FF_{16}, it belongs to the scan data and is appended to the `data` storage.

```
58                      if (byte != 0xFF) *(data + data_size++) = byte;
```

On the other hand, if the byte read is FF_{16}, it may be the first byte of a new marker. To fully grasp the situation, the converter makes an attempt to read the next byte

into the `next_byte` variable (lines 59–61).

```
59                  else {
60                      nbytes = read(fd, &next_byte, sizeof(next_byte));
61                      if (nbytes != sizeof(next_byte)) goto SOS_END;
```

If the second byte is non-zero, the two bytes really mark the beginning of a new marker. In that case, the converter relocates the file position indicator and breaks out of the inner loop so that the next iteration of the outermost loop starts at the beginning of the next marker (line 62–64).

```
62                      if (next_byte != 0x00) {
63                          if (lseek(fd, -2, SEEK_CUR) == -1) goto OUT_OF_LOOP;
64                          break;
```

If the two bytes comprise a byte sequence FF_{16} 00_{16}, the first byte FF_{16} is added to the end of the `data` storage but the second byte 00_{16} is discarded. Note that the byte value FF_{16} is encoded as the 2-byte sequence FF_{16} 00_{16} in the compressed scan data (line 65).

```
65                      } else *(data + data_size++) = byte;
```

When the image converter breaks out of the inner loop by detecting a new marker, it has successfully read all the bytes of the SOS marker segment and the following scan data. The converter proceeds to decode the compressed data in the `data` storage with information obtained from the other marker segments, such as the Huffman tables from the DHT marker segment and the quantization tables from the DQT marker segment.

Similarly to the SOS marker segment, the other marker segments (not shown here) are processed as resiliently as possible.

6.3 Input Specification

After the inner loop that searches for a new JFIF marker (lines 3–24 in Figure 6-3 on page 98), the developer specified that the current file position indicator should be placed immediately after a 2-byte sequence whose first byte is FF_{16} and whose second byte is not either 00_{16} or FF_{16}. In other words, it should be guaranteed that the image converter has found the 2-byte beginning of a JFIF marker segment and is ready for reading the Parameters field of the marker segment. In the specification language, the property is specified as follows (lines 25–27):

```
25      ASSERT(GET_CONTENT(fd, GET_POS(fd, "CURRENT") - 2) == 0xFF &&
26             GET_CONTENT(fd, GET_POS(fd, "CURRENT") - 1) != 0x00 &&
27             GET_CONTENT(fd, GET_POS(fd, "CURRENT") - 1) != 0xFF);
```

The first line of the assertion specifies that the byte value FF_{16} is supposed to be observed at offset -2 from the current file position indicator, denoted by `GET_POS(fd, "CURRENT") - 2`. The second and third lines require that the byte just before the current file position, denoted by `GET_POS(fd, "CURRENT") - 1`, should be neither 00_{16} nor FF_{16}.

The image converter strives to handle the Parameter field in a resilient way. If the dynamic allocation for the Parameter field fails, the converter attempts to pass over the current marker segment and go on the next one (lines 38–45 in Figure 6-4 on page 99).

```
38          if ((params = malloc(length - sizeof(length))) == NULL) {
39              if (lseek(fd, length - sizeof(length), SEEK_CUR) == -1)
40                  goto OUT_OF_LOOP;
41              ASSERT(GET_POS(fd, "CURRENT") == GET_POS(fd, "SOS_LEN_START") +
42                      GET_CONTENT(fd, GET_POS(fd, "SOS_LEN_START")) * 256 +
43                      GET_CONTENT(fd, GET_POS(fd, "SOS_LEN_START") + 1));
44              goto SOS_END;
45          }
```

For this strategy to work, the developer needs to give the `lseek` function the exact number of bytes to skip so that it skips to the end of the Parameters field. Because the size of the Parameters field is dynamically determined by the Length field, the amount to skip is specified by an arithmetic expression involving the `length` variable, which the result of endianness conversion was stored into (line 39). To ensure the correctness of the arithmetic expression, the developer specified that the current file position indicator should be at the end of the Parameter field (and therefore the end of the SOS marker segment) after the `lseek` call (lines 41–43). Specifically, the current file position indicator at line 41, denoted by `GET_POS(fd, "CURRENT")`, should be different from the file position indicator at the beginning of the Length field, denoted by `GET_POS(fd, "SOS_LEN_START")`, by the total size of the Length and Parameters fields, which is obtained by interpreting the Length field of the current marker segment in big-endian byte order. `SOS_LEN_START` is a C label attached to a program point where the file position indicator is located between the marker type field and the Length field (line 33 in Figure 6-4 on page 99). Therefore, the n-th byte of the Length field is found at `GET_POS(fd, "SOS_LEN_START")` $+ n$, and is multiplied by 2^{8-8n} because the most significant byte comes first in big-endian format.

On the other hand, if the dynamic allocation for the Parameter field succeeds and the image converter successfully reads the specified number of bytes into the allocated buffer (and the program has been implemented correctly in conformity with the JFIF specification), it means that the converter has just read all the bytes of the SOS marker segment and is ready for reading the following compressed scan data (lines 46–50). The property was explicitly specified using the same assertion as above: the current file position should be away from the file position at the program point `SOS_LEN_START` by a big-endian integer value in the Length field (line 48–50).

```
46            nbytes = read(fd, params, length - sizeof(length));
47            if (nbytes != length - sizeof(length)) goto SOS_END;
48            ASSERT(GET_POS(fd, "CURRENT") == GET_POS(fd, "SOS_LEN_START") +
49                    GET_CONTENT(fd, GET_POS(fd, "SOS_LEN_START")) * 256 +
50                    GET_CONTENT(fd, GET_POS(fd, "SOS_LEN_START") + 1));
```

Lastly, after the nested loop that reads the compressed scan data, the developer specified that the loop completes normally only when the current file position indicator is located at the start of a byte sequence whose first byte is FF_{16} and whose second byte is not 00_{16} (lines 68–69). In other words, the file position indicator must be placed just after the end of the compressed scan data at that moment.

```
68            ASSERT(GET_CONTENT(fd, GET_POS(fd, "CURRENT")) == 0xFF &&
69                   GET_CONTENT(fd, GET_POS(fd, "CURRENT") + 1) != 0x00);
```

The developer have added the similar input specifications for all the other marker segments, although they are not shown in Figure 6-4 on page 99.

6.4 Verification

As with the PNG image viewer in Chapter 2, the proposed system successfully synthesized verification conditions from the specifications for the JPEG image converter, verified all of them, and therefore confirmed that:

- The image converter has been written resiliently and correctly by starting with searching for a JFIF marker and skipping fill bytes.
- Even in the event of errors, the converter consumes all the bytes of each marker segment so that it can continue to read the following marker segments or scan data from the correct file position.
- The program accurately detects the end of the compressed scan data by searching for a byte sequence whose first byte is FF_{16} and whose second byte is not 00_{16}.

Chapter 7

Implementation

I have implemented a prototype of the proposed system to evaluate its practical aspects. The prototype is written in the OCaml language and targets programs written in the C language. The C Intermediate Language (CIL) program analysis infrastructure [38] is used to parse a C program and to perform a backward data-flow analysis that figures out what condition at each program point should be satisfied to guarantee the input specifications in the program. The data-flow analysis propagates verification conditions backwards against the control flow of a program, as illustrated in Section 2.4. Whenever a condition is propagated into a program point, the system decides whether to propagate the condition further by checking if the old condition stored at the program point implies the new condition. To do so, the system translates verification conditions into WhyML and feeds the result into the Why3 platform [18]. Why3 is a platform for deductive program verification and provides a unified front-end to multiple third-party theorem provers, such as Alt-Ergo [11], CVC3 [8], and Z3 [37], by means of a series of transformations that handle the differences among the input syntax and logic of those theorem provers. Although it turned out that Z3 was sufficient for verifying all of our case study programs, this architecture will facilitate employing other decision procedures as necessary. To precisely model the semantics of byte-wise operations on multi-byte fields, such as length fields and field delimiters, I extended an alias analysis [16] in an array-index-sensitive way. I have observed that such fields are usually read into small fixed-sized arrays and that those arrays are mainly indexed by constants.

Table 7.1: Verification Times.

Application	Lines of Code	Input Specifications	Verification Time
JPEG Converter	704	22	4m 26s
PNG Viewer	346	6	1m 45s

They are scalably and effectively handled by an array-index-sensitive pointer analysis.

Table 7.1 presents a summary of the specification and verification results for the two applications shown in Chapters 2 and 6. Column 1 identifies the applications and Column 2 presents their sizes in terms of source lines of code. For the PNG viewer, I needed to add specifications at 6 locations because the PNG file format is very regular. All the fields of a PNG image file share the same structure, which is shown in Figure 2-2 on page 26, and are therefore processed in the same code region of the PNG viewer. On the other hand, each field of a JPEG image file has its own structure, depending on its marker type, and is handled differently by the JPEG converter. As a result, I needed to specify each of them separately at 22 locations. The system successfully synthesized verification conditions from all the specifications and verified them within 5 minutes, when the Z3 decision procedure was given a 1-second timeout.

Chapter 8

Related Work

Parser Generators for Binary Formats. Researchers have been studying approaches for specifying a precise grammar for a binary data format and using a parser generator to automatically synthesize a correct parser for the format [6, 7, 19, 34].

Fisher and Gruber [19], for instance, presented the PADS system that provides a declarative data description language in which a developer can describe the physical layout of a binary data stream and specify the expected semantic properties of its fields. The language provides a range of base types for describing atomic data, such as integers and strings, and a range of structured types for describing compound data, such as record-like structures, unions, and arrays. Each of those types can be associated with a predicate that specifies the semantic properties that it must satisfy. From a PADS description, the system produces a C library that includes a function for parsing the described file format into its in-memory representation along with a parse descriptor. The parse descriptor records the locations and kinds of errors that occurred during parsing.

Recently, Bangert and Zeldovich [7] have investigated an approach to generate a packet parser from a protocol grammar. Their tool, which is named Nail, uses a single grammar to define both of the external and in-memory representations of a binary data format. To establish a semantic bijection between those representations, Nail does not support semantic actions. As a result, Nail can synthesize a generator for the data format as well as its parser. By allowing a developer to provide stream

transformations that are called during parsing and output generation, Nail can support protocol features that are challenging to handle in other tools, such as size and offset fields, checksums, and compressed data.

While these works have focused on how to use a parser generator and its domain-specific language to synthesize a correct parser from an input specification, I am more concerned with how to verify that a developer has implemented a parser correctly and reliably in his or her favorite programming language. Also, while the works have usually put a focus on parsing valid input and detecting errors, I am more interested in how to recover from those errors once they are detected. Specifically, this thesis presents error recovery strategies that make good use of the inherent resilience of input data and shows how to check if the error recovery strategies have been implemented reliably.

File Format Inference. Driscoll, Burton, and Reps [17] have developed a tool called PCCA (Producer-Consumer Conformance Analyzer), which determines whether two programs in a producer/consumer relationship are compatible. From the interprocedural control-flow graph of each program, PCCA infers a variant of a pushdown automaton, called a visibly pushdown automaton [4], that conservatively models the language that the program produces or consumes. Because a visibly pushdown automaton recognizes a restricted form of a context-free language that is closed under complementation and intersection, PCCA can determine if the language that the producer program generates is a subset of the language that the consumer program accepts.

The technique used in PCCA cannot be used to verify those correctness properties that this thesis aims at because PCCA over-approximately infers the file format that a program can handle. Also, it will be awkward, if possible, to represent a binary file format, especially its length fields, by a context-free language. Generally, context-sensitive features are still out of scope even for state-of-the-art format inference systems [10, 14, 25].

Pointer Analysis. I have used an alias analysis to relieve a developer of manually providing loop invariants on memory states at the beginning of a loop. The pointer

analysis analyzes arrays whose sizes are determined at compile time in an array-index-sensitive fashion. This makes it possible to model the semantics of byte-wise operations precisely, which is indispensable for verifying programs involving those operations. Most published works [5, 16, 23, 36, 43, 44] do not distinguish array elements and collapse them into one object for scalability. My observation is that those arrays are usually used for reading the fields of binary file formats that contain metadata information, such as the length of other fields and field delimiters, and relatively small. Arrays whose sizes are determined during runtime, such as dynamically allocated ones, are analyzed in an array-index-insensitive fashion, as do most other works.

Chapter 9

Conclusion

Many programs read inputs from files and must process the contents of the files properly to produce the correct output. Unfortunately, due to the complexity of the input files that modern software systems must deal with, the input file processing code has continued to be an annoying source of errors and security vulnerabilities in modern software systems.

This thesis presents new techniques for verifying correctness properties of those programs that handle input files. These techniques apply to programs written in standard programming languages and make it possible to verify that the programs establish the correct relationships among program execution points, the current locations of file position indicators, and the contents of the input files. The thesis presents a specification language that developers can use to specify these relationships involving the file position indicator and file contents at different program points. It also presents a program verification system that verifies that, for all possible input files, the specified relationships are established in all program executions. The syntax and semantics of the specification language is formally defined, and the soundness of the verification system has been shown. The system successfully synthesized verification conditions from the specifications for a PNG image viewer and a JPEG image converter, and managed to verify all of them.

Bibliography

[1] GNU Bison. https://www.gnu.org/software/bison.

[2] Mark Adler, Thomas Boutell, John Bowler, Christian Brunschen, Adam M. Costello, Lee Daniel Crocker, Andreas Dilger, Oliver Fromme, Jean-loup Gailly, Chris Herborth, Alex Jakulin, Neal Kettler, Tom Lane, Alexander Lehmann, Chris Lilley, Dave Martindale, Owen Mortensen, Keith S. Pickens, Robert P. Poole, Glenn Randers-Pehrson, Greg Roelofs, Willem van Schaik, Guy Schalnat, Paul Schmidt, Michael Stokes, Tim Wegner, and Jeremy Wohl. *Portable Network Graphics (PNG) Sepcification*, second edition, 2003.

[3] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling.* Prentice-Hall, 1972.

[4] Rajeev Alur and P. Madhusudan. Adding nesting structure to words. *Journal of the ACM*, 56(3), 2009.

[5] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, University of Copenhagen, 2001.

[6] Godmar Back. DataScript — a specification and scripting language for binary data. In *Proceedings of Generative Programming and Component Engineering (GPCE)*, 2002.

[7] Julian Bangert and Nickolai Zeldovich. Nail: A practical tool for parsing and generating data formats. In *Proceedings of Operating Systems Design and Implementation (OSDI)*, 2014.

[8] Clark Barrett and Cesare Tinelli. CVC3. In *Proceedings of Computer Aided Verification (CAV)*, 2007.

[9] Stephen C. Johnson. Yacc: Yet Another Compiler-Compiler. Technical Report No. 32, AT&T Bell Laboratories, 1975.

[10] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of Computer and Communications Security (CCS)*, 2007.

[11] Sylvain Conchon, Albin Coquereau, Mohamed Iguernlala, and Alain Mebsout. Alt-Ergo 2.2. In *Proceedings of Satisfiability Modulo Theories (SMT)*, 2018.

[12] Robert Paul Corbett. *Static Semantics and Compiler Error Recovery.* PhD thesis, University of California, Berkeley, 1985.

[13] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of Principles of Programming Languages (POPL)*, 1977.

[14] Weidong Cui, Marcus Peinado, Karl Chen, Helen J. Wang, and Luiz Irun-Briz. Tupni: Automatic reverse engineering of input formats. In *Proceedings of Computer and Communications Security (CCS)*, 2008.

[15] Pascal Cuoq, Boris Yakobowski, Matthieu Lemerre, André Maroneze, Valentin Perrelle, and Virgile Prevosto. *Frama-C's value analysis plugin-in*, aluminium-20160501 edition, 2016.

[16] Manuvir Das, Ben Liblit, Manuel Fähndrich, and Jakob Rehof. Estimating the impact of scalable pointer analysis on optimization. In *Proceedings of Static Analysis Symposium (SAS)*, 2001.

[17] Evan Driscoll, Amanda Burton, and Thomas Reps. Checking conformance of a producer and a consumer. In *Proceedings of Foundations of Software Engineering (FSE)*, 2011.

[18] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In *Proceedings of European Symposium on Programming (ESOP)*, 2013.

[19] Kathleen Fisher and Robert Gruber. PADS: A domain-specific language for processing ad hoc data. In *Proceedings of Programming Language Design and Implementation (PLDI)*, 2005.

[20] Paul Gazzillo and Robert Grimm. SuperC: Parsing all of C by taming the pre-processor. In *Proceedings of Programming Language Design and Implementation (PLDI)*, 2012.

[21] Matthias Gelbmann. The PNG image file format is now more popular than GIF. `https://w3techs.com/blog/entry/the_png_image_file_format_is_now_more_popular_than_gif`, 2013.

[22] Dick Grune and Ceriel J. H. Jacobs. *Parsing Techniques.* Springer, 2008.

[23] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of Program Analysis for Software Tools and Engineering (PASTE)*, 2001.

[24] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10), 1969.

[25] Matthias Höschele and Andreas Zeller. Mining input grammars with AUTO-GRAM. In *Proceedings of International Conference on Software Engineering Companion (ICSE-C)*, 2017.

[26] Edgar T. Irons. A syntax directed compiler for ALGOL 60. *Communications of the ACM*, 4(1), 1961.

[27] Chinawat Isradisaikul and Andrew C. Myers. Finding counterexamples from parsing conflicts. In *Proceedings of Programming Language Design and Implementation (PLDI)*, 2015.

[28] Donald E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6), 1965.

[29] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7), 2009.

[30] Alan Leung, John Sarracino, and Sorin Lerner. Interactive parser synthesis by example. In *Proceedings of Programming Language Design and Implementation (PLDI)*, 2015.

[31] Fan Long, Vijay Ganesh, Michael Carbin, Stelios Sidiroglou, and Martin Rinard. Automatic input rectification. In *Proceedings of International Conference on Software Engineering (ICSE)*, 2012.

[32] Fan Long, Stelios Sidiroglou-Douskos, Deokhwan Kim, and Martin Rinard. Sound input filter generation for integer overflow errors. In *Proceedings of Principles of Programming Languages (POPL)*, 2014.

[33] Fan Long, Stelios Sidiroglou-Douskos, and Martin Rinard. Automatic runtime error repair and containment via recovery shepherding. In *Proceedings of Programming Language Design and Implementation (PLDI)*, 2014.

[34] Peter J. McCann and Satish Chandra. Packet Types: Abstract specification of network protocol messages. In *Proceedings of SIGCOMM*, 2000.

[35] John Miano. *Compressed Image File Formats: JPEG, PNG, GIF, XBM, BMP*. Addison-Wesley Professional, 1999.

[36] Ana Milanova, Atanas Rountev, and Ryder G. Barbara. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology*, 14(1), 2005.

[37] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[38] George C. Necula, Scott McPeark, Shree P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of Compiler Construction (CC)*, 2002.

[39] Terence Parr, Sam Harwell, and Kathleen Fishe. Adaptive LL(*) parsing: The power of dynamic analysis. In *Proceedings of Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2014.

[40] Martin Rinard. Living in the comfort zone. In *Proceedings of Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2007.

[41] Klaus Samuelson and Friedrich L. Bauer. Sequential formula translation. *Communications of the ACM*, 3(2), 1960.

[42] Stelios Sidiroglou-Douskos, Eric Lahtinen, Nathan Rittenhous, Paolo Piselli, Fan Long, Deokhwan Kim, and Martin Rinard. Targeted automatic integer overflow discovery using goal-directed conditional branch enforcement. In *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[43] Yannis Smaragdakis, Martin Bravenboer, and Ondřej Lhoták. Pick your contexts well: Understanding object-sensitivity. In *Proceedings of Principles of Programming Languages (POPL)*, 2011.

[44] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of Principles of Programming Languages (POPL)*, 1996.

[45] Robert W. Floyd. Assigning meanings to programs. In *Proceedings of Symposia in Applied Mathematics (PSAM)*, 1967.

[46] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of Programming Language Design and Implementation (PLDI)*, 2011.