ORSAY
N^o d'ordre: 9366

UNIVERSITÉ DE PARIS-SUD 11

CENTRE D'ORSAY

THÈSE

présentée
pour obtenir

le grade de docteur en sciences DE L'UNIVERSITÉ PARIS XI

PAR

Yannick MOY

SUJET :

Preuve automatique et modulaire de la sûreté de fonctionnement des programmes C

Automatic Modular Static Safety Checking for C Programs

soutenue le 15 janvier 2009 devant la commission d'examen

MM.	Burkhart Wolff
	K. Rustan M. Leino
	Xavier Leroy
	Michael Norrish
	Claude Marché
	Pierre Crégut

J'ai commencé cette thèse, après quatre années à la R&D de PolySpace, parce que je voulais comprendre comment mieux répondre au problème d'analyse et de vérification des programmes informatiques. Ces trois années m'ont largement satisfaites sur ce point, et m'ont permis de connaître des personnes admirables autant du point de vue professionnel que personnel.

Je voudrais remercier pour leur fantastique état d'esprit toutes les personnes de l'équipe ProVal, avec une mention spéciale à Évelyne et Jean-Christophe pour leurs gâteaux délicieux, et Jean-Christophe et Sylvain pour l'ambiance sonore chaleureuse.

Je remercie respectueusement mes deux directeurs de thèse, Claude et Pierre, pour m'avoir accompagné ces trois années sur le dur chemin de la recherche, et je leur dis mon admiration pour leur talent et leur travail.

Je veux remercier sincèrement toutes les personnes de mon jury en plus de mes directeurs, Burkhart, Rustan, Xavier et Michael, qui sont tous des chercheurs que j'admire profondément. Vos suggestions, vos remarques et vos questions m'ont permis de mieux comprendre mon sujet et d'avancer encore un peu plus.

Je décerne la palme du meilleur soutien de thèse à Jean-Christophe, pour m'avoir motivé tous les mercredis (enfin presque) à aller jouer au foot, en plus d'être une source inépuisable d'énergie et d'inventivité, toujours prêt à discuter d'une idée ou d'un problème.

J'octroie la coupe du meilleur prof à François pour qui j'ai donné des TD à l'Ecole Polytechnique, c'était un plaisir d'apporter ma présence à un travail aussi magnifiquement présenté.

Je veux remercier tous ceux avec qui j'ai travaillé, discuté de problèmes ou partagé des bons moments au travail durant ces trois années: Yves-Marie, Guillaume, Fayçal, Laurent et Cuit aux Orange Labs, Benjamin, Virgile, Julien, Pascal et Patrick au CEA, Dillon à Dassault Aviation, David à Airbus, Charles et Xavier à EADS, Francesco et Nikolaj à Microsoft Research, Matteo à Google, Sriram aux NEC Labs.

Je remercie tout spécialement ces amis qui sont passés ou sont encore à Orsay: Nicolas S., Jean-François, Aurélien, Alexandre, Stéphane, Johannes, Louis. Je réserve une mention spéciale à Florence pour sa bonne humeur contagieuse, en attendant sa soutenance !

Merci à tous mes amis, ceux qui ont pu se déplacer pour la soutenance et tous les autres, votre soutien m'a toujours aidé.

Merci papa, maman, Pierre, Jojo et Manue, vous êtes mon public admiratif préféré.

Enfin, un million de fois merci ma femme chérie, Alexandra, rien de tout cela n'aurait existé sans toi.

Contents

List of Figures

Chapter 1

Introduction

First, consider the appalling fact that most security flaws are caused by buffer overflow. Why is this appalling? Because there is absolutely no need for anyone, ever, to write a program that contains buffer overflows. That we continue to do so is a reflection our addiction to atrocious languages like C++. There are perfectly good languages around that make it simply impossible to write code that can cause buffer overflows. We should use them. No research is necessary. No proofs are necessary. It's a decidable - indeed solved - problem.

Anthony Hall

Contents

1.1 Language-Based Dependability of C Programs

1.1.1 Problem Statement

This thesis belongs to the broad domain of software **dependability** analyses, where one aims at getting higher degrees of confidence in the ability of a software system to deliver an appropriate service. Dependability is the integration of various concepts. Quoting the work of Avižienis, Laprie and Randell [6], we specifically target four components of dependability in this thesis:

- *reliability*, the continuity of correct service;
- *safety*, the absence of catastrophic consequences on the user(s) and the environment;
- *integrity*, the absence of improper system state alteration;
- *security*, by addressing the part that derives from integrity, the absence of unauthorized system state alteration.

Dependability is especially important for life-critical, mission-critical or business-critical software systems, for which a failure to meet their dependability requirements may lead to a human loss (military, avionics, automotive, health-care, *etc.*) or an economic threat (aerospace, consumer electronics, Internet, *etc.*). Due to the dematerialization of many services (banking, administration, personal information, *etc.*), dependability of software systems also appears to be at the cornerstone of privacy. Overall, due to (1) the ever increasing relative importance of software in complex systems, a.k.a. software-dominant systems, (2) the pervasive presence of software in many devices that therefore qualify as embedded systems, and (3) the unavoidable connectivity of software systems in *ad hoc* networks or over the Internet, dependability of software is a crucial challenge for the years to come.

Industries that develop software-dominant systems and embedded systems have started expressing the need for verification tools, much as they expressed the need for testing and debugging tools in the past. For a large part, this demand concerns programs written in the **C language**, a programming language designed in the 1970's for building operating systems [109]. Since then, C has become the standard programming language for system software, with most modern computer languages relying on a layer of C programs for their runtime environment (*e.g.*, PERL, PYTHON, JAVA, OCAML, *etc.*).

Dennis M. Ritchie, the initial author of C, once wrote that *C has been characterized (both admiringly and invidiously) as a portable assembly language*. Indeed, C shares the same advantages as assembly languages, the "native" languages of computers, which collectively qualify them as *low-level* languages:

- *simplicity*: there is no abstraction between the programmer's intent and the machine code actually run on the machine;
- *efficiency*: the programmer may directly manipulate the computer's memory.

Contrary to assembly languages, C makes it possible to write *portable* programs, *i.e.*, programs that can be compiled to run correctly on many different machine architectures. Due to its low-level orientation though, it remains the programmer's responsibility to ensure portability, by carefully crafting his programs with this goal in mind. Similarly, although most modern programming abstractions present in *high-level* languages (strong typing, objects, closures, transactions, *etc.*) can be encoded in C, their correct use depends on each programmer's craftsmanship.

Although C is an old and generic language, and many powerful specialized computer languages have been designed since then, knowledge of C still appears to be the second most demanded skill from programmers in job offers on the Web in 2007 [64], and the first most demanded general-purpose programming language. The continued industrial interest for C stems from many reasons independent of its own qualities: availability of a large pool of skilled programmers, support by a large range of libraries, tools and execution platforms with a proven record of dependability, *etc*. Whatever the reasons, C is still a very live language, with applications in many different industries, some of which critically depend on software.

The high number of critical C programs in industry combined to the lack of language support in C for ensuring the most basic properties explain the general interest for analysis techniques and tools targeting the C language. Our goal with this thesis is to provide automated techniques to verify the part of software dependability that depends on the use of the C language, with a focus on memory safety, which is the crucial property that the computer's memory is not misused. We call it **static safety checking** [181] throughout this thesis, rather than the more general term of *verification*, to emphasize that it is safety that we are after. Moreover, static safety checking defines a verification goal, not the means and techniques to reach it.

1.1.2 Technical Account

Generalities A *program* is a human-readable textual representation of a list of instructions that must be executed by a machine. A program is written in a *programming language* that defines the syntax (what is a valid sentence in this language) and semantics (what a valid sentence means) of the language. A program is generally structured as a set of *functions* which provide each a piece of functionality. A *programmer* is a skilled practitioner who writes programs. A *hacker* always means an above-than-average skilled programmer, sometimes with a negative connotation that he uses his skills to commit criminal actions.

A program is made up of *code*, which is the term to designate any piece of instructions, either in textual representation (source code) or machine representation (machine code). A program in textual representation is generally *compiled* into machine code by a *compiler* (which is another program). A program in machine code representation is also called an *executable*. The size of a program in textual representation is measured in source lines of code (*sloc*).

An executable can be *executed* or *run* on a host machine. During execution, software interacts with hardware to open files, input and output information, *etc*. Failing to comply with hardware interface specifications may result in *runtime errors*, *i.e.*, errors that manifest during execution. Among runtime errors, *memory errors* are especially obnoxious. They break the abstractions upon which a program's meaning relies, leading to erratic behavior or even opening the possibility that a different code is executed than the one intended, which can be exploited by an attacker.

Errors in source code that make runtime errors possible are called *vulnerabilities*. Attackers may profit from these vulnerabilities to build *exploits*, *i.e.*, pieces of software that interact with vulnerabilities to modify software execution in a way that was not intended.

Memory errors can be classified depending on the kind of memory perverted. A code being executed sees computer memory as divided into 5 segments:

- *code segment*: portion of the memory that holds the code executed. In general, it is read-only, meaning its content cannot be altered.
- *data and BSS segments*: portions of the memory that contain global data defined and accessed by the code. They do not contain control structures, but they can hold pointers to code, hence perverting them may allow one to build exploits.
- *stack*: portion of the memory that holds data local to functions and control elements, like the return address of functions. Perverting the stack is also known as “smashing the stack”. It is the easiest memory error to exploit to gain control over a computer, and thus the most common.
- *heap*: portion of the memory that holds dynamically allocated data, whose lifetime does not end when a function returns. Like the data and BSS segments, it can hold pointers to code, thus allowing exploits. Another kind of exploit is based on corrupting the internal data used for dynamic memory management.

Buffer Overflow Due to its low-level orientation, the C language does not protect against runtime errors in general, and memory errors in particular. Among memory errors, the infamous *buffer overflow* (a.k.a. buffer overrun) plays an crucial role, as the main door to remote code execution. Here is a perfectly mundane C program that contains a buffer overflow vulnerability:

```
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv) {
  char tab[100];
  if (argc != 2) return 1;
  strcpy(tab, argv[1]);
  printf("%s\n", tab);
  return 0;
}
```

This program simply expects to be called with one string of characters as argument, and it echoes this string back, much as the command `echo` on most systems. It does so by copying the argument string in its own buffer `tab` before it prints the content of this buffer. The problem is that this program does not guard against the possibility that a (malicious) user might pass in a string of more than 100 characters. Should that event occur, the buffer `tab` of size 100 would be overflown, which can be exploited on some systems by an attacker to take control. As an example, running this program on my laptop with a string of 102 characters (counting the final '\0') raises the error *** stack smashing detected *** and correctly aborts the program, but running it with a string of 101 characters does not. Thus, an off-by-one buffer overflow goes undetected, potentially opening the possibility of a take-over by an attacker, should such a program be executed on my laptop. In fact, such attacks do occur frequently in real life.

1.1.3 Historical Account

There are countless stories of *software bugs* (errors) leading to catastrophic failures, but we only report here those failures that stem from memory safety vulnerabilities in C programs, *i.e.*, the kind of errors this thesis targets. Most of the vulnerabilities we report occur in PCs or servers system software, because (1) these systems are typically subject to looser requirements than life-critical systems, which accounts for the presence of more bugs, (2) they are connected over the Internet, which makes worldwide attacks exploiting these bugs possible, thus focusing the general public interest, and (3) these systems equip general public services and devices (*e.g.*, PCs), so that the analysis of the errors that made the attacks possible is usually made public. There are many other reasons for this situation, from sociological ones such as the failure of systems administrators to apply security patches in a timely manner, to political ones that explain why some systems are attacked. The usual pattern of these attacks is that hackers build exploits, *i.e.*, pieces of software that take advantage of software vulnerabilities to modify target software execution in a way that was not intended, and spread it through the Internet from machines already under their control, a.k.a. zombies.

According to the data gathered by institutes such as CERT and SANS, memory safety errors in C programs have continuously been one of the major sources of attacks on the Internet since its early beginning (Morris worm, 1988) to these days (2008).

November 3, 1988 - The *Morris worm* infects 6,000 computers, 10% of Internet at the time, unintentionally causing the first worldwide Denial Of Service (DOS) attack (an attack in which computers infected fail to deliver proper service, either because they crash or because they are overwhelmed by malicious requests or information). It was created by student Robert T. Morris in an attempt to gauge the size of the Internet. The worm spreads by exploiting known vulnerabilities in C programs, among which a buffer overflow in `fingerd`.

December 13, 1988 - Creation of the Computer Emergency Response Team (CERT) at Carnegie Mellon University, to address computer security concerns over the Internet. This is a direct consequence of the Morris worm.

1989 - Creation of the SANS (SysAdmin, Audit, Network, Security) Institute as a cooperative research and education organization on information security, operating the Internet Storm Center, Internet's early warning system. This is also a direct consequence of the Morris worm.

November, 1996 - The hacker Elias Levy, under the name Aleph One, publishes an article in Phrack Magazine [144], in which he describes in detail the mechanisms to trigger a buffer overflow error in a C program similar to our echoing program above. This makes it possible for non-experts to write exploits based on buffer overflow vulnerabilities.

June 2, 2000 - The SANS Institute publishes for the first time a list of the 10 most critical Internet security threats (later known as the annually updated "SANS Top 10 list"), half of which are related to buffer overflows in C programs.

June 19, 2001 - CERT publishes note CA-2001-13 describing a buffer overflow vulnerability in a C program in Microsoft Indexing Services software. A worm based on this vulnerability named *Code Red* starts spreading on July 13, 2001, quickly infecting more than 300,000 machines in a few days, before it launches a DOS attack on a set of fixed IP addresses. The estimated cost is in excess of $2.6 billion [1].

January 15, 2002 - Bill Gates, CEO of Microsoft, issues a memo to all Microsoft employees, the *Trustworthy Computing* memo, which launches the first Microsoft Security Initiative. It makes security a top priority in Microsoft products. It encourages the diffusion of the book *Writing Secure Code*, which identifies buffer overflow in C programs as the public enemy #1. This is a direct consequence of the Code Red worm.

July 24, 2002 - Microsoft issues security bulletin MS02-039 which warns against possible remote code execution in Microsoft SQL Server and Microsoft Desktop Engine, based on a buffer overflow in a C program. As usual in these cases, it also gives a security patch to remove this vulnerability. On January 25, 2003, a worm named *SQL Slammer* based on this vulnerability starts spreading, infecting 75,000 unpatched Internet servers. Although its only action is to replicate by sending messages to random IP addresses, the parasitic traffic generated by the attack causes a dramatic slowdown of overall Internet traffic for an estimated cost of $1 billion[2].

January 2003 - C.A.R. Hoare, a leading scientist in computer programming logics, publishes a position paper [91] where he urges the research community to gather around the construction of a new kind of compiler that would guarantee correctness of programs. Following his impulse, many researchers have recognized the importance of software dependability, and granted it the status of a Grand Challenge (GC6: dependable systems evolution) whose goal is to federate research and industrial efforts on this subject for the 15 years to come.

July 16, 2003 - Microsoft issues security bulletin MS03-026 which warns against possible remote code execution in Microsoft Windows, based on a buffer overflow in a C program, and gives an appropriate patch. On August 11, 2003, a worm named *Blaster* based on this vulnerability starts spreading, infecting more than 16 million unpatched computer systems (mostly PCs). It floods server `windowsupdate.com` with messages to create a DOS attack, for an estimated cost of $1.3 billion[3] in cleaning and lost productivity.

April 13, 2004 - Microsoft issues security bulletin MS04-011 which warns against possible remote code execution in Microsoft Windows, based on a buffer overflow in a C program, and gives an appropriate patch. On April 30, 2004, a worm named *Sasser* based on this vulnerability starts spreading, forcing computers to restart and generating parasitic Internet traffic. It infects more than 8 million unpatched computer systems (mostly PCs) for an estimated cost of $980 million.

2005 - Researchers from Microsoft publish an article [84] in which they describe a tool to detect buffer overflows in large legacy codebases, based on logical annotations intro-

[1] *The cost of Cod Red*, USA Today, Aug 1, 2001

[2] *Counting the cost of Slammer*, Robert Lemos, CNET news, Jan 31, 2003

[3] *Cops take a bite, or maybe a nibble, out of cybercrime*, Jon Swartz, USA Today, Sept 2, 2003

duced by programmers to document their intents. According to their report, using their tool allowed them to discover and correct +3,000 buffer overflows in C programs from the Microsoft Vista Windows release.

2004-2007 - More worms based on buffer overflows and memory safety errors in C programs disrupt Internet traffic and industries (Witty worm, Zotob, *etc.*). Meanwhile, software companies and organizations daily send security updates to prevent zero-day exploits, in which a worm based on a known vulnerability infects unpatched software systems.

November 28, 2007 - The SANS Institute issues its annual list of Top 20 Software Vulnerabilities, 13 of which are still related to buffer overflows or memory corruption in C programs.

2008 - An analysis of the first 60 Ubuntu Security Notices, from the leading Linux distribution Ubuntu, shows that 45% of vulnerabilities stem from buffer overflows.

1.1.4 Work in Progress

In 2002, the National Institute of Standards and Technology (NIST) evaluated to $59.5 billion the annual cost of software errors for the U.S. economy, which represented about 0.6 percent of its gross domestic product. Over the years, the human, economical and societal cost of software failures has lead to a steady increase of the combined efforts of researchers, companies and governments to create techniques and tools that help build dependable software. We list here those that contribute to increase the state-of-the-art and the state-of-practice in static safety checking of C programs.

Scientific Conferences Four annual international conferences now focus on verification of software.

CAV - Computer Aided Verification. Since 1989, this conference is the leading international conference on formal analysis methods for software and hardware systems.

VMCAI - Verification, Model Checking and Abstract Interpretation. Since 2000, this conference brings together researchers to combine formal analysis methods for software and hardware systems.

SAFECOMP - Computer Safety, Reliability and Security. Since 1979, this conference is the leading international conference on building dependable systems.

VSTTE - Verified Software: Theories, Tools, Experiments. Initiated in 2005 as a response to Hoare's call, this conference inaugurates in 2008 the Verified Software Initiative, *a fifteen-year, cooperative, international project aimed at the scientific challenges of large-scale software verification.*

Online Repositories Comparison of techniques and tools has been greatly improved with the creation of online repositories describing real errors.

MITRE CVE - Common Vulnerabilities and Exposures. Created in 1999, this is a dictionary of publicly known information security vulnerabilities and exposures (exposures are milder than vulnerabilities). Every publicly known software error gets a unique identifier CVE-year-number (*e.g.*, CVE-2008-3606 for a recent buffer overflow vulnerability in a C program). At the time of this writing, querying “buffer overflow” in this list returns 4,410 matches for a total of 31,972 entries.

MITRE CWE - Common Weakness Enumeration. Created in 2005, this is a formal list of software weakness types created to serve as common language for error reporting and tool evaluation. In this list, buffer overflow errors correspond to a set of entries dominated by CWE-118, *range errors*. *E.g.*, the classic buffer overflow is classified as CWE-120, *unbounded transfer*, and put in relation with related errors, such as improper null termination (CWE-170) or integer overflow (CWE-190).

NIST SAMATE - Software Assurance Metrics And Tool Evaluation. This project created in 2005 offers a taxonomy of software security assurance tools, as well as the SRD, SAMATE Reference Dataset, a list of test cases for software assurance. To this date, it lists 1,762 test cases, 1,224 of which are buffer overflows in C programs.

1.2 C Language Safety Issues

1.2.1 Lack of Precise Semantics

Initially developed by Dennis M. Ritchie at Bell Labs between 1969 and 1973, the C language gets a first informal specification in 1978 with the publication of the book *The C Programming Language* by Kernighan and Ritchie [109]. This definition of C is now referred to as K&R C. In 1989, the American National Standards Institute (ANSI) publishes the first standard for the C language, referred to as ANSI C or C89 [45]. This standard is adopted a year later by the International Organization for Standardization (ISO), therefore it is also known as ISO C or C90 [97]. The current standard was adopted in 2000 by ISO and is known as C99 [98]. Most notably, standardization lead to formally separating the C language from low-level memory manipulations, that must be additionally specified in an Application Binary Interface (ABI), although the separation is not so clear in practice.

C Semantic Blur Despite considerable standardization work, probably more than any other programming language, most C compilers propose a different set of non-standard extensions. Such extensions contribute to the coexistence of many different C dialects, the most famous being GNU C (from GCC compiler suite) and Visual-C (from Visual Studio compiler suite).

More subtly, the same C program written without compiler extensions and compiled with different standard-compliant C compilers can still lead to different results when executed. This originates in the freedom that the C standard leaves to compilers when it comes to low-level details such as data layout, *i.e.*, the sequence of bytes that define a data type. Although a strictly standard-conforming C program should not depend on such low-level decisions, many programs do in fact, which makes them vulnerable to changes of compiler or compiler version. In practice, programmers tend to rely on ABI decisions shared by most (if not all) compilers. This set of decisions forms a *de facto* unauthorized standard. *E.g.*, this is the case for the algorithm deciding the layout of data structures, given values for the size and alignment of base types. To help separate the official standard from common practice, people discussing C programs portability (mostly on the `comp.lang.c` newsgroup) have even informally defined a hypothetical computer architecture, the DeathStation 9000 (DS9K), that behaves as erratically as possible while still conforming to the C standard.

The C standard defines 3 kinds of portability and validity issues. A non-portable or invalid behavior may be:

- *implementation-defined* - Various choices may be valid. A standard-conforming compiler should document which choice it makes. *E.g.*, the number of bits in a byte or character (`char`) is implementation-defined.
- *unspecified* - Various choices may be valid. A standard-conforming compiler is free to choose from a set of behaviors without documenting it. *E.g.*, the order of evaluation of function arguments in a call is unspecified.
- *undefined* - No valid choice exists. A standard-conforming compiler may choose any behavior at all, from stopping the execution to proceeding with execution in an inconsistent state. *E.g.*, dereferencing an invalid pointer is undefined.

Typically, it is considered an error to trigger an undefined behavior or to depend on an unspecified behavior. On the contrary, it is common to rely on specific implementation-defined behaviors, which depend on the compiler and the execution platform. Usually, these decisions are left as user options in tools analyzing C programs.

Due to both (1) the large number of extensions defined by compilers and (2) the large amount of freedom left to compilers by the C standard, formally defining the semantics of C is a challenge. Indeed, no attempt at formally defining the semantics of C has been made by the standards committee.

Proposals of Semantics for C The first detailed formal semantics for C is due to Gurevitch and Huggins [82], in the framework of evolving algebras. Since then, various authors have proposed semantics for C, but most lack a complete treatment of subtle issues like the order of evaluation of expressions between sequence-point, and the subsequent possibility of undefined behavior when the same location is read or written more than once.

Norrish [143] and Papaspyrou [146] both propose a complete treatment of these issues in their respective PhD theses. Papaspyrou manually defines a denotational semantics for C with an implementation in Haskell, while Norrish formalizes the semantics of C in the HOL proof assistant using a structural operational semantics. Although sound, complete and implemented, this last approach necessitates too much human work to mechanically verify a simple BDD package program written in C originally written by Norrish.

In order to build the formally certified C compiler CompCert [23, 22, 125], Leroy *et al.* have completely defined the semantics of a large subset of C called Clight inside the Coq proof assistant. Contrary to Norrish's work, they choose to impose a set of implementation-defined behaviors. *E.g.*, they impose that expressions are evaluated left-to-right.

1.2.2 Lack of Language-Based Safety Mechanisms

Strong Typing and Allocation The C language was designed to allow low-level access to data structures, making it possible to address any individual byte of memory. Although the

C language provides *abstractions* to the programmer in the form of data abstractions (data types) and control abstractions (call graph), its low-level orientation makes it impossible to enforce these abstractions. This allows an attacker to execute arbitrary programs whenever the initial C program executed does not purposely guard against abstraction violations.

Data abstractions are the easiest to violate, and allow compromising control abstractions as well. It comes from the lack of support in the language for the two most important programming abstractions regarding safety: (strong) *typing* and *allocation*.

Due to the lack of (strong) typing in C, one can never guarantee that accessing in the program some data statically typed with type `T` effectively turns at runtime into an access to some data of type `T`. The C standard [98] enumerates many such examples of possible data types violations, the simplest of which is reading uninitialized data. Although it is deemed as a "bad thing" by the standard, it is not enforced by the language, thus making it possible for a program to read some random bit-pattern and interpret it as a value of type `T`, possibly breaking the consistency of the execution state. This fault might be revealed right away, *e.g.*, if the hardware rejects some bit-patterns as invalid values of type `T`, or remain dormant until it triggers a failure.

Due to the lack of support for allocation in C, it remains entirely the programmer's responsibility to access only valid memory, *i.e.*, memory properly allocated and not yet deallocated. This is where failure to take into account all possible cases, *e.g.*, improper inputs from an attacker, leads to vulnerable programs. In particular, there is no language mechanism to know at runtime the size of an array (a.k.a. buffer), which is one of the reasons for the frequency of buffer overflow errors in C programs, where a buffer is accessed beyond its bounds.

The situation only worsens when one considers control abstractions. Usually, during execution of a program, memory chunks for data and control are juxtaposed in computer memory, which allows a data violation to pervert control as well. Since control defines what steps a program is allowed to take, control violations make it possible to execute whatever program lies in the computer memory, with every possible outcome: crash, reboot, information stealing, remote code execution, *etc*.

Memory Safety *Language-based safety* is obtained when a computer language design guarantees that no runtime error can possibly occur while executing a program written in this language, provided the execution environment is conform to its specifications (thus excluding hardware failures, electrostatic glitches and the like). If exceptions are not considered as runtime errors but as a standard error reporting mechanism, languages like JAVA and OCAML can be defined as safe.

Memory safety is the property that (1) only properly allocated data is accessed in the programmer's code, which prevents control violations, and (2) allocated data is accessed only through proper accessors, which forbids unintended modification of neighboring data. Of course, machine code run on a computer does access bytes encoding control information, but those machine code instructions should correspond to code added by a compiler during the translation from source code to machine code, not to source code instructions which should only manipulate program data. Most high-level languages guarantee memory safety (JAVA, OCAML, *etc.*). Although memory safety is not sufficient to completely ensure safety, *i.e.*, absence of runtime errors, it already prevents the most dangerous forms of safety violations. This is why we focus on memory safety in this thesis.

C provides no guarantee regarding safety or memory safety. Since the first detailed analysis of a buffer overflow exploit by Aleph One in 1996 [144], many different ways of exploiting a data violation, in particular buffer overflows, have been described [92] and used in attacks. They differ in the kind of memory corrupted (stack, heap), the amount of data corrupted (from several words to one byte, the so-called off-by-one errors), the vulnerability that makes it possible (buffer overflow, use-after-free, *etc.*) and the possible benefit for the attacker (Denial of Service, remote code execution, *etc.*). The range of possible exploits makes it doubtful that any technique that does not completely enforce memory safety captures them all. This makes memory safety the #1 property of interest to ensure safety of C programs.

Aliasing The term *aliasing* describes a situation in which a data location in memory can be accessed through different symbolic names in the program. In C, aliasing originates in the use of *pointers* to address memory chunks. A pointer is a data value that designates a memory cell. A typed pointer designates a memory chunk, starting at the cell pointed-to by the pointer and whose size is the same as the size of the type pointed-to. Pointers are said to be aliased when the memory chunks they point to overlap, and unaliased otherwise.

Figures 1.1, 1.2 and 1.3 present all possible configurations. The underlying tape represents the computer memory, a large array of individual bytes, represented as cells on this tape. On top of it, rectangular forms represent chunks of memory, which group together contiguous memory bytes. In Figure 1.1, `x` and `y` are the name of pointers which point to non-overlapping memory chunks, thus they are unaliased. In Figure 1.2, `x` and `y` are the name of pointers which point to the same memory chunk, thus they are aliased. In general, memory chunks pointed-to may overlap, as in Figure 1.3, in which pointers `x` and `y` are aliased.

The problem is that there is no language mechanism in C for aliasing, which greatly complicates in general the analysis of C programs, and in particular static safety checking of C programs. Although the programmer is well-aware of possible aliasing in most cases, either because it is syntactically obvious, or because aliasing is used as a programming discipline (see Section 6.1.4 for a survey), it remains implicit in the program. Without any other information, an analysis must assume that any two pointers in the program may alias, due to the lack of strong typing in C. The C standard [98] defines a keyword `restrict` that a programmer may use to indicate absence of aliasing between parameters, but this is not checked by the compiler, which may lead to subtle errors if the function is called in an inappropriate context. Indeed, the danger lies in the possibility that the programmer relies on some locations to be unaliased for correctness, while they may be aliased in reality. Therefore, presence of the `restrict` keyword is usually ignored by aliasing analyses. Despite considerable research effort [88, 178], aliasing remains a major problem when analyzing C programs.

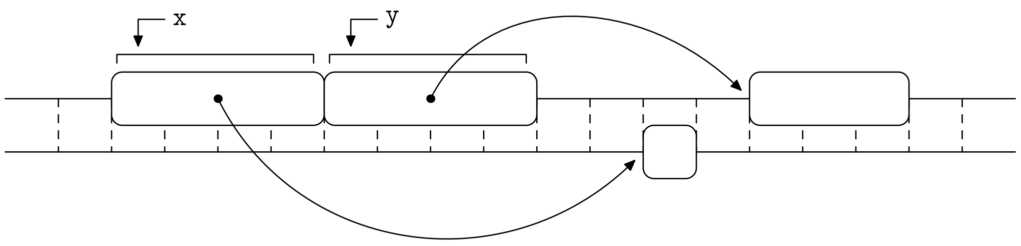

Figure 1.1: Non-Aliasing of pointers

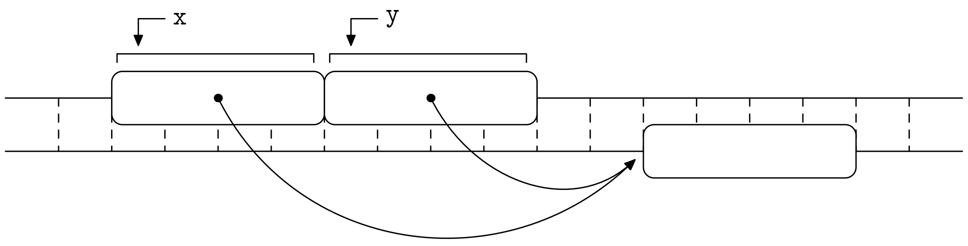

Figure 1.2: Aliasing of pointers

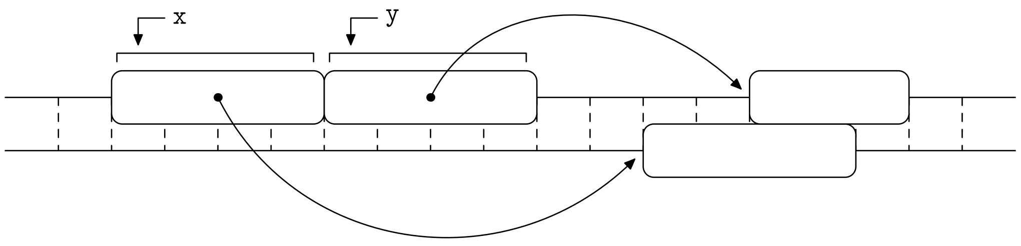

Figure 1.3: Partial aliasing of pointers

1.2.3 Remediation Techniques

Various remediation or mitigation techniques attempt to limit the consequences of memory errors, in particular buffer overflow errors, by detecting them as they occur and preferring to gracefully abort the program and recover in some way rather than proceeding with a possibly malicious execution. These techniques share common limitations:

- *reliance on end-users* - These techniques rely heavily on system administrators to set up a special compilation or link or runtime environment. This asks for a lot of work from many people. As already mentioned in 1.1.3, failure to apply security patches in a timely manner is the main cause of memory errors exploitation.

- *incompleteness* - Except for the use of safe C compiler, these techniques cannot target all possible memory errors, possibly leaving some space for an attacker to sneak in, either through a return-to-libc attack where control is redirected to a resident library function, a double free attack, *etc*.

- *late-stage* - These techniques occur too late in the development process to prevent errors from occurring, they can only limit the damage incurred by stopping the execution.

Despite these limitations, remediation techniques are today the best way to protect against exploitation of memory errors and buffer overflows. By preventing remote code execution, if not the error from occurring in the first place, they effectively limit the propagation of worms.

Executable Space Protection Often, a buffer overflow exploit places a *shellcode* (a piece of assembly code) on the stack before executing it. But the stack is a part of the computer memory that is normally used to store data, not code. By making the stack non-executable, one can prevent exploits of this kind to execute their shellcode.

This technique appeared as early as 1961 in the Burroughs 5000, not in relation to buffer overflows at that time. It can be supported either by hardware or software. Since 2000, it has been adopted in many operating systems distributions to protect against buffer overflow attacks. Although it completely protects against code injection, this technique does not protect against other exploits based on resident code, like return-into-libc attacks.

Safe Runtime Library Many buffer overflow attacks are based on vulnerable uses of the standard C library, more specifically functions manipulating strings (sequences of characters). By replacing the code of these libraries with code that performs additional checks to ensure memory safety, one can protect against these specific buffer overflow exploits.

This technique was first presented in Libsafe [9] in 2000. It successfully prevents stack overflows that redirect the flow of control, but not those that overwrite stack data, notably pointers. Moreover, it cannot prevent overflows in functions other than the vulnerable library functions explicitly treated.

Address Space Layout Randomization Attacks that redirect the flow of control need to locate the address in memory of some target code, whether a shellcode on the stack or resident code on the host machine, *e.g.*, library code. Address space layout randomization makes this phase much harder by randomly distributing these addresses.

This technique was initiated in the PaX patch for the Linux kernel in 2000. To date, it is the most effective way to prevent remote code execution, but it does not offer a total protection.

Stack-Smashing Protection Stack-based buffer overflows are the easiest ones to exploit, and thus the most common ones. They usually redirect control by overwriting the return address of the current function being executed, which is stored on the stack. Two techniques prevent such overwriting. By placing a "canary", a group of specific bytes, just before the return address, one can detect with a high probability whether these bytes have been overwritten. The other technique is to place return addresses on a separate stack. Contrary to previous ones, these techniques require that the source code of the program is available.

These techniques were pioneered by StackGuard and StackShield, two patches for the GCC compiler suite, in 1997. Although useful, they can still be bypassed, as shown by an article published in Phrack Magazine [29] in January 2000. In 2000, ProPolice, another patch for GCC, proposed an enhanced stack-smashing protection. Since 2002, Microsoft Visual C proposes the same functionality as StackGuard, when option /GS is set. It can also be bypassed, as described in an article from Cigital researchers [150] in 2002.

Pointer Protection Even when the address in memory of some code can be located, it must be stored as pointer data before it can be executed. Pointer protection encrypts the value of pointers, so that it is very hard to guess the encrypted value that corresponds to a given address.

This technique was first described by Cowan *et al.* in PointGuard [53] in 2003, and latter implemented in Windows XP SP2 and Windows Server 2003 SP1. There exists various ways to bypass this protection.

Deep Packet Inspection Before a shellcode gets executed, it must be sent to the target executable in the form of an innocuous string. Deep packet inspection attempts to discover the executable nature of the crafted string by looking at its content.

This technique first appeared in 2003 as a combination of the functionalities of an intrusion detection system and a stateful firewall. Unfortunately, it is not very effective at preventing buffer overflow attacks, even those based on code injection, as there are many ways to encode a shellcode into a string.

Safe C Compilers Various compilers for C programs instrument the generated machine code so that memory errors are detected and reported right away.

Safe C [5] performs a C to C translation based on a safe pointer representation that ensures memory safety at runtime. It inserts a runtime check before every memory access through a pointer. Authors report execution time overheads from 130% to 540% and memory footprint overheads up to 100%, on the benchmark tested.

CCured [138] statically recovers types of pointers from C source code in order to perform instrumentation only where needed. It relies on garbage collection for memory reclamation, rather than the unsafe hand-written calls to `free`. Authors report execution time overheads up to 250%, on the benchmark tested.

This approach suffers from a slowdown in execution time and an increase in memory footprint that may be unacceptable for some applications. Furthermore, it requires all libraries to be compiled with a safe compiler to be effective.

1.2.4 Better C Initiatives

Due to the inherent difficulties in analyzing C programs and ensuring safety of the generated executables, many projects have tried to improve on the C language, with as few modifications as possible. All these initiatives have in common to improve on memory safety, when not guaranteeing it.

Safe C Libraries Since many buffer overflows occur in standard library functions that do not check bounds before writing in buffers, it is considered good practice to avoid these functions altogether. This includes the (in)famous `gets`, `scanf` and `strcpy` functions. Many alternatives have been proposed to replace these unsafe functions, *e.g.*, `strcpy_s`, `strlcpy` and `strncpy` for replacing `strcpy`, the function that copies strings.

Safe string libraries go beyond the standard C string library, by defining an abstract data type to be used instead of plain strings, and safe operations on this abstract data type, including bound checking. There exists various such libraries: The Better String Library, the C String Library (SafeStr), Vstr, Erwin vectors of characters, CERT managed string library, *etc*.

Adopting safe libraries defines in fact a domain-specific language whose basic constructs are those library functions coded in plain C. Still, failure to use these library functions correctly can result in buffer overflows, and it is always possible that these functions themselves contain vulnerabilities.

Safe C Subsets Most software companies impose that their developers follow a strict set of coding guidelines in C, which allows them to reduce the likelihood of memory errors. At the most extreme, these coding guidelines may even define a memory-safe subset of C, by constraining the form of loops allowed, by forbidding problematic features such as casts and unions, *etc*.

Such a set of coding guidelines defined by the MISRA (Motor Industry Software Reliability Association) for the automotive industry is known under the name MISRA-C [133]. It strongly constrains the kind of programs that can be written, by excluding the most problematic language features in C: dynamic allocation, unions, casts between pointer types, *etc*. Although widely adopted in some industries, MISRA-C has received much criticism for its lack of a rigorous definition. A similar Pascal-like subset of C named C0 [116] is the base language of project Verisoft, which aims at a completely verified software and hardware platform. Neither MISRA-C nor C0 is a memory-safe subset of C.

Safe C Dialects Some projects have tried to completely replace C with a different but equivalent language, that could be used instead of C for systems programming. These approaches obviously suffer from not being applicable to legacy C programs.

Cyclone [104] is a clone of C with language mechanisms for strong typing, allocation and aliasing. It restricts in particular pointer casts and (discriminated) unions to maintain type-safety. BitC [160] takes the more extreme position of completely departing from the syntax and semantics of C, to adhere instead to SML/Scheme syntax and semantics, with access to low-level memory as in C. D [16] is a strongly typed systems programming language whose goal is to correct bad design decisions in C and C++, while keeping the same power and efficiency.

Annotated C Another approach exemplified by Deputy [184] consists in adding annotations to the C language to express additional specifications, like the expected size of a buffer. Then, a dedicated compiler is responsible for checking that these annotations are indeed respected. The same kind of annotations is used in SAL (Standard Annotation Language) at Microsoft, to prevent memory errors in large legacy C programs [84]. These annotation languages are presented as a set of type qualifiers or declaration specifiers to facilitate adoption by C programmers, which also makes them too simple to handle every possible situation. Although these annotation languages cannot in general express all the properties that are required to completely ensure safety, they have achieved far greater levels of dependability than was previously known.

1.3 Techniques and Tools

Many techniques and tools have been developed for analyzing C programs at the source level, more than for any other language. We detail here those techniques and tools that target *static safety checking for existing C programs*. This excludes most techniques presented so far:

- C *programs* - This excludes using all variants of C presented in Section 1.2.4, with the exception of annotated C proposals.
- *existing programs* - This excludes all software engineering techniques that tend to improve code quality and understanding in order to lower the rate of errors: code reviews, coding guidelines, defensive programming, software development methodologies (eXtreme programming, CMMI, *etc.*).
- *static safety checking* - This excludes all remediation techniques that detect and recover from errors at runtime presented in Section 1.2.3.

The main difficulty in analyzing a program in any language is the infinite number of possible situations to consider, a.k.a. the infinite state space. The techniques that target static safety checking for existing C programs (and tools based on those) generally follow one of three paradigms, or a combination thereof, that try to solve the infinite state space problem in a different way:

- **Enumeration** techniques consider each possible situation in turn.
- **Abstraction** techniques build a finite abstraction of the system to analyze, that covers all possible real situations, and analyze it instead.
- **Deduction** techniques transform the problem into logical formulas and apply logical rules of reasoning to decide their truth value.

1.3.1 Enumeration Techniques

Enumeration techniques take the straightforward approach of considering every possible situation in turn, while trying to benefit from the problem symmetries to treat more than one situation at once. Eventually, in case this enumeration is still infinite, these techniques provide a sense of exhaustivity by relying on some notion of coverage of the state space (*e.g.*, structural, functional) or by bounding the problem parameters to resort to a finite enumeration.

Testing [107] is currently the most common and useful technique to check the safety of industrial programs. In its simplest form, it consists in running the program on the selected input, and checking the output generated to be the one expected. There are many branches in testing, roughly grouped into white-box testing, in which the source code is available, black-box testing, in which only the executable is available, and regression testing, in which the differences between successive versions of a program are tested. In general, a program compiled for testing may behave differently as the same program compiled for production, as it may contain additional debugging code or instrumented code whose purpose is to apply extra checks during test execution. Testing is non-exhaustive by nature, and relies instead on notions of coverage.

Simulation is similar to testing, with the program being simulated by software instead of being executed on hardware. Like testing, it is seldom exhaustive.

Model checking [42] is an exhaustive simulation, usually based on efficient data structures. Explicit model checking simply considers each possible situation, while bounded model checking considers situations up to some predefined bounds depending on the problem. Software model checking is always based on bounded model checking techniques, not exhaustive ones, as the state space is usually infinite. This means a program is only checked up to some bounds in its parameter sizes. Model checking techniques, including bounded model checking, suffer from the *state-explosion problem*, where the analysis time or space requirements may be too large to be practical. Software model checking techniques first build a boolean model of the program, where data and control are both transformed into boolean variables, and the program itself is translated into a set of formulas over these boolean variables describing the possible transitions between states. VeriSoft [75] is the first software model checker for C programs.

Symbolic model checking applies the simulation techniques of model checking to sets of states, usually represented using Binary Decision Diagrams (BDDs) to provide a canonical representation for data and efficient operations on those. Software model checkers for C programs based on symbolic model checking include Cadence Incisive, CBMC [40] and F-Soft [99].

1.3.2 Abstraction Techniques

Abstraction techniques rely on the existence of suitable finite abstractions of the system to analyze. Their main task is to find an abstraction of the program that is precise enough that all the concrete executions it represents are safe. There is here an obvious tradeoff between the precision of the abstraction and the cost of building it. The difficulty lies in the definition of a Precise enough Static Analysis or PSA [127] for the safety problem at hand.

Abstract interpretation [51] is a theory of program abstractions. As such, all abstraction techniques can be seen as instances of abstract interpretation. Various *abstract domains* have been defined, which correspond to different directions of abstractions: positivity, range, linear relations, *etc.* Given an abstract domain, abstract interpretation simulates an execution of the program where variables take values in the abstract domain, thus simulating all executions at once. By a suitable choice of abstract domain and operations on this abstract domain, this simulation eventually reaches a finite over-approximation of the program, on which it is possible to check safety.

Abstract interpretation is the core technology in all currently available industrial program verifiers: The Mathworks PolySpace, Astrée [21], C Global Surveyor [174], CodeHawk [3], Clousot [66], Penjili [96]. Since these tools conservatively report every possible safety violation, they may suffer from a large number of *false positives*, *i.e.*, possible errors reported by the tool which do not correspond to real errors. Some avoid this pitfall by targeting only specific C programs, *e.g.*, command control C programs in Astrée or flight mission software in C Global Surveyor and Penjili. Others rely on dynamic instrumentation to catch errors at runtime where the tool cannot prove statically that operations are safe. This is the case for CodeHawk and Clousot. Otherwise, these tools may still be used as bug finders rather than program verifiers, by focusing only on the most dangerous possible errors reported. This is the case in practice for the leading such tool, The Mathworks PolySpace. Bug finder Sparrow [183] even claims to perform a sound abstract interpretation before it selects a fraction of the generated warnings to show to the user.

Static analysis designates the application of abstractions to analyze programs. This encompasses abstract interpretation, as well as other abstraction techniques that are not formalized in the framework of abstract interpretation. To overcome the natural imprecisions faced when scaling to large programs, some static analyses are built on unsound abstractions. These analyses may forget some concrete executions in order to improve scaling and precision. In general, static analysis may be sound or not depending on the choice of abstraction.

Most pattern-based bug finders currently available are based on static analysis, from their publicly available descriptions: Microsoft Prefix/Prefast, Fortify SCA, Grammatech CodeSonar, Klocwork Insight, Coverity Prevent. Although little information on their algorithms is public, these tools seem to deliberately abstract data in a possibly unsound way to improve scaling and precision. This unsoundness allows them to catch bugs more efficiently.

1.3.3 Deduction Techniques

Deduction techniques rely on the generation of logical formulas whose validity is equivalent to the safety of the initial program. These **verification conditions** (VC), a.k.a. proof obligations (PO), can be proved in a theorem prover, a tool which applies logical rules of reasoning to decide the truth value of mathematical formulas. This can be done either automatically inside an automatic prover, or with assistance from a user inside a proof assistant.

Deductive verification is the application of deduction to program verification. It requires that the safety property is expressed as assertions in the source program. Then, Hoare logics [89] allow one to systematically transform assertions into verification conditions, at the cost of annotating the program with intermediate assertions at each program point. Dijkstra's calculus [59], either by *weakest preconditions* or *strongest postconditions*, allows one to reduce the annotation burden on the user side, only requiring annotations at a few program points: precondition at function beginning, postcondition at function end, loop invariant at each loop beginning. Still, this approach suffers from not being completely automatic, requiring at least that annotations are inserted in the program, at worst that a user interacts with the proving system.

A few program verification tools implement this approach for C programs, for different logics. Frama-C [73] and VCC [43] are based on classical first-order logic. Thus, they can rely on many existing theorem provers to discharge verification conditions. *E.g.*, Frama-C allows one to call many different automatic provers and proof assistants, while VCC allows to choose between automatic prover Z3 [135] or proof assistant HOL [31]. Other tools rely on different logics, for which they provide a dedicated automatic prover or proof assistant, or both, *e.g.*, first-order dynamic logic for Key-C [137] and separation logic for SLAyer [182].

Automatic provers systematically explore the state space of logical formulas to return a "yes/no/don't know" answer to the satisfiability or validity of logical formulas. Automatic provers generally handle simpler formulas than proof assistants, and do not require expert users if used completely automatically inside other tools. Many such tools are available: Alt-Ergo [46], CVC3 [12], Simplify [57], Yices [63], Z3 [135], *etc*.

Proof assistants provide a mechanical support for manually proving hard theorems, usually requiring expert users. There are many such tools available [177], of which Coq [19], HOL [142] , Isabelle/HOL [141] and PVS [145] are well-known examples. When automatic provers fail to prove a valid verification condition, proof assistants provide a way to complete the proof.

1.3.4 Combination Thereof

Enumeration, abstraction and deduction techniques can be used in combination. It is the main goal of the VMCAI international annual conference since 2000.

Predicate abstraction [78] is the best example of a powerful combination of abstraction and deduction. It is a variant of abstract interpretation where the abstract domain used is the domain of conjunctions over a set of predefined predicates mentioning program variables. Operations on this logical abstract domain rely on querying an automatic theorem prover for validity of formulas.

CEGAR [41] (Counter-Example Guided Abstraction Refinement) is a refinement of predicate abstraction that combines all three paradigms. Given an initial set of predicates, an abstraction of the program is built using deduction techniques. Model checking techniques are used to enumerate possible abstract executions. Whenever an error is reached in an abstract execution, either it corresponds to a real error, or it is a spurious error that is removed by refining the predicates upon which the abstraction relies. Software model checkers for C programs based on a CEGAR approach include SLAM [8], BLAST [20] and MAGIC [33].

Invariant generation, *i.e.*, the generation of valid formulas at specific program points, can be used to improve the results of most verification techniques. Therefore, techniques that generate such invariants, such as abstract interpretation techniques, can be used as a first step in combination with other techniques. *E.g.*, invariants can be used as function preconditions, postconditions and loop invariants in deductive verification [71].

1.4 Statement of Purpose

Since recently, there is a shared diagnosis amongst software-intensive industries that, as software systems keep growing in size, the current reliance on compliance with a qualified process and validation by testing will not be sufficient to assess the dependability of software systems. The recent multiplication of bug finders based on static analysis (Fortify SCA in 2004, Coverity Prevent in 2004, Microsoft Prefix/Prefast in 2005, Klocwork Insight in 2005, Grammatech CodeSonar in 2006) is only a first step in this mind-shift.

It remains to develop techniques and tools that can **guarantee** safety of C programs, which cannot be based on unsound abstractions like the aforementioned bug finders. The current trend for industrial use is to focus on sound abstraction techniques. In 2003, the tool Astrée [21] showed it was possible to reach completely automatic static safety checking for C programs of 100,000+ lines of code, although for a very restricted range of C programs, namely generated command control C programs. Other works insist that program verification requires the use of *-sensitive analyses [83], which is not easily obtained with abstract interpretation alone.

Techniques for program verification should make it possible to discriminate between the same program with or without a specific error. In 2004, the benchmark of Zitser *et al.* [185] showed that none of the five modern static analysis tools tested was better than a random choice when discriminating between an unsafe program and its patched version. In 2006, Hackett *et al.* presented a tool based on SAL lightweight annotations [84] that succeeded in discriminating most Zitser's test cases. The next step should be to get the same discriminating results without the need for annotations and with sound techniques guaranteeing safety.

In order to gain industrial acceptance, these sound techniques should scale to millions of lines of code and apply to existing C programs as originally written. Then, industrial interest may vary depending on whether the tool is:

- **automatic**, not needing expert users to give usable (*i.e.*, precise) results;
- **modular**, not requiring all the source code of the application and its libraries;
- **tunable**, making it possible to improve results given user input.

Deductive verification techniques naturally comply with the last two requirements, by being inherently modular and allowing users to finely tune the properties proved through logical annotations. Weakest preconditions generate the most precise verification conditions. Function-level modularity allows scaling easily. This thesis contributes to applying these techniques to existing C programs in an automated way.

1.5 Summary of Contributions

This thesis presents techniques for static safety checking of industrial C programs by deductive verification. These techniques are both automatic and modular: they neither require human intervention nor the complete source code.

More precisely, we propose an answer to each of the three main problems one must face when trying to apply deductive verification to industrial C programs [93]:

- *annotation generation* - Deductive verification without the ability to automatically generate the necessary logical annotations may only be undertaken for very few projects, due to the cost of manually adding annotations.

  We present a technique to generate logical annotations based on abstract interpretation and weakest preconditions. In particular, it generates precise sufficient function preconditions, which has not been shown before.

- *modular memory separation* - Fine grain memory separation is the only way to generate verification conditions that can be verified by automatic provers. This is especially true in a context where annotations are automatically generated.

  We present an alias control technique based on Talpin's alias analysis, a context sensitive variant of Steensgaard's type-based alias analysis. It is the first instance of an alias analysis that generates necessary function preconditions of correctness, thus relying on deductive verification to discharge these preconditions. This technique allows one to express separation properties clearly in verification conditions, in a way that is optimal for automatic provers.

- *support for unions and casts* - Industrial C programs do use the low-level memory management capabilities of the C language, most notably unions and casts of pointers. Failure to support these features in previous tools has been recognized as the major barrier to adoption of these tools in an industrial context.

We present a mixed typed and byte-level memory model that allows one to handle unions and casts in deductive verification, while keeping as much as possible the benefits of the typed memory model. It relies on the modular memory separation technique mentioned above.

These techniques have been implemented in Frama-C [73], an open-source platform for modular analysis of C programs, and the Why Platform [69], an open-source platform for deductive verification of programs. Figure 1.4 sketches the relations between both platforms. Frama-C is built on CIL, an open-source front-end for the main C dialects, to which it adds (1) support for logical annotations expressed in the ANSI C Specification Language [14] (ACSL) and (2) a plugin-based framework to facilitate the integration of various analyses. The Why Platform is a multi-language multi-prover verification tool, that generates verification conditions for both automatic provers and proof assistants. Inside Frama-C, the JESSIE plugin translates programs from the internal Frama-C representation to an input representation for the Why Platform. Although Frama-C and the Why Platform are distinct softwares, the Why Platform is also distributed inside Frama-C for an easy integration. Notice that Frama-C replaces the tool Caduceus [68] previously developed in the Why Platform. For completeness, we also sketch the relation to Krakatoa [129], a tool of the Why Platform for deductive verification of JAVA programs.

Figure 1.4 also sketches the flow of translation from a C program to the generated verification conditions. The source C program may be decorated with logical annotations in ACSL. The program and its annotations are first translated to an annotated variant of CIL [139] (C Intermediate Language), and then to JESSIE, the intermediate language considered in all the analyses described in this thesis. In fact, the JESSIE language we present in this thesis is not exactly the JESSIE language that serves as interface between Frama-C and the Why Platform. The latter is not at all minimal, in order to facilitate the interfacing and to delay common transformations after the translation to JESSIE, thus avoiding duplicate work. We present instead an internal intermediate language used in the Jessie2Why tool, which translates the JESSIE program into a WHY program. Finally, the VC generator inside the Why Platform generates the corresponding verification conditions, which can be discharged in a number of automatic provers and proof assistants.

Using Frama-C, we applied the techniques described in this thesis to:

- check the safety of existing string libraries: an implementation of the C standard string library in MINIX 3, a secure open-source operating system, and CERT Managed String Library, a secure string library coded in a defensive programming style;

- discriminate between unsafe and patched versions of open-source programs with vulnerabilities: we check such pairs of programs in both Verisec Suite and Zitser's benchmark.

1.6 Organization of This Thesis

This thesis is divided into three parts.

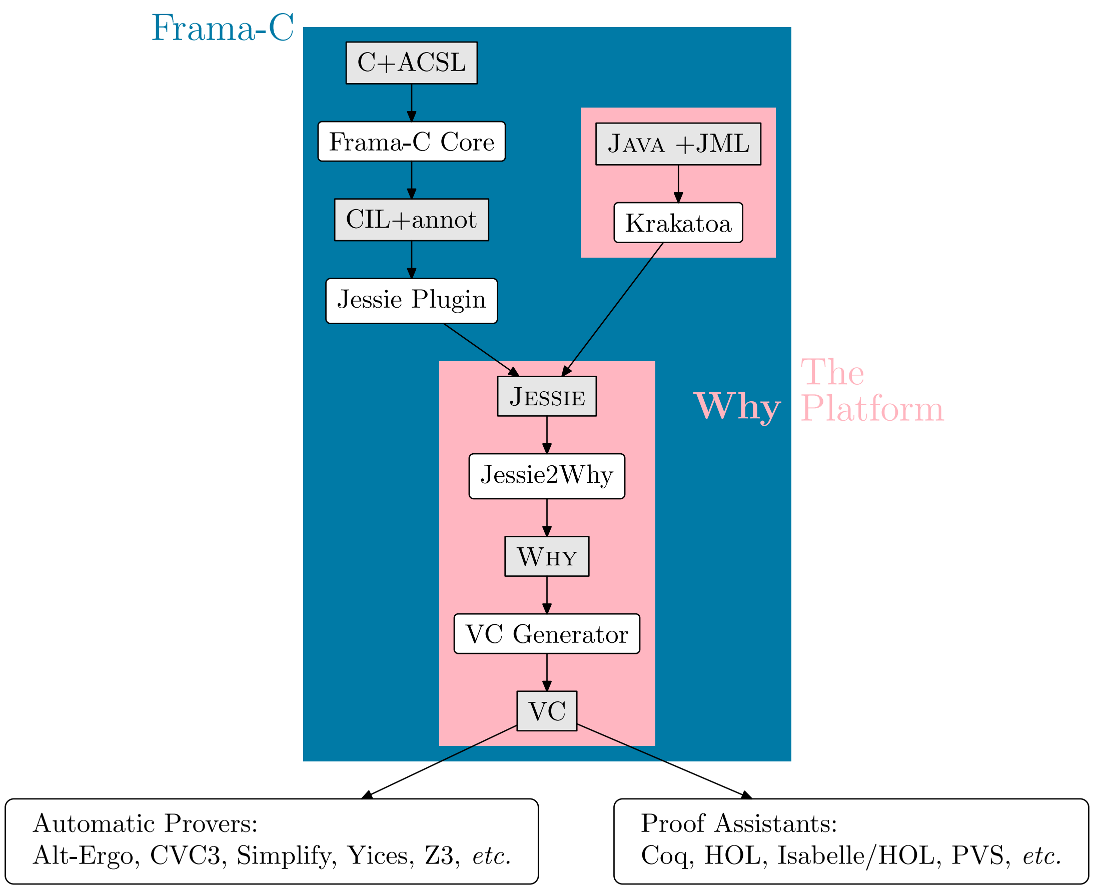

Figure 1.4: Frama-C and the Why Platform

In Part I, we present the language JESSIE, the translation of C programs into JESSIE programs, and techniques for static safety checking of JESSIE programs.

Chapter 2 presents the syntax and semantics of the intermediate language for program verification JESSIE, as well as the translation of C programs into JESSIE programs.

Chapter 3 describes how to check automatically and modularly that annotated JESSIE integer programs are safe and respect their annotations, using abstract interpretation and deductive verification.

Chapter 4 describes how to check automatically and modularly that annotated JESSIE pointer programs are safe and respect their annotations, using abstract interpretation and deductive verification.

In Part II, we present restrictions on the kind of input JESSIE programs considered that allow the design of improved analysis techniques. Chapters 5, 6 and 7 each presents such a restriction and the associated analyses, in increasing order of generality.

Chapter 5 restricts our analysis to alias-free type-safe JESSIE programs. In this context, we present a technique to generate automatically and modularly annotations, based on a combination of abstract interpretation and deductive verification, so that programs can be checked safe as in Chapter 4.

Chapter 6 restricts our analysis to type-safe JESSIE programs. In this context, we present a technique to control aliasing automatically and modularly, based on a context sensitive variant of Steensgaard's type-based alias analysis. We also describe how to adapt the generation of annotations described in Chapter 5 to this new context, so that unannotated type-safe JESSIE programs can be checked safe as in Chapter 4.

Chapter 7 considers the full set of JESSIE programs translated from C programs with unions and pointer casts. We present a mixed typed and byte-level memory model that allows us to handle unions and casts, while minimizing the impact of this untyped translation on analyses presented previously. We describe how to further adapt the generation of annotations described in Chapter 5 to this new context, so that unannotated JESSIE programs can be checked safe as in Chapter 4.

In Part III, we present the results of experiments on real C programs, both existing C string libraries and benchmarks of vulnerabilities, before we finally conclude.

Part I

Integer and Memory Safety Checking

Chapter 2

Intermediate Language Definition

Contents

In this chapter, we present the intermediate language for program verification JESSIE and the translation of C programs into JESSIE programs.

JESSIE is a simple typed imperative language, with precisely defined semantics, which allows for an easy exposure and understanding of analyses. Section 2.1 discusses the rationale behind JESSIE. Section 2.2 formally defines JESSIE semantics w.r.t. a byte-level block memory model. It provides a firm ground on which to define safety for JESSIE programs.

Section 2.3 describes the translation of C programs into JESSIE programs, so that results of analyses on JESSIE programs are well understood in terms of the corresponding C programs.

Section 2.4 presents the first-order logic annotation language included in JESSIE. It allows specifying JESSIE programs and communicating results between analyses. ACSL annotations for C programs are translated into JESSIE annotations for the corresponding JESSIE programs.

2.1 JESSIE Rationale

As described in Section 1.2, C is a complex language, both in terms of syntax and semantics. It makes it challenging to describe and understand any analysis directly performed at the level of C source programs. Instead, we translate C programs into a simpler language, JESSIE. All the analyses we describe in this thesis apply to JESSIE programs.

Intermediate Languages for C Analysis There exists many intermediate languages for the analysis of C programs. They vary in expressiveness, level of abstraction and simplicity, depending on the kind of analysis they were designed for. They usually borrow features from C and assembly languages.

GENERIC [163] is an abstract syntax representation of C that takes care of parsing and linking issues. It remains at the same level of abstraction as C source: control structures are preferred over control-flow graph; expression trees are preferred over three-address code expressions; typed variables are preferred over pseudo-registers or registers; stack frames remain implicit. It is the first main intermediate language used in the GCC compiler suite.

SIMPLE [87] is an intermediate language in the McCAT compiler that facilitates alias and dependency analyses. It is simpler than GENERIC, with expression trees translated into three-address code expressions.

GIMPLE [163] is a clone of SIMPLE where control structures are translated into control-flow graphs. It is the second main intermediate language used in the GCC compiler suite.

CIL [139] (for C Intermediate Language) sits in between GENERIC and GIMPLE: control structures are kept and side-effects are hoisted out of expression trees by introducing instructions. It is originally the intermediate language used in safe compiler CCured. While GENERIC is best suited for source code syntactic analyses and GIMPLE for optimization, CIL is ideally suited for source code semantic analyses. As such, it is the frontend of many analyses for C programs, ranging from type-safe compilation to symbolic evaluation and slicing.

MSIL [39, 131] (for Microsoft Intermediate Language) is a unique case of object-oriented assembly language in human-readable form. It is closer to assembly than GIMPLE, with local variables being translated into stack offsets, while still maintaining types. It is also known as CIL for Common Intermediate Language. It is indeed the first common intermediate language to which all languages in the Common Language Infrastructure or the .Net Framework translate, in the Visual Studio compiler suite.

Newspeak [96] is at the same level as MSIL, while maintaining some control structures and expression trees. These design decisions notably facilitate source code analyses. It is the intermediate language used in static analyzer Penjili.

The CompCert project [22] for building a verified compiler presents a chain of intermediate languages for compilation of C programs, from Clight, a large subset of C, to plain assembly, through Cminor, which is similar to MSIL and Newspeak.

C0 [116] is a Pascal-like subset of C, similar to MISRA-C [133]. It excludes the features of C that make full verification problematic: pointer arithmetic, unions, pointer casts. It is the intermediate language used in the Verisoft project for pervasive formal verification of computer systems.

Simpl [158] is a very generic Sequential IMperative Programming Language. It offers high-level constructs like closures, pointers to procedures, dynamic method invocation, all above an abstract state that can be instantiated differently depending on the language analyzed. In his PhD thesis, Schirmer presents a two-fold embedding of a programming language in HOL theorem prover through Simpl. On the one hand, Simpl statements are deeply embedded inside HOL, which allows one to formally verify correctness of the analysis on statements. On the other hand, the abstract state is shallowly embedded inside HOL, for an easier application to program analysis of many different languages. Simpl is used as target language in the C0 compiler of the Verisoft project.

On a different track, BoogiePL [11] and WHY are generic intermediate languages for deductive verification. None of them is specifically tailored for the analysis of C, although BoogiePL is the intermediate language used in C analysis tools HAVOC [47] and VCC [43]. BoogiePL and WHY both explicitate memory as a collection of heap variables, which is necessary for deductive verification, but cumbersome for most static analyses.

Behavioral Interface Specification Languages JESSIE is a direct successor of the intermediate languages used in Caduceus [68] and Krakatoa [129], which aim at deductive verification of C and JAVA languages, in the framework of the Why Platform [69]. It inherits from these predecessors part of its memory model and annotation language. These two annotation languages are themselves strongly inspired from JML [115], the JAVA Modeling Language, whose purpose is to annotate JAVA programs with logic formulas for testing and static analysis.

JESSIE was developed in parallel with ACSL [14], the ANSI C Specification Language, with which it shares most of its constructs. The memory model constructs are essentially specific to the analysis of C programs, partly inspired by the annotation language of Caduceus, while most other logic constructs can already be found in JML.

Comparison with JESSIE JESSIE follows most intermediate languages for C analysis in targeting all of C, at the exclusion of embedded assembly code, *i.e.*, the ability to embed instructions in an assembly language inside a C program. Like Newspeak and Clight, JESSIE relies on a byte-level block memory model. Like CIL and Clight, JESSIE remains as much as possible at the level of types, while allowing byte-level operations. As presented in Figure 1.4, JESSIE is a target of translation from CIL, from which it inherits a collection of implementation-defined decisions w.r.t. the target architecture and the real C compiler. *E.g.*, we consider by default that a byte, the lowest addressable entity, is 8 bits, which is the case in almost all existing architectures.

JESSIE is original in at least two directions. First, the JESSIE memory model and data types are notably simple while staying at the level of structured types, which allows more easily to generate annotations. Secondly, JESSIE combines operational and logical features. Its operational part consists in *statements* which describe the flow of control and *instructions* which perform operations on data, including memory updates. Its logical part is described

range-def	::=	`range` *id* = *integer* `..` *integer*	integer range def
fields	::=	`{` (*type id* `:` *integer*)* `}`	list of fields
struct-def	::=	`struct` *id* = *fields*	structure def
type	::=	`boolean` $\mid$ `integer` $\mid$ `real`	mathematical types
	$\mid$	`unit`	void type
	$\mid$	*id*	integer range type
	$\mid$	*id* `[` *integer*$^?$ `..` *integer*$^?$ `]`	pointer to structure type

Figure 2.1: Grammar of JESSIE types

through first-order logic *propositions*, which annotate statements and functions. Statements, instructions and propositions are all built upon side-effect free *terms*. Contrary to Simpl, JESSIE does not offer an embedding inside a theorem prover, which means that translations to and from JESSIE must be trusted.

In the following, we illustrate analyses on JESSIE programs with their results on the corresponding C programs annotated with ACSL.

2.2 JESSIE Syntax and Operational Semantics

In this section, we present the syntax, typing and semantics of the core operational language in JESSIE. Section 2.4 and Chapters 4, 6 and 7 will extend this core language in order to better support static safety checking of JESSIE programs.

2.2.1 Abstract Syntax

JESSIE is an intermediate language, meaning programmers are not expected to create JESSIE source programs. As such, we are only interested in its abstract syntax, not its concrete one. We present this abstract syntax in the following. *E.g.*, we will not use separators (semicolon) between instructions or statements in a sequence. However, to make it more intuitive and easier to follow, we present it in a concrete form, adding parentheses whenever needed to disambiguate between concrete forms. We will also use this convenient concrete form to show examples of C to JESSIE translations.

Types Figure 2.1 presents the abstract syntax of JESSIE types. JESSIE types are much simpler than C types, which accounts a lot for the gain in analyzing a JESSIE program instead of a C program. A type is either a base type or a user-defined type. Base types are mathematical integers, booleans and real numbers plus unit (the void type). User-defined types duplicate in JESSIE the bound restrictions of operational C types: integer ranges have a minimal and maximal value, while pointers may be limited in the range of indices they address, with an optional minimal and maximal index. An integer range type in JESSIE is very similar to an integer type in ADA (see also [74] for a similar proposition for C, that is part of the Secure Coding initiative by CERT).

The complex interplay of pointers, arrays and structures in C is replaced in JESSIE with much simpler design decisions. The only type of aggregate in JESSIE is the array of structures, which can only be accessed by pointer, and the only type of pointer in JESSIE is the pointer to an array of structures. In pointer type $\texttt{S}[min..max]$, $\texttt{S}$ is the name of a structure while min and max, when present, are the indices of the bounds of the underlying array of structures. *E.g.*, a pointer to a single structure can be typed $\texttt{S}[0..0]$ ($\texttt{S}[0]$ for short) because 0 is both the minimal and maximal index at which the pointer can be accessed. Absence of bounds, like in $\texttt{S}[..]$, does not mean the underlying array is infinite, but rather that its bounds are not specified in the pointer type. We will say a pointer has type $\texttt{S}[min..max]$ to indicate its bounds, or more simply type `S` when bounds do not matter.

A structure is defined as a named record of typed fields. C fields and C implicit padding both translate to JESSIE fields. Every field is given a size in bits, similarly to what is done for bitfields in C. The size in bits of field `m` is denoted $\mathit{bitsizeof}(\texttt{m})$, and its offset in bits is $\mathit{bitoffsetof}(\texttt{m})$, which is the sum of the sizes in bits of the fields that precede it in the structure. The size of a structure is defined as the sum of the sizes of its fields. It should be a multiple of 8, so that a structure size can be expressed in bytes too. The size in bytes of structure `S` is denoted $\mathit{sizeof}(\texttt{S})$. For a term `x` of pointer type `S`, $\mathit{sizeof}(\texttt{x})$ is the same as $\mathit{sizeof}(\texttt{S})$. There is no notion of alignment in JESSIE, which is not needed for the layout of fields in memory as padding is made explicit.

An *embedded field* `m` is a field of structure `S` of pointer type $\texttt{T}[min..max]$ where both minimal and maximal bounds are known. Such a field with pointer type can always be accessed safely between its bounds, which seems to raise a contradiction. Indeed, between the program point where some pointer `x` of type `S[..]` is allocated and the program point where `x.m` is set, `x.m` may not have a valid pointer value. In fact, such fields are implicitly allocated when the enclosing structure is allocated, and implicitly deallocated when the enclosing structure is deallocated. They may not be assigned to a different value, nor can they be deallocated independently. Finally, they should also have an offset that can be expressed in bytes, so that we can define $\mathit{offsetof}(\texttt{m})$ as $\mathit{bitoffsetof}(\texttt{m})/8$. JESSIE embedded fields behave like C fields of a structure or array type.

At this point, it may seem difficult to translate C complex types into such a reduced set of types. In fact, it is possible, given the semantics of JESSIE defined in the following, which will become clear when presenting the translation from C to JESSIE.

Terms Figure 2.2 presents the abstract syntax of JESSIE terms, to which C side-effect free expressions translate. The usual arithmetic and comparison operators operate on mathematical integers and real numbers. In particular, JESSIE operators do not overflow. Hence, JESSIE operations do not suffer from the wrap-up behavior typical of operations on machine integers in C, when operations that overflow the integer capacity return a correct result modulo the capacity. Operations on pointers are distinguished:

- $\oplus$ denotes pointer arithmetic, taking as operands a pointer and an integer, and returning a pointer;

bin-op	::=	`+` | `-` | `*` | `/`	integer/real arithmetic
	|	`%`	integer modulo
	|	$\leq$ | $\geq$ | $<$ | $>$	integer/real comparison
	|	$\oplus$	pointer arithmetic
	|	$\ominus$	pointer difference
	|	$\otimes$ | $\oslash$	pointer comparison
	|	$\equiv$ | $\not\equiv$	(dis)equality test
	|	$\wedge$ | $\vee$	boolean operations
unary-op	::=	`-`	unary minus
	|	$\neg$	boolean negation
location	::=	*id*	variable
	|	`(` *location* $\oplus$ *term* `)` `.` *id*	memory location
term	::=	`void`	
	|	`true` | `false`	boolean constant
	|	`null`	null pointer
	|	*integer*	integer literal
	|	*real*	real literal
	|	*location*	
	|	*unary-op term*	unary operation
	|	*term bin-op term*	binary operation
	|	*term* `?` *term* `:` *term*	choice operation
	|	*term* $\triangleright$ *type*	cast

Figure 2.2: Grammar of JESSIE terms

instr	::=	*location* `:=` *term*	assignment
	|	*id* `:=` `new` *id* `[` *term* `]`	allocation
	|	`free` *term*	deallocation
	|	*id* `:=` *id* `(` (*term* (`,` *term*)*)$^?$ `)`	function call
stat	::=	*instr*	
	|	*stat stat*	sequence
	|	`if` *term* `then` *stat* `else` *stat*	conditional
	|	`loop` *stat*	infinite loop
	|	`return` *term*	function return
	|	`throw` *id*	exception raise
	|	`try` *stat* `catch` *id* *stat*	exception block

Figure 2.3: Grammar of JESSIE statements

- ⊖ denotes pointer difference, taking as operands two pointers of the same type, and returning an integer;
- ⧀ and ⧁ denote pointer strict comparison, taking as operands two pointers of the same type, and returning a boolean.

Equality and disequality for a pair of booleans, integers, real numbers, identical integer ranges or pointers are denoted respectively $\equiv$ and $\not\equiv$. The usual boolean operators operate on booleans.

A *location* denotes either a variable or a memory location. Like a left-value (lvalue for short) in C, it is the syntactic category of the left operand in an assignment. A variable is either local to a function or global to the program. A memory location is always reached through a succession of pointer arithmetic and field access. Field access expects a location operand of pointer type and a field operand, and it denotes the field location pointed to. Location `x.m` is only a convenient shorthand for `(x⊕0).m` that we will use in examples, not so much in reasoning about memory locations. In location `(x⊕i).m`, location `x` should have type pointer and term `i` type integer.

A *cast* takes a term operand and returns a term of a different type. The operand and result terms are either both of a numerical type (integer, real number or integer range), or both of a pointer type. In the first case, casting reinterprets the operand value. In the second case, casting reinterprets the memory chunk pointed-to.

Finally, a term is one of: a constant, an integer or real number literal, a location, a unary, binary or ternary (choice) operation or a cast.

Statements Figure 2.3 presents the abstract syntax of JESSIE statements. JESSIE is mostly a structured language, with conditionals and loops, but it also relies intimately on intraprocedural exceptions. *E.g.*, raising an exception is the only way to escape a loop. Therefore, exceptional control flow is as important as the normal one. These exceptions may not escape a function body, which can be checked statically, hence the name intraprocedural

$$
\begin{array}{rcll}
\textit{var-def} & ::= & \textit{type id} & \\
\textit{parameters} & ::= & (\textit{type id}\ (\texttt{,}\ \textit{type id})^*)^? & \\
\textit{fun-def} & ::= & \textit{type id}\ \texttt{(}\ \textit{parameters}\ \texttt{)}\ =\ \textit{var-def}^*\ \textit{stat} & \\
\textit{glob} & ::= & \textit{range-def} & \text{range integer} \\
 & \mid & \textit{struct-def} & \text{structure} \\
 & \mid & \textit{var-def} & \text{global variable} \\
 & \mid & \textit{fun-def} & \text{function}
\end{array}
$$

Figure 2.4: Grammar of JESSIE globals

$$
\frac{}{\text{void} : \text{unit}}\ \text{CONST-VOID} \qquad \frac{}{\{\text{true}, \text{false}\} : \text{boolean}}\ \text{CONST-BOOL}
$$

$$
\frac{}{\text{null} : T[..]}\ \text{CONST-NULL}\ \ T\ \textit{any structure}
$$

$$
\frac{}{\textit{integer} : \text{integer}}\ \text{CONST-INT} \qquad \frac{}{\textit{real} : \text{real}}\ \text{CONST-REAL}
$$

Figure 2.5: Typing of JESSIE constants

exceptions. As in CIL, we draw a distinction between instructions and statements. Instructions are basic statements that, upon performing their task, always transfer control to the next statement in sequence. Statements are either instructions, or more complex statements which structure the control flow. For simplicity, allocation and function call always store their result in a temporary variable, whose scope begins at that instruction.

Globals Figure 2.4 presents the abstract syntax of JESSIE global entities, except type definitions (integer range and structure) already described in Figure 2.1. A global variable is simply defined by its type and name. A function is defined by its return type, name, parameters, local variables and body statement. A special `main` function, with `unit` return type and no parameters, is the entry point of the program. There are no implicit initializations of global variables like in C.

2.2.2 Typing Rules

Typing rules restrict the kind of JESSIE programs that can be written. We present typing rules for all terms, in the form of typing judgments $t : \tau$, in which t is a term and τ a type.

Figure 2.5 presents the typing of JESSIE constants. According to rule CONST-NULL, constant `null` can have any unbounded pointer type.

$$\frac{t : \text{integer}}{-t : \text{integer}}\ \textsc{unop-int} \qquad \frac{t : \text{boolean}}{\neg t : \text{boolean}}\ \textsc{unop-bool}$$

$$\frac{t_1 : \text{integer} \quad t_2 : \text{integer}}{t_1\{+, -, \times, /, \%\}t_2 : \text{integer}}\ \textsc{arith-int} \qquad \frac{t_1 : \text{real} \quad t_2 : \text{real}}{t_1\{+, -, \times, /\}t_2 : \text{real}}\ \textsc{arith-real}$$

$$\frac{t_1 : \text{integer} \quad t_2 : \text{integer}}{t_1\{\leq, \geq, <, >\}t_2 : \text{boolean}}\ \textsc{comp-int} \qquad \frac{t_1 : \text{real} \quad t_2 : \text{real}}{t_1\{\leq, \geq, <, >\}t_2 : \text{boolean}}\ \textsc{comp-real}$$

$$\frac{t_1 : \text{boolean} \quad t_2 : \text{boolean}}{t_1\{\wedge, \vee\}t_2 : \text{boolean}}\ \textsc{oper-bool}$$

Figure 2.6: Typing of JESSIE base type operations

$$\frac{t_1 : T[..] \quad t_2 : \text{integer}}{t_1 \oplus t_2 : T[..]}\ \textsc{shift} \qquad \frac{t_1 : T[..] \quad t_2 : T[..]}{t_1 \ominus t_2 : \text{integer}}\ \textsc{subptr}$$

$$\frac{t_1 : T[..] \quad t_2 : T[..]}{t_1\{\circledless, \circledgtr\}t_2 : \text{boolean}}\ \textsc{comp-ptr}$$

Figure 2.7: Typing of JESSIE pointer operations

$$\frac{t : \text{real}}{t \triangleright \text{integer} : \text{integer}}\ \textsc{real-cast} \qquad \frac{t : \text{integer}}{t : \text{real}}\ \textsc{real-promotion}$$

$$\frac{t : \text{integer}}{t \triangleright E : E}\ \textsc{range-cast} \qquad \frac{t : E \quad E\ \mathit{is\ an\ integer\ range\ type}}{t : \text{integer}}\ \textsc{range-promotion}$$

$$\frac{t : T[..]}{t \triangleright S[min^?..max^?] : S[min^?..max^?]}\ \textsc{ptr-cast}$$

$$\frac{t : T[min^?..max^?]}{t : T[..]}\ \textsc{ptr-promotion} \qquad \frac{t : \tau}{t \triangleright \tau : \tau}\ \textsc{ident-cast}$$

Figure 2.8: Typing of JESSIE casts and promotions

$$\frac{}{x : \mathbf{Type}(x)}\ \textsc{var} \qquad \frac{t_1 : T[..] \quad t_2 : \text{integer} \quad m\ \mathit{is\ a\ field\ of}\ T}{(t_1 \oplus t_2).m : \mathit{typeof}(m)}\ \textsc{field}$$

Figure 2.9: Typing of JESSIE locations

Figure 2.6 presents the typing of JESSIE operations on base types: integer, real numbers and booleans. Figure 2.7 presents the typing of JESSIE operations on pointers. They all expect pointer subterms to be of unbounded pointer types.

Figure 2.8 presents the typing of JESSIE allowed casts and implicit promotions. Rules REAL-PROMOTION, RANGE-PROMOTION and PTR-PROMOTION define implicit promotions of terms: a term of type *integer* can always be promoted to type *real*; a term of integer range type can always be promoted to type *integer*; a pointer of bounded pointer type can always be promoted to the corresponding unbounded pointer type.

Figure 2.9 presents the typing of JESSIE locations. The type for a global or local variable is given by type environment **Type**. The type of a field access is given by the type of the field. Figure 2.10 presents the typing of JESSIE remaining terms. Equality and disequality apply to any pair of terms of the same type.

Figure 2.11 presents the typing of JESSIE instructions, and Figure 2.12 presents the

$$\frac{t_1 : \tau \quad t_2 : \tau}{t_1 \equiv t_2 : \text{boolean}}\ \textsc{eq} \qquad \frac{t_1 : \tau \quad t_2 : \tau}{t_1 \not\equiv t_2 : \text{boolean}}\ \textsc{neq}$$

$$\frac{t_1 : \text{boolean} \quad t_2 : \tau \quad t_3 : \tau}{(t_1\ ?\ t_2 : t_3) : \tau}\ \textsc{choice}$$

Figure 2.10: Typing of JESSIE terms

$$\frac{t : \textbf{Type}(x)}{\{x := t\}\ \textit{well-typed}}\ \textsc{assign-var}$$

$$\frac{l : S[..] \quad t_1 : \text{integer} \quad m\ \textit{field of}\ S \quad t_2 : \textit{typeof}(m)}{\{(l \oplus t_1).m := t_2\}\ \textit{well-typed}}\ \textsc{assign-field}$$

$$\frac{x : S[..] \quad t : \text{integer}}{\{x := \text{new } S[t]\}\ \textit{well-typed}}\ \textsc{new} \qquad \frac{t : S[..]}{\{\text{free } t\}\ \textit{well-typed}}\ \textsc{free}$$

$$\frac{x : \textit{typeof}(\textit{result}_f) \quad t_i : \textbf{Type}(\textit{param}_i)}{\{x := f(\overrightarrow{t_i})\}\ \textit{well-typed}}\ \textsc{call}$$

Figure 2.11: Typing of JESSIE instructions

$$\frac{\{s_1\}\ \textit{well-typed} \quad \{s_2\}\ \textit{well-typed}}{\{s_1\ s_2\}\ \textit{well-typed}}\ \textsc{seq}$$

$$\frac{t : \text{boolean} \quad \{s_1\}\ \textit{well-typed} \quad \{s_2\}\ \textit{well-typed}}{\{\text{if } t \text{ then } s_1 \text{ else } s_2\}\ \textit{well-typed}}\ \textsc{if-true}$$

$$\frac{\{s\}\ \textit{well-typed}}{\{\text{loop } s\}\ \textit{well-typed}}\ \textsc{loop-normal}$$

$$\frac{t : \textbf{Type}(\textit{result})}{\{\text{return } t\}\ \textit{well-typed}}\ \textsc{return} \qquad \frac{}{\{\text{throw } X\}\ \textit{well-typed}}\ \textsc{throw}$$

$$\frac{\{s_1\}\ \textit{well-typed} \quad \{s_2\}\ \textit{well-typed}}{\{\text{try } s_1 \text{ catch } X\ s_2\}\ \textit{well-typed}}\ \textsc{try-normal}$$

Figure 2.12: Typing of JESSIE statements

typing of JESSIE statements.

2.2.3 Execution Model

JESSIE is meant to be an analyzable language, not an executable one. What is analyzed is not a possible run of a JESSIE executable on a real machine, but an interpretation of the JESSIE program in an imaginary machine, deliberately ignoring implementation issues. In the following, we detail the model of the imaginary machine we rely on, by describing its memory model and state model. Data fits into three categories in JESSIE:

- *global data*: it corresponds to the content of global variables;
- *local data*: it corresponds to the content of local variables and function parameters;
- *heap*: it corresponds to memory, *i.e.*, dynamically allocated data.

The state model describes the result of accessing this data. The memory model describes how memory is organized, so that valid and invalid memory accesses can be distinguished.

Memory Model In a real computer, memory is often simply described by its size N, meaning every byte in memory can be described by its address, an integer between 0 and $N-1$. Then, a pointer is simply an address, and a block of memory is a finite set of contiguous addresses. This defines a simple *byte-level memory model*, illustrated in Figure 2.13, where a block of memory is typically represented as a pair (a, s) of an address and a size, and a pointer is an address. This is roughly the memory model used in VCC [43]. The level of abstraction provided by the C program is partly lost in this model of memory. *E.g.*, it is possible to access beyond the bounds of the allocated block pointed-to, provided the memory accessed is also allocated, which is forbidden by the C standard.

A usual abstraction of memory that avoids this problem is the *block memory model*, illustrated in Figure 2.14, where memory is made up of disjoint blocks of memory, without any reference to addresses. A block of memory is typically represented as a pair (l, s) of a label and a size, where a label completely identifies a block. A pointer is a pair (l, i) of a block label and an index into the block. Byte-size memory cells pointed to by pointers (l, i) where $i \in [0..s\text{-}1]$ define the part of memory that uniquely belong to the block. Most intermediate languages and tools for analyzing C programs rely on a block memory model. With this memory model, it is not possible to analyze common but non-standard conforming C programs that compare pointers pointing into different memory blocks. In particular, it is not possible to analyze any implementation of the C standard library function `memmove`.

Notice that using the address of a block as label in a mixed byte-level and block memory model, as illustrated in Figure 2.15, is not a correct solution to overcome the previously mentioned limitations, because of deallocation and reallocation of memory. Indeed, an invalid pointer to a deallocated block (in black on the Figure) may have the same value as a valid pointer to a newly allocated block (in white on the Figure), if these blocks happen to start at the same address. Then, it is not possible to distinguish between a valid and an invalid pointer. Thus, it would be possible to prove the safety of program `perverse`, although it clearly accesses pointer `x` after its memory has been deallocated.

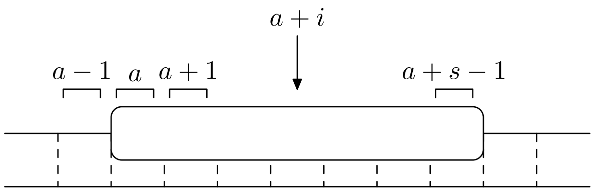

Figure 2.13: Byte-level memory model

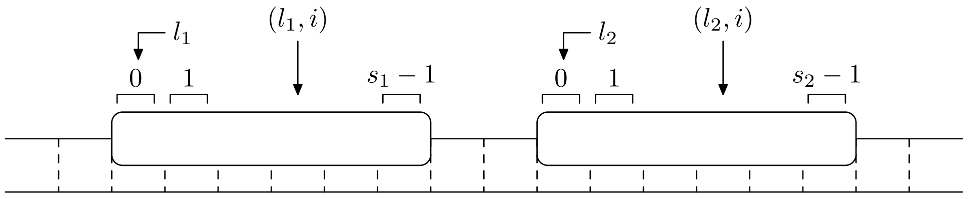

Figure 2.14: Block memory model

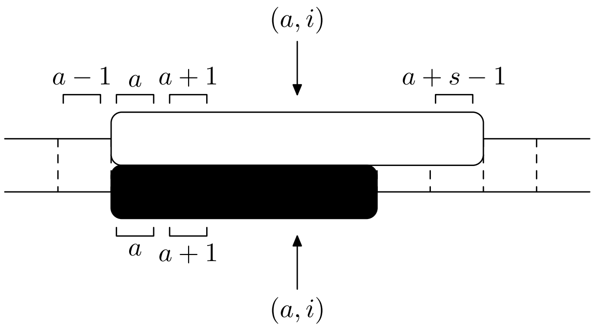

Figure 2.15: Incorrect byte-level block memory model

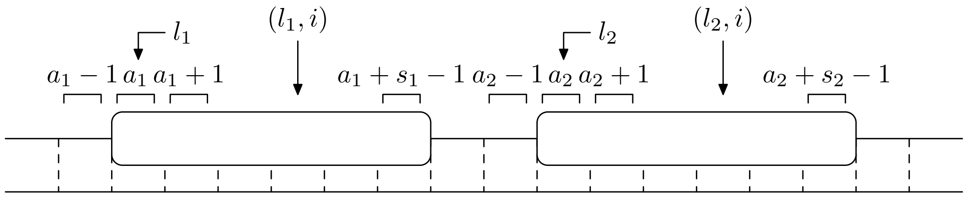

Figure 2.16: JESSIE byte-level block memory model

```
void perverse() {
 int *x = (int*) malloc(sizeof(int));
 free(x);
 int *y = (int*) malloc(sizeof(int));
 if (x == y) {
   *x = 0;
 }
}
```

JESSIE relies on a *byte-level block memory model*, illustrated in Figure 2.16, that correctly merges the byte-level and the block approaches in order to avoid previously mentioned limitations. A block of memory is represented as a tuple (l, a, s) of a label, an address and a size in bytes. A label completely identifies a block, meaning that no two blocks can share the same label, even after one block has been deallocated. Address a and size s can be seen as functions of label l. An address only identifies an allocated block, meaning that no two allocated blocks can share the same address, but they may share it with deallocated blocks. More precisely, allocated blocks are completely separated, so that the range of addresses from a to $a + s - 1$ is uniquely associated to allocated block (l, a, s). A pointer is a tuple (l, i) of a block label and a byte offset into the block. With this memory model, it is possible to both forbid accesses beyond the allocated block pointed-to and to allow comparison of pointers into different blocks. This is in fact just a slight modification of the block memory model with addresses.

Finally, the JESSIE memory model is unbounded, as if the computer memory was infinite. Although it could be included in the model, the finite bound on the memory size is not needed for memory safety checking as defined in Section 1.2.2.

Memory Updates and Accesses Although memory blocks could be typed at creation (**new** `S[n]` must return a pointer to an array of `S` structures of size `n`), casting between pointer types may reinterpret a memory chunk in a different type. To account for the coexistence of different views of a same memory chunk, memory is considered to be untyped, so that accesses to the same chunk through different pointer types write or read from the same underlying memory block. Hence, **new** `S[n]` returns a pointer to a block of *sizeof*(`S`)$\times$`n` bytes, interpreted as an array of `n` elements of type `S`.

Pointer arithmetic allows moving a pointer forward or backward along a memory block, but it does not allow jumping from one allocated block to a neighboring one. Indeed, reading and writing through a pointer of type S is valid only if the individual bytes for structure S accessed are within the bounds of the original underlying memory block.

The *offset* of pointer `p` pointing into memory block a is the difference in bytes between the byte pointed-to by `p` and the beginning of memory block a. Pointer difference takes two pointer operands of the same type `S`, pointing into the same memory block. It returns the difference of their offsets, divided by *sizeof*(`S`). Notice that this difference is not necessarily a multiple of *sizeof*(`S`), which means that `p` $\ominus$ `q` can be null without `p` and `q` being equal. Pointer comparison $\circledless$ returns *true* if the difference of addresses is strictly negative, *false* otherwise.

Finally, memory blocks can be deallocated through calls to `free`, after which they should not be accessed anymore.

State Model A state is a snapshot of every significant piece of information at some point during execution of a JESSIE program. In particular, it maps every piece of data to its value. By extension, it maps every term to a value that depends on its type. Terms of mathematical type evaluate to the corresponding mathematical values: integers, real numbers, booleans (and `void` for type `unit`). Terms of integer range type naturally take value in integers, although in a restricted range of integers defined by their type. Terms of pointer type evaluate to tuples (l, a, i) of a memory block label, an address for the block and an integer offset, as defined in the JESSIE memory model. In particular, null pointers evaluate to $(null, 0, 0)$, where $null$ is a reserved label. Here, we describe the underlying state model which we rely on to describe JESSIE semantics, not the actual model we use to prove JESSIE programs. A state is essentially a tuple of mappings from variables and memory chunks to values:

- **Glob** maps global variables to their value. The value of global variable x is denoted **Glob**$(\mathtt{x})$. Assigning a value v to x results in a modified mapping denoted **Glob**$[\mathtt{x} \mapsto v]$, where x has value v while all other global variables retain their value from **Glob**.

- **Env** maps local variables to their value. The value of local variable x is denoted **Env**$(\mathtt{x})$ while assigning value v to x results in a modified mapping **Env**$[\mathtt{x} \mapsto v]$.

- **Heap** maps memory chunks to bit-vectors. **Heap**(ba, bn) denotes the bit-vector stored in contiguous bits from bit address ba to bit address $ba + bn - 1$, where the bit address of a memory byte is 8 times its address (a byte being 8 bits). Changing the bit-vector stored between these bit addresses to bv results in a modified mapping **Heap**$[(ba, bn) \mapsto bv]$. This notation really means that all bits between ba and $ba + bn - 1$ are updated to the corresponding value from bv, so that further reads of any of these bits returns the new value, whether it is through the same chunk of bits or not.

- **Alloc** maps memory block labels to their allocated size. A positive size means the corresponding memory block is allocated, while a non-positive size means the memory block has been deallocated. Notice that **Alloc** stores all memory block labels previously created, so that no two blocks get the same label.

We will denote the empty mapping as ϵ, so that the empty state is the tuple $\epsilon, \epsilon, \epsilon, \epsilon$.

Bitwise Representation of Types We assume the existence of a family of functions $\mathit{of\text{-}bits}_{\mathtt{m}}$ indexed by structure fields, such that for a given field m, $\mathit{of\text{-}bits}_{\mathtt{m}}$ takes a sequence of bits of length $\mathit{bitsizeof}(\mathtt{m})$ and returns the value of type $\mathit{typeof}(\mathtt{m})$ represented by this bit-vector. Likewise, we assume there exists a family of functions $\mathit{to\text{-}bits}_{\mathtt{m}}$ such that for a given field m, $\mathit{to\text{-}bits}_{\mathtt{m}}$ takes a value v of type $\mathit{typeof}(\mathtt{m})$ and returns a bit-vector of length $\mathit{bitsizeof}(\mathtt{m})$ that encodes value v. We are not interested in their precise definition for defining the semantics of JESSIE, we only rely on their existence. For every field m, function $\mathit{of\text{-}bits}_{\mathtt{m}}$ should be the left inverse of function $\mathit{to\text{-}bits}_{\mathtt{m}}$, meaning they should satisfy the equation

$$\mathit{of\text{-}bits}_{\mathtt{m}}(\mathit{to\text{-}bits}_{\mathtt{m}}(v)) \equiv v \tag{2.1}$$

for every value v of type $\mathit{typeof}(\mathtt{m})$.

$$\frac{}{[\![c]\!] = \overline{c}}\ \textsc{const} \qquad \frac{[\![t]\!] = v}{[\![\odot\, t]\!] = \overline{\odot} v}\ \textsc{unop} \qquad \frac{[\![t_1]\!] = v_1 \quad [\![t_2]\!] = v_2}{[\![t_1 \odot t_2]\!] = v_1 \overline{\odot} v_2}\ \textsc{binop}$$

$$\frac{[\![t_1]\!] = \mathit{true} \quad [\![t_2]\!] = v}{[\![t_1\ ?\ t_2 : t_3]\!] = v}\ \textsc{choice-true} \qquad \frac{[\![t_1]\!] = \mathit{false} \quad [\![t_3]\!] = v}{[\![t_1\ ?\ t_2 : t_3]\!] = v}\ \textsc{choice-false}$$

$$\frac{}{[\![\mathit{null}]\!] = (\mathit{null}, 0, 0)}\ \textsc{null} \qquad \frac{t_1 : S[..] \quad [\![t_1]\!] = (l, a, i) \quad [\![t_2]\!] = j}{[\![t_1 \oplus t_2]\!] = (l, a, i + j \times \mathit{sizeof}(S))}\ \textsc{shift}$$

$$\frac{t_1 : S[..] \quad t_2 : S[..] \quad [\![t_1]\!] = (l, a, i) \quad [\![t_2]\!] = (l, a, j)}{[\![t_1 \ominus t_2]\!] = (i - j)/\mathit{sizeof}(S)}\ \textsc{subptr}$$

$$\frac{[\![t_1]\!] = (l_1, a_1, i) \quad [\![t_2]\!] = (l_2, a_2, j)}{[\![t_1 \odot t_2]\!] = a_1 + i\ \overline{\odot}\ a_2 + j}\ \textsc{compar-ptr} \quad \odot \in \{⧀, ⧁, \equiv, \not\equiv\}$$

Figure 2.17: Evaluation of JESSIE terms - constants and operations

As an example, let us consider the specific case of a field `m` of signed integral type on n bytes in C, when the target machine integers are represented in two's complement notation, and highest bytes are stored at lowest addresses (*i.e.*, it is a big-endian architecture). It translates in JESSIE to a field `m` of integer range type with minimal value -2^{n-1} and maximal value $2^{n-1} - 1$, of size $8 \times n$ bits. In this case, function $\mathit{to\text{-}bits}_{\mathtt{m}}$ can be defined as the n-bit two's complement representation of its argument, and $\mathit{of\text{-}bits}_{\mathtt{m}}$ as its inverse function. By definition, these functions respect Formula 2.1.

2.2.4 Operational Semantics

We describe here correct executions of well-typed JESSIE programs, in the style of big-step operational semantics [105, 106], also called natural semantics.

Terms Figures 2.17, 2.18 and 2.19 define an evaluation function $[\![.]\!]$ for JESSIE terms. Concrete counterparts of abstract constants and operators are denoted with an overline, *e.g.*, operator $\overline{\odot}$ is the concrete counterpart of abstract operator $\odot$. Rules CONST, UNOP and BINOP deal with base type constants and operations on base type operands, including equality and disequality of base type operands. Rules SHIFT, SUBPTR and COMPAR-PTR deal with operations on pointers, namely pointer arithmetic, pointer difference and pointer comparison, which also treats pointer equality and pointer disequality. In rule COMPAR-PTR, the concrete counterpart $\overline{\odot}$ of a pointer comparison is the corresponding integer comparison. Rules GLOB-VAR and LOC-VAR deal with (global or local) variable evaluation.

The semantics of field access depends on whether the field is embedded or not, which is described respectively by rules EMBED-FIELD and FIELD. There is no indirection involved

$$\frac{}{\llbracket x \rrbracket = \textbf{Glob}(x)}\;\textsc{glob-var} \qquad \frac{}{\llbracket x \rrbracket = \textbf{Env}(x)}\;\textsc{loc-var}$$

$$\frac{\llbracket t_1 \rrbracket = (l,a,i) \quad \llbracket t_2 \rrbracket = j \quad m\ \mathit{is\ embedded}}{\llbracket (t_1 \oplus t_2).m \rrbracket = (l, a, i + j \times \mathit{sizeof}(S) + \mathit{offsetof}(m))}\;\textsc{embed-field}$$

$$\frac{\begin{array}{c} t_1 : S[..] \quad \llbracket t_1 \rrbracket = (l,a,i) \quad \llbracket t_2 \rrbracket = j \quad m\ \mathit{not\ embedded} \\ 0 \le i + j \times \mathit{sizeof}(S) \quad i + (j+1) \times \mathit{sizeof}(S) \le \textbf{Alloc}(l) \end{array}}{\begin{array}{l} \llbracket (t_1 \oplus t_2).m \rrbracket = \mathit{of\text{-}bits}_m(\\ \quad \textbf{Heap}((a + i + j \times \mathit{sizeof}(S)) \times 8 + \mathit{bitoffsetof}(m), \mathit{bitsizeof}(m))) \end{array}}\;\textsc{field}$$

Figure 2.18: Evaluation of JESSIE terms - variables and field access

$$\frac{\llbracket t \rrbracket = f \quad i = \mathit{truncate}(f)}{\llbracket t \rhd \mathit{integer} \rrbracket = i}\;\textsc{from-real} \qquad \frac{\llbracket t \rrbracket = i}{\llbracket t \rhd \mathit{real} \rrbracket = i}\;\textsc{to-real}$$

$$\frac{\llbracket t \rrbracket = i}{\llbracket t \rhd \mathit{integer} \rrbracket = i}\;\textsc{from-range} \qquad \frac{\llbracket t \rrbracket = i \quad \mathit{min}_E \le i \le \mathit{max}_E}{\llbracket t \rhd E \rrbracket = i}\;\textsc{to-range}$$

$$\frac{\llbracket t \rrbracket = (l,a,i)}{\llbracket t \rhd S[..] \rrbracket = (l,a,i)}\;\textsc{ptr-cast} \qquad \frac{\begin{array}{c} \llbracket t \rrbracket = (l,a,i) \\ 0 \le i + \mathit{min} \times \mathit{sizeof}(S) \end{array}}{\llbracket t \rhd S[\mathit{min}..] \rrbracket = (l,a,i)}\;\textsc{low-ptr-cast}$$

$$\frac{\begin{array}{c} \llbracket t \rrbracket = (l,a,i) \\ i + (\mathit{max}+1) \times \mathit{sizeof}(S) \le \textbf{Alloc}(l) \end{array}}{\llbracket t \rhd S[..\mathit{max}] \rrbracket = (l,a,i)}\;\textsc{up-ptr-cast}$$

$$\frac{\begin{array}{c} \llbracket t \rrbracket = (l,a,i) \quad 0 \le i + \mathit{min} \times \mathit{sizeof}(S) \\ i + (\mathit{max}+1) \times \mathit{sizeof}(S) \le \textbf{Alloc}(l) \end{array}}{\llbracket t \rhd S[\mathit{min}..\mathit{max}] \rrbracket = (l,a,i)}\;\textsc{bound-ptr-cast}$$

Figure 2.19: Evaluation of JESSIE terms - casts

when accessing an embedded field. Instead, accessing an embedded field amounts to pointer arithmetic, returning a pointer into the same memory block as its subterm. Accessing a regular field is only possible if the memory chunk for the complete underlying structure completely fits in an allocated memory block. It returns the interpretation of the bit-vector stored in field m as a value of the type of m. Notice access validity does not depend on the type of the pointer being accessed. *E.g.*, a pointer of type $S[0..1]$ could be accessed at index 2 provided the underlying memory chunk is large enough. Bounds in pointer types are minimal requirements on the underlying memory chunk that can be used to easily prove the validity of many memory accesses.

The semantics of casts depends on the type of cast considered. Rules FROM-REAL and TO-REAL describe respectively casts from `real` to `integer` and the opposite way. Casting from `real` to `integer` is interpreted as truncation and the opposite cast as an injection. Rules FROM-RANGE and TO-RANGE describe respectively casts from integer range to `integer` and the opposite way. The first one is an injection while the semantics of the second one depends on the integer model chosen. In the *bounded integer model* presented here, it is the identity, provided the value cast fits in the destination type. In the *modulo integer model*, casting integer i returns $min_E + (i - min_E)\ mod\ (max_E - min_E + 1)$, which corresponds to the wrap-up behavior of machine integers. Pointer cast is the identity, provided the destination type completely fits into allocated memory.

Instructions The big-step semantics for instruction s is described by judgment

$$\{s\} \vdash \textbf{Glob}_1, \textbf{Env}_1, \textbf{Heap}_1, \textbf{Alloc}_1 \Rightarrow \textbf{Glob}_2, \textbf{Env}_2, \textbf{Heap}_2, \textbf{Alloc}_2,$$

which can be abbreviated in

$$\{s\} \vdash \textbf{<S>}_1 \Rightarrow \textbf{<S>}_2,$$

with the meaning that executing instruction s from a state $\textbf{<S>}_1$ leads to a state $\textbf{<S>}_2$. In general, only part of the state is modified, in which case we only mention the modified part in the corresponding judgment, like in

$$\{s\} \vdash \textbf{Env}_1 \Rightarrow \textbf{Env}_2.$$

Figure 2.20 presents the semantics of JESSIE instructions. Notice that, by definition of $\textit{to-bits}_{\mathtt{m}}$ in Section 2.2.3, $\textit{to-bits}_{\mathtt{m}}(v)$ is a bit-vector of size $\textit{bitsizeof}(\mathtt{m})$, as expected in rule ASSIGN-FIELD. In rule NEW, the condition that all the bytes in the memory block returned were not previously allocated is a necessary and sufficient condition for the correctness of the concrete allocation algorithm. Although we do not formalize it here, it can be expressed as a modification of the allocation table that extends it, so that previously allocated pointers cannot point to the newly allocated memory. There is no need to further specify the allocation algorithm beyond this condition to define the semantics of JESSIE programs.

Statements The big-step semantics of statement s is very similar to the one for instructions, except statements also have an outcome: Normal if the statement terminates normally,

$$\frac{[\![t]\!] = v}{\{x := t\} \vdash \textbf{Glob} \Rightarrow \textbf{Glob}[x \mapsto v]}\ \textsc{Assign-Glob}$$

$$\frac{[\![t]\!] = v}{\{x := t\} \vdash \textbf{Env} \Rightarrow \textbf{Env}[x \mapsto v]}\ \textsc{Assign-Loc}$$

$$\frac{t_1 : S[..] \quad [\![t_1]\!] = (l, a, i) \quad [\![t_2]\!] = v \quad 0 \leq i \quad i + \mathit{sizeof}(S) \leq \textbf{Alloc}(l)}{\begin{array}{l}\{t_1.m := t_2\} \vdash \textbf{Heap} \Rightarrow \\ \quad \textbf{Heap}[((a + i) \times 8 + \mathit{bitoffsetof}(m), \mathit{bitsizeof}(m)) \mapsto \mathit{to\text{-}bits}_m(v)]\end{array}}\ \textsc{Assign-Field}$$

$$\frac{\begin{array}{c}[\![t]\!] = n \quad 0 \leq n \quad l \notin \mathit{dom}(\textbf{Alloc}) \\ \forall\, i.\ 0 \leq i < n \times \mathit{sizeof}(S) \rightarrow a + i \ \mathit{not\ allocated}\end{array}}{\begin{array}{l}\{x := \text{new } S[t]\} \vdash \textbf{Env}, \textbf{Alloc} \Rightarrow \\ \quad \textbf{Env}[x \mapsto (l, a, 0)], \textbf{Alloc}[l \mapsto n \times \mathit{sizeof}(S)]\end{array}}\ \textsc{New}$$

$$\frac{[\![t]\!] = (l, a, 0) \quad 0 \leq \textbf{Alloc}(l)}{\{\text{free } t\} \vdash \textbf{Alloc} \Rightarrow \textbf{Alloc}[l \mapsto -1]}\ \textsc{Free}$$

$$\frac{\begin{array}{l}\quad\ \{\overrightarrow{\mathit{param}_i := t_i}\} \vdash \textbf{<S>}_1 \Rightarrow \mathit{Normal}, \textbf{<S>}_2 \\ \{\mathit{body}_f\} \vdash \textbf{Glob}_2, \epsilon[\overrightarrow{\mathit{param}_i \mapsto \textbf{Env}_2(\mathit{param}_i)}], \textbf{Heap}_2, \textbf{Alloc}_2 \Rightarrow \\ \quad \mathit{Return}(v), \textbf{<S>}_3 \\ \quad\ \{x := v\} \vdash \textbf{Glob}_3, \textbf{Env}_1, \textbf{Heap}_3, \textbf{Alloc}_3 \Rightarrow \textbf{<S>}_4\end{array}}{\{x := f(\overrightarrow{t_i})\} \vdash \textbf{<S>}_1 \Rightarrow \textbf{<S>}_4}\ \textsc{Call}$$

Figure 2.20: Semantics of JESSIE instructions

$$\frac{\{s_1\} \vdash \textsf{\textbf{<S>}}_1 \Rightarrow \text{Normal}, \textsf{\textbf{<S>}}_2 \quad \{s_2\} \vdash \textsf{\textbf{<S>}}_2 \Rightarrow out, \textsf{\textbf{<S>}}_3}{\{s_1\ s_2\} \vdash \textsf{\textbf{<S>}}_1 \Rightarrow out, \textsf{\textbf{<S>}}_3}\ \text{SEQ-NORMAL}$$

$$\frac{\{s_1\} \vdash \textsf{\textbf{<S>}}_1 \Rightarrow \text{Return(v)}, \textsf{\textbf{<S>}}_2}{\{s_1\ s_2\} \vdash \textsf{\textbf{<S>}}_1 \Rightarrow \text{Return(v)}, \textsf{\textbf{<S>}}_2}\ \text{SEQ-RETURN}$$

$$\frac{\{s_1\} \vdash \textsf{\textbf{<S>}}_1 \Rightarrow \text{Throw(X)}, \textsf{\textbf{<S>}}_2}{\{s_1\ s_2\} \vdash \textsf{\textbf{<S>}}_1 \Rightarrow \text{Throw(X)}, \textsf{\textbf{<S>}}_2}\ \text{SEQ-THROW}$$

$$\frac{[\![t]\!] = true \quad \{s_1\} \vdash \textsf{\textbf{<S>}}_1 \Rightarrow out, \textsf{\textbf{<S>}}_2}{\{\text{if } t \text{ then } s_1 \text{ else } s_2\} \vdash \textsf{\textbf{<S>}}_1 \Rightarrow out, \textsf{\textbf{<S>}}_2}\ \text{IF-TRUE}$$

$$\frac{[\![t]\!] = false \quad \{s_2\} \vdash \textsf{\textbf{<S>}}_1 \Rightarrow out, \textsf{\textbf{<S>}}_2}{\{\text{if } t \text{ then } s_1 \text{ else } s_2\} \vdash \textsf{\textbf{<S>}}_1 \Rightarrow out, \textsf{\textbf{<S>}}_2}\ \text{IF-FALSE}$$

$$\frac{\{s\} \vdash \textsf{\textbf{<S>}}_1 \Rightarrow \text{Normal}, \textsf{\textbf{<S>}}_2 \quad \{\text{loop } s\} \vdash \textsf{\textbf{<S>}}_2 \Rightarrow out, \textsf{\textbf{<S>}}_3}{\{\text{loop } s\} \vdash \textsf{\textbf{<S>}}_1 \Rightarrow out, \textsf{\textbf{<S>}}_3}\ \text{LOOP-NORMAL}$$

$$\frac{\{s\} \vdash \textsf{\textbf{<S>}}_1 \Rightarrow \text{Return(v)}, \textsf{\textbf{<S>}}_2}{\{\text{loop } s\} \vdash \textsf{\textbf{<S>}}_1 \Rightarrow \text{Return(v)}, \textsf{\textbf{<S>}}_2}\ \text{LOOP-RETURN}$$

$$\frac{\{s\} \vdash \textsf{\textbf{<S>}}_1 \Rightarrow \text{Throw(X)}, \textsf{\textbf{<S>}}_2}{\{\text{loop } s\} \vdash \textsf{\textbf{<S>}}_1 \Rightarrow \text{Throw(X)}, \textsf{\textbf{<S>}}_2}\ \text{LOOP-THROW}$$

$$\frac{[\![t]\!] = v}{\{\text{return } t\} \vdash \textsf{\textbf{<S>}} \Rightarrow \text{Return(v)}, \textsf{\textbf{<S>}}}\ \text{RETURN}$$

Figure 2.21: Semantics of JESSIE statements - normal control

$$\frac{}{\{\text{throw } X\} \vdash \textbf{<S>} \Rightarrow \text{Throw(X)}, \textbf{<S>}}\ \textsc{Throw}$$

$$\frac{\{s_1\} \vdash \textbf{<S>}_1 \Rightarrow \text{Normal}, \textbf{<S>}_2}{\{\text{try } s_1 \text{ catch } X\ s_2\} \vdash \textbf{<S>}_1 \Rightarrow \text{Normal}, \textbf{<S>}_2}\ \textsc{Try-Normal}$$

$$\frac{\{s_1\} \vdash \textbf{<S>}_1 \Rightarrow \text{Return(v)}, \textbf{<S>}_2}{\{\text{try } s_1 \text{ catch } X\ s_2\} \vdash \textbf{<S>}_1 \Rightarrow \text{Return(v)}, \textbf{<S>}_2}\ \textsc{Try-Return}$$

$$\frac{\{s_1\} \vdash \textbf{<S>}_1 \Rightarrow \text{Throw(X)}, \textbf{<S>}_2 \quad \{s_2\} \vdash \textbf{<S>}_2 \Rightarrow out, \textbf{<S>}_3}{\{\text{try } s_1 \text{ catch } X\ s_2\} \vdash \textbf{<S>}_1 \Rightarrow out, \textbf{<S>}_3}\ \textsc{Try-Catch}$$

$$\frac{\{s_1\} \vdash \textbf{<S>}_1 \Rightarrow \text{Throw(X)}, \textbf{<S>}_2 \quad X \not\equiv Y}{\{\text{try } s_1 \text{ catch } Y\ s_2\} \vdash \textbf{<S>}_1 \Rightarrow \text{Throw(X)}, \textbf{<S>}_2}\ \textsc{Try-Throw}$$

Figure 2.22: Semantics of JESSIE statements - exceptional control

Return(v) if the statement terminates on **`return`**, with returned value v, and Throw(X) if the statement terminates on **`throw`** `X`. Thus, the judgment for statement s is

$$\{s\} \vdash \textbf{Glob}_1, \textbf{Env}_1, \textbf{Heap}_1, \textbf{Alloc}_1 \Rightarrow out, \textbf{Glob}_2, \textbf{Env}_2, \textbf{Heap}_2, \textbf{Alloc}_2,$$

which can be abbreviated in

$$\{s\} \vdash \textbf{<S>}_1 \Rightarrow out, \textbf{<S>}_2.$$

Figures 2.21 and 2.22 present the semantics of JESSIE statements.

Programs The semantics of a JESSIE program is the same as the semantics of a call to its entry function `main` (with no arguments), in an empty state:

$$\{main()\} \vdash \epsilon, \epsilon, \epsilon, \epsilon \Rightarrow out, \textbf{Glob}, \textbf{Env}, \textbf{Heap}, \textbf{Alloc}.$$

Erroneous Executions So far, we have only presented the semantics of correct executions of JESSIE programs, both terminating and diverging ones. Erroneous executions can be characterized as those that eventually block when following these semantic rules, because no semantic rule applies anymore. Then, it is possible to complete the set of rules for correct executions with rules for erroneous executions.

The big-step semantic rule for evaluating erroneous term t is described by a judgment of the form

$$[\![t]\!] = \text{err}$$

Examples of such rules are given in Figure 2.23.

$$\frac{[\![t_1]\!] = v_1 \quad [\![t_2]\!] = 0}{[\![t_1 / t_2]\!] = \text{err}} \text{ DIV-ERR}$$

$$\frac{t_1 : S[..] \quad t_2 : S[..] \quad [\![t_1]\!] = (l_1, a, i) \quad [\![t_2]\!] = (l_2, a, j) \quad l_1 \not\equiv l_2}{[\![t_1 \ominus t_2]\!] = \text{err}} \text{ SUBPTR-ERR}$$

$$\frac{t : S[..] \quad [\![t]\!] = (l, a, i) \quad \textbf{Alloc}(l) < i + \mathit{sizeof}(S) \quad m \textit{ not embedded}}{[\![t.m]\!] = \text{err}} \text{ FIELD-ERR}$$

Figure 2.23: Semantics of JESSIE erroneous terms

$$\frac{[\![t]\!] = \text{err}}{\{x := t\} \vdash \textbf{<S>} \Rightarrow \text{err}} \text{ ASSIGN-GLOB-ERR}$$

$$\frac{t_1 : S[..] \quad [\![t_1]\!] = (l, a, i) \quad \textbf{Alloc}(l) < i + \mathit{sizeof}(S)}{\{t_1.m := t_2\} \vdash \textbf{<S>} \Rightarrow \text{err}} \text{ ASSIGN-FIELD-ERR}$$

$$\frac{\{s_1\} \vdash \textbf{<S>} \Rightarrow \text{err}}{\{s_1\ s_2\} \vdash \textbf{<S>} \Rightarrow \text{err}} \text{ SEQ-ERR}$$

Figure 2.24: Semantics of JESSIE erroneous instructions and statements

The big-step semantic rule for erroneous instruction or statement s is similarly described by a judgment of the form

$$\{s\} \vdash \textsf{\textbf{Glob}}, \textsf{\textbf{Env}}, \textsf{\textbf{Heap}}, \textsf{\textbf{Alloc}} \Rightarrow \text{err}$$

Examples of such rules are given in Figure 2.24. Some rules like rule ASSIGN-FIELD-ERR complete the corresponding correct execution rule with cases that lead to an error, while others like rules ASSIGN-GLOB-ERR and SEQ-ERR simply propagate the erroneous outcome.

Statements that neither have a normal or an erroneous outcome are said to diverge [169]. This clearly distinguishes programs that go wrong from programs that diverge.

2.3 C to JESSIE Translation

From C to JESSIE Through CIL As presented in Figure 1.4 describing the Frama-C platform, we first translate C programs into CIL [139]. This translation decides on a set of implementation-defined behaviors that usually depend on the host architecture, such as the size of types, and the C compiler, such as the order of evaluation of expressions. By default, the order of evaluation of binary expressions and function arguments is fixed left-to-right, but this default can be overwritten through options. There is no warning that different orders of evaluation might change the value of an expression, or that such an evaluation is even undefined according to C standard (due to various reads and writes of the same location bewteen two sequence points). Overall, the translation from C to CIL fixes a particular Application Binary Interface. CIL is a simplified abstract syntax of the C program. Most notably, it isolates all side-effects into instructions, thus making expressions side-effect free. As a consequence, it also obviates the need to consider shortcut evaluation of C logical operators `&&` and `||`. Since CIL is at the same level of abstraction as C, one can consider CIL as a proper subset of C, and C to CIL translation as a semantics-preserving rewriting in C. Figure 2.25 presents in a concrete form the abstract syntax of CIL expressions and statements used as intermediate language in Frama-C. In the following, we present our translation of this subset of C into JESSIE. We present all examples in the context of an i386 architecture, which is the default architecture target of CIL and Frama-C.

The translation from the restricted subset of C defined by CIL to JESSIE is straightforward. It is in a large part similar to the translation from C to the normalized C used as the Caduceus intermediate language [2, 93]. Correctness of the overall translation from C to JESSIE can be stated in Claim 1.

Claim 1 *Given a* C *program and a set of implementation-defined decisions implemented in a* C *compiler* CC *and a host architecture* H*, the translation of this program into a* JESSIE *program has the same semantics, as defined in Section 2.2.4, as the* C *executable obtained with compiler* CC *and executed on architecture* H*. In particular, the* C *program is safe iff the corresponding* JESSIE *program is safe.*

The proof of this claim, which depends on both translations from C to CIL and from CIL to JESSIE, is certainly not straightforward. First, it requires that CIL tool and the C

bin-op	::=	+ | -	integer/real/pointer arith.
	|	* | / | %	integer/real arithmetic
	|	%	integer modulo
	|	« | » | & | |	bitwise operations
	|	≤ | ≥ | < | >	integer/real/pointer compar.
	|	≡ | ≢	(dis)equality test
	|	∧ | ∨	logical operations
unary-op	::=	- | ~ | ¬	arith./bit./log. negation
contant	::=	*string*	printed constant
location	::=	*id*	variable
	|	(*location* ⊕ *term*) . *id*	memory location
lval	::=	*lhost offset**	lvalue
lhost	::=	*id*	variable
	|	* *exp*	dereference
offset	::=	. *id*	field access
	|	[*exp*]	array index access
exp	::=	*constant*	constant
	|	*lval*	lvalue
	|	& *lval*	address-of operation
	|	*unary-op exp*	unary operation
	|	*exp bin-op exp*	binary operation
	|	(*typ*) *exp*	cast
instr	::=	*lval* = *exp*	assignment
	|	(*lval* =)$^?$ *exp* ((*exp* (, *exp*)*))	call
switch-lab	::=	`case` *exp*	case label
	|	`default`	default label
stmt	::=	*switch-lab** *stmtkind*	statement
stmtkind	::=	*instr*	instruction
	|	`return` *exp*$^?$	return
	|	`goto` *stmt*	goto
	|	`if` *exp* `then` *block* `else` *block*	conditional
	|	`switch` *exp block*	switch
	|	`loop` *block*	infinite loop
	|	*block*	sequence
block	::=	*stmt**	sequence

Figure 2.25: Grammar of CIL expressions and statements

compiler make the same decisions regarding implementation-defined behaviors. Secondly, it depends on our ability to formally describe both source and target languages. While this could be done (with much work) for CIL and JESSIE, there is no known way to describe the formal semantics of the textual representation of a program. Only proving the correctness of the translation from CIL to JESSIE would be as hard as proving compiler correctness [23, 22, 125], which is beyond the scope of this thesis. Nonetheless, we admit the correctness of Claim 1, the same way one trusts a compiler without a proof of its correctness.

For the sake of clarity, we split the presentation into data translation and control translation, although they are not separated in reality. No preliminary analysis is needed for this translation, except for a little syntactic global information computed by CIL (*e.g.*, knowing whether the address of a variable is taken).

C Constructs Not Supported The translation from C to JESSIE presented in the following does not deal with a few constructs of C, most notably floating-point numbers and function pointers.

C real floating types (`float`, `double`, `long double`) currently translate to the JESSIE `real` type, and operations on floating-point numbers translate to the corresponding operations on real numbers, encoded as uninterpreted functions. Thus, we avoid encoding the complex semantics of machine floating-point arithmetic in JESSIE [76]. Programs relying on floating-point computations for safety can be proved in a proof assistant with a different encoding of floating-point numbers [24].

```
float f                  ⇝   real f
f = 0.5;                 ⇝   f := 0.5
f + 0.5                  ⇝   f + 0.5
```

C function pointers cause difficult problems [159] that are beyond the scope of this thesis. We do not support these currently.

2.3.1 Data Translation

Variables Global (resp. local) variables in C translate to global (resp. local) variables in JESSIE. Function parameters translate to function parameters, which are just special local variables. Variable initializations translate to assignments in JESSIE, with global variable initializations packed together in a global initialization function which is called at the beginning of the `main` function.

Type Definitions Implicit definitions for C integer types (`char`, `short`, `int`, `long`, `long long`), in their signed and unsigned versions, translate to definitions of integer range types in JESSIE. The exact range for each type is architecture dependent. In the bounded integer model, the definition of an enumeration translates to the definition of an integer range type between its minimal enumerator value and its maximal enumerator value. In the modulo integer model, an enumeration translates to its underlying type, which is less precise, to avoid an incorrect modulo semantics in JESSIE on the range of enumerator values while it is performed on the full range of the underlying type in C. Overall, each (implicit or explicit) definition of an arithmetic type σ ranging from value min to value max in C translates into the definition of an integer range type σ' in JESSIE:

$$\sigma \quad \leadsto \quad \mathit{range}\ \sigma' = min..max$$

E.g.,

```
unsigned char           ⇝  range uint8 = 0..255
short                   ⇝  range int16 = −32768..32767
_Bool                   ⇝  range uint1 = 0..1
enum E { a=3, b };      ⇝  range E = 3..4
```

The definition of a C structure type Σ translates to the definition of a JESSIE structure type Σ'. Each C field m_i translates to a JESSIE field m_i', with its type τ translated into τ'. The architecture-dependent bit-size of every field is also made explicit. Finally, padding is made explicit with unnamed fields of type $unit$, whose bit-size is computed as the difference between the bit-offset of the following field (or the complete structure bit-size for the last field) and the first free bit-offset following the current field. This default type for padding does not prevent underlying bits from taking wathever value. It just shows that this value is unimportant to retrieve the structure fields values.

$$\mathit{struct}\ \Sigma\ \{\ \tau_i\ m_i\ ;\ \}\ ; \quad \leadsto \quad \begin{array}{l} \mathit{struct}\ \Sigma' = \{ \\ \quad \tau_i'\ m_i' : \mathit{bitsizeof}(m_i) \\ \quad \mathit{unit}\ \_ : \mathit{bitoffsetof}(m_{i+1}) \\ \qquad\quad - \mathit{bitoffsetof}(m_i) - \mathit{bitsizeof}(m_i) \\ \} \end{array}$$

E.g.,

```
struct S {                 struct S = {
  int i: 3;           ⇝      int3 i: 3   unit _: 5
  char j;                    int8 j: 8   unit _: 16
};                         }
```

The definition of a C union type Σ translates to the definition of a set of JESSIE structure types. The union type itself translates to a JESSIE structure type Σ' with a unique unnamed field of type $unit$. Each field m_i translates to a JESSIE field in its own structure type Σ_i', with a padding field added as necessary.

$$\mathit{union}\ \Sigma\ \{\ \tau_i\ m_i\ ;\ \}\ ; \quad \leadsto \quad \begin{array}{l} \mathit{struct}\ \Sigma' = \{\mathit{unit}\ \_ : \mathit{bitsizeof}(\Sigma)\} \\ \mathit{struct}\ \Sigma_i' = \{ \\ \quad \tau_i'\ m_i' : \mathit{bitsizeof}(m_i) \\ \quad \mathit{unit}\ \_ : \mathit{bitsizeof}(\Sigma) - \mathit{bitsizeof}(m_i) \\ \} \end{array}$$

E.g.,

```
union U {                    struct U = { unit _: 32 }
  int i: 3;            ⇝    struct Ui = { int3 i: 3   unit _: 29 }
  char j;                    struct Uj = { int8 j: 8   unit _: 24 }
};
```

Types Type aliases in C (introduced by **`typedef`**) are replaced by their underlying type. Type qualifiers (`const`, `volatile`, `restrict`) are ignored.

Based on the translation of type definitions, the type τ of a variable or a field in C translates into a JESSIE type τ'. Each arithmetic type σ in C translates into the corresponding integer range type σ' in JESSIE. Pointers and aggregate types in C translate to pointer types in JESSIE.

A variable of type *struct* Σ in C gets type $\Sigma'[0]$ in JESSIE, *i.e.*, a pointer whose validity is guaranteed by typing. Similarly, a variable of type *struct* $\Sigma[n]$ in C gets type $\Sigma'[0\,..\,n-1]$ in JESSIE, *i.e.*, a pointer that can be safely dereferenced between indices 0 and $n-1$.

Likewise, a field of type structure or array of structures translates into an embedded pointer field in JESSIE, *i.e.*, a pointer field whose bounds are known by typing, and which do not represent an additional level of dereference w.r.t. to its parent structure.

Pointers to scalar types (arithmetic types + pointer types) and arrays of scalar types translate to JESSIE pointers. The underlying scalar type is encapsulated in a JESSIE structure type. Pointers translate to unbounded JESSIE pointers, while arrays translate to pointers with statically known bounds.

The following rules summarize the translation of types. Notice that τ is an arbitrary C type, which translates to τ' in JESSIE according to the same rules.

σ	$\leadsto$	σ'
$struct\ \Sigma$	$\leadsto$	$\Sigma'[0]$
$union\ \Sigma$	$\leadsto$	$\Sigma'[0]$
$\tau\star$	$\leadsto$	$\Sigma'[\,..\,]$ *where* Σ' *is defined as* $struct\ \Sigma' = \{\tau'\ m' : bitsizeof(\tau)\}$
$struct\ \Sigma\star$	$\leadsto$	$\Sigma'[\,..\,]$
$union\ \Sigma\star$	$\leadsto$	$\Sigma'[\,..\,]$
$\tau[n]$	$\leadsto$	$\Sigma'[0\,..\,n-1]$ *where* Σ' *is defined as* $struct\ \Sigma' = \{\tau'\ m' : bitsizeof(\tau)\}$
$struct\ \Sigma[n]$	$\leadsto$	$\Sigma'[0\,..\,n-1]$
$union\ \Sigma[n]$	$\leadsto$	$\Sigma'[0\,..\,n-1]$

E.g.,

```
struct S                ⇝   S[0]
struct S*               ⇝   S[..]
union U                 ⇝   U[0]
union U*                ⇝   U[..]
struct S {                   struct S = {
  struct T t;           ⇝     T[0] t: 32
  struct T[2] a;               T[0..1] a: 64
};                           }
```

```
int *                  ⇝  Int32[..]
int [10]               ⇝  Int32[0..9]
```

where `Int32` is a JESSIE structure defined as

```
 struct Int32 = { int32 int32m: 32 }
```

Address-of There is no address-of operator in JESSIE, which means that JESSIE pointers can only point to memory, or equivalently that global and local data cannot be accessed through pointer.

Consequently, variables whose address is taken must be typed differently from variables whose address is not taken, so that the JESSIE variable corresponds to the address of the C variable. If the variable has an aggregate type in C, type translation already gives it type pointer in JESSIE. Otherwise, its JESSIE type is changed to add a level of indirection, much as what is done in HAVOC. We do not consider that all global variables may have their address taken, even if analyzing only part of a program. Rather, we consider that the program part analyzed already takes the address of those global variables that have their address taken.

```
int x                  ⇝  Int32[0] x
&x                     ⇝  x
x                      ⇝  x.int32m
```

Likewise, fields whose address is taken are typed differently from fields whose address is not taken, so that the JESSIE field corresponds to the address of the C field. If the field has an aggregate type in C, type translation already gives it type pointer in JESSIE. Otherwise, its JESSIE type is changed to add a level of indirection. Such fields become embedded fields in JESSIE, so that the indirection remains at a syntactic level, not a semantic one.

```
struct S { int i; };   ⇝  struct S = { Int32[0] i: 32 }
&x.i                   ⇝  x.i
x.i                    ⇝  x.i.int32m
```

Constants Integer, enumeration and character constants in C translate to `integer` constants in JESSIE, so that assigning such a constant to a C variable generates an additional cast from `integer` to the appropriate integer range type in JESSIE. The `NULL` constant in C (in its various forms `0`, `(void*)0`) translates to the `null` constant in JESSIE.

```
i = 5;                 ⇝  i := 5 ▷ int32
p = NULL;              ⇝  p := null
```

Operations Usual arithmetic and comparison operations in C translate in a straightforward way to the corresponding JESSIE operations. Although they are not mentioned here for the sake of simplicity, bitwise operations in C also translate into the corresponding bitwise operations in JESSIE. Field selection and array subscripting in C translate into field selection and pointer arithmetic in Jessie, while respecting the translation of types. Assignment in C translates to assignment in Jessie, with a special treatment for structure assignment in C, so that fields are individually assigned. Casts in C translate to casts in Jessie, for those casts that are supported.

Arithmetic operations in C translate to the corresponding arithmetic operations in Jessie. Since Jessie integer operations return a result of type `integer`, it must be converted to the appropriate integer range if necessary. Casts from an integer range to `integer`, *e.g.*, for operands of arithmetic operators, are left as implicit promotions. Comparison and (dis)equality operations in C translate to the corresponding Jessie operations, returning a `boolean`. When a `boolean` term is used where an `integer` is expected, a choice operation is generated, with value `1` when the boolean is true and `0` otherwise. When an `integer` or a pointer term is used where a `boolean` is expected, the corresponding non-null test is generated.

```
i + 3                ⇝   (i + 3) ▷ int32
j = (i <= 9)         ⇝   j := ((i ≤ 9) ? 1 : 0) ▷ int32
if (p)               ⇝   if (p ≢ null)
```

Field selection in C with either one of operators → and **.** translates to Jessie field selection, which expects a left operand of pointer type. If the left operand has union type in C, a cast to the appropriate structure type is generated in Jessie, before proper field selection. Finally, indirection operator ⋆ in C translates to field selection of the appropriate field in Jessie, after the underlying type is translated to a Jessie structure type.

```
s→i                  ⇝   s.i
s.i                  ⇝   s.i
u.i                  ⇝   (u ▷ Ui[..]).i
*p                   ⇝   p.int32m
```

Array subscripting in C translates readily into pointer arithmetic and field selection in Jessie.

```
a[i].f               ⇝   (a ⊕ i).f
a[i]                 ⇝   (a ⊕ i).int32m
```

Casts between types are more strictly delimited in Jessie than in C. They are of two kinds: casts between mathematical types and integer ranges, a.k.a. base casts, and casts between pointer types, a.k.a. pointer casts. As seen in Jessie semantics in Section 2.2.4, base casts to and from `integer` are precisely defined, based on the integer model chosen. Other base casts behave as if casting first to `integer` and then from `integer` to the original destination type. Pointer casts in C translate to pointer casts in Jessie. They allow one to reinterpret freely the underlying byte pattern. The remaining casts in C cannot be translated into Jessie. In C, a compiler can always choose to reinterpret a bit-pattern from one type to another type, even if the C standard forbids it. *E.g.*, casting between a pointer type and an integer is undefined according to the standard, but most compilers allow it by reinterpreting the bit-pattern. In Jessie, no such cast between base type and pointer type is allowed.

```
(short)i                ⇝   i ▷ int16
(int*)x                 ⇝   x ▷ Int32[..]
```

Allocation Like in Caduceus, calls to the standard allocation functions are recognized, so that a proper type is usually given to the allocated block.

Since JESSIE does not provide initializations, all variables whose type changes from non-pointer in C to pointer in JESSIE, due to the translation, should be properly allocated before use, and, for some of them, properly deallocated before going out of scope. A variable `x` of a structure type or whose address is taken is thus allocated by inserting `x := new S[1]` in the appropriate function if `x` is a local variable, or in the global initialization function if `x` is a global variable. If it is a local variable, deallocation should be performed before the function returns, by inserting `free x`. A variable of an array type is similarly allocated (resp. deallocated) by inserting `a := new S[size]` (resp. `free a`).

```
x = (int*)malloc(n * sizeof(int));  ⇝   x := new Int32[n]
```

2.3.2 Control Translation

Translating control structures from C to JESSIE is much simpler than translating data structures. Most of it has been described extensively in the context of tools ESC/Java [123, 122], Loop [95] and Caduceus [68, 2, 93].

CIL infinite loops with **`break`** and **`continue`** translate into infinite loops in JESSIE with intraprocedural exceptions. Likewise, forward outward gotos translate into intraprocedural exceptions. The CIL translation of **`switch`** into a sequence of **`if`** statements is modified so that it does not generate inward gotos.

C programs with arbitrary gotos could in theory be translated into programs without gotos [148, 65], or at least only those forward outward gotos that we already translate into intraprocedural exceptions. This is not currently implemented in our tool.

2.3.3 A Simple Example: Linear Search

Here is an implementation of linear search in C.

```
int linear_search(int arr[], unsigned int len, int key) {
  int idx = 0;
  while (idx < len) {
    if (arr[idx] == key) {
      return idx; // key found
    }
    idx = idx + 1;
  }
  return −1; // key not found
}
```

It translates into the following JESSIE program.

```
range int32 = −2147483648..2147483647
range uint32 = 0..4294967295

struct Int32 = { int32 int32m : 32 }

int32 linear_search(Int32[..] arr, uint32 len, int32 key) =
  int32 idx
  idx := 0
  try
    loop
      if (¬ (idx < len)) then
        throw Break
      else if ((arr ⊕ idx).int32m ≡ key) then
        return idx
      else
        idx := (idx + 1) ▷ int32
  catch Break
    return −1
```

2.4 JESSIE Annotation Language

JESSIE is not limited to operational constructs that mimic the ones found in C. It contains additional constructs in the form of an *annotation language*, that allows reasoning about execution of JESSIE programs. All these logical constructs are also part of ACSL [14], the ANSI C Specification Language, that was designed in parallel with JESSIE. Both languages were largely inspired by JML, the JAVA Modeling Language.

Some of these logical constructs belong to a new syntactic class of propositions, while others are just new terms that are added to those presented before. JESSIE also contains other types and globals than those presented so far, whose purpose is to express logical properties of programs. We present these logical constructs as extensions of the JESSIE syntax presented before.

Chapters 4, 6 and 7 will extend this annotation language beyond the usual first-order logical constructs presented here. These extensions give a handle to semantic properties in a JESSIE program by making its execution model presented in JESSIE semantics (see Section 2.2) explicit. In particular, it will be possible to express the following questions: *is this memory access valid? do these memory accesses interfere?* Expressing these constructs at the level of JESSIE programs, and not buried into various analyses on JESSIE, has two advantages. First, it makes it possible for a human to interact with the proof at the level of source code, either by specifying properties of interest, or by querying the properties found automatically. Secondly, it allows analyses to communicate their results in a common language.

In the following, we define extensions of the syntactic categories of JESSIE already presented in Section 2.2.1, as well as new syntactic categories for purely logical constructs. While explaining the informal semantics of each construct, we may freely reference a category only defined later on.

type	::=	...	
	|	*id*	logic type
logic-type-def	::=	`logic` *id*	logic type def

Figure 2.26: Grammar of JESSIE extended types

location	::=	...	
	|	`(` *location* $\oplus$ `[` *term*$^?$ `..` *term*$^?$ `]` `)` `.` *id*	range location
	|	`{` *location* `:` *prop* `}`	comprehension location
	|	`result`	function result
term	::=	...	
	|	*id* `(` (*term* (`,` *term*)*)$^?$ `)`	logic function application
	|	`old` `(` *term* `)`	initial value

Figure 2.27: Grammar of JESSIE extended terms

Logic Types Figure 2.26 presents the abstract syntax of JESSIE types, extended with logical types. These types can be used as parameter and result types in logical functions and predicates.

Logic Terms Figure 2.27 presents the abstract syntax of JESSIE terms, extended with logical terms and locations used when describing effects of a function.

Location `(t`$\oplus$`[i..j]).m` extends the notation for memory locations to ranges of offsets, while notation by comprehension `{t:p}` refines a set of locations by designating only those locations `t` that satisfy proposition `p`. Any pointer arithmetic location `(x`$\oplus$`i).m` can be expressed as a range location `(x`$\oplus$`[i..i]).m`, and any range location `(x`$\oplus$`[i..j]).m` can be expressed as a comprehension location `{(x`$\oplus$`[i..j]).m : true}`. Therefore, we will freely use locations in comprehension notation to denote any location in algorithms. Furthermore, any location can be over-approximated by a *path*, which is the location where all predicates in comprehensions have been changed to *true*, which rewrites immediately to a location without comprehension. We will freely use paths in range notation to denote any path in algorithms.

Location `result` denotes the special location used by a function to store its returned value; it should be used only in a function postcondition. Term `old(t)` gives the value of term `t` in the precondition of a function; it should be used only in a function postcondition.

Propositions Figure 2.28 presents the abstract syntax of JESSIE propositions, containing the usual constructs of first-order logic.

rel-op	::=	$\equiv$ | $\not\equiv$ | $\leq$ | $\geq$ | $<$ | $>$ | $\olessthan$ | $\ogreaterthan$	
prop	::=	`true` | `false`	
	|	*term rel-op term*	comparison
	|	*id* `(` (*term* (`,` *term*)*)$^?$ `)`	predicate application
	|	*prop* $\wedge$ *prop*	conjunction
	|	*prop* $\vee$ *prop*	disjunction
	|	*prop* $\Longrightarrow$ *prop*	implication
	|	*prop* $\Longleftrightarrow$ *prop*	equivalence
	|	$\neg$ *prop*	negation
	|	$\forall$ *type id* `;` *prop*	universal quantification
	|	$\exists$ *type id* `;` *prop*	existential quantification

Figure 2.28: Grammar of JESSIE propositions

instr	::=	...	
	|	`assert` *pred*	assertion
stmt	::=	...	
	|	`loop invariant` *pred stmt*	infinite loop

Figure 2.29: Grammar of JESSIE extended statements

var-def	::=	*type id*	
parameters	::=	$(type\ id\ (,\ type\ id)^{*})^{?}$	
locations	::=	`nothing`	empty set of locations
	$\mid$	*location* (, *location*)*	union of locations
logic-fun-def	::=	*type id* (*parameters*)	
		(`reads` *locations* $\mid$ = *term*)	memory footprint or def
pred-def	::=	*id* (*parameters*)	
		(`reads` *locations* $\mid$ = *prop*)	memory footprint or def
fun-def	::=	(`requires` *prop*)$^{?}$	precondition
		(`ensures` *prop*)$^{?}$	postcondition
		(`assigns` *locations*)$^{?}$	memory footprint
		(`allocates` *term* (, *term*)*)$^{?}$	allocation footprint
		(`frees` *term* (, *term*)*)$^{?}$	deallocation footprint
		type id (*parameters*)	
		(= (*type id*)* *stat*)$^{?}$	optional def
axiom-def	::=	`axiom` *id* = *prop*	axiom def
	$\mid$	`lemma` *id* = *prop*	lemma def
glob	::=	*range-def*	ranged integer
	$\mid$	*struct-def*	structure
	$\mid$	*logic-type-def*	logic type
	$\mid$	*var-def*	global variable
	$\mid$	*fun-def*	function
	$\mid$	*logic-fun-def*	logic function
	$\mid$	*pred-def*	predicate
	$\mid$	*axiom-def*	axiom

Figure 2.30: Grammar of JESSIE extended globals

Statement Annotations Figure 2.29 presents the abstract syntax of JESSIE statements, extended with assertions and invariants on loops. The semantics of an asserted proposition is that it should hold when execution reaches the assertion. The semantics of a loop invariant is that it should hold whenever execution reaches the loop beginning, either for the first time or while looping.

Logic Functions and Predicates Figure 2.30 presents the abstract syntax of JESSIE globals, extended with logical functions, predicates and axioms. A logical function (resp. a predicate) may either be defined by a term (resp. a proposition) or less precisely with a set of locations on which it depends, its *memory footprint*. An axiom is a closed valid formula in our model, that can be trusted by provers. It can come in the form of a lemma, which means that it should be possible to derive it from non-lemma axioms, although this proof cannot in general be done automatically.

Function Contracts Figure 2.30 also presents an extended definition of functions. A function may be defined by a body, like before, or by a *function contract*, or both. A function contract is made up of various optional parts:

- a *precondition*, which should hold every time the function is called;
- a *postcondition*, which should hold every time the function returns;
- a *frame condition*, which approximates the effects of calling the function.

The frame condition usually describes which global variables and locations a call to the function may assign to, so that the value stored in these global variables and locations in the pre-state cannot be relied upon in the post-state. The crucial point here is that all other global variables and locations remain untouched. More generally, the frame condition can be defined as a compact representation of the set of possible changes to the state when calling a function, similar to the frame problem in artificial intelligence [25]. From the point of view of the caller, a callee's frame condition is all it needs to perform a sound analysis of the effects of calling the function.

Thus, the frame condition may be richer than only a set of global variables and locations possibly assigned to. It may provide information about the set of exceptions possibly raised (in a language with interprocedural exceptions), on the set of locations allocated and deallocated (in a language with dynamic allocation), on the maximum time taken to call the function, *etc*. The set of global variables and locations possibly assigned to by a function is called its *memory footprint* [32]. Contrary to the memory footprint of logic functions and predicates, it does not include the global variables and locations possibly read. Memory footprint is a part of the frame condition for that function. The only other way a call can modify the state is to allocate or deallocate memory. Therefore, we define an $assigns$ clause for the memory footprint, an $allocates$ clause for the allocation footprint and a $frees$ clause for the deallocation footprint. Like memory footprint, the deallocation part of a frame condition is an over-approximation of what is deallocated. Contrary to memory footprint, the allocation part of a frame condition is an under-approximation of what is allocated. This allows one to safely approximate which pointers are valid after a call: a pointer remains valid if it does not alias with a pointer possibly deallocated, and it becomes valid if it belongs to the set of newly allocated pointers.

2.5 JESSIE to WHY Translation

WHY [67] is a unique language specialized for deductive verification. Like JESSIE, it contains both operational and logical constructs, as well as exceptions. However, there are two crucial differences between JESSIE and WHY:

- *monadic style* - There is no implicit memory in WHY. Thus, memory must be explicitly passed in calls as an extra argument to every function that accesses memory, whether for reading or writing it.
- *alias-free* - By severely restricting the use of references (only single-level references) and parameter passing (no argument can be passed in twice), WHY is a rare case of an alias-free language, where two syntactically different locations cannot represent the same semantic location at runtime.

The translation from JESSIE to WHY bears much resemblance with the translation from the Caduceus intermediate language to WHY [67, 68]. We only detail in the following those elements that are useful to understand later on which changes to this translation we propose in this thesis.

Special types are defined for pointers and memory:

- *pointer* is the type of pointers;
- *heap* is the type of memories.

Special functions are defined for accessing memory through a pointer:

- $select$ takes a memory variable *Heap*, a pointer x, an integer offset *off* and an integer size siz in arguments, and it returns the bit vector stored in *Heap* at offset *off* from the location pointed-to by x, spanning siz bytes.
- $update$ takes a heap variable *Heap*, a pointer x, an integer offset *off*, an integer size siz and a bit vector value v in arguments, and it returns a modified *Heap* such that the bit vector value stored in the returned heap at offset *off* from the location pointed-to by x, spanning siz bytes, is now v.

Then, functions $of\text{-}bits_\tau$ and $to\text{-}bits_\tau$ for each type τ allow one to convert between a value of type τ and the bit vector which represents this value in memory.

E.g., function `linear_search` translated in JESSIE in Section 2.3.3 translates into the following WHY program.

```
logic type int32
logic type uint32

logic type Int32

int32 linear_search
      (pointer arr, uint32 len, int32 key, heap Heap) =
  int32 ref idx
  idx := 0
  try
    loop
      if (¬ (idx < len)) then
        throw Break
      else if (of_bits_int32m(
                   select(Heap,arr ⊕ idx,
                          offsetof(int32m),sizeof(int32m)))
```

```
                ≡ key) then
          return idx
        else
          idx := (idx + 1) ▷ int32
    catch Break
      return −1
```

2.6 Other Related Work

Many tools for program verification use an intermediate language to simplify the complex task of generating verification conditions. Caduceus [68] and Krakatoa [129] use the intermediate language WHY for that purpose, ESC/Java [70] relies on a guarded-command language, HAVOC [47], Spec# [10] and VCC [43] rely on language BoogiePL which makes memory explicit as a single heap variable.

Many other tools directly work on the input language. LOOP [18] directly translates Java and JML code into a set of higher-order theories for PVS. Jack [13] directly generates verification conditions from Java bytecode. KeY [15] and KIV [149] rely on dynamic logic to simulate program execution in logic. Smallfoot [17] directly translates the effects of program statements in terms of separation logic effects.

Our intermediate language JESSIE is not meant to be the intermediate language used for generating verification conditions, but rather an intermediate language for generating annotations, prior to generating verification conditions.

2.7 Chapter Summary

We presented the intermediate language JESSIE, which combines a simple imperative operational language with a logic annotation language. C programs annotated in ACSL can be translated into equivalent JESSIE programs so that a program in C is safe *iff* its translation to JESSIE is safe.

JESSIE considerably simplifies the task of creating analyses for the verification of C programs. It offers a simple yet powerful syntax and semantics, which allows for an easy exposure and understanding of analyses. Its embedded logic annotation language allows the communication of results between analyses in a uniform way.

Chapter 3

Integer Safety Checking

Contents

In this chapter, we present two techniques for integer safety checking of annotated JESSIE programs, in an automatic and modular way: abstract interpretation and deductive verification.

The most essential data type in programming languages is the type of integers. Most program analysis techniques were initially developed for programs that exclusively manipulate integers, a.k.a. integer programs. Static safety checking for JESSIE integer programs reduces to checking the absence of arithmetic errors: integer overflow and division or modulo by zero. Section 3.1 reduces static safety checking for JESSIE integer programs to assertion checking.

Since abstract interpretation and deductive verification target assertion checking, it is possible to apply them to check those assertions that guard against integer safety errors. Section 3.2 presents the theory and practice of abstract interpretation in general, and its application to check integer safety of JESSIE programs in particular. Section 3.3 presents the theory and practice of deductive verification in general, and its application to check integer safety of JESSIE programs in particular.

3.1 Assertions for Integer Safety

3.1.1 Integer Checks

Absence of errors due to operations on integers during execution of a JESSIE program can be expressed as *checks* in JESSIE, *i.e.*, assertions that guard against erroneous executions. According to JESSIE semantics presented in Section 2.2.4, there are two such possible errors that stem from using integers: division and modulo by zero and integer overflow in the bounded integer model. All other errors stem from the use of pointers. In the following, we reuse the variable names from each semantic rule considered.

In rule BINOP, a binary operation should be defined on its operands. In JESSIE, only the division and modulo are not defined for some values of operands, namely when the divisor is null. This can be guarded by assertion

$$t_2 \neq 0.$$

In rule TO-RANGE, casting an integer to an integer range is allowed only if the operand is within the bounds of the integer range type, which is expressed by assertion

$$min_E \leq t \leq max_E.$$

JESSIE semantics ensures that well-typed JESSIE programs are safe w.r.t. operations on integers *iff* these checks hold.

3.1.2 Integer Safety for Linear Search

Searching is a simple task, and searching in an unordered list of items, a.k.a. linear search, is probably one of the simplest programming assignments given to every beginner. Although it is very simple, it is not trivial, thus being a perfect candidate as a running example for program verification [28]. Here is the program `linear_search` presented in Section 2.3.3, annotated in ACSL, where `INT_MAX` is used as a convenient shorthand for the constant `2147483647`. In the following, we freely use constants from the C standard library for the sake of clarity.

```
/*@ requires len ≤ INT_MAX;
  @ ensures −1 ≤ \result < len;
  @ assigns \nothing;
  @*/
int linear_search(int arr[], unsigned int len, int key) {
  int idx = 0;
  //@ loop invariant 0 ≤ idx ≤ len;
  while (idx < len) {
    if (arr[idx] == key) {
      return idx; // key found
11      }
12      idx = idx + 1;
13    }
14    return −1; // key not found
15 }
```

Annotations only deal with ranges of integer variables here. Still, these annotations guarantee integer safety, and that the value returned is within simple bounds. The C program translates into the following JESSIE program.

```
 1 range int32 = −2147483648..2147483647
 2 range uint32 = 0..4294967295
 3
 4 struct Int32 = { int32 int32m : 32 }
 5
 6 requires len ≤ INT_MAX
 7 ensures −1 ≤ result < len
 8 assigns nothing
 9 int32 linear_search(Int32[..] arr, uint32 len, int32 key) =
10   int32 idx
11   idx := 0
12   try
13     loop invariant 0 ≤ idx ≤ len
14       if (¬ (idx < len)) then
15         throw Break
16       else if ((arr ⊕ idx).int32m ≡ key) then
17         return idx
18       else
19         idx := (idx + 1) ▷ int32
20   catch Break
21     return −1
```

This program contains one check for integer safety on line 19, plus a loop invariant on line 13, a postcondition on line 7 and a frame condition on line 8. On line 19 (line 12 in C), addition on `idx` should not overflow, which gives check C_{12}:

$$C_{12} \doteq \texttt{INT\_MIN} \leq \texttt{idx} + 1 \leq \texttt{INT\_MAX}.$$

It is already known by typing that the following invariant holds at the same program point:

$$typ_{12} \doteq \texttt{INT\_MIN} \leq \texttt{idx} \leq \texttt{INT\_MAX}.$$

This proves that the lower bound on $idx + 1$ holds in C_{12}, but not that the upper bound holds. In fact, a precondition is needed to ensure that C_{12} holds. The precondition stated in line 6 (line 1 in C) requires from the calling context to pass in a length `len` not bigger than the maximum signed integer. This condition effectively ensures that C_{12} holds. We are going to present how to apply abstract interpretation and deductive verification to JESSIE programs to automatically prove that such integer checks and annotations hold.

3.1.3 Assertions from Annotations

In Section 2.4, we added assertions, loop invariants and function contracts to JESSIE. In order to use them for proving that checks hold, we should prove the corresponding logical assertions.

An assertion of proposition p asks for p to hold at that point. A loop invariant of proposition p asks for p to hold each time control reaches the beginning of the loop. Calling a function f is valid only when its precondition pre_f is satisfied, where function parameters $\overrightarrow{\mathit{param}_i}$ are replaced with actual arguments $\overrightarrow{t_i}$:

$$\mathit{pre}_f[\overrightarrow{\mathit{param}_i} \mapsto \overrightarrow{t_i}].$$

Returning from a function f is valid only when its postcondition post_f and frame condition frame_f hold:

$$\mathit{post}_f \wedge \mathit{frame}_f.$$

3.2 Abstract Interpretation for Integer Programs

Abstract interpretation [51] is a theory of abstract semantics of programs. From a theoretical point of view, it allows one both to relate an abstract semantics to the concrete semantics of a program, and to compare abstract semantics between them. From a practical point of view, it gives techniques to automatically build models of programs in some chosen directions of abstraction, which allows one to infer invariants on the models that are true on the programs too. It works by successive iterations on the program's control flow that collect the sets of values of variables in the model at each point.

3.2.1 Theory of Abstract Interpretation

An *abstract domain* defines a direction of abstraction for building an abstract model of the program. It relies on an internal *abstract lattice*, an algebraic structure that describes the ordering relation between elements in the model, so that each iteration on the program's control flow can only increase abstract values w.r.t. this ordering. A lattice is usually best described by a tuple $(L, \sqsubset, \perp, \top, \sqcup, \sqcap)$, where:

- L is the set of elements in the lattice;
- $\sqsubset$ is the (non-strict) ordering relation;
- $\perp$ is the least element in the lattice;
- $\top$ is the greatest element in the lattice;
- $\sqcup$ is the union (join) of elements in the lattice;
- $\sqcap$ is the intersection (meet) of elements in the lattice.

An important property is that $\sqcup$ and $\sqcap$ are consistent with respect to the ordering $\sqsubset$, which amounts to

$$x \sqsubset y \Rightarrow (x \sqcup y = y \wedge x \sqcap y = x),$$
$$x \sqcup y = y \Rightarrow x \sqsubset y,$$
$$x \sqcap y = x \Rightarrow x \sqsubset y.$$

Connection between the program and its model is done through a Galois connection (α, γ), where the abstraction function α maps each set of concrete program states into an abstract element of L, and the concretization function γ maps each abstract element of L to a set of concrete program states. For the lattice to correctly over-approximate the set of possible program behaviors, we need the following properties on the concrete side:

$$\gamma(x \sqcup y) \supseteq \gamma(x) \cup \gamma(y), \tag{3.1}$$

$$\gamma(x \sqcap y) \subseteq \gamma(x) \cap \gamma(y), \tag{3.2}$$

$$\gamma \circ \alpha(s) \supseteq s. \tag{3.3}$$

Most lattices we use in practice are convex, or stable by intersection, which means that we can rewrite equation (3.2) into

$$\gamma(x \sqcap y) = \gamma(x) \cap \gamma(y). \tag{3.4}$$

Given an abstract lattice L, an abstract domain **D** can be built on L by defining additional transfer functions that over-approximate the effect of a program statement on a value of L:

- **D**.test is the transfer function for test;
- **D**.assign is the transfer function for assignment;
- **D**.forget is the transfer function for reset.

The effect of an assignment is over-approximated either by **D**.assign when the right-hand side of the assignment is simple enough, or by **D**.forget otherwise, in which case all information about the value of the location assigned is lost. These transfer functions are usually built upon the underlying lattice operations. The essential property of these transfer functions is monotonicity, which amounts to

$$x_1 \sqsubset x_2 \Rightarrow \mathbf{D}.f(x_1) \sqsubset \mathbf{D}.f(x_2),$$

for **D**.f any of **D**.test, **D**.assign or **D**.forget.

Other operations of the abstract domain are simple wrappers on underlying lattice operations:

- **D**.union is the join operation $\sqcup$;
- **D**.included is the ordering relation $\sqsubset$.

D.union is used to perform unions of abstract values at junctions of paths during the propagation leading to the definition of the abstract model of a program. **D**.included is used to check the validity of a proposition expressed as an element of L.

There is generally a last operation defining an abstract domain:

- **D**.widen is the widening operation.

When the lattice L has bounded height, convergence of the propagation is ensured by the monotonicity of transfer functions. When L has infinite height, convergence is not ensured by monotonicity. **D**.widen allows one to ensure or to accelerate convergence. It is used to jump to an over-approximation of the set of reachable states of the program, possibly after several applications of **D**.widen. Given an ascending chain of abstract values $\mathcal{A}_i$, the chain $\mathcal{A}'_i$ defined by

$$\begin{array}{rcl} \mathcal{A}'_0 & \doteq & \mathcal{A}_0 \\ \mathcal{A}'_{i+1} & \doteq & \mathbf{D}.\mathsf{widen}(\mathcal{A}'_i, \mathcal{A}_i) \end{array}$$

is guaranteed to converge, *i.e.*,

$$\exists\, k \,.\, \forall\, i \,.\, k \leq i \Longrightarrow \mathcal{A}'_{i+1} \equiv \mathcal{A}'_i .$$

Finally, in order to facilitate some analyses, we require that the domain considered provides two query operations:

- **D**.lbound is the lower bound query;
- **D**.ubound is the upper bound query.

D.lbound provides a (possibly empty) set of lower bounds $\overrightarrow{x_l}$ for the abstract value x given in argument, such that each lower bound verifies $x_l \sqsubset x$. **D**.ubound provides a (possibly empty) set of upper bounds $\overrightarrow{x_u}$ for the abstract value x given in argument, such that each upper bound verifies $x \sqsubset x_u$. Neither of these operators is required to return a set of all lower or upper bounds. Returning the empty set is a valid behavior for both operations.

Propagation of abstract values through the control-flow graph can follow different iteration strategies [27].

3.2.2 Practical Abstract Domains

Many abstract domains have been devised for inferring bounds on the value of integer variables, or relations between the values of two or more integer variables. These abstract domains usually belong to one of four main categories:

- *non-relational domains* - These domains bound or restrict in some way the value of individual variables.
- *relational domains* - These domains restrict the possible relations between the value of two or more variables.
- *combination domains* - These domains build more complex domains based on simpler ones.

Their time and space complexity can be measured in terms of the number of variables considered, whether it is the number of variables in the term considered (for non-relational domains), the number of variables in *packs*, *i.e.*, sets of variables which might be related (*e.g.*, for octagons), or the number of variables related (*e.g.*, for polyhedrons). In the following, we denote the set of these variables by $\mathcal{V}$.

The simplest abstract domains are non-relational domains. They are also the cheapest ones, when considering both time and space complexity of abstract domain operations (constant or linear), and convergence (by being of finite height or with a cheap widening operation).

- *sign*: $L \doteq \{\bot, -, +, \top\}$ - Sign $-$ represents all negative integers, while sign $+$ represents all positive integers. It can be refined to represent zero, non-negative and non-positive integers as well. It has finite height 3.
- *intervals*: $L \doteq \{\ [a\ ;\ b]$ such that $\ a, b \in \mathbb{Z} \cup \pm\infty\}$ - It bounds the value of integer variables. It is very cheap while providing generally useful information, which makes it the most commonly used abstract domain.
- *congruence*: $L \doteq \mathbb{Z} \times \mathbb{N}$ - Pair (a, b) represents the set of integers $\{a + i \times b$ such that $\ i \in \mathbb{Z}\}$. It is notably used for representing addresses, which should be aligned both w.r.t. machine architecture and w.r.t. the enclosing structure if any.

The most complex abstract domains still commonly used are relational domains. They vary in complexity, from quadratic or cubic complexity for weakly relational domains which relate the value of 2 or 3 variables to exponential complexity in the worst case for full relational domains which relate the value of an unbounded number of variables.

- *DBM* (Difference Bound Matrices) [60]: $L \doteq \mathcal{P}(\{a \leq x - y \leq b$ such that $a, b \in \mathbb{Z} \cup \pm\infty \wedge x, y \in \mathcal{V}\})$ - It represents sets of potential constraints between variables.
- *octagons* [132]: $L \doteq \mathcal{P}(\{a \leq x \pm y \leq b$ such that $a, b \in \mathbb{Z} \cup \pm\infty \wedge x, y \in \mathcal{V}\})$ - It represents a refinement of DBM, so that both differences and sums of variables are represented.
- *linear equalities* [108]: $L \doteq \mathcal{P}(\{\sum a_i \times x_i \equiv b$ such that $a_i, b \in \mathbb{Z} \wedge x_i \in \mathcal{V}\})$ - It represents linear equalities between program variables.
- *polyhedrons* [52, 44]: $L \doteq \mathcal{P}(\{\sum a_i \times x_i \leq b$ such that $a_i, b \in \mathbb{Z} \wedge x_i \in \mathcal{V}\})$ - It represents linear inequalities between program variables.

Combination domains build on simpler domains **A** and **B**, with respective lattices A and B, to combine their results or to express disjunctive properties.

- *cross product* [50]: $L \doteq A \times B$ - It simply adds up the results of **A** and **B**.
- *reduced product* [50]: $L \doteq A \times B$ - It combines the results of **A** and **B** in a way that the results of one abstract domain are used to refine the results of the other and conversely.

$$\frac{t \textit{ treated by } \mathbf{D}.\mathsf{assign}}{\{x := t\} \vdash \mathcal{A} \Rightarrow \mathbf{D}.\mathsf{assign}(\mathcal{A}, x, t)} \ \textsc{Assign-Var}$$

$$\frac{t \textit{ not treated by } \mathbf{D}.\mathsf{assign}}{\{x := t\} \vdash \mathcal{A} \Rightarrow \mathbf{D}.\mathsf{forget}(\mathcal{A}, x)} \ \textsc{Forget-Var}$$

$$\frac{}{\{t_1.m := t_2\} \vdash \mathcal{A} \Rightarrow \mathcal{A}} \ \textsc{Ignore-Field}$$

$$\frac{}{\{x := \text{new } S[t]\} \vdash \mathcal{A} \Rightarrow \mathcal{A}} \ \textsc{Ignore-New} \qquad \frac{}{\{\text{free } t\} \vdash \mathcal{A} \Rightarrow \mathcal{A}} \ \textsc{Ignore-Free}$$

$$\frac{}{\{x := f(\overrightarrow{t_i})\} \vdash \mathcal{A} \Rightarrow \mathbf{D}.\mathsf{test}(\mathbf{D}.\mathsf{forget}(\mathcal{A}, x_j), \mathit{post}_f[\overrightarrow{\mathit{param}_i \mapsto t_i}, \mathit{result} \mapsto x])} \ \textsc{Call}$$

Figure 3.1: Intraprocedural abstract interpretation of instructions

- *power domain* [50]: $L \doteq A \times B$ - An abstract element (a, b) expresses that concrete elements represented by a should also be represented by b. It is also called the implication domain. It is a form of disjunctive property.

- *disjunctive completion* [50]: $L \doteq A \times A \times \ldots \times A$ - An abstract element $(a_1, a_2, \ldots, a_n)$ represents the disjunction of elements $\bigvee a_i$. As any disjunctive representation, it may quickly diverge.

- *logical product* [80]: $L \doteq A \times B$ - It combines the results of **A** and **B** in a more precise way than a reduced product would, provided A and B are logical lattices (*i.e.*, whose elements are finite conjunctions of atoms from a theory) and the underlying theories are convex, stably infinite and disjoint. It relies on the Nelson-Oppen combination of theories.

3.2.3 Application to JESSIE Integer Programs

The intraprocedural abstract interpretation of JESSIE programs can be defined as a dataflow analysis on instructions and statements. For the sake of simplicity, we only consider the case of a single intraprocedural exception X, but the generalization to more than one exception is obvious. We denote the abstract value at the current program point as $\mathcal{A}$.

The effect of an instruction s is described by judgment

$$\{s\} \vdash \mathcal{A}_1 \Rightarrow \mathcal{A}_2,$$

with the meaning that executing instruction s from a state in the concretization of abstract state $\mathcal{A}_1$ leads to a state in the concretization of abstract state $\mathcal{A}_2$. Figure 3.1 presents the

$$\frac{\{s\} \vdash \mathcal{A}_1^N \Rightarrow \mathcal{A}_2^N}{\{s\} \vdash \mathcal{A}_1^N \Rightarrow (\mathcal{A}_2^N, \epsilon, \epsilon)}\ \textsc{instr}$$

$$\frac{\{s_1\} \vdash \mathcal{A}_1^N \Rightarrow (\mathcal{A}_2^N, \mathcal{A}_2^R, \mathcal{A}_2^X) \quad \{s_2\} \vdash \mathcal{A}_2^N \Rightarrow (\mathcal{A}_3^N, \mathcal{A}_3^R, \mathcal{A}_3^X)}{\{s_1\ s_2\} \vdash \mathcal{A}_1^N \Rightarrow (\mathcal{A}_3^N, \textbf{D}.\mathsf{union}(\mathcal{A}_2^R, \mathcal{A}_3^R), \textbf{D}.\mathsf{union}(\mathcal{A}_2^X, \mathcal{A}_3^X))}\ \textsc{seq}$$

$$\frac{\begin{array}{c}\{s_1\} \vdash \textbf{D}.\mathsf{test}(\mathcal{A}_1^N, t) \Rightarrow (\mathcal{A}_2^N, \mathcal{A}_2^R, \mathcal{A}_2^X) \\ \{s_2\} \vdash \textbf{D}.\mathsf{test}(\mathcal{A}_1^N, \neg t) \Rightarrow (\mathcal{A}_3^N, \mathcal{A}_3^R, \mathcal{A}_3^X)\end{array}}{\begin{array}{l}\{\text{if } t \text{ then } s_1 \text{ else } s_2\} \vdash \mathcal{A}_1^N \Rightarrow \\ \quad (\textbf{D}.\mathsf{union}(\mathcal{A}_2^N, \mathcal{A}_3^N), \textbf{D}.\mathsf{union}(\mathcal{A}_2^R, \mathcal{A}_3^R), \textbf{D}.\mathsf{union}(\mathcal{A}_2^X, \mathcal{A}_3^X))\end{array}}\ \textsc{if}$$

$$\frac{\begin{array}{c}\{s\} \vdash \mathcal{A}_1^N \Rightarrow (\mathcal{A}_2^N, \mathcal{A}_2^R, \mathcal{A}_2^X) \\ \{\text{loop } s\} \vdash \textbf{D}.\mathsf{union}(\mathcal{A}_1^N, \mathcal{A}_2^N) \Rightarrow (\mathcal{A}_3^N, \mathcal{A}_3^R, \mathcal{A}_3^X)\end{array}}{\{\text{loop } s\} \vdash \mathcal{A}_1^N \Rightarrow (\mathcal{A}_3^N, \mathcal{A}_3^R, \mathcal{A}_3^X)}\ \textsc{loop-unroll}$$

$$\frac{\begin{array}{c}\{s\} \vdash \mathcal{A}_1^N \Rightarrow (\mathcal{A}_2^N, \mathcal{A}_2^R, \mathcal{A}_2^X) \\ \textbf{D}.\mathsf{included}(\mathcal{A}_2^N, \mathcal{A}_1^N)\end{array}}{\{\text{loop } s\} \vdash \mathcal{A}_1^N \Rightarrow (\mathcal{A}_1^N, \mathcal{A}_2^R, \mathcal{A}_2^X)}\ \textsc{loop-converge}$$

$$\frac{}{\{\text{return } t\} \vdash \mathcal{A}_1^N \Rightarrow (\epsilon, \mathcal{A}_1^N, \epsilon)}\ \textsc{return}$$

$$\frac{}{\{\text{throw } X\} \vdash \mathcal{A}_1^N \Rightarrow (\epsilon, \epsilon, \mathcal{A}_1^N)}\ \textsc{throw}$$

$$\frac{\{s_1\} \vdash \mathcal{A}_1^N \Rightarrow (\mathcal{A}_2^N, \mathcal{A}_2^R, \mathcal{A}_2^X) \quad \{s_2\} \vdash \mathcal{A}_2^X \Rightarrow (\mathcal{A}_3^N, \mathcal{A}_3^R, \mathcal{A}_3^X)}{\{\text{try } s_1 \text{ catch } X\ s_2\} \vdash \mathcal{A}_1^N \Rightarrow (\textbf{D}.\mathsf{union}(\mathcal{A}_2^N, \mathcal{A}_3^N), \textbf{D}.\mathsf{union}(\mathcal{A}_2^R, \mathcal{A}_3^R), \mathcal{A}_3^X)}\ \textsc{try}$$

Figure 3.2: Intraprocedural abstract interpretation of statements

abstract interpretation of JESSIE instructions. All instructions that do not modify the value of integer variables are ignored. In rule CALL, x_j denotes any variable that can be modified by calling f. Of course, if an actual parameter t_i mentions a modified variable x_j, then the value of this parameter is lost by operation **D**.forget. This can be improved upon by adding temporary variables to hold the value of parameters in the pre-state. Moreover, the analysis can be made more precise by filtering valid calls through an application of **D**.test to the precondition of the call.

The effect of a statement s is described by judgment

$$\{s\} \vdash \mathcal{A}_1^N \Rightarrow (\mathcal{A}_2^N, \mathcal{A}_2^R, \mathcal{A}_2^X),$$

with the meaning that executing statement s from a state in the concretization of abstract state $\mathcal{A}_1^N$ leads to a state in the concretization of one of three abstract states depending on the outcome of s, as defined in JESSIE semantics (Section 2.2.4):

- $\mathcal{A}_2^N$ if the outcome is Normal;
- $\mathcal{A}_2^R$ if the outcome is Return;
- $\mathcal{A}_2^X$ if the outcome is Throw(X).

Figure 3.2 presents the abstract interpretation of JESSIE statements. Rule LOOP-UNROLL is the only recursive rule. It implicitly suggests an iterative procedure to discover a fix-point to the data-flow propagation, namely to iterate the propagation through the loop body until it converges, which is expressed by rule LOOP-CONVERGE. When convergence is not ensured, it is the role of the widening function to provide it.

Taking type invariants and annotations (*e.g.*, preconditions) into account simply consists in calling **D**.test to further constrain the current abstract value. Checking that a proposition p holds is possible if p can be expressed as an abstract value $\mathcal{A}_p$. Then, p holds if **D**.included$(\mathcal{A}, \mathcal{A}_p)$, where $\mathcal{A}$ is the current abstract value.

In this thesis, we only consider intraprocedural abstract interpretation. Rousset, in his PhD thesis [156], presents the interprocedural abstract interpretation of JESSIE programs, with interprocedural exceptions, in the context of the verification of JAVA programs.

3.2.4 Illustration on Linear Search

Various abstract domains among those presented in Section 3.2.2 can be considered to perform abstract interpretation of unannotated function `linear_search` presented in Section 2.3.3:

- **Sign**, the abstract domain of signs;
- **Interv**, the abstract domain of intervals;
- **Oct**, the abstract domain of octagons;
- **Poly**, the abstract domain of polyhedrons.

	loop invariant	invariant at C_{12}	postcondition
Sign	$0 \leq$ `idx`	$0 \leq$ `idx`	
Interv	$0 \leq$ `idx`	$0 \leq$ `idx`	$-1 \leq \mathit{result}$
Oct	$0 \leq$ `idx` $\leq$ `len`	$0 \leq$ `idx` $<$ `len`	$-1 \leq \mathit{result} <$ `len`
Poly	$0 \leq$ `idx` $\leq$ `len`	$0 \leq$ `idx` $<$ `len`	$-1 \leq \mathit{result} <$ `len`

Figure 3.3: Abstract interpretation of unannotated function `linear_search`

The results, presented in Figure 3.3, do not allow one to prove integer safety for function `linear_search` which, according to Section 3.1.2, amounts to verification of check C_{12}:

$$C_{12} \doteq \texttt{INT\_MIN} \leq \texttt{idx} + 1 \leq \texttt{INT\_MAX}.$$

Indeed, inequality $\texttt{idx} + 1 \leq \texttt{INT\_MAX}$ is not implied by any invariant at C_{12}, since unsigned integer `len` might be as big as `UINT_MAX`. Taking into account typed invariants to strengthen invariants does not allow one to prove C_{12} either.

	loop invariant	invariant at C_{12}	postcondition
Sign	$0 \leq$ `idx` $0 \leq$ `len`	$0 \leq$ `idx` $0 \leq$ `len`	
Interv	$0 \leq$ `idx` $\leq$ `INT_MAX` $0 \leq$ `len` $\leq$ `INT_MAX`	$0 \leq$ `idx` $\leq$ `INT_MAX` $0 \leq$ `len` $\leq$ `INT_MAX`	$-1 \leq \mathit{result} \leq$ `INT_MAX`
Oct	$0 \leq$ `idx` $\leq$ `len` `len` $\leq$ `INT_MAX`	$0 \leq$ `idx` $<$ `len` `len` $\leq$ `INT_MAX`	$-1 \leq \mathit{result} <$ `len` `len` $\leq$ `INT_MAX`
Poly	$0 \leq$ `idx` $\leq$ `len` `len` $\leq$ `INT_MAX`	$0 \leq$ `idx` $<$ `len` `len` $\leq$ `INT_MAX`	$-1 \leq \mathit{result} <$ `len` `len` $\leq$ `INT_MAX`

Figure 3.4: Abstract interpretation of annotated function `linear_search`

The same abstract domains can be considered to perform abstract interpretation of function `linear_search` annotated in ACSL presented in Section 3.1.2. The results, presented in Figure 3.4, allow one to prove C_{12} with either one of **Oct** or **Poly**.

3.3 Deductive Verification for Integer Programs

Deductive verification, through the definition of a suitable Hoare logics, reduces the verification of checks in a fully annotated program to the validity of formulas. Computation of Dijkstra's weakest preconditions or strongest postconditions allows one to reduce the annotation burden to only a few specific annotations: function pre- and postconditions, loop invariants. Classically, weakest preconditions are preferred over strongest postconditions, because they avoid the introduction of existential quantifiers which may increase the difficulty of automatically proving the resulting formula.

$$\frac{p_1 \Rightarrow p_2[x \mapsto t]}{\{p_1\}x := t\{p_2\}}\ \text{HOARE-ASSIGN} \qquad \frac{\{p_1\}s_1\{p_2\} \quad \{p_2\}s_2\{p_3\}}{\{p_1\}s_1\ ;\ s_2\{p_3\}}\ \text{HOARE-SEQ}$$

$$\frac{\{p_1 \wedge t\}s_1\{p_2\} \quad \{p_1 \wedge \neg t\}s_2\{p_2\}}{\{p_1\}\text{if } t \text{ then } s_1 \text{ else } s_2\{p_2\}}\ \text{HOARE-IF}$$

$$\frac{p_1 \Rightarrow p_i \quad \{p_i \wedge t\}s\{p_i\} \quad p_i \wedge \neg t \Rightarrow p_2}{\{p_1\}\text{while } t \text{ do } s\{p_2\}}\ \text{HOARE-LOOP}$$

Figure 3.5: Generic rules of Hoare logics

3.3.1 Hoare Logics and Dijkstra's Weakest Preconditions

Hoare logics is a general framework for the definition of programs logics, allowing one to reason about program executions. It defines rules for the validity of Hoare triples

$$\{p_1\}\ s\ \{p_2\},$$

where p_1, p_2 are propositions and s is an instruction or a statement with a possible effect on the state. The meaning of such a Hoare triple is that whenever proposition p_1 holds before the execution of s, proposition p_2 holds afterwards.

Generic rules include the rule for assignment HOARE-ASSIGN, the rule for sequence HOARE-SEQ, the rule for branching HOARE-IF and the rule for loop HOARE-LOOP. Notice that the generic rule for loop expects a loop with a test to indicate when to exit the loop, which is not the case in JESSIE.

To verify that a proposition holds at some program point, one must manually annotate all statements before this program point in the control-flow graph of the function. This is not feasible except for very small programs. Dijkstra's calculus, either through weakest preconditions or strongest postconditions, allows the automatic generation of most annotations. Weakest preconditions can be defined as a function $\mathcal{W}$ st

$$\{\mathcal{W}\{s\}(p)\}\ s\ \{p\},$$

i.e., whenever $\mathcal{W}\{s\}(p)$ holds before s is executed, p holds afterwards. Similarly, strongest postconditions can be defined as a function $\mathcal{S}$ st

$$\{p\}\ s\ \{\mathcal{S}\{s\}(p)\},$$

i.e., whenever p holds before s is executed, $\mathcal{S}\{s\}(p)$ holds afterwards. By a suitable definition of $\mathcal{W}$ and $\mathcal{S}$, these functions should be the best such functions in the logic considered. Figure 3.6 and Figure 3.7 describe respectively generic rules for defining weakest preconditions and strongest postconditions. Notice that the definition for the loop depends on an externally provided loop invariant I, that should hold at the loop's first entry (loop invariant initialization) and that should be maintained by each loop iteration (loop invariant preservation). The rules described here are not totally standard, in that they check loop invariant initialization and preservation together with other properties, by simply renaming variables $\overrightarrow{x_i}$ modified in the loop, a.k.a. inductive variables, with fresh variable names $\overrightarrow{y_i}$. This allows one to preserve information about variables that the loop does not modify, through the loop.

3.3.2 Application to JESSIE Integer Programs

The intraprocedural deductive verification of JESSIE programs can be defined as the verification of the validity of formulas computed by weakest preconditions. For the sake of simplicity, we only consider the case of a single intraprocedural exception X, but the generalization to more than one exception is obvious.

The weakest preconditions function $\mathcal{W}$ for instructions takes an instruction s and a proposition p in arguments, and returns the weakest precondition $\mathcal{W}\{s\}(p)$ that should hold before s so that p holds afterwards. Figure 3.8 defines $\mathcal{W}$ for instructions. All instructions that do not modify the value of integer variables are ignored. The rule for calls renames variables $\overrightarrow{x_i}$ modified in the function with fresh variable names $\overrightarrow{y_i}$.

The weakest preconditions function $\mathcal{W}$ for statements takes a statement s and three propositions in arguments, depending on the outcome of s, as defined in JESSIE semantics 2.2.4:

- p^N if the outcome is Normal;
- p^R if the outcome is Return;
- p^X if the outcome is Throw(X).

It returns the weakest precondition $\mathcal{W}\{s\}(p^N, p^R, p^X)$ that should hold before s so that the proper propositions hold afterwards. Figure 3.9 defines $\mathcal{W}$ for statements. Notice that, contrary to the rules for abstract interpretation, these rules do not provide an iterative procedure. This is fortunate, as there is no such thing as a widening in Hoare logics to force convergence. It is the presence of loop invariants and function pre- and postconditions that effectively cut the iterative propagation.

Taking type invariants and annotations (*e.g.*, preconditions) into account simply consists in treating them as tests in every state, or initially and after each assignment and function call. Checking that a proposition p holds consists in conjoining it to the weakest precondition formula when the computation described in Figure 3.8 and Figure 3.9 reaches the corresponding program point. Then, all checks in the function hold if the function precondition implies the validity of the weakest precondition propagated through the function's body.

3.3.3 Illustration on Linear Search

Computing the weakest precondition of unannotated function `linear_search` presented in Section 2.3.3 leads to formula

$$\begin{array}{rcl}
\mathcal{W}\{x := t\}(p) & \doteq & p[x \mapsto t] \\
\mathcal{W}\{s_1\ ;\ s_2\}(p) & \doteq & \mathcal{W}\{s_1\}(\mathcal{W}\{s_2\}(p)) \\
\mathcal{W}\{\text{if } t \text{ then } s_1 \text{ else } s_2\}(p) & \doteq & (t \Rightarrow \mathcal{W}\{s_1\}(p)) \wedge (\neg t \Rightarrow \mathcal{W}\{s_2\}(p)) \\
\mathcal{W}\{\text{while}_I\ t \text{ do } s\}(p) & \doteq & I \wedge (I \wedge t \Rightarrow \mathcal{W}\{s\}(I))[\overrightarrow{x_i \mapsto y_i}] \\
 & & \wedge (I \wedge \neg t \Rightarrow p)[\overrightarrow{x_i \mapsto y_i}]
\end{array}$$

Figure 3.6: Generic rules of Dijkstra's weakest preconditions

$$\begin{array}{rcl}
\mathcal{S}\{x := t\}(p) & \doteq & \exists y.p[x \mapsto y] \wedge x \equiv t[x \mapsto y] \\
\mathcal{S}\{s_1\ ;\ s_2\}(p) & \doteq & \mathcal{S}\{s_2\}(\mathcal{S}\{s_1\}(p)) \\
\mathcal{S}\{\text{if } t \text{ then } s_1 \text{ else } s_2\}(p) & \doteq & \mathcal{S}\{s_1\}(t \wedge p) \vee \mathcal{S}\{s_2\}(\neg t \wedge p) \\
\mathcal{S}\{\text{while}_I\ t \text{ do } s\}(p) & \doteq & I \wedge (\mathcal{S}\{s\}(I \wedge t) \Rightarrow I)[\overrightarrow{x_i \mapsto y_i}] \\
 & & \wedge (I \wedge \neg t \Rightarrow p)[\overrightarrow{x_i \mapsto y_i}]
\end{array}$$

Figure 3.7: Generic rules of Dijkstra's strongest postconditions

$$\begin{array}{rcl}
\mathcal{W}\{x := t\}(p) & \doteq & p[x \mapsto t] \\
\mathcal{W}\{t_1.m := t_2\}(p) & \doteq & p \\
\mathcal{W}\{x := \text{new } S[t]\}(p) & \doteq & p \\
\mathcal{W}\{\text{free } t\}(p) & \doteq & p \\
\mathcal{W}\{x := f(\overrightarrow{t_i})\}(p) & \doteq & \mathit{pre}_f \wedge (\mathit{post}_f \Rightarrow p)[\overrightarrow{x_i \mapsto y_i}][x \mapsto y][\mathit{result} \mapsto x]
\end{array}$$

Figure 3.8: Weakest preconditions over instructions

$$\begin{array}{rcl}
\mathcal{W}\{s_1\ s_2\}(p^N, p^R, p^X) & \doteq & \mathcal{W}\{s_1\}(\mathcal{W}\{s_2\}(p^N, p^R, p^X), p^R, p^X) \\
\mathcal{W}\{\text{if } t \text{ then } s_1 \text{ else } s_2\}(p^N, p^R, p^X) & \doteq & (t \Rightarrow \mathcal{W}\{s_1\}(p^N, p^R, p^X)) \\
 & & \wedge (\neg t \Rightarrow \mathcal{W}\{s_2\}(p^N, p^R, p^X)) \\
\mathcal{W}\{\text{loop invariant } I\ s\}(p^N, p^R, p^X) & \doteq & I \wedge (I \Rightarrow \mathcal{W}\{s\}(I, p^R, p^X))[\overrightarrow{x_i \mapsto y_i}] \\
\mathcal{W}\{\text{return } t\}(p^N, p^R, p^X) & \doteq & p^R[\mathit{result} \mapsto t] \\
\mathcal{W}\{\text{throw } X\}(p^N, p^R, p^X) & \doteq & p^X \\
\mathcal{W}\{\text{try } s_1 \text{ catch } X\ s_2\}(p^N, p^R, p^X)) & \doteq & \mathcal{W}\{s_1\}(p^N, p^R, \mathcal{W}\{s_2\}(p^N, p^R, p^X))
\end{array}$$

Figure 3.9: Weakest preconditions over statements

$$\texttt{idx} < \texttt{len} \Rightarrow C_{12},$$

where C_{12} is

$$C_{12} \doteq \texttt{INT\_MIN} \leq \texttt{idx} + 1 \leq \texttt{INT\_MAX}.$$

This formula obviously does not hold. Taking into account typed invariants leads to

$$(\texttt{INT\_MIN} \leq \texttt{idx} \leq \texttt{INT\_MAX} \wedge \texttt{idx} < \texttt{len} \wedge 0 \leq \texttt{len} \leq \texttt{UINT\_MAX}) \Rightarrow C_{12},$$

which does not prove C_{12} either.

Considering instead the weakest precondition of annotated function `linear_search` presented in Section 3.1.2 leads to formula

$$(\texttt{INT\_MIN} \leq \texttt{idx} \leq \texttt{INT\_MAX} \wedge \texttt{idx} < \texttt{len} \wedge 0 \leq \texttt{len} \leq \texttt{INT\_MAX}) \Rightarrow C_{12},$$

which is valid. Validity can be checked here using linear programming. In general, it requires calling an automatic prover or a proof assistant.

3.4 Other Related Work

Floyd-Hoare logics were formulated to allow axiomatic reasoning on programs [89, 72]. Dijkstra's computations allow to automate part of this reasoning [59].

Division of execution into normal/return/throw has been studied extensively [54, 7, 128, 110, 124, 95].

Some tools, like ASTRÉE [21], rely on abstract interpretation both to generate invariants and to perform safety checking, which stresses the precision of abstract domains. Other tools, like LOOP [18] and KeY [15], rely on the programmer to provide invariants. The combination of abstract interpretation followed by weakest preconditions generation is implemented in Boogie [11].

3.5 Chapter Summary

We presented integer safety checking of JESSIE annotated programs using either abstract interpretation or deductive verification.

First, checks that guarantee integer safety are generated, thus reducing integer safety checking to assertion checking. Then, an intraprocedural analysis based on logical annotations (function pre- and postconditions, loop invariants) is performed to prove that the generated checks hold.

Chapter 4

Memory Safety Checking

Contents

In this chapter, we extend the techniques for integer safety checking presented in Chapter 3 to tackle memory safety checking of annotated JESSIE programs, in an automatic and modular way.

Section 4.1 presents annotations that give access to the JESSIE memory model. Based on these annotations, we show how static safety checking for JESSIE programs reduces to assertion checking. In order to check these assertions, we assume all functions are completely annotated with pre- and postconditions and loop invariants.

Section 4.2 extends the notion of variable to memory locations, in order to be able to reason about the value of these abstract variables. Then, Sections 4.3 and 4.4 respectively describe the application of abstract interpretation and deductive verification to prove that those checks which guarantee the absence of memory errors hold.

4.1 Assertions for Memory Safety

4.1.1 Memory Model Accessors

Section 2.2.3 presented the byte-level block memory model of JESSIE. In Section 2.2.4, this memory model was used to define JESSIE semantics. In order to check memory safety, accessors to this memory model should be defined in the JESSIE language to allow writing specifications of memory properties.

Although the semantics of JESSIE is defined w.r.t. a byte-level memory model, it is more convenient to consider typed accessors to this memory model in specifications. This does not preclude the translation of these typed accessors as byte-level ones internally.

Natural Encoding A natural idea is to give direct accessors to the JESSIE memory model. In this model, a memory block is a tuple of a label, an address and an offset, with each label being associated with the size of the corresponding allocated block. This is roughly the approach adopted in HAVOC, VCC and Caduceus, where the following logical constructs are defined for a pointer t of type $S[..]$:

- $\mathit{base\text{-}addr}(t)$ is the label of the memory block pointed-to by t;
- $\mathit{offset}(t)$ is the typed offset of pointer t in its memory block;
- $\mathit{block\text{-}length}(t)$ is the size of the memory block pointed-to by t.

Figure 4.1 relates these logical constructs. A memory access $t.m$ is valid *iff* pointer t of type $S[..]$ is valid, which is expressed as

$$\mathit{valid}(t) \doteq 0 \leq \mathit{offset}(t) \wedge \mathit{offset}(t) < \mathit{block\text{-}length}(t).$$

Notice that offset and $\mathit{block\text{-}length}$ depend on the type of t, so that, in general

$$\mathit{offset}(t) \not\doteq \mathit{offset}(t \triangleright T[..]) \wedge \mathit{block\text{-}length}(t) \not\doteq \mathit{block\text{-}length}(t \triangleright T[..]),$$

where T and S have different sizes. Pointer arithmetic has no effect on $\mathit{base\text{-}addr}$ and $\mathit{block\text{-}length}$, and its effect on offset is described by

$$\mathit{offset}(t \oplus i) \doteq \mathit{offset}(t) + i.$$

Then, a memory access $(t \oplus i).m$ involving pointer arithmetic is valid *iff*

$$\mathit{valid}(t \oplus i) \doteq -i \leq \mathit{offset}(t) < \mathit{block\text{-}length}(t) - i.$$

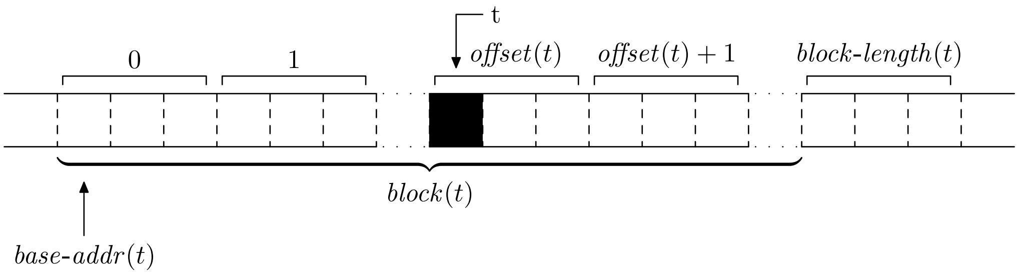

Figure 4.1: Natural accessors for the JESSIE memory model (t of type S[..], S of size 3)

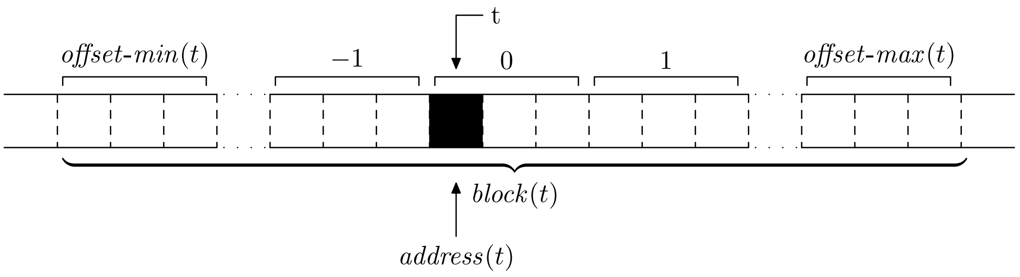

Figure 4.2: Local accessors for the JESSIE memory model (t of type S[..], S of size 3)

Thus, to prove that the upper bound is respected, which is the most likely to be violated, one must know the values of both $offset(t)$ and $block\text{-}length(t)$. The corresponding inequality has at least 3 variables (counting i), maybe more if i is a term with more than one variable, and arithmetic operations. As a result, generating automatically such propositions is costly. *E.g.*, generation of such an invariant by abstract interpretation requires using the domain of polyhedrons.

Local Encoding We devised a slightly different set of accessors to JESSIE memory model [136] for a pointer t of type $S[..]$:

- $base\text{-}block(t)$ is the label of the memory block pointed-to by t;
- $address(t)$ is the address pointed-to by t;
- $offset\text{-}min(t)$ is the (usually non-positive) minimal offset that can be added to pointer t to obtain a valid pointer $t \oplus i$;
- $offset\text{-}max(t)$ is the (usually non-negative) maximal offset that can be added to pointer t to obtain a valid pointer $t \oplus i$.

Figure 4.2 relates these logical constructs. A memory access $t.m$ is valid *iff*

$$offset\text{-}min(t) \leq 0 \leq offset\text{-}max(t).$$

Notice again that $\mathit{offset\text{-}min}$ and $\mathit{offset\text{-}max}$ depend on the type of t, so that, in general

$$\mathit{offset\text{-}min}(t) \not\equiv \mathit{offset\text{-}min}(t \triangleright T[..]) \wedge \mathit{offset\text{-}max}(t) \not\equiv \mathit{offset\text{-}max}(t \triangleright T[..]),$$

where T and S have different sizes. Pointer arithmetic has an effect on $\mathit{offset\text{-}min}$ and $\mathit{offset\text{-}max}$ described by

$$\begin{aligned}\mathit{offset\text{-}min}(t \oplus i) &\doteq \mathit{offset\text{-}min}(t) - i,\\ \mathit{offset\text{-}max}(t \oplus i) &\doteq \mathit{offset\text{-}max}(t) - i.\end{aligned}$$

Then, a memory access $(t \oplus i).m$ involving pointer arithmetic is valid *iff*

$$\mathit{offset\text{-}min}(t) \leq i \leq \mathit{offset\text{-}max}(t).$$

Thus, proving that the upper and lower bounds are respected is equally difficult. In particular, whenever the lower bounds holds trivially, which is quite common, the proof for the upper bound is easier than with the model of Caduceus, because it only involves 2 variables, when i is a variable. Many more abstract domains allow the generation of such propositions by abstract interpretation (*e.g.*, octagons), and automatic provers do a better job of proving it.

This encoding has another interesting locality property, not present in the natural encoding: it is possible to restrict the set of locations accessible through a pointer. *E.g.*, in the following JESSIE code, `q`, `r` and `s` are obtained from `p` by restricting the range of indices allowed.

```
1 p := new T[10] ;
2 q := p ▷ T[3..5] ;
3 r := p ▷ T[3..] ;
4 s := p ▷ T[..5]
```

This restriction can be reflected in the semantics of such casts, so that, after line 4, `offset_min(q)` and `offset_min(r)` are `3`, `offset_max(q)` and `offset_max(s)` are `5`. There is no way to do the same with the natural encoding, because it refers to total block length, that does not change (unless $\mathit{block\text{-}length}$ is reinterpreted so that it does not mean total block length anymore). Therefore, we call this new encoding the *local encoding* of block memory model, to distinguish it from the natural encoding of block memory model.

The natural encoding of block memory model is quite common. It is the model used in CSSV [62] and many other works since then [162, 157, 179, 43]. BOON [175] uses an even simpler version of it. Recently, others have started to use the same encoding as us [77].

We define convenient shorthands based on the local encoding:

$$\begin{aligned}\mathit{same\text{-}block}(x,y) &\doteq \mathit{base\text{-}block}(x) \equiv \mathit{base\text{-}block}(y)\\ \mathit{valid}(x) &\doteq \mathit{offset\text{-}min}(x) \leq 0 \leq \mathit{offset\text{-}max}(x)\\ \mathit{valid\text{-}index}(x,i) &\doteq \mathit{offset\text{-}min}(x) \leq i \leq \mathit{offset\text{-}max}(x)\\ \mathit{valid\text{-}range}(x,i,j) &\doteq \mathit{offset\text{-}min}(x) \leq i \wedge j \leq \mathit{offset\text{-}max}(x)\end{aligned}$$

```
term  ::=  ...
        |  base_block ( term )    memory block
        |  address ( term )       address
        |  offset_min ( term )    minimal offset
        |  offset_max ( term )    maximal offset
```

Figure 4.3: Grammar of JESSIE extended terms

Extended Annotation Language Memory model accessors are injected in the JESSIE annotation language in a straightforward way. Figure 4.3 presents the abstract syntax of JESSIE terms, extended with logical terms.

Since ACSL defines roughly the same memory model accessors, the translation of these logical constructs from ACSL to JESSIE is straightforward.

4.1.2 Memory Checks

In Section 2.2.4, we presented the semantics of erroneous executions of JESSIE programs, with examples of inference rules for erroneous executions. Then, it is possible to completely guard against erroneous executions in JESSIE by going through each one of these rules, and check that the conditions to trigger them do not arise. We only need to consider those rules that directly complement the rules for correct execution, as found in the semantics of terms presented in Figure 2.17, the semantics of instructions presented in Figure 2.20 and the semantics of statements presented in Figure 2.21, and not those rules that simply propagate the erroneous outcome. In each case, the condition for not triggering the rule for erroneous execution can be expressed as a check, *i.e.*, an assertion that guards against an erroneous execution.

Section 3.1.1 already presented the checks that guard against integer errors. It remains to show the checks that guard against memory errors. In the following, we reuse the variable names from each semantic rule considered.

In rule SUBPTR, pointers can be subtracted only when they point into the same memory block, which can be expressed as

$$\mathit{same\text{-}block}(t_1, t_2).$$

In rules FIELD and ASSIGN-FIELD, reading or writing the value of a field in memory is possible only if the underlying structure accessed is allocated, which can be expressed as

$$\mathit{valid}(t).$$

In rule NEW, allocation should be passed a non-negative argument, which can be expressed as

$$0 \leq t.$$

In rule FREE, deallocation should be called only on a pointer to the start of an allocated block, which can be expressed as

$$\mathit{offset\text{-}min}(t) \equiv 0 \wedge 0 \leq \mathit{offset\text{-}max}(t).$$

In rule LOW-PTR-CAST, casting a pointer to a pointer type with a lower bound is allowed only if

$$\mathit{offset\text{-}min}(t \triangleright S[..]) \le \mathit{min}.$$

In rule UP-PTR-CAST, casting a pointer to a pointer type with an upper bound is allowed only if

$$\mathit{max} \le \mathit{offset\text{-}max}(t \triangleright S[..]).$$

In rule BOUND-PTR-CAST, casting a pointer to a bounded pointer type is allowed only if

$$\mathit{offset\text{-}min}(t \triangleright S[..]) \le \mathit{min} \wedge \mathit{max} \le \mathit{offset\text{-}max}(t \triangleright S[..]).$$

Theorem 1 *A well-typed* JESSIE *program executes without any error (possibly not terminating), on an imaginary machine with an infinite memory, if integer checks and memory checks defined respectively in Sections 3.1.1 and 4.1.2 hold.*

Proof Sketch. Suppose all integer checks and memory checks defined in Sections 3.1.1 and 4.1.2 hold, according to some verification technique. By correction of the verification technique, the corresponding assertions are valid on all executions of the program. By definition of the original checks, all side-conditions of rules for correct execution described in JESSIE semantics are valid whenever a rule is applicable. Thus, rules for erroneous execution are never triggered, which is the same as saying that the program executes without any error. □

4.1.3 Memory Safety for Linear Search

Here is the program `linear_search` presented in Section 2.3.3, annotated in ACSL. It repeats annotations for integer safety already presented in Section 3.1.2, and it adds a precondition for memory safety.

```
/*@ requires len ≤ INT_MAX ∧ \valid_range(arr,0,len−1);
  @ ensures −1 ≤ \result < len;
  @ assigns \nothing;
  @*/
int linear_search(int arr[], unsigned int len, int key) {
  int idx = 0;
  //@ loop invariant 0 ≤ idx ≤ len;
  while (idx < len) {
    if (arr[idx] == key) {
      return idx; // key found
    }
    idx = idx + 1;
  }
  return −1; // key not found
}
```

Annotations are not the most precise possible, *e.g.*, they do not ensure that function `linear_search` terminates or relate the result of `linear_search` to the presence of the value searched in the array. Still, these annotations guarantee that executing `linear_search` does not lead to a runtime error, and that the value returned is within simple bounds. It translates to the following JESSIE program.

```
1  range int32 = −2147483648..2147483647
2  range uint32 = 0..4294967295
3
4  struct Int32 = { int32 int32m : 32 }
5
6  requires len ≤ INT_MAX
7           ∧ offset_min(arr) ≤ 0 ∧ len − 1 ≤ offset_max(arr)
8  ensures −1 ≤ result < len
9  assigns nothing
10 int32 linear_search(Int32[..] arr, uint32 len, int32 key) =
11   int32 idx
12   idx := 0
13   try
14     loop invariant 0 ≤ idx ≤ len
15       if (¬ (idx < len)) then
16         throw Break
17       else if ((arr ⊕ idx).int32m ≡ key) then
18         return idx
19       else
20         idx := (idx + 1) ▷ int32
21   catch Break
22     return −1
```

This program contains two checks at lines 17 and 20, plus a loop invariant at line 14, a postcondition at line 8 and a frame condition at line 9. The check at line 20 (line 12 in C) is the integer overflow check C_{12} already described in Section 3.1.2. At line 17 (line 9 in C), pointer `arr` should be accessed within bounds, which gives check C_9:

$$C_9 \doteq \mathit{offset\text{-}min}(\mathtt{arr}) \leq \mathtt{idx} \leq \mathit{offset\text{-}max}(\mathtt{arr})$$

A precondition is needed to ensure the safety of function `linear_search`. The precondition stated in line 7 (line 1 in C) requires from the calling context to pass in an array `arr` that can be safely dereferenced between its indices `0` and `len−1`. This condition effectively ensures that check C_9 holds.

Altogether, function `linear_search` is completely safe, as soon as it is called in a context specified by its precondition. It is easy to check it with specific arguments for the function `linear_search`, by applying the semantics rules given Section 2.2.4 and checking that they never block. We are going to present techniques for proving this for any set of arguments.

4.2 Abstract Variables

The most important issue to deal with in order to analyze a program is the choice of variables. Quite naturally, program variables translate into abstract variables. This is easy in JESSIE because neither global nor local variables can be modified through pointers, which makes named access the only way to modify the content of such a variable.

Following a common pattern too [62, 77], we choose to abstract integer memory model accessors associated with pointer program variables: if `x` is a program variable of type pointer, then $\mathit{address}(\mathtt{x})$, $\mathit{offset\text{-}min}(\mathtt{x})$ and $\mathit{offset\text{-}max}(\mathtt{x})$ are the pseudo-variables associated with `x`; $\mathit{base\text{-}block}(\mathtt{x})$ is not a pseudo-variable, because it does not have type integer.

These pseudo-variables associated with x can only be modified when x is assigned or the underlying memory is deallocated.

It remains to define abstract variables for memory locations and logical constructs attached to program entities.

4.2.1 Abstract Memory Locations

Summary Abstract Locations The tricky part is to abstract memory locations. There is an unbounded and possibly infinite number of memory locations, which should be abstracted to make the analysis tractable. Usually, when doing abstract interpretation over pointer programs, one associates abstract variables with *summary locations* (a.k.a. shrunk array cells in Astrée [21] or abstract memory references in C GLOBAL SURVEYOR [174]). A summary location represents a set of concrete locations, *e.g.*, the set of concrete locations corresponding to an array when performing abstract interpretation over arrays. This partitioning of concrete locations into summary locations can be either static or dynamic. Then, when a concrete operation like an assignment (resp. a test) modifies a concrete location (resp. constrains a concrete location), the corresponding abstract operation can be performed on the single summary location corresponding to the concrete location. The main advantage of summary locations is that they isolate the problem of alias analysis from other analyses.

The problem with this approach is that, in general, a summary location corresponds to more than one concrete location, so a concrete operation (on one concrete location) cannot make the value of a summary location more precise, only less precise. Assignment to a concrete location translates in the abstract world into a union between the previous abstract value and the abstract value assigned. *E.g.*, function ptrzero below nullifies the value pointed-to by x. Although its postcondition seems trivial, it is not possible to check it by abstract interpretation when using summary locations. If the concrete location pointed-to by x is abstracted into summary location α, assigning 0 to *x does not nullify the value associated with α, because α may represent more than one concrete location, and not all of them are nullified.

```
//@ ensures *x ≡ 0;
void ptrzero(int *x) {
  *x = 0;
}
```

Even worse, constraining the value of one concrete location in this set does not make the abstract value of the corresponding summary location more precise. *E.g.*, it makes it impossible to check the postcondition of function ptrabs below, which changes the value pointed-to by x into its absolute value. If the concrete location pointed-to by x is abstracted into summary location α, testing that *x is negative does not make the value associated with α more precise, because α may represent more than one concrete location, and not all of them are strictly negative.

```
//@ ensures *x ≥ 0;
void ptrabs(int *x) {
  if (*x < 0) {
    *x = -(*x);
  }
}
```

Access Path Abstract Locations Various works on context-sensitive pointer analyses refine summary locations into *access path locations* [58, 37, 38] or object names [113]. These access path locations describe concrete locations by how they are accessed from an initial variable. Hence, they are mostly local to a function. *E.g.*, `x→f` is not defined outside the current function when `x` is a function parameter, although the concrete memory locations it represents may not be local to the function. Contrary to the summary location approach where an lvalue may be associated with different abstract variables (depending on the program point, the control path followed, *etc.*), in the access path location approach, an lvalue is always associated with the same abstract variable. But the set of concrete locations represented by a syntactic location may vary (depending on the program point, the path followed, *etc.*).

Given some hypotheses on the calling context of the function, these access path locations can partition memory, like summary locations do. Hence, associating an abstract variable with these locations when analyzing a function correctly isolates the problem of alias analysis from other analyses. Like a summary location, an access path location may represent more than one concrete location for three different reasons:

- *aggregate abstraction* - An access path location may represent various locations at the same time, *e.g.*, all the cells in an array.
- *control path insensitivity* - An access path location may represent only one location at a time, but more than one location due to the merge of control paths in a function control-flow graph.
- *context insensitivity* - An access path location may represent only one location at a time, but more than one location due to the merge of calling contexts for a function.

Based on the local description of an access path location, it is easy to distinguish the case where context insensitivity is the reason why the access path location may represent more than one concrete location. In that case, this access path location can be considered as representing a single location during the intraprocedural analysis of a function, which allows precise treatment of the associated assignments and tests and greatly improves precision of the analysis. Testing a concrete location simply translates into testing an access path location, as in the summary location approach. Here, we additionally get the advantage that the associated abstract value gets constrained if the access path location represents a single location. Assigning a concrete location translates to assigning the access path location, with a strong update if the path location represents a single location.

E.g., in both examples above, we define an abstract variable $\alpha_{\texttt{*x}}$ that represents the set of concrete locations pointed-to by `x`. Although functions `ptrzero` and `ptrabs` may be called with pointer arguments to different concrete locations, the lvalue `*x` only represents one concrete location inside each function body. Therefore, by restricting the analysis to one function, it may be as precise on access path locations as on program variables. Indeed, with such path abstract variables, we can easily check the postconditions of `ptrzero` and `ptrabs` by abstract interpretation.

Syntactic Abstract Locations In this thesis, we make a different choice about abstract memory locations. We do not require that access path locations partition memory. This makes it possible to associate an abstract variable with every location mentioned in the program, not worrying about possible aliasing between them. We call these *syntactic abstract locations*. This choice does not completely separate alias analysis from other analyses, but it makes it possible to improve precision. *E.g.*, using access path locations, one cannot properly analyze function `ptrmax` below, because it can be called in a context where parameters `x` and `y` are aliased. Then, a single access path location $\alpha_{*\mathtt{x}}$ and $\alpha_{*\mathtt{y}}$ represents both. A context-sensitive analysis is required to improve on this situation, in order to consider separately contexts in which `x` and `y` are aliased, and contexts in which they are unaliased. With syntactic locations, we can prove the postcondition of function `ptrmax`.

```
//@ ensures *\result ≥ *x ∧ *\result ≥ *y;
int *ptrmax(int *x, int *y) {
  int *res = x;
  if (*y >= *x) {
    res = y;
  }
  return res;
}
```

Things are especially easy with this function, as it does not assign memory. On the path that successfully passes the test, the invariant $\alpha_{*\mathtt{y}} \geq \alpha_{*\mathtt{x}}$ holds, and assigning `y` to `res` translates in the abstract into assigning $\alpha_{*\mathtt{y}}$ to $\alpha_{*\mathtt{res}}$. Therefore, $\alpha_{*\mathtt{res}} \equiv \alpha_{*\mathtt{y}} \geq \alpha_{*\mathtt{x}}$ holds at the end of the function on the path through the test, while on the other path $\alpha_{*\mathtt{res}} \equiv \alpha_{*\mathtt{x}} > \alpha_{*\mathtt{y}}$ holds. Altogether, we can check the postcondition of `ptrmax` by abstract interpretation, using syntactic locations.

Assigning a concrete location is not as simple as in the summary and access path location approaches. Indeed, it does not translate into assigning a single summary location, because more than one syntactic location may represent the same concrete location. In function `sort` below, syntactic locations $\alpha_{*\mathtt{x}}$ and $\alpha_{*\mathtt{y}}$ may well represent the same location, if `x == y` at function beginning. Therefore, assigning to `*x` at line 5 should translate into an abstract assignment to $\alpha_{*\mathtt{x}}$ and a possible abstract assignment to $\alpha_{*\mathtt{y}}$. Adding `x != y` to the precondition does not solve the problem though. In our byte-level memory model, there is also the possibility that `x` and `y` are different but only a few bytes apart, so that assigning to `*x` assigns some bytes of the concrete location pointed-to by `y`. Therefore, without any more information on the locations pointed-to by `x` and `y`, the assignment to `*x` results in an abstract assignment to $\alpha_{*\mathtt{x}}$ and the loss of any information on $\alpha_{*\mathtt{y}}$. The converse occurs for the assignment to `*y` at line 6, which makes it impossible to check the postcondition of `sort`. We will address these problems of memory separation in Section 6.

```
//@ ensures *x ≤ *y;
void sort(int *x, int *y) {
  if (*y >= *x) {
    int tmp = *x;
    *x = *y;
    *y = tmp;
  }
}
```

Syntactic abstract locations are a simple and powerful solution to the aliasing problem in the context of intraprocedural analysis. This solution is much better than the *ad hoc* solutions presented sometimes [62], that are both partial (working on a very limited set of examples) and complex (associating various alias analyses). Syntactic abstract locations can be seen as a simplified version of the symbolic values used by Chang and Leino [36], without the congruence-closure domain and the repeated renamings and projections.

4.2.2 Abstract Logic Function Applications

In some cases, it is not sufficient to track the value of variables and memory locations to prove safety. *E.g.*, consider function `string_search` which is a variant of `linear_search` where the bound for search is not given explicitly as a parameter, but corresponds to the position of the implicit sentinel null character in the argument string `arr`.

```
/*@ ensures −1 ≤ \result;
  @ assigns \nothing;
  @*/
int string_search(char arr[], char key) {
  int idx = 0;
  //@ loop invariant 0 ≤ idx;
  while (arr[idx] != '\0') {
    if (arr[idx] == key) {
      return idx; // key found
    }
    idx = idx + 1;
  }
  return −1; // key not found
}
```

The precondition of `string_search` should express that it must be called in a context where its parameter `arr` can be safely dereferenced between indices `0` and `strlen(arr)`, which is the length of string `arr` as returned by a call to C standard library function `strlen`. Annotations cannot refer to the result of calling a C function. Instead, the functionality of `strlen` can be duplicated in the logic by defining a logic function *strlen*:

```
//@ logic integer strlen(char *s) reads s[0..];
```

Since logic function *strlen* cannot be defined with a body, it must be given a memory footprint, which assesses that it reads the memory pointed-to by its pointer argument, starting from the location pointed-to and upwards. Its functionality can be defined through a proper axiomatization. Notice *strlen* is a complete function, implicitly defined for all argument pointer values, although the value for non-string pointers is left unspecified (any negative integer would do, see Section 8.1.1).

Then, the following precondition expresses the desired constraint on valid contexts for safely calling `string_search`.

```
//@ requires 0 ≤ strlen(arr) ∧ \valid_range(arr,0,strlen(arr));
```

The memory safety of function `string_search` depends on the fact that index `idx` used to access `arr` remains bounded from above by *strlen*(`arr`), while *strlen*(`arr`) cannot be greater than `INT_MAX` by typing, which translates into loop invariant:

```
//@ loop invariant 0 ≤ idx ≤ strlen(arr) ≤ INT_MAX;
```

Therefore, it is crucial to track the value of *strlen*(`arr`). In general, it is necessary to track the value of *strlen*(`s`) for every string `s` to prove memory safety of string manipulating functions.

This motivates the definition of abstract values for logic function applications. Whether the logic function is defined with a body or a memory footprint, it is possible to compute the set of locations on which the result of the logic function depends. Any modification of the corresponding syntactic abstract variables leads to the loss of any information on the logic function application abstract variable. Still, it allows inferring invariants about the value of such logic function applications without requiring the development of a dedicated abstract domain. Thus, the value of such an abstract variable can only be maintained until the underlying memory locations mentioned in the reads clause get assigned, which causes the value of the abstract variable to be lost.

Like in expression abstraction [79], it could be possible to use inference rules to make the value of such abstract variables more precise, but it is best left to a prover to deal with the axiomatization of logic functions.

4.2.3 Overlaps Between Locations

As already mentioned in Section 4.2.1, syntactic abstract locations may alias, *i.e.*, they can represent overlapping concrete memory locations. To track such overlaps, we define a function *paths-may-overlap* on paths such that *paths-may-overlap*(π_1, π_2) conservatively over-approximates the property of overlap between the concrete locations represented by these paths. It is presented in Figure 4.4. It consists in a pattern matching over the structure of argument paths π_1 and π_2, with three different cases:

1. π_1 and π_2 represent the same variable x, in which case they do overlap;
2. π_1 and π_2 represent arbitrary memory locations (underscores are used as anonymous names), in which case they may overlap;
3. π_1 or π_2 represents a variable x, and the other one represents either a different variable or a memory location, in which case they do not overlap.

```
define paths-may-overlap:
  input paths $\pi_1$ and $\pi_2$
  output whether $\pi_1$ and $\pi_2$ represent overlapping locations
  match ($\pi_1$,$\pi_2$) with
  (1)    | (x,x) $\rightarrow$ return true
  (2)    | ((_$\oplus$_)._,(_$\oplus$_)._) $\rightarrow$ return true
  (3)    | (x,_) | (_,x) $\rightarrow$ return false
```

Figure 4.4: Overlapping of paths

```
define path-of-location:
  input location $\lambda$
  output path $\pi$ that over-approximates location $\lambda$
  match $\lambda$ with
  | x $\rightarrow$ return x
  | ($\lambda_1 \oplus [i^?..j^?]$).m $\rightarrow$ return (path-of-location($\lambda_1$) $\oplus [i^?..j^?]$).m
  | { $\lambda_1$ : _ } $\rightarrow$ return path-of-location($\lambda_1$)

define locations-may-overlap:
  input locations $\lambda_1$ and $\lambda_2$
  output whether $\lambda_1$ and $\lambda_2$ represent overlapping locations
  return paths-may-overlap(path-of-location $\lambda_1$, path-of-location $\lambda_2$)
```

Figure 4.5: Overlapping of locations

Function *paths-may-overlap* is the cornerstone of the memory analyses presented in Sections 4.3 and 4.4. Although its definition presented in Figure 4.4 is quite imprecise, we will show how to refine it given restrictions on the programs analyzed in Chapters 5, 6 and 7.

JESSIE locations can be over-approximated by paths, as described in Section 2.4. Function *path-of-location* defines this computation, which allows one to lift function *paths-may-overlap* to locations in function *locations-may-overlap*.

4.2.4 Application to Linear Search

According to our definition of abstract variables, there are 8 syntactic abstract variables in the JESSIE program `linear_search` presented in Section 2.3.3, and no abstract logic function application. To increase readability, we use C names for locations (*e.g.*, `arr[idx]`) instead of JESSIE names (*e.g.*, `(arr`$\oplus$`idx).int32_f`). These abstract variables can be grouped into

- 4 local variables: `len`, `key`, `idx`, `result`,
- 3 pseudo-variables: *address*(`arr`), *offset-min*(`arr`), *offset-max*(`arr`),
- and 1 syntactic abstract location: `arr[idx]`.

Since `arr` is the only pointer, there is no possible overlapping between abstract locations in `linear_search`.

4.3 Abstract Interpretation for Pointer Programs

Section 3.2 presented abstract interpretation for JESSIE integer programs. All information about memory was ignored in that analysis. Here, we present an extension of the analysis presented in Section 3.2 to handle JESSIE pointer programs, *i.e.*, arbitrary JESSIE programs with pointers.

In order to prove memory safety using abstract interpretation, memory checks should be represented exactly in the abstract domain, as mentioned in Section 3.2. As shown in Section 4.1, the checks generated for memory safety are mostly linear (in)equalities, which can be represented exactly in many relational domains.

4.3.1 Lifting Abstract Domains

As presented in Section 3.2, there exists many efficient domains for abstract interpretation over integer variables, both relational and non-relational ones. Take such a domain $\mathbf{D}$, defined by functions

- $\mathbf{D}$.test, the transfer function for test;
- $\mathbf{D}$.assign, the transfer function for assignment;
- $\mathbf{D}$.forget, the transfer function for reset;
- $\mathbf{D}$.union, the join operation;
- $\mathbf{D}$.included, the inclusion test;
- $\mathbf{D}$.widen, the widening operation;
- $\mathbf{D}$.lbound, the lower bound query;
- $\mathbf{D}$.ubound, the upper bound query.

When using summary abstract locations or path abstract locations, an assignment through path π in a JESSIE program directly translates to an abstract assignment of the corresponding summary abstract location α_π in $\mathbf{D}$. This is not the case anymore with syntactic abstract locations, because they may represent overlapping locations.

Instead, it is possible to define an abstract domain $\overline{\mathbf{D}}$ such that assigning through path π in a JESSIE program translates to an abstract assignment of the corresponding syntactic abstract location α_π in $\overline{\mathbf{D}}$. Abstract domain $\overline{\mathbf{D}}$ lifts abstract domain $\mathbf{D}$ to work with syntactic abstract locations. Testing, union, inclusion, widening and queries are the same on syntactic abstract variables as on regular abstract variables:

$$\overline{\mathbf{D}}.\mathsf{test} \doteq \mathbf{D}.\mathsf{test} \quad \overline{\mathbf{D}}.\mathsf{union} \doteq \mathbf{D}.\mathsf{union} \quad \overline{\mathbf{D}}.\mathsf{included} \doteq \mathbf{D}.\mathsf{included}$$
$$\overline{\mathbf{D}}.\mathsf{widen} \doteq \mathbf{D}.\mathsf{widen} \quad \overline{\mathbf{D}}.\mathsf{lbound} \doteq \mathbf{D}.\mathsf{lbound} \quad \overline{\mathbf{D}}.\mathsf{rbound} \doteq \mathbf{D}.\mathsf{rbound}$$

Assignment should take into account possible overlaps of locations, so that assigning through path π_1 forgets about the value of π_2 whenever π_1 and π_2 might overlap. If d is an abstract value from domain $\mathbf{D}$, π is a path assigned, α is the syntactic abstract location for π, α_i is the syntactic abstract location for some location π_i and v is a value assigned to π (*e.g.*, a linear combination of abstract variables), then assignment in $\overline{\mathbf{D}}$ of a value v supported by $\mathbf{D}$.assign is defined as

$$\begin{aligned}&\overline{\mathbf{D}}.\mathsf{assign}(d, \alpha, v) \doteq \\ &\quad \mathbf{D}.\mathsf{forget}(\mathbf{D}.\mathsf{assign}(d, \alpha, v), \{\alpha_i \text{ such that } \alpha_i \not\equiv \alpha \wedge \mathit{paths\text{-}may\text{-}overlap}(\pi, \pi_i)\})\end{aligned}$$

Likewise, assignment in $\overline{\mathbf{D}}$ of a value v not supported by $\mathbf{D}$.assign is defined as

$$\begin{aligned}&\overline{\mathbf{D}}.\mathsf{forget}(d, \alpha) \doteq \\ &\quad \mathbf{D}.\mathsf{forget}(\mathbf{D}.\mathsf{forget}(d, \alpha), \{\alpha_i \text{ such that } \alpha_i \not\equiv \alpha \wedge \mathit{paths\text{-}may\text{-}overlap}(\pi, \pi_i)\})\end{aligned}$$

With the definition of *paths-may-overlap* presented in Section 4.2.3, the precision of $\overline{\mathbf{D}}$.assign and $\overline{\mathbf{D}}$.forget is poor. Indeed, any assignment to memory leads to forgetting all memory information previously computed. Refinements of *paths-may-overlap* will improve on the precision of $\overline{\mathbf{D}}$.assign and $\overline{\mathbf{D}}$.forget.

In their work on symbolic values [36], Chang and Leino choose the opposite way of maintaining as much information as possible about the current state: they look for locations whose value does not change by maintaining a stack of heap updates in a special heap succession abstract domain. However, this special domain only seems to be able to handle assignments to the heap that occur in sequence, while losing all information on joins, *e.g.*, after an if-statement.

4.3.2 Application to JESSIE Pointer Programs

The intraprocedural abstract interpretation of JESSIE programs can be defined as a data-flow analysis on instructions and statements. Rules presented in Figures 4.6 and 4.7 adapt the rules for abstract interpretation of integer programs presented in Section 3.2. This time, in rule CALL, α_j denotes any abstract variable that overlaps with an abstract variable possibly modified by calling f. Rules ASSIGN-VAR and FORGET-VAR assign to abstract variables $\mathit{offset\text{-}min}(x)$ and $\mathit{offset\text{-}max}(x)$ only for those variables x of pointer type. Rules for statements are identical, except $\overline{\mathbf{D}}$ operations are called instead of $\mathbf{D}$ ones.

4.3.3 Illustration on Linear Search

It is now possible to apply abstract interpretation to check the memory safety of program `linear_search` presented in Section 4.1.3. We assume the abstract domain $\mathbf{D}$ chosen is the abstract domain of octagons.

The precondition of `linear_search` is a conjunction of inequalities between at most two variables, which can be exactly encoded into the weakly relational domain of octagons. It is the conjunction of the typing precondition

$$\mathit{typ} \doteq 0 \leq \texttt{len} \leq \texttt{UINT\_MAX} \wedge \texttt{INT\_MIN} \leq \texttt{key} \leq \texttt{INT\_MAX}$$

$$\frac{t\ \textit{treated by}\ \mathbf{D}.\mathsf{assign}}{\begin{array}{l}\{x := t\} \vdash \mathcal{A} \Rightarrow \\ \quad \overline{\mathbf{D}}.\mathsf{assign}(\overline{\mathbf{D}}.\mathsf{assign}(\overline{\mathbf{D}}.\mathsf{assign}(\mathcal{A}, \alpha_{\textit{offset-min}(x)}, \alpha_{\textit{offset-min}(t)}), \\ \qquad\qquad \alpha_{\textit{offset-max}(x)}, \alpha_{\textit{offset-max}(t)}), \alpha_x, t)\end{array}}\ \textsc{Assign-Var}$$

$$\frac{t\ \textit{not treated by}\ \mathbf{D}.\mathsf{assign}}{\begin{array}{l}\{x := t\} \vdash \mathcal{A} \Rightarrow \\ \quad \overline{\mathbf{D}}.\mathsf{forget}(\overline{\mathbf{D}}.\mathsf{assign}(\overline{\mathbf{D}}.\mathsf{assign}(\mathcal{A}, \alpha_{\textit{offset-min}(x)}, \alpha_{\textit{offset-min}(t)}), \\ \qquad\qquad \alpha_{\textit{offset-max}(x)}, \alpha_{\textit{offset-max}(t)}), \alpha_x)\end{array}}\ \textsc{Forget-Var}$$

$$\frac{t_2\ \textit{treated by}\ \mathbf{D}.\mathsf{assign}}{\{t_1.m := t_2\} \vdash \mathcal{A} \Rightarrow \overline{\mathbf{D}}.\mathsf{assign}(\mathcal{A}, \alpha_{t_1.m}, t_2)}\ \textsc{Assign-Field}$$

$$\frac{t_2\ \textit{not treated by}\ \mathbf{D}.\mathsf{assign}}{\{t_1.m := t_2\} \vdash \mathcal{A} \Rightarrow \overline{\mathbf{D}}.\mathsf{forget}(\mathcal{A}, \alpha_{t_1.m})}\ \textsc{Forget-Field}$$

$$\frac{t\ \textit{treated by}\ \mathbf{D}.\mathsf{assign}}{\begin{array}{l}\{x := \text{new } S[t]\} \vdash \mathcal{A} \Rightarrow \\ \quad \overline{\mathbf{D}}.\mathsf{assign}(\overline{\mathbf{D}}.\mathsf{assign}(\mathcal{A}, \alpha_{\textit{offset-min}(x)}, 0), \alpha_{\textit{offset-max}(x)}, t-1)\end{array}}\ \textsc{Assign-New}$$

$$\frac{t\ \textit{not treated by}\ \mathbf{D}.\mathsf{assign}}{\begin{array}{l}\{x := \text{new } S[t]\} \vdash \mathcal{A} \Rightarrow \\ \quad \overline{\mathbf{D}}.\mathsf{forget}(\overline{\mathbf{D}}.\mathsf{assign}(\mathcal{A}, \alpha_{\textit{offset-min}(x)}, 0), \alpha_{\textit{offset-max}(x)})\end{array}}\ \textsc{Forget-New}$$

$$\frac{}{\{\text{free } t\} \vdash \mathcal{A} \Rightarrow \overline{\mathbf{D}}.\mathsf{assign}(\overline{\mathbf{D}}.\mathsf{assign}(\mathcal{A}, \alpha_{\textit{offset-min}(t)}, 0), \alpha_{\textit{offset-max}(t)}, -1)}\ \textsc{Free}$$

$$\frac{}{\{x := f(\overrightarrow{t_i})\} \vdash \mathcal{A} \Rightarrow \overline{\mathbf{D}}.\mathsf{test}(\overline{\mathbf{D}}.\mathsf{forget}(\mathcal{A}, \overrightarrow{\alpha_j}), \textit{post}_f[\overrightarrow{\textit{param}_i \mapsto t_i}, \textit{result} \mapsto x])}\ \textsc{Call}$$

Figure 4.6: Intraprocedural abstract interpretation of instructions

$$\frac{\{s\} \vdash \mathcal{A}_1^N \Rightarrow \mathcal{A}_2^N}{\{s\} \vdash \mathcal{A}_1^N \Rightarrow (\mathcal{A}_2^N, \epsilon, \epsilon)} \text{ INSTR}$$

$$\frac{\{s_1\} \vdash \mathcal{A}_1^N \Rightarrow (\mathcal{A}_2^N, \mathcal{A}_2^R, \mathcal{A}_2^X) \quad \{s_2\} \vdash \mathcal{A}_2^N \Rightarrow (\mathcal{A}_3^N, \mathcal{A}_3^R, \mathcal{A}_3^X)}{\{s_1\ s_2\} \vdash \mathcal{A}_1^N \Rightarrow (\mathcal{A}_3^N, \overline{\mathbf{D}}.\mathsf{union}(\mathcal{A}_2^R, \mathcal{A}_3^R), \overline{\mathbf{D}}.\mathsf{union}(\mathcal{A}_2^X, \mathcal{A}_3^X))} \text{ SEQ}$$

$$\frac{\begin{array}{c}\{s_1\} \vdash \overline{\mathbf{D}}.\mathsf{test}(\mathcal{A}_1^N, t) \Rightarrow (\mathcal{A}_2^N, \mathcal{A}_2^R, \mathcal{A}_2^X) \\ \{s_2\} \vdash \overline{\mathbf{D}}.\mathsf{test}(\mathcal{A}_1^N, \neg t) \Rightarrow (\mathcal{A}_3^N, \mathcal{A}_3^R, \mathcal{A}_3^X)\end{array}}{\begin{array}{l}\{\text{if } t \text{ then } s_1 \text{ else } s_2\} \vdash \mathcal{A}_1^N \Rightarrow \\ \quad (\overline{\mathbf{D}}.\mathsf{union}(\mathcal{A}_2^N, \mathcal{A}_3^N), \overline{\mathbf{D}}.\mathsf{union}(\mathcal{A}_2^R, \mathcal{A}_3^R), \overline{\mathbf{D}}.\mathsf{union}(\mathcal{A}_2^X, \mathcal{A}_3^X))\end{array}} \text{ IF}$$

$$\frac{\begin{array}{c}\{s\} \vdash \mathcal{A}_1^N \Rightarrow (\mathcal{A}_2^N, \mathcal{A}_2^R, \mathcal{A}_2^X) \\ \{\text{loop } s\} \vdash \overline{\mathbf{D}}.\mathsf{union}(\mathcal{A}_1^N, \mathcal{A}_2^N) \Rightarrow (\mathcal{A}_3^N, \mathcal{A}_3^R, \mathcal{A}_3^X)\end{array}}{\{\text{loop } s\} \vdash \mathcal{A}_1^N \Rightarrow (\mathcal{A}_3^N, \mathcal{A}_3^R, \mathcal{A}_3^X)} \text{ LOOP-UNROLL}$$

$$\frac{\begin{array}{c}\{s\} \vdash \mathcal{A}_1^N \Rightarrow (\mathcal{A}_2^N, \mathcal{A}_2^R, \mathcal{A}_2^X) \\ \overline{\mathbf{D}}.\mathsf{included}(\mathcal{A}_2^N, \mathcal{A}_1^N)\end{array}}{\{\text{loop } s\} \vdash \mathcal{A}_1^N \Rightarrow (\mathcal{A}_1^N, \mathcal{A}_2^R, \mathcal{A}_2^X)} \text{ LOOP-CONVERGE}$$

$$\frac{}{\{\text{return } t\} \vdash \mathcal{A}_1^N \Rightarrow (\epsilon, \mathcal{A}_1^N, \epsilon)} \text{ RETURN}$$

$$\frac{}{\{\text{throw } X\} \vdash \mathcal{A}_1^N \Rightarrow (\epsilon, \epsilon, \mathcal{A}_1^N)} \text{ THROW}$$

$$\frac{\{s_1\} \vdash \mathcal{A}_1^N \Rightarrow (\mathcal{A}_2^N, \mathcal{A}_2^R, \mathcal{A}_2^X) \quad \{s_2\} \vdash \mathcal{A}_2^X \Rightarrow (\mathcal{A}_3^N, \mathcal{A}_3^R, \mathcal{A}_3^X)}{\{\text{try } s_1 \text{ catch } X\ s_2\} \vdash \mathcal{A}_1^N \Rightarrow (\overline{\mathbf{D}}.\mathsf{union}(\mathcal{A}_2^N, \mathcal{A}_3^N), \overline{\mathbf{D}}.\mathsf{union}(\mathcal{A}_2^R, \mathcal{A}_3^R), \mathcal{A}_3^X)} \text{ TRY}$$

Figure 4.7: Intraprocedural abstract interpretation of statements

and the annotated precondition

$$annot \doteq \textit{offset-min}(\texttt{arr}) \leq 0 \wedge \texttt{len} - 1 \leq \textit{offset-max}(\texttt{arr}) \wedge \texttt{len} \leq \texttt{INT\_MAX}.$$

Thus, the complete precondition for `linear_search` is

$$pre \doteq typ \wedge annot.$$

The invariant computed by abstract interpretation for the loop at line 8 of `linear_search` is

$$inv_8 \doteq pre \wedge 0 \leq \texttt{idx} \leq \texttt{len}.$$

This invariant implies trivially that loop invariant $0 \leq \texttt{idx} \leq \texttt{len}$ holds. Then, the invariant computed at line 9 where `arr` is accessed is almost the same, with the additional information that $\texttt{idx} < \texttt{len}$, which gives

$$inv_9 \doteq pre \wedge 0 \leq \texttt{idx} < \texttt{len}.$$

This invariant trivially implies that check C_9 holds. Notice that the manually annotated loop invariant was never necessary here. On the contrary, abstract interpretation allows us to infer this invariant.

4.4 Deductive Verification for Pointer Programs

Section 3.3 presented deductive verification for JESSIE integer programs. All information about memory was ignored in this analysis. Here, we present an extension of the analysis presented in Section 3.3 to handle JESSIE pointer programs, *i.e.*, arbitrary JESSIE programs with pointers.

4.4.1 Lifting Weakest Preconditions

The usual way to define weakest preconditions for pointer programs is to make memory explicit as one (or as a set of) additional variable(s), and rely on the axioms of the theory of arrays to model rules for memory accesses. Then, one needs a theorem prover to check the validity of the generated formulas or to simplify them (*e.g.*, by eliminating quantifiers). This is what we do in the Why Platform, when we translate a JESSIE program into WHY [68, 69].

Given the low-level memory model for JESSIE defined in Section 2.2.3, any two pointers may point to overlapping memory locations. Thus, a single heap variable $Heap$ is defined. Writes and reads to memory location $t_1.m$ are encoded by applications of logic functions $select_m$ and $update_m$, for each field m:

- $select_m$ takes a heap variable $Heap$ and a pointer x as arguments, and it returns the value of field m in the structure pointed-to by x stored in $Heap$;
- $update_m$ takes a heap variable $Heap$, a pointer x and a value v as arguments, and it returns a modified $Heap$ such that the value of field m in the structure pointed-to by x stored in the returned heap is now v.

$$\begin{array}{rcl}
\mathcal{W}\{x := t\}(p) & \doteq & p[x \mapsto t] \\
\mathcal{W}\{t_1.m := t_2\}(p) & \doteq & p[\mathit{Heap} \mapsto \mathit{update}_m(\mathit{Heap}, t_1, t_2)] \\
\mathcal{W}\{x := \text{new } S[t]\}(p) & \doteq & p[\mathit{Alloc} \mapsto \\
 & & \quad \mathit{update}(\mathit{Alloc}, \mathit{base\text{-}block}(x), \mathit{sizeof}(S) \times t)][x \mapsto y] \\
\mathcal{W}\{\text{free } t\}(p) & \doteq & p[\mathit{Alloc} \mapsto \mathit{update}(\mathit{Alloc}, \mathit{base\text{-}block}(t), -1)] \\
\mathcal{W}\{x := f(\overrightarrow{t_i})\}(p) & \doteq & \mathit{pre}_f \wedge (\mathit{post}_f \Rightarrow p)[\overrightarrow{x_i \mapsto y_i}] \\
 & & \qquad [\mathit{Heap} \mapsto \mathit{Heap}'][x \mapsto y][\mathit{result} \mapsto x]
\end{array}$$

Figure 4.8: Weakest preconditions over instructions

Those functions are only declared in WHY. Their definition is provided through a proper axiomatization of the theory of arrays. Here, only one of the classical axioms of the theory of arrays makes sense, since we cannot express non-overlap between memory locations yet:

$$\mathit{select}_m(\mathit{update}_m(\mathit{Heap}, x, v), x) \equiv v$$

It states that writing value v in field m of the structure pointed-to by pointer x and later on reading this field gives back the value v.

Likewise, a single allocation variable Alloc is defined. Allocations and deallocations perform writes on this allocation variable, while accesses perform reads, to check that only valid pointers are accessed. These writes and reads are encoded by applications of logic functions select and update, similarly to what is done for accesses to Heap.

Figure 4.8 redefines $\mathcal{W}$ for instructions w.r.t. the definition given in Section 3.3. If calling f modifies the heap, Heap variable gets renamed. The definition for statements stays the same.

The problem with this translation is that the generated formulas cannot be easily understood by other analyses working at the level of JESSIE programs, as it mentions memory explicitly, whereas memory is implicit at the level of JESSIE programs, which is better for the kind of analyses we want to perform. Therefore, we define lightweight preconditions in which memory remains implicit. These preconditions no longer correspond to the "best" or weakest preconditions. Rather, they lift the weakest preconditions presented in Section 3.3 to work with syntactic abstract variables. For the sake of simplicity though, we will still refer to these as our weakest preconditions on JESSIE programs.

It requires that the substitution rule is modified. Given a path π and α the syntactic abstract location for π, a set of paths π_i that may overlap with π (*i.e.*, $\mathit{paths\text{-}may\text{-}overlap}(\pi, \pi_i)$ returns true) and α_i the corresponding syntactic abstract locations, we redefine the substitution

$$p[\alpha \mapsto t],$$

as first renaming all syntactic abstract locations α_i before substituting t for α. This has the effect of first generalizing the formula being propagated over all possible values for possibly

$$\begin{array}{rcl}
\mathcal{W}\{x := t\}(p) & \doteq & p[\alpha_x \mapsto t] \\
\mathcal{W}\{t_1.m := t_2\}(p) & \doteq & p[\alpha_{t_1.m} \mapsto t_2] \\
\mathcal{W}\{x := \text{new } S[t]\}(p) & \doteq & p[\mathit{offset\text{-}min}(x) \mapsto 0][\mathit{offset\text{-}max}(x) \mapsto t-1][x \mapsto y] \\
\mathcal{W}\{\text{free } t\}(p) & \doteq & p[\mathit{offset\text{-}min}(t) \mapsto 0] \\
 & & \quad [\mathit{offset\text{-}max}(t) \mapsto \ -1][\overrightarrow{\alpha_i \mapsto \alpha_j}] \\
\mathcal{W}\{x := f(\overrightarrow{t_i})\}(p) & \doteq & \mathit{pre}_f \wedge (\mathit{post}_f \Rightarrow p)[\overrightarrow{\alpha_i \mapsto \alpha_j}][x \mapsto y][\mathit{result} \mapsto x]
\end{array}$$

Figure 4.9: Preconditions over instructions

overlapping syntactic abstract locations. Say the formula p propagated is $\alpha \equiv 0 \wedge \alpha_1 \equiv 1$, where α and α_1 correspond to possibly overlapping paths. While the simple substitution rule for computing $p[\alpha \mapsto t]$ returns formula $t \equiv 0 \wedge \alpha_1 \equiv 1$, our modified substitution rule returns here $\forall x.t \equiv 0 \wedge x \equiv 1$.

4.4.2 Application to JESSIE Pointer Programs

Given the modified substitution rule, it is possible to adapt the weakest preconditions function $\mathcal{W}$ for instructions defined in Section 3.3. Figure 4.9 redefines $\mathcal{W}$ for instructions. Function $\mathcal{W}$ for statements defined in Section 3.3 stays the same.

4.4.3 Illustration on Linear Search

It is now possible to apply deductive verification to check the memory safety of program `linear_search` presented in Section 4.1.3. Starting from check C_9, we get the following weakest precondition at line 9

$$wp_9 \doteq C_9.$$

After passing the loop entry test, we get weakest precondition

$$wp_8 \doteq \texttt{idx} < \texttt{len} \Longrightarrow wp_9,$$

which becomes the following after passing the loop invariant and universally quantifying over variables modified in the loop

$$wp_7 \doteq \forall\, \texttt{idx}.\ 0 \leq \texttt{idx} \leq \texttt{len} \Longrightarrow wp_8.$$

Initialization of `idx` simply gives

$$wp_6 \doteq \texttt{idx} \equiv 0 \Longrightarrow wp_7,$$

thus leading to the weakest precondition at function beginning

$$wp_5 \doteq \mathit{pre} \Longrightarrow wp_6.$$

Formula wp_5 is valid, which proves that check C_9 holds at line 9. Notice that the presence of a loop invariant was essential in proving that check C_9 holds. This is different from what we observed for abstract interpretation.

4.5 Chapter Summary

We presented memory safety checking of JESSIE annotated programs using either abstract interpretation or deductive verification.

First, the JESSIE annotation language is extended with constructs that give a handle on the JESSIE memory model. Then, checks that guarantee memory safety are generated, based on this extended annotation language, thus reducing memory safety checking to assertion checking.

Secondly, abstract variables are defined to track the value of memory locations and logic function applications. In particular, we define a new kind of abstraction for memory locations, syntactic abstract locations. Overlapping between locations is made explicit through function *paths-may-overlap*.

Finally, we show how to lift abstract interpretation and deductive verification to work with possibly overlapping syntactic abstract locations, based on function *paths-may-overlap*.

Part I devoted to automatic and modular safety checking of annotated JESSIE programs is thus complete. Part II will focus on the generation of annotations for unannotated JESSIE programs.

Part II

Inference, Separation, Unions and Casts

Chapter 5

Alias-Free Type-Safe Programs

Contents

In this chapter, we describe how to generate automatically and modularly annotations for alias-free type-safe JESSIE programs, so that they can be checked safe as in Chapter 4.

Section 5.1 presents the type safety and aliasing restrictions on input programs. Given these restrictions, the checking techniques for annotated programs presented in Chapter 4 can be refined.

Section 5.2 presents known techniques, either based on abstract interpretation or deductive verification, to infer function pre- and postconditions, as well as loop invariants. Their relative strengths and limitations are compared. We show that these techniques are not sufficient, even when used together, to generate sufficient preconditions on typical C pointer programs.

Section 5.3 presents a new combination of abstract interpretation and deductive verification that builds on their respective strengths. This combination improves on current techniques. It generates satisfying sufficient preconditions on typical C pointer programs.

Finally, Section 5.4 shows how this new technique relates to other work.

5.1 Problem Overview

5.1.1 Type Safety Restriction

Type safety is the property that each byte of data is always interpreted as the same byte in the same type. In C, type safety can be simply obtained by forbidding the use of unions and pointer casts, as well as pointer arithmetic. More precisely, pointer arithmetic can be allowed in type-safe programs as long as the resulting pointer is within the bounds of the underlying array when it is dereferenced. This is similar to the restriction on source C programs in Caduceus [68].

Not only should the C program part analyzed be free from unions, pointer casts and unbounded pointer arithmetic, but the overall program where this program part belongs should be free from unions, pointer casts and unbounded pointer arithmetic. This makes the JESSIE program analyzed type-safe, as if originally written in a type-safe language instead of C. An important consequence of type safety is that two memory accesses through non-embedded fields `m` and `n` in JESSIE do not interfere if fields `m` and `n` are different. Our translation ensures that accesses to embedded fields are only used as intermediate steps in some non-embedded field access.

Figures 5.1 and 5.2 present refined operational semantics rules w.r.t. those presented in Section 2.2.4. These rules additionally guarantee type safety. A pointer now evaluates to a tuple (l, a, i, min, max) with (l, a, i) as before a block label, an address for the block and a byte offset into the block, and (min, max) the current minimal and maximal offsets allowed, as in the local encoding of the block memory model presented in Section 4.1.1.

Component-as-array Memory Model In the context of type-safe programs, the byte-level block memory model of JESSIE can be refined successively into a type-level block memory model, where blocks of memory remain typed from allocation to deallocation, and then into the component-as-array memory model of Burstall [30]. In this memory model, blocks of memory are logically divided into collections of memory chunks that store the same field of the same structure type, for different pointers. Type safety guarantees that accesses into different collections cannot interfere, because these collections represent different fields.

This is the memory model used in many works on programs semantics and verification [55, 90, 173, 118] as well as many verification tools: ESC/Modula-3 [117], ESC/Java [70], Caduceus [68], Krakatoa [129], Jack [13], *etc.* Figure 5.3 shows such collections, where all memory chunks in the same collection share the same color.

$$\frac{}{\llbracket null \rrbracket = (null, 0, 0, 0, -1)}\ \text{NULL}$$

$$\frac{t_1 : S[..] \quad \llbracket t_1 \rrbracket = (l, a, i, min, max) \quad \llbracket t_2 \rrbracket = j}{\llbracket t_1 \oplus t_2 \rrbracket = (l, a, i + j \times \mathit{sizeof}(S), min\text{-}j, max\text{-}j)}\ \text{SHIFT}$$

$$\frac{t_1 : S[..] \quad t_2 : S[..] \quad \llbracket t_1 \rrbracket = (l, a, i, \_, \_) \quad \llbracket t_2 \rrbracket = (l, a, j, \_, \_)}{\llbracket t_1 \ominus t_2 \rrbracket = (i - j)/\mathit{sizeof}(S)}\ \text{SUBPTR}$$

$$\frac{\llbracket t_1 \rrbracket = (l_1, a_1, i, \_, \_) \quad \llbracket t_2 \rrbracket = (l_2, a_2, j, \_, \_)}{\llbracket t_1 \odot t_2 \rrbracket = a_1 + i\ \overline{\odot}\ a_2 + j}\ \text{COMPAR-PTR} \quad \odot \in \{\varolessthan, \varogreaterthan, \equiv, \not\equiv\}$$

$$\frac{\llbracket t \rrbracket = (l, a, i, \_, \_) \quad m\ \mathit{is\ embedded} \quad \mathit{typeof}(m) = S[min..max]}{\llbracket t.m \rrbracket = (l, a, i + \mathit{offsetof}(m), min, max)}\ \text{EMBED-FIELD}$$

$$\frac{\begin{array}{c}\llbracket t \rrbracket = (l, a, i, min, max) \quad min \le 0 \le max \\ 0 \le i \quad i + \mathit{sizeof}(t) \le \mathbf{Alloc}(l) \quad m\ \mathit{is\ not\ embedded}\end{array}}{\llbracket t.m \rrbracket = \mathit{of\text{-}bits}_m(\mathbf{Heap}((a + i) \times 8 + \mathit{bitoffsetof}(m), \mathit{bitsizeof}(m)))}\ \text{FIELD}$$

$$\frac{t : S[..] \quad \llbracket t \rrbracket = (l, a, i, min_1, max_1)}{\llbracket t \triangleright S[..] \rrbracket = (l, a, i, min_1, max_1)}\ \text{PTR-CAST}$$

$$\frac{\begin{array}{c}t : S[..] \quad \llbracket t \rrbracket = (l, a, i, min_1, max_1) \quad min_1 \le min_2 \\ 0 \le i + min_2 \times \mathit{sizeof}(S)\end{array}}{\llbracket t \triangleright S[min_2..] \rrbracket = (l, a, i, min_2, max_1)}\ \text{LOW-PTR-CAST}$$

$$\frac{\begin{array}{c}t : S[..] \quad \llbracket t \rrbracket = (l, a, i, min_1, max_1) \quad max_2 \le max_1 \\ i + (max_2 + 1) \times \mathit{sizeof}(S) \le \mathbf{Alloc}(l)\end{array}}{\llbracket t \triangleright S[..max_2] \rrbracket = (l, a, i, min_1, max_2)}\ \text{UP-PTR-CAST}$$

$$\frac{\begin{array}{c}t : S[..] \quad \llbracket t \rrbracket = (l, a, i, min_1, max_1) \\ min_1 \le min_2 \quad max_2 \le max_1 \\ 0 \le i + min_2 \times \mathit{sizeof}(S) \\ i + (max_2 + 1) \times \mathit{sizeof}(S) \le \mathbf{Alloc}(l)\end{array}}{\llbracket t \triangleright S[min_2..max_2] \rrbracket = (l, a, i, min_2, max_2)}\ \text{BOUND-PTR-CAST}$$

Figure 5.1: Type-safe evaluation of JESSIE terms

$$\frac{\begin{array}{c} t_1 : S[..] \quad [\![t_1]\!] = (l, a, i, \mathit{min}, \mathit{max}) \quad \mathit{min} \le 0 \le \mathit{max} \\ [\![t_2]\!] = v \quad 0 \le i \quad i + \mathit{sizeof}(S) \le \textsf{\textbf{Alloc}}(l) \end{array}}{\begin{array}{l} \{t_1.m := t_2\} \vdash \textsf{\textbf{Heap}} \Rightarrow \\ \quad \textsf{\textbf{Heap}}[((a+i) \times 8 + \mathit{bitoffsetof}(m), \mathit{bitsizeof}(m)) \mapsto \mathit{to\text{-}bits}_m(v)] \end{array}} \ \textsc{Assign-Field}$$

$$\frac{\begin{array}{c} [\![t]\!] = n \quad 0 \le n \quad l \notin \mathit{dom}(\textsf{\textbf{Alloc}}) \\ \forall\, i.\, 0 \le i < n \times \mathit{sizeof}(S) \rightarrow a + i \ \mathit{not\ allocated} \end{array}}{\begin{array}{l} \{x := \text{new } S[t]\} \vdash \textsf{\textbf{Env}} \Rightarrow \\ \quad \textsf{\textbf{Env}}[x \mapsto (l, a, 0, 0, n\text{-}1)], \textsf{\textbf{Alloc}} \Rightarrow \textsf{\textbf{Alloc}}[l \mapsto n \times \mathit{sizeof}(S)] \end{array}} \ \textsc{New}$$

$$\frac{[\![t]\!] = (l, a, 0, \_, \_) \quad 0 \le \textsf{\textbf{Alloc}}(l)}{\{\text{free } t\} \vdash \textsf{\textbf{Alloc}} \Rightarrow \textsf{\textbf{Alloc}}[l \mapsto -1]} \ \textsc{Free}$$

Figure 5.2: Type-safe semantics of JESSIE instructions

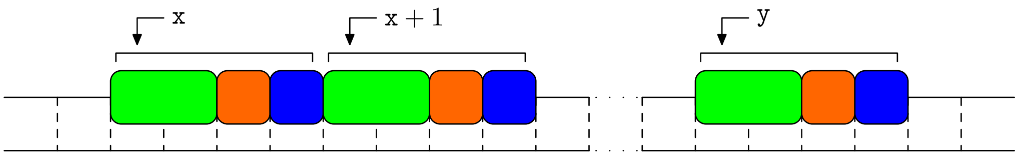

Figure 5.3: Component-as-array memory model

```
define paths-may-overlap:
  input paths $\pi_1$ and $\pi_2$
  output whether $\pi_1$ and $\pi_2$ represent overlapping locations
  match ($\pi_1$,$\pi_2$) with
  (1)      | (x,x) → return true
  (2.1)    | ((_⊕_).m,(_⊕_).m) → return true
  (2.2)    | ((_⊕_)._,(_⊕_)._) → return false
  (3)      | (x,_) | (_,x) → return false
```

Figure 5.4: Overlaping of paths for type-safe programs

Reduced Overlap Between Locations Type safety positively impacts the precision of checking techniques, as it guarantees that fewer memory accesses may interfere. This directly translates into an improvement over the naive definition of *paths-may-overlap* given in Section 4.2.3. This new definition is presented in Figure 5.4. Previously, case (2) concluded that two memory locations could always overlap. It now refines into cases (2.1) and (2.2) which conclude that two memory locations can overlap only if they access the same field m.

Improved JESSIE Analyses Abstract interpretation and deductive verification for JESSIE programs as presented in Sections 4.3 and 4.4 cannot check safety and annotations in function `type_safe` below. With the hypothesis that `type_safe` is part of a type-safe program, and the refined *paths-may-overlap* algorithm presented in Figure 5.4, locations `x→m` and `y→n` are known not to overlap. Thus, both abstract interpretation and deductive verification can now check safety and annotations in function `type_safe`.

```
struct S { int m; };
struct T { int n; };

/*@ requires \valid(x) ∧ \valid(y) ∧ y→n ≡ 1;
  @ ensures x→m ≡ 0;
  @*/
void type_safe(struct S *x, struct T *y) {
  x→m = 0;
  //@ assert y→n ≡ 1;
}
```

Improved Translation to WHY Likewise, the translation of JESSIE into WHY benefits from the type safety restriction. Global variable *Heap* as shown in Section 2.5 can be replaced by a collection of $Heap_m$ variables, one for each field m, which naturally encodes the absence of interference between accesses to the corresponding fields. Thus, separation of fields is directly encoded in the generated verification conditions, which greatly simplifies the proofs.

```
unit type_safe(pointer x, pointer y, heap Heap_m, heap Heap_n) =
  update_m(Heap_m,x,0)
  assert select_n(Heap_n,y) == 1
```

5.1.2 Aliasing Restriction

In a type-safe context, overlap of locations reduces to aliasing of pointers, without the need to observe at low-level whether two locations may partially overlap. In a language with arrays like C (and JESSIE), there is no point in completely banning aliasing: array locations `arr[i]` and `arr[j]` in C (or `(arr⊕i).m` and `(arr⊕j).m` in JESSIE) are aliases whenever `i ≡ j`.

Definition 1 *Alias-free programs are programs which contain no aliasing between syntactically different paths except aliasing due to equality of array indices.*

```
define paths-may-overlap:
  input paths $\pi_1$ and $\pi_2$
  output whether $\pi_1$ and $\pi_2$ represent overlapping locations
  match ($\pi_1$,$\pi_2$) with
  (1)     | (x,x) → return true
  (2.1')  | (($\pi_3$⊕_).m,($\pi_4$⊕_).m) → return paths-may-overlap($\pi_3$,$\pi_4$)
  (2.2)   | ((_⊕_)._,(_⊕_)._) → return false
  (3)     | (x,_) | (_,x) → return false
```

Figure 5.5: Overlaping of paths for alias-free programs

Reduced Overlap Between Locations Like type safety, restricting aliasing guarantees that fewer memory accesses may interfere. Again, this translates into an improved definition of *paths-may-overlap* presented in Figure 5.5 w.r.t. function *paths-may-overlap* presented in Section 5.1.1 for type-safe programs. Previously, case (2.1) concluded that two memory locations accessing the same field could always overlap. It now refines into case (2.1') which concludes that two memory locations accessing the same field can overlap only if the two paths obtained by ignoring the last pointer arithmetic and field access overlap themselves.

Improved JESSIE Analyses With the hypothesis that function `alias_free` below is part of an alias-free program, and the refined *paths-may-overlap* algorithm presented above, both abstract interpretation and deductive verification can check safety and annotations in function `alias_free`, which is not possible otherwise.

```
struct S { int m; };

/*@ requires \valid(x) ∧ \valid(y) ∧ y→m ≡ 1;
  @ ensures x→m ≡ 0;
  @*/
void alias_free(struct S *x, struct S *y) {
  x→m = 0;
  //@ assert y→m ≡ 1;
}
```

Unlike the type safety restriction, the aliasing restriction is not reflected in the translation from JESSIE to WHY.

5.1.3 Without Logic Annotations

First, we consider checking the safety of program `linear_search` introduced in Section 2.3.3 without any logic annotation. There are two checks in this function: a buffer overflow check C_4 on line 4:

$$C_4 \doteq \mathit{offset\text{-}min}(\texttt{arr}) \leq \texttt{idx} \leq \mathit{offset\text{-}max}(\texttt{arr}),$$

and an integer overflow check C_7 on line 7:

$$C_7 \doteq \texttt{INT\_MIN} \leq \texttt{idx} + 1 \leq \texttt{INT\_MAX}.$$

Without additional information about this program, it is not possible to be confident that these checks will succeed on every run. In fact, it is possible to find inputs to this program that make any of those two checks false. Taking null for `arr` and 1 for `len` makes C_4 false the first time execution reaches it. Taking `INT_MAX` $+ 2$ for `len` makes C_7 false after `INT_MAX` iterations through the loop. In a real execution, where integers follow an overflow semantics, `idx` then becomes negative, in fact it takes the least possible value `INT_MIN`, which leads to a buffer index bounds error in the iteration of the loop that follows.

Therefore, however simple this program is, it contains two serious vulnerabilities that could lead to a buffer overflow, thus making the whole surrounding program unsafe. A common practice to remove such vulnerabilities is to program defensively, by checking dynamically the validity of operations before performing them. It can be done by the programmer or a safe compiler like CCured. *E.g.*, it is possible here to defend against violations of the second check, by testing whether `idx` belongs to the range where increment does not overflow, that is `idx` $<$ `INT_MAX`. However, it impacts the efficiency of the program, possibly for no benefit if the check can be shown to be valid in all calling contexts. For the first check, such a dynamic monitoring is not even possible from the user perspective, as the C language does not provide constructs like $\mathit{offset\text{-}min}$ and $\mathit{offset\text{-}max}$. It is possible though with a safe compiler like CCured which automatically inserts guards before pointer dereferences, based on extra variables added to keep track of pointer offsets. From a verification perspective, the solution to these problems is to add annotations to the program that specify which calling contexts are allowed, thus making all checks valid.

5.1.4 Problem Statement

In this chapter, we consider the problem of generating logic annotations for type-safe alias-free programs. As seen in Sections 5.1.1 and 5.1.2, type safety and aliasing restrictions allow us to improve the results of abstract interpretation and deductive verification in checking safety and annotations of JESSIE programs. Hopefully, the same hypotheses should help in generating annotations.

Our main goal is to generate, in a modular and automatic way, a sufficient precondition and a necessary postcondition for each function. The precondition inferred should be sufficient to prove safety and annotations in the function, in particular the generated postcondition. The postcondition inferred should be as precise as possible, given the precondition. Of course, we should seek the weakest possible precondition and the strongest possible postcondition, as *false* is otherwise a valid precondition and *true* a valid postcondition.

When a function body contains loops, we should also generate inductive loop invariants implied by the context given by the function precondition. The crucial point here is that the loop invariant should be inductive, *i.e.*, provably true either by abstract interpretation or deductive verification.

5.2 Inferring Logic Annotations

```
define AbsInterp:
  input program P
  output invariants I and logical annotations for P
  compute invariants I by forward abstract interpretation
  use $I_{Loop}$ to strengthen loop invariants in P
  use $I_{Post}$ to strengthen P postcondition
done
```

Figure 5.6: Algorithm AbsInterp

5.2.1 Approach by Abstraction

Postcondition and Loop Invariant Generation Abstract interpretation of a program builds an over-approximation of a program semantics. At every program point, abstract interpretation returns an invariant that over-approximates the set of possible states reached during execution at this program point. Therefore, abstract interpretation naturally computes loop invariants and function postconditions. We call AbsInterp this classical application of forward abstract interpretation to generate invariants and annotations, that we present in Figure 5.6. *E.g.*, on program `linear_search`, abstract interpretation produces loop invariant:

$$0 \leq \texttt{idx} \leq \texttt{len}, \tag{5.1}$$

and postcondition:

$$0 \leq \mathit{result} < \texttt{len} \vee \mathit{result} \equiv -1. \tag{5.2}$$

More precisely, abstract interpretation computes invariant $I_{C_4} \doteq 0 \leq \texttt{idx} < \texttt{len}$ at check C_4 and invariant $I_{C_7} \doteq 0 \leq \texttt{idx} < \texttt{len} \wedge \texttt{arr[idx]} \not\equiv \texttt{key}$ at check C_7. Then, the corresponding check is proved if the invariant implies it. Check C_4 is proved if the following formula holds:

$$\underbrace{0 \leq \texttt{idx} < \texttt{len}}_{I_{C_4}} \implies \underbrace{\textit{offset-min}(\texttt{arr}) \leq \texttt{idx} \leq \textit{offset-max}(\texttt{arr})}_{C_4}$$

This is not the case, therefore C_4 is not proved. C_7 is proved if the following formula holds:

$$\underbrace{0 \leq \texttt{idx} < \texttt{len}}_{I_{C_7}} \implies \underbrace{\texttt{INT\_MIN} \leq \texttt{idx} + 1 \leq \texttt{INT\_MAX}}_{C_7}$$

This is not the case either, therefore C_7 is not proved, but the first part of check C_7 alone is proved, since the following is a valid implication:

$$\underbrace{0 \leq \texttt{idx} < \texttt{len}}_{I_{C_7}} \implies \underbrace{\texttt{INT\_MIN} \leq \texttt{idx} + 1}_{C_7\ 1^{st}\ conjunct}$$

The fact C_4 and C_7 cannot be proved by forward abstract interpretation is not surprising, as we saw in Section 5.1.3 that a precondition should be added to `linear_search` to prevent both overflows.

Precondition Generation Unfortunately, abstract interpretation does not lead naturally to the generation of preconditions. Bourdoncle introduced abstract debugging [26] as a technique based on abstract interpretation that computes preconditions. It propagates an over-approximation of the program state *backwards* through the program control-flow graph. Abstract debugging only considers those paths that may lead to a specific program point of interest, typically a check point. This allows us to perform some kind of trace partitioning as defined by Mauborgne [130], which amounts to the generation of disjunctive invariants, where each disjunct represents an invariant for a set of traces (execution paths through the program). To that end, abstract debugging requires that the results of a forward abstract interpretation pass are available.

For program `linear_search`, the forward invariant at check C_4 is $0 \leq \texttt{idx} < \texttt{len}$. Abstract debugging starts with strengthening this invariant with check C_4, which gives invariant

$$0 \leq \texttt{idx} < \texttt{len} \land \textit{offset-min}(\texttt{arr}) \leq \texttt{idx} \leq \textit{offset-max}(\texttt{arr}).$$

Notice `len` and $\textit{offset-max}(\texttt{arr})$ get similar roles in this formula, although they have quite opposite responsibilities w.r.t. the assertion holding or not. Backward propagating this invariant through the loop leads to invariant

$$0 \leq \texttt{idx} < \texttt{len} \land \texttt{idx} \leq \textit{offset-max}(\texttt{arr})$$

at loop beginning. Indeed, the *offset-min* lower bound on `idx` is lost when performing widening, due to assignment to `idx` on line 7. If we keep working with a usual convex domain that cannot treat disjunctions, the path that never enters the loop forces us to lose all information at loop entry. Still propagating the formula backward, if we allow some kind of trace partitioning based on loop entrance, we get invariant

$$\texttt{idx} \equiv 0 \land (\texttt{len} \equiv 0 \lor (0 < \texttt{len} \land 0 \leq \textit{offset-max}(\texttt{arr})))$$

at loop entry. This finally leads to precondition

$$\texttt{len} \equiv 0 \lor (0 < \texttt{len} \land 0 \leq \textit{offset-max}(\texttt{arr})). \tag{5.3}$$

A very similar propagation performed from check C_7 leads to some stronger precondition.

This precondition does not guard against any of the possible overflows in `linear_search`. The problem is that backward abstract interpretation computes an over-approximation of program states. While forward abstract interpretation computes necessary postconditions, which is fine, backward abstract interpretation computes necessary preconditions, which is not enough. In very special cases, where forward invariants are very precise, and with a user specifying control points that every execution should reach (a.k.a. *intermittent assertions* [26]), the technique of abstract debugging generates sufficient preconditions for checking assertions.

Rival introduced alarm diagnosis [153], a variant of abstract debugging which propagates backwards sets of forbidden states instead of sets of desirable states. By taking the negation of all such undesirable preconditions found, we may reach a sufficient precondition. On `linear_search`, we can start with forbidden states

$$\underbrace{0 \leq \texttt{idx} < \texttt{len}}_{I_{C_4}} \land \underbrace{\texttt{idx} < \textit{offset-min}(\texttt{arr})}_{\neg C_4}$$

at check C_4. This leads to the same forbidden set at loop beginning, and finally to

$$0 < \texttt{len} \land 0 < \textit{offset-min}(\texttt{arr})$$

at function entry. By negating it, we get sufficient precondition

$$\texttt{len} \leq 0 \lor \textit{offset-min}(\texttt{arr}) \leq 0. \tag{5.4}$$

that guards against underflow at check C_4. Unfortunately, starting from $\textit{offset-max}(\texttt{arr}) < \texttt{idx}$ at check C_4 or $\texttt{INT\_MAX} < \texttt{idx} + 1$ at check C_7 leads to the overly conservative precondition $\texttt{len} \leq 0$. We do not consider this precondition in the following.

5.2.2 Approach by Deduction

Pre- and Postcondition Generation Techniques based on Dijkstra's weakest preconditions or strongest postconditions rely on preexisting loop invariants. *E.g.*, functions `sequence` and `looping` below are identical except for line 4 in both. In `sequence`, line 4 simply adds `x` to local variable `z`. Absence of loops in `sequence` makes it easy to derive function precondition $\texttt{x} \not\equiv \texttt{y}$ by weakest preconditions from check $\texttt{z} \not\equiv 0$ on line 5. Likewise, postcondition $\textit{result} \equiv 1$ is derivable by strongest postconditions from function beginning.

```
1 //@ requires 0 ≤ x ∧ 0 ≤ y;
2 int sequence(int x, int y) {
3   int z = −y;
4   z += x;
5   return (x − y)/z;
6 }
```

In `looping`, on the contrary, nothing is known on the value of `z` after the loop, which prevents the generation of any useful precondition and postcondition.

```
1 //@ requires 0 ≤ x ∧ 0 ≤ y;
2 int looping(int x, int y) {
3   int z = −y;
4   for (int i = 0; i < x; i++) z++;
5   return (x − y)/z;
6 }
```

By adding a sufficiently precise loop invariant, we can compute for `looping` the same precondition and postcondition as for `sequence`.

```
//@ loop invariant 0 ≤ i ≤ x ∧ z ≡ i − y;
```

Indeed, in that case, the result of weakest preconditions is

$$\exists\, i_1, i_2, z_2.\, \underbrace{0 \le x \wedge 0 \le y}_{precondition} \wedge \underbrace{z_1 \equiv \text{-}y \wedge i_1 \equiv 0}_{assignments} \wedge \underbrace{0 \le i_2 \le x \wedge z_2 \equiv i_2 - y}_{loop\ invariant} \wedge \underbrace{x \le i_2}_{loop\ exit} \implies \underbrace{z_2 \not\equiv 0}_{check},$$

which, together with the hand-written precondition, is equivalent to

$$0 \le x \wedge 0 \le y \wedge x \not\equiv y$$

by mechanical elimination of quantifiers. The same loop invariant makes it possible to infer postcondition $result \equiv 1$.

Going back to our running example `linear_search`, both checks happen to occur inside a loop, which makes the generation of a suitable precondition by weakest preconditions very dependent on the associated loop invariant. Without any loop invariant, the weakest precondition of check C_4 is

$$\exists\, \mathtt{idx}_1.\, \forall\, \mathtt{idx}_2.\ \mathit{typ} \wedge \underbrace{\mathtt{idx}_1 \equiv 0}_{assignment} \wedge \underbrace{\mathtt{idx}_2 < \mathtt{len}}_{loop\ test} \implies \underbrace{\mathit{offset\text{-}min}(\mathtt{arr}) \le \mathtt{idx}_2 \le \mathit{offset\text{-}max}(\mathtt{arr})}_{C_1}$$

where typ is the invariant given by typing:

$$\begin{aligned} & 0 \le \mathtt{len} \le \mathtt{UINT\_MAX} \wedge \mathtt{INT\_MIN} \le \mathtt{key} \le \mathtt{INT\_MAX} \\ \wedge\ & \mathtt{INT\_MIN} \le \mathtt{idx}_1 \le \mathtt{INT\_MAX} \wedge \mathtt{INT\_MIN} \le \mathtt{idx}_2 \le \mathtt{INT\_MAX}. \end{aligned}$$

After quantifier elimination, this precondition is equivalent to

$$\mathit{false},$$

but, by considering one conjunct at a time, for the second conjunct in check C_4, we get instead

$$\exists\, \mathtt{idx}_1.\, \forall\, \mathtt{idx}_2.\ \mathit{typ} \wedge \underbrace{\mathtt{idx}_1 \equiv 0}_{assignment} \wedge \underbrace{\mathtt{idx}_2 < \mathtt{len}}_{loop\ test} \implies \underbrace{\mathtt{idx}_2 \le \mathit{offset\text{-}max}(\mathtt{arr})}_{C_1\ 2^{nd}\ conjunct},$$

which is equivalent to a useful precondition

$$\mathtt{len} \le \mathit{offset\text{-}max}(\mathtt{arr}) + 1.$$

Likewise, starting from check C_7, we can compute the weakest precondition

$$\exists\, \mathtt{idx}_1.\, \forall\, \mathtt{idx}_2.\ \mathit{typ} \wedge \underbrace{\mathtt{idx}_1 \equiv 0}_{assignment} \wedge \underbrace{\mathtt{idx}_2 < \mathtt{len}}_{loop\ test} \implies \underbrace{\mathtt{INT\_MIN} \le \mathtt{idx}_2 + 1 \le \mathtt{INT\_MAX}}_{C_2},$$

which is equivalent to

$$\mathtt{len} \le \mathtt{INT\_MAX}.$$

Overall, we obtain the following precondition by weakest preconditions:

$$\texttt{len} \leq \textit{offset-max}(\texttt{arr}) + 1 \wedge \texttt{len} \leq \texttt{INT\_MAX}. \tag{5.5}$$

This proves the absence of overflow from above in both C_4 and C_7, but not the absence of overflow from below, a.k.a. underflow.

The postcondition obtained by strongest postconditions is simply

$$\textit{result} < \texttt{len} \vee \textit{result} \equiv\ -1 \tag{5.6}$$

Loop Invariant Generation Induction-iteration by Suzuki and Ishihata [166, 181] is a technique that generates loop invariants by repeated applications of weakest preconditions. The basic idea is to strengthen repeatedly a candidate loop invariant by computing its weakest precondition through the loop body, until an inductive invariant is found. For our `linear_search` example, the following formula obtained by weakest preconditions from C_4 is a candidate loop invariant:

$$W_0 \doteq \underbrace{\texttt{idx}_2 < \texttt{len}}_{\textit{loop test}} \Longrightarrow \underbrace{\textit{offset-min}(\texttt{arr}) \leq \texttt{idx}_2 \leq \textit{offset-max}(\texttt{arr})}_{C_4}.$$

To check whether W_0 is an inductive invariant, one computes its weakest precondition through the loop body, denoted W_1:

$$\begin{aligned} W_1 \quad \doteq \quad & \underbrace{\texttt{idx}_2 < \texttt{len}}_{\textit{loop test}} \wedge \underbrace{\texttt{idx}_2 + 1 < \texttt{len}}_{\textit{loop test}} \\ & \wedge \underbrace{\textit{offset-min}(\texttt{arr}) \leq \texttt{idx}_2 \leq \textit{offset-max}(\texttt{arr})}_{C_4}. \\ & \Longrightarrow \underbrace{\textit{offset-min}(\texttt{arr}) \leq \texttt{idx}_2 + 1 \leq \textit{offset-max}(\texttt{arr})}_{C_4}. \end{aligned}$$

W_0 is an inductive invariant *iff* the following formula holds:

$$W_0 \Longrightarrow W_1. \tag{5.7}$$

This is not the case. Then, the process can be repeated from the stronger candidate invariant

$$W_0 \wedge W_1,$$

and possibly next $W_0 \wedge W_1 \wedge W_2$, *etc*. In our example, no such finite conjunct is inductive, therefore strengthening does not terminate. In that case, induction-iteration resorts to generalization, which further strengthens the invariant by eliminating all variables modified in the loop, a.k.a. inductive variables. Applying generalization right away on W_0 gives

$$\textit{false},$$

but, by considering one conjunct at a time, the same computation of W_0 on the second conjunct in C_4 followed by generalization results in formula

$$\texttt{len} \leq \textit{offset-max}(\texttt{arr}) + 1,$$

which is not only an inductive invariant, but also a precondition that guards against an overflow (but not an underflow) at check C_4.

Here, induction-iteration with generalization is reduced to plain weakest preconditions with quantifier elimination. Theoretically though, induction-iteration could generate an inductive invariant that still mentions inductive variables, which is not possible with plain weakest preconditions. The first case that shows this situation is when induction-iteration reaches an inductive invariant without the need for generalization, as in functions `induction1` and `induction2` below.

```c
void induction1(int x, int y)
{
  while (1) {
    //@ assert x < y;
    x = x - 1;
  }
}
```

```c
void induction2(int x, int y)
{
  while (1) {
    if (x > 0) {
      //@ assert x < y;
    }
    x = -x;
  }
}
```

In function `induction1`, Formula 5.7 becomes

$$x < y \Longrightarrow x - 1 < y,$$

which always holds. Therefore, $x < y$ is an inductive loop invariant for function `induction1`. In function `induction2`, Formula 5.7 becomes

$$(x > 0 \Longrightarrow x < y) \Longrightarrow ((x > 0 \Longrightarrow x < y) \wedge (\text{-}x > 0 \Longrightarrow \text{-}x < y)),$$

which does not always hold, but $W_0 \wedge W_1$ in this case is

$$(x > 0 \Longrightarrow x < y) \wedge (\text{-}x > 0 \Longrightarrow \text{-}x < y),$$

which is indeed an inductive invariant, as shown by the validity of

$$\begin{aligned} &((x > 0 \Longrightarrow x < y) \wedge (\text{-}x > 0 \Longrightarrow \text{-}x < y)) \\ &\Longrightarrow ((\text{-}x > 0 \Longrightarrow \text{-}x < y) \wedge (x > 0 \Longrightarrow x < y)), \end{aligned}$$

which trivially holds.

The second case where induction-iteration outperforms weakest preconditions plus quantifier elimination is when induction-iteration reaches an inductive invariant by eliminating only some inductive variables, but not all. This is the case in function `induction3` below.

```
void induction3(int x, int y, int z) {
  while (x < z) {
    //@ assert x < y;
    x = x + 2;
    y = y + 1;
  }
}
```

In function `induction3`, Formula 5.7 becomes

$$(\mathtt{x} < \mathtt{z} \Longrightarrow \mathtt{x} < \mathtt{y}) \Longrightarrow (\mathtt{x} < \mathtt{z} \wedge \mathtt{x} < \mathtt{y} \wedge \mathtt{x} + 2 < \mathtt{z} \Longrightarrow \mathtt{x} + 2 < \mathtt{y} + 1)$$

This formula is not inductive, and more generally, no finite conjunct of W_i is inductive. Therefore, one must perform generalization at one stage. Generalizing W_0 on x results in

$$\mathtt{z} \leq \mathtt{y},$$

which is indeed an inductive loop invariant, as shown by the validity of

$$\mathtt{z} \leq \mathtt{y} \Longrightarrow (\mathtt{x} \leq \mathtt{z} \wedge \mathtt{x} < \mathtt{y} \Longrightarrow \mathtt{z} \leq \mathtt{y} + 1).$$

Notice that the same technique can be applied to generate loop invariants from strongest postconditions, starting from an initial candidate loop invariant obtained by weakest preconditions.

5.2.3 Abstraction and Deduction Together

As shown in Sections 5.2.1 and 5.2.2, both approaches by abstraction and deduction allow us to generate logic annotations: (i) abstract interpretation naturally generates loop invariants and postconditions; (ii) abstract debugging, a technique based on abstract interpretation, generates necessary preconditions; (iii) alarm diagnosis, a variant of abstract debugging, generates sufficient preconditions; (iv) weakest preconditions and strongest postconditions naturally generate the most precise pre- and postconditions, provided loop invariants are provided, and (v) induction-iteration, a technique based on weakest preconditions, generates inductive loop invariants.

Putting it all together, it seems that the conjunction of results of each technique could succeed in generating all the annotations needed. For our `linear_search` example, the loop invariant given by Formula 5.1 is

$$\underbrace{0 \leq \mathtt{idx} \leq \mathtt{len}}_{(i)}, \tag{5.8}$$

the conjunction of preconditions given by Formulas 5.3, 5.4 and 5.5 is

$$\begin{aligned} & \underbrace{(\mathtt{len} \leq 0 \vee (0 < \mathtt{len} \wedge 0 \leq \mathit{offset\text{-}max}(\mathtt{arr})))}_{(ii)} \\ \wedge\ & \underbrace{(\mathtt{len} \leq 0 \vee \mathit{offset\text{-}min}(\mathtt{arr}) \leq 0)}_{(iii)} \\ \wedge\ & \underbrace{\mathtt{len} \leq \mathit{offset\text{-}max}(\mathtt{arr}) + 1 \wedge \mathtt{len} \leq \mathtt{INT\_MAX}}_{(iv)+(v)}, \end{aligned} \tag{5.9}$$

and the conjunction of postconditions given by Formulas 5.2 and 5.6 is

$$\underbrace{(0 \leq \mathit{result} < \texttt{len} \vee \mathit{result} \equiv -1)}_{(i)} \wedge \underbrace{(\mathit{result} < \texttt{len} \vee \mathit{result} \equiv -1)}_{(iv)} . \tag{5.10}$$

Once rewritten, these are almost exactly the same as the hand-written annotations added to `linear_search` in Section 4.1.3. In particular, they guarantee the safety of calls to `linear_search`. To see this, notice first that Formula 5.8 is the same as the hand-written loop invariant. Secondly, in conjunction with typing precondition *typ*, Formula 5.9 is the same as the hand-written precondition. Only Formula 5.10 is slightly weaker than the hand-written postcondition, but this has no effect on the safety of calls to `linear_search`.

The generated precondition could be weaker though. Indeed, both checks C_4 and C_7 occur in the loop, which means that executions that do not enter the loop need not guard against C_4 and C_7. This corresponds to executions where $\texttt{len} \leq 0$, the first conjunct in the preconditions obtained by abstract debugging and alarm diagnosis. Thus, the following is a weaker sufficient precondition than Formula 5.9:

$$\texttt{len} \leq 0 \vee (\underbrace{\mathit{offset\text{-}min}(\texttt{arr}) \leq 0}_{\mathit{underflow}} < \underbrace{\texttt{len} \leq \mathit{offset\text{-}max}(\texttt{arr}) + 1 \wedge \texttt{len} \leq \texttt{INT\_MAX}}_{\mathit{overflow}}). \tag{5.11}$$

We have marked the part of the precondition that guards against underflow and the one that guards against overflow. The latter could be generated in this weakened form, by applying the same weakest preconditions technique we presented to a modified `linear_search` program where the loop is unrolled once:

```
int linear_search(int arr[], unsigned int len, int key) {
  int idx = 0;
  if (idx < len) {
    if (arr[idx] == key)
      return idx; // key found
    idx = idx + 1;
    while (idx < len) {
      if (arr[idx] == key)
        return idx; // key found
      idx = idx + 1;
    }
  }
  return −1; // key not found
}
```

5.3 Combining Abstraction and Deduction

It is possible that putting together the results of abstraction techniques and deduction techniques leads to a satisfactory set of generated logic annotations, like on our simple `linear_search` example. Notice though the many twists we had to perform to make it work for such a simple example: we had to discard overly conservative preconditions generated by alarm diagnosis, and we performed loop unrolling to improve the precision of weakest preconditions. More generally, it is not easy to recognize and select the best result

```
1  define AbsGeneric:
2    input program P
3    output logical annotations for P
4    compute invariants I by InvGen
5    for each check C do
6      define $\phi_C$ as C weakened by $I_C$: $\phi_C = I_C \Longrightarrow C$
7      define $\phi$ as the result of applying PreCond to $\phi_C$
8      define $\psi$ as the result of applying QuantElim to $\phi$
9      use $\psi$ to strengthen P precondition
10   done
```

Figure 5.7: Algorithm AbsGeneric

in case a technique outperforms another, and adding each technique's results to the combination does not solve the limitations of each one. On one hand, abstraction techniques poorly handle disjunctions and under-approximations, thus generating overly liberal preconditions. On the other hand, deduction techniques poorly handle over-approximations, thus generating imprecise loop invariants.

5.3.1 Precondition Inference Algorithm

An efficient combination of abstraction and deduction would rely on abstraction for over-approximations, and on deduction for under-approximations and disjunctions. Therefore, the forward propagation should better be left to abstraction and the backward propagation to deduction. We are going to present such an algorithm for inferring sufficient function preconditions.

A Plugin Architecture We suppose we are given an algorithm InvGen for computing invariants, *e.g.*, by forward abstract interpretation, an algorithm PreCond for computing preconditions, *e.g.*, by weakest preconditions and an algorithm QuantElim for computing stronger (possibly equivalent) quantifier-free formulas, *e.g.*, by (equivalence-preserving) quantifier elimination. Figure 5.7 presents our inference algorithm AbsGeneric, which is parametric over InvGen, PreCond and QuantElim.

Algorithm AbsGeneric starts with computing invariants on line 4. These invariants should be used to strengthen loop invariants and function postcondition.

Then, AbsGeneric treats each check C in turn, to generate a sufficient function precondition for C to hold. First, C is weakened by the invariant I_C computed at the same program point. Indeed, since I_C is known to hold, proving C is equivalent to proving $I_C \Longrightarrow C$, as shown by the validity of Formula 5.12:

$$I_C \Longrightarrow (C \Longleftrightarrow (I_C \Longrightarrow C)). \tag{5.12}$$

Finally, AbsGeneric computes a precondition ϕ of this formula on line 7. Since our goal is to generate a usable precondition, that can be easily taken into account by analyses, we would rather not leave quantifiers in this formula. Therefore, AbsGeneric eliminates quantifiers from ϕ on line 8, returning a stronger (possibly equivalent) formula ψ without quantifiers. Since it is stronger, this quantifier-free precondition still implies that C holds.

ABSGENERIC consists in a special arrangement of existing well-established techniques in program analysis: invariant generation, preconditions, quantifier elimination. This plugin architecture allows both reuse of existing building blocks and specialization to specific assertions and programs.

We use the same algorithm to generate loop invariants on the way, by treating loop beginnings like function starts. The generated loop invariant is sufficient to prove the property of interest, but it is not guaranteed to be inductive, thus it should always be proved.

Application to JESSIE Programs Sections 4.3 and 4.4 describe the application of abstract interpretation and weakest preconditions to JESSIE programs, so that it remains to define a quantifier elimination procedure for formulas obtained by weakest preconditions on JESSIE programs.

According to the rules presented in Sections 4.3.2 and 4.4.2, and for most checks presented in Sections 3.1.1 and 4.1.2, if the underlying abstract domain $\mathbf{D}$ chosen generates invariants in the form of conjunctions or disjunctions of linear (in)equalities of integer variables, which is the case for most useful abstract domains, and if the program manipulates linear combinations of variables, then the formula generated fits in the theory of Presburger arithmetic, *i.e.*, arithmetic without multiplication between variables. Then, any quantifier elimination for Presburger arithmetic can be used: Cooper's method, Omega test, Fourier-Motzkin method. The popular and efficient Simplex method cannot be used because it does not apply to a formula where some variables are not quantified.

Quantifier elimination for Presburger arithmetic is inherently triply exponential [176], which is far too expensive in practice, as we checked it in our experiments with Cooper's method. Instead, we turn to quantifier elimination for rational (or real) linear arithmetic, for which there exists algorithms with a doubly exponential complexity [85, 140, 28]. In fact, it has been proved that this problem has at least exponential complexity, but no algorithm has a better complexity than doubly exponential.

Classically, we exploit the particular form of the formulas generated by weakest preconditions. Our quantified formulas are universally quantified prenex formulas, meaning only universal quantifiers $\forall$ appear in front of a quantifier-free formula. Then, quantifier elimination over the rationals only returns a stronger formula than quantifier elimination over the integers, which is a correct behavior for QUANTELIM.

In practice, we rewrite the universal formula $\forall\overrightarrow{x}.\phi$ into the equivalent $\neg\exists\overrightarrow{x}.\neg\phi$ and we transform formula $\neg\phi$ into its disjunctive normal form (DNF), so that the existential quantifier distributes over all disjuncts and the Fourier-Motzkin method to eliminate quantifiers can be applied individually to each disjunct. The conversion to DNF formula has exponential complexity and the Fourier-Motzkin method has doubly exponential complexity in theory, but closer to exponential in practice [134]. Therefore, we obtain this way a doubly exponential complexity in practice, which is practical only for small examples.

Overall, we define algorithm ABSWEAK as the specialization of ABSGENERIC with:

- forward abstract interpretation ABSINTERP for invariant generation method IN-

```
define ABSWEAK:
  input program P
  output logical annotations for P
  compute invariants I by ABSINTERP
  for each check C do
    define $\phi_C$ as C weakened by $I_C$: $\phi_C = I_C \Longrightarrow C$
    define $\phi$ as the result of applying $\mathcal{W}$ to $\phi_C$
    define $\psi$ as the result of applying Fourier-Motzkin to $\phi$
    use $\psi$ to strengthen P precondition
  done
```

Figure 5.8: Algorithm ABSWEAK

VGEN;

- weakest preconditions $\mathcal{W}$ defined in Section 4.4.2 for precondition generation method PRECOND;
- Fourier-Motzkin method for quantifier elimination method QUANTELIM.

Illustration on Linear Search On unannotated program `linear_search` presented in Section 2.3.3, ABSWEAK works as follows:

line 4: invariant I_L is computed at loop entry, and I_P at function exit
loop invariant becomes $I_L \doteq 0 \leq \texttt{idx} \leq \texttt{len}$
postcondition becomes $I_P \doteq 0 \leq \mathit{result} < \texttt{len} \vee \mathit{result} \equiv -1$
line 5: treat check C_4

line 7:
$$\begin{aligned}\phi_1 \doteq \forall\, \texttt{idx}_1, \texttt{idx}_2.\ \mathit{typ} \wedge \texttt{idx}_1 \equiv 0 \\ \Longrightarrow (I_L[\texttt{idx} \mapsto \texttt{idx}_2] \wedge \texttt{idx}_2 < \texttt{len} \\ \Longrightarrow \mathit{offset\text{-}min}(\texttt{arr}) \leq \texttt{idx}_2 \leq \mathit{offset\text{-}max}(\texttt{arr}))\end{aligned}$$

line 8:
$$\phi_1 \doteq 0 < \texttt{len} \Longrightarrow (\mathit{offset\text{-}min}(\texttt{arr}) \leq 0 \wedge \texttt{len} \leq \mathit{offset\text{-}max}(\texttt{arr}) + 1)$$

line 9: precondition becomes $\mathit{typ} \wedge \phi_1$
line 5: treat check C_7

line 7: $\phi_2 \doteq \forall\, \texttt{idx}_1, \texttt{idx}_2.\ \mathit{typ} \wedge \texttt{idx}_1 \equiv 0$
$$\Longrightarrow (I_L[\texttt{idx} \mapsto \texttt{idx}_2] \wedge \texttt{idx}_2 < \texttt{len} \Longrightarrow \texttt{INT\_MIN} \leq \texttt{idx}_2 + 1 \leq \texttt{INT\_MAX})$$

line 8:
$$\phi_2 \doteq 0 < \texttt{len} \Longrightarrow \texttt{len} \leq \texttt{INT\_MAX}$$

line 9: precondition becomes $\mathit{typ} \wedge \phi_1 \wedge \phi_2$

Notice that we obtain the same results with ABSWEAK alone as with the addition of all the techniques mentioned previously, and without any of the twists.

5.3.2 Comparing Inference Techniques

Our goal is to generate a sufficient precondition ensuring safety, that allows as many calling contexts as possible. Thus, a technique outperforms another if (1) it is the only one to generate a sufficient precondition or (2) it generates a weaker sufficient precondition.

Theorem 2 *For each check C,* AbsWeak *generates a precondition ensuring that check C holds. The precondition generated is both sufficient, contrary to the precondition generated by abstract debugging, and better than the precondition generated by induction-iteration with immediate generalization.*

Proof. The weakest precondition computed on line 7 returns a sufficient precondition for $I_C \implies C$ to hold, by definition of weakest preconditions. According to Formula 5.12, it is also a sufficient precondition for C to hold. By the property that quantifier elimination returns a stronger (possibly equivalent) formula, the quantifier-free formula computed on line 8 is also a sufficient precondition for C to hold. Then, this formula is a sufficient precondition ensuring that C holds.

The precondition generated by abstract debugging is not guaranteed to be a sufficient precondition, as shown in Section 5.2.1.

The precondition generated by induction-iteration, where generalization is performed at the first iteration, is obtained with weakest preconditions from C, the same way the precondition generated by AbsWeak is obtained from $I_C \implies C$. Since $I_C \implies C$ is weaker than C, the generated precondition is also weaker, thus better. □

Theorem 3 AbsWeak *and abstract debugging are not comparable.*

Proof. It is sufficient to show that, in some cases, AbsWeak is better than abstract debugging, while in other cases the opposite holds.

For function `linear_search`, AbsWeak generates a sufficient precondition while abstract debugging does not, thus AbsWeak is better in this case.

```
void dummy1(int i, int n) {
  while (1) {
    i = i + 1;
    i = i - 1;
    assert (i < n);
  }
}
```

On function `dummy1` above, forward abstract interpretation in both abstract debugging and AbsWeak cannot generate a non-trivial loop invariant, no matter which abstract domain is used, because nothing is known at function entry. Then, AbsWeak generates no precondition, because the weakest precondition is

$$\forall i.\, i < n$$

which is equivalent, after quantifier elimination, to *false*. Abstract debugging generates loop invariant

$$i < n$$

by backward abstract interpretation around the loop, which finally leads to the generation of necessary precondition $i < n$, which is also a sufficient precondition in this case. Thus abstract debugging is better in this case. □

As a side remark, our implementation in Frama-C does generate an appropriate sufficient precondition `i < n` thanks to the automatic introduction of intermediate offset variables (see Section 8.1). Introducing an offset variable for variable `i` to track relative changes to `i`'s value allows one to generate the expected loop invariant that this offset is null at each loop iteration. This in turn allows one to generate the expected precondition.

More generally, the case of `dummy1` where backward abstract interpretation succeeds in generating a loop invariant that forward abstract interpretation cannot generate is very unlikely in practice. Thus, ABSWEAK should be preferred in general over abstract debugging.

Theorem 4 ABSWEAK *and induction-iteration are not comparable.*

Proof. It is sufficient to show that, in some cases, ABSWEAK is better than induction-iteration, while in other cases the opposite holds.

On function `linear_search`, ABSWEAK generates a sufficient precondition while induction-iteration does not, thus ABSWEAK is better in this case.

```
void dummy2(int i, int j, int n) {
  while (1) {
    assert (i < n);
    i = j;
  }
}
```

On function `dummy2` above, forward abstract interpretation in ABSWEAK cannot generate a non-trivial loop invariant, no matter which abstract domain is used, because nothing is known at function entry. Then, the weakest precondition is

$$\forall\, i.\, i < n$$

which is equivalent, after quantifier elimination, to *false*. This is the trivial sufficient precondition that ABSWEAK generates in this case. Induction-iteration quickly converges if generalization is not performed:

$$W_0 \doteq i < n \qquad W_1 \doteq j < n \qquad W_2 \doteq j < n.$$

This leads to the generation of sufficient precondition $i < n \wedge j < n$. Thus induction-iteration is better in this case. □

It should be clear that `dummy2` is a carefully crafted example that seldom occurs in practice. Most loop invariants in programs result from the natural forward sequencing of instructions, which depends a lot on what precedes the loop. This is precisely what cannot be captured by induction-iteration. Thus, ABSWEAK should be preferred in general over induction-iteration.

5.3.3 Taming Time and Space Complexity

More Efficient Preconditions ABSWEAK results may be very satisfying on a small scale, but its time and space complexity prevent applying it to larger programs. Indeed, it suffers from multiple sources of exponential explosion, both in space and time:

1. The classical computation of weakest preconditions is exponential in the number of branches in the program [59]. This can be overcome with an efficient computation of weakest preconditions [119], that is quadratic in theory but linear in practice. We did not implement these optimized weakest preconditions, as the benefit of it would be wasted by the transformation to DNF before Fourier-Motzkin quantifier elimination.

2. Conversion to DNF has exponential complexity in the size of the formula.

3. Fourier-Motzkin quantifier elimination for the rationals (or real numbers) from a DNF formula has doubly exponential complexity in theory, and simple exponential complexity in practice.

Overall, the classical computation of weakest preconditions can be seen as performing a part of the conversion to DNF, which leads to a combined doubly exponential complexity in practice. This is still too costly to be applicable to larger programs, which we checked in our experiments in Chapter 8. Thus, we devised variants of ABSGENERIC that differ from ABSWEAK by the precondition method PRECOND used:

- Algorithm ABSSTRONG is based on a precondition method that computes a stronger formula than plain weakest preconditions, by ignoring statements that do not interfere directly with the formula being propagated backwards. A statement interferes with a formula ϕ either by constraining (in a test) or modifying (in an assignment or a call) a location that overlaps with a syntactic abstract variable in ϕ. Overlap of locations is determined by calling function *paths-may-overlap*. This lifts the heuristic back-propagation described by Janota [102] to work with syntactic abstract variables.

- Algorithm ABSELIM completely replaces the weakest preconditions propagation by quantifying the desired initial formula over all possibly modified syntactic abstract variables, which relies on a computation of effects and function *paths-may-overlap* to define possible overlaps.

Although ABSSTRONG and ABSELIM have the same complexity as ABSWEAK, they generate in practice much simpler quantified formulas, which leads to practical quantifier elimination for programs of a few hundred loc (see Chapter 8).

On our `linear_search` example, ABSSTRONG and ABSELIM perform as well as ABSWEAK, leading to the same sufficient precondition. This is not always the case, as ABSSTRONG and ABSELIM may fail to take into account relations between variables that stem from statements that ABSSTRONG ignores or that were not caught by the invariant used in ABSELIM. Indeed, ABSELIM performs as well as ABSWEAK in those cases where the invariant obtained by abstract interpretation is so precise that a backward propagation by weakest preconditions is not needed. Likewise, ABSSTRONG performs as well as ABSWEAK in those cases where either the invariant obtained by abstract interpretation is precise enough, like for ABSELIM, or the strong preconditions computation captures the missing relational information.

More Efficient Quantifier Elimination The efficiency of our technique depends crucially on the efficiency of the quantifier elimination method chosen. A recent technique shows great promise in this matter [134], by removing the need for conversion to DNF and relying instead on an SMT-solver to explore the possible models of a formula.

Indeed, ABSELIM remains impractical for large programs, with many variables. To see this, let us consider the case of function `elim`.

```
int a1, a2, b1, b2, c1, c2, d1, d2;

void elim(int x, int y, int z) {
  while (1) {
    if (a1 <= x && a2 <= x && x <= b1 && x <= b2 &&
        c1 <= y && c2 <= y && y <= d1 && y <= d2) {
      //@ assert y < z;
    }
    x = y + 1;
    y = x - 1;
  }
}
```

In function `elim`, the invariant computed by abstract interpretation at the assertion point is

$$I_C \doteq \mathtt{a1} \le \mathtt{x} \wedge \mathtt{a2} \le \mathtt{x} \wedge \mathtt{x} \le \mathtt{b1} \wedge \mathtt{x} \le \mathtt{b2} \wedge \mathtt{c1} \le \mathtt{y} \wedge \mathtt{c2} \le \mathtt{y} \wedge \mathtt{y} \le \mathtt{d1} \wedge \mathtt{y} \le \mathtt{d2}.$$

Since `x` and `y` are both modified in the loop, they should be universally quantified, as in

$$\begin{aligned}\phi \;&\doteq\; \forall \mathtt{x}, \mathtt{y}.\;\; \mathtt{a1} \le \mathtt{x} \wedge \mathtt{a2} \le \mathtt{x} \wedge \mathtt{x} \le \mathtt{b1} \wedge \mathtt{x} \le \mathtt{b2} \wedge \mathtt{c1} \le \mathtt{y} \\ &\qquad\quad \wedge \mathtt{c2} \le \mathtt{y} \wedge \mathtt{y} \le \mathtt{d1} \wedge \mathtt{y} \le \mathtt{d2} \Longrightarrow y < z.\end{aligned}$$

Eliminating those quantifiers leads to precondition

$$\begin{aligned}\psi \;&\doteq\; \mathtt{a1} \le \mathtt{b1} \wedge \mathtt{a1} \le \mathtt{b2} \wedge \mathtt{a2} \le \mathtt{b1} \wedge \mathtt{a2} \le \mathtt{b2} \wedge \mathtt{c1} \le \mathtt{d1} \\ &\quad \wedge \mathtt{c1} \le \mathtt{d2} \wedge \mathtt{c2} \le \mathtt{d1} \wedge \mathtt{c2} \le \mathtt{d2} \Longrightarrow (\mathtt{d1} < \mathtt{z} \vee \mathtt{d2} < \mathtt{z}).\end{aligned} \tag{5.13}$$

Checking this big disjunctive precondition at `elim` call sites usually leads to exponential blowup of the initial formula when performing Fourier-Motzkin elimination. We noticed that most of these disjuncts correspond to escaping conditions, that are satisfied on executions that do not reach the check. Then, a possible solution is to remove these escaping disjuncts from the precondition inferred [147]. Instead of the precise precondition 5.13, this would generate the stronger

$$\mathtt{d1} < \mathtt{z} \vee \mathtt{d2} < \mathtt{z}. \tag{5.14}$$

The problem is that the resulting precondition is usually too strong, therefore we did not adopt this solution in our implementation.

A similar result could be obtained by modifying quantifier elimination. The first idea is to limit quantifier elimination to those variables that appear in the check considered. In our example, only `y` appears in check $\texttt{y} < \texttt{z}$. Therefore, eliminating only `y` leads to formula

$$\forall\,\texttt{x}.\quad \texttt{a1} \leq \texttt{x} \wedge \texttt{a2} \leq \texttt{x} \wedge \texttt{x} \leq \texttt{b1} \wedge \texttt{x} \leq \texttt{b2} \wedge \texttt{c1} \leq \texttt{d1} \wedge \texttt{c1} \leq \texttt{d2} \wedge \texttt{c2} \leq \texttt{d1} \wedge \texttt{c2} \leq \texttt{d2} \implies (\texttt{d1} < \texttt{z} \vee \texttt{d2} < \texttt{z}).$$

Then, removing from the DNF those disjuncts that still mention quantified variables leads to precondition

$$\texttt{c1} \leq \texttt{d1} \wedge \texttt{c1} \leq \texttt{d2} \wedge \texttt{c2} \leq \texttt{d1} \wedge \texttt{c2} \leq \texttt{d2} \implies (\texttt{d1} < \texttt{z} \vee \texttt{d2} < \texttt{z}).$$

This formula is indeed a stronger precondition than precondition 5.13. The second idea is to apply Fourier-Motzkin only on pairs of inequalities that mention the check. This leads to precondition

$$\texttt{d1} < \texttt{z} \vee \texttt{d2} < \texttt{z}. \tag{5.15}$$

This is indeed the same formula as precondition 5.14. With these improved elimination methods, AbsElim indeed scales to large programs.

Now, the precondition generated might be too strong. In particular, it might be false, which prevents any call to this function, or it may be inconsistent with the precondition of the function, whether a user precondition or the implicit one guaranteed by typing. Therefore, we systematically test the consistency of the conjunction before we add an inferred precondition. Of course, it might still be that no possible context satisfies this precondition. *E.g.*, precondition

$$0 < 2 \times i < 2$$

does not have solutions for an integer parameter i, but it has solutions in rationals, so a quantifier elimination method for rationals such as the Fourier-Motzkin quantifier elimination might return such a formula, and we will keep it as a valid precondition.

5.4 Other Related Work

Loop Invariant Inference Historically, array bound checking has been one of the first difficult properties about programs that people tried to prove, the hardest part of the verification task being the automatic inference of loop invariants. In 1978, Cousot and Halbwachs [52] applied abstract interpretation over polyhedra and managed to check memory safety of an implementation of `heapsort`, using manual preconditions. A year earlier, Suzuki and Ishihata [166] devised a method based on weakest preconditions to check memory safety of an implementation of tree sort. They used Fourier-Motzkin elimination at loop entrance as a heuristic to make their induction-iteration method converge.

More recently, Xu *et al.* [181, 180] have refined with success the induction-iteration method for safety checking of machine code. They use forward abstract interpretation and induction-iteration separately to generate loop invariants, and they rely on user preconditions to provide a valid calling context.

More generally, much work has targeted loop invariant inference by abstract interpretation, predicate abstraction and weakest preconditions/strongest postconditions, or a combination thereof. This is likely to continue to be a major research goal in the years to come. Promising techniques combine abstract interpretation and deductive verification. Leino and Logozzo [120] build a real feedback loop between a theorem prover and an abstract interpretation module to generate loop invariants. In [121], the same authors present an embedding of the abstract interpretation technique of widening inside a theorem prover. The opposite approach of performing abstract interpretation on logic formulas has been presented by Tiwari and Gulwani [80].

Precondition Inference Bourdoncle defines abstract debugging [26] as backward abstract interpretation from assertions. Along the way, he generates loop invariants and preconditions in order to prove these assertions. He focuses on array bound checking too. His backward propagation merges the conditions to reach the program point where the assertion is checked and the conditions to make this assertion valid. The dual approach of propagating backward a superset of the forbidden states is described by Rival [153]. We have shown in this chapter the limitations of these approaches.

Gulwani and Tiwari [81] consider the problem of assertion checking for the special case of equalities in a restricted language with only non-deterministic branching. Using a method based on unification, they manage to generate necessary and sufficient preconditions for assertions to hold. Unfortunately, unification does not work for the relations that arise most often in practice for safety checking, namely less-than and greater-than relations. Our method only generates sufficient preconditions, but it applies to those arithmetic relations found in practice.

In a recent article, Popeea *et al.* [147] describe a technique similar to ours to generate sufficient preconditions. They combine forward abstract interpretation with constraint solving to generate preconditions for optimization of C programs.

5.5 Chapter Summary

We described a new method to infer annotations for functions, *i.e.*, loop invariants and function contracts, when there exists an elimination method for those atoms mentioning modified variables. This method combines the strengths of the two most effective techniques at inferring annotations: abstract interpretation and deductive verification. It is highly modular, as it can be applied to a single function, and its precision/cost ratio can be finely tuned.

We showed that this new method applies well to memory safety, where atoms consist in linear inequalities in the program variables and pseudo-variables introduced by our model of memory and programming idioms (*e.g.*, strings).

Chapter 6

Type-Safe Programs with Aliasing

Contents

In this chapter, we describe how to generate automatically and modularly annotations for type-safe JESSIE programs with aliasing, so that they can be checked safe as in Chapter 4.

Section 6.1 extends the JESSIE annotation language to express (non-)overlapping locations in type-safe programs. This allows one to express the frame condition as a normal function postcondition. Then, the pros and cons of aliasing in programs are detailed, to better understand what aliasing is used for in type-safe programs, and what it entails for

type	::=	...	
	|	`pset`	set of pointers

Figure 6.1: Grammar of JESSIE extended types

analyzing such programs. Finally, the limitations of existing alias analyses and alias control techniques are discussed.

To provide a better solution to this aliasing problem, Section 6.2 starts with presenting Talpin's alias analysis [94], a contextual variant of Steensgaard's alias analysis. It allows refining the checking techniques for annotated programs presented in Chapter 4. We show that this analysis is an instance of the more general solution of syntactic control of interference by Reynolds [152].

Section 6.3 exploits this parallel to define extensions of Talpin's analysis that make it both modular and complete. These extensions refine the type checking phase of Talpin's analysis with a verification phase involving automatic provers, when typing is not sufficient.

Finally, Section 6.4 shows how this new technique relates to other work.

6.1 Problem Overview

6.1.1 Memory Aliasing and Separation

Non-Overlap of Locations Aliasing is the property that a memory location can be referenced by two different names at some point in the program, or equivalently that two paths may refer to overlapping locations. As already mentioned in Section 4.2.3, any two memory locations may overlap in general. As discussed in Section 5.1.1, type safety restricts possible overlap, as locations which correspond to different fields may not overlap in type-safe programs. This coarse separation is not sufficient in many cases. Therefore, it is necessary to provide a fine grain separation through a new predicate $\mathit{separated}$, such that $\mathit{separated}(x, y)$ indicates that pointers x and y point to non-overlapping locations.

Predicate $\mathit{separated}$ can be defined in terms of the memory model encoding of JESSIE, as defined in Section 4.1.1. Given a pointer x of type $T[..]$ and a pointer y of type $S[..]$, non-overlap of the locations pointed-to by x and y can be expressed as

$$\begin{aligned} \mathit{separated}(x, y) \quad &\doteq \quad \mathit{base\text{-}block}(x) \not\equiv \mathit{base\text{-}block}(y) \\ &\quad\quad \vee\ \mathit{address}(x) + \mathit{sizeof}(T) \leq \mathit{address}(y) \\ &\quad\quad \vee\ \mathit{address}(y) + \mathit{sizeof}(S) \leq \mathit{address}(x). \end{aligned}$$

In practice, it is convenient to express the pairwise separation of two or more sets of pointers using the same $\mathit{separated}$ predicate, with the meaning that any two pointers in different sets are separated as above.

Extending the JESSIE Annotation Language Figure 6.1 presents the abstract syntax of JESSIE types, extended with a special logic type for sets of pointers. Pointers in a set should

prop	::=	...	
	|	`separated` (*pset* (, *pset*)$^+$)	memory separation

Figure 6.2: Grammar of JESSIE extended propositions

have the same type.

Figure 6.2 presents the abstract syntax of JESSIE propositions, extended with a predicate `separated` that expresses separation of sets of pointers.

6.1.2 Frame Condition Equivalent Postcondition

Based on the *separated* predicate, the frame condition of a function f in a type-safe program can be translated into an additional postcondition for this function. In fact, each part of the frame condition can be translated into an additional postcondition.

Memory Footprint Without loss of generality, the memory footprint of a function f can be written

$$\text{assigns } \overrightarrow{x_i}, \overrightarrow{\lambda_j}$$

where each x_i denotes a global variable and each λ_j denotes a memory location. By definition of memory locations in JESSIE, each λ_j can be rewritten in comprehension notation as

$$\{t_j.m_j \ : \ p_j\}$$

where t_j is a term of type $T_j[..]$, m_j is a field of structure T_j and p_j is the conjunction of the original predicates from locations in comprehension notation in λ_j.

Then, the memory footprint of f is equivalent to postcondition

$$\begin{array}{rcl} \mathit{post\text{-}mem}_f & \doteq & \displaystyle\bigwedge_{x \notin \overrightarrow{x_i}} x \equiv \mathit{old}(x) \\ & \wedge & \displaystyle\bigwedge_{T,m} \forall\, T[..]\; x\; ;\; \mathit{old}(\mathit{valid}(x)) \wedge \mathit{valid}(x) \Longrightarrow \\ & & \left(\displaystyle\bigwedge_{m_j \equiv m} (p_j \Longrightarrow \mathit{separated}(x, \mathit{old}(t_j))) \right) \Longrightarrow x.m \equiv \mathit{old}(x.m), \end{array}$$

where the first conjunction ranges over global variables, and the second conjunction ranges over structures and their fields.

```
int i, j;
//@ assigns i, *x;
void f_mem(int *x);
```

As an example, the memory footprint of function `f_mem` is equivalent to postcondition

```
/*@ ensures j ≡ \old(j)
  @         ∧ (∀ int *y; \old(\valid(y)) ∧ \valid(y) ⟹
  @             separated(y,x) ⟹ *y ≡ \old(*y))
  @         ∧ ∀ char *y; old(valid(y)) ∧ valid(y) ⟹ *y ≡ \old(*y)
  @         ∧ ...
  @*/
```

where the quantification is limited to pointers to integers and characters. Indeed, all other pointers should be treated similarly.

Deallocation Footprint Likewise, without loss of generality, the deallocation footprint of f can be written

$$\text{deallocates } \overrightarrow{t_i}$$

where each t_i denotes a term of type $T_i[..]$. Then, f deallocation footprint is equivalent to postcondition

$$\begin{aligned} \textit{post-dealloc}_f \quad \doteq \quad & \bigwedge_T \forall\, T[..]\ x\ ;\ \textit{old}(\textit{valid}(x)) \Longrightarrow \\ & \left(\bigwedge_{T_i \equiv T} \textit{separated}(x, \textit{old}(t_i)) \right) \Longrightarrow \textit{valid}(x) \end{aligned}$$

```
//@ frees x;
void f_dealloc(int *x);
```

As an example, the deallocation footprint of function `f_dealloc` is equivalent to postcondition

```
/*@ ensures
  @   ∀ int *y; \old(\valid(y)) ⟹ separated(y,x) ⟹ \valid(y);
  @   ∧ ∀ char *y; \old(\valid(y)) ⟹ \valid(y)
  @   ∧ ...
  @*/
```

where the quantification is limited to pointers to integers and characters. Indeed, all other pointers should be treated similarly.

Allocation Footprint Finally, without loss generality, the allocation footprint of f can be written

$$\text{allocates } \overrightarrow{t_i}$$

where each t_i denotes a term of type $T_i[..]$. Then, the allocation footprint of f is equivalent to postcondition

$$\begin{aligned} \textit{post-alloc}_f \quad \doteq \quad & \bigwedge_i \textit{valid}(t_i) \\ \wedge \quad & \bigwedge_T \forall\, T[..]\ x\ ;\ \textit{old}(\textit{valid}(x)) \wedge \textit{valid}(x) \Longrightarrow \bigwedge_{T_i \equiv T} \textit{separated}(x, t_i) \end{aligned}$$

```
//@ allocates x;
void f_alloc(int *x);
```

As an example, the allocation footprint of function `f_alloc` is equivalent to postcondition

```
/*@ ensures \valid(x)
  @        ∧ ∀ int *y; \old(\valid(y)) ∧ \valid(y)
  @                    ⟹ separated(y,x);
  @*/
```

where the quantification is complete here.

6.1.3 Aliasing Considered Harmful

Aliasing can be rightfully considered harmful from the point of view of most program analyses, because it degrades their performance, both in terms of scalability and precision. This is true for analyses directed either at optimization or at verification. Optimization analyses tend to err on the safe side, assuming more aliasing than necessary, although some compiler optimizations assume less aliasing than necessary when instructed to do so by a user, like `--strict-aliasing` in `gcc`, thus leading to subtle errors when misused. On the other hand, many verification analyses prefer being unsound w.r.t. aliasing to fully supporting it, in order to give generally accurate, if not always correct, results. This is the case in bug finders and program understanding tools.

```
1  void linear_search_ptr
2          (int arr[], unsigned int *len, int *key, int *idx) {
3    *idx = 0;
4    while (*idx < *len) {
5      if (arr[*idx] == *key) {
6        return; // key found
7      }
8      *idx = *idx + 1;
9    }
10   *idx = -1; // key not found
11 }
```

It is easy to see why aliasing is such a nuisance on a simple example. Consider a modified version of function `linear_search` presented in Section 2.3.3, where `len` and `key` are now pointers, and `idx` is a pointer parameter updated to the correct value before returning. Without any more information on the calling context of `linear_search_ptr`, there could be aliases between any two of `key`, `idx` and `arr[i]` for any integer index `i`. As a consequence, assigning to `*idx` on line 8 might change the value of any element in `arr`, as well as the value pointed-to by `key`. In the case of `linear_search_ptr`, since `idx` is the only pointer assigned-to, and the value of `*idx` is the only one that matters for safety, the contract of function `linear_search` presented in Section 4.1.3 is still correct. This is quite fortunate. In general, the presence of aliasing may either lead to errors, or totally change the behavior of a function.

6.1.4 Aliasing as a Programming Discipline

C programs usually rely on aliasing in many ways, some of which can even be considered idiomatic, *i.e.*, characteristic of C programming. Hackett and Aiken have presented an in-depth study of uses of aliasing in C programs [83] (for systems software). They identify nine patterns of aliasing which account for almost all aliasing. Quoting their work, these patterns belong to two categories:

- incidental aliasing*: when a pointer targets locations outside unbounded structures;*
- cursor aliasing*: when a pointer may target locations within an unbounded structure;*

and they can arise at four different levels:

- entry aliasing *is specific to the entry state of a function;*
- exit aliasing *is specific to the exit state of a function;*
- global invariant aliasing *is specific to a global variable;*
- type invariant aliasing *is specific to all values of a type.*

Quoting again their work, incidental aliasing represents five patterns:

- parent pointers *are references to particular data closer to the root of a structure;*
- child pointers *are additional references to particular data stored deeper in a structure;*
- shared immutable pointers *are multiple references to data at the same level, where all are used only for reading;*
- shared I/O pointers *are two references to data at the same level, where one is used only for reading and the other only for writing;*
- global pointers *are references to a global variable and an alias of the global within the same scope;*

while cursor aliasing represents four patterns:

- index cursors *support the use of an additional index for a structure;*
- tail cursors *hold the end point of an existing index;*
- query cursors *read data internal to an existing index;*
- update cursors *write data internal to an existing index.*

Parent and child pointers are typical of type and global invariant aliasing. Indeed, many data structures in C rely on aliased pointer fields. *E.g.*, circular lists as below with `prev` and `next` fields alias `x` with `x`→`next`→`prev`.

```
struct List {
  struct List *prev;
  struct List *next;
};
```

Failing to take such aliasing into account in our analysis would lead to wrong results. *E.g.*, if we assumed function `remove_head` below alias-free and applied the techniques from Chapter 5 to check its safety, we would end up guaranteeing it cannot fail, while it does fail for a circular list with only one element, where `list→next` and `list→prev` represent the same element.

```
/*@ requires \valid(list)
  @          ∧ \valid(list→prev) ∧ \valid(list→next);
  @*/
void remove_head(struct List *list) {
  list→next→prev = list→prev;
  list→next = 0;
  list→prev→next = list→next; // list→prev may be null
  list→prev = 0;
}
```

Shared immutable pointers, shared I/O pointers and global pointers are mostly used in entry and exit aliasing. They allow a program to pass around parts of larger data structures between functions. When data is only read, as in shared immutable pointers, aliasing does not impact the function's behavior.

```
struct Node { int id; };

/*@ requires \valid(x) ∧ \valid(y);
  @ ensures \result ≡ x→id − y→id;
  @*/
int compare(struct Node *x, struct Node *y) {
  return x→id − y→id;
}
```

E.g., in function `compare`, the structures pointed-to by parameters `x` and `y` are only read, therefore there is no problem in allowing `x` and `y` to alias. This may be the case if `compare` is used as a comparison function in a sorting algorithm for a collection with repeated elements. Shared I/O pointers and global pointers are trickier to handle correctly, as writes and reads may interfere. In general, writes are performed after reads, so that values read do not depend in an indirect way on values written.

Finally, cursor aliasing serves to iterate through elements of a collection, whether a simple array or a more complex recursive structure.

```
/*@ requires \valid_range(x,0,end-x);
  @ ensures *x ≡ \old(*x) − 1 ∧ *end ≡ \old(*end) − 1;
  @*/
void scan(int *x, int *end) {
  int *iter = x;
  while (iter != end) {
    *iter = *iter − 1; // iter and x are aliases the first time
    iter = iter + 1;
  }
  *iter = *iter - 1; // iter and end are aliases at that point
}
```

In function `scan` above, `x` is an index cursor, `end` a tail cursor associated to `x`, and `iter` an update cursor associated to `x`. Failing to take both sources of aliasing into account can lead to wrong results. *E.g.*, assuming function `scan` is alias-free and applying the techniques from Chapter 5 (without using function *paths-may-overlap*) would lead us to prove `scan`'s postcondition cannot hold, because it depends on the aliasing of `iter` with `x` and `end`.

6.1.5 Problems with Alias Analyses

There is considerable work on alias analyses [88, 178]. Most focus on may-aliasing, computing an over-approximation of real aliasing, while a few deal also with must-aliasing, computing an under-approximation of real aliasing. In our setting, non-interference of reads and writes to memory is the property of most interest, therefore may-aliasing is the interesting one. Analyses for may-aliasing fall into two categories: points-to analyses compute for each pointer the set of locations it points to, while alias analyses determine whether two pointers point to the same location. Points-to analyses do not fit well with our situation: (1) points-to analyses do not handle well full modularity, sometimes downward modularity [154] where a function is missing, never upward modularity where the context is missing (or in that case the analysis depends on a previous global analysis [155]); (2) there is no technique to build function summaries expressing the results of a points-to analysis in logic. Therefore, we are left with may-alias analyses [37, 38].

May-alias analyses compute an over-approximation of the set of pointers aliased at every program point. Like points-to analyses, they are essentially not modular, usually requiring the complete program to work. Another problem is the precision of results: alias analyses that scale are context-, flow- and path-insensitive [88], while program verification asks for context-, flow- and path-sensitive analyses [83]. In a modular setting, where the calling context of a function is not known, the usual solution is to assume as much aliasing as possible. In our type-safe setting, this would lead to possible aliasing between any two pointers of the same type. This only makes the precision problem worse.

```
void plus_minus(int* x, int* y,
                int* z, int* t) {
  *z = *x + *y;
  *t = *x - *y;
}
```

```
void id(int* p, int* q) {
}
void opp(int* p, int* q) {
  *q = - *q;
}
```

To see why we need to consider that much aliasing, consider function `plus_minus` above. A "natural" contract for function `plus_minus` would be the following paraphrase of its code:

```
/*@ requires \valid(x) ∧ \valid(y) ∧ \valid(z) ∧ \valid(t);
  @ ensures *z ≡ *x + *y ∧ *t ≡ *x − *y;
  @*/
```

Unfortunately, this contract is only correct if aliasing between `plus_minus` parameters is restricted. Functions `id` and `opp` are special instances of function `plus_minus` that do not respect this contract. Indeed, whenever `p` and `q` do not alias, `id(p,q)` is the same as `plus_minus(p,q,p,p)`, but it is false in general that $\star\text{p} \equiv \star\text{p} - \star\text{q} \wedge \star\text{p} \equiv \star\text{p} + \star\text{q}$ when `id` returns. Likewise, `opp(p,q)` is the same as `plusminus(p,q,q,q)`, but it is false in general that $\star\text{q} \equiv \star\text{p} - \star\text{q} \wedge \star\text{q} \equiv \star\text{p} + \star\text{q}$ when `opp` returns. With only 4 parameters in `plus_minus`, there are already 15 partitions of the set {`x`,`y`,`z`,`t`}, that represent each a different aliasing context, possibly leading to a different behavior. Both common sense and software engineering practice command that function `plus_minus` cannot behave correctly in all these different cases. To say it another way, the author of function `plus_minus` cannot have intended to support that many different usages of `plus_minus`.

A naive solution would require all parameters of `plus_minus` to be pairwise different. On one hand, it would make `plus_minus` alias-free, thus allowing us to apply the results of Chapter 5. On the other hand, it could rule out desirable aliasing like those cases mentioned in Section 6.1.4. In particular, aliasing between `x` and `y` is not a problem, as it does not falsify the postcondition of `plus_minus`. Some kinds of aliasing should definitely be allowed, while others could be forbidden. Existing alias analyses do not provide any solution to analyze modularly function `plus_minus` while allowing some kinds of aliasing.

6.1.6 Problems with Alias Control Techniques

To overcome the lack of modularity of alias analyses, many alias control techniques have been described. These techniques have not been widely adopted industrially, either because they overly restrict the kind of programs allowed, or because they require programs to be annotated with additional types or complex logical formulas.

The reason we are interested in aliasing analyses is non-interference of reads and writes to memory. Only separation propositions can give us the assurance that parts of the heap do not overlap. In fact, this non-interference property is also useful in optimizing code, which motivated the introduction of keyword `restrict` in C99. A few annotation-based systems help programmers specify pointer separation [1, 111] for use in dedicated analyzers.

Reynolds's separation logic [151] is currently the most researched alias control technique for C-level programs. It allows one to specify separation of pointers in a special pointer logic, in a very concise way. Despite much work, separation logic annotations are still difficult to check, although tools do provide some level of automatic checking [17, 61, 101]. Separation logic annotations are also difficult to infer automatically, and they do not offer the same flexibility as simple applications of a separation predicate, *e.g.*, when dealing with partially shared data structures.

6.1.7 Problem Statement

In this chapter, we consider the problem of automatically generating and checking logic annotations for type-safe incomplete programs with aliasing. Existing techniques do not fit well with these constraints: techniques based on abstraction like alias analyses are poorly modular, while techniques based on deduction like alias control techniques necessitate heavyweight annotations and either overly restrict valid programs or cannot be verified easily.

Our goal is to generate an appropriate non-aliasing context for each function, that is strong enough that function safety can be checked modularly, and weak enough that usual usages of aliasing are still allowed. In fact, we should seek the weakest possible non-aliasing context that still allows us to prove safety. The inherent difficulty of this task is that aliasing is not a local property.

Ideally, the solution aliasing context should make it possible both to improve on the definition of *paths-may-overlap* presented in Section 4.2.3, and to generate a better translation to WHY.

6.2 Inferring Regions: Existing Type-Based Approaches

As seen in Section 5.1.1, the restriction to type-safe programs allows the partition of memory into disjoint parts, which makes both inference and checking of annotations in JESSIE easier. Another crucial benefit from this partitioning of memory is that it translates easily to WHY, so that deductive verification in WHY is also much easier. This is the basis for our solution.

6.2.1 Steensgaard's Region Inference

When all the program is available, Steensgaard's unification-based alias analysis [165] allows one to partition memory. Instead of partitioning memory according to types, which is possible with type safety, Steensgaard's analysis partitions memory according to aliasing.

Although presented initially in terms of equivalence classes of pointers, we present it here in terms of sets of memory locations, a.k.a. *regions*, such that two pointers aliased necessarily point to the same region. The algorithm is based on unification, with all paths initially assumed to be in a different region, and regions unified as necessary, based on a few rules:

- *assignment* - regions pointed to by both sides of an assignment are unified;
- *function call* - regions pointed to by corresponding parameters and arguments of a call are unified;
- *return* - region pointed to by a function result and a term returned are unified.

This ensures that aliases necessarily point to the same region. In particular, pointer arithmetic does not change regions: `p` and `p⊕i` point to the same region. Likewise, embedded field access, whose semantics is similar to pointer arithmetic, does not change regions. Finally, normal field access to pointer field `m` from different pointers to region ρ_1 always result in pointers to the same region ρ_2. Figure 6.3 presents locations that belong to the same region with different colors, while Figure 6.4 refines the separation obtained by safe typing with regions.

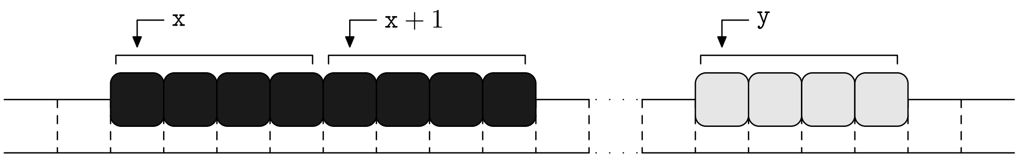

Figure 6.3: Regions in a byte-level memory model

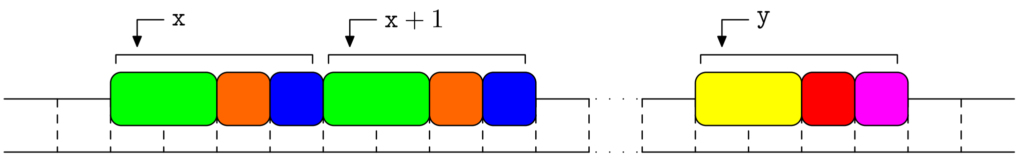

Figure 6.4: Regions in a component-as-array memory model

Reduced Overlap Between Locations After regions are computed, function *region-of-path* simply returns the region of a path. Then, it is easy to refine function *paths-may-overlap* presented in Section 5.1.1 for type-safe programs. This new definition is presented in Figure 6.5. Previously, case (2.1) concluded that two memory locations accessing the same field could always overlap. It now refines into case (2.1") which concludes that two memory locations accessing the same field can overlap only if they belong to the same region.

Improved JESSIE Analyses The improvement of function *paths-may-overlap* translates into better JESSIE analyses. *E.g.*, function `alias_free` below is the same as the one presented in Section 5.1.2, except we do not assume it is alias-free anymore. Instead, we present it in the context of a complete program. There are 2 Steensgaard regions for this program: {`sx`,`x`→`m`} and {`sy`,`y`→`m`}. Then, both abstract interpretation and deductive ver-

```
define region-of-path:
  input path $\pi$
  output region $\rho$ of $\pi$

define paths-may-overlap:
  input paths $\pi_1$ and $\pi_2$
  output whether $\pi_1$ and $\pi_2$ represent overlapping locations
  match $(\pi_1,\pi_2)$ with
  (1)      | (x,x) $\rightarrow$ return true
  (2.1")   | ((_$\oplus$_).m,(_$\oplus$_).m) $\rightarrow$ return region-of-path($\pi_1$) $\equiv$ region-of-path($\pi_2$)
  (2.2)    | ((_$\oplus$_)._,(_$\oplus$_)._) $\rightarrow$ return false
  (3)      | (x,_) | (_,x) $\rightarrow$ return false
```

Figure 6.5: Overlap of paths with regions

ification on JESSIE can benefit from the non-overlap of `x→m` and `y→m` in `alias_free`, so that all annotations can be checked.

```
struct S { int m; };

/*@ requires \valid(x) ∧ \valid(y) ∧ y→m ≡ 1;
  @ ensures x→m ≡ 0;
  @*/
void alias_free(struct S *x, struct S *y) {
  x→m = 0;
  //@ assert y→m ≡ 1;
}

void main() {
  struct S sx;
  struct S sy = { 1 };
  alias_free(&sx,&sy);
}
```

Improved Translation to WHY The translation from JESSIE to WHY also benefits from the partitioning of memory into regions. Memory variable $Heap_m$ for field m can be replaced by a collection of $Heap_m^R$ variables, one for each region R, which naturally encodes the absence of interference between accesses to the corresponding regions.

E.g., here is the translation of function `alias_free` into WHY:

```
unit alias_free
     (pointer x, pointer y, heap Heap_m_x, heap Heap_m_y) =
  update_m(Heap_m_x,x,0)
  assert select_m(Heap_m_y,y) ≡ 1
```

6.2.2 Talpin's Region Inference

There are two problems with Steensgaard's regions: (i) lack of context sensitivity, which aliases more pointers than necessary, and (ii) lack of modularity, as the complete program is needed. Problem (i) manifests on `alias_free` with the following code for `main`:

```
11 void main() {
12   struct S sx, sz;
13   struct S sy = { 1 };
14   alias_free(&sx,&sy);
15   sx.m = sx.m + 1;
16   alias_free(&sz,&sx);
17 }
```

In this program, `sx` is used both in position of first argument to `alias_free` on line 14, and as second argument to `alias_free` on line 15. Then, there is only one Steensgaard's region {`sx`,`x→m`,`sy`,`y→m`,`sz`} for the complete program, which prevents checking annotations in this program.

A Contextual Variant of Steensgaard's Region Inference Hubert and Marché have described a solution to this problem, Talpin's alias analysis [94], following an original idea from Talpin and Jouvelot for computing effects [167, 168], used later on by Tofte and Talpin for static memory allocation [170]. Instead of computing only global regions, like in Steensgaard's analysis, they compute both global and parametric regions. A parametric region can be viewed as an additional parameter of the function, so that calls to this function must pass in either a global or a parametric region in scope. This allows various calls to the same function to specify different region instances for each call, thus preventing aliasing due to merging of contexts.

There is no need to change function *paths-may-overlap* to use these new regions. The only change is that *region-of-path* now returns the result of Talpin's analysis instead of plain Steensgaard's analysis. On our modified complete `alias_free` program, Talpin's analysis computes 2 local regions in function `alias_free`, namely regions {`x`→`m`} and {`y`→`m`}, and 3 local regions in function `main`, namely regions {`sx`}, {`sy`} and {`sz`}.

Then, analyses on JESSIE programs can benefit from the improvement in function *paths-may-overlap*, as before. *E.g.*, function `alias_free` with the `main` function given above can be checked safe by both abstract interpretation and deductive verification of the corresponding JESSIE program. Talpin's region also translate into a partitioning of memory in WHY, like Steensgaard's regions.

Incompleteness of Talpin's Region Inference A strict constraint to ensure soundness of the approach is that no two regions accessed in a function should be in fact be equal for a particular call. This is the same condition as the one expressed by Reynolds in his work on syntactic control of interference [152], where regions play in our case the role of collections in his work. This is guaranteed by failing to compute contextual regions in any of the following two cases:

1. if calling a function leads to passing twice the same region in parameter;
2. if calling a function leads to passing as a parameter a global region already accessed in the callee.

Talpin's region inference trades incompleteness for precision and scaling: it either succeeds quickly with precise results, or it fails.

```
int glob;

/*@ requires \valid(x) ∧ \valid(y);
  @ ensures *x ≡ *y + 1 ∧ glob ≡ *y + 1;
  @*/
void bad_regions(int *x, int *y) {
  *x = *y + 1;
  glob = *y + 1;
}

void main() {
  int loc;
  bad_regions(&loc,&loc);
14   bad_regions(&loc,&glob);
15 }
```

Function `bad_regions` illustrates why both cases should not be allowed. Talpin's analysis computes 2 local regions {`*x`} and {`*y`} in function `bad_regions`, plus global region {`glob`}. These 3 regions being different, it is possible to check `bad_regions` annotations by abstract interpretation or deductive verification in JESSIE. Then, the call to `bad_regions` on line 13 would not establish postcondition `*x` $\equiv$ `*y` $+ 1$. This is not allowed, as this call passes region {`loc`} as a parameter twice, which violates (1). Likewise, call to `bad_regions` on line 14 would not establish postcondition `glob` $\equiv$ `*y` $+ 1$. This is not allowed, as this call passes region {`glob`} in parameter, which violates (2).

6.3 Refining Regions: a New Type-and-Effect Approach

Talpin's alias analysis [94] computes regions in a way that solves problem (i), context insensitivity, but still suffers from problem (ii), lack of modularity, while it adds a new problem (iii), incompleteness. We propose a solution to both problems, provided aliasing in the program analyzed is reasonably restricted, which is precisely defined.

6.3.1 Equivalence of Paths and Regions

We have already defined function $\mathit{region\text{-}of\text{-}path}$ that returns the region of a path. Since a region is an equivalence class of paths, it is also possible to define function $\mathit{paths\text{-}of\text{-}region}$ that maps a region to its preimage by $\mathit{region\text{-}of\text{-}path}$. In fact, this preimage is restricted to those paths not mentioning local variables, a.k.a. *interface paths*, that can be used in a function contract.

Unfortunately, $\mathit{paths\text{-}of\text{-}region}$ is not always computable, as there might be an infinite number of interface paths in the preimage, due to recursive structures. For those paths going through recursive structures, we use a notation by comprehension to express the corresponding locations. This allows us to define a function $\mathit{locations\text{-}of\text{-}region}$ such that $\mathit{locations\text{-}of\text{-}region}(\rho)$ returns the finite set of locations denoting all interface paths in $\mathit{paths\text{-}of\text{-}region}(\rho)$.

6.3.2 Modular Region Inference

Beyond Enforced Non-Modularity Talpin's analysis is mostly modular, as regions are computed in a modular way. Only the verification that different regions in a callee are not instantiated with the same region in the caller is not modular. We propose to delay this verification for incomplete programs by computing equivalent separation preconditions.

In order to be precise enough, we assume invariants have been computed by abstract interpretation over domain $\overline{\mathbf{D}}$ at every program point. Then, Figure 6.6 defines function $\mathit{immutable\text{-}lower\text{-}bound}$ (resp. $\mathit{immutable\text{-}upper\text{-}bound}$) that optionally returns a lower bound (resp. an upper bound) for a term that is valid at every program point in the function, in particular in the function precondition. *E.g.*, in function `linear_search`, an

```
define immutable-lower-bound:
  input term $t_1$ and invariant $I$
  output either nothing or a term $t_2$ such that $t_2$ is immutable and $I \Longrightarrow t_2 \leq t_1$
  if $t_1$ is immutable then
    return $t_1$
  else
    define $E = \overline{\mathbf{D}}.lbound(I,t_1)$
    if E has no immutable element then return nothing else return such an element

define immutable-upper-bound:
  input term $t_1$ and invariant $I$
  output either nothing or a term $t_2$ such that $t_2$ is immutable and $I \Longrightarrow t_1 \leq t_2$
  if $t_1$ is immutable then
    return $t_1$
  else
    define $E = \overline{\mathbf{D}}.ubound(I,t_1)$
    if E has no immutable element then return nothing else return such an element
```

Figure 6.6: Immutable bounds for terms

immutable lower bound for term `idx` is `0`, and an immutable upper bound for `idx` is `len`.

Based on these, function *path-effects* returns a set of **immutable paths** and regions accessed by its argument path. Paths are more precise than regions, therefore they are preferred whenever possible. Immutable paths are those paths that are syntactically built from terms whose value does not change during the function. *E.g.*, the path effect for path `arr[idx]` in `linear_search`, given invariant $0 \leq$ `idx` $\leq$ `len` is `arr[0..len]`.

Memory can also be accessed indirectly through a call. Given the effects of a callee, function *call-effects* returns a set of immutable paths and regions accessed by the caller. Function *function-effects* simply collects all effects of a function due to its memory accesses and calls. Functions are processed in reverse topological order of the call-graph, and recursive calls are handled by iterating on strongly connected components until a fix-point is reached.

Once effects of a function are computed, function *generate-function-precondition* defined in Figure 6.8 generates a precondition that guarantees soundness of Talpin's analysis. For each pair of locations possibly accessed in different regions, it generates a separation condition that guarantees they may not overlap. It calls *regions-may-overlap* which returns false if type safety ensures the regions may not overlap, or if one region is internal to a function, defined as a region inaccessible by any interface path. Indeed, in this last case, the internal region cannot possibly overlap with any other internal region or a region external to the function.

Illustration To see how this works, we recall the code of function `bad_regions`:

```
6  void bad_regions(int *x, int *y) {
7    *x = *y + 1;
```

```
define path-effects:
  input path $\pi_1$ and invariant I
  output pair of a set of immutable paths $\Pi$ and a set of regions $\mathrm{P}$ accessed by $\pi_1$
  match $\pi_1$ with
  | x $\longrightarrow$ if x is not modified in the current function then return $(x,\varnothing)$
       else return $(\varnothing$,region-of-path$(x))$
  | $(\pi_2 \oplus [t_1..t_2]).m \longrightarrow$
     define $(\Pi_2,\mathrm{P}_2)$ = path-effects$(\pi_2,I)$
     if $\mathrm{P}_2 \equiv \varnothing$ then
       define $t_{low}$ = immutable-lower-bound$(t_1,I)$
       define $t_{up}$ = immutable-upper-bound$(t_2,I)$
       define $\pi_3$ as the longest path in $\Pi_2$ (others are just prefixes of $\pi_3$)
       define $\pi_4$ =
         match $(t_{low},t_{up})$ with
         | (nothing,nothing) $\longrightarrow (\pi_3 \oplus [..]).m$
         | $(t_{low}$,nothing$) \longrightarrow (\pi_3 \oplus [t_{low}..]).m$
         | (nothing,$t_{up}) \longrightarrow (\pi_3 \oplus [..t_{up}]).m$
         | $(t_{low},t_{up}) \longrightarrow (\pi_3 \oplus [t_{low}..t_{up}]).m$
       return $(\{\pi_4\} \cup \Pi_2,\varnothing)$
     else return $(\Pi_2,\{$region-of-path$(\pi_1)\} \cup \mathrm{P}_2)$

define call-effects:
  input function f with parameters $x_1..x_n$ called with arguments $t_1..t_n$, and invariant I
  output set of immutable paths and regions accessed by $f(t_1..t_n)$
  define $(paths_1,regions_1)$ = function-effects(f)
  define $paths_2$ as the empty set
  define $regions_2$ as the current function instances for $regions_1$
  for each $\pi_1 \in paths_1$ do
    substitute f parameters by call arguments in $\pi_1$
    add path-effects$(\pi_1,I)$ to $(paths_2,regions_2)$
  done
  return $(paths_2,regions_2)$

define function-effects:
  input function f and invariant I
  output set of immutable paths and regions accessed by f
  define effects = empty set
  for each path $\pi$ accessed in f do add path-effects$(\pi_1,I)$ to effects done
  for each call $g(t_1..t_n)$ in f do add call-effects$(g(t_1..t_n),I)$ to effects done
  return effects
```

Figure 6.7: Computation of effects

```
define locations-of-region:
  input region $\rho$
  output set of locations containing all interface paths that belong to region $\rho$

define equivalent-locations:
  input set S of paths and regions
  output equivalent set of locations
  define locations = empty set
  for each path $\pi \in S$ do add $\pi$ to locations done
  for each region $\rho \in S$ do add locations-of-region($\rho$) to locations done
  return locations

define regions-may-overlap:
  input regions $\rho_1$ and $\rho_2$
  output whether $\rho_1$ and $\rho_2$ may overlap or not
  if $\rho_1$ and $\rho_2$ cannot overlap due to type safety then return false
  else if $\rho_1$ or $\rho_2$ is internal to a function then return false
  else return true

define generate-function-precondition:
  input function f and invariants I
  output separation precondition for function f
  define effects = function-effects(f,I)
  define locations = equivalent-locations(effects)
  define condition = true
  for each $(\lambda_1,\lambda_2) \in$ locations $\times$ locations such that $\lambda_1 \not\equiv \lambda_2$ do
    define $\pi_1$ = path-of-location($\lambda_1$) and $\pi_2$ = path-of-location($\lambda_2$)
    define $\rho_1$ = region-of-path($\pi_1$) and $\rho_2$ = region-of-path($\pi_2$)
    if $\rho_1 \not\equiv \rho_2 \wedge$ regions-may-overlap($\rho_1$,$\rho_2$) then
      add conjunct separated($\lambda_1$,$\lambda_2$) to condition
  done
  return condition
```

Figure 6.8: Generation of function preconditions for separation

```
  glob = *y + 1;
}
```

On this function, we compute effects

$$\textit{function-effects}(\texttt{bad\_regions}) \equiv \texttt{*x}, \texttt{*y}, \texttt{glob},$$

where all three paths have a different region, which leads to precondition

$$\textit{separated}(\texttt{x}, \texttt{y}, \texttt{\&glob}).$$

We recall too the code of function `alias_free`:

```
void alias_free(struct S *x, struct S *y) {
  x→m = 0;
  //@ assert y→m ≡ 1;
}
```

On this function, we compute effects

$$\textit{function-effects}(\texttt{alias\_free}) \equiv \texttt{x}\rightarrow\texttt{m}, \texttt{y}\rightarrow\texttt{m}$$

where both paths have a different region, which leads to precondition

$$\textit{separated}(\texttt{x}, \texttt{y}).$$

Separation preconditions make Talpin's alias analysis completely modular, thus solving problem (ii).

6.3.3 Complete Region Inference for Interference-Free Programs

Beyond Typing The separation preconditions just found are not only useful for making Talpin's analysis modular. They also provide a solution to express separation logically when expressing it by typing fails. *E.g.*, take again function `bad_regions` in a new context:

```
int glob;

/*@ requires \valid(x) ∧ \valid(y) ∧ \separated(x,y,&glob);
  @ ensures *x ≡ *y + 1 ∧ glob ≡ *y + 1;
  @*/
void bad_regions(int *x, int *y) {
  *x = *y + 1;
  glob = *y + 1;
}

void main() {
  int loc[2];
  int *p = &glob;
  p = loc;
  bad_regions(p,p+1);
}
```

Region typing asks that the regions of `*x`, `*y` and `glob` are all different. This is not the case with the call to `bad_regions` on line 15, as `*p` and `*(p+1)` both have the same region, which is also the region of `glob`. In this case, the separation condition that guarantees soundness of Talpin's analysis can be expressed as precondition

$$\mathit{separated}(\texttt{x}, \texttt{y}, \texttt{\&glob}). \tag{6.1}$$

Since it is indeed the case that `p`, `p+1` and `&glob` are separated pointers when `bad_regions` is called, the program can still be checked, although memory separation cannot completely be checked by typing.

Notice that this use of Talpin's region inference requires one to refine the translation from JESSIE to WHY presented in Section 6.2, because the partitioning of memory into regions is not the same in the caller and the callee. *E.g.*, `bad_regions` must be translated into two different functions: a function f_1 where the regions of `*x`, `*y` and `glob` are all different, as previously, and a function f_2 where `*x`, `*y` and `glob` are of the same region, and the non-interference is guaranteed by precondition 6.1. The contract for function `bad_regions` is still checked on f_1, while f_2 is a function without body but the same contract as f_1, except for the added separation condition, whose purpose is to be called where f_1 cannot.

In fact, our technique is complete on interference-free programs, where pairs of locally non-overlapping paths do not access overlapping locations.

Theorem 5 *Given a complete* JESSIE *program $\mathcal{P}$, if $\mathcal{P}$ is checked safe by a combination of* JESSIE *analyses using function paths-may-overlap defined in Section 6.2 with Talpin's regions or deductive verification of the corresponding* WHY *program, and the separation preconditions generated by function generate-function-precondition in Figure 6.8 are proved at each function call on the* WHY *program, then program $\mathcal{P}$ is indeed safe.*

Proof. The separation precondition generated by function *generate-function-precondition* ensures that all accesses to different regions inside a function f correspond indeed to non-overlapping locations. Since JESSIE analyses described in Chapter 4 and deductive verification of WHY programs only refer to these locations, it is correct to assume they belong to different regions when checking the safety and contract of function f. □

Illustration on String Copy The following implementation is the usual implementation of C standard function `strcpy` that copies a source string `src` to a destination buffer `dest`:

```
char *strcpy(char *dest, const char *src) {
  char *s = dest;
  while (*s++ = *src++) ;
  return dest;
}
```

Given a very coarse analysis of effects that does not provide any immutable bounds, we generate the separation precondition

$$\mathit{separated}(\texttt{dest} + (..), \texttt{src} + (..)),$$

where ACSL term $s + (..)$ denotes the set of pointers that can be obtained by pointer arithmetic from pointer s in the same memory block. Given more precise immutable bounds provided by abstract interpretation, we get the weaker precondition

$$\mathit{separated}(\texttt{dest} + (0..\mathit{strlen}(\texttt{src})), \texttt{src} + (0..\mathit{strlen}(\texttt{src})),$$

where strlen is the logic function at the center of our logical model of strings described in Section 8.1.1. The results of our experiments on safety checking of function `strcpy` and related string functions are reported in Section 8.2.1.

6.3.4 Refined Region Inference

As already noted by Reynolds [152], asking for the separation of all pairs of regions is overly restrictive. Regions that are only read cannot interfere, thus there is no need to ask for separation of these read-only regions, or *passive phrases* in Reynolds's terminology. It is illustrated by function `read_regions` below, where Talpin's analysis infers precondition $\mathit{separated}(\texttt{*x}, \texttt{*y})$. In fact, function `read_regions`'s behavior does not depend on the possible aliasing of its parameters, as it only reads the corresponding regions.

```
//@ requires separated(*x,*y);
int read_regions(int *x, int *y) {
  return *x + *y;
}
```

To avoid generating undue separation conditions between regions only read, we redefine variants of function *path-effects*: function *path-read-effects* returns the set of immutable paths and regions corresponding to all but the last access on the path, while function *path-read-or-write-effects* returns the set of immutable paths and regions corresponding to the last access on the path.

Then, *function-effects* presented in Figure 6.7 can be obviously refined into *function-read-effects* and *function-write-effects* that over-approximate respectively the set of locations read and written by the function, by distinguishing those paths that originate in left-hand side of assignments. From these functions, it is possible to redefine function *generate-function-precondition* so that only pairs of paths where one path at least is potentially written are required to be separated. This is the meaning of line 7 in Figure 6.10.

With this improved algorithm, the separation precondition generated for function `read_regions` is simply *true*.

Theorem 6 *Given a complete* JESSIE *program $\mathcal{P}$, if $\mathcal{P}$ is checked safe by a combination of* JESSIE *analyses using function paths-may-overlap defined in Section 6.2 with Talpin's regions or deductive verification of the corresponding* WHY *program, and the separation preconditions generated by **refined** function generate-function-precondition **in Figure 6.10** are proved at each function call on the* WHY *program, then program $\mathcal{P}$ is indeed safe.*

```
define path-read-effects:
  input path $\pi_1$ and invariant $I$
  output set of immutable paths and regions only read by $\pi_1$
  match $\pi_1$ with
  | $x \rightarrow$ return empty set
  | $(\pi_2 \oplus [t_1..t_2]).m \rightarrow$ return path-effects($\pi_2$,$I$)
  | $(\pi_2 \oplus t).m \rightarrow$ return path-effects($\pi_2$,$I$)
  | $\pi_2.m \rightarrow$ return path-effects($\pi_2$,$I$)

define path-read-or-write-effects:
  input path $\pi_1$ and invariant $I$
  output set of immutable paths and regions possibly read or written by $\pi_1$
  define $\rho_1$ = region of $\pi_1$
  match $\pi_1$ with
  | $x \rightarrow$ if $x$ is not modified in the current function then return $(x,\varnothing)$
        else return $(\varnothing,\rho_1)$
  | $(\pi_2 \oplus [t_1..t_2]).m \rightarrow$
     define $t_{low}$ = immutable-lower-bound($t_1$,$I$)
     define $t_{up}$ = immutable-upper-bound($t_2$,$I$)
     define (paths$_2$,regions$_2$) = path-effects($\pi_2$,$I$)
     if regions$_2$ is empty then
       define $\pi_3$ as the longest path in paths$_2$ (others are just prefixes of $\pi_3$)
       define $\pi_4$ =
         match $(t_{low},t_{up})$ with
         | (nothing,nothing) $\rightarrow (\pi_3 \oplus [..]).m$
         | $(t_{low}$,nothing) $\rightarrow (\pi_3 \oplus [t_{low}..]).m$
         | (nothing,$t_{up}$) $\rightarrow (\pi_3 \oplus [..t_{up}]).m$
         | $(t_{low},t_{up}) \rightarrow (\pi_3 \oplus [t_{low}..t_{up}]).m$
       return $\{\pi_4\}$
     else return $\{\rho_1\}$
  | $(\pi_2 \oplus t).m \rightarrow$ return path-read-or-write-effects($(\pi_2 \oplus [t..t]).m$,$I$)
  | $\pi_2.m \rightarrow$ return path-read-or-write-effects($(\pi_2 \oplus [0..0]).m$,$I$)
```

Figure 6.9: Computation of refined read/write effects

```
define generate-function-precondition:
  input function $f$ and invariants $I$
  output separation precondition for function $f$
  define reads = equivalent-locations(function-read-effects($f$,$I$))
  define writes = equivalent-locations(function-write-effects($f$,$I$))
  define condition = true
  for each $(\lambda_1,\lambda_2) \in$ writes $\times$ (writes $\cup$ reads) such that $\lambda_1 \not\equiv \lambda_2$ do
    define $\pi_1$ = path-of-location($\lambda_1$) and $\pi_2$ = path-of-location($\lambda_2$)
    define $\rho_1$ = region-of-path($\pi_1$) and $\rho_2$ = region-of-path($\pi_2$)
    if $\rho_1 \not\equiv \rho_2 \wedge$ regions-may-overlap($\rho_1$,$\rho_2$) then
      add conjunct separated($\lambda_1$,$\lambda_2$) to condition
  done
  return condition
```

Figure 6.10: Generation of refined function preconditions for separation

Proof. The separation precondition generated by the refined function *generate-function-precondition* ensures that all accesses to different regions inside a function f correspond indeed to non-overlapping locations, except for those regions that are only read. Then, JESSIE analyses described in Chapter 4 and deductive verification of WHY programs can only wrongly assume overlapping locations to be of different regions when they are only read. Thus these locations keep their original value from the start of the function, and their overlap is not a problem. Therefore, it is correct to assume they belong to different regions when checking the safety and contract of function f. □

6.3.5 Incompleteness of Refined Region Inference

Refined pre- and postconditions do not completely solve problem (iii), incompleteness of Talpin's alias analysis. As mentioned in Section 6.1, there are many valid uses of aliasing in C. For those uses where aliasing is not explicit in the function body, and where interference may be possible between aliased paths, Talpin's analysis may wrongly assume separation. It is the case for function `swap` below.

```
/*@ requires \valid(x) ∧ \valid(y);
  @ ensures *x ≡ \old(*y) ∧ *y ≡ \old(*x);
  @*/
void swap(int *x, int *y) {
  int tmp = *x;
  *x = *y;
  *y = tmp;
}
```

Our technique generates precondition $\mathit{separated}(\mathtt{x},\mathtt{y})$ for function `swap`, which allows us to prove `swap` postcondition. This is a stronger precondition than necessary, as `swap` contract is still correct for aliased parameters. Indeed, a call to `swap(x,x)` does ensure that $\star\mathtt{x} \equiv \mathit{old}(\star\mathtt{x})$ when `swap` returns.

For those cases where our technique generates stronger separation conditions than necessary, a solution would be to manually annotate function parameters with explicit regions, much as what is done in Cyclone [104], as in

```
void swap(int *R x, int *R y);
```

Then, it is sufficient to take these annotations into account in Talpin's alias analysis to prevent the generation of wrong separation preconditions by our analysis.

6.4 Other Related Work

Alias Analyses Based on Regions The possibility of dividing memory into regions with the results of a type-based alias analyses dates back to the work of Talpin and Jouvelot on higher-order functional languages [167, 168]. The purpose of their work is to compute effects, so overlap between regions is allowed, which makes it easy to compute regions with a context-sensitive analysis. Tofte and Talpin apply this analysis to perform static memory allocation [170]. Again, overlap between regions is not a concern.

Steensgaard's alias analysis [165] is the global context-insensitive counterpart of Talpin's local context-sensitive analysis. It is a real alias analysis, meaning that different regions truly cannot overlap. It is the best known scalable alias analysis, but, being $\star$-insensitive, its precision is low. Liang and Harrold present a context-sensitive variant of Steensgaard's alias analysis to improve its precision [126, 114]. Contrary to Talpin's analysis, they merge callee's regions when they correspond to the same region in the caller, possibly losing some precision. Hubert and Marché have chosen instead to fail in this case, thus gaining in precision at the cost of completeness [94]. We manage to retain this precision and still be complete by generating function contracts to be checked by deductive verification.

Alias Control Techniques Alias control techniques have been pioneered by Reynolds in his work on syntactic control of interference [152], where collections play the role of regions in our work. This notion has been granted a keyword, `restrict`, in C99 standard [98], that conveys the programmer's "guarantee" that a pointer is the unique reference on some memory. Various authors have described annotation-based systems to help programmers specify pointer separation [4, 1, 111]. Our treatment of separation with a dedicated first-order predicate is inspired from these works. It is simple enough that inferring sufficient separation preconditions is possible and general-purpose automatic theorem provers correctly handle our separation annotations.

Heap and Shape Analyses It may come to a surprise that we do not need a deeper understanding of the heap to analyze programs with lists, trees, or other pointer-based data structures. This is because we only consider here safety checking, which is not so much concerned with the shape of the heap, contrary to program termination and verification of behavioral properties. In particular, we do not relate to separation logic or shape analysis. Calcagno *et al.* [32] present an analysis to infer sufficient preconditions for list manipulating programs.

6.5 Chapter Summary

We presented a simple criterion for modular analysis and proof of functions in the presence of aliasing. Enforcing it prevents interfering accesses to memory due to aliases, thus enabling many optimizations and simplifications in analysis and proof. This criterion is both easy to understand from a user point of view, and automatically checkable thanks to the generation of dedicated annotations.

This criterion allowed us to lift the annotation inference method presented in the previous chapter in an alias-free context to fit the fully aliased context. We showed how a function could be analyzed only once, while the results of this analysis could be used in different aliasing contexts.

Chapter 7

Programs with Unions and Casts

Contents

In this chapter, we describe how to take unions and casts into account in deductive verification without resorting to a completely untyped memory model.

Section 7.1 considers the special case of prefix casts, the most common form of pointer casts in C. It extends the JESSIE language so that C programs with prefix casts can be translated to type-safe JESSIE programs. Then, it extends the techniques presented in Chapter 6 to programs with this restricted form of pointer casts.

Section 7.2 considers the special case of moderated unions, the most common form of unions in C, which divide into discriminated unions and byte-level unions. It extends the JESSIE language so that C programs with discriminated unions can be translated to type-safe JESSIE programs, and C programs with byte-level unions can be translated to type-safe JESSIE programs based upon a different memory model. Then, it extends the techniques presented in Chapter 6 to programs with these restricted forms of unions.

Section 7.3 considers the cases of C unions and casts not treated before. It defines a locally untyped memory model in which these unions and casts can be interpreted. This local memory model relates to the global component-as-array memory model, so that the effect of a union or cast does not propagate beyond function boundaries. Then, it extends the techniques presented in Chapter 6 to C programs with arbitrary unions and pointer casts.

Finally, Section 7.4 shows how this new interpretation relates to other work.

7.1 Prefix Casts

A *prefix cast* in C is a pointer cast between two structure types such that the fields of one structure form a prefix of the fields of the other structure. Due to the freedom left by the C standard to compilers w.r.t. layout of structures, there is no guarantee that fields in this common prefix are laid out similarly in both structures. In fact, the C standard only requires that the first field of a structure has the same address as the enclosing structure. In only one special case, when structures belong to a same union, the C standard requires that their common prefix is laid out similarly. However, it is generally supported by the Application Binary Interfaces (ABI) implemented in C compilers, that give the same layout to structures on a common prefix. In our case, the layout of each structure is computed during the translation from C to CIL, which makes it easy to restrict prefix casts to those cases where fields in the common prefix are exactly laid out at the same offsets.

Such casts are commonly used in C to encode downcasts and upcasts as usually defined in object-oriented languages. In 1999, Siff, Chandra, Reps *et al.* noticed the prevalence of such pointer casts in telecommunication C programs [161, 35]. They define a notion of physical subtyping between structures to support the view of prefix casts as upcasts and downcasts (for an example, see function `get_color` in the following). An upcast corresponds to a prefix cast where the destination structure fields form a prefix of the source structure fields. In particular, casting to `void*` counts as an upcast, since the destination type has no fields. A downcast corresponds to a prefix cast where the source structure fields form a prefix of the destination structure fields. Authors of the safe compiler CCured claim that such upcasts and downcasts, together with discriminated unions, account for 99% of pointer casts in C programs [48], based on their experience at compiling millions loc of open-source C systems code.

7.1.1 Extending JESSIE with Subtyping

Figure 7.1 presents an extension of JESSIE types to support named inheritance, in which a structure may be defined to extend another structure. As usual, a structure inherits the fields of the structure it extends. These inherited fields can be considered as a prefix of the ordered list of fields of a structure. Inheritance relations form a tree-like directed graph. A structure is a subtype of the structure it extends, which is the only possible subtyping relation in JESSIE.

Figure 7.2 presents the abstract syntax of JESSIE propositions to support subtyping. Operator **::** expresses that a term has a specific dynamic type, that should be a subtype of its static type. Operator $\triangleleft$ expresses that the dynamic type of a term is a subtype of a given type.

Figure 7.3 presents the extended semantics supporting subtyping. It requires the addition of a new part to the state model described in Section 2.2.3: **DynType** maps memory block labels to their dynamic structure type. The allocation instruction additionally updates the table for dynamic types.

7.1.2 Crawling the Type Hierarchy

JESSIE Translation Function `get_color` is an example of a use of a prefix cast (from Siff *et al.* [161]). Although function `get_color` takes a parameter of static type `Point*`, it should really be of type `ColorPoint*`, as expressed in the function precondition, so that its `color` field can be retrieved.

```
typedef struct { short x; short y; } Point;
typedef struct { short x; short y; int color; } ColorPoint;

//@ requires \valid(pt) ∧ pt ◁ ColorPoint*;
int get_color(Point *pt) {
  return ((ColorPoint*)pt)→color;
}
```

The physical subtyping between `ColorPoint` and `Point` in C can be translated into structure subtyping in JESSIE, and the prefix cast in C can be translated into a hierarchical cast in JESSIE, as shown by the translation of this program in JESSIE:

```
struct Point = { int16 x; int16 y }
struct ColorPoint extends Point = { int32 color }

requires valid(pt) ∧ typeof(pt) ◁ type(ColorPoint)
int32 get_color(Point[..] pt) =
  return (pt ▷ ColorPoint[0]).color
```

Figure 7.4 illustrates this situation, in which the same fields can be seen as belonging to a `Point` or a `ColorPoint`.

It should be noted that such hierarchical casts should restrict the set of allowed offsets to the only offset `0` to prevent pointer arithmetic on the resulting pointer. This ensures that we retain type safety. Figure 7.5 presents the semantics of such pointer casts.

Theorem 7 *A well-typed* JESSIE *program with subtyping executes without any error (possibly not terminating), on an imaginary machine with an infinite memory, if integer checks, memory checks and type checks defined respectively in Sections 3.1.1, 4.1.2 and Figure 7.5 hold.*

Proof Sketch. A JESSIE program with subtyping is equivalent to a similar JESSIE program without subtyping, where all inherited fields are explicitly mentioned. Theorem 1 presented in Section 4.1.2 already showed that validity of checks implies program safety, not even considering type checks. Thus, the desired implication trivially holds. Type checks are only added to facilitate the translation from JESSIE to WHY. □

JESSIE Analysis Much like pointer arithmetic, pointer casts do not change the region of pointers: `p` and `p ▷ T` point to the same region. So, function *paths-may-overlap* defined in Section 6.2 correctly over-approximates overlap for programs with subtyping. Thus, the techniques for inferring annotations and checking safety presented in Chapter 6 still apply to programs with subtyping.

struct-def	::=	`struct` *id* (`extends` *id*)$^?$ = *fields*	structure def

Figure 7.1: Grammar of JESSIE types with inheritance

prop	::=	...	
	|	*term* **: :** *type*	dynamic typing
	|	*term* $\triangleleft$ *type*	subtyping relation

Figure 7.2: Grammar of JESSIE propositions for subtyping

$$\frac{[\![t]\!] = n \quad 0 \leq n \quad l \notin dom(\textbf{Alloc}) \qquad \forall\, i.\, 0 \leq i < n \times sizeof(S) \rightarrow a + i \textit{ not allocated}}{\{x := \text{new } S[t]\} \vdash \textbf{DynType} \Rightarrow \textbf{DynType}[l \mapsto S]}\ \text{NEW}$$

Figure 7.3: Semantics of JESSIE constructs for subtyping

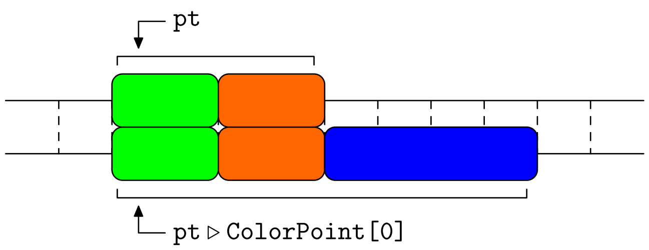

Figure 7.4: Prefix Cast

$$\frac{[\![t]\!] = (l, a, i, min, max) \quad \textbf{DynType}(l) \triangleleft S \qquad 0 \leq i \quad i + sizeof(S) \leq \textbf{Alloc}(l)}{[\![t \triangleright S[0]]\!] = (l, a, i, 0, 0)}\ \text{PREFIX-CAST}$$

Figure 7.5: Semantics of JESSIE hierarchical casts

WHY Translation The translation already presented in Section 5.1.1 for type-safe programs is still applicable for JESSIE programs with subtyping. Additional tag tables track the dynamic types of memory blocks, much like allocation tables track their size, as already shown in the context of tool Krakatoa for deductive verification of JAVA programs [129].

7.2 Moderated Unions

A *moderated union* in C is a union whose field addresses are not taken. This makes it mandatory to pass through the surrounding union to access any field inside, thus giving the union a role of moderator. There are two kinds of moderated unions, depending on the kind of accesses to union fields:

- discriminated unions, in which all fields can always be written, but only the last field written should be read (writing thus performs a kind of *strong update*);
- byte-level unions, in which all fields can always be written and read, which means a field read is not necessarily the last one written.

In the following, we show how to translate discriminated unions and byte-level unions in JESSIE. Both unions in JESSIE are defined as a collection of structures, as indicated in Figure 7.6. No such structure should be involved in a hierarchy of structures as defined in 7.1. A union name can be used anywhere a structure name is allowed in JESSIE.

7.2.1 Discriminated Unions in JESSIE

A discriminated union in C is a moderated union type such that a field of the union is read only if it is the last field written to. It is similar to variant types in OCAML or discriminated unions in Ada, in that no byte-level reinterpretation is needed to get the value of a union field. But, contrary to these safe constructs, discriminated unions rely on programming idioms which can be misused by the programmer. Typically, C programmers use discriminated unions inside fields of structures, so that the value of another field indicates which field of the union is currently set [103].

JESSIE Translation Function `compact` is an example of a use of a discriminated union `Item`. At function entry, its parameter `x` should have its `net` field set, while at function exit, `x` has its `gross` field set.

```
typedef struct { short net_price; char tax; } Net;
typedef struct { short gross_price; } Gross;
typedef union { Net net; Gross gross } Item;

/*@ requires \valid(x) ∧ x :: Net*;
  @ ensures x :: Gross*
  @          ∧ x→gross.gross_price
  @               ≡ \old(x→net.net_price + x→net.tax);
  @*/
void compact(Item *x) {
  short price = x→net.net_price + x→net.tax;
  x→gross.gross_price = tmp;
}
```

Discriminated union `Item` in C can be translated into JESSIE. All three types `Net`, `Gross` and `Item` are designated collectively as discriminated union types. Figures 7.7 and 7.8 illustrate this situation.

```
struct Net = { int16 net_price: 16; int8 tax: 8; unit _: 8 }
struct Gross = { int16 gross_price: 16; unit _: 16 }
discrunion Item = [ Net | Gross ]

requires valid(x) ∧ x :: Net[..]
ensures x :: Gross[..]
        ∧ (x ▷ Gross[..]).gross_price ≡
           \old((x ▷ Net[..]).net_price + (x ▷ Net[..]).tax);
unit compact(Item[..] x) =
  int16 price
  price := (x ▷ Net[..]).net_price + (x ▷ Net[..]).tax
   (x ▷ Gross[..]).gross_price := price
```

Figure 7.9 presents the semantics of such pointer casts, which is similar to the semantics of type-safe casts presented in Section 5.1.1, with the addition of a type check, as in the semantics of prefix casts presented in Section 7.1.2. Figure 7.10 presents the semantics of assignment to fields of discriminated unions. Indeed, assigning to a field of a discriminated union changes the dynamic type of the corresponding memory block. In rule ASSIGN-VFIELD, t_1 is obtained from a pointer of discriminated union type T with a succession of embedded fields and pointer arithmetic. Thus, T is the enclosing discriminated union type. Notice that in this semantics, embedded fields cannot have discriminated union type, since the dynamic type of the memory block for an embedded field is the dynamic type of the enclosing structure.

Theorem 8 *A well-typed* JESSIE *program with subtyping and discriminated unions executes without any error (possibly not terminating), on an imaginary machine with an infinite memory, if integer checks, memory checks and type checks defined respectively in Sections 3.1.1, 4.1.2, Figure 7.5 and Figure 7.9 hold.*

Proof Sketch. Theorem 7 presented in Section 7.1.2 already shows this property for well-typed JESSIE programs with subtyping. Discriminated unions only restrict casts and assignments in order to facilitate the translation from JESSIE to WHY, thus Theorem 7 still holds. □

JESSIE Analysis Function *paths-may-overlap* presented in Section 6.2 for type-safe programs with aliasing must be modified to take into account the possible overlap between fields of a union. This new definition is presented in Figure 7.11. Previously, case (2.2) concluded that two memory locations accessing different fields could never overlap. It now refines into cases (2.2') which concludes that two memory locations accessing different fields can overlap only if they correspond to overlapping fields in enclosing locations of the same union type that possibly overlap.

hierarchy-def	::=	`discr-union` *id* = *id**	discriminated union def
	|	`plain-union` *id* = *id**	byte-level union def

Figure 7.6: Grammar of JESSIE unions

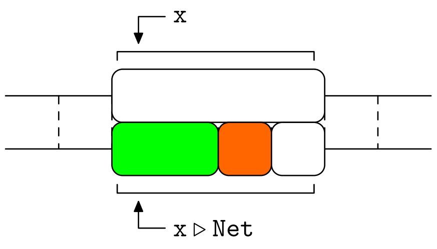

Figure 7.7: Function entry

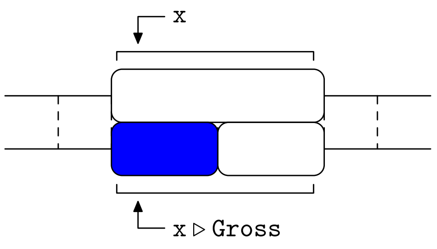

Figure 7.8: Function exit

$$\frac{[\![t]\!] = (l, a, i, min_1, max_1) \quad \textbf{DynType}(l) \lhd S}{[\![t \rhd S[..]]\!] = (l, a, i, min_1, max_1)}\ \textsc{VPTR-CAST}$$

$$\frac{\begin{array}{c}[\![t]\!] = (l, a, i, min_1, max_1) \quad min_1 \leq min_2 \quad \textbf{DynType}(l) \lhd S \\ 0 \leq i + min_2 \times sizeof(S)\end{array}}{[\![t \rhd S[min_2..]]\!] = (l, a, i, min_2, max_1)}\ \textsc{LOW-VPTR-CAST}$$

$$\frac{\begin{array}{c}[\![t]\!] = (l, a, i, min_1, max_1) \quad max_2 \leq max_1 \quad \textbf{DynType}(l) \lhd S \\ i + (max_2 + 1) \times sizeof(S) \leq \textbf{Alloc}(l)\end{array}}{[\![t \rhd S[..max_2]]\!] = (l, a, i, min_1, max_2)}\ \textsc{UP-VPTR-CAST}$$

$$\frac{\begin{array}{c}[\![t]\!] = (l, a, i, min_1, max_1) \quad min_1 \leq min_2 \quad max_2 \leq max_1 \\ \textbf{DynType}(l) \lhd S \quad 0 \leq i + min_2 \times sizeof(S) \\ i + (max_2 + 1) \times sizeof(S) \leq \textbf{Alloc}(l)\end{array}}{[\![t \rhd S[min_2..max_2]]\!] = (l, a, i, min_2, max_2)}\ \textsc{BOUND-VPTR-CAST}$$

Figure 7.9: Semantics of JESSIE discriminated union casts

$$\frac{\begin{array}{c} t_1 : S[..] \quad [\![t_1]\!] = (l, a, i, min, max) \quad min \leq 0 \leq max \\ [\![t_2]\!] = v \quad 0 \leq i \quad i + sizeof(S) \leq \textsf{Alloc}(l) \\ T \textit{ enclosing discriminated union type} \end{array}}{\begin{array}{l} \{t_1.m := t_2\} \vdash \textsf{Heap}, \textsf{DynType} \Rightarrow \\ \quad \textsf{Heap}[((a+i) \times 8 + bitoffsetof(m), bitsizeof(m)) \mapsto to\text{-}bits_m(v)] \\ \quad \textsf{DynType}[l \mapsto T] \end{array}} \text{ ASSIGN-VFIELD}$$

Figure 7.10: Semantics of JESSIE discriminated union assignment

```
define union-path:
  input path $\pi_1$
  output a tuple ($\pi_2$,min,max) for a union type, nothing otherwise
  match $\pi_1$ with
  | x $\rightarrow$ if x is of union type then return (x,0,0) else return nothing
  | ($\pi_2 \oplus [t_1..t_2]$).m $\rightarrow$
     if m is an embedded field then
       match union-path($\pi_2$) with
       | ($\pi_3$,$t_3$,$t_4$) $\rightarrow$ ($\pi_3$,$t_3$+$t_1$$\times$sizeof($\pi_2$)+offsetof(m),$t_4$+$t_2$$\times$sizeof($\pi_2$)+offsetof(m))
       | nothing $\rightarrow$ nothing
     else return ($\pi_1$,0,0)

define intervals-may-overlap:
  input integer intervals ($min_1$,$max_1$) and ($min_2$,$max_2$)
  output whether input intervals overlap
  return $min_1 \leq max_2 \wedge min_2 \leq max_1$

define paths-may-overlap:
  input paths $\pi_1$ and $\pi_2$
  output whether $\pi_1$ and $\pi_2$ represent overlapping locations
  match ($\pi_1$,$\pi_2$) with
  (1)      | (x,x) $\rightarrow$ return true
  (2.1”)   | ((_$\oplus$_).m,(_$\oplus$_).m) $\rightarrow$ return region-of-path($\pi_1$) $\equiv$ region-of-path($\pi_2$)
  (2.2’)   | ((_$\oplus$_).m,(_$\oplus$_).n) $\rightarrow$
              match (union-path $\pi_1$,union-path $\pi_2$) with
              | (($\pi_3$,$off_1$,$off_2$),($\pi_4$,$off_3$,$off_4$)) $\rightarrow$
                 return paths-may-overlap($\pi_3$,$\pi_4$)
                        $\wedge$ intervals-may-overlap
                          (($off_1$,$off_2$+sizeof(m)$-$1),($off_3$,$off_4$+sizeof(n)$-$1))
              | _ $\rightarrow$ return false
  (3)      | (x,_) | (_,x) $\rightarrow$ return false
```

Figure 7.11: Overlap of paths with unions

WHY Translation The translation presented in Section 7.1.2 for type-safe programs with subtyping is still applicable for JESSIE programs with unions. Assignment to a field inside a union modifies the dynamic type of the corresponding memory block, which translates into an update of the tag table.

7.2.2 Byte-Level Unions in JESSIE

A byte-level union in C is a moderated union type such that a field read is not necessarily the last one written. Therefore, the value written may have to be reinterpreted in a different type.

JESSIE Translation Function `init` is an example of use of a byte-level union `Sock`. It is convenient to set field `all` to zero instead of individually setting each sub-field of field `s` to zero. This amounts to exactly the same result, as `all` and `s` have the same size, and C standard mandates that the all-zeros bit-pattern is a valid representation of integer zero for any type of integer. Then, it should be possible to read the value pointed-to by `x` through its `s` field, as in the postcondition.

```
typedef struct {
  short socket_num; char flags; char filters;
} Socket;
typedef union { int all; Socket s; } Sock;

/*@ requires \valid(x);
  @ ensures x→s.socket_num ≡ 0
  @         ∧ x→s.flags ≡ 0 ∧ x→s.filters ≡ 0;
  @*/
void init(Sock *x) {
  x→all = 0;
}
```

Byte-level union `Sock` can be translated into a union type in JESSIE. All three types `Int32P`, `Socket` and `Sock` are designated collectively as union types. Figure 7.12 illustrates this situation.

```
struct Socket =
  { int16 socket_num: 16; int8 flags: 8; int8 filters: 8 }
struct Int32P = { int32 int32m: 32 }
plainunion Sock = [ Int32P & Socket ]

requires valid(x)
ensures (x ▷ Socket[..]).socket_num ≡ 0
        ∧ (x ▷ Socket[..]).flags ≡ 0
        ∧ (x ▷ Socket[..]).filters ≡ 0
unit init(Sock[..] x) =
  (x ▷ Int32P).int32m := 0
```

Figure 7.13 presents the semantics of such pointer casts, which is similar to the semantics of type-safe casts presented in Section 5.1.1, without the requirement that the argument pointer type and the destination type refer to the same structure.

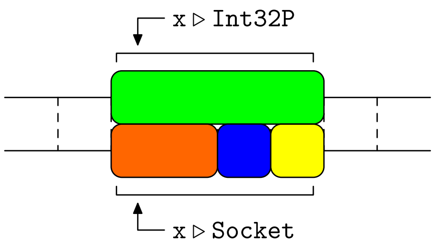

Figure 7.12: Byte-level union

$$\frac{[\![t]\!] = (l, a, i, min_1, max_1)}{[\![t \triangleright S[..]]\!] = (l, a, i, min_1, max_1)}\ \textsc{UPTR-CAST}$$

$$\frac{[\![t]\!] = (l, a, i, min_1, max_1) \quad min_1 \leq min_2 \quad 0 \leq i + min_2 \times sizeof(S)}{[\![t \triangleright S[min_2..]]\!] = (l, a, i, min_2, max_1)}\ \textsc{LOW-UPTR-CAST}$$

$$\frac{[\![t]\!] = (l, a, i, min_1, max_1) \quad max_2 \leq max_1 \quad i + (max_2 + 1) \times sizeof(S) \leq \textbf{Alloc}(l)}{[\![t \triangleright S[..max_2]]\!] = (l, a, i, min_1, max_2)}\ \textsc{UP-UPTR-CAST}$$

$$\frac{[\![t]\!] = (l, a, i, min_1, max_1) \quad min_1 \leq min_2 \quad max_2 \leq max_1 \quad 0 \leq i + min_2 \times sizeof(S) \quad i + (max_2 + 1) \times sizeof(S) \leq \textbf{Alloc}(l)}{[\![t \triangleright S[min_2..max_2]]\!] = (l, a, i, min_2, max_2)}\ \textsc{BOUND-UPTR-CAST}$$

Figure 7.13: Semantics of JESSIE byte-level union casts

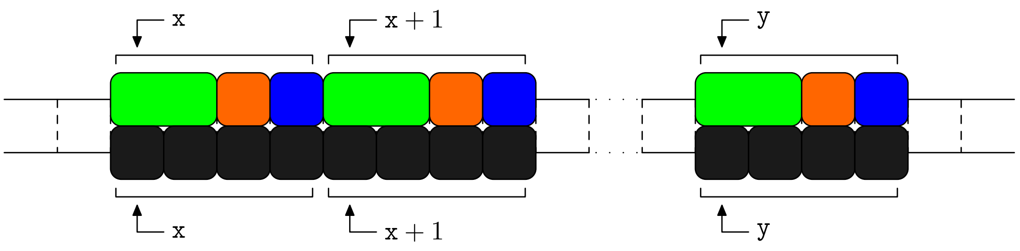

Figure 7.14: Coexisting byte-level and typed memory models

Theorem 9 *A well-typed* JESSIE *program with subtyping, discriminated unions and byte-level unions executes without any error (possibly not terminating), on an imaginary machine with an infinite memory, if integer checks, memory checks and type checks defined respectively in Sections 3.1.1, 4.1.2, Figure 7.5 and Figure 7.9 hold.*

Proof Sketch. Adding byte-level unions only restrict casts in order to facilitate the translation from JESSIE to WHY, so that Theorem 8 still holds. □

JESSIE Analysis Function *paths-may-overlap* must be modified to take into account the possible overlap between fields of a union, the same way as for a discriminated union.

WHY Translation Except for unions, the translation already presented in Section 5.1.1 for type-safe programs is still applicable for JESSIE programs with unions. Union types are replaced by bit-vectors of the appropriate size, so that any access to a field inside a union translates into an access to the corresponding bit-vector. This translation uses functions $of\text{-}bits_{\mathtt{m}}$ and $to\text{-}bits_{\mathtt{m}}$ for a field m such as those defined in Section 2.2.3.

7.2.3 Choice of Union in JESSIE

C unions can be translated either into JESSIE discriminated unions or into JESSIE byte-level unions. Although the latter are more flexible, by not restricting reads to the last field written to, they do not mix well with region inference as presented in Section 6.2. Roughly, it is not possible to give different regions to different pointer fields in a byte-level union, while it is possible for fields of a discriminated union. Thus, we choose to translate C unions into JESSIE discriminated unions whenever some field or subfield of the union has pointer type. Otherwise, C unions are translated into byte-level unions.

7.3 Other Unions and Casts

Other unions and casts between pointers in C reinterpret the bit-pattern in memory as a value a type or another. This makes it necessary to consider multiple coexisting memory models, a byte-level memory model as defined in Section 2.2.3, and various typed memory models on top of it, as sketched in Figure 7.14.

JESSIE Translation Byte-level unions and pointer casts in C are translated into pointer casts in JESSIE, as presented in Section 2.3.

JESSIE Analysis Byte-level casts in JESSIE do not preserve type safety. Thus, it is not possible anymore to use function *paths-may-overlap* presented in Section 5.1.1 for type-safe programs. Without region inference presented in Chapter 6, the very imprecise function *paths-may-overlap* defined in Section 4.2.3 must be used. With region inference, the more precise function *paths-may-overlap* presented in Section 6.2 for type-safe programs can be modified to handle programs with pointer casts. This new definition is presented in

```
define paths-may-overlap:
  input paths $\pi_1$ and $\pi_2$
  output whether $\pi_1$ and $\pi_2$ represent overlapping locations
  match $(\pi_1,\pi_2)$ with
  (1)     | (x,x) → return true
  (2.3)   | ((_⊕_)._,(_⊕_)._) → return region-of-path($\pi_1$) ≡ region-of-path($\pi_2$)
  (3)     | (x,_) | (_,x) → return false
```

Figure 7.15: Overlap of paths with regions only

Figure 7.15. Previously, case (2) concluded that two memory locations could always overlap. It now refines into case (2.3) which concludes that two memory locations can overlap only if they belong to the same region.

WHY Translation Similarly to what is done for unions in JESSIE, accesses to byte-level memory can be translated as accessing a bit-vector in WHY, with the appropriate calls to functions $of\text{-}bits_{\mathtt{m}}$ and $to\text{-}bits_{\mathtt{m}}$, to convert values between a field type and bit-vectors. This allows one to check the safety and postcondition of function `reverse_endian`, which accesses individual bytes in the representation of a short to switch them. Endianness is defined by axiomatization.

```
/*@ axiom little_endian_low_byte_short:
  @  ∀ short *s; *(char*)s ≡ *s % 256;
  @
  @ axiom little_endian_high_byte_short:
  @  ∀ short *s; *((char*)s+1) ≡ *s / 256;
  @*/

/*@ requires \valid(s);
  @ ensures *s ≡ 256 * (\old(*s) % 256) + (\old(*s) / 256);
  @*/
void reverse_endian(short *s) {
  char *c = (char*)s;
  char tmp = *c;
  *c = *(c+1);
  *(c+1) = tmp;
}
```

A first remark is that, when regions are used, only those regions accessed through casted pointers need to be interpreted in a byte-level memory model. All other regions can still be interpreted in a component-as-array memory model refined with regions. Another remark is that different regions can be interpreted as different bit-vectors, so that the separation between regions translates into WHY.

The problem with this translation is that it quickly propagates to all the program through calls [2], thus completely replacing the component-as-array memory model and annihilating the associated benefit, automatic separation of fields.

To avoid this problem, we propose to enforce the locality of byte-level accesses inside a function f, so that both f callers and f callees can be interpreted and verified independently of the byte-level accesses in f.

Isolating the byte-level memory model of f from its callers can be obtained by:

- *typed function declaration* - Although f is verified w.r.t. a byte-level memory model, another function f' should be called, which is obtained by translating the declaration of f in a typed memory model. In particular, effects of f' should be expressed in a typed memory model.
- *typed memory model accessors* - Validity of pointers in the byte-level memory model is not sufficient to ensure validity of pointers in the typed memory model, as a valid pointer of type $T[..]$ is not a valid pointer of type $S[..]$. Therefore, the body of f should be verified w.r.t. typed memory model accessors: $\mathit{offset\text{-}min}$ and $\mathit{offset\text{-}max}$ should be replaced by $\mathit{offset\text{-}min\text{-}S}$ and $\mathit{offset\text{-}max\text{-}S}$ for every structure type S.

Isolating the byte-level memory model of f from its callees can be obtained by:

- *bytes-to-types call prelude* - Before calling a function g, byte-level pieces of memory accessed by g should be translated to typed pieces of memory. In order to ensure separation of fields in the component-as-array memory model, those pieces of memory accessed in g through different types should correspond to separated locations in f. This is similar to the generation of separation preconditions in the refined Talpin's alias analysis presented in Section 6.3.
- *types-to-bytes call postlude* - After calling a function g, typed pieces of memory written to by g which correspond to byte-level pieces of memory in f should be translated back.

In function `reverse_endian` below, function `swap` is called to switch the two bytes in the representation of a short. Function `swap` does not contain any union or pointer cast, thus it is interpreted in a component-as-array memory model. Function `reverse_endian`, on the contrary, contains a cast of pointer `s`. Thus, pointers `s` and `c` are interpreted in a byte-level memory model. Before calling `swap`, a component-as-array memory model is reconstructed from the byte-level one, and effects on this model are translated back on the byte-level memory model after the call. In this case, no separation precondition is generated. We manage to completely prove the safety and the annotations in `reverse_endian` using any automatic prover among Alt-Ergo, Simplify and Z3.

```
/*@ requires \valid_range(x,0,1);
  @ ensures x[0] ≡ \old(x[1]) ∧ x[1] ≡ \old(x[0]);
  @*/
void swap(char *x) {
  char tmp = *x;
  x[0] = x[1];
  x[1] = tmp;
}

/*@ requires \valid(s);
```

```
  @ ensures *s ≡ 256 * (\old(*s) % 256) + (\old(*s) / 256);
  @*/
void reverse_endian(short *s) {
  char *c = (char*)s;
  swap(c);
}
```

The separation precondition generated by (1) the conversion between byte-level and component-as-array memory models, and (2) the region alias analysis presented in Section 6.3 may add up, as in `reverse_endian` below. The precondition (1) is *true* while the precondition (2) is that `*c` and `*(c+1)` are separated, which is true. This time, we manage to completely prove the safety and the annotations in `reverse_endian` only using automatic prover Alt-Ergo.

```
/*@ requires \valid(x) ∧ \valid(y);
  @ ensures *x ≡ \old(*y) ∧ *y ≡ \old(*x);
  @*/
void swap(char *x, char *y) {
  char tmp = *x;
  *x = *y;
  *y = tmp;
}

/*@ requires \valid(s);
  @ ensures *s ≡ 256 * (\old(*s) % 256) + (\old(*s) / 256);
  @*/
void reverse_endian(short *s) {
  char *c = (char*)s;
  swap(c,c+1);
}
```

As a slight improvement over the translation presented so far, casts that occur in the arguments of a call could be considered as casts in the surrounding function, so that the corresponding regions remain interpreted in a component-as-array memory model. This could simplify the proof of function `reverse_endian` below.

```
/*@ requires \valid(s);
  @ ensures *s ≡ 256 * (\old(*s) % 256) + (\old(*s) / 256);
  @*/
void reverse_endian(short *s) {
  swap((char*)s,(char*)s+1);
}
```

7.4 Other Related Work

Siff and others [161] collected uses of casts in large C programs. They introduce the notion of platform-dependent physical subtyping. Their algorithm emphasizes equality of field names and sub-structure boundaries for deciding subtyping, which is not needed for safe access. It is rather an additional requirement they impose as a good software engineering practice.

In [161], they use this notion of physical subtyping to statically classify casts in upcasts, downcasts or mismatch (remaining cases). Chandra and Reps use it in [35] to devise a physical type checking algorithm for C, that statically rejects programs that cannot be proved correct with respect to physical subtyping. As mentioned in their paper, their algorithm does not target programs which emulate inheritance using discriminated union and cast as we do.

Andronick, in her PhD thesis [2], treats unions and casts in Caduceus like byte-level unions, by providing an *ad hoc* synchronization function for each such case. This only applies to casts and unions on structures whose address is not taken.

Jhala, Majumdar and Xu have described in [103] an algorithm to automatically discover the type invariants that guarantee proper use of union as discriminated union in C. Although quite effective at discovering the proper invariants in large C programs, their method is limited to simple forms of invariants. Moreover, we believe that the overhead of manually annotating union types is not big when doing program verification.

Another direction was taken by Tuch, Klein and Norrish in [172] to allow reasoning about C union and cast in deductive verification. They choose to work with a byte-level memory model (described in [171]), with lifting functions providing a typed view of the heap. It is possible in their model to reason about any cast and union, even those that reinterpret typed memory through a different type, at the cost of manually providing the rules for how the lifted views of the heap change during these unsafe operations. Our structure subtyping feature is more restricted but also better suited for automatic proof. In their case, they do manual proofs in Isabelle/HOL.

7.5 Chapter Summary

Based on existing statistics of the kind of unions and pointer casts most commonly found in C programs, we propose a classification of unions and pointer casts.

A majority of them, prefix casts and discriminated moderated unions, can be directly translated at the type level, using an encoding of subtyping in Jessie. Byte-level moderated unions can be translated into Jessie unions, thus using a byte-level memory model for union fields while keeping a component-as-array memory model everywhere else. Finally, remaining unions and casts must be interpreted in a local byte-level memory model, which is not propagated to callers and callees of the function thanks to the generation of appropriate separation preconditions.

We showed how to adapt the techniques presented in previous chapters to programs with such unions and casts. In particular, we further refined our criterion for separation, so that functions can still be analyzed modularly. Independently from other functions, a function can either be analyzed and proved at the type level, or a mixed type and byte level, or even at the byte level, which has not been shown previously.

Part III

Experiments

Chapter 8

Experiments on Real C Programs

Contents

The techniques described in this thesis have been implemented in Frama-C [73], an open-source platform for the modular analysis of C programs. Target C programs are translated to CIL, then JESSIE and finally WHY, before verification conditions (VC) are generated and sent to automatic provers to prove the safety of the original C programs, as described in Figure 1.4 in Section 1.5.

In Section 8.1, we mention those implementation details needed to understand our experiments. In Section 8.2, we describe the results of applying our tool to check the safety of available string libraries. In Section 8.3, we describe the results of applying our tool to discriminate between unsafe and patched versions of open-source programs with vulnerabilities. The verification was performed completely automatically and modularly, with each function analyzed separately.

Source programs for these benchmarks as well as scripts to replay tests can be downloaded from `www.lri.fr/~moy`. Source lines of code are measured using David A.

Wheeler's *SLOCCount*. Provers are run with a time limit of 10 s on each verification condition. We present reference results obtained on a 3.19 GHz processor with 2 G RAM.

8.1 Notes of Implementation

8.1.1 Logical Model of Strings

We developed a logical model of strings to use in annotations. In particular, we define a logic function *strlen* as in Section 4.2.2 to provide a handle on the length of a string.

```
//@ logic integer strlen(char *s) reads s[0..];
```

The behavior of *strlen* should be the following:

- in general, $strlen(s)$ should be the value of the smallest non-negative index i that fits in a C `int` at which $s[i]$ is null;
- if such an index does not exist, $strlen(s)$ can have any negative value (the exact value can be left underspecified).

This behavior can be defined by the following axioms:

```
/*@ axiom strlen_pos_or_null:
  @   ∀ char* s; ∀ integer i;
  @      (0 ≤ i ≤ INT_MAX
  @       ∧ (∀ integer j; 0 ≤ j < i ⟹ s[j] ≢ '\0')
  @       ∧ s[i] ≡ '\0') ⟹ strlen(s) ≡ i;
  @
  @ axiom strlen_neg:
  @   ∀ char* s;
  @      (∀ integer i; 0 ≤ i ≤ INT_MAX ⟹ s[i] ≢ '\0')
  @      ⟹ strlen(s) < 0;
  */
```

Of course, automatic provers require the definition of auxiliary lemmas, that could be proved from the above axioms in a proof assistant.

```
/*@ lemma strlen_upper_bound:
  @   ∀ char* s; strlen(s) ≤ INT_MAX;
  @
  @ lemma strlen_before_null:
  @   ∀ char* s; ∀ integer i; 0 ≤ i < strlen(s) ⟹ s[i] ≢ '\0';
  @
  @ lemma strlen_at_null:
  @   ∀ char* s; 0 ≤ strlen(s) ⟹ s[strlen(s)] ≡ '\0';
  @
  @ lemma strlen_not_zero:
  @   ∀ char* s; ∀ integer i;
  @      0 ≤ i ≤ strlen(s) ∧ s[i] ≢ '\0' ⟹ i < strlen(s);
  @
  @ lemma strlen_zero:
  @   ∀ char* s; ∀ integer i;
  @      0 ≤ i ≤ strlen(s) ∧ s[i] ≡ '\0' ⟹ i ≡ strlen(s);
```

```
@
@ lemma strlen_sup:
@   ∀ char* s; ∀ integer i;
@       0 ≤ i ≤ INT_MAX ∧ s[i] ≡ '\0' ⟹ 0 ≤ strlen(s) ≤ i;
@
@ lemma strlen_shift:
@   ∀ char* s; ∀ integer i;
@       0 ≤ i ≤ strlen(s) ⟹ strlen(s + i) ≡ strlen(s) − i;
@
@ lemma strlen_create:
@   ∀ char* s; ∀ integer i;
@       0 ≤ i ≤ INT_MAX ∧ s[i] ≡ '\0' ⟹ 0 ≤ strlen(s) ≤ i;
@
@ lemma strlen_create_shift:
@   ∀ char* s; ∀ integer i; ∀ integer k;
@       0 ≤ k ≤ i ≤ INT_MAX ∧ s[i] ≡ '\0'
@       ⟹ 0 ≤ strlen(s+k) ≤ i − k;
@*/
```

In order to deduce useful invariants by abstract interpretation based on this *strlen* logic function, we define a predicate expressing what it means to be a string:

```
/*@ predicate valid_string(char *s) =
@   0 ≤ strlen(s) ∧ \valid_range(s,0,strlen(s));
@*/
```

After initial experiments, we realized no precise annotations could be inferred without a small amount of manual annotations which identify those pointer variables that can hold strings. In order to minimize the annotation burden, we made it possible to define declaration specifiers naming a predicate that expand into preconditions for function parameters and postconditions for function returns, similarly to what is done in SAL [84] or Deputy [184]. *E.g.*, here is a macro defining such a declaration specifier for predicate *valid-string*:

```
#define FRAMA_C_STRING __declspec(valid_string)
```

This macro can be used to decorate function signatures. *E.g.*, function `strcpy` that copies a string `src` into a buffer `dest` gets decorated as follows:

```
char *strcpy(char *ret, const char *FRAMA_C_STRING s2);
```

8.1.2 Preprocessing

We implemented in Frama-C two simple syntactic transformations:

- a cursor-to-base translation of pointers and integers;
- an insertion of assertions for string usage.

The cursor-to-base translation consists in identifying those *cursor* variables that are only defined as offsets from another *base* variable. Then, it is possible to explicit the value of such an offset as a new integer variable, such that the cursor variable can be replaced by the sum of its base variable and this offset. This applies in particular to index cursor pointers [83] which point into a buffer, as well as integer parameters used as counters (incremented or decremented).

The insertion of assertions for string usage recognizes patterns of string usage in the program and automatically inserts corresponding *hints* in the program. A hint is an assertion that can be used in our annotation inference methods, provided it is also proved, but which should not be used as target assertion to infer preconditions. Indeed, validity of hints usually relies on complex properties that can only be captured by axiomatization. Thus, our annotation inference methods are not powerful enough to generate sufficient preconditions for such hints. We generate two kinds of hints for string variables, *i.e.*, variables decorated with declaration specifier `FRAMA_C_STRING`:

- *accessing strings*: reading or writing through string variable `s` at index `i` generates hint $0 \leq \mathtt{i} \leq \mathit{strlen}(\mathtt{s})$.
- *testing string termination*: testing the nullity of string variable `s` at index `i` generates hint $\mathtt{i} < \mathit{strlen}(\mathtt{s})$ for the non-null branch and hint $\mathtt{i} \equiv \mathit{strlen}(\mathtt{s})$ for the null branch.

E.g., this automatic insertion of assertions in function `strcpy` is almost equivalent to the following hand-written annotations, where `Pre` is a predefined label in ACSL for denoting the pre-state of a function:

```
//@ requires valid_string(s2);
char *strcpy(char *ret, const char *s2) {
  char *s1 = ret;
  //@ assert 0 ≤ s2 − \at(s2,Pre) ≤ strlen(\at(s2,Pre));
  while (*s1++ = *s2++)
    //@ assert s2 − \at(s2,Pre) < strlen(\at(s2,Pre));
    /* EMPTY */ ;
  //@ assert s2 − \at(s2,Pre) ≡ strlen(\at(s2,Pre));
  return ret;
}
```

It is best seen on the real intermediate program that we generate and analyze, where `integer` is ideally the logical type of integers, but currently the largest C integer type:

```
//@ requires valid_string(s2);
char *strcpy(char *ret, const char *s2) {
  integer s1_offset = 0;
  integer s2_offset = 0;
  while (1) {
    //@ assert 0 ≤ s2_offset ≤ strlen(s2);
    char tmp = s2[s2_offset]
    ret[s1_offset] = tmp;
    s1_offset++;
    s2_offset++;
    if (tmp == 0) break;
    //@ assert s2_offset < strlen(s2);
  }
  //@ assert s2_offset ≡ strlen(s2);
  return ret;
}
```

Notice that although some tests contain pointer casts between types `void*`, `char*` and `unsigned char*`, these casts are not considered as pointer casts in JESSIE, as these types are all synonyms for `char*` in our implementation.

8.1.3 Filtering Results

Without further care, the generated preconditions are too strong. *E.g.*, some preconditions take the form:

$$strlen(s) \leq 0 \tag{8.1}$$

or

$$\mathit{offset\text{-}max}(s) < strlen(s). \tag{8.2}$$

Precondition 8.1 requires that string s has null length, while precondition 8.2 requires that string s is larger than the size of the underlying buffer. Such preconditions are generated in those cases where our techniques cannot generate an appropriate precondition, because the validity of the original check depends on axiomatized properties.

We filter out such results by removing from conjuncts those inequalities that bound $strlen$ and $\mathit{offset\text{-}min}$ from below or $\mathit{offset\text{-}max}$ from above, which cannot correspond to appropriate preconditions for the programs analyzed.

8.2 String Libraries

Strings are character buffers guarded by a sentinel null character. Before this sentinel, characters belong to the string; after it, characters are ignored. Strings are both pervasive, as the native way of communicating information in C programs, and potentially unsafe, as their safety depends on the presence of a null character somewhere in the buffer. Unintentionally erasing this character usually leads to a buffer overflow. Therefore, it is particularly useful to check the safety of string libraries implementations.

8.2.1 MINIX 3 Standard String Library

MINIX 3 is an open-source operating system designed to be highly reliable, flexible, and secure. To reach these goals, its code is intentionally small and simple. In particular, it implements a library for strings in an idiomatic and straightforward style, quite closely following the C standard. *E.g.*, here is the code for function `strcpy` that copies the content of a string `s2` to a buffer `ret`:

```
char *strcpy(char *ret, const char *s2) {
  char *s1 = ret;
  while (*s1++ = *s2++)
    /* EMPTY */ ;
  return ret;
}
```

Overall, the standard C library for strings specifies 22 functions, all of which are implemented in the MINIX 3 library. We applied the different annotation inference methods described in this thesis to check the safety of all of these functions at the exclusion of `strtok` whose safety relies on the proper sequencing of successive calls to `strtok`, a temporal property that cannot be catched by our inference methods.

By default, we run our tests with the abstract domain of octagons and region analysis set. A few tests required a different set of options:

- `strcat`, `strncat` and `memmove` require the use of polyhedrons to express preconditions or loop invariants on 3 variables;
- `memmove` requires that region analysis is not set, so that it is analyzed in a context where its pointer parameters may overlap.

Exact Integer Model In a first phase, we ran the benchmark with a mathematical model for integers, where it is assumed that integers neither overflow nor wrap around. Annotations inferred by abstract interpretations were not reproved by deductive verification. We compared 4 runs of the benchmark:

1. *no annotation inference* - No annotation at all is inferred.
2. *abstract interpretation* - Loop invariants are inferred by abstract interpretation, following algorithm ABSINTERP.
3. *quantifier elimination* - First, loop invariants are inferred by abstract interpretation, and then preconditions are inferred by quantifier elimination, following algorithm ABSELIM.
4. *weakest preconditions* - First, loop invariants are inferred by abstract interpretation, and then preconditions are inferred by weakest preconditions and quantifier elimination. We tried both algorithms ABSSTRONG and ABSWEAK, which gave the exact same results, due to the small size of source programs.

Results for these runs are summarized in Figures 8.1, 8.2, 8.3 and 8.4. The total number of verification conditions (VC) varies between runs as it depends on the annotations inferred. Provers are summarized by their initial letter: A for Alt-Ergo v0.8, S for Simplify v1.5.4, Y for Yices v1.0.16 and Z for Z3 v1.3. As expected, automatic provers succeed in proving more verification conditions when more annotations are inferred. When the most precise method is used, prover Z3 manages to prove all verification conditions, which is shown by filling the corresponding column on Figure 8.4.

Not surprisingly, the total time elapsed decreases with the number of annotations inferred, as shown in Figure 8.5. Indeed, in this case, the time spent for inferring annotations is far smaller than the time spent trying to prove unprovable verification conditions.

With the most precise inference method, a satisfying sufficient precondition to ensure the safety of each function is inferred. A precondition is satisfying when it is not too strong, so that usual patterns of usage for this function are allowed. We detail these preconditions in

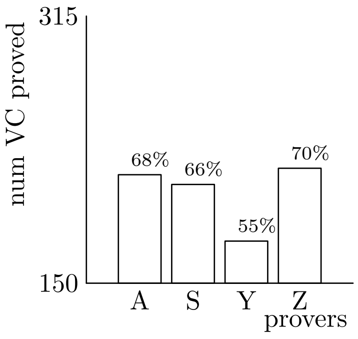

Figure 8.1: No annotation inference

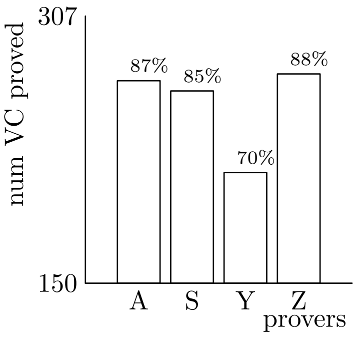

Figure 8.2: Abstract interpretation

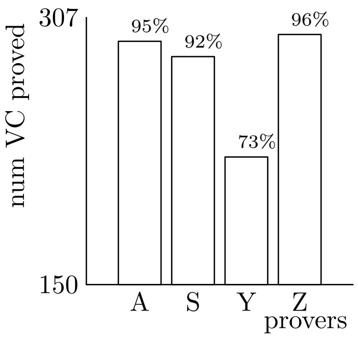

Figure 8.3: Quantifier elimination

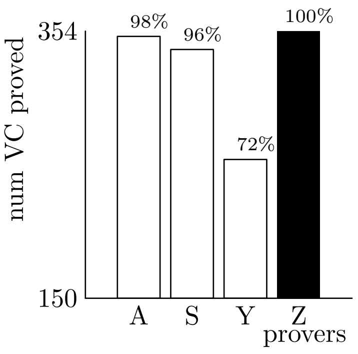

Figure 8.4: Weakest preconditions

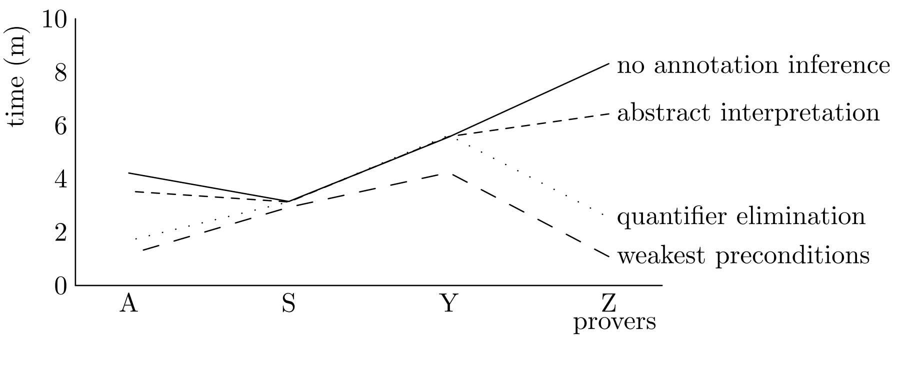

Figure 8.5: Total time elapsed

the following. For the sake of clarity, we rewrite $\mathit{offset\text{-}min}(x) \leq a \wedge b \leq \mathit{offset\text{-}max}(x)$ into $\mathit{valid\text{-}range}(x, a, b)$, and $\mathit{offset\text{-}min}(x) \leq c \leq \mathit{offset\text{-}max}(x)$ into $\mathit{valid\text{-}index}(x, c)$. Loop invariants are inferred too, but they usually mention intermediate variables introduced by the transformation from C to JESSIE, so they are not detailed here. They can be found in the log of the testing run. ACSL term $s + (..)$ denotes the set of pointers that can be obtained by pointer arithmetic from pointer s in the same memory block.

`memmove(void *s1, const void *s2, size_t n)`
- copy characters from buffer to possibly overlapping buffer -

$$\mathtt{n} \leq -1 \vee (\mathit{valid\text{-}range}(\mathtt{s1}, 0, \mathtt{n} - 1) \wedge \mathit{valid\text{-}range}(\mathtt{s2}, 0, \mathtt{n} - 1))$$

`strcpy(char *ret, const char *FRAMA_C_STRING s2)`
- copy characters from string to buffer -

$$\mathit{separated}(\mathtt{ret} + (..), \mathtt{s2} + (..)) \wedge \mathit{valid\text{-}range}(\mathtt{ret}, 0, \mathit{strlen}(\mathtt{s2}))$$

`strncpy(char *ret, const char *FRAMA_C_STRING s2, size_t n)`
- copy characters from string to buffer -

$$\mathit{separated}(\mathtt{ret} + (..), \mathtt{s2} + (..)) \wedge (\mathtt{n} \leq 0 \vee \mathit{valid\text{-}range}(\mathtt{ret}, 0, \mathtt{n} - 1))$$

`strcat(char *FRAMA_C_STRING ret, const char *FRAMA_C_STRING s2)`
- concatenate strings -

$$\mathit{separated}(\mathtt{ret} + (..), \mathtt{s2} + (..)) \wedge \mathit{strlen}(\mathtt{ret}) + \mathit{strlen}(\mathtt{s2}) \leq \mathit{offset\text{-}max}(\mathtt{ret})$$

At first glance, it may seem strange that only $\mathit{offset\text{-}max}(\mathtt{ret})$ is bounded. In fact, by joining this precondition inferred with the one implicit in `FRAMA_C_STRING` declaration specifiers, we get the expected precondition:

$$\mathit{separated}(\mathtt{ret} + (..), \mathtt{s2} + (..)) \wedge \mathit{valid\text{-}range}(\mathtt{ret}, 0, \mathit{strlen}(\mathtt{ret}) + \mathit{strlen}(\mathtt{s2}))$$

```
strncat(char *FRAMA_C_STRING ret,
        const char *FRAMA_C_STRING s2, size_t n)
```
- concatenate strings up to some bound -

$$\begin{array}{ll} & \mathit{separated}(\mathtt{ret} + (..), \mathtt{s2} + (..)) \\ \wedge & \left(\begin{array}{ll} & \mathtt{n} \leq 0 \\ \vee & \mathit{strlen}(\mathtt{ret}) + \mathit{strlen}(\mathtt{s2}) \leq \mathit{offset\text{-}max}(\mathtt{ret}) \\ & \wedge \mathit{strlen}(\mathtt{ret}) + \mathtt{n} \leq \mathit{offset\text{-}max}(\mathtt{ret}) \\ \vee & \mathit{strlen}(\mathtt{s2}) \leq \mathtt{n} - 2 \wedge \mathit{strlen}(\mathtt{ret}) + \mathtt{n} - 1 \leq \mathit{offset\text{-}max}(\mathtt{ret}) \end{array} \right) \end{array}$$

The precondition inferred is strictly stronger than the expected one:

$$\begin{array}{ll} & \mathit{separated}(\mathtt{ret} + (..), \mathtt{s2} + (..)) \\ \wedge & (\mathit{strlen}(\mathtt{ret}) + \mathit{strlen}(\mathtt{s2}) \leq \mathit{offset\text{-}max}(\mathtt{ret}) \\ & \vee \mathit{strlen}(\mathtt{ret}) + \mathtt{n} - 1 \leq \mathit{offset\text{-}max}(\mathtt{ret})) \end{array}$$

```
memcmp(const void *s1, const void *s2, size_t n)
```
- compare the content of buffers -

$$\mathtt{n} \leq 0 \vee (\mathit{valid\text{-}range}(\mathtt{s1}, 0, \mathtt{n} - 1) \wedge \mathit{valid\text{-}range}(\mathtt{s2}, 0, \mathtt{n} - 1))$$

```
strcmp(const char *FRAMA_C_STRING s1, const char *FRAMA_C_STRING s2)
```
- compare the content of strings -

$$\mathit{true}$$

The precondition inferred is indeed sufficient to prove safety of this function, as `FRAMA_C_STRING` declaration specifiers already assess that parameters are strings.

```
strcoll(const char *FRAMA_C_STRING s1, const char *FRAMA_C_STRING s2)
```
- compare the content of strings w.r.t. current locale -

$$\mathit{true}$$

```
strncmp(const char *FRAMA_C_STRING s1,
        const char *FRAMA_C_STRING s2, size_t n)
```
- compare the content of strings up to some bound -

$$\mathit{true}$$

```
strxfrm(char *s1, const char *FRAMA_C_STRING save, size_t n)
```
- transform a string taking into account the current locale -

$$\begin{array}{rl} & \mathit{separated}(\mathtt{s1} + (..), \mathtt{save} + (..)) \\ \wedge & \left(\begin{array}{rl} & \mathtt{n} \leq 0 \wedge \mathit{strlen}(\mathtt{save}) \leq 0 \\ \vee & \mathtt{n} \leq 0 \wedge \mathit{valid\text{-}range}(\mathtt{s1}, 0, \mathtt{n} - 2) \\ \vee & \mathtt{n} \leq 1 \wedge \mathit{valid\text{-}range}(\mathtt{s1}, 0, \mathtt{n} - 1) \\ \vee & \mathit{strlen}(\mathtt{save}) \leq 0 \wedge \mathit{valid\text{-}range}(\mathtt{s1}, 0, \mathtt{n} - 1) \\ \vee & \mathit{valid\text{-}range}(\mathtt{s1}, 0, \mathit{strlen}(\mathtt{save})) \end{array} \right) \end{array}$$

The precondition inferred is strictly weaker than the expected one:

$$\mathit{separated}(\mathtt{s1} + (..), \mathtt{save} + (..)) \wedge (\mathtt{n} \leq 0 \vee \mathit{valid\text{-}range}(\mathtt{s1}, 0, \mathtt{n} - 1))$$

```
memchr(const void *s, int c, size_t n)
```
- locate the presence of a character in a buffer -

$$\mathtt{n} \leq 0 \vee \mathit{valid\text{-}range}(\mathtt{s}, 0, \mathtt{n} - 1)$$

```
strcspn(const char *FRAMA_C_STRING string,
        const char *FRAMA_C_STRING notin)
```
- filter a prefix of a string based on exclusion -

$$\mathit{true}$$

```
strpbrk(const char *FRAMA_C_STRING string,
        const char *FRAMA_C_STRING brk)
```
- locate the presence of a string character in a string -

$$\mathit{true}$$

```
strrchr(const char *FRAMA_C_STRING s, int c)
```
- locate the last presence of a character in a string -

$$\mathit{true}$$

```
strspn(const char *FRAMA_C_STRING string,
       const char *FRAMA_C_STRING in)
```
- filter a prefix of a string based on inclusion -

$$\mathit{true}$$

```
strstr(const char *FRAMA_C_STRING s,
       const char *FRAMA_C_STRING wanted)
```
- find a substring in a string -

$$\mathit{true}$$

```
memset(void *s, int c, size_t n)
```
- initialize the content of a buffer -

$$\mathtt{n} \leq -1 \vee \mathit{valid\text{-}range}(\mathtt{s}, 0, \mathtt{n} - 1)$$

```
strerror(int errnum)
```
- return an error string -

$$\mathtt{errnum} \leq -1 \vee \mathtt{\_sys\_nerr} \leq \mathtt{errnum} \vee \mathit{valid\text{-}index}(\mathtt{\_sys\_errlist}, \mathtt{errnum})$$

```
strlen(const char *FRAMA_C_STRING org)
```
- compute the length of a string -

$$\mathit{true}$$

Bounded Integer Model In a second phase, we ran the benchmark with a bounded model for integers, where it is verified that integers do not overflow. Annotations inferred by abstract interpretations were not reproved by deductive verification. We used the abstract domain of polyhedrons. We ran all tests in the same setting, where loop invariants are first inferred by abstract interpretation, and then preconditions are inferred by weakest preconditions and quantifier elimination, with algorithm ABSSTRONG.

Analysis of function `memmove` fails due to a limitation of our implementation: various instrumentation integer variables introduced to serve as pointer and integer offsets (see

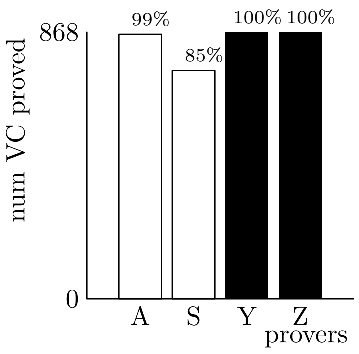

Figure 8.6: Bounded integer model

Section 8.1.2) should have logical integer type. This is not yet possible in our tool, instead we had to give them a large C integer type. In function `memmove`, this leads to the generation of a loop invariants with many useless inequalities involving large integers, which later on makes algorithm ABSSTRONG run out of space. Thus, we report our results on the remaining 20 functions.

Only one additional precondition inferred w.r.t. those inferred in the exact integer model is not obvious from the type of parameters. It requires that the integer used to initialize an array of characters in function `memset` fits indeed in a character:

$$-128 \leq \mathtt{c} \leq 127$$

Figure 8.6 summarizes the results of these runs. There are 868 verification conditions (VC), which is well above the 354 VC obtained with the exact integer model. This is expected, as all integer operations now require that the result fits in the corresponding C integral type. It comes as a surprise that prover Yices proves all VC with this integer model, like prover Z3, which is shown by filling the corresponding columns on Figure 8.6. After checking that it does not come from an inconsistency allowing to prove a false formula, we can conclude that our encoding of bounded integers fits better prover Yices than plain arithmetic on logical integer variables.

8.2.2 CERT Managed String Library

Source Code and Annotations CERT managed string library defines an abstract data type `string_m` to be used instead of plain strings. It is defined as a pointer to structure `string_mx` which encapsulates a few fields to store the string and maintain its consistency.

```
union str_union_t {
  char *cstr;
  wchar_t *wstr;
};

```

```
struct string_mx {
  size_t size;                  // allocated size of buffer
  size_t maxsize;               // maximum size of string
  unsigned char strtype;        // tag denoting the type of string
  union str_union_t charset;    // set of valid characters
  union str_union_t str;        // the real string
};

typedef struct string_mx *string_m;
```

String operations defined by the standard are implemented for these managed strings in a defensive programming style, including bound checking, so that misuses of the library should not lead to safety errors. Additionally, the library can ensure proper data sanitization by checking that all characters in a string (in field `str`) belong to a predefined set of safe characters (in field `charset`). Ignoring sanitization, the invariant maintained by library operations can be expressed as predicate *managed-string* in ACSL. Due to its complexity, this invariant was not inferred automatically but rather generated manually from the implementation of managed strings during the iterative proof process. In particular, some functions only ensure that the weakest *almost-managed-cstring* or *almost-managed-wstring* are verified.

```
/*@ predicate almost_managed_cstring(string_m s) =
  @   s→strtype ≡ STRTYPE_NTBS                 // a plain character string
  @   ∧ (valid_string(s→str.cstr)              // content is a valid string
  @       ∧ strlen(s→str.cstr) < s→size        // and string size is bounded
  @       ∧ \valid_range(s→str.cstr,           // and buffer size is bounded
  @                        0,s→size − 1)
  @     ∨ s→str.cstr ≡ NULL                    // or pointer is NULL
  @       ∧ s→size ≡ 0);                       // and size is null too
  @
  @ predicate managed_cstring(string_m s) =
  @   almost_managed_cstring(s)
  @   ∧ valid_string_or_null(                  // filter is a valid string
  @                      s→charset.cstr)
  @   ∧ (s→maxsize ≡ 0                         // maximum not set
  @      ∨ s→size ≤ s→maxsize);                // or size is below maximum
  @
  @ predicate almost_managed_wstring(string_m s) =
  @   s→strtype ≡ STRTYPE_WSTR                 // a wide character string
  @   ∧ (valid_wstring(s→str.wstr)             // content is a valid string
  @       ∧ wcslen(s→str.wstr) < s→size        // and string size is bounded
  @       ∧ \valid_range(s→str.wstr,           // and buffer size is bounded
  @                        0,s→size − 1)
  @     ∨ s→str.wstr ≡ NULL                    // or pointer is NULL
  @       ∧ s→size ≡ 0);                       // and size is null too
  @
  @ predicate managed_wstring(string_m s) =
  @   almost_managed_wstring(s)
  @   ∧ valid_wstring_or_null(                 // filter is a valid string
  @                      s→charset.wstr)
  @   ∧ (s→maxsize ≡ 0                         // maximum not set
  @      ∨ s→size ≤ s→maxsize);                // or size is below maximum
  @
  @ predicate managed_string(string_m s) =
  @   \valid(s) ∧ (managed_cstring(s) ∨ managed_wstring(s));
  @*/
```

Overall, we analyzed 49 functions operating over managed strings. From a total of 64 functions over 6,156 sloc, we cannot analyze 15 functions:

- 12 are input/output functions with a variadic number of arguments, which we do not support;
- the remaining 3 are the poorly designed `strtok_m`, whose comment warns that "this function is really messed up–need to redesign", and two related functions.

Thus, we manually annotated 49 functions with pre- and postconditions, as well as:

- 103 behaviors to further specify functions;
- 14 loop invariants for those functions with loops;
- 153 intermediate assertions to help provers;

for a total of 782 sloc for ACSL annotations. We did not generate any annotations for these functions, but we used the automatic separation of memory regions. Here is the implementation of the simplest function over managed strings, `strlen_m`, which computes the length of a managed string, together with annotations in ACSL that are needed to prove both `strlen_m` safety and the safety of functions that call `strlen_m`.

```
/*@ requires managed_string(s) ∨ s ≡ NULL;
  @ requires \valid(size) ∨ size ≡ NULL;
  @ assigns *size;
  @ behavior ok_cstr:
  @   assumes \valid(s) ∧ managed_cstring(s) ∧ \valid(size);
  @   assigns *size;
  @   ensures managed_string(s) ∧ \result ≡ 0;
  @   ensures \old(s→str.cstr ≡ NULL) ⟹ *size ≡ 0;
  @   ensures \old(s→str.cstr ≢ NULL) ⟹ *size ≡ strlen(s→str.cstr);
  @ behavior ok_wstr:
  @   assumes \valid(s) ∧ managed_wstring(s) ∧ \valid(size);
  @   assigns *size;
  @   ensures managed_string(s) ∧ \result ≡ 0;
  @   ensures \old(s→str.wstr ≡ NULL) ⟹ *size ≡ 0;
  @   ensures \old(s→str.wstr ≢ NULL) ⟹ *size ≡ wcslen(s→str.wstr);
  @ behavior bad:
  @   assumes s ≡ NULL ∨ size ≡ NULL;
  @   assigns *size;
  @   ensures \result ≢ 0;
  @*/
errno_t strlen_m(const string_m s, size_t *size) {
  register size_t n;

  if (!size) ERROR(EINVAL);
  *size = 0;

  //validate s
  if (!s){
    ERROR(EINVAL);
  }

  if (s→strtype == STRTYPE_WSTR) {
```

```
  wchar_t *lp;

  if (!s→str.wstr) { // Null string has length 0
    *size = 0;
    return 0;
  }
  n = 0;
  /*@ loop invariant
    @   lp ≡ s→str.wstr + n ∧ 0 ≤ n ≤ wcslen(s→str.wstr);
    @*/
  for (lp = s→str.wstr; n < s→size && *lp; lp++, n++)
  ;
  if (n >= s→size) ERROR(EINVAL);

  *size=n;
} else if (s→strtype == STRTYPE_NTBS) {
  char *lp;

  if (!s→str.cstr) { // Null string has length 0
    *size = 0;
    return 0;
  }
  n = 0;
  /*@ loop invariant
    @   lp ≡ s→str.cstr + n ∧ 0 ≤ n ≤ strlen(s→str.cstr);
    @*/
  for (lp = s→str.cstr; n < s→size && *lp; lp++, n++)
  ;
  if (n >= s→size) ERROR(EINVAL);

  *size=n;
} else{
  ERROR(EINVAL);
}
return 0;
} // end strlen_m
```

Source Code Modifications and Bugs In order to fully prove the safety of managed string functions, we had to patch the code for various reasons:

- We modeled allocation function `realloc` as `malloc` in our tool, thus missing the fact that `calloc` returns zero-initialized memory. We patched the source code to make for this incomplete modeling of `realloc`.

- Arrays of `wchar_t` were copied through calls to `memcpy`, thus requiring a cast to `char*`. Although our mixed memory model handles this situation, we did not try to come up with appropriate lemmas for automatic provers to discover that, with appropriate preconditions, a string over wide characters is copied this way. Thus we changed such calls to a new function `memcpy_wchar_t` which we specified.

- Various calls to deallocation function `free` made it much more difficult to automatically prove the validity of subsequent accesses to newly allocated memory in the same region, thus we ignored these calls.

Although these modifications change the semantics of the program analyzed, in particular the one that removes calls to `free`, proving the resulting program safe was still challenging. In particular, the first invariant we manually inferred for managed strings was far from being the actual one ensured by their implementation. Through an iterative process, we corrected both the invariant and function annotations and we proved implementation respected its annotations. In the process, we uncovered various bugs in the implementation that can lead to safety violations, plus a few functional bugs and typos that cannot lead to safety violations, that we do not report here. We report each occurence of a bug once, although it may appear more than once due to copy-paste.

1. In function `makestr`, tag `STRTYPE_WSTR` for wide characters strings is set instead of `STRTYPE_NTBS`, for the special case of an empty string. This can lead to safety violations after field `charset` is set to a string instead of a wide string.
2. In function `makestr`, where `len` is the rounding to the upper nearest multiple of 4 of `strlen(cstr), len` is used as size argument to `memcpy` instead of `strlen(cstr)`, which can lead to a safety violation.
3. In `str2wstr_m`, calls to `mbstowcs` are wrongly supposed to set `errno` on failure. This can lead to safety violations if `errno`'s value is zero, in which case `str2wstr_m` returns success code zero while the managed string has not been transformed into a wide managed string.
4. In `str2wstr_m`, variable `ncharset` is allocated with a size `len`, while it should have size `len+1` in order to accomodate the final null character in the string copied.
5. In function `csetcharset_m`, the call to `mbstowcs` is missing, which wrongly causes variable `t` to be the empty string.
6. In function `setmaxlen_m`, the nullity test on `maxlen` should be performed before so that the function aborts if the size of the managed string is already greater than the default maximum size. Although the functions analyzed do not directly use this maximum size, failure to respect managed strings invariant could lead to safety violations in client code.
7. In `cstrcat_m`, code does not guard against the possibility that the newly created string has a size greater than its maximum size.
8. In `cstrcmp_m`, allocation is performed with wrong size `strlen(cstr)` instead of `(strlen(cstr) + 1) * sizeof(wchar_t)`, which can cause a safety violation when function `mbstowcs` is called on the resulting buffer.
9. In `strcreate_m`, in the case where the call to `makestr` fails, the return statement is missing, which causes control-flow to continue with an invalid managed string.
10. In `wstrcreate_m`, in the case the input string is null, a wrong return of success is performed before fields `charset` and `maxsize` are set, which may cause safety violations in the client code.

11. In `cstrslice_m`, in the case a wide string `s1` is passed in argument, the nullity test for parameter `cstr` is missing, which can cause a safety violation when `strlen` is applied to `NULL`.

12. In `cstrright_m`, the test that `s1→size < ilen + s1_size` holds before copying the string is missing, so that a string too long for the actual capacity can be copied.

13. In `wstrright_m`, in the case of a plain string argument, test `!rv` is used instead of `rv != 0` to test for the failure of a call to `str2wstr_m`. As the result, execution can only proceed from this point with an incorrect managed string.

For each bug, we patched the library to ensure a correct behavior. Altogether, we reported 45 bugs to the authors of the library at CERT, which confirmed their status and included the corresponding patches for future releases. To assess the difficulty of finding these bugs, we also run the bug-finder Coverity on the complete set of 64 functions. It found 20 errors, 9 of which in the same 49 functions we analyzed: 2 errors are memory errors we found; the 7 remaining errors are related to uses of the deallocation function which we ignore. In the following, we describe the safety checking of the modified and patched library.

Results We ran the benchmark with a mathematical model for integers. Results for this run are summarized in Figures 8.7, 8.8, 8.9 and 8.10. Provers are summarized by their initial letter: A for Alt-Ergo v0.8, S for Simplify v1.5.4 and Z for Z3 v1.3. Yices v1.0.16 was not precise enough for this benchmark, which implied it unsuccessfully searched for a proof during the full 10 s time limit for too many VC. As a consequence, it ran too long to be either practical or competitive with other provers, so we did not try it on the complete benchmark.

Figure 8.7 shows that no single prover proves all VC. In fact, even their combination does not prove all VC. While 99.9 % of VC are proved automatically (18,080 VC for a total of 18,089), 9 VC are not proved automatically:

- In `makestr` and `wmakestr`, 2 VC for an intermediate assertion we added are not proved. They require reasoning about arithmetic with division (coming from bitwise arithmetic in the program), which provers do not handle well.

- In `strdup_m`, 4 VC for an intermediate assertion we added are not even provable with the given annotations. To be provable in theory, they would require specifying that managed strings respect a relation involving division between their actual and maximum sizes, which provers do not handle well, thus we did not even write these annotations.

- In `wstrright_m`, 3 VC for intermediate assertions we added are not proved. Our guess is that the number of variables involved in arithmetic relations and the size of the context leads provers in unsuccessful parts of the search tree.

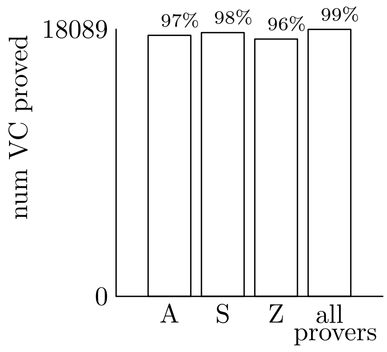

Figure 8.7: Proof results

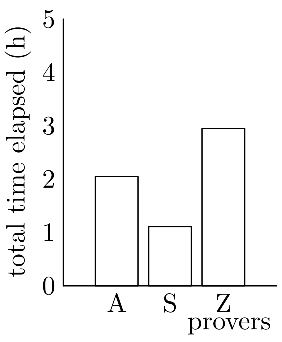

Figure 8.8: Time results

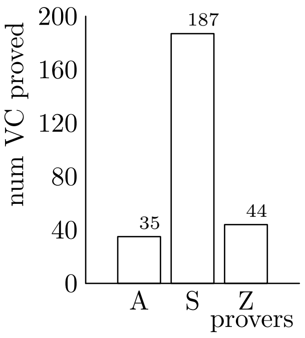

Figure 8.9: Provers strength

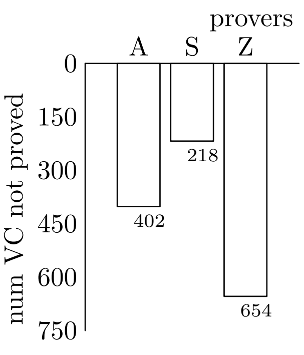

Figure 8.10: Provers weakness

For each one, we provided a manual proof that the corresponding goal holds.

Figure 8.8 presents the total running time for all three provers, which does not exceed 3 hours for each one.

Figure 8.9 and Figure 8.10 show why it is in general a good idea to use a combination of provers rather than a single prover. Figure 8.9 presents the number of VC that each prover is the only one to prove. Notice that using all 3 provers is indeed mandatory to reach an almost complete automatic proof. Figure 8.10 presents the number of VC that each prover does not prove although the VC is proved by some other prover.

8.2.3 Related Works

In his Master's thesis [164], Starostin fully verified a string library he implemented in C0. While our work focuses on automatic verification of safety only, his work is a manual verification inside Isabelle/HOL of the complete behavior of functions. Also, he codesigned the implementation and the proof, while we want to check the safety of existing libraries.

In his PhD thesis [143], Norrish presents a complete verification of the behavior of function `strcpy` as implemented by Kernighan and Ritchie [109]. It is in fact the same as the one still implemented in most systems, like the one in MINIX 3 presented in Section 8.2.1. His work is a manual verification inside HOL based on a deep embedding of C semantics, but he still manages to automate the proof of some properties involving arithmetic, most notably safety properties and separation properties. However, he notices the poor performance of automated techniques in HOL on the particular verification goals he generates.

8.3 Benchmarks of Vulnerabilities

We ran our tool on two benchmarks of real code vulnerabilities, the Verisec Suite and Zitser's benchmark. These benchmarks consist in snippets of open-source code containing buffer overflow vulnerabilities, together with their patched versions. Vulnerabilities are identified by their CVE number (ex: CVE-2004-0940). They are extracted from popular open-source server programs such as apache, samba and sendmail. Variations over each vulnerability are presented as a set of "bad" and "ok" snippets of code, usually in pairs, so that each "ok" version corresponds to the patch of a "bad" version. The two benchmarks differ in the number and difficulty of snippets:

- The Verisec Suite [112] targets 22 vulnerabilities in 12 programs, for a total of 144 "bad" and 140 "ok" snippets of code. Each snippet has a size between 16 and 233 loc, with an average of 69 loc, not counting include files.

- Zitser's benchmark [185] targets 14 vulnerabilities in 3 programs, with a "bad" and an "ok" snippet of code for each. Each snippet has a size between 218 and 777 loc, with an average of 506 loc, not counting include files.

As expected from these figures, Zitser's benchmark is more difficult to verify than the Verisec Suite, while the latter allows a finer analysis of results due to its large number of snippets with small variations. Although these snippets come equipped with a main function, we do not perform any global analysis on programs. Indeed, our target is to test the performance of our techniques to check safety both automatically and modularly, not needing the complete program. Thus, each function is analyzed independently of its calling context, in reverse topological order of the call-graph (*i.e.*, leaf functions first).

8.3.1 Verisec Suite

Source Code and Annotations To give an idea of the kind of programs analyzed, here is the code of an average size "bad" snippet `close-angle_ptr_two_tests_bad.c`:

```c
 1 int main (void)
 2 {
 3   char buffer[BASE_SZ+1];
 4   char input[BASE_SZ+70];
 5   char *buf;
 6   char *buflim;
 7   char *FRAMA_C_STRING in;
 8   char cur;
 9   int anglelev;
10   int skipping;
11
12   input[BASE_SZ+70-1] = EOS;
13
14   in = input;
15   buf = buffer;
16   buflim = &buffer[sizeof buffer - 1];
17   skipping = 0;
18
19   cur = *in;
20   while (cur != EOS)
21   {
22     if (buf >= buflim)
23       skipping = 1;
24     else
25       skipping = 0;
26
27     if (cur == '<')
28     {
29       if (!skipping)
30         anglelev = 1;
31     }
32     else
33       goto out;
34
35     if (!skipping)
36     {
37       *buf = cur;
38       buf++;
39     }
40
41 out:
42     in++;
43     cur = *in;
44   }
45
46   if (anglelev > 0)
47   {
48     *buf = '>';
49     buf++;
50   }
51
52   /* BAD */
53   *buf = EOS;
54   buf++;
55
56   return 0;
57 }
```

The */* BAD */* comment in the code identifies a possibly out-of-bound buffer access, that can indeed be triggered on some inputs. The corresponding "ok" snippet `close-angle_ptr_two_tests_ok.c` only differs from one line of code:

```c
16   buflim = &buffer[sizeof buffer - 2];
```

First, we added `FRAMA_C_STRING` user annotations in the code, like the annotation on line 7 in the code of `close-angle_ptr_two_tests_bad.c`. Overall, we added 389 `FRAMA_C_STRING` annotations denoting those parameters, returns and variables that should be strings.

For some cases where our inference technique does not generate precise enough annotations, we selectively added manual annotations in the code, in the form of assertions. *E.g.*, on the code of `close-angle_ptr_two_tests_ok.c`, we added the following assertion at line 36, to palliate the lack of control path sensitivity of our abstract interpretation pass:

```
36   //@ assert buf < buflim;
```

We did not complete this process until all VC are proved, for lack of time. During this process, we found 16 bugs in the "ok" snippets, where a buffer was either incorrectly not null-terminated, or a buffer possibly accessed beyond its bounds. We reported the corresponding patches to the authors of the suite, which recognized their suite was mostly designed to distinguish between "ok" and "bad" accesses at one particular point in each program, not necessarily granting safety of "ok" snippets.

Setup of Experiments We chose a mathematical model for integers. Annotations inferred by abstract interpretations were not reproved by deductive verification. Finally, we selected the following set of options:

- *provers Alt-Ergo v0.8, Simplify v1.5.4 and Z3 v1.3*: these are the provers that perform best on our verification conditions, denoted A, S and Z respectively. In addition, we ran Alt-Ergo with some heuristics for selecting hypotheses (denoted A_S), triggered by positioning option `-select 1` [49, 46], which focuses proof search on the goal and consequently allows to find easy proofs more quickly.

- *annotation inference* ABSSTRONG : it is the most precise annotation inference method that scales to these functions of up to 200 loc with many conditions and loops. The cheaper ABSINTERP and ABSELIM are not precise enough in many cases, and the more precise ABSWEAK does not scale.

- *abstract domain of octagons*: it is the most appropriate abstract domain for these tests, whose safety essentially depends on relations between pairs of variables. It is cheaper that the abstract domain of polyhedrons, and it leads to better widening results in many cases.

Results Due to current limitations in our tool, only 105 "ok" snippets out of 140 can be analyzed. On these snippets analyzed, Figure 8.11 shows that, while 99.7 % of VC are proved automatically (38,968 VC for a total of 39,080), 112 VC are not proved automatically. Overall, the snippets fall into the following cases:

- 35 snippets cannot be analyzed: 14 tests reach the allowed memory bound (0.25 M) or time bound (10 mn) during generation of annotations, 1 contains backward gotos, the remaining ones trigger limitations in our implementation;

- 78 snippets are completely proved;

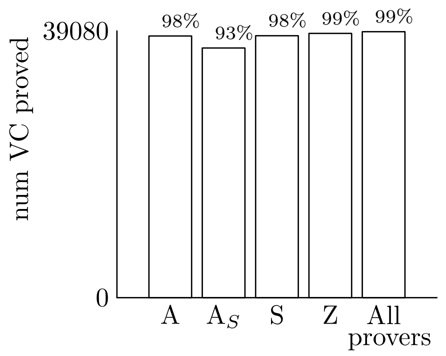

Figure 8.11: Proof results (105/140)

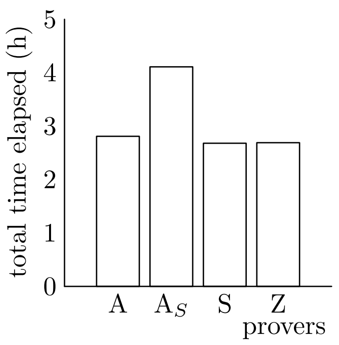

Figure 8.12: Time results (105/140)

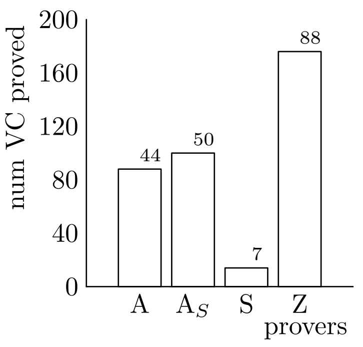

Figure 8.13: Provers strength (105/140)

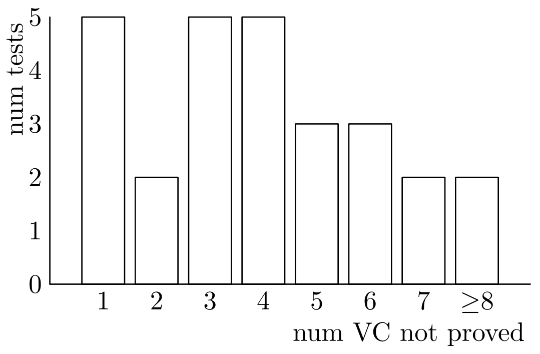

Figure 8.14: Unproved VC (27/140)

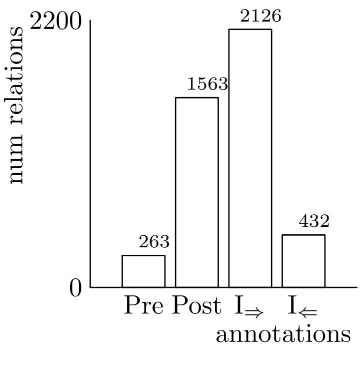

Figure 8.15: Verisec annotations

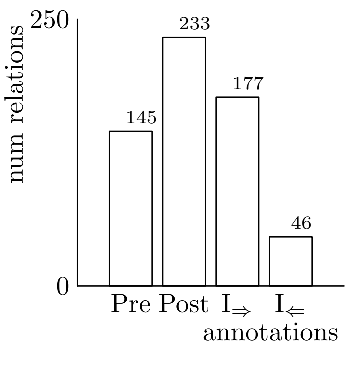

Figure 8.16: Zitser annotations

- 27 snippets are partly proved. Figure 8.14 shows that, among these tests, a majority have only a few unproved VC, which could be dealt with manually, either to add intermediate assertions to help provers, or by review.

Figure 8.12 presents the total running time for all four provers, which only exceeds 3 hours for Alt-Ergo with selection of hypotheses, because some goals then become unprovable, which causes the prover to reach the 10 s timeout while searching for an impossible proof.

Figure 8.13 shows why it is in general a good idea to use a combination of provers rather than a single prover. It presents the number of VC that each prover is the only one to prove. Notice that using all 4 provers, including Alt-Ergo with selection of hypotheses, is indeed mandatory to decrease the number of unproved VC.

Figure 8.15 presents the number of relations ((dis-)equalities and inequalities) in annotations inferred. Columns Pre and Post report the number of relations in preconditions and postconditions inferred, while columns $\mathrm{I}_{\Rightarrow}$ and $\mathrm{I}_{\Leftarrow}$ report the number of relations in loop invariants inferred respectively by forward abstract interpretation and backward ABSSTRONG algorithm. Formulas in $\mathrm{I}_{\Leftarrow}$ do not repeat those found in $\mathrm{I}_{\Rightarrow}$. The number of relations is larger in postconditions because they are built as a disjunction of cases for each return statement in the source program, and in loop invariants because they mention local variables, both from the source program and generated by our instrumentation.

8.3.2 Zitser's Benchmark

We tried generating annotations using forward abstract interpretation based on polyhedrons and algorithm ABSSTRONG. Our tool fails short of analyzing any of the 14 examples of the benchmark: 9 tests reach the allowed memory bound (0.5 M) or time bound (1 h) during generation of annotations, 1 contains backward gotos, the 4 remaining ones trigger limitations in our implementation. Although the verification is not complete, we can still report in Figure 8.16 the number of relations ((dis-)equalities and inequalities) in annotations inferred.

Despite these problems, there is no fundamental reason why our techniques should not apply to programs in Zitser's benchmark. What is certainly needed is a better quantifier elimination procedure, like the one presented by Monniaux [134], for which it would also make sense to generate a more compact formula using efficient weakest preconditions [119]. In particular, we manage to translate pointer casts in C programs into accesses to low-level accesses to memory. It is left to future work to analyze and prove these programs using Frama-C.

8.3.3 Related Works

Zitser's benchmark has had a great influence on the design of tools for safety checking of C programs. This was partly due to the integration of Zitser's programs in the SAMATE Reference Dataset used to compare tools for software assurance. By showing in their study [185] that none of the five modern static analysis tools tested was better than a random choice when discriminating between an unsafe program and its patched version, Zitser *et al.* have set a milestone for such tools. Since then, various tools have claimed to be able to improve on their results:

- Hackett *et al.* present a tool based on SAL lightweight annotations [84] that succeeds in discriminating most Zitser's test cases. However, since they use unsound static analysis techniques, they cannot make any claim about the safety of the patched programs.
- Chaki and Hissam present a tool [34] based on software model checking that improves on the confusion rate. They obtain that whenever their tool detects a potential buffer overflow in an unsafe program, it proves safety of the same buffer access in the patched version. Unfortunately, their tool also has lower detection and resolution rates than the two best tools presented in the study of Zitser *et al.*, namely PolySpace and Splint.

The Verisec Suite was developed with the same interface as Zitser's benchmark, to provide many simpler and diverse examples more amenable to verification. In particular, it makes it easier to bound the size of inputs for model checkers. Hart *et al.* manage to discriminate 49 tests out of 59 taken from the Verisec Suite by applying template-based model checking, where models of the program invariants are given by the programmer [86]. Like Hackett *et al.*, they only report their results on these identified potential overflows in the unsafe programs, not on all possible overflows. Contrary to our work, they perform a global analysis that takes profit from the simple crafted calling context of functions. In particular, they exploit the small bound on the size of buffers, which is expected in software model checking.

8.4 Chapter Summary

Experiments on real programs show that automatic and modular generation of annotations is effective. It allows us to prove the safety of most MINIX 3 string library functions and half "ok" snippets from the Verisec Suite of real vulnerabilities, while the remaining snippets analyzed has only a few VC not proved, which allows a manual review. Experiments also show that automatic separation of memory regions is effective. Without it, the safety of MINIX 3 string library functions and "ok" snippets from the Verisec Suite could not have been shown. It is also crucial in proving the safety of CERT managed string library functions. Finally, appropriate translation of unions and casts for deductive verification is successfully used both in proving the safety of CERT managed string library functions and in the failed attempts to prove safety of "ok" snippets from Zitser's benchmark.

A lesson learned from these experiments is that SMT automatic theorem provers are usually best used in combination, since each one may prove some VC no other prover handles. This extends to strategies in provers that simplify the goal at hand, as shown with prover Alt-Ergo and its option that prunes hypotheses. The particular set of provers selected for these experiments was chosen for their ability to handle axiomatized theories together with arithmetic, which is not the case of saturation-based provers [56]. It slighlty differs from the set of provers chosen in an experiment to show safety properties of SPARK Ada programs [100], which compared the results of Yices, CVC3, Simplify and Praxis's Simplifier, because Alt-Ergo, Z3, Yices and Simplify were found to be more effective than CVC3 on our VC. We did not experiment with Praxis's Simplifier.

As an aside, these experiments allowed us to discover a number of bugs in both CERT managed string library and Verisec Suite.

Conclusion

Je n'ai fait celle-ci plus longue que parce que je n'ai pas eu le loisir de la faire plus courte.

Blaise Pascal

Retrospective A few years before this thesis began, the results of two developments of specialized static analyzers based on abstract interpretation notably raised the state-of-practice in static safety checking of C programs. Both tools were developed for specific large programs written in a restricted subset of C (see Section 1.3.2).

2003 - Cousot *et al.* [21] report on static safety checking of 100+ kloc of command-control software written in C for the new airplane A380 by Airbus. This is the first time abstract interpretation is shown to scale to such large programs for checking non-trivial properties. Their tool, ASTRÉE, completely proves that the program analyzed is free from a set of runtime errors. This level of precision is reached thanks to the restricted subset of C in which command-control software is written, which is almost alias-free, and to the special-purpose abstract domains developed for this kind of programs.

2004 - Venet and Brat [174] report on array bound checking of 280+ kloc of NASA software written in C, with a level of precision of 80%. Their tool, C GLOBAL SURVEYOR, is specialized for NASA programs in the MPF family, which are written in an object-oriented style. Although not proving 100% of checks, their work sets a new record for scalability of abstract interpretation for checking non-trivial properties.

In order to assess the efficiency of the many tools available for static safety checking of arbitrary C programs, most of them being academic tools, Zitser *et al.* conducted an evaluation with unambiguous conclusions, that initiated the search for better techniques and tools (see Section 8.3.3).

2004 - Zitser *et al.* report on benchmarking various academic and industrial tools to check safety of C programs with real vulnerabilities extracted from well-known server programs [185]. Their main conclusion is that no tool discriminates between the unsafe and patched versions of the same programs.

2005 - NIST initiates the SAMATE project (see Section 1.1.4), most notably to support the improvement of tools for static safety checking of C programs. It creates the SAMATE Reference Dataset (SRD), a list of test cases for software assurance, most of which relate to safety vulnerabilities in C programs. Zitser benchmark belongs to the SRD.

Meanwhile, a few research projects succeeded in proving memory safety of unrestricted C programs with aliasing, strings and complex control-flow. These results were obtained on much smaller programs than those targeted with ASTRÉE and C GLOBAL SURVEYOR (see Section 8.2.3).

2003 - Dor *et al.* [62] apply integer constraint solving to check memory safety of C programs with strings. Their tool, CSSV, handles small string-manipulating functions from Airbus, for a total of 400 loc. This is the first time a sound tool taking aliasing into account manages to treat such delicate programs.

2005 - Beyer *et al.* [20] apply their model checking tool BLAST to check memory safety of C programs, expressed as checks in source code inserted by safe compiler CCured. They manage to discharge roughly half the runtime checks inserted in small C programs of a few hundred lines.

Contribution of This Thesis This thesis targets static safety checking of arbitrary C programs by deductive verification. While the current focus on industry is on static analysis for verification, the motivation behind our choice of deductive verification as technological core is two-fold:

- *modularity* - Although static analysis techniques and tools work on complete programs, the next generation of tools should be modular, *i.e.*, they should be able to check the safety of individual functions and modules (group of functions). This is driven both by the fast-pace growth of programs, that cannot be matched by similar gains in tool scalability, and the demand for early safety checking by developers themselves, before the complete program is built.

- *precision* - The level of complexity of arbitrary C programs can only be matched by completely modeling the behavior of an execution inside the analyzer, which is what deductive verification provides.

The need for modularity and precision is at the root of many existing tools for verification, which are also based on deductive verification: HAVOC, VCC, EAU CLAIRE, KEY-C, PERFECT C. This thesis presents techniques for static safety checking of industrial C programs by deductive verification. More precisely, we propose an answer to each of the three main problems one must face when trying to apply deductive verification to industrial C programs [93]:

- *annotation generation* - Deductive verification without the ability to automatically generate the necessary logical annotations may only be undertaken for very few projects, due to the cost of manually adding annotations.

  In Chapter 5, we present a technique to generate logical annotations based on abstract interpretation and weakest preconditions. In particular, it generates precise sufficient function preconditions, which has not been done before.

- *modular memory separation* - Fine memory separation is the only way to generate verification conditions that can be verified by automatic provers. This is especially true in a context where annotations are automatically generated.

  In Chapter 6, we present an alias control technique based on Talpin's alias analysis, a context sensitive variant of Steensgaard's type-based alias analysis. It is the first instance of an alias analysis that generates necessary function preconditions of correctness, thus explicitly dedicated to deductive verification. This technique allows one to express separation properties clearly in verification conditions, in a way that is optimal for automatic provers.

- *support for unions and casts* - Industrial C programs do use the low-level memory management capabilities of the C language, most notably unions and casts of pointers. Failure to support these features in past tools has been recognized as the major barrier to adoption of these tools in an industrial context.

  In Chapter 7, we present a mixed typed and byte-level memory model that allows one to handle unions and casts in deductive verification, while keeping as much as possible the benefits of the typed model. It relies on the modular memory separation technique mentioned above.

These techniques have been implemented in Frama-C [73], an open-source platform for modular analysis of C programs, and the Why Platform [69], an open-source platform for deductive verification of programs.

We manage to check the safety of existing C string libraries and to discriminate between unsafe and patched versions of the same programs in benchmarks of vulnerabilities.

Finally, safety is only the first "easy" step in dependability of programs. The techniques we present and deductive verification in general allow one to prove functional properties as well. It remains to be seen how to follow this path in practice.

Research Perspectives The techniques we developed in this thesis still suffer from various limitations:

- The annotation inference technique described in Chapter 5 depends on quantifier elimination for rational linear arithmetic, for which all known algorithms have worst-case doubly exponential complexity. Although it is possible to reduce this complexity in practice by restricting the use of quantifier elimination to simplified formulas (see Section 5.3.3), more efficient quantifier elimination methods are needed to scale to larger programs. In this respect, recent results show great promise [134].

- The region inference technique described in Chapter 6 is incomplete (see Section 6.3.5), which means that a program containing a function such as `swap` cannot be analyzed with region inference. A solution could be to manually express constraints between regions in annotations in such rare cases, much as what is done in Cyclone [104]. The status of these regions in ACSL and operations on those remains to be investigated.

- The separation preconditions generated by both region inference in Chapter 6 and interfacing of a low-level memory model with a typed memory model in Chapter 7 result in hard verification conditions for automatic provers. Developing a decision procedure for sets inside SMT-solvers would certainly ease the proof of such verification conditions.

This thesis would not be complete without an account of the current focus in research on static safety checking of C programs. Indeed, the state-of-the-art changed considerably in the past three years, this thesis's time.

Combination of techniques is a very active line of research. Abstract interpretation is used everywhere to compute invariants and discharge simpler checks. Symbolic model checking and deductive verification compete to provide an automatic proof that remaining checks hold.

Heap analysis is probably the most researched area in static safety checking. It consists in inferring and checking invariants on the heap structure. Separation logic is the key technology enabling these successes. As more decision procedures are built for separation logic, it might be profitable to generate the separation preconditions generated by our modular inference of region in this logic rather than as conjunctions of *separated* predicates in first-order logic, as we do it currently.

Finally, the emergence of certified compilers makes it all the more useful to have source programs checked safe, knowing that a certified compiler cannot introduce any error afterwards.

Industrial Trends This thesis was supported by a CIFRE fellowship from Orange Labs, as well as ANR project CAT (C Analysis Toolbox) involving industrial users such as Airbus, Dassault Aviation and Orange Labs. This project, which lead to the development of Frama-C, raised interest in the industry. Support and development for Frama-C will continue inside ANR project U3CAT, with even more industrial partners.

An implementation of the techniques described in this thesis is available in the Frama-C Tertium release, issued in October 2008. Early experiments at Airbus and Dassault Aviation have confirmed the interest for the techniques of annotation and region inference developed in this thesis.

At least one lightweight annotation language for C programs is now widely used in industry at Microsoft: SAL properties expressed as declaration specifiers allowed to discover +3000 buffer overflows in the code of Microsoft Vista Windows release [84], which prompted their adoption in the normal development process and lead to far fewer security problems with recent products.

Spec# for C is another annotations language for C developed in Microsoft. Using SAL and Spec# annotations, the Hyper-V project between University of Saarbrücken and Microsoft targets safety checking and concurrency properties verification of the 60 kloc Microsoft Hyper-V virtualization product shipped as a component of Windows Server 2008 [43].

Meanwhile, tools based on abstract interpretation are drawing more and more attention from the industry, due to the level of automation (w.r.t. deductive verification) and the guarantees (w.r.t. bug finders) they provide. This is currently leading to their industrialization, as exemplified by Astrée [21], Clousot [66], CodeHawk [3], Frama-C [73], F-Soft [99], Penjili, Sparrow [183].

A Few Final Words This thesis began in 2006, 10 years after Aleph One happily broadcasted on the Internet the details of how to smash the stack for fun and profit [144], based on vulnerabilities in C programs. It ends in 2008, 20 years after the Morris worm, the first worldwide Internet attack partly based on a buffer overflow in a C program. Since then, attacks based on safety vulnerabilities in C programs have cost businesses billions of dollars, while their anonymous authors remained hidden behind the names of the worms they created: Code Red, SQL Slammer, Blaster, Sasser, Witty worm, Zotob, *etc*.

Despite much efforts from researchers, companies and governments, safety of C programs still represents a challenge. In 2007, 13 of the SANS Institute Top 20 Software Vulnerabilities are still related to buffer overflows or memory corruption in C programs. In 2008, an analysis of the first 60 Ubuntu Security Notices, from the leading Linux distribution Ubuntu, shows that 45% of vulnerabilities stem from buffer overflows. Current software practice, based on compliance with a qualified process and validation by testing will not be sufficient to assess the dependability of software systems, as they keep growing in size: currently, C programs in phones are as big as 100k loc, those in planes contain millions loc and those in PCs tens of millions loc.

It is not possible to get rid of this safety problem, as some pretend to do, by saying that it has been solved in many modern programming languages. Despite all its safety issues, C still appears to be the first most demanded general-purpose programming language from programmers in job offers on the Web in 2007 [64]. Moreover, many aerospace, avionics and defense industries currently switch from the safer language Ada to C to benefit from better compiler and tool availability and better education and training support. Broad application of C to program safety critical systems requires the support from new techniques and tools.

Finally, I would like to pay a tribute to a few researchers which, 30 years ago, in 1978, laid down the basis the present work. At POPL conference, in Tucson, Arizona, three seminal works by Reynolds, Cousot and Halbwachs, German defined respectively *Syntactic Control of Interference*, *Automatic Discovery of Linear Restraints Among Variables of a Program* and *Automating Proofs of the Absence of Common Runtime Errors*. Indeed, my thesis is an attempt at building an effective tool for static safety checking of programs, based on deductive verification and automatic provers, much as in the Stanford Pascal Verifier of German *et al.*. The first key ingredient is a precondition inference technique that strongly relies on abstract interpretation to generate linear invariants, which was pioneered by the work of Cousot and Halbwachs. The second key ingredient is an aliasing control technique that originates in the work of Reynolds.

Ironically, while their work targeted array bound checking as a means to “simply” save runtime checks in Pascal programs, mine and other current work on similar buffer overflow issues more tragically attempt to prevent attacks on computer systems, due to the unsafe nature of C programs. It is all the more ironic to realize that it is during the very same year 1978 that Kernighan and Ritchie gave an initial definition of the C programming language.

Appendix A

Résumé en Français

A.1 Introduction

Sûreté de fonctionnement des programmes C Par rapport aux langages plus sûrs développés depuis, le langage C créé à la fin des années 70 offre de nombreuses facilités pour manipuler efficacement la mémoire de l'ordinateur et pour s'interfacer avec des composants matériels. Ces facilités en font le langage préféré pour la programmation système, au détriment de la sécurité dans sa composante sûreté de fonctionnement. En effet, les mêmes facilités de programmation bas-niveau rendent difficile la protection des programmes contre des utilisateurs malveillants. Elles rendent même possible la prise de contrôle à distance d'un ordinateur par un attaquant. C'est ce qui se passe avec la faille bien connue de dépassement de capacité ("buffer overflow") exploitée dans des attaques restées célèbres affectant des millions d'ordinateurs personnels à travers Internet, telles que Code Red (2001), Blaster (2003) ou Sasser (2004). Le coût humain, économique et sociétal de cette vulnérabilité des systèmes informatiques pousse les chercheurs, les entreprises et les gouvernements à accroître l'effort global en vue de garantir la sûreté de fonctionnement de ces systèmes, à travers notamment une meilleure identification des problèmes (base de connaissance CWE, base de problèmes CVE) et un partage des meilleures pratiques et outils (conférences scientifiques, projet SAMATE).

Problèmes de sûreté du langage C Malgré une standardisation précoce en 1978, et différentes actualisations depuis, de nombreux dialectes légèrement différents du C standardisé coexistent aujourd'hui (par ex. : Visual C, GNU C), ce qui complique l'analyse des programmes C. De plus, le standard laisse un certain nombre de choix importants au compilateur, l'outil qui transforme le texte d'un programme C en un programme exécutable par une machine (par ex. : l'ordre d'évaluation des arguments d'une fonction). Un même programme compilé par deux compilateurs différents peut donc se comporter de deux façons différentes. Enfin, il n'existe pas de sémantique formelle standard du langage C, donnée dans un langage logique non ambigu, mais seulement un texte en langue naturelle (l'anglais) sujet à interprétation. Norrish dans sa thèse de doctorat [143] et Leroy *et al.* dans la construction d'un compilateur certifié [23, 22, 125] définissent une sémantique formelle du langage C pouvant être mécanisée, c.à.d. exploitée dans des programmes d'analyse.

Par rapport aux langages assembleurs, le langage C définit une abstraction des données sous forme de types et une abstraction du contrôle sous forme de graphe d'appel, mais il ne garantit pas le respect de la première, ce qui permet en pratique de s'affranchir complètement des deux abstractions. Afin de protéger un programme contre les attaques les plus graves qui violent l'abstraction du contrôle, par ex. les prises de contrôle à distance, il suffit de prouver la sûreté des accès mémoire. C'est pourquoi nous avons choisi de nous concentrer sur la sûreté des accès mémoire dans cette thèse.

Différentes techniques permettent de limiter les conséquences des corruptions mémoire : empêcher l'exécution de code sur la pile, adopter une programmation défensive pour les fonctions de bibliothèque standard (Libsafe [9]), distribuer de façon aléatoire les adresses des programmes (PaX), détecter les réécritures des adresses de retour de fonction (StackGuard, StackShield), encrypter la valeur des pointeurs (PointGuard [53]), instrumenter le code à la compilation pour vérifier la validité des accès mémoire (Safe C [5], CCured [138]). Ces techniques partagent les mêmes limitations : elles doivent être mises en place par chaque utilisateur d'un programme ; en dehors de l'instrumentation à la compilation, elles sont incomplètes ; elles n'empêchent pas l'apparition d'erreurs mais seulement leur exploitation. Pourtant, ce sont ces techniques qui sont les plus efficaces en pratique aujourd'hui.

D'autres initiatives ont pour objectif d'améliorer le langage C afin de rendre les accès mémoire plus sûrs. Les propositions les plus intrusives sont à la fois les plus efficaces et les plus coûteuses à mettre en place. En ordre croissant d'efficacité et de coût, ces propositions sont : l'utilisation de bibliothèques standard sûres, notamment pour les chaînes de caractères (The Better String Library, SafeStr, Vstr, Erwin, CERT managed string library) ; la restriction à un sous-ensemble du C en excluant certaines facilités problématiques telles que les unions et les casts (MISRA-C [133], C0 [116]) ; la définition d'un dialecte plus sûr inspiré du C (Cyclone [104], BitC [160], D [16]) ; l'ajout d'annotations logiques pour spécifier les comportements des programmes (Deputy [184], SAL [84]). Les techniques d'annotation notamment ont permis récemment de corriger un grand nombre de failles liées aux accès mémoire dans des programmes industriels [84].

Afin de prouver statiquement (avant exécution) la sûreté de fonctionnement des programmes C existants, d'autres techniques et outils ont été développés. Il existe trois grandes familles de techniques, suivant qu'elles résolvent le problème d'exploration d'un nombre infini d'états du programme par énumération, abstraction ou déduction.

- *énumération* - Ces techniques explorent systématiquement tous les états en se limitant à une taille de problème finie. C'est le cas des tests [107], la technique de vérification la plus simple et la plus utilisée, où la notion de couverture remplace l'exhaustivité. C'est le cas aussi dans la simulation, la vérification de modèle [42] (VeriSoft [75]), la vérification de modèle symbolique (Cadence Incisive, CBMC [40], F-Soft [99]).

- *abstraction* - Ces techniques construisent une abstraction finie du programme adaptée au problème à résoudre. D'une part, la définition d'abstraction sûres par interprétation abstraite [51] permet de développer des vérificateurs pouvant être utilisés comme des détecteurs de bogues (Astrée [21], C Global Surveyor [174], CodeHawk [3], Clousot [66], Penjili, The Mathworks PolySpace, Sparrow [183]). D'autre part, l'utilisation d'abstraction non sûres par analyse statique permet de développer des détecteurs de bogues plus efficaces (Microsoft Prefix/Prefast, Fortify SCA, Grammatech CodeSonar, Klocwork Insight, Coverity Prevent).

- *déduction* - Ces techniques reposent sur la génération d'obligations de preuve (logique de Hoare [89], calcul de Dijkstra [59]), des formules logiques dont la validité garantit la correction du programme, validité qui peut être prouvée grâce à des prouveurs de théorèmes. Suivant l'outil, différentes logiques sont utilisées : logique classique du premier ordre (Frama-C [73], VCC [43]), logique dynamique du premier ordre (Key-C [137]), logique de séparation (SLAyer [182]). Certains prouveurs sont interactifs (Coq [19], HOL [142] , Isabelle/HOL [141], PVS [145]) et d'autres automatiques (Alt-Ergo [46], CVC3 [12], Simplify [57], Yices [63], Z3 [135]).

Objectifs de la thèse et résultats obtenus Seule la découverte de techniques de vérification modulaires et automatiques permettra de s'adapter à l'accroissement continu de la taille des systèmes logiciels. Cette thèse propose de s'appuyer sur les techniques de vérification déductive pour y parvenir. Plus précisément, nous répondons aux trois problèmes posés par l'application de la vérification déductive aux programmes C industriels [93], de manière modulaire et automatique : la génération d'annotations, la séparation de la mémoire en régions disjointes, le traitement des unions et des casts.

A.2 Opérations entières et accès mémoire

A.2.1 Définition d'un langage intermédiaire

Il existe de nombreux langages intermédiaires pour faciliter la compilation (GENERIC [163], SIMPLE [87], MSIL [39, 131]) et l'analyse (CIL [139], Newspeak [96], Cminor [22], C0 [116]) de programmes C. Nous définissons un langage intermédiaire JESSIE qui présente deux nouveautés : il réussit à conserver des types de données structurés tout en les simplifiant considérablement, et il combine des traits opérationnels hérités du C avec des traits logiques hérités des langages d'annotations de Caduceus [68] et Krakatoa [129], eux-mêmes hérités de JML [115] (JAVA Modeling Language).

Les types de données de JESSIE se répartissent en types de base, essentiellement les rangées entières similaires aux types entiers d'ADA, et les types structurés agrégés en tableaux et ne pouvant être accédés que par pointeur. Cela réduit la syntaxe des accès mémoire à une succession d'opérations arithmétiques sur des pointeurs et d'accès à des champs de structure à partir d'une variable, par ex. `(x⊕i).m`, avec `x` une variable de type pointeur, `i` un terme de type entier et `m` un champ de structure. La syntaxe abstraite du langage comporte ainsi des types, des termes, des instructions, des structures de contrôle et des entités globales (variables et fonctions). Des règles de typage permettent de définir précisément un certain nombre de contraintes à respecter pour écrire des programmes JESSIE valides. Le modèle mémoire du langage JESSIE comporte trois types de données : les variables globales, les variables locales et la mémoire, qui regroupe les données accédées par pointeur. Afin de concilier l'expressivité du modèle mémoire réaliste, qui considère toute la mémoire comme un grand tableau d'octets, et la précision du modèle mémoire par blocs, qui découpe la mémoire en blocs disjoints incomparables, nous définissons un nouveau modèle mémoire intermédiaire qui associe une adresse à chaque bloc. Nous définissons l'exécution d'un programme JESSIE à l'aide d'une sémantique naturelle basée sur ce modèle mémoire, qui différencie clairement les exécutions correctes, les exécutions en erreur et les exécutions qui ne terminent pas.

La traduction d'un programme C en un programme JESSIE utilise comme étape intermédiaire le langage CIL, une forme de syntaxe abstraite encore très proche du C. La traduction de C à CIL préexistante impose un certain nombre de choix d'implémentation, comme l'ordre d'évaluation des arguments. Nous présentons des règles précise pour la traduction de CIL à JESSIE, de telle sorte qu'un programme C traduit en JESSIE et interprété grâce à la sémantique de JESSIE ait le même comportement que s'il était exécuté après compilation sur une machine avec mémoire infinie. Cette traduction, qui exclut pour l'instant les flottants et les pointeurs de fonction, permet de prouver la sûreté de fonctionnement d'un programme C en analysant sa version équivalente en JESSIE. L'expression de propriétés des programmes JESSIE est facilitée par l'inclusion d'entités logiques dans le langage : types logiques, termes logiques, propositions, assertions, invariants de boucles, fonctions logiques, prédicats, contrats de fonction. Ce dernier se décompose en précondition, postcondition et condition d'effets. Les annotations logiques du programme C exprimées en ACSL [14] (ANSI C Specification Language), sont également traduites vers des entités logiques en JESSIE.

Après analyse du programme JESSIE, sa traduction en un programme WHY permet d'appliquer les techniques habituelles de vérification déductive [67, 68] : des obligations de preuve sont générées et leur preuve garantit la sûreté de fonctionnement du programme JESSIE équivalent, et donc du programme C original.

A.2.2 Preuve de la sûreté des opérations entières

Le type de données le plus essentiel en programmation est le type des entiers. La plupart des techniques d'analyse de programme ont d'ailleurs été développées initialement pour des programmes manipulant seulement des entiers. La sûreté de fonctionnement de ces programmes en JESSIE est garantie par l'absence de débordements entiers et de division (ou modulo) par zéro, ce qui peut s'exprimer par des assertions logiques dans les programmes. Prouver ces assertions par interprétation abstraite ou vérification déductive garantit donc la sûreté de fonctionnement du programme.

La théorie de l'interprétation abstraite repose sur la définition de valeurs abstraites qui surapproximent les valeurs concrètes manipulées par le programme. Les opérations concrètes du programme sont elles-aussi surapproximées par des opérations abstraites sur ces valeurs abstraites. La précision et l'efficacité de la construction d'un modèle abstrait du programme dépendent du choix du domaine abstrait qui représente les valeurs abstraites : les domaines non relationnels décrivent des valeurs individuelles (signe, intervalles, congruences) ; les domaines relationnels décrivent les relations entre au moins deux variables (DBM [60], octogones [132], égalités linéaires [108], polyèdres [52, 44]) ; les domaines produits combinent les résultats de plusieurs domaines (produit cartésien [50], produit réduit [50], complétion disjonctive [50], produit logique [80]). Nous décrivons l'interprétation abstraite intraprocédurale des programmes JESSIE sous forme de règles d'inférence.

La vérification déductive repose sur les règles de la logique de Hoare, qui donnent la possibilité de raisonner sur les opérations d'un programme complètement annoté. Les calculs par plus-faibles-préconditions ou plus-fortes-postconditions de Dijkstra réduisent les besoins d'annotations aux seuls invariants de boucles et contrats de fonction. Nous décrivons un calcul de plus-faibles-préconditions des programmes JESSIE sous forme de définition de fonction par récurrence structurelle.

A.2.3 Preuve de la sûreté des accès mémoire

En vue de prouver la validité des accès mémoire, nous définissons un encodage local du mémoire mémoire de JESSIE, qui réduit le nombre de variables de trois à deux dans les assertions générées, par rapport à l'encodage naturel. En effet, comme pour l'absence d'erreurs

```
define paths-may-overlap:
  input chemins $\pi_1$ et $\pi_2$
  output si oui ou non $\pi_1$ et $\pi_2$ représentent des zones mémoires non disjointes
  match ($\pi_1$,$\pi_2$) with
  (1)      | (x,x) $\rightarrow$ return true
  (2)      | ((_$\oplus$_)._,(_$\oplus$_)._) $\rightarrow$ return true
  (3)      | (x,_) | (_,x) $\rightarrow$ return false
```

Figure A.1: Superposition de chemins

entières, nous pouvons exprimer la validité des accès mémoire par des assertions logiques dans les programmes, qui prennent le plus souvent la forme d'inégalités entre deux variables, par ex. $\mathtt{i} \leq \textit{offset-max}(\mathtt{x})$ pour exprimer l'absence de débordement par valeur supérieure lors de l'accès mémoire `(x⊕i).m`.

Nous introduisons une nouvelle forme de variables abstraites pour représenter des zones mémoire, qui correspondent simplement à des chemins syntaxiques du programme C. Ces variables abstraites syntaxiques se différencient des variables abstraites résumé [21, 174] et des variables abstraites de chemin [58, 37, 38, 113] par la possibilité pour deux variables abstraites syntaxiques de représenter des zones mémoire non disjointes. Afin d'exprimer par inégalités des propriétés arbitrairement complexes sur les programmes, nous introduisons également des variables abstraites d'application, qui correspondent à l'application d'une fonction logique à des variables du programme, par ex. $\textit{strlen}(\mathtt{x})$ pour la longueur d'une chaîne de caractères `x`. L'algorithme *paths-may-overlap* présenté à la Figure A.1 définit de façon conservative dans quels cas deux chemins syntaxiques sont garantis de ne pas se superposer, c.à.d. qu'il représentent des zones mémoires disjointes. Avec cette version naïve de l'algorithme, toutes les zones mémoire sont considérées non disjointes.

Nous utilisons l'algorithme *paths-may-overlap* pour adapter les analyses de programmes JESSIE sans pointeurs présentée à la Section A.2.2 aux programmes JESSIE avec pointeurs. Il s'agit principalement d'ignorer les valeurs de variables abstraites pouvant se superposer à la variable abstraite réaffectée lors du traitement d'une affectation. Dans le cas de l'interprétation abstraite, cela revient à modifier l'affectation de variable abstraite ; dans le cas de la vérification déductive, il faut seulement redéfinir la règle de substitution de variable abstraite.

A.3 Inférence, séparation, unions et casts

A.3.1 Programmes typés sans partage mémoire

Nous commençons par nous restreindre aux programmes JESSIE fortement typés et sans partage mémoire. Dans ces programmes, une zone mémoire n'a qu'un seul type possible et n'est accédée que par un seul chemin. Ces restrictions permettent de donner une définition beaucoup plus précise de l'algorithme *paths-may-overlap*, qu'il est possible d'exploiter dans les analyses de programmes JESSIE par interprétation abstraite ou vérification déductive.

Pour ces programmes plus simples, il existe des techniques d'inférence d'annotations logiques, de façon à éviter l'ajout d'annotations manuelles par l'utilisateur pour les invariants de boucles et les contrats de fonctions. Différentes techniques sont applicables :

- *interprétation abstraite* - Il est naturel de générer des invariants de boucles et des postconditions de fonction par interprétation abstraite.
- *déboguage abstrait* - Cette technique introduite par Bourdoncle [26] permet de générer des préconditions de fonction nécessaires par interprétation abstraite arrière.
- *diagnostique d'alarme* - Cette variante du déboguage abstrait inventée par Rival [153] permet de générer des préconditions de fonction suffisantes.
- *plus-faibles-préconditions* - Il est naturel de générer des préconditions de fonctions par plus-faibles-préconditions.
- *induction-itération* - Cette technique introduite par Suzuki and Ishihata [166, 181] permet de générer des invariants de boucle par plus-faibles-préconditions et élimination de quantificateurs.

Aucune de ces techniques n'est pleinement satisfaisante, pas plus que leur accumulation. Les techniques à base d'interprétation abstraite sont peu efficaces pour traiter les sous-approximations et les disjonctions, alors que les techniques à base de plus-faibles-préconditions conduisent à de mauvaises surapproximations. Il est donc raisonnable d'espérer combiner ces techniques pour profiter de leurs forces respectives. L'algorithme ABSGENERIC présenté à la Figure A.2 propose une telle combinaison, à partir d'une technique de génération d'invariants INVGEN, d'un calcul de précondition PRECOND et d'une technique d'élimination de quantificateurs QUANTELIM. En remplaçant INVGEN par une instance d'interprétation abstraite, PRECOND par un calcul par plus-faibles-préconditions et QUANTELIM par l'élimination de Fourier-Motzkin, nous obtenons un algorithme d'inférence d'annotations qui, bien qu'incomparable en théorie, donne de bien meilleurs résultats que les techniques existantes sur des exemples typiques de programmes C. La complexité doublement exponentielle de cet algorithme le rend cependant trop coûteux pour les programmes réels, ce qui justifie l'utilisation d'un calcul par plus-faibles-préconditions moins précis mais plus efficace pour PRECOND, ce qui ne change pas la complexité en théorie mais rend l'algorithme utilisable en pratique.

A.3.2 Programmes typés avec partage mémoire

Nous considérons maintenant les programmes fortement typés avec partage mémoire. Comme précédemment, le typage fort garantit qu'une zone mémoire n'a qu'un seul type possible. En revanche, elle peut désormais être accédée par plusieurs chemins. Nous étendons le langage JESSIE avec un prédicat *separated* pour pouvoir exprimer la séparation de zones mémoires. Alors que le partage mémoire ou *aliasing* est utilisé dans de nombreux cas par les programmeurs C de façon contrôlée, les techniques d'analyse d'alias [88, 178] sont souvent trop imprécises et les techniques de contrôle d'alias [151] trop restrictives.

```
define ABSGENERIC:
  input programme P
  output annotations logiques pour P
  calculer des invariants I par INVGEN
  for chaque assertion C do
    define $\phi_C$ comme C affaibli par $I_C$: $\phi_C = I_C \Longrightarrow C$
    define $\phi$ comme le résultat de l'application de PRECOND à $\phi_C$
    define $\psi$ comme le résultat de l'application de QUANTELIM à $\phi$
    utiliser $\psi$ pour renforcer la précondition de P
  done
```

Figure A.2: Algorithme ABSGENERIC

L'analyse d'alias de Steensgaard découpe la mémoire en régions distinctes qui, parce qu'elles correspondent à un sous-typage du programme, peuvent aussi bien être utilisées pour rendre l'algorithme *paths-may-overlap* plus précis que pour améliorer la traduction de JESSIE à WHY. Cette analyse présente cependant deux problèmes : (i) l'absence de sensibilité au contexte, ce qui entraîne parfois un découpage grossier de la mémoire, et (ii) l'impossibilité d'analyser un programme de façon modulaire, fonction par fonction. Hubert et Marché [94] ont décrit une solution au problème (i) en s'inspirant de l'algorithme de calcul d'effet de Talpin et Jouvelot [167, 168]. Au lieu de calculer des régions globales comme dans l'algorithme de Steensgaard, ils calculent à la fois des régions globales et des régions paramétriques attachées à chaque fonction, qui sont instanciées différemment à chaque appel de fonction. Ce raffinement des régions contribue directement à améliorer la précision de l'algorithme *paths-may-overlap* et la traduction de JESSIE à WHY. Cependant, cette approche nécessite de vérifier des contraintes supplémentaires sur le programme, présentées initialement par Reynolds dans son travail sur le contrôle syntaxique des interférences [152] : à chaque appel, deux régions paramétriques ne doivent jamais être instanciées par la même région ou par une région globale déjà utilisée par la fonction. Ces contraintes garantissent en effet la séparation des zones mémoires représentées par deux régions différentes. Cette analyse est donc incomplète, c.à.d. que les programmes qui ne respectent pas ces contraintes ne peuvent pas être analysés, ce qui représente la plupart des programmes systèmes étudiés dans le cadre de cette thèse.

Nous proposons un raffinement de l'algorithme de Hubert et Marché qui résout les problèmes (ii) de modularité et (iii) d'incomplétude. Pour résoudre le problème (ii), nous proposons d'utiliser un calcul d'effets basé sur des invariants calculés par interprétation abstraite. Cela permet d'exprimer la séparation des régions dans une fonction sous forme d'applications du prédicat de séparation à des ensembles de pointeurs qui surapproximent les effets de la fonction sur ces régions. Ainsi, notre algorithme appliqué à une fonction en dehors de son contexte d'appel retourne une précondition de séparation devant être vérifiée à l'appel. Ces préconditions de séparation donnent aussi une solution au problème (iii) pour de nombreux programmes : au lieu d'échouer quand deux régions paramétriques sont instanciées par la même région lors d'un appel, il suffit d'ajouter une assertion à vérifier qui reprend les applications du prédicat de séparation à ces régions. Enfin, deux régions seulement lues ne peuvent pas être la cause d'interférences, comme déjà remarqué par Reynolds [152]. Cela permet de raffiner encore l'algorithme d'inférence de régions, de façon à fournir une solution complète pour les programmes sans interférences, qui représentent la plupart des programmes rencontrés en pratique. Pour les programmes comme `swap` utilisés dans des contextes interférant (avec `x` et `y` aliasés), nous indiquons des solutions possibles permettant d'interagir avec l'utilisateur.

```
void swap(int *x, int *y) {
  int tmp = *x;
  *x = *y;
  *y = tmp;
}
```

A.3.3 Programmes avec unions et casts

Nous considérons enfin des extensions du langage JESSIE permettant la traduction vers JESSIE des programmes C avec unions et casts, ainsi que l'adaptation des algorithmes déjà définis.

Comme observé par Siff *et al.* [161, 35], la majorité des casts dans les programmes C systèmes sont des casts entre sous-types physiques, c.à.d. que les types des structures source et cible du cast possèdent les mêmes champs aux mêmes décalages par rapport à l'adresse pointée. Ces casts révèlent donc une hiérarchie de types sous-jacente exploitée par le programme. Nous étendons le langage JESSIE avec une notion de sous-typage de façon à pouvoir exprimer ces relations entre types, et nous étendons la sémantique de JESSIE afin de mémoriser le type réel des objets en mémoire. En gardant les casts vers des types dérivés dans la hiérarchie par une assertion appropriée, nous garantissons que les programmes JESSIE avec sous-typage respectent un typage fort, comme précédemment. Les algorithmes étudiés précédemment pour l'inférence d'annotations et la séparation mémoire en régions peuvent donc leur être appliqués.

De même, la majorité des unions sont modérées, c.à.d. que leurs champs ne sont accédés qu'en accédant d'abord à l'union tout entière. Ceci est garanti par l'utilisation d'un critère syntaxique, l'absence de prise d'adresse des champs de l'union. Ces unions modérées sont de deux types :

- *unions discriminées* - Dans ces unions, seul le dernier champ écrit est lu. L'adaptation des algorithmes vus précédemment à ce type d'unions se fait facilement, en interprétant l'écriture dans un champ d'union comme modifiant en parallèle les autres champs de l'union d'une manière aléatoire.

- *unions bas-niveau* - Dans ces unions, tout champ peut être lu à tout moment. Nous traduisons ces unions par des structures avec un seul champ contenant un vecteur d'octets. L'accès à un champ spécifique de l'union correspond à l'interprétation vers un type spécifique d'une portion du vecteur d'octet complet. L'adaptation des algorithmes vus précédemment est immédiate, puisque l'union est traduite en une structure avec un seul champ, un cas déjà traité.

Nous choisissons de traduire les unions avec champs de type pointeur comme des unions discriminées, afin de pouvoir continuer à utiliser le découpage en régions mémoires décrit à la Section A.3.2.

Pour les programmes avec unions et casts qui ne rentrent pas dans les catégories évoquées précédemment, un nouveau modèle mémoire bas-niveau est nécessaire, de façon à pouvoir interpréter un même vecteur d'octets comme un type ou un autre. Le problème avec cette approche, même en limitant l'utilisation du modèle bas-niveau aux seules régions concernées par un tel cast ou une telle union, est la propagation du modèle bas-niveau à de nombreuses parties du programme [2]. Pour éviter ce problème, nous proposons de confiner le modèle bas-niveau à l'intérieur de chaque fonction f, de façon à ce que les fonctions appelant f ou appelées par f puissent être analysées indépendamment de f. Cette localité du modèle bas-niveau est obtenue en :

- *isolant f des fonctions appelantes* - Bien que f soit vérifiée dans un modèle mémoire bas-niveau, une déclaration de fonction f' équivalente dans un modèle typé doit être utilisée dans les fonctions appelantes. La validité de pointeur dans le modèle bas-niveau doit être vérifiée par rapport à des accesseurs dépendant de la structure accédée.

- *isolant f des fonctions appelées* - Avant l'appel d'une fonction g dans f, les zones mémoires passant d'un modèle mémoire bas-niveau à un modèle mémoire typé doivent être traduites. Une condition de séparation garantit la séparation des champs de structure. Une traduction inverse est effectuée après le retour de l'appel à g.

Les préconditions générées pour garantir la séparation en régions distinctes et celles générées pour garantir la localité du modèle bas-niveau s'ajoutent sans générer de conflit.

A.4 Expériences sur des programmes C réels

Les techniques décrites dans cette thèse ont été implémentées dans l'outil Frama-C [73], une plateforme libre pour l'analyse modulaire des programmes C. Les obligations de preuve (OP) générées ont été prouvées par les prouveurs automatiques Alt-Ergo (A), Simplify (S), Yices (Y) ou Z3 (Z).

A.4.1 Bibliothèques de chaînes de caractères

Les chaînes de caractères en C sont des tableaux de caractères terminés par un caractère nul. L'absence de cette sentinelle dans les bornes d'un bloc mémoire alloué n'est aucunement garantie par le langage, ce qui est la cause de la plupart des vulnérabilités les plus graves des programmes C. Il est donc important de vérifier la sûreté de fonctionnement des bibliothèques manipulant des chaînes de caractères, avant même de vérifier la sûreté de fonctionnement des programmes utilisant ces bibliothèques. Les techniques décrites dans cette thèse s'appliquent parfaitement dans ce contexte modulaire.

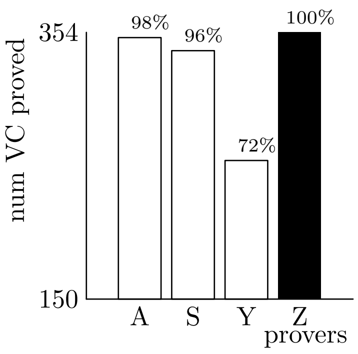

Figure A.3: Modèle d'entiers exacts

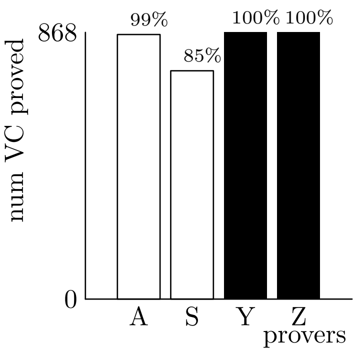

Figure A.4: Modèle d'entiers bornés

MINIX 3 est un système d'exploitation libre ayant comme objectif la sûreté de fonctionnement. En particulier, sa bibliothèque standard de chaînes de caractères est implémentée de façon simple et idiomatique. Voici par exemple le code de la fonction de copie de chaîne :

```
char *strcpy(char *ret, const char *s2) {
  char *s1 = ret;
  while (*s1++ = *s2++)
    /* EMPTY */ ;
  return ret;
}
```

En tout, la bibliothèque de chaînes comporte 22 fonctions. Nous avons analysé ces fonctions complètement automatiquement, ce qui comprend la génération d'annotations, le découpage en régions mémoires et la vérification de la sûreté des accès mémoire et des opérations entières. La Figure A.3 décrit les résultats obtenus avec différents prouveurs automatiques dans un modèle où les entiers mathématiques sont utilisés (21 fonctions analysées) et la Figure A.4 décrits les mêmes résultats dans un modèle où les véritables entiers machines bornés sont utilisés (20 fonctions analysées). Dans les deux cas, nous réussissons à prouver complètement la sûreté de fonctionnement des fonctions analysées vis-à-vis des préconditions générées.

La bibliothèque de chaînes sûres du CERT définit un type abstrait `string_m` à utiliser à la place des chaînes natives du C. Cette structure encapsule un ensemble de champs devant respecter un invariant complexe pour garantir la sûreté de fonctionnement, les fonctions de la bibliothèque devant établir ou maintenir cet invariant suivant les cas. Voici l'invariant *managed-string* reconstruit à partir du code de la bibliothèque, à partir d'essais-erreurs lors de la vérification :

```
/*@ predicate almost_managed_cstring(string_m s) =
  @   s→strtype ≡ STRTYPE_NTBS              // a plain character string
  @   ∧ (valid_string(s→str.cstr)           // content is a valid string
  @       ∧ strlen(s→str.cstr) < s→size     // and string size is bounded
  @       ∧ \valid_range(s→str.cstr,        // and buffer size is bounded
  @                         0,s→size − 1)
  @    ∨ s→str.cstr ≡ NULL                 // or pointer is NULL
  @       ∧ s→size ≡ 0);                   // and size is null too
  @
  @ predicate managed_cstring(string_m s) =
  @   almost_managed_cstring(s)
  @   ∧ valid_string_or_null(              // filter is a valid string
  @                       s→charset.cstr)
  @   ∧ (s→maxsize ≡ 0                     // maximum not set
  @      ∨ s→size ≤ s→maxsize);            // or size is below maximum
  @
  @ predicate almost_managed_wstring(string_m s) =
  @   s→strtype ≡ STRTYPE_WSTR             // a wide character string
  @   ∧ (valid_wstring(s→str.wstr)         // content is a valid string
  @       ∧ wcslen(s→str.wstr) < s→size   // and string size is bounded
  @       ∧ \valid_range(s→str.wstr,       // and buffer size is bounded
  @                         0,s→size − 1)
  @    ∨ s→str.wstr ≡ NULL                 // or pointer is NULL
  @       ∧ s→size ≡ 0);                   // and size is null too
  @
  @ predicate managed_wstring(string_m s) =
  @   almost_managed_wstring(s)
  @   ∧ valid_wstring_or_null(             // filter is a valid string
  @                       s→charset.wstr)
  @   ∧ (s→maxsize ≡ 0                     // maximum not set
  @      ∨ s→size ≤ s→maxsize);            // or size is below maximum
  @
  @ predicate managed_string(string_m s) =
  @   \valid(s) ∧ (managed_cstring(s) ∨ managed_wstring(s));
  @*/
```

Etant donné la complexité de l'invariant à maintenir, nous avons annoté manuellement 49 fonctions de la bibliothèque (à l'exclusion des fonctions d'entrée-sortie). Nous avons cherché à prouver l'absence d'erreurs lors des accès mémoire dans un modèle mémoire simplifié ignorant les déallocations mémoire. Cela nous a permis de découvrir 45 erreurs de programmation pouvant être exploitées lors d'attaques, erreurs confirmées par l'équipe du CERT. Une fois ces erreurs corrigées, nous avons réussi à prouver la sûreté des accès mémoire dans le modèle mémoire simplifié utilisé :

- *preuve automatique* - Comme la Figure A.5 le montre, nous avons prouvé 99,9% des obligations de preuve automatiquement, soit 18.080 sur un total de 18.089. La Figure A.6 montre que chacun des trois prouveurs automatiques prouve seul un nombre non négligeable d'obligations de preuves, ce qui est un argument pour l'utilisation conjointe de plusieurs prouveurs.

- *revue de code manuelle* - Nous avons démontré la validité des 9 obligations de preuve restantes manuellement, par un raisonnement non formel.

A.4.2 Jeux de tests de vulnérabilités

Deux jeux de tests, la suite Verisec [112] et le jeux de tests de Zitser [185], isolent des vulnérabilités connues de dépassement de capacité dans des programmes libres populaires,

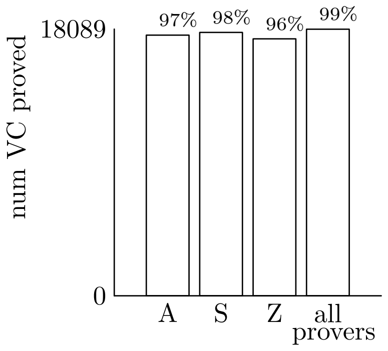

Figure A.5: OP prouvées

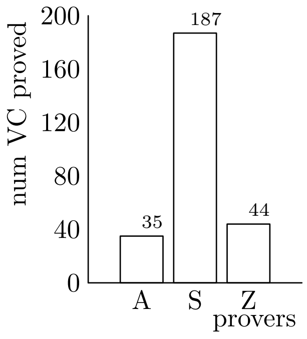

Figure A.6: OP prouvées une seule fois

servant par ex. à l'envoi d'email ou à la gestion d'impression. Chaque test est fourni dans une version "bad" qui reproduit la vulnérabilité, et une version "ok" corrigée. La difficulté principale, observée par Zitser lors de son expérience initiale, est de distinguer les versions corrigées des versions incorrectes.

La suite Verisec contient 140 tests de degré de difficulté variable et le jeux de tests de Zitser contient seulement 14 tests de difficulté élevée. Bien que chaque test soit équipé d'une fonction `main`, nous analysons chaque fonction indépendamment de son contexte d'appel. Dans les deux cas, nous avons au préalable annoté le code avec des qualificateurs de type indiquant quelles variables et champs de structures sont utilisés comme des chaînes de caractères. Sur les 140 tests "ok" de la suite Verisec :

- 35 tests n'ont pas pu être analysés (limitations) ;
- 78 tests ont été complètement prouvés sûrs ;
- 27 tests ont été partiellement prouvés, la majorité n'ayant que quelques obligations de preuve non prouvées.

La Figure A.7 montre que 99,7% des OP sont prouvées automatiquement, alors que la Figure A.8 montre encore une fois que chaque prouveur prouve seul un nombre non négligeable d'OP. Le prouveur A_S est une modification du prouveur Alt-Ergo qui applique certaines heuristiques pour sélectionner les hypothèses les plus pertinentes.

Aucun des tests de Zitser n'a pu être analysé. Dans 9 cas sur 14, cela a été causé par le dépassement de la limite imposée en espace ou en temps. Nous envisageons d'implémenter une meilleure technique d'élimination de quantificateurs que l'élimination de Fourier-Motzkin pour pouvoir analyser ces programmes, par ex. la technique proposée par Monniaux [134].

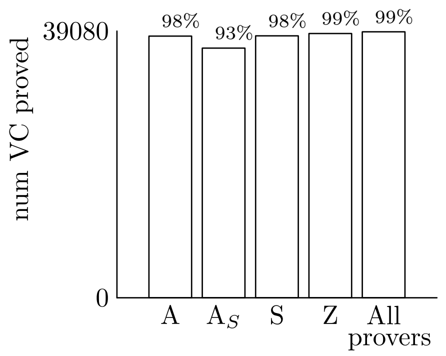

Figure A.7: OP prouvées

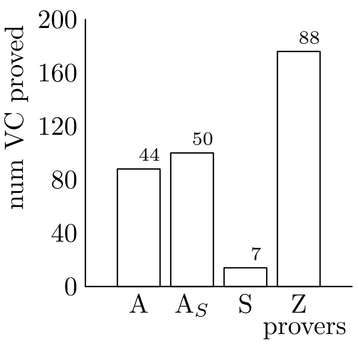

Figure A.8: OP prouvées une seule fois

A.5 Conclusion

Quelques années avant que cette thèse ne débute, les résultats obtenus par deux outils (Astrée [21], CGS [174]) spécialisés pour des programmes C restreints utilisés en avionique ont démontré que la preuve de sûreté de fonctionnement de programmes C réels de plusieurs centaines de milliers de lignes était possible. Ces outils basés sur l'interprétation abstraite laissaient ouverts un certain nombre de problèmes, essentiellement l'analyse de programmes incomplets (modularité), la prise en compte du partage mémoire (aliasing), l'adaptation aux programmes C non contraints (unions et casts). Dans sa thèse de doctorat portant sur la preuve de programmes C industriels par vérification déductive [93], Hubert concluait par la nécessité de répondre aux trois mêmes problèmes, en plus du problème de génération d'annotations propre à la vérification déductive.

Dans cette thèse, nous proposons des solutions à chacun de ces problèmes, dans le cadre de la vérification déductive de programmes C systèmes. Ces solutions sont à la fois automatiques et modulaires :

- *génération d'annotations* - Nous présentons une technique de génération d'annotations logiques basée sur une combinaison nouvelle des techniques d'interprétation abstraite, de plus-faibles-préconditions et d'élimination de quantificateurs. En particulier, nous générons des préconditions suffisantes précises, ce qui n'a pas été fait précédemment.

- *séparation de la mémoire en régions* - Nous présentons une technique de contrôle du partage mémoire basée sur l'analyse de régions de Hubert et Marché. Cette technique permet d'exprimer les conditions de séparation comme des obligations de preuve adaptées à la preuve automatique.

- *traitement des unions et casts* - Nous présentons un modèle mémoire mixte typé ou bas-niveau qui permet de traiter les unions et les casts en vérification déductive, tout en gardant au maximum les bénéfices du modèle mémoire typé. Cette technique repose sur la génération de régions mémoires susmentionnée.

Ces techniques ont été implémentées dans Frama-C [73], une plateforme libre pour l'analyse modulaire des programmes C. Nous avons ainsi réussi à prouver la sûreté de fonctionnement d'une bibliothèque de chaînes de caractères standard, à trouver un nombre importants d'erreurs de programmation dans une bibliothèque de chaînes "sûres" et à discriminer entre les versions incorrectes et corrigées dans des jeux de tests de vulnérabilités réelles.

La taille modeste des programmes analysés lors de cette thèse ne permet pas de conclure que les techniques développées s'appliqueront aussi facilement à des programmes beaucoup plus gros, mais la propriété de modularité de ces techniques indique que c'est envisageable. Enfin, la preuve de sûreté de fonctionnement des programmes n'est qu'un premier pas vers la preuve de propriétés fonctionnelles souvent plus complexes, pour lesquelles les mêmes techniques seront certainement utiles.

Index

Bibliography

[1] Alex Aiken, Jeffrey S. Foster, John Kodumal, and Tachio Terauchi. Checking and inferring local non-aliasing. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 129–140, New York, NY, USA, 2003. ACM.

[2] June Andronick. *Modélisation et Vérification Formelles de Systèmes Embarqués dans les Cartes à Microprocesseur Plate-Forme Java Card et Système d'Exploitation*. PhD thesis, Université Paris-Sud, 2006.

[3] J. Anton, E. Bush, A. Goldberg, K. Havelund, D. Smith, and A. Venet. Towards the industrial scale development of custom static analyzers. In *Proceedings of the Static Analysis Summit*. U.S. National Institute of Standards and Technology, June 2006.

[4] M. G. Assaad and G. T. Leavens. Alias-free parameters in C for better reasoning and optimization. Technical Report 01-11, Department of Computer Science, Iowa State University, 2001.

[5] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. pages 290–301. Association for Computing Machinery, 1994.

[6] A. Avizienis, J. Laprie, and B. Randell. Fundamental concepts of dependability. Research Report N01145, LAAS-CNRS, April 2001.

[7] R.-J. R. Back and M. Karttunen. *A predicate transformer semantics for statements with multiple exits*, 1983. unpublished manuscript.

[8] Thomas Ball and Sriram K. Rajamani. The slam project: debugging system software via static analysis. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–3, New York, NY, USA, 2002. ACM.

[9] A. Baratloo, N. Singh, and T. Tsai. Libsafe: Protecting critical elements of stacks. White paper, December 1999.

[10] Mike Barnett, K. Rustan, M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Proceedings of CASSIS 2004: Construction and Analysis of Safe, Secure and Interoperable Smart devices*, pages 49–69. Springer, 2004.

[11] Mike Barnett, Bor yuh Evan Chang, Robert Deline, Bart Jacobs, and K. Rustanm. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005, volume 4111 of Lecture Notes in Computer Science*, pages 364–387. Springer, 2006.

[12] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.

[13] G. Barthe, L. Burdy, J. Charles, B. Grégoire, M. Huisman, J.-L. Lanet, M. Pavlova, and A. Requet. JACK: a tool for validation of security and behaviour of Java applications. In *FMCO: Proceedings of 5th International Symposium on Formal Methods for Components and Objects*, Lecture Notes in Computer Science. Springer-Verlag, 2007. To appear.

[14] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. ACSL: ANSI/ISO C specification language. Technical report, 2008.

[15] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.

[16] Kris Bell, Lars Ivar Igesund, Sean Kelly, and Michael Parker. *Learn to Tango with D*. The Expert's Voice. Apress, 2008.

[17] Josh Berdine, Cristiano Calcagno, and Peter W. O'hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *In Proceedings of the Fourth International Symposium on Formal Methods for Components and Objects*, pages 115–137. Springer, 2005.

[18] Joachim van den Berg and Bart Jacobs. The loop compiler for java and jml. In *TACAS 2001: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 299–312, London, UK, 2001. Springer-Verlag.

[19] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development*. Springer-Verlag, 2004.

[20] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Checking memory safety with BLAST. In M. Cerioli, editor, *Proceedings of the 8th International Conference on Fundamental Approaches to Software Engineering (FASE 2005, Edinburgh, April 2-10)*, LNCS 3442, pages 2–18. Springer-Verlag, Berlin, 2005.

[21] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proceedings of*

the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03), pages 196–207, San Diego, California, USA, June 7–14 2003. ACM Press.

[22] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In *FM 2006: Int. Symp. on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475. Springer, 2006.

[23] Sandrine Blazy and Xavier Leroy. Formal verification of a memory model for C-like imperative languages. In *International Conference on Formal Engineering Methods (ICFEM 2005)*, volume 3785 of *Lecture Notes in Computer Science*, pages 280–299. Springer, 2005.

[24] Sylvie Boldo and Jean-Christophe Filliâtre. Formal verification of floating-point programs. In Peter Kornerup and Jean-Michel Muller, editors, *Proceedings of the 18th IEEE Symposium on Computer Arithmetic*, pages 187–194, Montpellier, France, June 2007.

[25] Alex Borgida, John Mylopoulos, and Raymond Reiter. On the frame problem in procedure specifications. *IEEE Trans. Softw. Eng.*, 21(10):785–798, 1995.

[26] François Bourdoncle. Assertion-based debugging of imperative programs by abstract interpretation. In *ESEC '93: Proceedings of the 4th European Software Engineering Conference on Software Engineering*, pages 501–516, London, UK, 1993. Springer-Verlag.

[27] François Bourdoncle. Efficient chaotic iteration strategies with widenings. In *In Proceedings of the International Conference on Formal Methods in Programming and their Applications*, pages 128–141. Springer-Verlag, 1993.

[28] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[29] Bulba and Kil3r. Bypassing Stackguard and Stackshield. *Phrack Magazine*, 56, January 2000.

[30] Rod Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7:23–50, 1972.

[31] Sascha Böhme, Michal Moskal, Wolfram Schulte, and Burkhart Wolff. HOL-Boogie - an interactive prover-backend for the verified C compiler. *Journal of Automated Reasoning (JAR)*, 2007.

[32] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Footprint analysis: A shape analysis that discovers preconditions. In *Proceedings of the 14th International Static Analysis Symposium*, volume 4634 of *Lecture Notes in Computer Science*, pages 402–418. Springer-Verlag, August 2007.

[33] Sagar Chaki, Edmund Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 385–395, Washington, DC, USA, 2003. IEEE Computer Society.

[34] Sagar Chaki and Scott Hissam. Certifying the absence of buffer overflows. Technical Note CMU/SEI-2006-TN-030, Carnegie-Mellon University/Software Engineering Institute, September 2006.

[35] Satish Chandra and Thomas Reps. Physical type checking for C. *SIGSOFT Softw. Eng. Notes*, 24(5):66–75, 1999.

[36] Bor-Yuh Evan Chang and K. Rustan M. Leino. Abstract interpretation with alien expressions and heap structures. In *VMCAI '05: Proceedings of The 6th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 147–163, 2005.

[37] Ramkrishna Chatterjee, Barbara G. Ryder, and William A. Landi. Relevant context inference. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 133–146, New York, NY, USA, 1999. ACM.

[38] Ben-Chung Cheng and Wen-Mei W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 57–69, New York, NY, USA, 2000. ACM.

[39] Information technology – common language infrastructure (CLI) partitions I to VI. Technical Report ISO/IEC 23271:2006, 2006.

[40] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[41] Edmund Clarke and Yuan Lu. Counterexample-guided abstraction refinement. In *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*, pages 154–169. Springer, 2000.

[42] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[43] Ernie Cohen, Michał Moskal, Wolfram Schulte, and Stephan Tobies. A precise yet efficient memory model for c. oct 2008.

[44] Michael Colon, Sriram Sankaranarayanan, and Henny Sipma. Linear invariant generation using non-linear constraint solving. In *Proc. of the Int. Conf. on Computer Aided Verification (CAV)*, volume 2725 of *Lecture Notes in Computer Science*, pages 420–432, 2003.

[45] X3J11 committee. *Programming Language C*. American National Standards Institute, 1989.

[46] Sylvain Conchon, Evelyne Contejean, Johannes Kanig, and Stéphane Lescuyer. Cc(x): Semantic combination of congruence closure with solvable theories. *Electronic Notes in Theoretical Computer Science*, 198(2):51–69, May 2008.

[47] Jeremy Condit, Brian Hackett, Shuvendu Lahiri, and Shaz Qadeer. Unifying type checking and property checking for low-level code. In *POPL '09: Proceedings of the 36th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2009.

[48] Jeremy Condit, Matthew Harren, Scott McPeak, George C. Necula, and Westley Weimer. CCured in the real world. *SIGPLAN Not.*, 38(5):232–244, 2003.

[49] Jean-François Couchot and Thierry Hubert. A Graph-based Strategy for the Selection of Hypotheses. In *FTP 2007 - International Workshop on First-Order Theorem Proving*, Liverpool, UK, September 2007.

[50] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY.

[51] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 269–282, New York, NY, USA, 1979. ACM.

[52] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96, New York, NY, USA, 1978. ACM.

[53] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointGuard: protecting pointers from buffer overflow vulnerabilities. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 7–7, Berkeley, CA, USA, 2003. USENIX Association.

[54] Flaviu Cristian. Correct and robust programs. 10(2):163–174, March 1984. Special Section on Specification and Verification.

[55] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors. *Structured programming*. Academic Press Ltd., London, UK, UK, 1972.

[56] Ewen Denney, Bernd Fischer, and Johann Schumann. An empirical evaluation of automated theorem provers in software certification. *International Journal on Artificial Intelligence Tools*, 15(1):81–107, February 2006.

[57] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[58] Alain Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. *SIGPLAN Not.*, 29(6):230–241, 1994.

[59] Edsger W. Dijkstra. *A discipline of programming*. Series in Automatic Computation. 1976.

[60] David L. Dill. *Trace theory for automatic hierarchical verification of speed-independent circuits*. MIT Press, Cambridge, MA, USA, 1989.

[61] Dino Distefano and Matthew J. Parkinson J. jstar: towards practical verification for java. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 213–226, New York, NY, USA, 2008. ACM.

[62] Nurit Dor, Michael Rodeh, and Mooly Sagiv. CSSV: towards a realistic tool for statically detecting all buffer overflows in C. *SIGPLAN Not.*, 38(5):155–167, 2003.

[63] Bruno Dutertre and Leonardo de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *Proceedings of the 18th Computer-Aided Verification conference*, volume 4144 of *LNCS*, pages 81–94. Springer-Verlag, 2006.

[64] Nicholas Enticknap. IT salary survey. *Computer Weekly*, February 2008.

[65] Ana M. Erosa and Laurie J. Hendren. Taming Control Flow: A Structured Approach to Eliminating GOTO Statements. In *ICCL*, 1994.

[66] Pietro Ferrara, Francesco Logozzo, and Manuel Fähndrich. Safer unsafe code for .NET. In ACM Press, editor, *Proceedings of the 23rd ACM Conference on Object-oriented Programming (OOPSLA 2008)*, October 2008.

[67] J.-C. Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, March 2003.

[68] Jean-Christophe Filliâtre and Claude Marché. Multi-prover verification of C programs. In Jim Davies, Wolfram Schulte, and Mike Barnett, editors, *6th International Conference on Formal Engineering Methods*, volume 3308 of *LNCS*, pages 15–29, Seattle, WA, USA, November 2004.

[69] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *19th International Conference on Computer Aided Verification*, Berlin, Germany, July 2007.

[70] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 2002. ACM.

[71] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 191–202, New York, NY, USA, 2002. ACM.

[72] Robert W. Floyd. Assigning meanings to programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32, 1967.

[73] Framework for the modular analysis of C, 2008. `http://www.frama-c.cea.fr`.

[74] Jeff Gennari, Shaun Hedrick, Fred Long, Justin Pincar, and Robert C. Seacord. Ranged integers for the C programming language. Technical Report CMU/SEI-2007-TN-027, Software Engineering Institute, September 2007.

[75] Patrice Godefroid. Model checking for programming languages using VeriSoft. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 174–186, New York, NY, USA, 1997. ACM.

[76] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, 1991.

[77] Denis Gopan and Thomas W. Reps. Low-level library analysis and summarization. In Werner Damm and Holger Hermanns, editors, *CAV '07: Proceedings of the 19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 68–81. Springer, 2007.

[78] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *CAV '97: Proceedings of the 9th International Conference on Computer Aided Verification*, pages 72–83, London, UK, 1997. Springer-Verlag.

[79] Bhargav Gulavani and Sumit Gulwani. A numerical abstract domain based on "expression abstraction" and "max operator" with application in timing analysis. In *CAV '08: Proceedings of the 20th International Conference on Computer Aided Verification*, 2008.

[80] Sumit Gulwani and Ashish Tiwari. Combining abstract interpreters. In *Annual ACM Conference on Programming Language Design and Implementation*. ACM, June 2006.

[81] Sumit Gulwani and Ashish Tiwari. Assertion checking unified. In *The 8th International Conference on Verification, Model Checking and Abstract Interpretation*. Springer, January 2007.

[82] Yuri Gurevich and James K. Huggins. The semantics of the C programming language. In *Computer Science Logic, volume 702 of LNCS*, pages 274–308. Springer, 1993.

[83] Brian Hackett and Alex Aiken. How is aliasing used in systems software? In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 69–80, New York, NY, USA, 2006. ACM.

[84] Brian Hackett, Manuvir Das, Daniel Wang, and Zhe Yang. Modular checking for buffer overflows in the large. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 232–241, New York, NY, USA, 2006. ACM.

[85] John Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, February 2009.

[86] Thomas E. Hart, Kelvin Ku, David Lie, Marsha Chechik, and Arie Gurfinkel. Ptyasm: Software model checking with proof templates. In *In Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE'08)*, September 2008.

[87] Laurie J. Hendren, C. Donawa, Maryam Emami, Guang R. Gao, Justiani, and B. Sridharan. Designing the McCAT compiler based on a family of structured intermediate representations. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, pages 406–420, London, UK, 1993. Springer-Verlag.

[88] Michael Hind. Pointer analysis: haven't we solved this problem yet? In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 54–61, New York, NY, USA, 2001. ACM.

[89] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580 and 583, October 1969.

[90] C. A. R. Hoare and N. Wirth. An axiomatic definition of the programming language pascal. *Acta Informatica*, (Volume 2, Number 4), December 1973.

[91] Tony Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, 2003.

[92] Greg Hoglund and Gary Mcgraw. *Exploiting Software : How to Break Code*. Addison-Wesley Professional, February 2004.

[93] Thierry Hubert. *Analyse Statique et preuve de Programmes Industriels Critiques*. PhD thesis, Université Paris-Sud, 2008.

[94] Thierry Hubert and Claude Marché. Separation analysis for deductive verification. In *Heap Analysis and Verification (HAV'07)*, Braga, Portugal, March 2007.

[95] M. Huisman. *Reasoning about Java Programs in Higher Order Logic with PVS and Isabelle*. PhD thesis, University of Nijmegen, 2001.

[96] Charles Hymans and Olivier Levillain. Newspeak, Doubleplussimple Minilang for Goodthinkful Static Analysis of C. Technical Note 2008-IW-SE-00010-1, EADS IW/SE, 2008.

[97] International Organization for Standardization. *ISO/IEC 9899:1990: Programming Languages – C*, 1990.

[98] International Organization for Standardization. *ISO/IEC 9899:1999: Programming Languages – C*, 2000.

[99] Franco Ivancic, Ilya Shlyakhter, Aarti Gupta, and Malay K. Ganai. Model checking C programs using F-SOFT. In *ICCD '05: Proceedings of the 2005 International Conference on Computer Design*, pages 297–308, Washington, DC, USA, 2005. IEEE Computer Society.

[100] Paul B. Jackson, Bill J. Ellis, and Kathleen Sharp. Using SMT solvers to verify high-integrity programs. In *AFM '07: Proceedings of the second workshop on Automated formal methods*, pages 60–68, New York, NY, USA, 2007. ACM.

[101] Bart Jacobs and Frank Piessens. The verifast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, August 2008.

[102] M. Janota. Assertion-based loop invariant generation. In *In Proceedings of the 1st International Workshop on Invariant Generation (WING '07)*, Hagenberg, Austria, 2007. Workshop at CALCULEMUS 2007.

[103] Ranjit Jhala, Rupak Majumdar, and Ru-Gang Xu. State of the union: Type inference via Craig interpolation. In Orna Grumberg and Michael Huth, editors, *TACAS '07: Proceedings of The 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 4424 of *Lecture Notes in Computer Science*. Springer, 2007.

[104] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *Proc. 2002 USENIX Annual Technical Conference*, pages 275–288, Berkeley, CA, USA, 2002.

[105] Gilles Kahn. Natural semantics. In *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science*, volume 247, pages 22–39, 1987.

[106] Gilles Kahn. Natural semantics. In K. Fuchi and M. Nivat, editors, *Programming of Future Generation Computers*, pages 237–257. Elsevier, 1988.

[107] Cem Kaner, Jack L. Falk, and Hung Quoc Nguyen. *Testing Computer Software, Second Edition*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[108] Michael Karr. Affine relationships among variables of a program. In *Acta Informatica*, 1976.

[109] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[110] Steve King and Carroll Morgan. Exits in the refinement calculus. *Formal Asp. Comput.*, 7(1):54–76, 1995.

[111] David Koes, Mihai Budiu, and Girish Venkataramani. Programmer specified pointer independence. In *MSP '04: Proceedings of the 2004 workshop on Memory system performance*, pages 51–59, New York, NY, USA, 2004. ACM.

[112] Kelvin Ku, Thomas E. Hart, Marsha Chechik, and David Lie. A buffer overflow benchmark for software model checkers. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 389–392, New York, NY, USA, 2007. ACM.

[113] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. *SIGPLAN Not.*, 27(7), 1992.

[114] Chris Lattner, Andrew Lenharth, and Vikram Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. pages 278–289, 2007.

[115] Gary T. Leavens, K. Rustan M. Leino, Erik Poll, Clyde Ruby, and Bart Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA 2000 Companion, Minneapolis, Minnesota*, pages 105–106, 2000.

[116] Dirk Leinenbach, Wolfgang Paul, and Elena Petrova. Towards the formal verification of a C0 compiler: Code generation and implementation correctnes. In *SEFM '05: Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*, pages 2–12, Washington, DC, USA, 2005. IEEE Computer Society.

[117] K. R. M. Leino and G. Nelson. An extended static checker for Modula-3. 1383, 1998.

[118] K. Rustan M. Leino. *Toward Reliable Modular Programs*. PhD thesis, Caltech, 1995. Technical Report Caltech-CS-TR-95-03.

[119] K. Rustan M. Leino. Efficient weakest preconditions. *Inf. Process. Lett.*, 93(6):281–288, 2005.

[120] K. Rustan M. Leino and Francesco Logozzo. Loop invariants on demand. In *APLAS '05: Proceedings of The 3rd ASIAN Symposium on Programming Languages and Systems*, LNCS, pages 119–134. Springer-Verlag, 2005.

[121] K. Rustan M. Leino and Francesco Logozzo. Using widenings to infer loop invariants inside an SMT solver, or: A theorem prover as abstract domain. Technical Report RISC-Linz Report Series No. 07-07, RISC, Hagenberg, Austria, June 2007. Proc. WING'07.

[122] K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Java to guarded commands translation. Technical Report ESCJ 16c, Compaq Research Labs, August 1998.

[123] K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking java programs via guarded commands. In *Proceedings of the Workshop on Formal Techniques for Java-like Programs*, pages 110–111, London, UK, 1999. Springer-Verlag.

[124] K. Rustan M. Leino and Jan L. A. van de Snepscheut. Semantics of exceptions. In *PROCOMET '94: Proceedings of the IFIP TC2/WG2.1/WG2.2/WG2.3 Working Conference on Programming Concepts, Methods and Calculi*, pages 447–466, Amsterdam, The Netherlands, The Netherlands, 1994. North-Holland Publishing Co.

[125] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. *SIGPLAN Not.*, 41(1):42–54, 2006.

[126] Donglin Liang and Mary Jean Harrold. Efficient computation of parameterized pointer information for interprocedural analyses. In *SAS '01: Proceedings of the 8th International Symposium on Static Analysis*, pages 279–298, London, UK, 2001. Springer-Verlag.

[127] Francesco Logozzo and Manuel Fähndrich. On the relative completeness of bytecode analysis versus source code analysis. In *CC '08: Proceedings of The 17th International Conference on Compiler Construction*, volume 4959 of *LNCS*. Springer-Verlag, 2008.

[128] M.S. Manasse and C.G. Nelson. Correct compilation of control structures. Technical report, AT&T Bell Laboratories, 1984.

[129] Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):89–106, 2004. `http://krakatoa.lri.fr`.

[130] Laurent Mauborgne and Xavier Rival. Trace partitioning in abstract interpretation based static analyzers. In M. Sagiv, editor, *Proc. ESOP'05*, volume 3444 of *Lecture Notes in Computer Science*, pages 5–20, 2005.

[131] E. Meijer and J. Gough. Technical overview of the Common Language Runtime, 2000.

[132] A. Miné. The octagon abstract domain. *Higher Order Symb. Comp.*, 19(1):31–100, 2006.

[133] MISRA-C:2004 - Guidelines for the use of the C language in critical systems. Technical report, October 2004.

[134] David Monniaux. A quantifier elimination algorithm for linear real arithmetic. In *LPAR'08: Proceedings of the 15th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, 2008.

[135] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *In Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS*, 2008.

[136] Yannick Moy and Claude Marché. Inferring local (non-)aliasing and strings for memory safety. In *Heap Analysis and Verification (HAV'07)*, Braga, Portugal, mar 2007.

[137] Oleg Mürk, Daniel Larsson, and Reiner Hähnle. KeY-C: A tool for verification of C programs. In Frank Pfenning, editor, *Proc. 21st Conference on Automated Deduction (CADE), Bremen, Germany*, volume 4603 of *LNCS*, pages 385–390. Springer-Verlag, 2007.

[138] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, 2005.

[139] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Computational Complexity*, pages 213–228, 2002.

[140] Tobias Nipkow. Reflecting quantifier elimination for linear arithmetic. In O. Grumberg, T. Nipkow, and C. Pfaller, editors, *Formal Logical Methods for System Security and Correctness*, pages 245–266. IOS Press, 2008.

[141] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer-Verlag, London, UK, 2002.

[142] Michael Norrish. Hol 4 kananaskis-4. `http://hol.sourceforge.net/`.

[143] Michael Norrish. *C Formalised in HOL*. PhD thesis, University of Cambridge, November 1998.

[144] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 42, November 1996.

[145] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[146] Nikolaos S. Papaspyrou. *A Formal Semantics for the C Programming Language*. PhD thesis, National Technical University of Athens, February 1998.

[147] Corneliu Popeea, Dana N. Xu, and Wei-Ngan Chin. A practical and precise inference and specializer for array bound checks elimination. In *PEPM '08: Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 177–187, New York, NY, USA, 2008. ACM.

[148] Lyle Ramshaw. Eliminating go to's while preserving program structure. *Journal of the ACM*, 35(4):893–920, 1988.

[149] W. Reif. The KIV-approach to Software Verification. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software – Final Report*, LNCS 1009. Springer, Berlin, 1995.

[150] Chris Ren, Michael Weber, and Gary McGraw. Microsoft compiler flaw technical note. Technical report, Cigital, Inc., February 2002.

[151] John Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Millennial Perspectives in Computer Science*. Palgrave, 2000.

[152] John C. Reynolds. Syntactic control of interference. In *POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 39–46, New York, NY, USA, 1978. ACM.

[153] X. Rival. Understanding the origin of alarms in ASTRÉE. In *12th Static Analysis Symposium (SAS'05)*, volume 3672 of *LNCS*, pages 303–319, London (UK), September 2005. Springer-Verlag.

[154] Atanas Rountev and Barbara G. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. In *CC '01: Proceedings of the 10th International Conference on Compiler Construction*, pages 20–36, London, UK, 2001. Springer-Verlag.

[155] Atanas Rountev, Barbara G. Ryder, and William Landi. Data-flow analysis of program fragments. *SIGSOFT Softw. Eng. Notes*, 24(6):235–252, 1999.

[156] Nicolas Rousset. *Automatisation de la Spécification et de la Vérification d'applications Java Card*. Thèse de doctorat, Université Paris-Sud, June 2008.

[157] Radu Rugina and Martin Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *SIGPLAN Not.*, 35(5):182–195, 2000.

[158] Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2005.

[159] Steve M. Shaner, Gary T. Leavens, and David A. Naumann. Modular verification of higher-order methods with mandatory calls specified by model programs. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 351–368, New York, NY, USA, 2007. ACM.

[160] Jonathan Shapiro, Swaroop Sridhar, and Scott Doerrie. BitC language specification. Technical report, Department of Computer Science, Johns Hopkins University, 2008.

[161] Michael Siff, Satish Chandra, Thomas Ball, Krishna Kunchithapadam, and Thomas Reps. Coping with type casts in C. In *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 180–198, London, UK, 1999. Springer-Verlag.

[162] Axel Simon and Andy King. Analyzing string buffers in C. In *AMAST '02: Proceedings of the 9th International Conference on Algebraic Methodology and Software Technology*, pages 365–379, London, UK, 2002. Springer-Verlag.

[163] Richard M. Stallman and the GCC Developer Community. *GCC Internals Manual.* Free Software Foundation, Inc., 2008.

[164] Artem Starostin. Formal verification of a C-library for strings. Master's thesis, Saarland University, March 2006.

[165] Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, New York, NY, USA, 1996. ACM.

[166] Norihisa Suzuki and Kiyoshi Ishihata. Implementation of an array bound checker. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 132–143, New York, NY, USA, 1977. ACM.

[167] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type region and effect inference. Technical Report EMP-CRI E/150, 1991.

[168] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. In *Seventh Annual IEEE Symposium on Logic in Computer Science, Santa Cruz, California*, pages 162–173, Los Alamitos, California, 1992. IEEE Computer Society Press.

[169] Mads Tofte. *Operational semantics and polymorphic type inference*. Phd thesis, 1988.

[170] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 1997.

[171] Harvey Tuch and Gerwin Klein. A unified memory model for pointers. In *12th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR-12), volume 3835 of LNCS*, pages 474–488, 2005.

[172] Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 97–108, New York, NY, USA, 2007. ACM.

[173] Mark Utting. Reasoning about aliasing. In *In The Fourth Australasian Refinement Workshop*, pages 195–211, 1995.

[174] Arnaud Venet and Guillaume Brat. Precise and efficient static array bound checking for large embedded C programs. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 231–242, New York, NY, USA, 2004. ACM.

[175] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *NDSS Symposium*, pages 3–17, San Diego, CA, February 2000.

[176] Volker Weispfenning. Complexity and uniformity of elimination in Presburger arithmetic. In *ISSAC '97: Proceedings of the 1997 international symposium on Symbolic and algebraic computation*, pages 48–53, New York, NY, USA, 1997. ACM.

[177] Freek Wiedijk. *The Seventeen Provers of the World.* Lecture Notes in Computer Science / Lecture Notes in Artificial Intelligence. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[178] Qiang Wu. Survey of alias analysis. `www.cs.princeton.edu/~jqwu/Memory/`.

[179] Yichen Xie, Andy Chou, and Dawson Engler. Archer: using symbolic, path-sensitive analysis to detect memory access errors. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 327–336, New York, NY, USA, 2003. ACM.

[180] Zhichen Xu. *Safety checking of machine code*. PhD thesis, Univ. of Wisconsin, Madison, December 2000.

[181] Zhichen Xu, Barton P. Miller, and Thomas Reps. Safety checking of machine code. *ACM SIGPLAN Notices*, 35(5):70–82, 2000.

[182] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Scalable shape analysis for systems code. In Aarti Gupta and Sharad Malik, editors, *CAV*, volume 5123 of *Lecture Notes in Computer Science*, pages 385–398. Springer, 2008.

[183] Kwangkeun Yi. *Catching Software Bugs Early at Build Time*, 2007. Online manual.

[184] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. SafeDrive: safe and recoverable extensions using language-based techniques. In *OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, Berkeley, CA, USA, 2006. USENIX Association.

[185] Misha Zitser, Richard Lippmann, and Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. *SIGSOFT Softw. Eng. Notes*, 29(6):97–106, 2004.