On Validating Well Typed Programs Written in the Weakly Typed Programming Language C

A Dissertation

Presented in Partial Fulfillment of the Requirements for the

Degree of Doctorate of Philosophy

with a

Major in Computer Science

in the

College of Graduate Studies

University of Idaho

by

Kevin Arnold Krause

Major Professor: Jim Alves-Foss, Ph.D.

Committee Members: Clinton Jeffery Ph.D.; Paul Oman, Ph.D.;

Michael O'Rourke, Ph.D.

Department Administrator: Gregory Donohoe, Ph.D.

August 2015

Authorization to Submit Dissertation

This dissertation of Kevin Arnold Krause, submitted for the degree of Doctorate of Philosophy with a Major in Computer Science and titled "On Validating Well Typed Programs Written in the Weakly Typed Programming Language C," has been reviewed in final form. Permission, as indicated by the signatures and dates given below, is now granted to submit final copies to the College of Graduate Studies for approval.

Major Professor ____________________ Date __________
Jim Alves-Foss, Ph.D.

Committee Members ____________________ Date __________
Clinton Jeffery, Ph.D.

____________________ Date __________
Paul Oman, Ph.D.

____________________ Date __________
Michael O'Rourke, Ph.D.

Department Administrator ____________________ Date __________
Gregory Donohoe, Ph.D.

Abstract

This dissertation is a case study of type safety with respect to the C programming language. In short, C is not type safe, as its integer data types are not protected against entering one of several possible error conditions. Once a single integer error occurs, an entire system is potentially at risk to fail or is vulnerable to hostile takeover. The consequence of error can be devastating, depending on the critical nature of the system. At worst, the losses could have major implications on national security.

Contained within, the problem space is explored after defining the concepts behind type safety. Then, a syntax like typing specification for the language is introduced and a simplified static typing semantics for its expressions and statements are expressed before a solution is offered with a prototype tool that statically analyzes an abstraction of the original C source code for type safety violations. Algorithms for the tool are based on an enumeration of the likely causes to enter an integer error, a formalization of the static typing semantics of C, and the requirements for safe C language constructs. Because of the tool's underling language and by using the aforementioned formalizations, the tool has the ability to prove that its reasoning about the code it is analyzing is correct.

Acknowledgements

I would like thank my dissertation committee chair, Dr. Jim Alves-Foss and the remaining committee members: Dr. Clinton Jeffery, Dr. Paul Oman, and Dr. Michael O'Rourke. This dissertation would not have been possible without their numerous commitments and constructive criticism.

In addition, this dissection was made possible by funds received from the National Science Foundation under Grant No. DUE 1027409 (CyberCorps: Scholarship for Service), the Idaho State Board of Education, Idaho Global Entrepreneurial Mission (IGEM) grant program.

Table of Contents

Authorization to Submit Dissertation ii

Abstract iii

Acknowledgements iv

Table of Contents v

List of Tables xv

List of Figures xvi

Listings xviii

1 Statement of Purpose 1

1.1 Introduction 1

1.2 Type Safety 2

1.2.1 Building Blocks 2

1.2.2 Security 4

1.2.3 Type Safety 4

1.2.4 The Relationship of Type Safety and Type Strength 6

1.3 Objectives and Contribution 7

1.4 Structure of Remaining Dissertation 11

2 The C Programming Language 12

2.1 Humble Beginnings 12

2.2 Strengths 13

2.3 Pitfalls 14

2.4 Integer Error Conditions 15

2.4.0.1 Integer Overflow 18

2.4.0.2 Integer Sign Error 21

2.4.0.3 Integer Truncation Error 22

2.4.1 Casting 22

2.4.2 Integer Errors Introduced by Operators 28

2.4.3 Other Conditions Leading to Vulnerabilities 30

2.4.3.1 Undefined Behaviors 31

3 Current Error Mitigation Measures 32

3.1 Optional Compiler Warnings 32

3.2 Safe Coding Guidelines and Practices 33

3.3 Safe Integer Libraries 34

3.4 Safe Subsets of C 35

3.5 Proposed Extensions to C 37

3.5.1 Ranged Integers 37

3.5.2 Infinitely Ranged Integers 37

3.6 Source Code Analysis Tools 39

3.6.1 Runtime Analysis 39

3.6.2 Static Analysis 40

4 Introduction to Language Formalism 42

4.1 Elements of Language: Syntax and Semantics 42

4.2 Common Methods to Formalize Language Semantics 43

4.2.1 Axiomatic Semantics 43

4.2.2 Denotational Semantics 44

4.2.3 Operational Semantics 47

4.2.3.1 Small-step Semantics 47

4.2.3.2 Big-step Semantics 49

4.3 Approaches to the Formalization of Language Semantics 50

5 Historic Attempts to Formalize C 51

5.1 Evolving Algebras 51

5.2 An Abstract Dynamic Semantics for C 52

5.3 An Operational and Denotational Typing Semantics for C 53

5.4 A Formalism of C++ 54

5.5 In Summary 54

6 C Type Systems 56

6.1 Introduction 56

6.2 Syntax of C Types 57

6.3 Object Types 59

6.3.1 Scalar Types 59

6.3.1.1 Arithmetic Types 62

6.3.1.2 `char` Type 64

6.3.1.3 Extended Integer Types 64

6.3.1.4 Enumerated Types 64

6.3.1.5 Floating Types 65

6.3.1.6 Pointer Type 66

6.3.2 Aggregate Types 66

6.3.2.1 Array Type 67

6.3.2.2 `struct` Type 67

6.4 `union` Type 68

6.5 Function Types 68

6.5.1 Function Specifiers 68

6.6 Incomplete Types 69

6.7 Type Qualifiers and Storage Class Specifiers 69

6.7.1 Type Qualifiers 70
6.7.2 Storage Class Specifiers 71
6.7.2.1 `typedef` Name 72
6.8 Compatible Type 72
6.9 Composite Type 73
6.10 lvalue and rvalue 74
7 Formalizing C Expressions and Statements 75
7.1 Introduction 75
7.2 C Type Casting Rules 76
7.2.1 Integer Conversion Rank 76
7.2.2 Integer Promotions 76
7.2.3 Usual Arithmetic Conversions 77
7.2.4 Other Conversion Rules 78
7.2.4.1 Conversion to Type `_Bool` 78
7.2.4.2 Conversions Between Signed and Unsigned Integers . . . 78
7.2.4.3 Conversions Between Real Floating and Integers 79
7.2.4.4 Conversions Between Real Floating Types 79
7.2.4.5 Conversions Between Complex Types 80
7.2.4.6 Conversions Between Real and Complex Types 80
7.2.4.7 Conversions Involving Pointers 81
7.3 Expressions 82
7.3.1 Static Typing Semantics 84
7.3.2 Type Safety Requirements 89
7.4 Statements 91
8 The Static Type Safety Analysis Tool 96
8.1 The Fundamental Question 96

8.2 A Naïve Approach 96

8.3 A Better Attack Plan 97

8.3.1 Influence of Literals 98

8.4 The Underlying Language ACL2 99

8.4.1 Computational Logic 100

8.4.2 Applicative Common Lisp 101

8.4.2.1 Numbers 101

8.4.2.2 Characters and Strings 102

8.4.2.3 Symbols 102

8.4.2.4 Cons Pairs 103

8.4.3 Reasons Behind the Use of ACL2 106

8.4.4 In Summary 107

9 Leveraging State in a Static Analysis Environment 109

9.1 A State-full Introduction 109

9.2 Application of Functions Run, Next, and Exit 110

9.3 Tracking State 111

9.3.1 Type Safety Decision Algorithms 113

9.3.2 Tracking State-wise Changes with an Annotated Look-up Table . 117

9.3.2.1 The Fields `TOKEN-ID` and `("NAME")` 119

9.3.2.2 The `(TYPE-INFORMATION)` field 120

9.3.2.3 The Type Subfield `(TYPE-SPECIFIERS)` 121

9.3.2.4 The Type Subfield `(TYPE-QUALIFIERS)` 122

9.3.2.5 The Type Subfield `(STORAGE-CLASS-SPECIFIERS)` . . . 123

9.3.3 The `(VALUE)` Field 124

9.3.3.1 A Note on Modeling Other Data Objects 125

10 Conclusions and Future Work 127

10.1 Review and Conclusions 127

10.1.1 Organization of this Dissertation 127

10.1.2 Formalization 128

10.1.3 The C Type Safety Verification Tool 129

10.2 Assumptions 130

10.3 Limitations 131

10.4 Test Suite and Observed Performance 132

10.5 Future Work 135

10.6 Final Observations 136

Bibliography 138

Appendices 153

A The Syntax of C Types 153

B Required Predicate and Valuation Functions for Expressing Static Typing Semantics 156

B.1 Populating the Lookup Table Type Specifier Field 157

B.1.1 $getDeclaredArithType(\tau)$ 157

B.2 Truth Returning Functions for Types 159

B.2.1 $isInteger(\tau)$ 159

B.2.2 $isFloat(\tau)$ 161

B.2.3 $isArithmetic(\tau)$ 162

B.2.4 $isPointer(\tau)$ 163

B.2.5 $isScalar(\tau)$ 164

B.2.6 $isArray(\tau)$ 164

B.2.7 $isStruct(\tau)$ 165

B.2.8 *isAggregate*(τ) 165

B.2.9 *isUnion*(τ) 166

B.2.10 *isObject*(τ) 166

B.2.11 *isVoid*(τ) 167

B.2.12 *isNull*(τ) 167

B.2.13 *isQualified*(q) 167

B.2.14 *isModifiable*(*identifier*) 168

B.3 Type Returning Type Functions 169

B.3.1 *intPromote*(τ) 169

B.3.2 *arithConv*($\tau_{\mathbf{1}}, \tau_{\mathbf{2}}$) 174

B.3.3 *funcArgPromote*($\tau_{\mathbf{1}}, \ldots, \tau_{\mathbf{n}}$) 174

B.4 Truth Returning Functions for Literals 175

B.4.1 *isDecimal*(*lit*) 175

B.4.2 *prefix*(*lit*) 176

B.4.3 *isOctal*(*lit*) 177

B.4.4 *isHexadecimal*(*lit*) 177

B.4.5 *isChar*(*lit*) 178

B.4.6 *isWideChar*(*lit*) 179

B.4.7 *isStringLit*(*lit*) 179

B.4.8 *isWideString*(*lit*) 180

B.5 Type Returning Literal Functions 180

B.5.1 *suffix*(*lit*) 180

B.5.2 *firstToRepresent*(*lit*, $\tau_{\mathbf{1}}$, ..., $\tau_{\mathbf{n}}$) 181

B.6 Value Returning Literal Functions 182

B.6.1 *lengthOfString*(*string*) 182

B.6.2 *charToDecimalVal*(*lit*) 183

B.6.3 *octToDecVal*(*lit*) 185

B.6.4 *hexToDecVal*(*lit*) 185

B.7 Value to Type Range Functions 186

B.7.1 *isValidIntValue*(*val*,$\tau_{\mathbf{1}}, \tau_{\mathbf{2}}$) 186

B.7.2 *isValidRealValue*(*val*,$\tau_{\mathbf{1}}, \tau_{\mathbf{2}}$) 187

B.7.3 *isValidArrayIndex*(*index*,*size*) 187

List of Tables

1.1 System criticality levels based on consequence of failure. 3

2.1 C operators and their potential to produce overflow 29

2.2 C operators and their potential to produce a wrap 30

3.1 MISRA recommended `typedef`ed arithmetic type names for 32-bit platforms 34

3.2 Critical undefined behaviors. 38

5.1 Abstract syntax of the C++ type system [108] 55

7.1 Operator order of precedence and operand associativity 83

9.1 High level instruction nodes 111

9.2 Sub-expressions of `EXSTMT` node 116

List of Figures

2.1 von Neumann architecture 13
2.2 Number of integer errors generating CVE OS advisories 2002-11 16
2.3 Distribution of CVE OS integer error types 2002-11 17
2.4 Distribution of CVE OS integer error types 2012 18
2.5 Yearly exploit distribution resulting from integer error 2002-11 19
2.6 2012 individual OS advisory count due to integer error 20
2.7 Illustrative silent wrapping of 4-bit *unsigned-integer-type* [116], the mechanics are similar for larger unsigned integer types such as the 32-bit . . 21

6.1 The universe of C types 58
6.2 C identifier syntax 58
6.3 `keywords` 58
6.4 C declaration syntax 60
6.5 Type-specifier `keywords` 61
6.6 Atomic-type-specifier syntax 61
6.7 Object-type 61
6.8 Scalar-type 61
6.9 Arithmetic-type 62
6.10 Enum-specifier syntax 65
6.11 Pointer-type 66
6.12 Aggregate-type 66
6.13 Struct-or-union-specifier syntax 67
6.14 Union-type 68
6.15 Function-type 68
6.16 Function-specifier syntax 69
6.17 Incomplete-type 69

6.18 Type-qualifiers 71

6.19 Storage-class-specifiers 71

6.20 Typedef-name syntax 72

7.1 Syntax of C expressions 84

7.2 Syntax of C statements 92

8.1 The 32-bit patterns for 10 and -10 in two's complement 99

9.1 A simplistic AST 112

9.2 Basic precondition algorithm 114

9.3 Basic postcondition algorithm 117

9.4 C type-specifier `keywords`. 122

9.5 C type-qualifier `keywords`. 123

9.6 C storage-class-specifier `keywords`. 123

Listings

2.1 Signed integer overflow example . . . 21

2.2 Unsigned integer overflow example . . . 22

2.3 Intended integer overflow in the function `htons` . . . 22

2.4 Integer overflow is reversed *a posteriori* . . . 23

2.5 Integer overflow is checked *a posteriori* . . . 23

2.6 Integer sign error example . . . 24

2.7 Integer truncation error example . . . 24

2.8 When $90 + 40 = -126$ . . . 29

3.1 Typing errors missed by `gcc` . . . 33

3.2 A safe multiplication function call . . . 35

3.3 SafeMult function checking result value to value range of return type . . . 35

3.4 SafeMult function checking operands for potential overflow . . . 41

6.1 `const` Declaration . . . 70

6.2 `typedef` declaration . . . 72

8.1 When $10 \not> -10$ . . . 99

8.2 ACL2 `not` function . . . 104

8.3 ACL2 `member` function . . . 105

8.4 An iterative solution in C for `n!` . . . 105

8.5 A recursive solution in C for `n!` . . . 106

8.6 A secursive solution in ACL2 for `n!` . . . 106

8.7 A simple C program that adds operands of mixed integer type . . . 107

8.8 AST for a simple C program that adds operands of mixed integer types . 108

9.1 Annotated fields for each basic data object in the lookup table . . . 118

10.1 Simple C program that adds operands of mixed integer type . . . 133

10.2 AST for simple C program that adds operands of mixed integer type . . 134

10.3 Example analysis output for simple C program that adds operands of mixed integer type . . . 137

B.1 The four tuple of the `c2acl2` generated lookup table for each data object 158

B.2 ACL2 isInteger . . . 160

B.3 ACL2 isAritmeticType . . . 163

B.4 ACL2 isPointerType . . . 164

B.5 ACL2 get-type-rank . . . 171

B.6 ACL2 get-type-rank-list . . . 172

B.7 ACL2 get-max-type-rank . . . 172

B.8 ACL2 get-dominate-type . . . 172

B.9 ACL2 get-type-from-type-rank . . . 173

B.10 ACL2 example usage of `RATIONALP` to find a decimal numeric literal . . . 176

B.11 ACL2 conditional identifying character literals to get ASCII decimal value 179

B.12 ACL2 get-ascii-dec-val . . . 184

Chapter 1: Statement of Purpose

1.1 Introduction

Without a doubt, computational technology has become an ubiquitous component in the fabric of modern Western life. So much so, that at the dawn of the 21st century, Kurzweil declared [69] that singularity would soon be realized, as mankind would merge with machines (computers included) to achieve not only super human strength, but undreamed of cognitive prowess as well. In part, Kurzweil's conjecture was based on the fact that computer technology has adhered to the precepts of Moore's Law [83] by becoming increasingly smaller, faster, and smarter while the code driving these devices has become ever larger and complex. This phenomena was accomplished by a periodic doubling of computational circuitry. On average, the cycle has been every eighteen months, leaving many "*state of the art*" technologies obsolete shortly after their debut. As evidence, just consider how much more computational power is embedded in today's smart phones than that of the room sized behemoths driving the technology behind mankind's successful lunar conquest.

Others, such as Garreau [36], have expressed reservations about the seemingly rapidly approaching singularity. Among their concerns are the questions of the impact of computer failure. For example, many of the systems steering the technological revolution we continue to witness have been programmed with the programming language C. But, programs written in C are prone to unexpected behaviors from calculations returning unexpected results.

Even if an unexpected result does not produce an unexpected behavior, it does have adverse implications on the state of a system's health, such as safety, security, and even survivability. According to Cardelli, a program that is free of unexpected results is said to have the property of type soundness [17]. Likewise, Milner [76] claims

> "*a well typed program is semantically free of type violations.*"

Both Cardelli and Milner are describing the computer program property of **type safety**.

1.2 Type Safety

Central to this dissertation is the concept of type safety. But, what is the meaning of type safety? Upon examination, the terms security and safety appear to be interchangeable. A quick search in any leading dictionary (*e.g.,* [103]) may provide the following definitions for safety and security.

> **Safety** - *the state of being safe from the risk of experiencing or causing injury, danger, or loss.*
>
> **Security** - *the freedom from danger, risk, etc.: safety.*

As a result, the two terms are often used interchangeably creating an even greater disconnect in the lexicon. But in the computer and technology realm, the two terms have very disjunct and distinct meanings.

- *security* protects computers (technology) from the outside world and
- *safety* protects the outside world from computers (technology).

1.2.1 Building Blocks

It would be safe to say that modern society has become dependent on computers and technology in general without fully realizing the extent of its dependency. Technology oversees, if not manages, everything from mundane tasks to critical security-safety systems. Technology's ubiquity remains largely unnoticed until something in the technology fails. As noted by Hatton [43], a least one fault is required for a failure to occur. It is possible that many faults may never lead to failure.

Definition 1.1. A *failure* is a dynamic system property that includes any difference between actual and expected behavior.

Table 1.1: System criticality levels based on consequence of failure.

Criticality Level	**Consequence of Failure**
Security and safety critical	Reduced National defenses Disruption of critical infrastructure
Safety critical	Injury and/or loss of life
Security critical	Compromised proprietary information leading to future business loss Possible fines by regulators Compromised classified government information leading to national security threat
Business critical	Loss of customer trust and customers Disruption of supply chain Fines, license revocation, cessation of business
Non-Critical	Minimal impact limited to users May affect reputation of system supplier

Definition 1.2. A *fault* is a static system property that is a point of potential failure.

A fault can be regarded as a vulnerability. We use the following definition provided by Seacord [116].

Definition 1.3. A *vulnerability* is a set of conditions allowing any violation of an explicit or implicit security policy.

When computers fail, the consequences of the failure can run the gamut between a mere inconvenience and a major catastrophe. By catastrophic, the implication is that physical harm including death to humans is likely, untold financial loss or environmental damage is possible, our national security compromised, etc. In any case, reputations of the companies who either supply or use the technology are at stake.

The extent of the damage caused by failure depends on the critical nature of the failed system (Table 1.1). For example, critical systems are those systems that manage our national defense and the national infrastructures such as energy, finance, and transportation. Critical systems may be classified as security critical, safety critical, and safety-security critical.

1.2.2 Security

Information security and assurance is generally presented in terms of the CIA triad whose components include confidentiality, integrity, and availability. According to Bishop [11, 12], confidentiality offers concealment; integrity guarantees and indemnifies the trustworthiness of both data and users; and, availability provides continuous authorized on-demand access.

Definition 1.4. *Confidentiality* is the system property that hides user data and processes.

Let X be a set of entities and let I be some data. I has the property of confidentiality with respect to X if no member X can obtain information about I.

Definition 1.5. *Integrity* is a twofold system property as it validates the correctness of data and processes in addition to validating user authorization to data and processes.

Let X be a set of entities and let I be some data or resource. I has the property of integrity with respect to X if all members X trust I.

Definition 1.6. *Availability* is a system property that insures timely, uninterrupted authorized access to data and processes.

Let X be a set of entities and let I be a resource. I has the property of availability with respect to X if all members X can access I.

Security is realized through a combination of policy and mechanism that target one or more goals of the CIA triad.

1.2.3 Type Safety

The bulk of the available literature about type safety, *e.g.,* Wright and Felleisen [127], Milner [76] and Mitchell [80], define type safety in terms of two main program properties: preservation and progress. Preservation generally means that data types persist during

program execution. Formally, preservation is defined as

$$\text{if} \vdash e : \tau \text{ and } e \to e', \text{ then } \vdash e' : \tau; \tag{1.1}$$

and is read as "if there is an expression e of a certain type τ and e transitions to its next state e', then there is an implication that e' is also of same type". If preservation holds, one can assert that the value v of expression e of type τ also has type τ. That is,

$$\text{if} \vdash e : \tau \text{ and } e \to v \text{ then, } \vdash v : \tau. \tag{1.2}$$

Progress is said to show a relationship between the initial expression e and its final value v through every step of a computation. Progress implies that a well typed program never enters a stuck state; it either moves on to the next state or is in its expected terminal state. Formally, progress is expressed as

$$\text{if} \vdash e : \tau, \text{ then either (i) } e \to e' \text{ for some } e' \text{ or (ii) } e \text{ is a terminal value.} \tag{1.3}$$

According to Pfenning [98], preservation and progress together are called type safety. However, there are programming languages, generally considered not to be type safe, that demonstrate both preservation and progress. For example, one can argue that the language C preserves data types with its casting rules such as the integer promotions and the usual arithmetic conversions [52, 53, 54]. Likewise, most systems programmed in C are assumed to demonstrate progress once all of the faulty logic leading to a "*segmentation fault*" or other failures have been flushed out during the development cycle.

Without diminishing the importance of what preservation and progress give to a program's property of type safety, we expand the notion of both properties by incorporating the purpose of a type system as presented by Cardelli [17] to say type safety is a requirement that a program exhibits no unexpected behaviors. By doing so, one can show that

programs written in non type safe languages such as C can have the property of being type safe.

1.2.4 The Relationship of Type Safety and Type Strength

There is a correlation between the type safety of a program and the type strength of the language used to write the program where as the type strength increases, so too does type safety. At times, type strength has been described by the popular anonymous anecdote:

> "*Pascal keeps your hands tied while C gives you enough rope to hang yourself.*"

To be sure, type strength is a characteristic of a programming language with respect to its usage of mixed (disjoint) data types. Strongly typed languages such as Standard ML [77], Haskell [56], and Objective Caml [44], prohibit the use of mixed data types within an expression. A weakly typed language, such as C, places no restrictions on the use of mixed data types. The process used to detect typing errors is known as type-checking.

According to Liskov and Wing, type-checking can be relied upon to capture only a small part of a program's overall correctness [72]. For example, in between the strongly and the weakly typed languages are the nearly strongly typed languages, such as Ada [55]. Like strongly typed languages, nearly strongly typed languages also prohibit using mixed data types. However, they offer a collection of libraries containing special functions that allow programmers to use mixed types when needed. Once a programmer exercises a library to relax type-checking, the onus of insuring correct program behavior falls squarely on the programmer. As Lyons has noted [73],

> "*one can write bad software in any language; it is harder to do so with the nearly strongly typed language Ada than it is with the weakly typed languages such as C*"

Ada was designed to meet the needs of the U.S. Department of Defense (DoD) and, perhaps, went through the most rigorous development process of any programming lan-

guage because of the security/safety critical nature of DoD systems. Although developed for the DoD, Ada has been used in the coding of several critical systems outside DoD auspices. For example, the European Space Agency (ESA) relied on the type strength of Ada in the flight controllers of its Ariane rocket series. Nevertheless, the practice of code reuse was attributed to a type safety violation contributing to the maiden flight of the Ariane 5 abruptly ending in explosion [71]. The suspected culprit in the $5B loss was traced to the use of a 32-bit integer variable that was allowed to accept a 64-bit floating point value before it could be properly initialized. This code worked fine for the Ariane 4 rocket, which used a slower processor. Thus, the primary goal of type safety enforcement is to guarantee that a program remains well typed despite the underlying programming language type strength and according to Milner, [76]

> "*a well typed program is semantically free of type violations.* "

1.3 Objectives and Contribution

This dissertation is a study of the type safety problem inherent with the programming language C. It is the cumulation of research to identify the root causes of the problem and to offer a viable solution. The solution, however, could not be obtainable without first having a thorough and unambiguous understanding of C. This was achieved by formalizing not only static typing semantics for all C expressions and statements, but that of the type safety requirements for each C expression and statement as well. The formalizations were then incorporated into a prototype static analysis tool to verify that a program written in C source code has the property of being type safe [66]. The tool is the proposed solution, because it has the ability to prove the correctness of its reasoning about a program. In other words, if the tool can show that a program is without error, potential or real, then the program is considered to be generally type safe.

Admittedly, the problem of type safety is nothing new. With respect to C, the problem has been around since the inception of C. Because C is the "*de facto*" systems programming language, its type safety problems have been widely studied. Several solutions have been proposed over the years. However, the problem is both hard and persistent. It has been said [20, 48] that "*bug*" finding, using static analysis techniques, is non-trivial and as such, is equivalent to Rice's Theorem that states all non-trivial questions asked about a program eventually reduce to the "*halting problem*" [109]. A definitive solution to the C type safety problem seems to be unlikely because the problem appears to be undecidable. Even Dijkstra said [27],

> "*program testing can be used to show the presence of bugs, but to never show their absence!*"

At best, most tools endorsed by National Institute of Standards and Technology (NIST) Software Assurance Metrics and Tool Evaluation (SAMATE) project [89] are merely designed to catch a limited subset of all program code bugs potentially lurking in the wild. Since each tool catches its own unique bug subset, there is often a disconnect in truly understanding or tackling the type safety problem in C programs.

Complicating matters is the fact that the C language continues to evolve in an effort to adapt to the advances in underlying hardware of the architectures it supports. For example, the newest standard, C11 was released in late 2011 and adds language support for parallel processing [54]. Since every newer standard supersedes all previous standards, proposed solutions based on prior standards often become inadequate. For instance, many of the prior attempts to formalize either the operational semantics and static typing semantics of C [42, 43, 95], were applied to C90 [52] or it predecessor ANSI C [6]. C99 [53] introduced support for 64-bit architectures whose native integer size increased to 32-bits of precision from the 16-bit precision integer common to 32-bit platforms C90 supported. The same can be said about the code analysis tools based on

the aforementioned formalizations or the other means attempting to ensure type safety in C programs.

The research of this dissertation is based on C99 instead of C11. The reason is that C11 was released well after this project was underway. However, the changes introduced in C11 have no direct bearing on goals or outcomes of this project. Except for adding a new type specifier and a type qualifier with the keyword `_Atomic`, the typing syntax remains largely unchanged. In fact, the `_Atomic` types are only applicable to those platforms designed to support such types. Likewise, the syntax, semantics, and typing constraints for expressions and statements remain unchanged. Therefore, the underlying reasoning about C type safety in this research can be readily applied to C11.

What is unique about this dissertation is the approach taken. For example, the formalization includes a formal presentation of the syntax of C types not seen before in the literature. The typing syntax presented fully depicts the relational hierarchy among all C type categories and is useful in understanding what is meant when the standard imposes certain typing constraints on the operands of an expression. In addition, a relational syntax of C types is crucial in understanding the often "*quirky and flawed*" [110] and "*unintuitive*" [15] casting rules defined in the standard [52, 53, 54].

The formalism of the static typing semantics in this dissertation is similar to the work of Papspyrou [95]. For simplicity, however, it is presented in a higher level of abstraction without diving too deeply into category theory [99, 38, 100, 7, 9] and the use of monads [9, 82]. The formalism presented in this document also makes provisions for 64-bit data types as opposed to Papspyrou's work on a 32-bit standard. As mentioned earlier, the inference rules defining the semantics of the type safe requirements for each expression and statement are also presented.

The prototype static type safety analysis tool is unique for several reasons. First, it is coded in ACL2 [58]. Other tools have been written in a diverse collection of languages, such as, but not limited to OCaml [44] for Astrée [24] and Csur [96]; C for UNO [47, 48]

and Smatch [64]; and metal for `xg++` [29]. Unlike the other programming languages, ACL2 provides the ability to write the executables required to analyze the code and the ability to write proofs to support and verify the reasoning the tool made about the analyzed source code. ACL2 has been successfully used to verify a variety of projects such as hardware [106, 104, 111], operational semantics [85, 86], and secure data flow [51, **?**]. But, the use of ACL2 to verify type safety of source code written any programming language, let alone C, has yet to be documented in the published literature.

Similar to the commercially successful static analysis tool, Astrée [24], that is currently used by AIRBUS [26] to verify its flight control systems, the tool introduced in this document is applied to an abstract interpretation of the source code. The difference between the two is that Astrée uses a project driven selection of abstract domains out of the dozens it has on hand and combines the domains through reduction processes and user defined assertions. On the other hand, the tool introduced in this document traverses a simple abstract syntax tree (AST) of the source code and applies one or more predicate functions at each tree node.

Other tools may also use some form of code abstractions similar to an AST; but, they employ techniques such as `grep` like utilities on regular expressions (Yasca [115]), simple pattern matching to a repository of known bugs and vulnerabilities previously stored in their databases (Flawfinder [126] and Rats (rough auditing tool for security) [117]), or annotated code (CQual [33, 34], LCLint [30, 31], and UNO [47, 48]). While not endorsed by SAMATE, the Berkeley Lazy Abstraction Software Verification Tool (BLAST) uses a "counterexample" driven automatic refinement in the construction its abstract model of the code it analyzes [28, 10].

While not complete, the prototype tool introduced within this document provides a large enough foundation from which an all encompassing static C type safety analysis tool can be perfected, if not commercialized, and used to verify safety critical applications.

1.4 Structure of Remaining Dissertation

The structure of this dissertation largely follows the general methodology briefly described in Section 1.2 and is divided into four main sections. Transitions into each section should be readily recognized as the first few chapters define the problem space and the motivation of this research. Chapters dedicated to the formalization of the C language follow and precede the chapters on the prototype tool design and its preliminary results. Finally, the dissertation ends with a chapter devoted to lessons learned, remaining work, and conclusions.

Chapter 2: The C Programming Language

Stroustrup [102] once said,

> "*C makes it easy to shoot yourself in the foot.*"

This conjecture about C is supported by the fact that the language is not type safe in the sense that programs written in C have historically produced unexpected results. This chapter explores the why C continues to be a favorite systems programming language despite its shortcomings.

2.1 Humble Beginnings

The C programming language was developed as a systems programming language at the Bell Telephone Laboratories (Bell Laboratories) between the years 1969 and 1973. The principal investigator, Ritchie [110] affirmed that C's development was closely tied to the concurrent development of the Unix operating system [123]. Prior to C, most operating system kernels were written in low level assembly languages. However, C introduced the `struct` data type which, among its other language features, made the language expressive enough for systems programming. In time, the Unix kernel became the first operating system to be coded mainly in C. In addition to `struct`, other innovative features C offered included:

- a standard input/output (I/O) library
- the 32-bit `long int` data type
- a collection of `unsigned` integer types
- several compound assignment operators in the form of `op=` where `op` is one of several other distinct unary or binary operators, *e.g.,* `-=` or `+=`

The successful Unix launch not only helped bring C into existence, it generated considerable interest and increased usage of the language. However, several years passed before a standardized usage of the language would be agreed upon, leaving many system

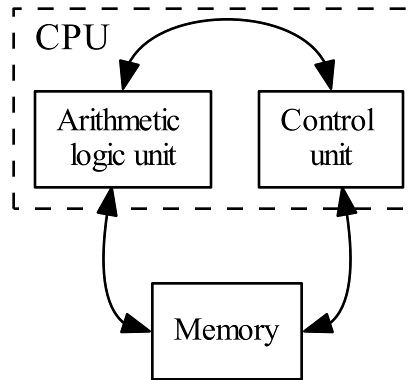

Figure 2.1: von Neumann architecture

programmers during that time frustrated after writing unstable, unworkable code. That all changed in the 1980s when Kernighan joined Ritchie to publish the first edition of the *The C Programming Language* which served as the first "official *informal*" specification of the language known as "K&R C [65]."

The first "*formal*" specification of C, known as ANSI C, was released in 1989 by the American National Standards Institute (ANSI). The ANSI specification was in turn, ratified by the International Organization of Standardization (ISO) in 1990 and became known as C90 [52]. C90 was superseded with another ISO standardization, C99 in 1999 [53] which in turn, was outdated with the 2011 ISO standardization of C11 [54].

2.2 Strengths

The popularity of the Unix kernel, and the subsequent language standardizations positioned C to be quickly anointed as the "*de facto*" systems programming language. C continues to hold that prominence today and C's lasting popularity can be attributed to several factors.

Most computational hardware based on the von Neumann [125] architecture operates within an imperative execution paradigm where computations are spelled out. This is necessary by design as both data and the instructions used for manipulating the data share the same hardware (*i.e.,* registers, data buses, memory, etc., see Fig. 2.1).

C is not only an imperative language, it makes provisions for most machine integer types. Programs written in C tend to be high level abstractions of the machine hardware. Thus, C programs are highly portable across many architectural platforms. Once compiled for a specific architecture, the program usually maintains a fast execution that is relatively equal to the execution times when the program is compiled on other platforms. Finally, C provides a rich set of data types in addition to its integer types and operators. Its collection of low level bitwise operations enable programmers to "*feel the bits*" while the high level operations allows for the computation of complex problems. C has a liberal usage syntax of its operators and operands making it highly expressible. As such, programmers are seemingly unlimited in writing expressions containing multiple operators of both levels. Furthermore, programmers are free to mix operand types. If a programmer breaks a syntactic rule with respect to mixing operators and operands in a single expression, the compiler will generate an error.

2.3 Pitfalls

The strengths of the C programming language were obtained at the expense of C's notorious lack of bounds checking. In 1996, the seminal paper "*Smashing the stack for fun and profit*" demonstrated how a lack of bounds checking gave attackers a relatively easy inroad to take control of a target system written in C with simple buffer overflow techniques [2]. Although the problem of buffer overflow had been widely documented and researched since well before 1996, the problem has persisted, and remains every bit as serious today. For example, Christey and Martin [21] noted in 2007 that buffer overflow kept its distinction as the leading cause of operating system (OS) vendor advisories in the 2006 Common Vulnerabilities and Exposures (CVE) list maintained by the MITRE Corporation [81].

2.4 Integer Error Conditions

C's lack of bounds checking is not limited to just buffers. The same Christey and Martin report cited integer overflow errors as the second leading cause of OS vendor advisories. In the opinion of Brumley *et al.*[15], the known integer bugs reported by the CVE between 1999 and 2006 merely represent the "*tip of the iceberg*" as many, many more remain undiscovered. Meanwhile, Seacord contends that the C integer errors are the most overlooked and the least understood memory errors found in C programs [116]. On the other hand, fractional numeric types (of the *real floating types*) in C are supported by an extensive runtime check library and are generally not as prone to error conditions.

Prior to 2002, reported integer errors resulting in a CVE OS advisory was rare. Since the Christey and Martin report, however, there has been a marked increase in the number of integer errors reported as the root cause for a CVE OS vendor advisory (Fig. 2.2) and integer overflow remains to the leading integer bug cause (Fig. 2.3 and Fig. 2.4).

The CVE advisories were issued because the integer errors made the systems vulnerable to execution of arbitrary code (EAC), denial of service (DoS), or escalation of privilege (EoP) attacks (Fig. 2.5). As fas as operating systems are concerned, it appears that all are equally at risk from suffering one of these vulnerabilities resulting from an integer bug (Fig. 2.6).

Definition 2.1. *Execution of arbitrary code* is the ability of an attacker to execute any commands on a target machine or target process.

Most EAC attacks are achieved through a vulnerability that allows an attacker to inject and execute code commonly called shellcode. The name shellcode comes from the fact that the injected code often opens up a command shell for the attacker to gain control of the victim machine. Typically, shellcode is written in machine code and the attacks may take place locally or remotely. Both local and remote attacks can exploit buffer overflows while remote attacks usually take place over TCP/IP (Transmission Control

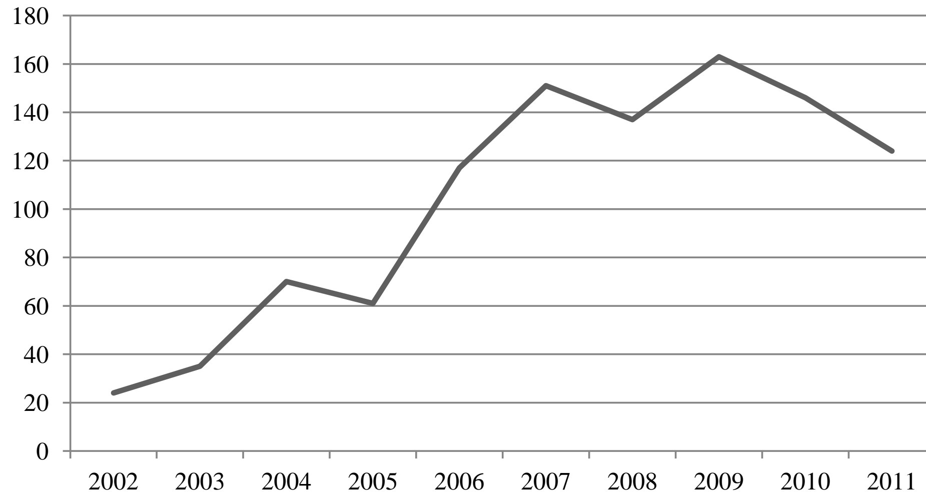

Figure 2.2: Number of integer errors generating CVE OS advisories 2002-11

Protocol and Internet Protocol respectively) socket connections. When the attacks take place over the Internet, the attack is commonly known as *remote code execution.* A sampling of the attack (injection) avenues opened up by the integer errors reported in 2012 include altered .jpg, .bmp, or .png images, a heap-based buffer overflow triggers, altered MPEG video or QTVR movie files, and an altered True Type font.

Definition 2.2. *Denial of Service* is an attempt to make a machine or network resource unavailable to its intended users.

A few of the DoS vulnerabilities exploited by the integer errors reported in the 2012 CVE include kernel panic, application crash, memory corruption, segmentation fault, and infinite loops. There are five basic means to conduct a DoS attack:

1. Consumption of resources.
2. Disruption of configuration information.
3. Disruption of state information.
4. Disruption of physical network components.
5. Obstruction of communication between user and victim.

A distributed denial of service (DDoS) is the use of many attack systems against one victim system.

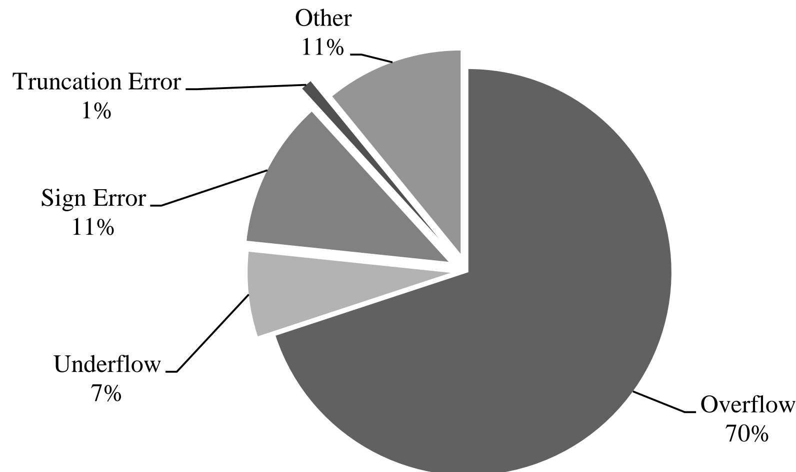

Figure 2.3: Distribution of CVE OS integer error types 2002-11

Definition 2.3. *Escalation of privilege* is when a user acquires privileges not normally allowed to the user.

An escalation of privilege attack may be vertical or horizontal. A *vertical* EoP is when a lower privileged user or application gains higher privileges. Whereas, a *horizontal* EoP is when a user or application gains access to another user's private information and the victim user has the same level of privileges as the attacker.

Because C belongs to the imperative programming paradigm, all variables (memory stores) and their types, including the integers, must be declared prior to usage. The amount of memory reserved for each variable is based on its numeric type. The current C standard provides support for eight machine integer types including the integer type native to the underlying architecture. Each machine integer type is unique from the others by the constraints imposed by its bitwise precision and the interpretation of its high order bit. Machine integers are either signed or unsigned. Signed integer values maybe either positive or negative and use the high order bit to represent the $+$ or $-$ sign. Since unsigned integers are strictly positive (0^+), their high order bit is used as a precision bit. Because of these bitwise constraints, all C integer types are vulnerable

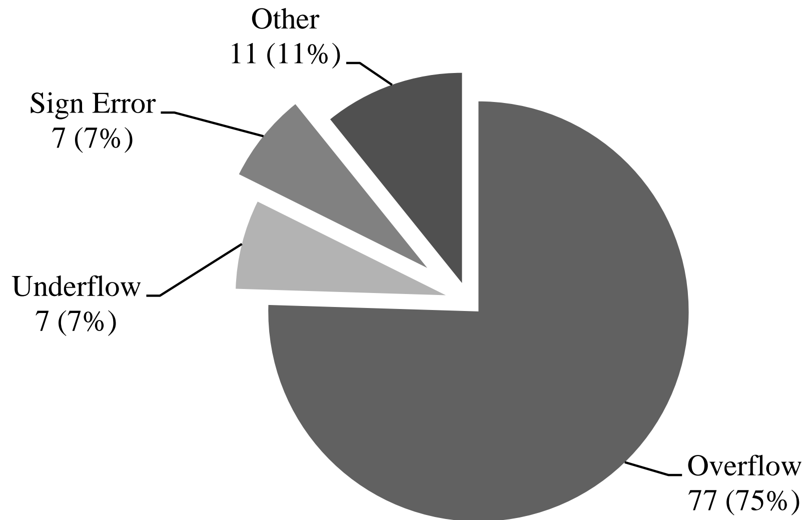

Figure 2.4: Distribution of CVE OS integer error types 2012

to enter at least one of three error conditions: overflow/underflow error, sign error, and truncation error.

2.4.0.1 Integer Overflow

Definition 2.4. An *integer overflow error* occurs whenever the value of an integer is increased beyond the limit of its maximum value. An integer underflow error occurs whenever the value is decreased beyond the limit of its minimum value. According to the standard, operations on unsigned-integer-type operands can never overflow. Instead, the value of a result that cannot be represented by the resulting unsigned-integer-type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type (Fig. 2.7 [116]). However, Keaton *et al.*[63], claim that many programmers are prone to mistakenly assume wrapping is a well defined behavior, while not fully understanding its implications.

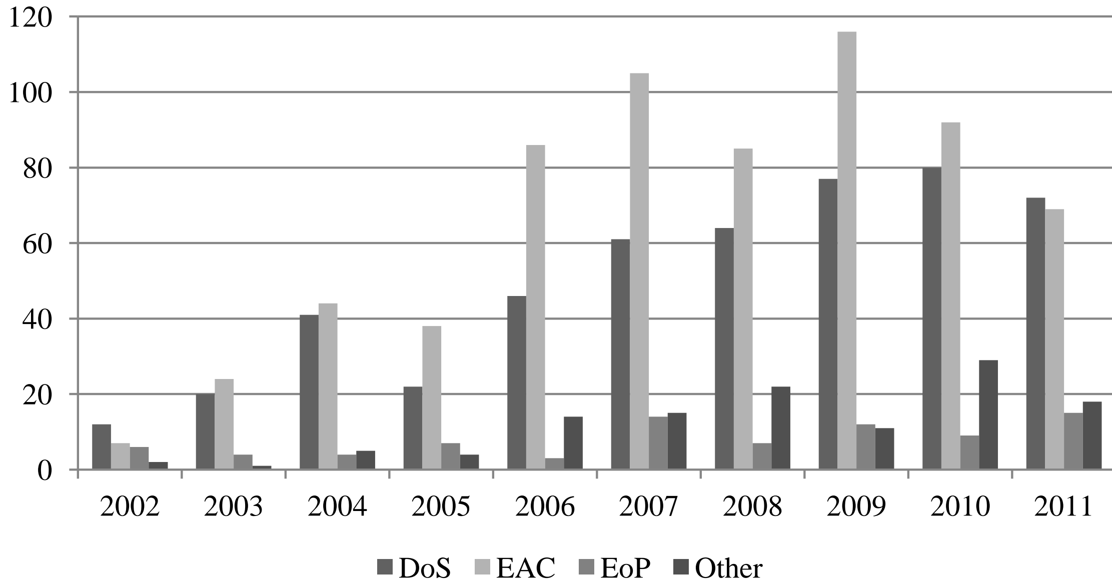

Figure 2.5: Yearly exploit distribution resulting from integer error 2002-11

However, the behavior of the signed-integer-type overflow is that the behavior is left undefined by the standard. According to §3.4.3 of the standard, any of the following reactions to program behavior are applicable:

> "*ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a document manner characteristic of the environment* (*with or without the issuance of a diagnostic message*), *to terminating a translation or execution* (*with the issuance of a diagnostic message*)[53]."

Overflow/underflow may be signed (Listing 2.1) or unsigned (Listing 2.2). Signed overflows occur when a value is carried over to the sign bit. Unsigned overflows occur when the underlying representation can no longer represent a value.

There are times, when overflow conditions are intentionally created. Moy *et al.* [88] cite three such cases:

1. The overflow is intentionally caused with an explicit cast such as in the function `htons` (Listing 2.3) which is used to convert an `unsigned short int` from host

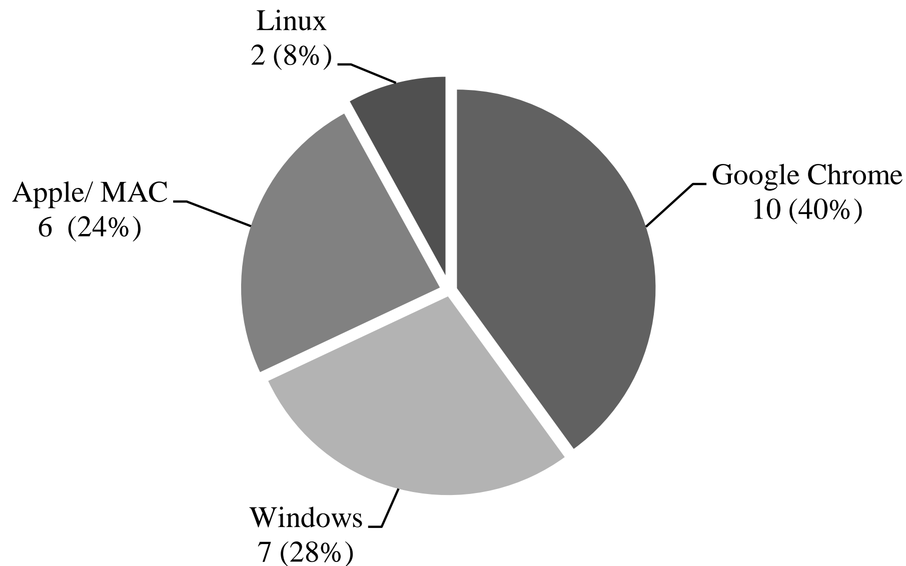

Figure 2.6: 2012 individual OS advisory count due to integer error

byte order to a big-endian network byte order. The big-endian format stores the numbers as one typically writes Arabic numbers where the first byte is the most significant or the highest magnitude number and the last byte is the least significant number. A little-endian byte order is the reverse of big-endian.

2. The overflow is reversed a posteriori. These are generally associated with loop increments and decrements. For example, when an element is removed from vector `vect`, index `i` is decremented. If the element at index 0 is removed, then the unsigned integer `i` overflows to the largest value `i` can represent. The subsequent increment removes the overflow condition (Listing 2.4).

3. The overflow is checked a posteriori. Many dedicated functions designed to execute safe operations execute a posteriori checks. For example, the unsigned addition function `safe_uadd` performs the test `r < a` after the operation `r = a + b` to detect if an overflow condition has occurred (Listing 2.5).

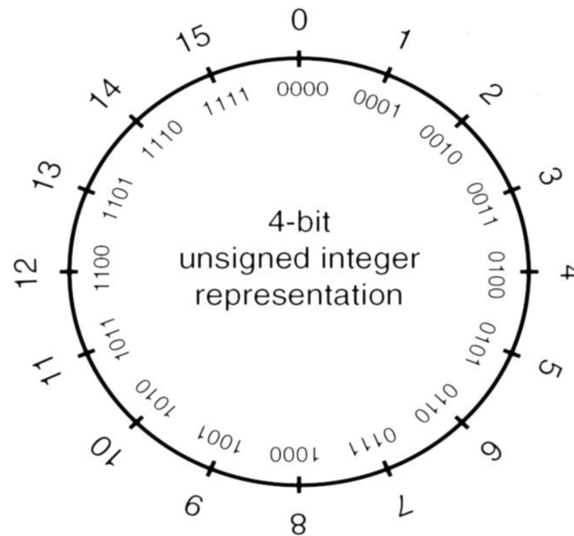

Figure 2.7: Illustrative silent wrapping of 4-bit *unsigned-integer-type* [116], the mechanics are similar for larger unsigned integer types such as the 32-bit

Listing 2.1: Signed integer overflow example

```
int i;
int INT_MAX = 2147483647;
int INT_MIN = -2147483648;

i = INT_MAX;            // i = 2,147,483,647
i++;
printf("i %d\n",i);    // i = -2,147,483,648

i = INT_MIN;            // i = -2,147,483,648
i--;
printf("i = %d\n",i); // i = 2,147,483,647
```

2.4.0.2 Integer Sign Error

Definition 2.5. An *integer sign error* occurs when there is casting between signed and unsigned integer types that changes the meaning of the high order bit from a sign bit to a precision bit or vice-versa (Listing 2.2).

Listing 2.2: Unsigned integer overflow example

```
unsigned int j;
unsigned int UINT_MAX = 4294967295;

j = UINT_MAX;           // 4,294,967,295
j++;
printf("j = %u\n", j);  // j = 0

j = 0;
j--;
printf("j = %u\n", j);  // j = 4,294,967,295
```

Listing 2.3: Intended integer overflow in the function `htons`

```
uint16 htons(uint16 a)
{
  uint x = a << 8;
  uint y = a >> 8;
  return (uint16)(x / y);
}
```

2.4.0.3 Integer Truncation Error

Definition 2.6. An *integer truncation error* occurs when an integer is being converted to a smaller integer type, holds a value larger than the representable value range of the new type, and the cast discards the high order bits of the original value during the down cast (Listing 2.7).

As a truncation error example, the `unsigned short int` is 16-bits wide and the `unsigned char` is 8-bits wide. If an `unsigned short` holding the maximum value (`1111 1111 1111 1111`) for its type is to be converted to an `unsigned char`, the new value after the conversion will be the maximum value of the `unsigned char` type (`1111 1111`).

2.4.1 Casting

A casting is an operation performed to "find a common type" for an expression. The cast operation may be either explicitly defined by the programmer or implicitly defined by

Listing 2.4: Integer overflow is reversed *a posteriori*

```
uint i = 0;
while(i < max)
{
  if(sel(vect[i]))
  {
    vect.remove(i);
    --i;
  }
  i++;
}
```

Listing 2.5: Integer overflow is checked *a posteriori*

```
uint safe_uadd(uint a, uint b)
{
  uint r = a + b;
  if(r < a)
    error();
  return r;
}
```

the compiler. The later is sometimes called a coercion. While most C casts are implicit, all casts have been attributed to be the leading cause of integer errors [116, 15].

Definition 2.7. A *cast* is the conversion of one data type to another. Casting in C is mainly governed by the rules integer conversion rank (§6.3.1.1), integer promotions (§6.3.1.1), and usual arithmetic conversions (§6.3.1.8).

Definition 2.8. An *integer conversion rank* is the basis of the ranking system employed by C to govern integer casting and is based on the bitwise precision of the types standard integer and extended integer where as the precision increases, so too does the rank.

Definition 2.9. The *integer promotions* rule defines the automatic coercion of any expression operand of integer type with a rank less than the rank of type `int` to the integer type `int`. The intent of the integer promotions rule is to preserve signage as well as value. Therefore, if the type `int` can fully represent the value of lessor integer type, then the

Listing 2.6: Integer sign error example

```
int i = -3;
unsigned short u;

u = i;
printf("u = %hu\n", u); // u = 65533
```

Listing 2.7: Integer truncation error example

```
long long int li = 214783647; // llong maximum value
short int si;                 // short maximum value = 32767

si = li;
printf("si = %d\n", si);      // si = 32767
```

lesser type is converted to type `int`. If not, the lesser type is converted to type `unsigned int`.

Definition 2.10. The *usual arithmetic conversions* rule is an expansion of the integer conversion rank and integer promotions rules as it applied to the other arithmetic types bigger than `int`.

For the float types, the usual arithmetic conversion promotes the smaller type to the larger found in a single expression. If an expression has a mix of floats and integers, then the integers are promoted to the largest float type. If the expression contains only integer types, then the integers are promoted to the largest integer type. The unsigned integer type takes precedence whenever there is a mix of signed and unsigned integer types unless the signed integer has a larger bitwise precision than the largest unsigned integer type.

According to Mitchell [80], if a language performs coercion, then a sub-typing relationship exists among its types. Indeed, the C coercions based on integer promotions and the usual arithmetic promotions adhere to a set of sub-type based rules. In a sub-type relation, denoted as

$$\tau_1 \subseteq \tau_2, \tag{2.1}$$

τ_1 is a sub-type of τ_2 iff τ_2 can represent all of the representable τ_1 values. Sub-type relations are reflexive, transitive and antisymmetric:

- *reflexive* – $\tau \subseteq \tau$
- *transitive* – if $\tau_1 \subseteq \tau_2$ and $\tau_2 \subseteq \tau_3$, then $\tau_1 \subseteq \tau_3$
- *antisymmetric* – if $\tau_1 \subseteq \tau_2$ and $\tau_2 \subseteq \tau_1$, then $\tau_1 = \tau_2$

In C, all arithmetic data types belong to a sub-type relation where integer types $\subseteq$ real floating types and are ordered by the bitwise precision of each individual type. The C integer conversion rank that orders the integer types by bit width is an example [52, 53, 54].

However, the typing semantics of integer casting in C are often complex and unintuitive [15]. Even Ritchie admitted that the typing rules were quirky and flawed [110]. For example, suppose we have variable `x` of an unsigned integer type with the value of `10` and we use `x` in a relational expression such as `x > -10`. If the bit width of the unsigned integer type for `x` is equal to or greater than the native integer type (`int`) of the underlying hardware, `x > -10` will evaluate to false. If the unsigned integer type has a bit width less than `int`, `x > -10` will evaluate to true. Integer promotions and usual arithmetic conversions are both responsible for the phenomena behind our `x > -10` example, as explained below.

Integer promotion rules require all integer type of operands in an expression smaller than `int` to be promoted to `int` before the expression is evaluated. The rationale behind the rule is that the smaller integer types are more likely to enter an error condition during expression evaluations. For example, the representable value range of `signed char` integer type is `-127` to `127`. Suppose we have an expression, `a * b - c` where `a`, `b`, and `c` are of type `signed char` and hold the values `65`, `2`, and `3` respectively. Without the integer promotions rule, the evaluation of `a * b` would cause an overflow condition before the subtraction of 3 is applied.

In the case of `x > -10` where `x` is an unsigned integer type that is smaller than `int`, the integer promotion rule is applied. Both operands are promoted to type `int` and the values they hold represent `10` and `-10` respectively. Thus, `10 > -10` is true.

Likewise, the goal of the usual arithmetic conversions is to find a common real type for all operands of an expression and for the result of the expression [52, 53, 54]. Under usual arithmetic conversions, the arithmetic data types of all operands in an expression are basically cast to the largest arithmetic data type contained within the expression. The usual arithmetic conversions are applied after the application of the integer promotions. If it is the case that all operands are of integer type, the usual arithmetic conversions mandate that the resulting expression type will be the same type as the largest integer type.

There is a caveat, and that is if all integers in an expression have the same bit width, but at least one is unsigned and the other(s) signed, then the operands assume the largest unsigned integer type. In the case of `x > -10`, if `x` is of type `unsigned int` (having the same bit width as `int`) and holds the value `10`, integer promotions are first applied to the literal `-10` to type the literal as an `int` and then, usual arithmetic conversions converts the literal type of `int` to the type of `unsigned int`. Signed integers use the most significant bit (MSB) as the sign bit. If set, the value is negative, otherwise, the value is positive. In unsigned integers the MSB is a precision bit. Since the MSB was set when the literal `-10` was initially typed as an `int`, the set MSB becomes a precision bit when the literal is converted to type `unsigned int` and the once small negative value is interpreted as the very large positive value of `4294967286` (on a 64-bit platform). Thus, `10 > 4294967286` is false.

From the usual arithmetic conversions between signed and unsigned integer types, it is apparent that the integers are divided into two disjoint sub-type sets, signed and unsigned. The signed integer subset relation is

Sub-Type: Signed Integers

$$\frac{}{\texttt{CHAR} \subseteq \texttt{SHORT} \subseteq \texttt{INT} \subseteq \texttt{LONG} \subseteq \texttt{LLONG}} \tag{2.2}$$

and the unsigned integer relation is

Sub-Type: Unsigned Integers

$$\frac{}{\texttt{UCHAR} \subseteq \texttt{USHORT} \subseteq \texttt{UINT} \subseteq \texttt{ULONG} \subseteq \texttt{ULLONG}} \tag{2.3}$$

It is also apparent that, except for one cast, most casts involving integers can cause an error. The exception is an up-cast between integers of the same sign type. The inference rules of an up-cast, depicted as

Cast: Up-Cast

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{e} : \tau \\ \tau \subseteq \tau'\end{array}}{\Gamma \vdash \tau'(\mathsf{e}) : \tau'} \tag{2.4}$$

states that in the typing environment Γ where the implication is an expression e of type τ and τ is a subset of type τ', then in the same typing environment the implication is the cast operation $\tau'(e)$ will result in e being of τ'.

On the other hand, the sub-typing relationship in a down-cast is backward, where $\tau \not\subseteq \tau'$ and $\tau' \subseteq \tau$. Likewise, a sign-conversion-cast, does not have a subtype relationship because neither $\tau \not\subseteq \tau'$ nor $\tau' \not\subseteq \tau$. Whenever a $\not\subseteq$ relationship is present in the typing environment Γ, the cast is unsafe because errors can be produced. The inference rule for unsafe-cast is

Cast: Unsafe

$$\frac{\Gamma \vdash \mathsf{e} : \tau \quad \tau \not\subseteq \tau'}{\Gamma \vdash \tau'(\mathsf{e}) : \tau'} \tag{2.5}$$

as it shows the possibility for error.

Whenever an integer enters an error condition during the execution of a C program, it stays in it until it is reassigned another legal representable value. If it is in error condition and is used as an operand in an expression other than the left hand side (LHS) of an assignment operator, the error propagates. Integer errors place a system in a state that is potentially vulnerable to either failure resulting from a possible denial of service condition or several other security exploitations, such as the execution of arbitrary code and privilege escalation.

In many cases, however, integer error conditions and the potential vulnerabilities they introduce can be eliminated by the means of range checking a value prior to subjecting it to a cast, using it as an operand, or before assigning it to a variable. Let v represent the value in question, $\tau_{\mathrm{MIN}_{val}}$ represent the smallest representable value of type τ, and $\tau_{\mathrm{MAX}_{val}}$ represent the largest representable value of type τ. If v is to be assigned to a variable of type τ or the type of v is to be cast to type τ and

$$\tau_{\mathrm{MIN}_{val}} \leq v \leq \tau_{\mathrm{MAX}_{val}}, \tag{2.6}$$

then the operation is safe.

2.4.2 Integer Errors Introduced by Operators

In addition to casting errors, C integers are also prone to integer vulnerabilities resulting from arithmetic operations (Table 2.1 and Table 2.2). According to the standard, most operators require their operands to undergo the usual arithmetic conversions prior to the

Listing 2.8: When $90 + 40 = -126$

```
char c1 = 90;
char c2 = 40;
char cResult;

cResult = c1 + c2;
printf("%d", cResult); // cResult = -126
```

Table 2.1: C operators and their potential to produce overflow

Op	Overflow	Op	Overflow	Op	Overflow	Op	Overflow
+	yes	-=	yes	<<	yes	<	no
-	yes	*=	yes	>>	no	>	no
*	yes	/=	yes	&	no	>=	no
/	yes	%=	yes	|	no	<=	no
%	yes	<<=	yes	^	no	==	no
++	yes	>>=	no	~	no	!=	no
--	yes	&=	yes	!	no	&&	no
=	no	|=	no	unary +	no	||	no
+=	yes	^=	no	unary -	yes	? :	no

operation on the operands. Suppose we were to add two integer values, say 90 and 40. We would expect the value to be 130. Yet, in the environment shown in Listing 2.8 the result is different.

After compiling Listing 2.8 in `gcc` [35] without a single warning or error message, the output of the code is `-126` upon execution. The short answer to what went wrong is that since `cResult` is an `8` bit signed `char`, the MSB is a sign bit and the maximum positive value that it can hold is `127` with a binary bit pattern of `0111 1111`. The value 130 can be held in an eight-bit pattern as `1000 0010` when the MSB is used as a precision bit. However, when the binary bit pattern for `130` is held by a signed integer type, the MSB is set and on a two's complement platform, the interpreted value is -126. This error is a classic example of an integer overflow.

Table 2.2: C operators and their potential to produce a wrap

<table>
<tr><th>Op</th><th>Wrap</th><th>Op</th><th>Wrap</th><th>Op</th><th>Wrap</th><th>Op</th><th>Wrap</th></tr>
<tr><td>+</td><td>yes</td><td>-=</td><td>yes</td><td><<</td><td>yes</td><td><</td><td>no</td></tr>
<tr><td>-</td><td>yes</td><td>*=</td><td>yes</td><td>>></td><td>no</td><td>></td><td>no</td></tr>
<tr><td>*</td><td>yes</td><td>/=</td><td>no</td><td>&</td><td>no</td><td>>=</td><td>no</td></tr>
<tr><td>/</td><td>no</td><td>%=</td><td>no</td><td>|</td><td>no</td><td><=</td><td>no</td></tr>
<tr><td>%</td><td>no</td><td><<=</td><td>yes</td><td>^</td><td>no</td><td>==</td><td>no</td></tr>
<tr><td>++</td><td>yes</td><td>>>=</td><td>no</td><td>~</td><td>no</td><td>!=</td><td>no</td></tr>
<tr><td>-</td><td>yes</td><td>&=</td><td>no</td><td>!</td><td>no</td><td>&&</td><td>no</td></tr>
<tr><td>=</td><td>no</td><td>|=</td><td>no</td><td>unary +</td><td>no</td><td>||</td><td>no</td></tr>
<tr><td>+=</td><td>yes</td><td>^=</td><td>no</td><td>unary -</td><td>yes</td><td>? :</td><td>no</td></tr>
</table>

2.4.3 Other Conditions Leading to Vulnerabilities

In addition to integer error conditions, undefined behavior can arise whenever a uninitialized variable is used as an operand in an expression other than the LHS of an assignment operator. Compilers, such as `gcc` [35], will provide a warning if one or more of its optional additional warning flags such as `-Wall` and `-Wextra` is set and it sees an uninitialized variable as it produces an executable file. However, if the warning is ignored and the uninitialized variable is used as an operand in an expression, other than the LHS of an assignment operator, it will contain a value of whatever bitwise artifacts remained in its memory address prior to the variable's declaration and likely produce an unexpected behavior.

Another potential error condition is a divide by `0`. Most programmers would not directly hard code a divide by `0`. But, it is possible for a variable to wind up holding `0` and being used as a divisor. Without checks prior to either a division or mod operation, this scenario could have serious consequences. For example, the Aegis class missile cruiser USS Yorktown lost its propulsion system lying dead in the water for nearly three hours, after a system administrator entered zero into a data field for the remote data base manager program, causing a divide by zero error [118].

C has a collection of operators other than those used for assignments that are designed to produce a side effect. In many cases, these operators, such as the post increment, offer

convenience for a programmer. For example, it is much more efficient to write `x++` than it is to write `x = x + 1`, when `x` is being used as a counter. However, there is nothing syntactically incorrect about using these operators in other expressions such as `a = b + x++`. Programmers who use such idioms need to be aware of the order of evaluation in any given expression, yet order is compiler dependent.

According to the standard [52, 53, 54], a variable may be modified only once between sequence points. A sequence point often marks the beginning and end of an expression. Thus, the evaluation behavior of an expression such as `a = x++ + x++` is unknown.

2.4.3.1 Undefined Behaviors

§3 of the C standard defines three types of unknown behavior:

1. A behavior is unspecified when the standard specifies two or more different possibilities without imposing requirements as to which possibility is correct. The evaluation of function arguments is one of the 50 unspecified behaviors enumerated in Annex J of the standard.
2. A behavior resulting from the use of a nonportable or erroneous program construct or of erroneous data that is unrestricted by the standard is undefined. The behavior of integer overflow is one of 191 undefined behaviors enumerated in Annex J of the standard.
3. A behavior is implementation-defined if it is unspecified by the standard, but the behavior choice is made and documented by individual implementations. How the high-order bit is propagated when a signed integer is right shifted is one of 112 implementation defined behaviors enumerated in Annex J of the standard.

Many of the errors described in §2.4.3 and §2.4.3.1 of this document are software errors and according to Cardelli, are not normally prevented by type systems [17].

Chapter 3: Current Error Mitigation Measures

Because C is well known as not being type safe, several approaches have been offered in "*building a better mouse trap*" to mitigate type safety violations. This chapter is an exploration of the implemented and proposed measures intended to make C more type safe.

3.1 Optional Compiler Warnings

C compilers, such as `gcc` [35], are designed to enforce C's casting rules. Whenever a compiler recognizes an error or other potential problem, it will issue either an error or a warning statement. The difference between an error and a warning is that if an error is recognized and reported, then the executable file is not generated. On the other hand, if only one or more warnings is reported, an executable is generated.

The `gcc` compiler has optional provisions to conduct a more rigorous error/warning check during the compilation process than it does when operating in its default mode. More thorough checks are activated by setting one or more flag options. For example, the `-ftrapv` flag is designed to spot potential integer overflow and wrapping conditions, `-Wall` is designed to give all warnings, and `-Wextra` is designed to generate more warnings than `-Wall`. Note, however, setting the `-Wall` and the `-Wextra` flag did not catch the intentional errors in Listing 3.1. The comments in Listing 3.1 are in place to describe `gcc`'s performance. In addition, it has been shown that `-ftrapv` catches only a limited number of wrapping vulnerabilities [15].

The flags `-Wall` and `-Wextra` will both issue a warning statement if a variable is declared and not initialized as `gcc` creates an executable. The problem is as mentioned above, the generation of only warnings does not prevent creation of an executable that may be later run. If an uninitialized variable is used as an operand during the execution, it will likely lead to an unexpected result. The reason for only a warning in this case is

Listing 3.1: Typing errors missed by gcc

```
int sx0 = -10;            // error free
unsigned int ux1 = sx0; // error missed
unsigned int ux2 = -10; // error caught
unsigned int ux3;         // error free
ux3 = -10;                // error missed
```

that an initialized variable does hold a value. The value is generally unknown and it is derived by the residual bit pattern left in memory from the previous memory store of the same address space.

3.2 Safe Coding Guidelines and Practices

Safe coding guidelines are offered to help programmers avoid making mistakes by describing established safe coding practices and standardized coding styles. For example, the 2004 the Motor Industry Research Association (MISRA) guidelines notes that the expressiveness of C can be leveraged to write either "*well laid out structured code*" or to write "*perverse and extremely hard-to-understand code.*" The latter coding style is unacceptable for use in type-safe environments [79].

MISRA has published two safe coding guidelines with respect to C. The first was published in 1998 and was geared toward the automotive industry for its control systems [78]. The second was released in 2004 and was more general in purpose by providing guidelines for critical systems [79]. Both guidelines are replete with sets of mandatory and advisory rules with a goal to create a safe subset of C suitable for embedded systems. The 2004 publication, for example, contained 121 mandatory and 20 advisory rules. An example 2004 mandatory rule is §5.10.6 Rule 10.1 that states that "*an integer type of an expression shall not be implicitly converted to another type under any of the following conditions*:

- *the conversion is not to a wider integer type of the same signedness*
- *the expression is complex*

Table 3.1: MISRA recommended `typedef`ed arithmetic type names for 32-bit platforms

```
typedef  char            char_t;
typedef  signed char     int8_t;
typedef  signed short    int16_t;
typedef  signed int      int32_t;
typedef  signed long     int64_t;
typedef  unsigned char   uint8_t;
typedef  unsigned short  uint16_t;
typedef  unsigned int    uint32_t;
typedef  unsigned long   uint64_t;
typedef  float           float32_t;
typedef  double          float64_t;
typedef  long double     float128_t;
```

- *the expression is not constant and it is a function argument*
- *the expression is not constant and it is a return expression*"

§6.6 Rule 6.3 is an example advisory rule that urges programmers to use `typedef` types whose defined type name denotes type details such as size and signedness to declare variables instead of using the basic arithmetic type names such as the integers and floats in a declaration statement. Table 3.1 lists the MISRA suggested `typedef` names when developing C source code for 32-bit platforms.

MISRA does have a licensing fee for usage. However, there are several good alternatives that can be had at no charge. For example, The Software Engineering Institute (SEI) of Carnegie Mellon University offers a free safe coding guideline on the Internet [18].

3.3 Safe Integer Libraries

Safe integer libraries, such as *SafeInt* [70], *IntSafe* [49], and the *Integral Security - Secure Integer Library* [19], are at the disposal of C programmers, to help protect integers from entering an error condition. These libraries are generally a collection of function calls designed to safely apply operations on integer operands. Instead of using a multiplication

Listing 3.2: A safe multiplication function call

```
if(safe_mult_32s(y, z))
  x = y * z;
```

Listing 3.3: SafeMult function checking result value to value range of return type

```
int safe_mult_32s(int lhs, int rhs)
{
  int INT_MIX = -2147483647;
  int INT_MAX = 2147483647;
  _Bool error_condition = 1; // set to true
  signed long long temp;
  temp = (signed long long)lhs * (signed long long)rhs;

  if ( (temp < INT_MIN) || (temp > INT_MAX))
  { // The product cannot fit in a 32-bit int OVERFLOW ERROR
    error_condition = 0;   // set to false
  }
  return error_condition;
}
```

expression such as `x = y * z;` where the three variables are of a signed 32 bit integer type, a programmer may use a function, such as Listing 3.2, that checks to see if the operation is safe before assigning `x` the product of `y` and `z`.

In the above example, the function simply checks the resultant value of the operation to see if it is within the valid range that the return type can represent (Listing 3.3). More complex functions may compare value of each operand to the other operand in order to determine if an overflow is the likely outcome of the operation (Listing 3.4).

3.4 Safe Subsets of C

Over the years, several safe subsets of C have been introduced as an alternative for use in programming type safe and critical systems. Among them are EC- [43], CCured [92], Clight [13] and Cyclone [124]. A safe subset of C is a language derived from C that disallows many of the unsafe constructs permitted by C. Cyclone, for example, was

designed with the intent that programmers accustomed to working with C could still "*feel the bits*." For the most part, Cyclone uses the same preprocessor, lexical conventions, and grammar that C uses. Cyclone uses the same data types and data representations that C uses, such as pointers, arrays, structures, unions, enumerations, and all of the usual floating point and integer types. The major differences between Cyclone and C are all safety related. Unlike C compilers, the Cyclone compiler performs a static analysis on the source code and inserts run-time checks into the compiled output at those places where the analysis cannot determine if an operation is safe. If the compiler cannot guarantee that a program is safe, the program is rejected. As a result, some perfectly safe programs may be rejected. A short list of restrictions imposed by Cyclone include:

- Pointer arithmetic is restricted
- Pointers must be initialized before use
- Limitations are placed on the key word `free` to prevent dangling pointers
- Only safe casts and unions are allowed
- Use of `goto` is prohibited

In addition to these restrictions, Cyclone has added several extensions to provide safety while maintaining C programming idioms. For example, the fat pointer was introduced to support pointer arithmetic with run-time bounds checking. The fat pointers in Cyclone are represented with three words: the base address, the bounds address, and the current pointer location. Compared to C pointers, the fat pointers have a larger space overhead, larger cache footprint, increased parameter passing overhead, and increased register pressure.

Although the original release of Cyclone used a combination of programmer-supplied annotations, an advanced type system, a flow analysis and run-time checks to ensure type safety, Cyclone has been extended to enhance its safety capabilities. For example, Hicks et al. [45] added *unique pointers* into the memory management framework to

locally track the state of an object by forbidding uncontrolled aliasing, and Grossman [41] introduced a new `lock_t`$\langle \ell \rangle$ where ℓ is the lock name to support multithreading.

3.5 Proposed Extensions to C

Others [37, 63, 25] have proposed making changes to the C types with new integer types as opposed to introducing safe subsets of the language. Their rationale is to preserve the ease of mixing data types without having to resort to cumbersome workarounds required for stronger typed languages. There have been two main proposals.

3.5.1 Ranged Integers

Gennari et al. [37] introduced the concept of ranged integers to return a guaranteed predefined value in the event of integer overflow. Ranged integers can be either signed or unsigned. Their valid value range is enforced after an assignment or an initialization through the use of storage policies.

The storage policy may enforce either modwrap semantics or saturation semantics. Modwrap semantics enforces a certain modulo behavior over a defined range of values while saturation semantics produces the legal maximum value for the integer type in the case of an overflow and the legal minimum value in the case of an underflow. Ranged integers may appear in an expression as an `rvalue`, a `lvalue`, or as a pointer or an array; and are intended to perform checks that would otherwise need to be done manually by control statements such as `if`.

3.5.2 Infinitely Ranged Integers

In the infinitely ranged integer model, all integer values are regarded as if they are being held in the greatest width integer supported by the underlying system before usage. The model is applied to both signed and unsigned integer types and relies on observation

Standard Section	Proposed Critical Undefined Behavior
6.2.4	An object is referred to outside of its lifetime
6.3.2.1	An lvalue does not designate an object when evaluated
6.3.2.3	A pointer is used to call a function whose type is not compatible with the pointed to type
6.5.3.2	The operand of the unary * operator has an invalid value
6.5.6	Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that points just beyond th array object and is used as the operand of a unary * operator that is evaluated
7.1.4	An invalid argument to a function can cause an invalid address computation and/or invalid access
7.20.3	The value of a pointer that refers to space deallocated by a call to the free or realloc function is used
7.21.1, 7.24.4	A string or wide string utility function is instructed to access an array beyond the end of an object

Table 3.2: Critical undefined behaviors.

points. The core idea behind the model is that it is acceptable to delay catching an incorrect integer value until an observation point is reached, but just before the bogus value is either used as an output or causes a critical undefined behavior. See Table 3.2 for a list of critical undefined behaviors.

Both Keaton et al. [63] and Dannenberg et al. [25], have proposed that the infinitely ranged integer model be implemented in `gcc` [35] such that compiled executables either produce a legal value that can obtained using an infinitely ranged integer or produce a runtime constraint violation. If an observation point is reached while the overflow traps are enabled and no traps have been raised, then the output value of an integer is assumed to be correctly represented. Observation points are those instances of time when there is an output, including volatile object accesses, a pointer dereference leading to undefined behavior, or a call to a library routine using arguments that lead to undefined behavior. To eliminate previously undefined behavior by providing definitions of optional predictable semantics, the model may be implemented in the following ways, and can be customized for either the execution environment or implementation:

- Overflow can set a flag which compiler-generated code will test later
- Overflow can immediately invoke a runtime-constraint handler
- The testing of flags can be performed at an early point or be delayed

3.6 Source Code Analysis Tools

Except for the safe coding guidelines, use of safe integer libraries or safe subsets of the C language may not be practical. Safe integer libraries generally add an operational overhead that may be too much for applications with limited memory or power, and for real time systems. Safe subsets of C may not be expressive enough to adequately program sophisticated systems. In these situations, programmers can deploy one of the available source code analysis tools. The analysis tools come in two flavors: runtime and static.

3.6.1 Runtime Analysis

The runtime analysis tools are designed to dynamically analyze code while the code is executing. An example runtime analysis tool is RICH (Run-time Integer CHecking), written in OCaml [44] introduced by Brumley *et al.*[15].

Operationally, RICH checks the intermediate representation of source code during the code-generation phase to flag any potentially unsafe integer operations. Central to the operation of RICH are two rules to find unsafe integer operations, R-UNSAFE and R-BINOP$_{\mathbb{Z}}$. R-UNSAFE relies on the three type safety rules: T-UNSIGNED, T-SIGNED, and T-UPCAST. All three are based on expanding the notion of sub-typing ($\subseteq$) and are supported by three type-safety check rules: D-CHECK (down cast), S-U-CHECK (signed to unsigned), and U-S-CHECK (unsigned to signed). Whenever a potentially unsafe cast $((\tau)e : \tau e')$ is encountered, expression e is rewritten to another expression e' which is evaluated to a value `x`. The value `x` is later checked during run-time for data loss. Specifically,

$$\text{if } \tau_{min} \leq \mathtt{x} \leq \tau_{max} \text{ is false,}$$

an `error` condition is generated.

RICH also uses the function R-BINOP$_{\mathbb{Z}}$ to check for binary math operations that may produce an overflow/underflow condition. The $\mathbb{Z}$ represents the next precision higher integer type than that of the largest integer type involved in the operation. A temporary variable of type $\mathbb{Z}$ is used to hold the intermediate result of the operation so that it can be compared against the minimum and maximum representable values of the expression's actual type. If the temporary value falls within the expression type's representable value range, then it type safe. Otherwise, an error condition is generated. The drawback to using R-BINOP$_{\mathbb{Z}}$, however, is that the checks on operations are limited to 32-bit integer operands. C's largest integer type is 64-bits.

3.6.2 Static Analysis

A project of the National Institute of Standards and Technology (NIST) is the Software Assurance Metrics and Tool Evaluation (SAMTE) program [89]. SAMTE is "dedicated to improving software assurance by developing methods to enable software tool evaluations, measuring the effectiveness of tools and techniques, and identifying gaps in tools and methods."

Of interest here is the collection of commercial and open source Source Code Security Analyzers that are intended to examine C source code to detect and report weaknesses that may lead to a security vulnerability [90]. Many of these tools were briefly discussed in Chapter 1. Most are coded in uniquely different languages and each targets a select sub-set of the C integer problem. The commercially available tool, Astrée [24] written in OCaml [44], appears to give the most complete coverage. However, the coverage is only realized by rerunning the tool in several different analysis modes. That said, major gaps remain in both tools and methods when it comes to analyzing C integer type safety.

Listing 3.4: SafeMult function checking operands for potential overflow

```
_Bool safe_mult_32s(int lhs, int rhs)
{
  int INT_MIX = -2147483647;
  int INT_MAX = 2147483647;
  _Bool error_condition = 1;

  if ( (lhs == 0) || (rhs == 0))
  {
    return error_condition;
  }
  if( lhs > 0)
  {
    if( rhs > 0)
    {
      if(lhs > (INT_MAX / rhs))
      { // OVERFLOW ERROR
        error_condition = 0;
      }
    }
    else
    {
      if ( rhs < (INT_MIN / lhs))
      { // OVERFLOW ERROR
        error_condition = 0;
      }
    }
  }
  else
  {
    if(rhs > 0)
    {
      if( lhs < (INT_MIN / rhs))
      { // OVERFLOW ERROR
        error_condition = 0;
      }
    }
    else
    {
      if( (lhs != 0) && (rhs < (INT_MAX / lhs)))
      { // OVERFLOW ERROR
        error_condition = 0;
      }
    }
  }
  return error_condition;
}
```

Chapter 4: Introduction to Language Formalism

With this chapter, the theme of this dissertation shifts to the formalism of the C language as it contains introductory material with respect to language formalization. The following chapters show the work to define the syntax of C types and the static typing semantics of both C expressions and statements.

4.1 Elements of Language: Syntax and Semantics

All human languages, natural and formal, have two elementary components: syntax and semantics.

Definition 4.1. *Syntax*, often called grammar, is the set of rules defining the proper, correct usage of language constructs including words and punctuation.

As prescribed by Cardelli, the first step to formalize a language is to describe its syntax of types and terms [17]. Accordingly, types express the static knowledge of the program and the terms, such as expressions and statements, express the algorithmic behavior. It is commonplace for the syntax of expressions and statements of programming languages such as C to be formally presented in Backus-Naur Form (BNF) [8, 91]. BNF is used because it exhibits the syntax in a context-free manner.

Spiser [119] defines a a context free grammar (CFG) using a four tuple

$$(V,\ \Sigma,\ R,\ S), \tag{4.1}$$

where:

1. V is a finite set of variables.
2. Σ is a finite set of terminals disjoint from the variables.
3. R is a finite set of rules (each rule is constructed of a variable, a string of variables, and terminals).

4. $S \in V$ is the start variable.

Definition 4.2. The *semantics* of a language is the meaning of its constructs.

The semantics of the programming language C is generally, informally, defined in the standard using natural language. As a result, the exact meaning of expressions and statements defined in the standard are often left open to interpretation.

4.2 Common Methods to Formalize Language Semantics

There are three main formal semantic approaches used by computer scientists to clear up confusion arising from informally defined semantics found in the standard: *axiomatic*, *denotational*, and *operational*.

4.2.1 Axiomatic Semantics

The axiomatic semantics often used by computer scientists was first introduced by Floyd [32] and further developed Hoare [46].

Definition 4.3. *Axiomatic semantics* is the means of using generally accepted truths (axioms) to provide meaning.

Fundamental to this semantics known as Hoare or Floyd-Hoare logic is the Hoare triple that describes how the execution of code changes the computational state. A Hoare triple is of the form

$$\{\ P\ \}\ C\ \{\ Q\ \} \tag{4.2}$$

where P and Q are assertions and C is either a command or a statement. Specifically, P is the precondition that must be satisfied so that after the execution of C and termination, the postcondition Q is satisfied.

Like denotational and operational semantics, Hoare logic is often presented in the form of inference rules. An inference rule takes the form

Inference Rule Structure:

$$\frac{P_1 \quad P_2 \quad \ldots \quad P_n}{C} \tag{4.3}$$

where each P_i are the premises and the C is the conclusion. The conclusion is true if and only if (iff) each premise is true.

Generally speaking, standard Hoare logic is considered to be partially correct in the sense that it does not show that C actually terminates. Termination is required for total correctness and proof of termination is critical for formal verification. To show termination, a measure from the domain of any well-founded-relation, such as the ordinal numbers introduced by Cantor [16], must be present. An ordinal number denoted as α can be any positive integer $\{1, \ldots, n-1, n\}$ used to create an ordered set of numbers (ordinals)

$$\{0, \ldots, \alpha - 1\}. \tag{4.4}$$

A measure based on a set of ordinals that decreases according to the relation along every possible step of an algorithm shows termination because the basic property of a well-founded relation is that no infinite descending chains can exist.

4.2.2 Denotational Semantics

Scott and Strachey [114] are credited for developing denotational semantics.

Definition 4.4. *Denotational semantics* is the construction of mathematical objects called denotations to represent the domains that are used to describe the meaning of programming language expressions.

To give meaning to a computer program, Scott and Strachey viewed a program as a function comprised of partial functions mapping inputs to outputs. To understand the mapping process, each program construct, such as commands, environments, expres-

sions, identifiers, states, values, variables, etc., are respectively grouped to their own unique domains. Partial functions that are denoted as the members of each domain are then expanded to form partially ordered sets called complete lattices or posets (partially ordered sets) represented as

$$(D, \sqsupseteq). \tag{4.5}$$

The basic structure of a complete lattice has two requirements: partial ordering and having a least upper bound. A partial ordering for any domain D is written as

$$x \sqsubseteq y \tag{4.6}$$

for

$$x, y \in D. \tag{4.7}$$

The relationship $x \sqsubseteq y$, intuitively meaning x approximates y, is reflexive, transitive, and anti-symmetric. If in D, $\chi \subseteq 0$, 0 exists and is the least upper bound (lub) of subset χ, written

$$\bigsqcup \chi. \tag{4.8}$$

Thus, for $\forall y \in D$

$$\bigsqcup \chi \sqsubseteq y \text{ iff } x \sqsubseteq y,\ \forall x \in \chi \tag{4.9}$$

uniquely characterizes $\bigsqcup \chi \in D$.

In addition to a lub for each subset contained in a complete lattice, the lattice also has an extreme bottom $\perp$ and an extreme top $\top$ lub. For Scott and Strachey, the $\perp$ was the empty subset $\emptyset$ and the $\top$ was the full subset of D respectively written as

$$\perp = \bigsqcup \emptyset \text{ and } \top = \bigsqcup D, \tag{4.10}$$

such that $\forall x \in D$, it is the case that

$$\bot \sqsubseteq x \sqsubseteq \top. \tag{4.11}$$

Because of this, each function defined and applied to a complete lattice has to be monotonic in the sense that they preserve ordering and have fixed point solutions.

According to Schmidt [113], all denotational semantic definitions consist of three parts: an abstract syntax of the language to be defined, the semantic algebras, and valuation functions. The abstract syntax and semantic algebras are used to define the appearance and meaning of a language. A semantic algebra is combination of a domain and its operations to show functionality and to specify mappings. Functionality is defined by the operation's domain and co-domain. For example the functionality of

$$f: \ D_1 \times D_2 \times \ldots \times \ D_n \rightarrow A \tag{4.12}$$

says that f needs an argument from each domain D_i to produce an answer in its co-domain A. Set functions are classified by they mappings they produce. Example classifications include:

1. $f: \ R \rightarrow S$ is an *one-one* (1-1) function iff for $\forall x \in R$ and $\forall y \in R.$, $f(x) = f(y)$ implies $x = y$.
2. $f: \ R \rightarrow S$ is an *onto* function iff $S = \{y \mid \text{there exists some } x \in R \text{ such that } f(x) = y\}$.
3. $f: \ R \rightarrow R$ is the *identity* function for R if $\forall x \in r$, $f(x) = x$.
4. for some $f: \ R \rightarrow S$, if f is one-one and onto, then the function $g: \ S \rightarrow R$, defined as $g(y) = x$ if $f(x) = y$ is called the *inverse* function of f. Function g is denoted by f^{-1}.

The valuation functions connect or map the abstract syntax to the semantic algebras; and, there is a valuation function for each syntactic domain to assign a uniform meaning to a construct regardless of its context.

4.2.3 Operational Semantics

There are two categories of operational semantics: structural or small-step semantics and natural or big-step semantics.

Definition 4.5. *Operational semantics* is a mathematical description of how a computer program behaves by defining its sequence of computational steps.

For example, the evaluation process of a simple expression e, such as

$$(\mathsf{1} + (\mathsf{2} + \mathsf{3})) + (\mathsf{4} + \mathsf{5}), \tag{4.13}$$

to its result m can be expressed as

$$e = e_1 \rightarrow \ldots \longrightarrow e_{n-1} \rightarrow e_n = m \tag{4.14}$$

where for $\forall e_i$,

$$e_i \rightarrow e_{i+1} \equiv e_{i+1} \text{ is the result of the first evaluation step of } e_i. \tag{4.15}$$

4.2.3.1 Small-step Semantics

Plotkin [101] is credited in developing small step semantics as a logical method to depict the behavior of a program as it executes by using a transition system expressed in a set of inference rules.

Definition 4.6. A *transition system* denoted ts, is a structure

$$\langle\, \Gamma, \longrightarrow \,\rangle \tag{4.16}$$

where Γ is a set of elements γ (configurations) and $\longrightarrow \subseteq \Gamma \times \Gamma$ is a binary relation called the transition relation.

Notation of ts is of the form

$$\gamma \longrightarrow \gamma' \tag{4.17}$$

and is read as "there is a transition from the configuration γ to the configuration γ'".

Definition 4.7. A *terminal transition system* denoted tts, is a structure

$$\langle\, \Gamma, \longrightarrow, T \,\rangle \tag{4.18}$$

where $\langle\, \Gamma, \longrightarrow \,\rangle$ is a ts and $T \subseteq \Gamma$ is the set of final configurations that satisfy the condition

$$\forall_{\gamma} \in T\ \forall_{\gamma'} \in \Gamma . \gamma \not\longrightarrow \gamma'. \tag{4.19}$$

Plotkin built configurations for small step semantics using the linguistics developed by Scott and Strachey [114] and from the main syntactic classes used by Gordon [39] and Tennet [122]; expressions, commands, and declarations. For example, the ts (configuration) of an expression is the expression e and a store (state) σ, $\langle\, e, \sigma \,\rangle$. The relation

$$\langle\, e, \sigma \,\rangle \longrightarrow \langle\, e', \sigma \,\rangle \tag{4.20}$$

means one step of the evaluation of e with store σ results in the expression e' without a change to the store σ. Unlike expressions, commands are stored in the state and the transition sequences of commands is seen as state changing by terminating in a new

state σ'. A command configuration $\langle\, c, \sigma \,\rangle$ is the transition sequences for the terminating executions of a command c from a store σ is formalized as

$$\langle c\ , \sigma\ \rangle = \langle\ c_0, \sigma_0\ \rangle \longrightarrow \langle\ c_1, \sigma_1\ \rangle \longrightarrow \ldots \longrightarrow \langle\ c_{n-1}, \sigma_{n-1}\ \rangle \longrightarrow \sigma_n \tag{4.21}$$

As is the case of statements, declarations change state. However, the evaluation steps of all three syntactic classes are syntax directed. If a terminal configuration T cannot be reached, *i.e.,* no γ' with $\gamma \longrightarrow \gamma'$, the final configuration reached from the stepwise evaluation is considered to be *stuck*. A stuck configuration is an error condition.

Definition 4.8. A *stuck* configuration γ is whenever $\gamma \notin T$ and $\neg\exists\gamma'.\ \gamma \longrightarrow \gamma'$ in a *tts* structure $\langle \Gamma, T, \longrightarrow \rangle$.

4.2.3.2 Big-step Semantics

Big step semantics was introduced by Kahn [57] as a "unified manner to present the different semantics of programming languages, such as *dynamic* semantics, *static* semantics, and translation." The original notation for big-step sequents took the form

$$\sigma \vdash\ a \Rightarrow i \tag{4.22}$$

that is interpreted as an implication that a evaluates to i in state (environment) σ.

Definition 4.9. *Big-step semantics* is a "denotational" style operational semantics.

Big-step definitions are definitions of functions or relations to interpret each language construct in an appropriate domain. The big-step sequents (consequences or results) are relations over configurations expressed as $C \Rightarrow R$ or $C \Downarrow R$ with the meaning that R is the configuration obtained after the evaluation of C. A big-step rule takes the form

Big Step Rule Structure:

$$\frac{C_1 \Rightarrow R_1 \quad C_2 \Rightarrow R_2 \quad \ldots \quad C_m \Rightarrow R_n}{C \Rightarrow R} \tag{4.23}$$

where $C,\ C_1,\ C_2,\ \ldots,\ C_n$ are configurations of program fragments and $R,\ R_2,\ \ldots,\ R_n$ are result configurations. Each R is irreducible as it cannot be advanced.

4.3 Approaches to the Formalization of Language Semantics

The previous discussion on the popular approaches to formalize semantics is intended for introductory or review purposes only. By no means was it an attempt to cast a more favorable light on any single approach. Which method to use is a matter of personal choice and what is being modeled. With complex languages such as C, a certain construct may be easier to formalize using one method while another method may be better suited to formalize a different construct. In fact, a formalization process may use elements from any approach, in any combination. In this dissertation, a denotational like big-step approach will be utilized. At times, a small-step feel will manifest with the formalization of the C standard required type transitions. As for the Hoare logic, the static type safety analysis tool to be introduced in Chapter 8 is designed with several sets of predicate functions to serve as the various premises to reach individual conclusions.

Chapter 5: Historic Attempts to Formalize C

According to Reis and Stroustrup [108], the static semantics of the C programming language has historically resisted formal approaches. This is in part due to the complexity of the C language because it was designed to perform both low (bitwise) and high level (math) level calculations. Furthermore, its syntax makes C highly expressive in the sense that the same task can ofter be expressed in many different ways. Despite this, there have been a few notable attempts to formalize the semantics, albeit mainly operational, of C. This chapter is a review of the previous attempts to do so.

5.1 Evolving Algebras

One of the first attempts to formalize C was by Gurevich and Huggins [42] and their use of evolving algebras applied to the ANSI Standard described in Kernighan and Ritchie [65] to develop an operational semantics. An evolving algebra $\mathcal{A}$ is an abstract state machine. Each signature of $\mathcal{A}$ is finite collection of function names that individually have a fixed arity. A state of $\mathcal{A}$ is a set, the super-universe, and interpretations of the function names in the signature called the basic functions of the state. The super-universe does not change as $\mathcal{A}$ evolves.

A basic function of arity r has r operations on the super-universe. The super-universe contains distinct elements such as true, false, and undef (undefined) which permits the use of partial functions. A program of $\mathcal{A}$ is a finite collection of transition rules taking the form

$$\textbf{if } t_0 \textbf{ then } f(t_1 \ldots t_r) \ := \ t_{r+1} \textbf{ endif} \tag{5.1}$$

where t_0, $f(t_1 \ldots t_r)$ and t_{r+1} are closed terms, i.e., without free variables in $\mathcal{A}$. The expression above means evaluate all the terms t_i in the given state; if t_0 evaluates to true, then change the value of the basic function f at the value of the tuple $(t_1 \ldots t_r)$ to the value of t_{r+1}. Otherwise, do nothing.

Gurevich and Huggins showed that evolving algebras could be applied to program written in C because they saw that programs were merely a set of rules. Four algebras were developed to model statements, expressions, memory allocation and initialization, and function invocation and return. Although errors such as division by zero or de-referencing a pointer to an invalid address admittedly stalled the evolving algebra to the error state forever, error handling was generally ignored.

5.2 An Abstract Dynamic Semantics for C

Norrish [94] presents an abstract dynamic semantics for C to prove that a language as complicated C can be formalized and could be modeled in the mechanized theorem prover HOL [40]. The HOL semantics used was called Cholera and the C Standard was C90 [52].

Dynamic semantics does not address the static issue of type correctness. Rather, it tries to abstract memory changes. Because the actions of side effects are the sole cause of memory changes in C, the semantics presented take an "arrow" relation form

$$\langle\, v_0, \sigma_0 \,\rangle \rightarrow \langle\, v, \sigma \,\rangle, \tag{5.2}$$

where σ_0 and σ are the initial and final states, and where v is the form of C syntax being defined. Each rule is defined by using one or more of the three different state related functions:

1. State independent functions such as a conversion to a memory value.
2. State functions such as addresses of variables or typing functions.
3. State modification functions such as add a side effect to memory or declare a variable with an initial value.

After building the rules in their appropriate contexts such as expressions, statements, etc., theorems were proved by mimicking the axiomatic rules of Hoare [46] such as the

assertion about the partial correctness of S in the triple $\{P\}$ S $\{Q\}$ that states if S is executed in a state where P holds and if S terminates, then the resulting state Q is true.

Omitted from the semantics presented here was the library, the `switch` and `goto` statements, qualified types, union type, string constants, and bit-fields within `struct`s.

5.3 An Operational and Denotational Typing Semantics for C

Perhaps the most ambitious and comprehensive typing semantics of the C language was presented in the dissertation by Papaspyrou [95]. The 1998 work offered both a dynamic and static typing semantics for the standard C90 [52].

It was a work heavily steeped category theory [99, 38, 100, 7, 9] and the use of monads [9, 82]. For example, data types found in memory and associated to program constructs were sorted by the usual base ("data") and object types. However, these types were categorized much further to include:

1. Qualified and unqualified versions of each data and array type.
2. Function types including return and parameter types.
3. Denotable types that are object and function types that are associated with an identifier.
4. member types of structures and unions including both object types and bit-fields which may or may not be qualified (bit-fields are one of three integer types: int, signed-int, and unsigned int).
5. Value types (r-values) associated with data and function types.
6. Identifier types that denote if the data type is an object, a typedef, or an enumeration constant.
7. Phrase types that are associated to program segments such as expressions, statements, and declarations and their initializations.

The work was intended to support a Haskell based abstract interpreter of the operational semantics of C to define an execution model. According to Papaspyrou [95],

Haskell is a strongly-typed lazy functional language [50, 97]. Haskell was chosen because it "provides non-strict semantics, a static polymorphic type system, algebraic data types, modules, monads and monadic I/O and a rich system of primitive data types."

5.4 A Formalism of C++

Reis and Stroustrup [108] took up the task to formalize C++ [121], a daunting challenge based on the fact that C++ is rooted in C. They begin with defining an abstract type system (Table 5.1) and then apply the type system to expressions, declarations, and statements. Finally, the type system was applied to a set of typing rules to account for implicit conversions and function overloading. Conceptually, expression operators were treated the same as other user defined function applications.

Their formalism was developed in line with a minimal representation of C++ code to clarify distinctions and interactions between *overloading*, *template partial specialization*, and *overriding*; three concepts considered to be the most confusing in literature yet key to C++'s template system.

5.5 In Summary

From the literature, it appears that Reis and Stroustrup [108] are correct in stating that the static semantics of the C programming language has historically resisted formal approaches. Most attempts in formalization have been dynamic in nature by showing the operational semantics of C [42, 94]. A static typing semantics was presented by Papaspyrou [95]. Except for Reis and Stroustrup [108] who offered a formal typing semantics of the closely related language C++, all other works reviewed in this chapter were applied to the standards C90 [52] and earlier. The static typing semantics presented in this dissertation is applied to C99 [53].

Table 5.1: Abstract syntax of the C++ type system [108]

$\tau ::=$		*types*
	$\mathtt{t}$	*type constants*
	$\mid \alpha$	*type variables*
	$\mid \mathit{ref}(\tau)$	*reference types*
	$\mid \mathit{ptr}(\tau)$	*pointer type*
	$\mid \mathit{const}(\tau)$	`const`-*qualified types*
	$\mid \mathit{volatile}(\tau)$	`volatile`-*qualified types*
	$\mid \mathit{array}(\tau, c)$	*array types with known bound*
	$\mid \mathit{array}(\tau, \varepsilon)$	*array types with unknown bound*
	$\mid (\tau_1, ..., \tau_n) \rightarrow \tau$	*function types*
	$\mid \Pi[\![\vec{p} : \mathbb{T}^n]\!] \rightharpoonup \tau$	*type template*
	$\mid \chi[\gamma_1, ..., \gamma_n]$	*type template instantiations*
$\mathbb{T} ::=$		*generalized types*
	τ	*ordinary types*
	$\mid \natural$	*type of ordinary type*
	$\mid \mathbb{T}_1 \times ... \times \mathbb{T}_n \rightharpoonup \mathbb{T}$	*template type*
$\gamma ::=$		*compile-time entities*
	c	*constant expressions*
	$\mid \tau$	*types*
	$\mid \chi$	*named type templates*

Chapter 6: C Type Systems

According to Cardelli [17], the formalization of a type system for any given programming language essentially requires the formalization of the entire language. However, the first step in this process is describing the syntax of types for the language. This chapter examines the C type system and formally presents a syntax of its type specifications.

6.1 Introduction

A dictionary, such as [103], may define type as

> "**type** (tĩp) *n.*, *v.*, **typed**, **typ·ing**. −*n.* **1**. a class, group, or category of things or persons sharing one or more characteristics: ..."

Plotkin [101] said types provide the true meaning to any given expression. Plotkin's truism holds for the expressions found in both computer programs and mathematics. Moreover, Cardelli [17] states that the fundamental purpose of type systems in programming languages is to prevent the occurrence of execution errors during program runtime. As such, an effective and safe type system should have three basic properties:

1. A type system should be decidedly verifiable through the use of one or more type checking algorithms.
2. A type system should be "self-evident."
3. A type system should be enforceable, routinely verifying type declarations through a multitude of static (compile time) and dynamic (runtime) checks.

Apart from performing type casting based on the conversion rules such as integer promotions and usual arithmetic promotions, on a limited basis during compile time, the type system of C fails to meet these properties. Therefore, C is not type safe.

According to Cardelli [17], a formal type system is a mathematical description of the informal type system of a programming language described in its manual (standard). For example, the C type system is defined in §6.2.5 of the standard [52, 53, 54] by 26

informal rules and two examples. Together the rules and examples attempt to introduce all members of the type system and their relational hierarchy.

In a C computational environment, the data types take on a variety of shapes and sizes according to their intended purposes. Some are designed to simply hold a binary (base-2) number, to represent value, an instruction, or a memory address. Others may be complex collections of like and/or disparate data types to facilitate the generation of character strings or enable record keeping. While others are specialized to represent void or null conditions.

6.2 Syntax of C Types

Unlike the formal presentation of the syntax for expressions and statements, the syntax of C types are informally specified in §6.2.5 of the standard. Accordingly,

> "*the meaning of a value stored in an object or returned by a function is determined by the type of the expression used to access it.*"

Each type in the universe of C types belongs to one of the three general type categories: object-type, function-type, or incomplete-type (Fig 6.1). The complete syntax of type specifications of C types is presented in BNF to show the typing relationships can be reviewed in Appendix A.

Definition 6.1. An *object-type* fully describes data objects, for example, structures, arrays, and the base types such as integers and rational numbers.

Definition 6.2. A *function-type* describes the return type of functions. The parameter types of a function are generally object types are not considered in the determination of a function's type.

Definition 6.3. An *incomplete-type* can be either a void pointer or an declared object-type that is not fully defined, such as an array without a declared size.

$\langle$ *c-type* $\rangle$ ::= $\langle$ *object-type* $\rangle$ | $\langle$ *function-type* $\rangle$ | $\langle$ *incomplete-type* $\rangle$

Figure 6.1: The universe of C types

```
⟨ identifier ⟩  ::=  ⟨ nondigit
                 |   ⟨ identifier ⟩ ⟨ nondigit ⟩
                 |   ⟨ identifier ⟩ ⟨ digit ⟩
⟨ non-digit ⟩    |   a | b | c | d | e | f | g | h | i | j | k | l | m
                 |   n | o | p | q | r | s | t | u | v | w | x | y | z
                 |   A | B | C | D | E | F | G | H | I | J | K | L | M
                 |   N | O | P | Q | R | S | T | U | V | W | X | Y | Z | _
    ⟨ digit ⟩    |   0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Figure 6.2: C identifier syntax

```
keyword ::= auto       | break    | case    | char   | const
          | continue   | default  | do      | double | else
          | enum       | extern   | float   | for    | goto
          | if         | inline   | int     | long   | register
          | return     | short    | signed  | sizeof | static
          | struct     | switch   | typedef | union  | unsigned
          | void       | volatile | while   | _Bool  | _Complex
          | _Imaginary
```

Figure 6.3: `keywords`

Because C is an imperative language, all memory objects must be declared with an identifier (Fig. 6.2) and type prior to usage. Identifiers can be composed of any mix of alphabet (nondigit including the underscore) or numeric (digit) characters. However, the first character of an identifier cannot be a number. An identifier's type is assigned in a declaration statement by one or more declaration-specifier `keywords`. A `keyword` (Fig. 6.3) is a case sensitive token reserved for translation purposes and cannot be used otherwise. The syntax for declaration statements (Fig. 6.4) is described in §6.7 of the standard.

The type-specifier `keywords` (Fig. 6.5) are defined in §6.7.2 of the standard. They are used in declaration expressions to specify the interpreted type and size of a reserved memory store based on the declared object-type. At least one or more type-specifier is used to declare an object-type. It should be noted that C11 [54] introduced the atomic-type-specifier (Fig 6.6) to be used with the new header file `stdatomic.h` to support multitasking threading and atomic operations. The atomic-type-specifier has two constraints:

1. They cannot be used if the implementation does not support atomic types.
2. The type name in an atomic-type-specifier cannot refer to an array-type, a function-type, an `atomic` type, or a qualified type.

6.3 Object Types

The object-type includes all types that fully describes data objects and is divided into one of three general type categories: scalar-type, aggregate-type, and union-type (Fig. 6.7).

6.3.1 Scalar Types

A scalar-type (Fig. 6.8) is either an arithmetic-type (Fig. 6.9) or a pointer-type (Fig. 6.11).

```
< declaration >             ::= < declaration-specifiers > < init-declarator-list_opt >
< declaration-specifiers >  ::= < storage-class-specifier > < declaration-specifiers_opt >
                              | < type-specifier > < declaration-specifiers_opt >
                              | < type-qualifier > < declaration-specifiers_opt >
                              | < function-specifier > < declaration-specifier_opt >
< init-declarator-list >    ::= < init-declarator >
                              | < init-declarator-list > , < init-declarator >
< init-declarator >         ::= < declarator >
                              | < declarator > = < initializer >
< declarator >              ::= < pointer_opt > < direct-declarator >
< direct-declarator >       ::= < identifer >
                              | ( < declarator > )
                              | < direct-declarator >
                                  [ < type-qualifier-list_opt >
                                    < assignment-expression_opt > ]
                              | < direct-declarator >
                                  [ static < type-qualifier-list_opt >
                                    < assignment-expression_opt > ]
                              | < direct-declarator >
                                  [ < type-qualifier-list > static
                                    < assignment-expression > ]
                              | < direct-declarator >
                                  [ < type-qualifier-list_opt > * ]
                              | < direct-declarator > ( < parameter-type-list > )
                              | < direct-declarator > ( < identifier-list_opt > )
< pointer >                 ::= * < type-qualifier-list_opt >
                              | * < type-qualifier-list_opt > < pointer >
< parameter-type-list >     ::= < parameter-list >
                              | < parameter-list > , ...
< parameter-list >          ::= < parameter-declaration >
                              | < parameter-list > , < parameter-declaration >
< parameter-declaration >   ::= < declaration-specifiers > < declarator >
                              | < declartion-specifiers > < abstract-declarator_opt >
< identifier-list >         ::= < identifier >
                              | < idenifier-list > , < identifier >
```

Figure 6.4: C declaration syntax

$$
\begin{aligned}
\langle \textit{ type-specifiers } \rangle \quad ::= \quad & \texttt{void} \\
| \quad & \texttt{char} \\
| \quad & \texttt{short} \\
| \quad & \texttt{int} \\
| \quad & \texttt{long} \\
| \quad & \texttt{float} \\
| \quad & \texttt{double} \\
| \quad & \texttt{signed} \\
| \quad & \texttt{unsigned} \\
| \quad & \texttt{_Bool} \\
| \quad & \texttt{_Complex} \\
| \quad & \texttt{_Imaginary} \\
| \quad & \langle \textit{ struct-or-union-specifier } \rangle \\
| \quad & \langle \textit{ enum-specifier } \rangle \\
| \quad & \langle \textit{ typedef-name } \rangle
\end{aligned}
$$

Figure 6.5: Type-specifier `keywords`

$$\langle \textit{ atomic-type-specifier } \rangle \quad ::= \quad \texttt{_Atomic} \ (\ \textit{type-name} \ )$$

Figure 6.6: Atomic-type-specifier syntax

$$\langle \textit{ object-type } \rangle \quad ::= \quad \langle \textit{ scalar-type } \rangle \quad | \quad \langle \textit{ aggregate-type } \rangle \quad | \quad \langle \textit{ union-type } \rangle$$

Figure 6.7: Object-type

$$\langle \textit{ scalar-type } \rangle \quad ::= \quad \langle \textit{ arithmetic-type } \rangle \quad | \quad \langle \textit{ pointer-type } \rangle$$

Figure 6.8: Scalar-type

```
⟨ arithmetic-type ⟩                  ::=  ⟨ std-integer-type ⟩
                                      |   ⟨ floating-type ⟩
                                      |   ⟨ enumerated-type ⟩
⟨ standard-integer-type ⟩            ::=  char
                                      |   ⟨ standard-signed-integer-type ⟩
                                      |   ⟨ standard-unsigned-integer-type ⟩
⟨ standard-signed-integer-type ⟩     ::=  signed char
                                      |   signed short int
                                      |   signed int
                                      |   signed long int
                                      |   signed long long int
⟨ standard-unsigned-integer-type ⟩   ::=  _Bool
                                      |   unsigned char
                                      |   unsigned short int
                                      |   unsigned int
                                      |   unsigned long int
                                      |   unsigned long long int
⟨ floating-type ⟩                    ::=  ⟨ real-float-type ⟩
                                      |   ⟨ complex-type ⟩
⟨ real-float-type ⟩                  ::=  float
                                      |   double
                                      |   long double
⟨ complex-type ⟩                     ::=  float _Complex
                                      |   double _Complex
                                      |   long _Complex
⟨ enumerated-type ⟩                   |   ⟨ enumeration ⟩
⟨ enumeration ⟩                       |   ⟨ standard-integer-type ⟩ . . .
                                            ⟨ standard-integer-type ⟩
```

Figure 6.9: Arithmetic-type

6.3.1.1 Arithmetic Types

An arithmetic-type is one of the base memory types such as an integer or a floating type. An arithmetic-type is declared with one or more type specifier `keyword`(s). If more than one `keyword` is required to declare an arithmetic-type, the `keyword`s can be used in any order. For example, the standard-signed-integer-type `long int` may be declared in any of the following semantically equivalent `keyword` combinations:

- `long`
- `signed long`

- `long signed`
- `long int`
- `int long`
- `signed long int`
- `long signed int`
- `long int signed`
- `signed int long`
- `int long signed`
- `int signed long`

Since the standard-integer-types are abstractions of the machine integers, the standard defines them in terms of bits and bytes.

Definition 6.4. A *bit* is an unit of data storage in an execution environment large enough to hold one of two values: `0` or `1`.

Definition 6.5. A *byte* is an addressable unit of data storage large enough to hold any member of the basic character set of the execution environment.

The precision or range of values a machine type can hold is determined by the number of bits allocated for precision. For example, an unsigned machine integer type with n precision bits can hold the value $(2^n - 1)$. The least significant bit (LSB) whose precision range is limited to $(0, 1)$, is called the low-order bit; and, the most significant bit (MSB) or magnitude bit is called the high-order bit. A machine integer type can either represent strictly positive values or represent both positive and negative values. The former are unsigned integer types and the latter signed integer types. For unsigned integer types, `0` is regarded as a positive value. For signed integer types, `0` may be either positive or negative, depending on the implementation.

For signed integers in C, the MSB loses its function as a precision bit to become a sign bit. If the MSB is set to `1`, the value of the signed integer is a negative number. Otherwise, if the MSB is unset (`0`), the value of the integer is a positive number. By

losing a precision bit to indicate sign, signed integer types with n precision bits have the value range $(-2^{(n-1)}, \ldots, 0, \ldots, 2^{(n-1)} - 1)$.

6.3.1.2 `char` Type

According to §6.2.5 of the standard, the type `char` is an object large enough to store any member of the basic execution character set, such that when stored as a `char`, the value of the execution character is "guaranteed" to be positive. Both the value and the sign of all other characters stored as a `char` object are implementation-defined.

6.3.1.3 Extended Integer Types

The members of the extended-integer-types, such as `int#_t`, `int_least#_t`, `int _fast#_t`, `intptr_t`, and `intmax_t`, are defined in `stddef.h` and in `stdint.h`. They were intentionally omitted from the arithmetic-type syntax (Fig. 6.9) because they are considered to be out of scope of this dissertation. Additional integer types defined in `stddef.h`, such as `ptrdiff_t`, `size_t`, and `wchar_t`, were also excluded for the same reason.

6.3.1.4 Enumerated Types

An enumerated-type is comprised of a set of named integer constants. According to §6.7.2.2 of the standard, the value of each integer constant must be value representable by the type `int` and compatible with either a `char`, a standard-signed-integer-type, or a standard-unsigned-integer-type. The choice of type compatibility is implementation-defined.

```
⟨ enum-specifier ⟩  ::=  enum ⟨ identifier_opt ⟩ { ⟨ enumeration-list ⟩ }
                     |   enum ⟨ identifier_opt ⟩ { ⟨ enumeration-list ⟩ , }
                     |   enum ⟨ identifier ⟩
⟨ enumerator-list ⟩ ::=  ⟨ enumerator ⟩
                     |   ⟨ enumeration-list ⟩ , ⟨ enumerator ⟩
⟨ enumerator ⟩      ::=  ⟨ enumeration-constant ⟩
                     |   ⟨ enumeration-constant ⟩ = ⟨ constant-expression ⟩
```

Figure 6.10: Enum-specifier syntax

6.3.1.5 Floating Types

The three real-floating-type types (`float`, `double`, and `long double`) and the three complex-type types (`_Complex`, `double _Complex`, and `long double _Complex`) are collectively called the floating-types. According to §5.2.4.2.2 of the standard, a floating-type is characterized by the following parameters:

- s — a sign (± 1)
- b — the base (radix) of the exponent representation (an integer > 1)
- e — an exponent (an integer between a minimum e_{min} and a maximum e_{max})
- p — precision (the number of base-b digits in the significand)
- f_k — nonnegative integers less than b (the significand digits)

Floating point arithmetic is implementation defined, however, each implementation must adhere to the following float model.

$$x = sb^e \sum_{k=1}^{p} f_k b^{-k}, \quad e_{min} \leq e \leq e_{max} \tag{6.1}$$

The complex-type types have the same representation and alignment requirements as an array-type of two elements of the corresponding real type. The first element is the real part and the second element is the imaginary part of the complex number. According to §6.7.2 of the standard, an implementation is not required to provide for complex types.

⟨ *pointer-type* ⟩	::=	⟨ *pointer* ⟩
⟨ *pointer* ⟩	|	⟨ *object-type* ⟩
	|	⟨ *function-type* ⟩
	|	⟨ *incomplete-type* ⟩

Figure 6.11: Pointer-type

⟨ *aggregate-type* ⟩	::=	⟨ *array-type* ⟩
	|	⟨ *structure-type* ⟩
⟨ *array-type* ⟩	::=	⟨ *element-type* ⟩
⟨ *element-type* ⟩	::=	⟨ *object-type* ⟩
⟨ *structure-type* ⟩	::=	⟨ *member-type* ⟩
⟨ *member-type* ⟩	::=	⟨ *object-type* ⟩

Figure 6.12: Aggregate-type

6.3.1.6 Pointer Type

The pointer-type pointer is defined in §6.7.5.1 in the standard (Pointer declarators). According to the standard [53], a pointer type is created in a declaration "`T D1`" of the form

$$* \ \textit{type-qualifier-list}_{opt} \ \textit{identifier}$$

where the type specified for identifier in the declaration is a "derived-declarator-type-list `T`" and the type specified for identifier is a "derived-declarator-type-list type-qualifier-list pointer to `T`".

6.3.2 Aggregate Types

Members of the aggregate-type category are the array-type and struct-type data structures (Fig. 6.12). Both types reserve a contiguous memory store large enough to contain all of their respective declared types of element-type and of member-type.

⟨ *struct-or-union-specifier* ⟩	::=	⟨ *struct-or-union* ⟩
		⟨ $identifier_{opt}$ ⟩ { ⟨ *struct-declaration-list* ⟩ }
	|	⟨ *struct-or-union* ⟩ ⟨ *identifier* ⟩
⟨ *struct-or-union* ⟩	::=	`struct`
	|	`union`
⟨ *struct-declaration-list* ⟩	::=	⟨ *struct-declaration* ⟩
	|	⟨ *struct-declaration-list* ⟩ ⟨ *struct-declaration* ⟩
⟨ *struct-declaration* ⟩	::=	⟨ *specifier-qualifier-list* ⟩ ⟨ *struct-declarator-list* ⟩ `;`
⟨ *specifier-qualifier-list* ⟩	::=	⟨ *type-specifier* ⟩ ⟨ $specifier\text{-}qualifier\text{-}list_{opt}$ ⟩
	|	⟨ *type-qualifier* ⟩ ⟨ $specifier\text{-}qualifier\text{-}list_{opt}$ ⟩
⟨ *struct-declarator-list* ⟩	::=	⟨ *struct-declarator* ⟩
	|	⟨ *struct-declarator-list* ⟩ `,` ⟨ *struct-declarator* ⟩
⟨ *struct-declarator* ⟩	::=	⟨ *declarator* ⟩
	|	⟨ $declarator_{opt}$ ⟩ `:` ⟨ *constant-expression* ⟩

Figure 6.13: Struct-or-union-specifier syntax

6.3.2.1 Array Type

The type of an array generally corresponds to the converted type of its set of elements. An element-type is often an arithmetic-type. However, an element-type may be another array-type, struct-type, or union-type.

6.3.2.2 `struct` Type

A struct-type is declared with a struct-or-union-specifier (Fig 6.13) as defined in §6.7.2.1 of the standard. A `struct` may also be comprised of diverse members of the member-type including members of struct-type, array-type, arithmetic-type, and bit-field.

According to the standard (§6.7.2.1, constraint 4.), a bit-field shall have a type that is a qualified or unqualified version of `_Bool`, `signed int`, `unsigned int`, or some other implementation-defined type. The only difference between a `bit-field` and the members of standard-integer-type is that a `bit-field` bit-wise precision is programmer defined and its precision may be as small as `1` (`_Bool`) bit or as large as the precision of an `unsigned long long int`. Collectively, the member types of struct-type are members

$$\langle \textit{ union-type } \rangle \quad ::= \quad \langle \textit{ member-type } \rangle$$

Figure 6.14: Union-type

$$\begin{array}{rcl} \langle \textit{ function-type } \rangle & ::= & \langle \textit{ return-type } \rangle \\ \langle \textit{ return-type } \rangle & ::= & \langle \textit{ object-type } \rangle \quad | \quad \texttt{void} \end{array}$$

Figure 6.15: Function-type

of object-type. A `struct` declaration reserves enough memory space to accommodate all of its members.

6.4 `union` Type

A union-type is also declared with a struct-or-union-specifier (Fig 6.13) as defined in §6.7.2.1 of the standard. The union-type (Fig. 6.14) appears to be similar to a `struct` type in the sense that it can also have the members of member-type. However, the union-type is different because the memory space reserved by an `union` declaration is only as large as that is required by the largest type of its member-type.

6.5 Function Types

Despite the fact that functions can have 0 to many arguments or parameters of diverse types, a function-type is determined by the object-type of a function's `return` type (Fig. 6.15). However, the `return` type cannot be an array-type object. If the function has no `return`, then the function `return` type is incomplete-type `void`.

6.5.1 Function Specifiers

The function-specifier is defined in §6.7.4 of the standard (Fig. 6.16). A function declared with the keyword `inline` is an *inline function*. The purpose of an inline function is to make the fastest calls to that function. However, the speed of the function calls is implementation defined. The function-specifier keyword `inline` has three constraints:

⟨ *function-specifier* ⟩ ::= `inline`

Figure 6.16: Function-specifier syntax

⟨ *incomplete-type* ⟩ ::= `void`
| ⟨ *array-type* (of unknown size) ⟩
| ⟨ *structure-type* (of unknown content) ⟩
| ⟨ *union-type* (of unknown content) ⟩

Figure 6.17: Incomplete-type

1. `inline` can only be used in the declaration of a function.
2. An `inline` definition of a function with external linkage cannot contain a definition of a modifiable object with static storage duration (see § 6.7.2 of this chapter), and cannot contain a reference to an identifier with internal linkage.
3. In a hosted environment, `inline` cannot be used in a declaration of `main`.

A hosted environment is any third party application, such as a virtual machine, that holds data and executes its own programs.

6.6 Incomplete Types

An incomplete-type is either the type `void` or it is an aggregate-type or union-type that is not fully defined during its declaration (Fig. 6.17). For example, an array may be declared without having its size fully defined.

6.7 Type Qualifiers and Storage Class Specifiers

The typing syntax just presented represents the unqualified C types without storage class assignments. However, the syntax can be replicated to include qualified types and storage class specifiers by adding the keywords of each respectively.

Listing 6.1: `const` Declaration

```
const int INT_MAX;
INT_MAX = 2147483647; // assuming 64 bit platform
```

6.7.1 Type Qualifiers

The keyword members of type-qualifier are defined in §6.7.3 of the C standard. A type-qualifier (Fig. 6.18) places usage restrictions on the declared object-type reserved in memory. For example, the declaration statement in Listing 6.1 reserves a memory store named `INT_MAX`. Because the type-qualifier `const` was used, the contents of the store cannot be altered once it is given a value in an assignment expression (Listing 6.1).

Use of a type-qualifier in a declaration statement is optional. If used, one or more may be used and as is the case with the type-specifier keywords, type-qualifier keywords may be used in any order. Semantically, however, type-qualifier keywords can conflict with one another. For example, the keyword `volatile` defines an object to be changeable internally or externally to the scope of the program for which it appears, and is diametrically opposed to the keyword `const`. The qualifier `restrict` can only be used on pointers to promote program optimization. While the use of two or all of the type-qualifier keywords within a single declaration is syntactically correct according to the standard, most compilers will translate the first type-qualifier keyword read and ignore the rest because of their conflicting semantics.

C11 introduced the additional type-qualifier keyword `_Atomic` to be used with the new header file `stdatomic.h` to support the multitasking threading and atomic operations. The type-qualifier `_Atomic` has two usage constraints:

1. Types other than pointer-type referencing an object-type cannot be `restrict` qualified.
2. The `_Atomic` qualifier cannot be applied to an array-type or a function-type.

⟨ *type-qualifier* ⟩	::=	⟨ *qualifier-list* ... *qualifier_list* ⟩
⟨ *qualifier-list* ⟩	::=	`const`
	|	`volatile`
	|	`restrict`

Figure 6.18: Type-qualifiers

⟨ *storage-class-specifier* ⟩	::=	`typedef`	|	`extern`		
	|	`static`	|	`auto`	|	`register`

Figure 6.19: Storage-class-specifiers

The keyword `_Atomic` may be used semantically in tandem with any other type-qualifier keyword on a C11 compliant compiler.

6.7.2 Storage Class Specifiers

The keywords of storage-class-specifier are defined in §6.7.1 of the standard (Fig. 6.19). The storage-class-specifier keywords instruct a compiler about of the storage related specification of an object or a function being declared, such as its duration, visibility, and/or storage.

The duration refers to the lifetime of an object or a function. The lifetime is either global (keyword `static`) or local (keyword `auto` - meaning automatic). A global lifetime is one that exists throughout the execution of a program. For example, functions have global lifetimes. An `auto` variable has a local lifetime and its storage space is allocated whenever the execution of the program enters the block in which the `auto` variable is defined. Visibility refers to whether an object or function was originally declared locally or externally (keyword `extern`) to the program. Finally, the storage directive `register` is a request to create a register object.

Like the type-qualifier keywords, use of a storage-class-specifier keyword in a declaration expression is optional. Unlike the type-specifier and the type-qualifier keywords where more than one can be used in a declaration, at most one storage-class-specifier keyword can be used. The storage-class-specifier keyword `typedef` is the exception to the

Listing 6.2: `typedef` declaration

```
typedef unsigned long uint_32t;
uint_32t x = 100000;
```

$$\langle \textit{ typedef-name } \rangle \quad ::= \quad \langle \textit{ identifier } \rangle$$

Figure 6.20: Typedef-name syntax

rule where the addition of at most one more storage-class-specifier keyword may follow the keyword `typdef`.

The storage-class-specifier keyword `typedef` is an anomaly among the other storage-class-specifier keywords in the sense that `typedef` does not specify a memory criteria. Instead it is used to create an alias for another object-type (Listing 6.2). The MISRA safe coding guidelines recommends that whenever an arithmetic-type is declared it should be `typedef`ed and the `typedef` name should describe the arithmetic-type [79] (Table 3.1). For example, the `typedef` name of the `unsigned long` (Listing 6.2) is `uint_32` to indicate the type is a `unsigned` 32 bit integer. According to the standard, the `typedef` keyword was placed in the storage-class-specifier set out of convenience.

6.7.2.1 `typedef` Name

The typedef-name is defined in §6.7.7 of the standard (Fig 6.20). It is the identifier that is associated with the type specified in a `typedef` declaration. Thus, the identifier becomes a derived type specifier (derived-declarator-type-list (Fig. 6.4)) in subsequent declarations where the identifier is used to declare type.

6.8 Compatible Type

A compatible type is defined in §6.2.7, §6.7.2 (for type specifiers), §6.7.3 (for type qualifiers), and §6.7.5 (for declarators). In §6.2.7, two types are said to be compatible if the two types are the same. Two structure, union, or enumerated types declared in sepa-

rate translation units are the same if their tags and members satisfy the following two requirements:

1. If one type is declared with a tag, then both types must have the same tag.
2. For two complete structure, `union` or enumerated types, the following conditions must be met:
   (a) There has to a be a one-to-one correspondence between the member types such that each pair has compatible type and identical names.
   (b) Members of two structures have to be declared in the same order.
   (c) For both structures and unions, corresponding bit-fields must have the same width.
   (d) For two enumerations, corresponding members must have the same value.

Furthermore, all declarations to the same object or function are of compatible type.

6.9 Composite Type

According to §6.2.7 of the standard, a composite type is constructed from two compatible types if the following four conditions are met and can be applied recursively:

1. For array-types, if one array has a known constant size, the composite type is an array of the known constant size; otherwise, if one array has a variable length size, the composite type also has a variable length size.
2. If only one type is a function type with a parameter type list (a function prototype), the composite type is a function prototype with the parameter type list.
3. For two function types with parameter type lists, the type of each parameter in the composite parameter type list is the composite type of the corresponding parameters.

6.10 lvalue and rvalue

They final types of note are that of the lvalue and the rvalue. According to §6.3.2.1 of the standard, a lvalue (location value) is an expression that designates an object-type or an incomplete-type other than the type `void`. As a memory location value, the lvalue type is that of an integer. A modifiable lvalue is any lvalue other the lvalue of an array-type, an incomplete-type, a `const`-qualified type, or a member-type of either a structure-type or an union-type.

A rvalue is the value of a data store or an expression. For an assignment expression, the rvalue is the right hand side (RHS) of assignment operator (`=`). The rvalue has the type of the data store or the expression.

Chapter 7: Formalizing C Expressions and Statements

Once the syntax of type specifications for a language is formalized, one can begin to formalize the remaining constructs of the language. In this chapter, the formalization of the static typing semantics of C expressions and statements is presented. In addition, the chapter addresses the formal type safety requirements that cannot expressed in the static typing semantics.

7.1 Introduction

The standard has somewhat formalized the syntax of both expressions and statements in BNF [8, 91] to insure consistent and identical parse trees among the various compilers. On the other hand, the operational and static typing semantics of C expressions and statements are not formalized. Instead, the standard uses natural language to define the semantics. To be sure, we can agree upon the operational semantics for most of the operators contained within expressions and statements. For example, the operational semantics of the binary addition operator + is the summation of its operands. However, the natural language presentation of the static typing semantics for both expressions and statements are ambiguous and left open to interpretation. The goal of this chapter is to remove the ambiguity of the static typing semantics

Because C allows programmers to write many expressions containing operands of mixed data types and the disjoint operand types are usually normalized during compile time, the C type casting rules are an essential element of the static typing semantics for expressions and statements. As such, we need to review the C type casting rules before presenting the formalization process.

7.2 C Type Casting Rules

The criteria for the compile time type coercion process is based on sub-typing. The numerous coercion rules are found in §6.3 of the standard.

7.2.1 Integer Conversion Rank

Every integer type has an integer conversion rank based on its bit width and that rank determines how the integer types are coerced during compile time. The following integer ranking system is found in §6.3.1.1 of the standard:

- No two signed integer types regardless of their precision shall have the same rank
- The rank of a signed integer type is greater than the rank of signed integer type of lesser precision
- The rank of an unsigned integer type shall equal the rank of a corresponding signed integer type
- The rank of any standard integer type shall be greater than any extended integer type with the same precision
- The rank of `char` is equal to the rank of `signed char` and `unsigned char`
- The rank of an enumerated type is equal to the rank of its compatible integer type
- The rank of any extended signed integer type with the same precision of another extended signed integer type is implementation defined
- Integer ranks are transitive in the sense that for all integer types τ_i, τ_1, τ_2, and τ_3, if τ_1 has greater rank than τ_2 and τ_2 has greater rank than τ_2, then τ_1 has greater rank than τ_3

7.2.2 Integer Promotions

The integer promotions rule states that if a `signed int` can represent all values of any integer type with less precision and a smaller rank than that of aa `signed int`, the

value of the smaller integer type is converted to a `signed int`; otherwise, the smaller integer type is converted to an `unsigned int`. The goal of integer promotions is to preserve signage as well as the value (avoiding potential overflow/underflow or signage error conditions) of the intermediate results of an expression containing smaller precision integer operands.

7.2.3 Usual Arithmetic Conversions

The goal of the usual arithmetic conversions is to find a common real type for the operands and the result of an expression. The rules comprising the usual arithmetic conversions are described in §6.3.1.8 of the standard:

1. If an operand of type `long double` exists, the other operand(s) are converted to `long double`.
    - Otherwise, if the largest corresponding real type of an operand is a `double`, the other operand is converted to a `double`
    - Otherwise, if the largest corresponding real type of an operand is a `float`, the other operand is converted to a `float`
    - Otherwise, the integer promotions are performed on the operands prior to applying the following rules:
        - If both operands have the same integer type, no further conversion is performed
        - Otherwise, if both operands have signed integer types or if both have unsigned integer types, the operand with the lesser integer conversion rank is converted to the integer type of the operand with the greater rank
        - Otherwise, if one operand has an unsigned integer type with a greater than or equal rank of the other operand of signed integer type, the signed integer operand is converted to the unsigned type

- Otherwise, if one operand has signed integer type and can fully represent the values of the other operand of unsigned integer type, the unsigned integer is converted to a signed integer
- Otherwise, both operands are converted to the unsigned integer type corresponding to the type of the signed integer type

2. If the values of floating operand and the results of floating expressions are represented in greater precision and range than what is required by the type, the types are not changed.

7.2.4 Other Conversion Rules

§6.3 of the standard contains numerous other rules that fall outside of the auspices of the integer promotions and the usual arithmetic conversion.

7.2.4.1 Conversion to Type `_Bool`

Any scalar value can be converted to `_Bool` (§6.3.1.2 of the standard). The result of the conversion is `0` if the value of the scalar type is equal to `0`. Otherwise, the result is `1`.

7.2.4.2 Conversions Between Signed and Unsigned Integers

Conversions between signed and unsigned integer types is defined in §6.3.1.3 of the standard. The three rules generally reflect the rules found in the usual arithmetic conversions with respect to integer conversions:

1. If the value of an integer type would remain unchanged if converted to another integer type other than `_Bool`, then the integer type is unchanged.
2. Otherwise, if the new type is an unsigned integer type, the value of the original integer type is converted by repeatedly adding or subtracting one more than

the maximum value that can be represented by the new unsigned type until the converted value is in the range of the new type.

3. Otherwise, if the new type is a signed integer type and the converted value cannot be represented in it; the behavior is implementation-defined.

7.2.4.3 Conversions Between Real Floating and Integers

§6.3.1.4 of the standard provides two rules with respect to conversions between real floating and integer types:

1. Whenever a value of a real floating type is converted to an integer type other than `_Bool`, the fractional part is discarded. If the integral part of a real floating type cannot be represented by the new integer type, the behavior is undefined.
2. When value of an integer type is converted to a real floating type and if the converted value can be represented exactly in the new real floating type, the value is unchanged. If the converted value is in the range of values that can be represented by real floating type, but cannot be represented exactly, the result is either the nearest higher or nearest lower representable value of the new type. The value chosen is implementation-defined. If the value being converted to a real floating type is outside of the range of values that can be represented by the new type, the behavior is undefined.

7.2.4.4 Conversions Between Real Floating Types

The two rules with respect to conversions between real floating types found in §6.3.1.5 of the standard largely reflect the usual arithmetic conversions:

1. When a `float` is promoted to a

   `double` or a `long double`, or when a `double` is promoted to a `long double`, the original value remains unchanged.

2. When either a `double` is demoted to either a `float` or a `long double` is demoted to a `double` or a `float` or if a value to be converted is of an integer type and has greater precision and range than required by the new real floating type, if the value being converted can be represented by the new floating type, the value remains unchanged. If the value being converted is in the range of values that can be represented by the new real floating type but cannot be represented exactly, the resulting value is either the nearest higher or nearest lower representable value according to rules of the underlying implementation. If the value being converted is outside of the range of values that can be represented by new real floating type, the behavior is undefined.

7.2.4.5 Conversions Between Complex Types

§6.3.1.6 of the standard states the when the value of one complex type is converted to another complex type, both the real and imaginary parts adhere to the real floating conversion rules previously stated in § 7.2.4.4 above.

7.2.4.6 Conversions Between Real and Complex Types

There are two rules in §6.3.1.7 of the standard with respect to conversions between real and complex types:

1. When a value of a real type is converted to a complex type, the real part of the complex result value is governed by the rules previously stated in §5.11.3.4 above. The imaginary part of a real to complex conversion results in zero.
2. When the value of a complex type is converted to a real type, the imaginary part of the complex value is discarded and the value of the real part is converted according to the corresponding rules found in § 7.2.4.4 above.

7.2.4.7 Conversions Involving Pointers

There are eight pointer conversion rules found in §6.3.2.3 of the standard:

1. A pointer to `void` can be converted to or from another pointer to any incomplete or object type. When converted back again, the result shall be equal to the original pointer.
2. For any qualifier q, a pointer to a non-qualified type may be converted to a pointer to a q-qualified version of the same pointed to type where the values stored in the original and converted pointer remain equal.
3. If an integer constant expression holds the value `0` and is cast to type `void *`, the result is called a null pointer constant. If a null pointer constant is converted to a pointer type, the resulting pointer is called a null pointer and is guaranteed to compare unequal to a pointer to any other object or function.
4. Conversion of a null pointer to another pointer type produces a null pointer of the same type as the original null pointer and the two null pointers shall compare equal.
5. An integer type can be converted to any pointer type. However, the resulting behavior is implementation-defined.
6. Any pointer type can be converted to an integer type. The result is implementation-defined. If the result cannot be represented by the integer type, the behavior is undefined.
7. A pointer to an object or incomplete type may be converted to another respective object or incomplete type. If the resulting pointer is not correctly aligned for the pointed-to type, the behavior is undefined. Otherwise, when converted back, the result shall compare equal to the original pointer. When a pointer to an object is converted to a pointer to a character type, the result points to the lowest addressed byte of the object. Successive increments of the result, up to the size of the object produces pointers to the remaining bytes of the object.

8. A pointer to a function of one type may be converted to a pointer to a function of another type and back again such that the result will compare equal to the original pointer. If a converted pointer is used to call a function whose type is not compatible with the pointed-to type, the behavior is undefined.

7.3 Expressions

C expressions are the "work horses" of the language. They perform calculations, make comparisons, and flip bits. According to §6.5 of the standard, an expression is a sequence of operators and operands that specifies computation, designates an object or a function, or generates side effects, or any combination thereof. However, a value may be modified at most once between sequence points during the evaluation of an expression. All sequence points are listed in Annex C of the standard.

Each expression form, whether it be unary, binary, or ternary, has its own syntax rules. Except for function calls, the logical `AND`, the logical `OR`, the conditional, and comma operator expressions, the order of evaluation of the subexpressions and the order in which side effects take place are unspecified. That said, expression operators have and adhere to an order of precedence. For example, the mathematical operators follow the traditional precedence rules found in mathematics unless otherwise specified by the use of enclosing parenthesis. For all C expressions, the operands to each operator have a specific associativity (Table 7.1). In Table 7.1, the larger precedence number takes precedence over the lower precedence number. If a mathematical operation is not defined, such as a divide by `0` or if the result of an expression evaluation is not in the range of representable values for the type of the expression, the behavior is undefined.

The standard provides for seven expression forms (Fig. 7.1). Except for the primary-expressions (identifiers, literals, and parenthesized expressions), all other expressions are comprised of an operator applied to one or more operands. The effective type of an operand object is determined at the time of the object's declaration. However, each

Table 7.1: Operator order of precedence and operand associativity

Precedence	**Operator**	**Description**	**Associativity**
	`++`	Suffix increment	Left-to-right
	`–`	Suffix decrement	Left-to-right
1	`()`	Function call	Left-to-right
	`[]`	Array subscripting	Left-to-right
	`.`	Element reference	Left-to-right
	`->`	Element pointer	Left-to-right
	`++`	Prefix increment	Right-to-left
	`–`	Prefix decrement	Right-to-left
	`+`	Unary plus	Right-to-left
	`-`	Unary minus	Right-to-left
2	`!`	Logical NOT	Right-to-left
	`~`	Bitwise NOT	Right-to-left
	(*type*)	Type cast	Right-to-left
	`*`	Indirection	Right-to-left
	&	Address-of	Right-to-left
	`sizeof`	Size-of	Right-to-left
	`*`	Multiplication	Left-to-right
3	/	Division	Left-to-right
	%	Modulo	Left-to-right
4	`+`	Addition	Left-to-right
	`-`	Subtraction	Left-to-right
5	«	Bitwise left shift	Left-to-right
	»	Bitwise right shift	Left-to-right
	<	Less than	Left-to-right
6	<=	Less than or equal to	Left-to-right
	>	Greater than	Left-to-right
	>=	Greater than or equal to	Left-to-right
7	==	Equal to	Left-to-right
	!=	Not equal to	Left-to-right
8	&	Bitwise AND	Left-to-right
9	^	Bitwise XOR	Left-to-right
10	|	Bitwise OR	Left-to-right
11	&&	Logical AND	Left-to-right
12	||	Logical OR	Left-to-right
	? :	Ternary conditional	Right-to-left
	=	Simple assignment	Right-to-left
	+=	Assignment by sum	Right-to-left
	-=	Assignment by difference	Right-to-left
13	*=	Assignment by product	Right-to-left
	/=	Assignment by quotient	Right-to-left
	% =	Assignment by remainder	Right-to-left
	«=	Assignment by bitwise left shift	Right-to-left
	»=	Assignment by bitwise right shift	Right-to-left
	& =	Assignment by bitwise AND	Right-to-left
	^=	Assignment by bitwise XOR	Right-to-left
	|=	Assignment by bitwise OR	Right-to-left
14	,	Comma	Left-to-right

$$
\begin{array}{rcl}
\langle \textit{ expression } \rangle & ::= & \langle \textit{ primary-expression } \rangle \\
& | & \langle \textit{ postfix-expression } \rangle \\
& | & \langle \textit{ unary-expression } \rangle \\
& | & \langle \textit{ cast-expression } \rangle \\
& | & \langle \textit{ binary-expression } \rangle \\
& | & \langle \textit{ ternary-expression } \rangle \\
& | & \langle \textit{ assignment-expression } \rangle
\end{array}
$$

Figure 7.1: Syntax of C expressions

operator places additional typing specific constraints on its operands. For example, the operands of the modulo operator % must be of integer type. In all, there are 23 unique operand typing constraints imposed by the various operators comprising all expression forms.

7.3.1 Static Typing Semantics

To express the static typing semantics of any given expression, the typing constraints are used in conjunction with the accompanying operational semantics that define the type conversion mechanisms within the standard to create a series of judgments in the form of

STATIC TYPING SEMANTICS: TYPING INFERENCE STRUCTURE

$$
\frac{Precondition}{Conclusion} \tag{7.1}
$$

where P are the predicates or pre-conditions that must hold for the conclusion Q to be true. More specifically, each P and Q is judgment of the form

STATIC TYPING SEMANTICS: JUDGEMENT

$$
\Gamma \vdash \mathbb{A} \tag{7.2}
$$

where Γ is the static typing environment and $\mathbb{A}$ is an assertion such that Γ implies $\mathbb{A}$. A typing judgment takes the form

Static Typing Semantics: Typing Judgement

$$\Gamma \vdash \mathsf{e} : \tau \tag{7.3}$$

which asserts that e has type τ with respect to the typing environment Γ.

The typing constraint premises that an operator imposes on its operands can be expressed in a predicate function returning either true or false according to the declared type of an operand object. For example, if an operand is required to be of integer type, then the predicate function *isInteger*() (equation 7.4) can be used to type check the operand. As a predicate function, *isInteger*() is a boolean-valued function of the type $f : X \rightarrow \mathbb{B}$, where X is an arbitrary set and $\mathbb{B}$ is 2-element set, $\mathbb{B} = \{0, 1\}$, and is interpreted in truth returning, logical applications as $\mathbb{B} = \{\mathit{false}, \mathit{true}\}$. In the case of *isInteger*(), $X = \mathbf{TYPE}$, where **TYPE** is the set of all C types whose members are denoted as τ. If τ is equivalent to one of the integer types listed in equation 7.4, then *isInteger*() returns *true*. Otherwise, *isInteger*() returns *false*.

$$\begin{aligned}
&\mathit{isInteger} : \mathbf{TYPE} \rightarrow \mathbb{B}\\
&\mathit{isInteger} = (\lambda\tau.\ \mathbf{case}\ \tau\ \mathbf{of}\\
&\qquad \texttt{_BOOL} \mid \texttt{SCHAR} \mid \texttt{UCHAR} \mid \texttt{SHORT} \mid \texttt{USHORT} \mid\\
&\qquad \texttt{INT} \mid \texttt{UINT} \mid \texttt{LONG} \mid \texttt{ULONG} \mid \texttt{LLONG} \mid \texttt{ULLONG} \rightarrow \mathit{true}\\
&\qquad \mathbf{Otherwise} \rightarrow \mathit{false}).
\end{aligned} \tag{7.4}$$

The sub-type relationships from the syntax of the type specifications (Chapter 6) allows some predicate functions to be built upon the predicate functions of the sub-types of a super-type. For example, the sub-types of the super-type, arithmetic type, are the standard integer types and the real floating types. Thus, the predicate function *isArithmetic*() can be defined as in equation 7.5.

$$\begin{aligned}
&\mathit{isArithmetic} : \mathbf{TYPE} \rightarrow \mathbb{B}\\
&\mathit{isAritmetic} = (\lambda\tau.\ \mathbf{case}\ \tau\ \mathbf{of}\\
&\qquad \mathit{isInteger}(\tau) \rightarrow \mathit{true}\\
&\qquad \mathbf{Otherwise} \rightarrow \mathit{isFloat}(\tau)).
\end{aligned} \tag{7.5}$$

Other premises are constructed of functions that return a type $f : X \rightarrow \mathbf{T}$. For example consider the type conversions based on the integer promotions that state if a `signed int` can represent all values of the original integer type with less precision than that of a `signed int`, the value of the original type is converted to a `signed int`; otherwise, the original type is converted to an `unsigned int`. If the native integer type is 32-bits, then the process is straight forward (equation 7.6) as the values of all smaller bitwise signed and unsigned integer types are within the range of a signed 32-bit `signed int`.

$$
\begin{aligned}
&\mathit{intPromote} : \mathbf{TYPE} \rightarrow \mathbf{TYPE} \\
&\mathit{intPromote} = (\lambda\tau.\ \mathbf{case}\ \tau\ \mathbf{of} \\
&\quad \texttt{char} \rightarrow \texttt{signed int} \\
&\quad \texttt{signed char} \rightarrow \texttt{signed int} \\
&\quad \texttt{unsigned char} \rightarrow \texttt{signed int} \\
&\quad \texttt{signed short} \rightarrow \texttt{signed int} \\
&\quad \texttt{unsigned short} \rightarrow \texttt{signed int} \\
&\quad \mathbf{otherwise} \rightarrow \tau)
\end{aligned} \tag{7.6}
$$

However, if the native integer type is 16-bits, then an additional value check has to be added to the type `unsigned short` before its promotion to either a `signed int` or to an `unsigned int`. Although the types `unsigned short` and `signed int` have the same bitwise size, the maximum value that can be held by an `unsigned short` is 65535 and the maximum value can be held by an `signed int` is 32767. Thus, if the value of an `unsigned short` is less than or equal to 32767, then the `unsigned short` is promoted to an `signed int`. Otherwise, the `unsigned short` is promoted to an `unsigned int`.

A listing of the main predicate functions for the typing constraints and other supportive functions appear in Appendix **??**.

For a simple example, consider a modulo expression (equation 7.7) such as `a % b`. There are five preconditions that must be satisfied in the static typing semantics of modulus expressions. The operands `a` and `b` have to have a type in the type environment Γ. Type is represented by τ_1 and τ_2. Although `e` generally represents an expression, it

is used here to represent each operand because an operand might be an expression. The typing constraints placed on the operands by modulo operator (§6.5.5 of the standard) requires the them to be of integer type. In addition, the semantics also say that the usual arithmetic conversions are performed on both operands before the operation is executed. The converted type is represented by τ'.

The operational semantics of the modulus operator states that the result is the integer remainder resulting from dividing the first operand from the second operand. The conclusion states that the type of the result is the same as the converted operand type prior to the operation.

Static Typing Semantics: Modulo

$$\frac{\begin{array}{cc} \Gamma \vdash \mathsf{e}_1 : \tau_1 & \Gamma \vdash \mathsf{e}_2 : \tau_2 \\ isInteger(\tau_1) & isInteger(\tau_2) \\ \multicolumn{2}{c}{arithConv(\tau_1, \tau_2) \rightsquigarrow \tau'} \end{array}}{\Gamma \vdash \mathsf{e}_1 \ \% \ \mathsf{e}_2 : \tau'} \tag{7.7}$$

For many expressions, such as those that are arithmetic by nature, the type of the result is the converted type of the operands. However, the result type of conditional and equality expressions may be different than the converted operand types. Consider the equality expression (equation 7.8) where the first of three distinct typing constraints require both operands of the equality operators to be of arithmetic type subjected to the usual arithmetic conversions. Recall, an arithmetic type is either a floating type or a standard integer type.

The operational semantics of the "is equal" `==` operator produces `1` to represent true if the two operands are equal. Otherwise, it produces `0` to represent false if the two operands are unequal. Likewise, the "is not equal" `!=` operator produces `1` to represent true if the two operands are unequal. Otherwise, it produces `0` to represent false if the two operands are equal. The result type of both operators is of type `int`. Let `eq_op` represent { `==` | `!=` }.

Static Typing Semantics: Equality Op Constraint 1

$$\frac{\begin{array}{cc} \Gamma \vdash \mathtt{e}_1 : \tau_1 & \Gamma \vdash \mathtt{e}_2 : \tau_2 \\ isArithmetic(\tau_1) & isArithmetic(\tau_2) \\ \multicolumn{2}{c}{arithConv(\tau_1, \tau_2) \leadsto \tau'} \end{array}}{\Gamma \vdash \mathtt{e}_1 \ \mathtt{eq_op} \ \mathtt{e}_2 : \tau_{\mathtt{INT}}} \tag{7.8}$$

At times, static typing considers location values (lvalue) as opposed to result values (rvalue). Recall, a lvalue is an integer type holding the memory address of an data object. As such, the lvalue cannot be a negative value. For example, consider the postfix expression of array subscripting. §6.5.2.1 of the standard states that a postfix expression followed by an expression enclosed in square brackets is a subscripted designator of an array object. The operational semantics of the subscript operator [ ] is defined as $\mathtt{e}_1[\mathtt{e}_2]$ is identical to $(*(\mathtt{e}_1) + (\mathtt{e}_2))$ where $*$ is pointer to $\mathtt{e}_1$ which is the initial element of a named array object in memory and $\mathtt{e}_2$ is an integer designating the $\mathtt{e}_2$-th element of $\mathtt{e}_1$ counting up from zero. In equation 7.9,

Static Typing Semantics: Array Subscript

$$\frac{\Gamma \vdash *(\mathtt{e}_1 + \mathtt{e}_2) : \text{lvalue}(\mathtt{ARRAY}\tau)}{\Gamma \vdash \mathtt{e}_1[\mathtt{e}_2] : \text{lvalue}(\mathtt{ARRAY}\tau,\ n),} \tag{7.9}$$

the premise states that in the typing environment Γ, a pointer `*` to the named array object $\mathtt{e}_1$ added to $\mathtt{e}_2$ element of $\mathtt{e}_1$ is the location value of an array of type τ. If the premise holds, the conclusion is that a named array object $\mathtt{e}_1$ followed by a bracket enclosed element value $[\mathtt{e}_2]$ is also a location value of an array of type τ of a size n.

For a final example, consider the static typing semantics of variables. Arrays and other operands of named memory objects are variables associated $\triangleleft$ to an identifier I. The static typing semantics of an identifier states (equation 7.10) that if in Γ, an identifier I associated $\triangleleft$ to a data object obj and obj has a type τ, then in Γ, I also has τ.

Static Typing Semantics: Identifier

$$\frac{\Gamma \vdash I \lhd obj : \tau}{\Gamma \vdash I : \tau} \tag{7.10}$$

7.3.2 Type Safety Requirements

The static typing semantics examples described above do not express the type safety requirements of the expressions. The most basic type safety requirement is that the result of any expression can be represented by the type of the expression. Let $\tau_{\mathtt{MIN}_{value}}$ represent the smallest value that can be represented by the expression type, result_{value} represent the evaluated resultant value of the expression, and $\tau_{\mathtt{MAX}_{value}}$ represent the largest value that can be represented by the expression type. If the following condition is met

$$\tau_{\mathtt{MIN}_{value}} \leq \mathrm{result}_{value} \leq \tau_{\mathtt{MAX}_{value}}, \tag{7.11}$$

the expression evaluation is type safe.

Each expression has its own set of other type safety requirements in addition to range checking the result value to the valid values representable by the expression type. Let us reconsider the above static typing examples. In the case of the modulo expression (equation 7.7), there are four additional, primary type safety requirements: if an operand represents a data object stored in memory, then it must have a value, an operand that represents a stored data object must not be in an error condition, such as in an overflow state. The divisor cannot be zero (division by 0 is undefined) and if one of the operands is of type `unsigned int` (`UINT`), `unsigned long int` (`ULONG`), or `unsigned long long int` (`ULLONG`), then the other operand has to ≥ 0. If it is the case, however, where the other operand is the divisor of the modulo (division) expression, then it has to be > 0.

Type Safey Requirements: Modulo

$$\frac{\begin{array}{c} \mathtt{e}_1 : \text{val} \neq \mathtt{nil} \ \wedge \ \mathtt{e}_2 : \text{val} \neq \mathtt{nil} \\ \mathtt{e}_1 : \text{val} \notin \mathtt{error} \ \wedge \ \mathtt{e}_2 : \text{val} \notin \mathtt{error} \\ \mathtt{e}_2 : \text{val} \neq \mathtt{0} \\ \text{if } \tau' \in \{ \ \mathtt{UINT} \ \mathtt{ULONG} \ \mathtt{ULLONG} \ \}, \ \text{then } \mathtt{e}_1 : \text{val} \geq \mathtt{0} \ \vee \ \mathtt{e}_2 : \text{val} > \mathtt{0} \\ \tau'_{\text{MIN}_{value}} \leq \ \mathtt{e}_1 \ \% \ \mathtt{e}_2 : \text{val} \ \leq \tau'_{\text{MAX}_{value}} \end{array}}{\mathtt{e}_1 \ \% \ \mathtt{e}_2 : \text{val}_{\tau'}} \tag{7.12}$$

The type τ' in TypeSafetyRequirements equation 7.8 is the converted type of the modulo expression after the operands have been subjected to the usual arithmetic conversions prior to the execution of the statement.

For equality expressions (equation 7.8), the type safety requirements are that both operands have to be initialized and not in an error condition. In addition, if one operand is an unsigned integer type greater than `UINT`, then the other operand value must be ≥ 0. In equation 7.13, let `eq_op` represent { `==` | `!=` }.

Type Safey Semantics: Equality Op

$$\frac{\begin{array}{c} \mathtt{e}_1 : \text{val} \neq \mathtt{nil} \ \wedge \ \mathtt{e}_2 : \text{val} \neq \mathtt{nil} \\ \mathtt{e}_1 : \text{val} \notin \mathtt{error} \ \wedge \ \mathtt{e}_2 : \text{val} \notin \mathtt{error} \\ \text{if } \mathtt{e}_1 \in \{ \ \mathtt{UNIT} \ \mathtt{ULONG} \ \mathtt{ULLONG} \ \}, \ \text{then } \mathtt{e}_2 : \text{val} \geq 0 \\ \text{if } \mathtt{e}_2 \in \{ \ \mathtt{UNIT} \ \mathtt{ULONG} \ \mathtt{ULLONG} \ \}, \ \text{then } \mathtt{e}_1 : \text{val} \geq 0 \end{array}}{\mathtt{e}_1 \ \mathtt{eq_op} \ \mathtt{e}_2 : \text{val}_{\tau_{\mathtt{INT}}}} \tag{7.13}$$

Type safe array subscripting (equation 7.9) requires that value of $\mathtt{e}_2$ (the array index) to be greater than or equal to `0` and less than or equal to the declared array size minus `1`. In addition, the array size has to greater than or equal to `1`. In equation 7.14, let n represent the declared array size.

Type Safe Semantics: Array Subscript

$$\frac{\begin{array}{c} n \geq 1 \\ \mathtt{e}_2 : \text{val} \geq \mathtt{0} \ \wedge \ \mathtt{e}_2 : \text{val} \leq (n-1) \ \wedge \ n \geq 1 \end{array}}{\Gamma \vdash \mathtt{e}_1[\mathtt{e}_2] : \text{lvalue}(\mathtt{ARRAY}_\tau, \ n)} \tag{7.14}$$

Type safe usage of an identifier (equation 7.10) is based on its constraints. In the case of the initial character being a digit and in the case of an identifier matching a `keyword`, it is assumed that a compiler will catch the error. However, it is possible for a compiler to create ambiguous identifiers in the rare chance that two or more identifiers are longer than 32 characters in length and that their first 32 characters equally match. Thus, the type safety rules for identifiers follow.

Type Safey Semantics: Identifier

$$\frac{\mathit{lengthOf}(I) \ \leq \ 32}{\Gamma \vdash I : \tau} \tag{7.15}$$

The static typing semantics for all expression defined in §6.5 of the standard and their corresponding type safety rules are enumerated in Appendix **??**.

7.4 Statements

While expressions do the so called "heavy lifting," C statements ultimately decide the order in which the expression will be executed. Statements and blocks are defined in §6.8 of the standard. Statements are semantically executed in sequence unless otherwise indicated and the purpose of each statement is to specify an action to be performed. A block is a set of declarations and statements grouped into one syntactic unit. The storage duration of declarations declared within a block is as long as a program is executing within the scope of the block. Finally, the standard defines a full expression as an initializer, an expression within an expression statement, the controlling expression of a selection statement (`if` or `switch`), the controlling expression of a `while` or a `do` statement, the optional expressions of a `for` statement, and the optional expression in a `return` statement. Other than the preceding expression types, a full expression is not part of another expression or of a declaration declarator, and the end of a full expression is a sequence point.

$$
\begin{array}{rcl}
\langle \textit{ statement } \rangle & ::= & \langle \textit{ labeled-statement } \rangle \\
& | & \langle \textit{ compound-statement } \rangle \\
& | & \langle \textit{ expression-statement } \rangle \\
& | & \langle \textit{ selection-statement } \rangle \\
& | & \langle \textit{ iteration-statement } \rangle \\
& | & \langle \textit{ jump-statement } \rangle
\end{array}
$$

Figure 7.2: Syntax of C statements

Syntactically, there are six statement forms (Fig. 7.2). Because most statements contain an embedded expression to enable flow control decisions, their static typing closely mirrors that of the static typing semantics of the embedded expression. Likewise, the individual expressions contained within a statement block have the same static typing semantics as discussed in the previous section.

For example, consider the `if` and the `if else` selection statements. Both statements are defined in §6.8.4.1 of the standard. The operational semantics for the `if` and the `if else` statements state that the first sub statement contained within the scope statement is executed if the controlling expression compares unequal to `0`. In the `else` form, the second sub statement is executed if the controlling expression compares equal to `0`. If the first sub-statement is reached via a `label`, the second sub statement is not executed. For both, the controlling expression is constrained to have scalar-type.

Equation 7.16 depicts the static typing semantics of the `if` statement. The terms exp and stmt respectively represent the controlling expression and the statements contained withing the scope of `if` block. The premises of the static typing semantics state that in the typing environment Γ there is a controlling expression that is an expression of type τ'. The type is labeled τ' because the statement contained with the `if` block may have different typing requirements. The type of the controlling expression τ' has to be of scalar type. Within the `if` block there is another typing environment Γ that has a statement of type τ.

Static Typing Semantics: If Statement

$$\frac{\begin{array}{c}\Gamma \vdash \text{exp} : \tau' \\ \mathit{isScalar}(\tau') \\ \Gamma \vdash \text{stmt} : \tau\end{array}}{\Gamma \vdash \texttt{if}(\text{exp})\ \text{stmt} :\ \tau} \tag{7.16}$$

If premises hold for equation 7.16, the conclusion is that in a typing environment Γ a `if` expression is the type of the statement τ. The `if else` statement in equation 7.17 differs from the `if` statement because it contains a typing environment for the `if` block containing stmt_1 and a typing environment for the `else` block containing stmt_2.

Static Typing Semantics: If Else Statement

$$\frac{\begin{array}{c}\Gamma \vdash \text{exp} : \tau' \\ \mathit{isScalar}(\tau') \\ \Gamma \vdash \text{stmt}_1 : \tau \\ \Gamma \vdash \text{stmt}_2 : \tau\end{array}}{\Gamma \vdash \texttt{if}(\text{exp})\ \text{stmt}_1\ \texttt{else}\ \text{stmt}_2 :\ \tau} \tag{7.17}$$

If the premises hold for equation 7.17, the conclusion of the `if else` statement is the same as that of the `if` statement in the sense that it is the type τ statement it branches to. The `else` is associated with the lexically nearest preceding `if` that is allowed by the syntax.

The static typing semantics of iteration statements are similar to selection statements in the sense that there a controlling expression containing a set of statements. Consider the `while` loop statement that is defined in §6.8.1 of the Standard. The operational semantics of the `while` state that the controlling expression exp is evaluated before each statement stmt execution of the loop body. In equation 7.18, the static typing premises and conclusion are the same as the `if` statement.

Static Typing Semantics: While Statement

$$\frac{\begin{array}{c}\Gamma \vdash \text{exp} : \tau' \\ isScalar(\tau') \\ \Gamma \vdash \text{stmt} : \tau\end{array}}{\Gamma \vdash \texttt{while}(\text{exp})\ \text{stmt} : \tau} \tag{7.18}$$

For a final example, consider the jump statements as defined in §6.8.6 of the standard. A jump statement, such as `return`, causes an unconditional jump to another place in the program. As for the `return` statement, it terminates execution of the current function and returns control to its caller. If a `return` statement with an expression is executed, the value of the expression is returned to the caller as the value of the function call expression. If the expression has a type different from the return type of the function in which it appears, the value is converted as if by assignment to an object having the return type of the function. Equation 7.19 represents the static typing semantics of `return` statements with an expression stmt. Rather than having a set of premises, the typing semantics is axiomatic in the sense that in the typing environment Γ, type τ is the type of the statement stmt.

Static Typing Semantics: Return Statement 1

$$\frac{}{\Gamma \vdash \texttt{return}\ \text{stmt} : \tau} \tag{7.19}$$

A `return` statement with an expression shall not appear in a function whose return type is `void`. A `void` statement without an expression, equation 7.20 shall only appear in a function whose return type is `void`.

Static Typing Semantics: Return Void

$$\frac{}{\Gamma \vdash \texttt{return} : \texttt{VOID}} \tag{7.20}$$

As for type safe operation of statements, the safety requirements are based on the expressions held in the statements.

The typing semantics for all statements defined in §6.8 of the standard are enumerated in Appendix **??**.

Chapter 8: The Static Type Safety Analysis Tool

With this chapter, the theme of this dissertation shifts to the prototype static type safety analysis tool introduced by Krause and Alves-Foss [66]. In the following chapters, the discussion will include the design specifications and functionality of the tool.

8.1 The Fundamental Question

In the preceding chapters, the conditions that can produce type safety violations in the programming language C, such as, but not limited to the integer errors resulting from either casting or arithmetic operations, were enumerated. In addition, the syntax of the C typing specifications and the static typing semantics of C expressions and statements were formalized. Armed with this knowledge, the main question is what should the tool do?

8.2 A Naïve Approach

A language that prohibits the casting of one type to another is said to be strongly typed. We could take a similar heavy handed prohibitive approach in designing the functionality of our type safety analysis tool. That is, every expression or statement containing a binary operation with type mismatched operands would be flagged and reported as unsafe. Likewise, the tool could flag and report every instance of a casting between different types contained within the code under its review. Safe upcasting between types within the same type family, such as the `unsigned` integers, would not be immune as they too would be flagged as a type mismatch.

By taking this naïve approach with languages such as C where implicit coercions by either an integer promotion or an usual arithmetic conversion is a normal and routine operating procedure, we would produce either too many false positives or too many false negatives to be very useful. Adelsbach *et al.* [1] provides the definitions of each decision.

Definition 8.1. A *false positive* is a *false alarm* where an event that did not produce an error condition was flagged as an error condition event.

Definition 8.2. A *false negative* is a *missed error* that was not flagged as an error condition.

Arriving at a false positive is straightforward. For example, the positive value `1` belongs to all signed and unsigned integer types. Suppose we had an `unsigned int x` with the value of `1` and added it to a `long int y` also with the value of `1`. The expression `x + y` is type mismatched and would be flagged taking the strictly prohibitive approach even though the result has not violated any type safety conditions. Clearly, the value of the result in this example is `2` which can be represented by all integer types.

On the other hand, false negatives would be generated on those operations where both operands are of the same type but, the result produces one of the following integer error conditions: over/underflow or truncation error. For example, if we had a `signed char result`, a `signed char x` with the value of `10`, and a `signed char y` with the value of `120`, the expression `result = x + y` would be evaluated to `result = 130` not raise an alarm even though the evaluation had produced an overflow error.

8.3 A Better Attack Plan

Because of the potentially large number of both false positives and false negatives, a functionality design of the analysis tool that takes a strictly prohibitive approach would be impracticable. Instead, the tool should have the capacity to not only have type awareness of the data objects held in memory, but to track and record every value associated with each memory store beginning with the initial value immediately following a declaration statement and continuing through each successive value change of the store state resulting from expression and statement evaluations. Whenever there is change to a value stored in memory, then the new value should be validated against the legal range of values with

respect to the type of the data object represented at that memory location. Fortunately, any tool designed to analyze C source code can leverage the fact that C is a declarative language which means all memory data stores must be declared before usage. Although good coding practices recommend that all declarations of data objects are made at the beginning of the block in which they appear, C will allow such declarations to occur anywhere within the scope of the respective block so long as the declaration is made prior to usage of that data object.

The type safety analysis tool needs to adopt a high level abstraction of the computational process, where a computation is merely the evaluation of an initial expression followed by a sequence of 0 or more expressions until the program terminates. An expression may contain singleton items, such as left hand side (LHS) and the right hand side (RHS) in simple assignment expressions similar to `x = y;` or it may be a rather complex expression comprised of several operators and operands, a control statement, leading to one of two or more expression statements, a function call, or any mix of expressions, control statements, and function calls.

Expressions are sorted into blocks and on a granular level, a block is basically the code contained within the scope of a function, a control statement such as the `if` or `if else` statement, a loop structure such as the `for`, `while`, and `do while`, and the `for` statements. Any block may contain 0 or more internal blocks by the inclusion of internal nested control statements.

8.3.1 Influence of Literals

In traditional data flow analysis such as the intra-procedural analysis techniques, such as live variables, available expressions, reaching definitions, and very busy expressions, explained in Nielson *et al.* [93], the objective is to track all memory store changes in relationship to the influence of other memory stores in the code. In other words, how is a particular memory store influenced by the state of another memory store? Often called

Listing 8.1: When $10 \not> -10$

```
x = 10;
if(x > -10)
{
  // do something important
}
```

$$\begin{aligned} \texttt{0000 0000 0000 0000 0000 0000 0000 1010} &= \texttt{ 10} \\ \texttt{1111 1111 1111 1111 1111 1111 1111 0110} &= \texttt{-10} \end{aligned}$$

Figure 8.1: The 32-bit patterns for 10 and -10 in two's complement

constant values, literals and their influence on the data flow are generally disregarded in most data flow analysis. For example, if a variable is stored in memory store then the assignment of a literal to a variable name, such as `x = 10`, erases or resets the history of influence behind the variable `x` prior to the assignment statement because in the assignment `x = 10`, `x` is not influenced by any other memory object. When it comes to type safety, literals have a direct influence io the integrity of the values held in memory. For example, consider the implication of the C code in listing 8.1.

On face value, `-10` in Fig. 8.1 is definitely less than the value represented by `x` is `10`. But what if `x` was declared as an `unsigned int`? According to the C usual arithmetic conversion rules 3 and 5 found in §6.1.1.8 of the Standard, the `signed int` value of `-10` is promoted to a very large positive `unsigned int` value, *i.e.,* `4294967286` on a 32-bit integer architecture or `65526` on a 16-bit architecture. Fig. 8.1 shows the 32-bit patterns of both numbers, `10` and `-10`. Clearly, if the sign bit is ignored, then the complement scheme, be it one's or two's complement, is lost.

8.4 The Underlying Language ACL2

This section is provided for the benefit of those unfamiliar with ACL2. ACL2 is an acronym for A Computational Logic for Applicative Common Lisp [58, 59, 60]. As it can be surmised, ACL2 is two languages in one.

8.4.1 Computational Logic

The computational logic component is a theorem prover in Boyer-Moore tradition [58] where theorems are proved in a first order mathematical theory of recursively defined functions and constructed objects using rewrites, decision procedures, mathematical induction, and other proof techniques. First order logic uses predicate functions returning a truth value (true or false) built from arguments that may be other predicate functions as well as variables. Arguments may be quantified with the universal quantifier $\forall$ (for all) and/or the existential quantifier $\exists$ (there exists) to allow reasoning with respect to sets and shared set properties.

ACL2 was designed to eliminate the many difficulties users often faced when applying the Boyer-Moore Theorem Prover, otherwise known as Nqthm [14], to large scale proof problems. Nqthm (pronounced *en-queue-thum*) is an acronym for "New, Qualified THeoreM Prover," a name derived from the directory containing its development files. The computational logic of ACL2 includes an extensive axiom database defining many primitive functions for use as the logic embeds propositional calculus and equality into a term structure that resembles Lisp. According to the Nqthm tutorial [62], new axioms are added to the database by defining one or more of the following:

- Boolean constants `T` and `F`
- the `IF` function with the property that `(IF x y z)` is `z` if `x` is `F` and `y` otherwise
- boolean connectives `AND`, `OR`, `NOT`, and `IMPLIES`
- the equality function `EQUAL` with the property that `(EQUAL x y)` is `T` or `F` according to whether `x` is equal to `y`
- and inductively constructed objects including natural numbers and cons pairs

8.4.2 Applicative Common Lisp

The Applicative Common Lisp component is a subset of the functional programming language Common Lisp [120] as it uses many of the same Lisp functions originally introduced by McCarthy [75]. Lisp is a functional language and uses functions in the same sense that functions are used in mathematics, where a function maps a domain to a range. In math, a function f defined on some input n is written $f(n)$. The same function in Lisp is written $(f\ n)$. In both mathematics and the Lisp language family, every time f is applied to n, f returns the same value. For example, the addition function `(+ a b)` in Lisp means the values represented by the inputs `a` and `b` are summed.

By being an applicative subset, ACL2 prohibits all the data destructive side effects that Common Lisp allows, such as the use of global variables. Instead, ACL2 creates new data sets as an output while preserving all input data. Executables written in ACL2 are functions applied to five disjoint atomic data types:

1. numbers, such as `0`, `-123`, `22/7`, or #`c`(2 3)
2. characters, such as #\\`A`, #\\`a`, #\\$, or #\\`Space`
3. strings, such as "`This is a string.`"
4. symbols, such as `nil`
5. cons pairs, such as `(a)`, `(1.2)`, `(a b c)`, or `((a.1)((b 2))`

8.4.2.1 Numbers

An ACL2 number is typically written in base 10 or decimal notation. However, ACL2 also supports binary, octal, and hexadecimal notation. For example, the decimal `123` may be written as `#b1111011` for binary, `#o173` for octal, and `#x7b` for hexadecimal.

ACL2 does not support decimal floating point numbers. Whenever a rational number is required, it is written in fractional form. ACL2 will reduce any fraction to its lowest terms. Example rational numbers include `0`, `-77`, `123`, `1/3`, or `22/7`. The real and imag-

inary parts of complex numbers, such as `3+5i` and written as `#c(3 0)` in ACL2, are also considered rational numbers. Moreover, ACL2 uses the arbitrary-precision arithmetic model known as `bignums` where numbers are typically stored in variable-length arrays of digits instead of a fixed number of binary bits. That means an ACL2 integer can greatly exceed the maximum and minimum representable limits of the C integer types.

8.4.2.2 Characters and Strings

An ACL2 character is any member of the 256 ASCII character set. Upper case alpha characters such as the letter A are written as `#\A`; likewise, the lower case alpha character is written `#\a`. Other characters such as digits, punctuation marks, and signs are written similarly. That is, the `#\` precedes the desired character. Whitespace characters require their names to be spelled out, such as, `#\Space`, `#\Tab`, and `#\Newline`. A finite sequence of characters enclosed in double quotation marks form ACL2 strings. For example, "`Hello World`" is a string. If a double quotation mark needs to be part of a string contents, it must be preceded with a backslash such as \" or \".

8.4.2.3 Symbols

The ACL2 symbol is a data type representing a word. For example, `T` or `t` and `NIL` or `nil` (case does not matter because all lower case alpha characters are automatically converted to upper case unless otherwise escaped) are the ACL2 boolean symbols for true and false. The usage of `nil` can be overloaded to indicate an empty list. ACL2 symbols include all function and variable names. As such, a symbol may be used as a constant and except for the symbols `t` and `nil`, must be quoted to indicate an intended constant usage, for example, `'x`.

A symbol may also be comprised of two strings to represent a package name and a symbol name. Both parts are joined with "`::`" operator. The `::` operator is used

to associate a symbol name to the package name in which the symbol was defined, for example `PackageName::SymbolName` means `SymbolName` is in the package linked to `PackageName`. Packages are beyond the scope of this introduction. But it is worth noting that symbols printed with a leading `:` are of the package named "`KEYWORD`". For example, `:HINTS` represents the symbol `HINTS` in the package `KEYWORD`.

8.4.2.4 Cons Pairs

The final ACL2 data type is the `cons`. Sometimes called a list, a cons pairs, a dotted pair, or a binary tree, a `cons` is an ordered pair of objects. Any two objects, including another `cons` pair may be used in any `cons`. The left hand side is the `car` and the right hand side is the `cdr`. `nil` is the empty list and may be written as `()`. If all the elements of the lists are `cons`es, then we have an association list or alist. An alist can be used as a lookup table with each `car` along the list serving as an unique lookup key.

ACL2 can be used to solve problems, by performing computations through a series of expressions. Simple expression in ACL2 may be one of the following forms:

- a variable symbol other than a constant such as `x`
- a constant symbol such as `t`, `nil`, a keyword, or any other symbol declared using `defconst`
- a constant expression that is either a number, character, string, or quoted ACL2 object
- the application of a function expression of n arguments that is either a function symbol applied to n arguments or a closed lambda expression of the form (`lamba` (v_1 ... v_n) *body*), where the v_i are n distinct variable names, *body* is a simple expression and no other variable other than the v_i occurs freely in the *body*

ACL2 expressions are used to formulate terms. A term in traditional first-order predicate calculus is a syntactic entity denoting some object in the universe of all objects. But in ACL2, terms are used in place of both atomic and non-atomic formulas because

Listing 8.2: ACL2 `not` function

```
(defun not (p)
  (if p nil t))
```

in the universe of ACL2 objects, an object is either true or false. Atomic formulas are terms built with predicate symbols such as `equal` and `member`. Non-atomic formulas are built from atomic formulas using propositional operators such as `not` and `implies`. The non-atomic formulas are generally referred to as formulas.

In ACL2, terms also provide the semantics of expression. For example, the term `3` is simply the decimal number three and a function term such as `(if x y a)` means the value of `a` if the value of `x` is `nil`, otherwise the value of `y`. The meaning of the term `x` depends on the environment. `x` may represent a function argument or it could be the target of a `let` construct. For example,

```
(let ((x (+ 5 3))))
```

gives `x` the value `8` if the addition of `5` and `3` is closed to `x` (contained within parenthesis).

ACL2 functions are either built-in or user defined. Most ACL2 functions, however, are user defined using the keyword `defun` for *define function*. A short list of built in functions include `car`, `cdr`, `integerp`, `zp`, $\leq$, and `not` (Listing 8.2 [58]). We can interpret the definition for `not` (Listing 8.2) as "define function `not`, with formal parameter list `(p)`, such that the application of `not` to `p` is equal to `(if p nil t)`." To clarify, `(if p nil t)` means if `p` is true then return `nil` (false), otherwise return `t` (true). A complete list of built in functions can be found via the "PROGRAMMING" link at the ACL2 on-line documentation site [61].

ACL2 functions are recursive. Consider the ACL2 function `member` (Listing 8.3 [58]) that determines if an object has set membership. The `member` function takes two parameters, parameter `e` is the search value and parameter `x` is a list to search. If `e` is in the list `x`, then `e` is a member of `x` and `t` is returned. Otherwise, `nil` is returned. The

Listing 8.3: ACL2 `member` function

```
(defun member (e x)
  (if (endp x)
   nil
  (if (equal e (car x))
   t
  (member e (cdr x)))))
```

Listing 8.4: An iterative solution in C for `n!`

```
int factorial(int n)
{
  int fact = 1;
  for(int i = n; i > 0; i--)
    {
      fact = fact * i;
    }
  return fact;
}
```

`cond` states that if the list is empty (using `endp` to test an empty list), exit. Otherwise, compare `e` to the `car` of `x`. If equal, return true. Otherwise, call the `member` function using `e` and the `cdr` of `x` as arguments.

ACL2 does not directly support iteration. However, iterative logic, such as a loop, can be defined recursively. For example, consider the mathematics factorial denoted $n!$ where $n!$ is the product of all positive integers less than or equal to n. If $n = 5$, then $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$. Using C, we can write a function to compute $n!$ iteratively (Listing 8.4) or recursively (Listing 8.5). Using ACL2, however, the function to compute $n!$ is recursive (Listing 8.6 [58]).

In ACL2, all functions are total meaning they must terminate. With application of both language features, ACL2's strength rests in supporting formal models of algorithms, compilers, microprocessors, and machine languages that are both executable and analyzable [58].

Listing 8.5: A recursive solution in C for `n!`

```
int factorial(int n)
{
  int fact = n;

  if(n == 1)
    return 1;
  else
    return fact * factorial(n - 1);
}
```

Listing 8.6: A secursive solution in ACL2 for `n!`

```
(defun factorial (n)
  (if (zp n)
   1
  (* n (factorial (- n 1)))))
```

8.4.3 Reasons Behind the Use of ACL2

In addition to the applicative nature of ACL2, three other factors that led to the choice of using ACL2 in constructing the C integer type safety checking tool [66] introduced here within. First, the data types supported by ACL2 do not suffer from the same error conditions common to the C integer types. In particular, ACL2 uses `bignums` which does not limit integers bit-wise precisions. We can leverage `bignums` to store intermediate values for range check comparisons against the representable minimum and maximum values of any given C integer type. Second is the proof generation capacity of ACL2. ACL2 can generate proofs to verify that the assumptions made about C static typing semantic are correct. In addition, ACL2 can be used to prove that the tool is working accurately and completely. Furthermore, ACL2 can be used produce an executable that readily models the type safety specifications of C programs.

Finally, the ACL2 based type safety analyzer will take advantage of, and add value to another ACL2 based tool, the `c2acl2` translator [3]. The `c2acl2` translator creates a symbol table (`SYMTAB`) and the abstract syntax parse tree (`AST`, Listing 8.8) from C

Listing 8.7: A simple C program that adds operands of mixed integer type

```
int main()
{
  signed char schar_1;
  short short_1 = -10;
  unsigned int uint_1 = 10;
  int int_1 = 120;

  schar_1 = schar_1 + uint_1;
  schar_1 = short_1 + uint_1;
  schar_1 = schar_1 + uint_1;
  schar_1 = short_1 + int_1;
  schar_1 = int_1 + uint_1;
  return schar_1;
}
```

source code (Listing 10.1). In its current version, however, the tool only acts upon the `c2acl2` generated AST.

8.4.4 In Summary

The combination of its programming and theorem proving capabilities makes ACL2 suitable for a wide range of tasks. According to Kaufman et al. [59], ACL2's strength rests in its support of formal models which gives ACL2 the ability to solve many modeling problems from algorithms, compilers, microprocessors, and machine languages that are both executable and analyzable. The formal models can specify results and act as efficient simulators. ACL2 has successfully modeled and verified several systems. The short list includes a Java like byte code and its interaction in the Java Virtual Machine [86], software-based fault isolation or "sandboxing" [74], flash memories [106], secure COTS in hardware design [104] and the AMD processor [111].

Listing 8.8: AST for a simple C program that adds operands of mixed integer types

```
(C2ACL2 (FILE ‘‘simpleAdd’’)
  (
  ";**********************************************"
  "; Function Definition for function: main"
  ";**********************************************"
  (FUNC (INT )(ID "main" 1 ) NIL
    (BLOCK
      (DECL (SIGNED CHAR )(ID "schar_1" 2 )NIL)
      (DECL (SHORT )(ID "short_1" 3 )
        (INIT (UNMINUS (LIT 10))))
      (DECL (UNSIGNED INT )(ID "uint_1" 4 )
        (INIT (LIT 10)))
      (DECL (INT )(ID "int_1" 5 )
        (INIT (LIT 120)))
      (EXPSTMT (ASSN (ID "schar_1" 2)
        (ADD (ID "schar_1" 2) (ID "uint_1" 4))))
      (EXPSTMT (ASSN (ID "schar_1" 2)
        (ADD (ID "short_1" 3) (ID "uint_1" 4))))
      (EXPSTMT (ASSN (ID "schar_1" 2)
        (ADD (ID "schar_1" 2) (ID "uint_1" 4))))
      (EXPSTMT (ASSN (ID "schar_1" 2)
        (ADD (ID "short_1" 3) (ID "int_1" 5))))
      (EXPSTMT (ASSN (ID "schar_1" 2)
        (ADD (ID "int_1" 5) (ID "uint_1" 4))))
      (RETURN (ID "schar_1" 2))
    )
  )" ; End of function main"
  )
)
```

Chapter 9: Leveraging State in a Static Analysis Environment

Although the type safety verification tool introduced in this dissertation performs a static analysis by traversing, line by line, an abstraction of C source code, it needs to be somewhat aware of state. In this chapter, the importance of state and how to track it is discussed.

9.1 A State-full Introduction

Procedures to verify programming languages and computer programs often rely on the concept of state [67]. Bishop defines state as the set of all data residing within a computational framework at any given time slice [11, 12]. This includes data held in memory, the registers, the program counter (PC), etc.

According to Ray et al. [107], program correctness is proved by showing that the program began in a state satisfying one or more certain preconditions and terminated in a state satisfying one or more certain postconditions. Without making a preferential claim, Ray et al. [107] present three main proof strategies, that may be used individually or interchangeably, to verify deterministic, sequential programs:

1. the employment of stepwise invariants,
2. the application of clock functions, and
3. by reasoning drawn from inductive assertions anchored to program cutpoints.

For example, Moore [85, 86, 87] applied clock functions to an abstract machine semantics of the Java Virtual Machine (JVM) to verify its operational semantics by showing that state changes of any given operation were both predictable and repeatable. State of the JVM was modeled as a tuple holding the PC, the stack (STK), the memory (MEM), the exit condition (HALT), and the code. Instructions for the operational semantics were derived from the assembly code generated from compiling JVM code. All three proof

strategies typically leverage state with abstract machine semantics to verify pre and post conditions using Hoare [32, 46] like axiomatic semantics.

Several of the tools to verify the operational semantics of programming languages [85, 86, 87] and the secure data flow of programs [4, ?, 51] have been written in ACL2 [58]. Most use an abstract machine semantics that the tools step through with the aid of two state wise functions Ray et al. [51, 105, 107] call next and run. Both functions, next and run, are derived from the abstract interactions of the PC and the source code. For example, the function $next : S \rightarrow S$ enables state to next state transitions where S is the set of member states individually denoted by s. For any s, $next(s)$ returns the state after executing one instruction $\in$ IN from s. Executions are handled in the function $run : S \times \text{IN} \rightarrow S$ which returns the state after n instructional transitions from s.

$$\text{run}(s,\ n) \triangleq \begin{cases} s & \text{if } exit(s) \\ run(next(s,\ n-1)) & \text{otherwise} \end{cases} \tag{9.1}$$

Exit is the terminal state and is a third predicate in addition to the pre and post conditions. The exact meaning of exit depends on the program being analyzed. For example, if the analysis is of a sub routine, such as a function, then exit is the point where control is given back to the calling program. If all three predicate types can be satisfied, then a program's correctness is said to be complete. If the analysis cannot reach the terminal state, however, all pre and post conditions have been satisfied, then correctness is said to be partial.

9.2 Application of Functions Run, Next, and Exit

The prototype type safety tool introduced in this dissertation uses an abstraction of C source code generated by the `c2acl2` translator [4]. The high level instructions (Table 9.1) of the abstraction are organized either globally or within scope or blocks of functions,

Declarations	Expression Statements	Flow Control
DECL	EXPSTMT	IF
FUNCDECL		ELSE
TYPEDEFDECL		CASE
		DO
		WHILE
		FOR
		COND

Table 9.1: High level instruction nodes

individually denoted by FUNC. Within each FUNC block, there is a series of declarations, expression, and flow control statements. The execution flow control statements introduce their own sub-scoping blocks of the high level instructions within the FUNC block (Fig. 9.1).

Each instruction set is arranged as nodes along a “linear tree”. The tool steps the instructions contained in this tree by sequentially evaluating one instruction at a time after beginning with the first instruction node on the tree. Evaluation of the first instruction is the current run function. Once evaluated, the remaining tree is passed back into the tool where the first instruction of that tree is then evaluated. The process of returning the remaining instruction tree is the run next function. Once all of the instructions contained in the tree are evaluated, the exit function is reached.

9.3 Tracking State

As each instruction is evaluated, the integer type safety tool makes one of three determinations.

1. The instruction is type safe.
2. The instruction is not type safe.
3. Not enough is known about the instruction to make a determination about its type safety.

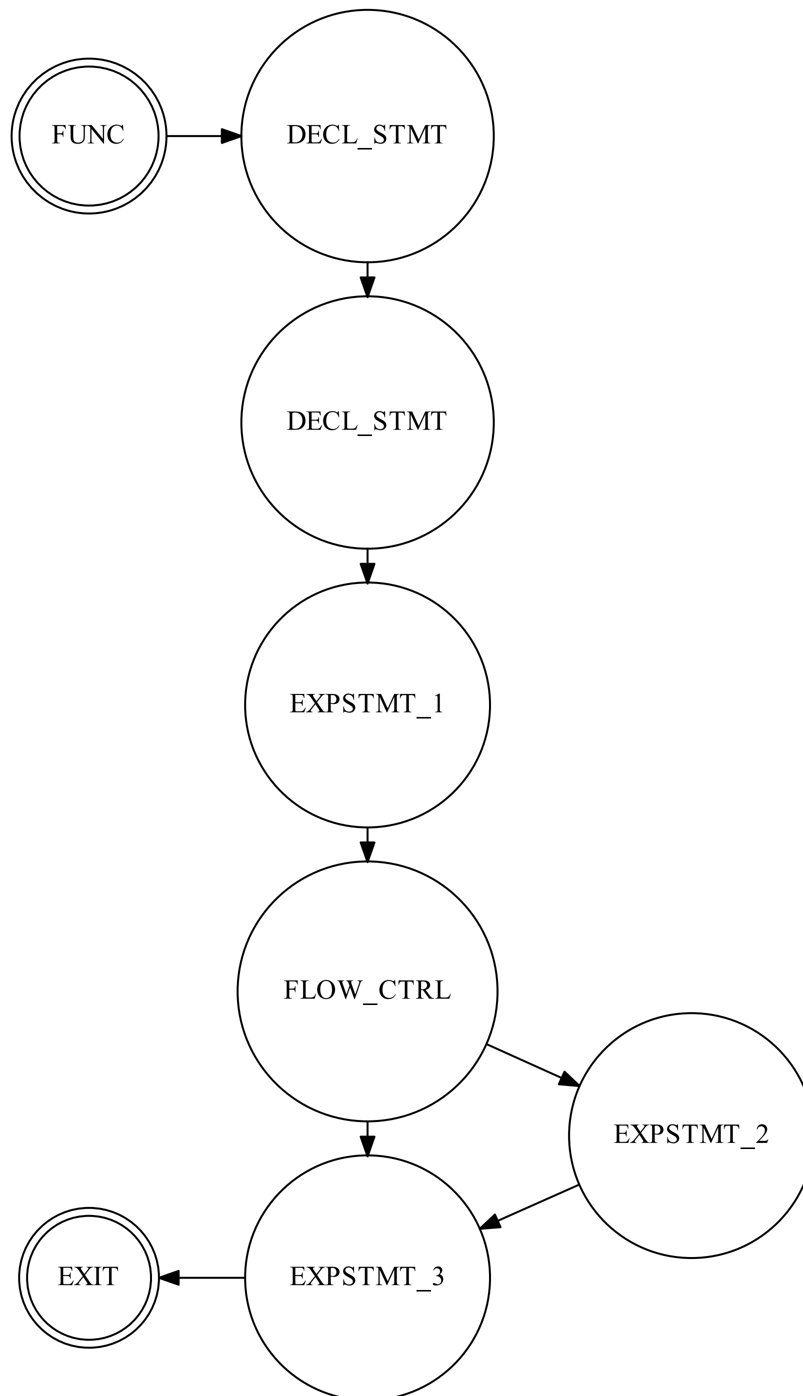

Figure 9.1: A simplistic AST

The first two type safety judgments are intuitive. The third is a reality of any static analysis of source code that calls upon external data sources or externally defined functions whose values and types are only realized during dynamic run time conditions.

The output of the tool to report its type safety decisions can be fine tuned. When the most verbose output is used, type safe instructions are echoed with an appended line number. If the instruction is a `DECL`, the change to memory is recorded. If the instruction is an `EXPSTMT` that has an assignment (`ASSN`), the memory updates resulting from the assignment are also recorded. Instructions that are deemed to be either not type safe or if the type safety cannot be determined are also echoed with an appended line number and a corresponding error or warning message. In the case of instructions considered to be unsafe, an error message is generated. If it is the case when a safety decision cannot be made, a warning message is generated. A warning message serves a verification marker to indicate that the function requires external data sources and needs additional type safety analysis. Moreover, the warning messages can be customized to produce verification conditions stating that the instruction will be type safe if one or more attributes of the data are satisfied. For example, the data value has to belong to a specified range of values within the representable values of the data type of the expression or statement.

9.3.1 Type Safety Decision Algorithms

For each instruction node along the AST, the integer type safety tool executes a series of precondition predicate checks based upon the static typing semantics and the type safety requirements with respect to the corresponding expression. If all predicates are satisfied, then the conditions prior to execution are type safe. If one predicate is not satisfied, a corresponding error message is generated. The general algorithm for the precondition checks is presented in Fig. 9.2.

```
expType = getExpType(operator)
for i = 1 to n do
  if operandType(operand_i) ∉ expType then
    return ERROR MSG
  end if
end for
for i = 1 to n do
  if operandWarningCond(operand_i) then
    return WARNING MSG
  end if
end for
for i = 1 to n do
  if operandErrorCond(operand_i) then
    return ERROR MSG
  end if
end for
for i = 1 to n do
  if getOperandVal(operand_i) = nil then
    return ERROR MSG
  end if
end for
unit_p = negVal_p = nil
for i = 1 to n do
  if getOperandType(operand_i) >= UINT then
    uint_p = T
  end if
end for
for i = 1 to n do
  if getOperandVal(operand_i) < 0 then
    negVal = T
  end if
end for
if uint_p and negVal_p then
  return ERROR MSG
end if
```

Figure 9.2: Basic precondition algorithm

There are five main precondition checks contained in Fig. 9.2. The first is built with respect to the typing constraints that the operator places on its operand. Admittedly, most compilers should catch operand types that are outside of the operator typing constraints. However, this precondition is included for coverage completeness of the tool. The second precondition checks to see if an operand residing in memory has been issued a warning condition. Warnings like error conditions reside in memory until that memory location has been reassigned with a valid value for its type. As such, the warning associated to a memory operand has to propagate until a suitable reassignment to that operand has been performed. The third precondition applies to those memory operands that have entered an error condition. These operands are treated the same as those operands that have been associated to a warning where the associated error propagates until the memory location is reassigned a valid value for its type. The fourth precondition checks to see if the operand residing in memory has been at least initialized with a valid value for its type. The fifth and final precondition checks compiler type coercions. Because the C coercion rules usually force all operands to the same type of the largest bitwise operand type and because all `ACL2` number types are of type `bignums`, this condition has been heuristically simplified. That is, since most up-casts are generally safe if there is not a sign-age change among the integer types, the tool will issue a warning on those expressions that contains one operand of type unsigned integer or a larger unsigned integer type based on integer rankings and if another operand contains a negative value.

If all preconditions are satisfied, the tool attempts to fully evaluate the high level instruction set. For declaration instructions, designated by the node `DECL` or `FUNCDECL` or `TYPEDEFDECL`, the instruction can only be fully evaluated if there is an initialization value. If an initialization value is present, the tool checks to see if the value can be fully represented by the declared type.

Expression statements, designated by the node `EXPSTMT`, are much more interesting. An `EXPSTMT` can be comprised of one or more sub-expression statements (Table 9.2).

<table>
<tr><th></th><th>Expression</th><th>Operator</th><th>AST node</th></tr>
<tr><td>Binary:</td><td>Logical OR</td><td>||</td><td>OR</td></tr>
<tr><td></td><td>Logical AND</td><td>&&</td><td>AND</td></tr>
<tr><td></td><td>Inclusive OR</td><td>|</td><td>BITOR</td></tr>
<tr><td></td><td>Exclusive OR</td><td>^</td><td>BITXOR</td></tr>
<tr><td></td><td>AND</td><td>&</td><td>BITAND</td></tr>
<tr><td></td><td>Equal</td><td>==</td><td>EQ</td></tr>
<tr><td></td><td>Not equal</td><td>!=</td><td>NEQ</td></tr>
<tr><td></td><td>Less than</td><td><</td><td>LT</td></tr>
<tr><td></td><td>Greater than</td><td>></td><td>GT</td></tr>
<tr><td></td><td>Less than or equal</td><td><=</td><td>LEQ</td></tr>
<tr><td></td><td>Greater than or equal</td><td>>=</td><td>GEQ</td></tr>
<tr><td></td><td>Shift left</td><td>«</td><td>SHIFTL</td></tr>
<tr><td></td><td>Shift right</td><td>»</td><td>SHIFTR</td></tr>
<tr><td></td><td>Addition</td><td>+</td><td>ADD</td></tr>
<tr><td></td><td>Subtraction</td><td>-</td><td>SUB</td></tr>
<tr><td></td><td>Multiplication</td><td>*</td><td>MULT</td></tr>
<tr><td></td><td>Division</td><td>/</td><td>DIV</td></tr>
<tr><td></td><td>Modulo</td><td>%</td><td>MOD</td></tr>
<tr><td>Cast:</td><td>Cast</td><td>()</td><td>CAST</td></tr>
<tr><td>Unary:</td><td>Address</td><td>&</td><td>AMP</td></tr>
<tr><td></td><td>Indirection (pointer)</td><td>*</td><td>STAR</td></tr>
<tr><td></td><td>Unary plus</td><td>+</td><td>UNPLUS</td></tr>
<tr><td></td><td>Arithmetic-negation</td><td>-</td><td>UNMINUS</td></tr>
<tr><td></td><td>Negation (bit-wise complement)</td><td>~</td><td>NEG</td></tr>
<tr><td></td><td>Not</td><td>!</td><td>NOT</td></tr>
<tr><td></td><td>sizeof</td><td>()</td><td>SIZEOF</td></tr>
<tr><td></td><td>Array subscripting</td><td>[]</td><td>[</td></tr>
<tr><td></td><td>Function call</td><td>()</td><td>(</td></tr>
<tr><td></td><td>Structure or Union member</td><td>.</td><td>DOT</td></tr>
<tr><td></td><td>Structure or Union member pointer</td><td>-></td><td>DEREF</td></tr>
<tr><td></td><td>Postfix increment</td><td>++</td><td>POSTINC</td></tr>
<tr><td></td><td>Postfix decrement</td><td>-</td><td>POSTDEC</td></tr>
<tr><td>Primary:</td><td>Identifier</td><td></td><td>ID</td></tr>
<tr><td></td><td>Constant</td><td></td><td>LIT</td></tr>
<tr><td>Ternary:</td><td>Conditional</td><td>? :</td><td>COND</td></tr>
</table>

Table 9.2: Sub-expressions of EXSTMT node

$expResult = evalExp(\mathsf{expression})$
if $expResult \subseteq expType_{valRange}$ **then**
  return expResult
else
  return ERROR MSG
end if

Figure 9.3: Basic postcondition algorithm

Each sub-expression is defined in §6.5 of the standard as any combination of a sequence of operators and operands to compute a value, a designation of an object or function, or generate a side effect. If all operand values for their respective operators are known in the source code, the tool fully evaluates the expression using a basic postcondition algorithm (Fig. 9.3). Basically, the tool performs a range check to see if the result value can be legally represented by the type of the expression. If the value cannot be represented by the expression type, an error message is generated. If it is the case that the operand is of a legal type of the expression, but its value is unknown, the tool generates a warning message to indicate that additional analysis is needed.

9.3.2 Tracking State-wise Changes with an Annotated Look-up Table

From the type safety checking algorithms, it is apparent that the type safety verification tool has to be somewhat state-wise aware as it performs a static analysis. In contrast, an executing program could care less if an operand from memory contains a legal value or if its value is in an error condition. Unless runtime safety checks are programmed into the source code, an executing C program merely shoves data bits from memory to the registers, performs the calculations, and if programmed to do so, stores the calculated results back in memory. Thus in many ways, a C program during runtime is not state-wise aware.

Listing 9.1: Annotated fields for each basic data object in the lookup table

```
((TOKEN-ID              ; mandatory field - lookup key
 ("NAME")               ; mandatory field - human readable name
 ((TYPE-SPECIFIERS)     ; mandatory TYPE-INFORMATION field
  (TYPE-QUALIFIERS)     ; with three typing sub fields
  (STORAGE-CLASS-SPECIFIERS))
 (VALUE)                ; mandatory field
 (ERROR\WARNING))       ; optional field - used as needed
```

To assist the tool in making type safety decisions that are often times based on state, an annotated lookup table was developed [68]. The concept of annotating data objects is not new. For example, the programming language Jif [112], a subset of Java, allows programmers to label its data objects to express how the data held by each data object can be used. Gennari et al. [37] proposed that C object types be programmer annotated with the representable value range of the data type to enable validation of state changes. Our tool differs from these proposals as it automatically annotates the types during evaluation without needing additional programmer intervention.

The annotation schema for the members of our model consists of four primary fields and one optional field (Listing 9.1). The rationale and purpose for each field are detailed in the following sections. The fields for each member of the lookup table are initially populated whenever a new member is introduced by a declaration statement beginning with either `DECL`, `TYPEDEFDECL`, or `FUNCDECL`. After the first member is added to the lookup table, the lookup table is updated with either additional members or state-wise memory changes from the subsequent functions for the examination of the remaining declarations, expressions, control statements, function calls and returns, and loop counters for type safety violations.

9.3.2.1 The Fields `TOKEN-ID` and `("NAME")`

According to §6.7 of the C standard, all declared variables must have an identifier name, and each name must be unique within the scope of which the variable is declared. Scope may be either global or local. Global is applicable to anywhere in a program and local is confined within the block of a procedure, such as a control statement or a function. A global identifier can only be used once, while an identifier for a local variable can be used more than once, so long as it is unique to the block in which it resides. For example, the identifier `i` is typically used as shorthand for the name "`index`" because it is often used as the counter in the controlling expression of iterative statements, such as a `for` loop with the construct

`for (` *clause-1* `;` *expression-3* `;` *expression-3* `)` *statement.*

If `i` is declared in *clause-1*, the scope of `i` is confined to the `for` loop body. Because of this, the identifier `i` may be declared in the *clause-1* of many different `for` loops within a single program and still remain unique to each `for` loop.

As `c2acl2` translates C source code, it issues an unique integer as the identifier to each newly declared object. This integer identifier is an abstraction of a memory location and it is what populates the `TOKEN-ID` field. For example, in the `c2acl2` translated declaration statement

```
(DECL (INT )(ID "i" 2 ) NIL)
```

the integer value `2` in the field `(ID "i" 2)` is the unique identifier issued by `c2acl2`. Our integer type safety tool uses the `TOKEN-ID` field as a lookup key to find the data members residing in the lookup table.

Likewise, the `("NAME")` field is populated with the identifier used in the C declaration statement. In the case of this example, the `"i"` populates `("NAME")`. The primary purpose of the `("NAME")` field is to keep the output of our tool human readable by using named memory variables.

9.3.2.2 The (TYPE-INFORMATION) field

The (TYPE-INFORMATION) field and its three sub-fields serve three primary purposes for our integer type safety checking tool. First, the (TYPE-INFORMATION) field assists in determining legal value ranges based on expression operand types. Each operand object type τ has a minimum and a maximum representable value it can hold based on its bitwise precision. Our tool uses this value range to validate values assigned to operand objects. If a value is a subset of the operand type representable value range

$$\tau_{\min_{val}} < \tau_{\min_{val+1}} < \; \dots \; < \tau_{\max_{val-1}} < \tau_{\max_{val}}, \tag{9.2}$$

then the value can legally be used by that object. If value cannot be represented by the operand type, our tool generates an appropriate `ERROR` statement and populates the optional fifth (ERROR\WARNING) field in the lookup table for that operand.

Second, the (TYPE-INFORMATION) field is used to validate type casting operations based on the C integer ranking, integer promotions, and usual arithmetic conversions rules. For example, the outcome of a relational expression with an unsigned integer operand (`x = 10`) and negative signed integer operand, such as

```
if(x > -10){ // do something important... },
```

can produce two different outcomes depending on the integer rank of `x`. If the rank of `x` is less than the rank of the `signed int`, such as the `unsigned char` or the `unsigned short`, then the integer promotion rule will promote `x` to type `int` which is `signed`. The literal `-10` is also promoted to `int` and because its value remains unchanged, the outcome of the relational expression is true. If `x` is of type `unsigned int`, the negative signed integer value is promoted to the unsigned integer type and its value becomes a very large positive value causing the outcome to be false. Whenever our tool evaluates an expression with integer operands of different sign types, it checks to see if the signed integer has a negative value in addition to applying the conversion rule. If it is determined that the

negative value will lose its sign bit, our tool generates an appropriate ERROR statement that populates the optional fifth (ERROR\WARNING) field.

Finally, many C expressions have operators that place specific typing constraints on their operands. The (TYPE-INFORMATION) field is used by our tool to check all expression operator typing constraints as defined in the C standard. Again, if a typing violation is found, an appropriate ERROR statement is populated in the optional fifth field. Whenever a fifth field error message is generated, the error message will propagate every time the data object is used as an expression operand other than the operand on the left hand side (LHS) of an assignment expression. If the LHS operand is reassigned a valid value, then the error message is dropped. Otherwise, the error stays in place until the data object is reassigned a legal value.

Each of the three TYPE-INFORMATION subfields,

- the (TYPE-SPECIFIERS),
- the (TYPE-QUALIFIERS), and
- the (STORAGE-CLASS-SPECIFIERS),

corresponds to one of the three kinds of declaration-specifiers used in the syntax of declarations as defined in §6.7 of the C standard and may be used individually or together to validate the type safety of any given program construct.

9.3.2.3 The Type Subfield (TYPE-SPECIFIERS)

TYPE-SPECIFIERS define the type of the declared memory object identifier by the use of at least one type-specifier keyword (Fig. 9.4). If more than one type-specifier keyword is required, the keywords may be used in any order. For example, the expressions `unsigned int x;` and `int unsigned x;` are semantically equivalent. In the case where more than one keyword is required to define a type such as `unsigned int`, the tool populates TYPE-SPECIFIERS with standardized type abbreviations (ordered by bitwise precision) _BOOL, UCHAR, USHORT, UINT, ULONG, and ULLONG for the unsigned integer types. For

```
type-specifier  ::=  void | char | short | int | long | float | double
                  |  signed | unsigned | _Bool | _Complex | _Imaginary
                  |  struct-or-union-specifier | enum-specifier
                  |  typedef-name
```

Figure 9.4: C type-specifier `keywords`.

the signed integer types, it uses `SCHAR`, `SHORT`, `LONG`, and `LLONG`. Since the signage of the C type `char` is implementation defined, our tool conservatively considers it to be an `UCHAR` for the time being. The tool applies the same technique to the float types and the incomplete-type `VOID`. The floating types `_Imaginary` and `_Complex` are rare because they are only used by the few implementations that support such types and, as such, they are not included in the current development of the type safety verification tool.

9.3.2.4 The Type Subfield (`TYPE-QUALIFIERS`)

`TYPE-QUALIFIERS` represent the type-qualifier keywords[1] assigning specific usage properties to a declared object (Fig. 9.5). For example, the type qualifier `const` declares an object to be unmodifiable. On the other hand, a `volatile` object can be changed either internally or externally with respect to the scope of the program for which it appears. The qualifier `restrict` can only be used on pointers and is intended to promote program optimization.

Because the semantics of the type-qualifier keywords can have conflicting meanings, an object generally may only have one type-qualifier keyword applied to it during its declaration. Nevertheless, the syntax of type-qualifier usage allows the presence of zero or more type-qualifier keywords in a single declaration statement. The keywords can be used in any order, however, compilers generally apply only the first while stripping off the rest when reserving an object's memory space. Our tool mimics the general compiler action by populating `(TYPE-QUALIFERS)` subfield with the first type-qualifier keyword

[1] C11 introduced the type-qualifier `_Atomic` to be used in conjunction with the new header file `stdatomic.h` to support multitasking such as threading and atomic operations.

type-qualifier ::= `const` | `restrict` | `volatile`

Figure 9.5: C type-qualifier `keywords`.

storage-class-specifier ::= `typedef` | `extern` | `static` | `auto` | `register`

Figure 9.6: C storage-class-specifier `keywords`.

appearing in a declaration statement. If a type-qualifier keyword is not used during the declaration, the `(TYPE-QUALIFIERS)` field of the type field is populated with `NOQUAL`.

9.3.2.5 The Type Subfield `(STORAGE-CLASS-SPECIFIERS)`

`STORAGE-CLASS-SPECIFIERS` (Fig. 9.6) records the storage-class-specifiers keywords that may be used in the declaration statement. Compilers use these keywords to determine the duration, the visibility, and the storage needs of the object or function being declared. The lifetime is either global or local. A global lifetime is static in the sense that it exists throughout the execution of a program. Functions, for example, have global lifetimes. An automatic object has a local lifetime in the sense that its storage space is allocated whenever the execution of the program enters the block where the automatic variable is defined. Visibility refers to whether an object was originally declared locally or externally to the program. Finally, the storage directive `register` is a request to create a register object. The syntax of storage-class-specifier declaration statements requires that at most one storage-class-specifier keyword may be used with the following caveat. If it is the case that the keyword is `typedef`, then at most one additional storage-class-specifier keyword may be used.Thus, the tool populates `(STORAGE-CLASS-SPECIFIERS)` with the single storage-class-specifier keyword or with `NOSTORE` to represent the absence of such a keyword.

9.3.3 The (VALUE) Field

VALUE is primary used to record state-wise value changes to a data object and is initially populated when our tool evaluates the declaration statement of that object. If an initializer is not present, it is populated with nil. The purpose of nil is to alert whenever an uninitialized operand is being used in an expression other than as the LHS operand of an assignment expression. Such an operand condition will produce an appropriate ERROR statement. If a declaration statement has an initializer, the tool first evaluates the expression to produce a result value (RESULT). Then, RESULT is range checked against the representable values the declared type can hold. If

$$\tau_{\min_{val}} \leq \texttt{RESULT} \leq \tau_{\max_{val}}, \tag{9.3}$$

VALUE is populated with RESULT. If not, VALUE is populated with nil and an appropriate ERROR message is issued.

VALUE gives our tool the ability to evaluate expressions, statements, function calls, loop counters, etc. for type safety violations. If a result of the evaluation can be determined, the (VALUE) field for the LHS operand of an assignment is repopulated with the new result which is range checked to the type of the LHS operand. If the result cannot be determined as is the case with unknown external input values or externally defined functions, a WARNING statement is appended in the fifth (ERROR\WARNING) field stating that the unknown input is legal if and only if it falls within the legal range of values that can be represented by the given type. The warning message is akin to a verification condition. Warning messages like error messages propagate until the data objects they are associated with are reassigned with a valid representable value.

9.3.3.1 A Note on Modeling Other Data Objects

While the focus of the previous discussion on the annotated lookup table has been on the base arithmetic-types, integers and floats, the model of our lookup table can be adapted to accommodate other data types by adding additional subfields within the main fields of the schema. The complete aggregate-type `array`, for example, has at least one declared size and a corresponding set of values. Its type is determined by its member objects, however, its size is of integer type and must have a value greater than `0`. The size information becomes a subfield of the `(TYPE-QUALIFIERS)` field. Since the typical length of the `(TYPE-QUALIFIERS)` equals `1` for the base arithmetic-types, the application of a couple built in ACL2 Lisp functions, such as

```
(> (length (TYPE-QUALIFIERS)) 1),
```

is a quick and simple way to determine if the member of the lookup table is either an aggregate-type, a pointer type, or an union-type. Only one additional test is required to determine if the member is an array.

Likewise, each index value of an array is represented by separate subfields in the `(VALUE)` field taking the form

$$\texttt{(((}i_0\texttt{)}\ldots\texttt{(}i_{n-1}\texttt{)))}$$

where i is the index (member) value and n is the size with the range $\{n-n=0,\ (n-n)+1,\ \ldots,\ (n-n)+n-1\}$. For multi-dimensional arrays, each dimension is contained within separate subfields of the `(VALUE)` field to hold their respective element values. For example, a two dimensional array with two sizes, n and m, and two sets of values, i and j, takes the form

$$\texttt{(((}i_0\texttt{)}\ldots\texttt{(}i_{n-1}\texttt{))((}j_0\texttt{)}\ldots\texttt{(}j_{m-1}\texttt{))).}$$

If an array is declared with a size, but without an initialization, then the number of value subfields corresponds to the size are populated with `nil`. If it is the case that there is at

least one member initialization, the remaining uninitialized members are initialized to `0` as prescribed by the standard.

Chapter 10: Conclusions and Future Work

This dissertation summarized an investigation into solving the ongoing problem of integer type safety in the programming language C. The problem is as old as the language itself and has been well studied. Over the years, many solutions have been proposed and yet, the problem persists. This chapter reviews the contributions this dissertation revealed in attacking the C integer type safety question.

10.1 Review and Conclusions

While justifying why another investigation into the problem of C integer type safety shortcoming is important, this dissertation noted that type safety is the program property that is free of unexpected behaviors manifesting from unexpected results. Unlike security that protects computer systems from external threats, safety strives to offer protection to the environment the system operates within. While not all safety violations result in actual damages, they can leave a system vulnerable to security breaches through the attack vectors that include denial of service, escalation of privileges, and execution of arbitrary, malicious code.

10.1.1 Organization of this Dissertation

This dissertation continued with three main sections. The first examined the root causes of type safety failure with respect to C. One reason C continues to be a leading systems programming language is because of the many freedoms it affords to system or computer programmers. But its strengths were derived at the cost of a lack of bounds checking on many of the data structures held in memory during runtime. This includes the integer types which are suspect to enter one of three error conditions: overflow or underflow error, sign error, and truncation error. Many of the error conditions are the result of a casting or a coercion operation. Whether the operation is explicit or implicit, there is

only one safe integer type conversion and that is the up-cast of a smaller integer type to a larger integer of the same sign type. This integer conversion is the only conversion that can be represented in a sub-type relationship. All other casting or coercion operations are potentially unsafe. An integer error condition may also become apparent following an operation applied to integers where the result of the operation results is a value that is either too larger or too small to be represented by the integer type of the operation or the type of the memory data store. In addition to integer errors, undefined operations such as divide by `0` will likely produce undefined behaviors. This section concluded with summary of the current mitigation strategies in use and proposed to avoid type safety violations. These range from safe coding practices to safe sub-sets of the language to the use of internal integer operation validation functions and the use of external validation tools.

10.1.2 Formalization

The second section of this dissertation was a two step exercise in formalizing C. In the first step, a specification of the C type system using a Backus-Naur Form like structure was introduced. Presentation of the type specification using this method is not only new, it also shows the sub-type relationships for all types. For example, both signed and unsigned integer types are a sub-type of the floating types but neither integer type is a subtype of the other integer type.

In the second step, the static typing semantics for C expressions and statements were expressed and enhanced with formal type safety requirements. Although C static typing semantics have been formalized before, the interpretation used in this dissertation is much simpler than that of the prior attempts. The goal here was that programmers and others without an extensive mathematical background could readily understand what was being expressed. Moreover, previous work in formalizing the static typing semantics

were applied to earlier standards of the C language and did not include the 64 bit integer types; and, the earlier works did not include the supplemental type safety requirements.

10.1.3 The C Type Safety Verification Tool

The final section of this dissertation introduced a new type safety verification tool for programs written in C. The logic of its functional algorithms was based on the contents of the previous two sections of this dissertation: the factors that can bring forth type safety issues and a formal understanding of the C type system, its static typing semantics and type safety requirements.

The tool was written in ACL2 because ACL2 provides a means to write executables that can model the typing semantics and operations of C source code. In addition, ACL2 theorem proving capabilities give the ability to reason about the above models and to show that the models are correct. ACL2 can do this because it was designed to preserve all original, input data sets and will create additive data sets whenever needed. As a result, all original abstractions of the source code remain unaltered. Finally, the representation of numbers using `bignums` in ACL2 allows for valid range without the bit width restrictions imposed on C data types.

The tool contains hundreds of functions contained within several thousand lines of code. Often, it is the case that the tool utilizes a collection of functions to perform a single task. As such, the inner workings of every function cannot be detailed in this dissertation. Although the tool performs a static analysis by traversing, line by line, over an abstraction of C source code, it maintains a limited notion of state by using a annotated lookup table whose members are populated from the declaration expressions contained within the source code. Each respective entity in the lookup table contains fields to assist with its identification, record its typing and value information, and indicate if the data object is in an error condition.

Whenever possible, the tool fully evaluates the abstracted expression or statement. If the expected input data is supplied from external data source, its value cannot be know during the static analysis. If that is the case, a verification condition (warning statement) will be generated. In either case, if the tool determines that an error condition is likely or certain, it will issue an error statement. Error and warning statements remain in the lookup table until the data object in question is reassigned a valid value representable by its type.

10.2 Assumptions

Success of the tool cannot be determined without the acknowledgment of certain assumptions and limitations. For example, it is generally believed that the tool can reason about fairly obfuscated C source code, the tool is designed to analyze code written to good coding practices. That said, certain constructs are not allowed and marked as `PROHIBITED` by the `c2acl2` translator:

- functions with variable length parameter lists
- compound assignment operators
  - `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `&=`, `^=`, and `|=`
- the unary prefix `++` and `--` operators

Also currently prohibited by the `c2acl2` translator are the type specifiers `FLOAT` and `DOUBLE`. It is also recommended that usage of the `goto` and `continue` should be limited.

Expressions should be as simple as possible with the intent of accomplishing a single step of a computational solution. Ideally, the source code should contain validation functions and it needs to be well documented. Although the latter is removed during the `c2acl2` abstraction, the documentation in the original code can provide valuable insight to the expected behaviors during the review process.

10.3 Limitations

At present, there are three major limitations to the tool introduced in this dissertation. First and by default, ACL2 is designed to manipulate rational numbers only. For example, ACL2 executables can handle values such as `11/10` but not its equivalent floating point representation of `1.1`. Thus, the tool in its current state is not equipped to reason about floating point values and will crash whenever it encounters a floating point value in the `c2acl2` abstraction. It is assumed that a script could be written that would translate the representation of a floating value to that of a rational number. However, such an option has not yet been explored.

Experience with another research project has shown a second limitation and that is an ACL2 `alist` has its own size constraints. That is, an `alist` containing somewhere between 20k and 25k contiguous linear nodes will cause a program stack overflow during routine list transversal operations such as those used in lookup functions. This could become problematic in analyzing large bodies of source code containing many thousands or millions of lines of code.

One of the largest test files analyzed to date contained 79 declared memory objects. While some of the declarations were uninitialized, many others were initialized with a variety of character and numeric literal expressions. The numeric literals were represented in either decimal or hexadecimal format, for example, `10` and `0xA` respectively. Extra overhead required to traverse the larger lookup table seemed to be unnoticeable. At some point, the overhead will become apparent when the lookup table contains several thousand nodes. Answering that question was out of the scope of this dissertation and the goal of delivering a working proof of concept type safety analysis tool. However, the anticipation here is that large bodies of source code will be segmented into thousands of functions to accomplish specific tasks by design. That said, the tool is designed to be capable of analyzing either a single function or a sub-set of all functions without losing its functionality.

The third limitation is the tool's current inability to reason about `struct` and `union` data types. This is because the `c2acl2` translator does not fully abstract these. When `c2acl2` was developed, the purpose of the abstraction was to verify secure data flow. Because most data flow analysis techniques are primarily concerned with finding those data objects in memory that influence the state of other memory data objects, the abstraction did not require data values. Since modeling the members of `struct` or `union` data structure was too complicated and was not required for the scope of the original `c2acl2` purposes, the abstraction for these objects remains to be finalized. Fortunately, `c2acl2` did fully abstract all other data objects which included their values.

10.4 Test Suite and Observed Performance

When a designing any tool, be it physical or software based, two fundamental questions must be answered. First, was the right tool built? Secondly, does the tool actually complete its task? The answer for both questions can be derived from extensive testing against a comprehensive test suite. For complete coverage, the test suite for this project needed to include all conditions that can lead to failure or, in the case of this dissertation, a violation of type safety.

The original test suite included several hundred files. It was compiled for another research project that investigated the validation of secure data flow. When constructed, the test suite intentionally contained numerous examples of type safety violations. Since most data flow validation procedures are primarily interested in what data objects residing in memory play a role in changing the state of other data stores in memory, the fact that several of the test files would produce unexpected results did not matter.

Listing 10.1 and its AST (Listing 10.2) are such examples. Within the source code and its respective AST, there are several expressions that can produce unexpected errors. For example, the first expression statement, `c1 = c2 + c1;`, contains an uninitialized

Listing 10.1: Simple C program that adds operands of mixed integer type

```
int main()
{
  char c1;
  unsigned char c2 = 'a';
  short s1 = 10;
  int i1 = 123;
  int i2 = -10;
  unsigned int ui1 = 1;

  c1 = c2 + c1;
  c1 = c2 + s1;
  c1 = i1 + s1;
  c1 = i1 + ui1;
  c1 = i2 + ui1;
  c1 = ui1;
}
```

addition operand, `c1`. Because of this, the evaluated result of the addition expression is likely to be unpredictable.

The tool was built in a stepwise fashion. The first step was to tackle the declarations and their initializations. This provided the required the necessary lookup tables. Then the collection of evaluation functions for each expression type were written in their entirety as single groups. This was done because of the disjoint typing requirements that operators placed on their operands. Whenever possible, those expressions with similar typing requirements would call the same set of type checking functions.

It became apparent during the development process of the type safety analysis tool, that the test suite did not provide complete coverage for detecting type safety violations. For example, the C99 standard states that for any given data object held in memory, the state (value) of that object can be changed at most one time in any given expression. The original test suite included basic cases for the post decrement and post increment operators that could lead to overflow or underflow conditions. However, it did not include examples of undefined state changes such as `x = x++ + y`. Whenever it was found that the test suite did not include a particular scenario the would lead to a type safety

Listing 10.2: AST for simple C program that adds operands of mixed integer type

```
(C2ACL2 (FILE "expAddVV")
 (
  ";***********************************************"
  "; Function Definition for function: main"
  ";***********************************************"
  (FUNC (INT )(ID "main" 1 ) NIL
   (BLOCK
    (DECL (CHAR )(ID "c1" 2 ) NIL)
    (DECL (UNSIGNED CHAR )(ID "c2" 3 )(INIT (LIT "'a'")))
    (DECL (SHORT )(ID "s1" 4 )(INIT (LIT 10)))
    (DECL (INT )(ID "i1" 5 )(INIT (LIT 123)))
    (DECL (INT )(ID "i2" 6 )(INIT (LIT -10)))
    (DECL (UNSIGNED INT )(ID "ui1" 7 )(INIT (LIT 1)))

    (EXPSTMT (ASSN (ID "c1" 2) (ADD (ID "c2" 3) (ID "c1" 2))))
    (EXPSTMT (ASSN (ID "c1" 2) (ADD (ID "c2" 3) (ID "s1" 4))))
    (EXPSTMT (ASSN (ID "c1" 2) (ADD (ID "i1" 5) (ID "s1" 4))))
    (EXPSTMT (ASSN (ID "c1" 2) (ADD (ID "i1" 5) (ID "ui1" 7))))
    (EXPSTMT (ASSN (ID "c2" 2) (ADD (ID "i2" 6) (ID "ui1" 7))))
    (EXPSTMT (ASSN (ID "c1" 2) (ID "ui1" 7))))
   )
 )" ; End of function main"
)
```

violation, one or more test files were added to the test suite for complete coverage. In every case during the development process, the tool performed as expected on those files that contained known type safety violations and those that did not. This was applicable and observable to both the original test files or those that had to be written as needed.

10.5 Future Work

The commercially viable C type safety analysis tool, Astrée provided the most validation of any single tool offered prior to this research. Astrée took 21 man years to develop as a team of no less than three scientists needed seven years to complete the project. By comparison, this project has been conducted by a single person for about half the time frame. As such, plenty of work remains.

In addition to adding reasoning capabilities for floating point numbers and the data types `struct` and `union` (see § 10.3 above), the tool is in desperate need of an graphical user interface (GUI). Currently without a GUI, the abstraction of the source code to be analyzed is physically inserted into the input parameter of the main analysis function call. A GUI would simplify this process by allowing a user to either enter a file name or select a file from a list as the input. From that point, the GUI would contain the necessary files to open and read the file into the tool. Moreover, a GUI could allow the tool to be further customized by allowing a user to specify the target and the scope of an analysis. For example, a feature for detecting buffer overflows resulting from poorly designed loop invariants is one of many possibilities. An example GUI feature would allow the user to fine tune the scope and depth of the analysis. This GUI feature is currently supported by Astrée. Finally, the GUI could let a user select how verbose the analysis output needs to be. In the current state, the output (Listing 10.3) of the tool is verbose as it includes every line of the abstract source code and the changes made to the annotated lookup table. It was designed this way to verify that the tool is working correctly. When analyzing thousands of lined of source code, it may be the case that the

analyst is only interested in seeing those lines of code that generate an error or warning statement.

Finally, the tool presented within this document is a proof of concept. The considerable effort expended to get to its current state has produced a tool that analyzes most declarations including `typedef` and `enum` declarations and the ability to reason about unary and binary expressions. More work is required for those conditional statements that can lead to one or more outcomes such as a conditional expression or the `if` and `if else` statements. A key question that must be answered is how should one track the alternative outcomes. This becomes evermore complicated by the depth of the scope of such nested constructs.

10.6 Final Observations

All things considered, the proof of concept has been successful. The logic built upon the knowledge of the conditions leading to type safety violations, the formalization of the static typing semantics of the language C, and the formal expressions of the condition for type safe operations appears to be sound. While plenty of work remains before the tool becomes fully functional and ready for some sort of public release, the prototype provides a solid foundation. As the tool matures, the goal is make it robust enough to be incorporated within other evaluation tools such as compilers. Realization of this end can be made possible with an investment of time. Moreover, the techniques used in this process can be adapted to programming languages other than C.

Listing 10.3: Example analysis output for simple C program that adds operands of mixed integer type

```
(TYPE-SAFETY-ANALYSIS
 ((FUNC (INT) (ID "main" 1) NIL (LINE 1))
  (BLOCK)
   (DECL (CHAR)(ID "c1" 2) NIL (LINE 2)
    (2 ("c1")((CHAR)(NOQUAL)(NOSTORE))(NIL)))
   (DECL (UNSIGNED CHAR)(ID "c2" 3)(INIT (LIT "'a'"))(LINE 3)
    (3 ("c2")((UCHAR)(NOQUAL)(NOSTORE))(97)))
   (DECL (SHORT)(ID "s1" 4)(INIT (LIT 10))(LINE 4)
    (4 ("s1")((SHORT)(NOQUAL)(NOSTORE))(10)))
   (DECL (INT )(ID "i1" 5 )(INIT (LIT 123))(LINE 5)
    (5 ("i1")((INT)(NOQUAL)(NOSTORE))(123)))
   (DECL (INT )(ID "i2" 6 )(INIT (LIT -10))(LINE 6)
    (6 ("i2")((INT)(NOQUAL)(NOSTORE))(-10)))
   (DECL (UNSIGNED INT )(ID "ui1" 7 )(INIT (LIT 1))(LINE 7)
    (7 ("ui1")((UINT)(NOQUAL)(NOSTORE))(1)))


   (EXPSTMT (ASSN (ID "c1" 2)
            (ADD (ID "c2" 3) (ID "c1" 2)))(LINE 8)
    (2 ("c1")((CHAR)(NOQUAL)(NOSTORE))(NIL)
       ("Error: rhs addition operand uninitialized")))
   (EXPSTMT (ASSN (ID "c1" 2)
            (ADD (ID "c2" 3) (ID "s1" 4)))(LINE 9)
    (2 ("c1")((CHAR)(NOQUAL)(NOSTORE))(107)))
   (EXPSTMT (ASSN (ID "c1" 2)
            (ADD (ID "i1" 5) (ID "s1" 4)))(LINE 10)
    (2 ("c1")((CHAR)(NOQUAL)(NOSTORE))(133)
       ("Error: value greater than valid range of type CHAR")))
   (EXPSTMT (ASSN (ID "c1" 2)
            (ADD (ID "i1" 5) (ID "ui1" 7)))(LINE 11)
    (2 ("c1")((CHAR)(NOQUAL)(NOSTORE))(124)))
   (EXPSTMT (ASSN (ID "c2" 2)
            (ADD (ID "i2" 6) (ID "ui1" 7)))(LINE 12)
    (2 ("c1")((CHAR)(NOQUAL)(NOSTORE))(NIL)
       ("Error: operand sign conversion error")))
   (EXPSTMT (ASSN (ID "c1" 2) (ID "ui1" 7))(line 13)
    (2 ("c1")((CHAR)(NOQUAL)(NOSTORE))(1)))
 )
)
```

Bibliography

[1] A. Aldelsbach, D. Alessandri, C. Cachin, S. Creese, Y. Deswarte, K. Kursawe, J. Laprie, D. Powell, B. Randell, J. Riordan, P. Ryan, W. Simmonds, R. Stroud, P. Verissimo, M. Waidner, and A. Wespi, "Conceptual Model and Architecture of MAFTIA (Malicious- and Accidental-Fault Tolerance for Internet Applications," University of Newcastle upon Tyne, Tech. Rep. MAFTIA deliverable D22, January 2003.

[2] AlephOne, "Smashing the stack for fun and profit," *PHRACK*, vol. 7(49), 1996. [Online]. Available: http://www.insecure.org/stf.smashstack.txt

[3] J. Alves-Foss, "C2ACL2 Translator Design Document," Computer Science Department, University of Idaho, Moscow, ID, USA, Tech. Rep., 2010.

[4] J. Alves-Foss and C. Taylor, "An analysis of the GWV security policy," in *In 5th Internat. Workshop on ACL2 Prover and Its Applications*, 2004, pp. 2–2004.

[5] American Standard Code for Information Interchange, *RFC 20: ASCII Format for Network Interchange*, American National Standards Institute Std. ANSI X3.4-1968, October 1969. [Online]. Available: http://tools.ietf.org/html/rfc20

[6] ANSI, *American National Standard for Information Systems—Programming Language C*, American National Standards Institute Std. X3.179-1989, 1989.

[7] A. Asperti and G. Longo, *Categories, Types, and Stuctures: An Introduction to Category Theory for the Working Computer Scientist*, ser. Foundations of Computing Series. Cambridge, MA: MIT Press, 1991.

[8] J. Backus, "The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference," In Proc. International Conf. on Information Processing, UNESCO, Clemson, SC, 1959, pp. 125–132.

[9] M. Barr and C. Wells, *Category Theory for Computing Science*, 2nd ed., ser. Prentice-Hall International Series in Computer Science. New York, NY: Prentice Hall, 1996.

[10] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker Blast: Applications to software engineering," *Int. J. Softw. Tools Technol. Transf.*, vol. 9, no. 5, pp. 505–525, Oct. 2007. [Online]. Available: http://dx.doi.org/10.1007/s10009-007-0044-z

[11] M. Bishop, *Computer Security: Art and Science*. Upper Saddle River, NJ: Pearson Education, 2003.

[12] ——, *Introduction to Computer Security*. Upper Saddle River, NJ: Pearson Education, 2005.

[13] S. Blazley and L. Xavier, "Mechanized semantics for the Clight subset of the C language," *Journal of Automated Reasoning*, vol. 43, no. 3, pp. 263–288, 2009.

[14] R. S. Boyer and J. S. Moore, *A Computational Logic*. New York, NY: Academic Press, 1979. [Online]. Available: http://www.cs.utexas.edu/users/boyer/acl.pdf

[15] D. Brumley, D. Song, and J. Slember, "Towards automatically eliminating integer-based vulnerabilities," School of Computer Science, Carnegie Mellon University, Tech. Rep. CMU-CS-06-136, March 2006.

[16] G. Cantor, "Contributions to the Founding of the Theory of Transfinite Numbers II," *Mathematische Annalen*, vol. 49, pp. 207–246, 1897.

[17] L. Cardelli, "Type Systems," in *Handbook of Computer Science and Engineering*. CRC Press, 1997, ch. 103.

[18] CERT, "CERT C Secure Coding Standard," Software Engineering Institute, Carnegie Mellon University, 1995 - 2011, last edited 11/10/2010. [Online]. Available: https://www.cert.org/certcc.html

[19] ——, "Integral Security - Secure Integer Library," Computer Emergency Readiness Team, Software Engineering Institute, Carnegie Mellon University, 1995 - 2011, last updated 11/18/2010. [Online]. Available: http://www.cert.org/secure-coding/IntegerLib.zip

[20] B. Chess and G. McGraw, "Static analysis for security," *Security Privacy, IEEE*, vol. 2, no. 6, pp. 76–79, 2004.

[21] S. Christey and R. A. Martin, "Vulnerability Type Distributions in CVE," May 2007, Document Version: 1.1. [Online]. Available: http://cve.mitre.org/docs/vuln-trends/vuln-trends.pdf

[22] A. Church, "A set of postulates for the foundation of logic," *Annals of Mathematics*, vol. XXXIII, pp. 346–366, 1932.

[23] ——, "A set of postulates for the foundation of logic," *Annals of Mathematics*, vol. XXXIV, pp. 839–864, 1933.

[24] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival, "Why does Astrée scale up?" *Form. Methods Syst. Des.*, vol. 35, no. 3, pp. 229–264, Dec. 2009. [Online]. Available: http://dx.doi.org/10.1007/s10703-009-0089-6

[25] R. Dannenberg, W. Dormann, D. Keaton, T. Plum, R. C. Seacord, D. Svoboda, A. Volkovitsky, and T. Wilson, "As-If Infinitely Ranged Integer Model," Software Engineering Institute, CERT Program, Carnegie Mellon University, Tech. Rep. CMU/SEI-2010-TN-008, April 2010, Second Ed.

[26] D. Delmas and J. Souyris, "ASTRÉE: from research to industry," in *Proceedings of 14th International Static Analysis Symposium/Workshop on Static Analysis*, ser. Lecture Notes in Computer Science, G. Filé and H. Riis-Nielson, Eds., vol. 4634. Berlin: Springer, August 2007, pp. 437–451.

[27] E. W. Dijkstra, "Notes on Structured Programming," 1997. [Online]. Available: http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF

[28] "BLAST: Berkeley Lazy Abstraction Software Verification Tool," Electrical Engineering and Computer Sciences, University of California, Berkeley, CA. [Online]. Available: http://mtc.epfl.ch/software-tools/blast/index-epfl.php

[29] D. Engler, B. Chelf, A. Chou, and S. Hallem, "Checking system rules using system-specific, programmer-written compiler extensions," in *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4*, ser. OSDI'00. Berkeley, CA, USA: USENIX Association, 2000, pp. 1–1. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251229.1251230

[30] D. Evans, *LCLint User's Guide 2.5*, University of Virginia, June 2003. [Online]. Available: http://www.splint.org/guide/guide.pdf

[31] D. Evans, J. Guttag, J. Horning, and Y. M. Tan, "LCLint: a tool for using specifications to check code," in *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, ser. SIGSOFT '94. New York, NY, USA: ACM, 1994, pp. 87–96. [Online]. Available: http://doi.acm.org/10.1145/193173.195297

[32] R. W. Floyd, "Assigning Meaning to Programs," pp. 19–32, 1967.

[33] J. Foster, "CQUAL: A tool for adding type qualifiers to C," 2004. [Online]. Available: http://www.cs.umd.edu/~jfoster/cqual

[34] J. S. Foster, *CQual User's Guide Version 0.991*, The Regents of the University of California., April 2007. [Online]. Available: http://www.cs.umd.edu/~jfoster/cqual/user-guide-0.991.pdf

[35] Free Software Foundation, Inc., "gcc 4.1.2," 51 Franklin St. Fifth Floor, Boston, MA 02110, released 2/13/2007. [Online]. Available: http://gcc.gnu.org

[36] J. Garreau, *Radical Evolution: The Promise and Peril of Enhancing Our Minds, our Bodies – and What it Means to be Human.* New York, NY: Doubleday (USA), 2005.

[37] J. Gennari, S. Hedrick, F. Long, J. Pincar, and R. C. Seacord, "Ranged Integers for the C Programming Language," Software Engineering Institute, CERT Program, Carnegie Mellon University, Tech. Rep. CMU/SEI-2007-TN-027, September 2007.

[38] J. A. Goguen, "A Categorical Manifesto," *Mathematical Structures in Computer Science*, vol. 1, pp. 49–68, 1991.

[39] M. J. Gordon, *The Denotational Description of Programming Languages: An Introduction.* New York, NY: Springer-Verlag, 1979.

[40] M. J. C. Gordon and T. Melham, *Introduction to HOL: a theorem proving environment.* Cambridge University Press, 1993.

[41] D. Grossman, "Type-safe multithreading in Cyclone," *Sigplan Notices*, vol. 38, no. 3, pp. 13–25, 2003.

[42] Y. Gurevich and J. K. Huggins, "The Semantics of the C Programming Language," *Selected papers from CSL'92* (Computer Science Logic)*, Springer Lecture Notes in Computer Science*, no. 702, pp. 274–308, 1993.

[43] L. Hatton, "EC-: A measurement based on safer subset of ISO C suitable for embedded system development," *Information and Software Technology*, vol. 47, no. 3, pp. 181–187, March 2003.

[44] J. Hickey, "Introduction to the Objective Caml Programming Language," September 2004. [Online]. Available: http://www.seas.upenn.edu/~cis500/cis500-f02/resources/ocaml-intro.pdf

[45] M. W. Hicks, G. Morrisett, D. Grossman, and J. Trevor, "Experience with safe manual memory-management in Cyclone," in *ACM International Symposium on Memory Management*, Vancouver, British Columbia, Canada, October 2004, pp. 73–84.

[46] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, Oct. 1969. [Online]. Available: http://doi.acm.org/10.1145/363235.363259

[47] G. J. Holzmann, *uno – static analysis tool for ANSI-C programs*. [Online]. Available: http://www.spinroot.com/uno/uno_man.pdf

[48] ——, "Static source code checking for user-defined properties," in *Proc. IDPT 2002*, Pasadena, CA, USA, 2002. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.13.1408

[49] M. Howard, "Safe integer arithmetic in C," February 2002. [Online]. Available: blogs.msdn.com/b/michael_howard/archive/2006/02/02/523392.aspx

[50] P. Hudak, J. Fasel, and J. Peterson, "A Gentle Introduction to Haskell," Yale University, Dept. of Computer Science, Tech. Rep. YALEU/DCS/RR-901, May 1996.

[51] W. A. Hunt, Jr., R. B. Krug, S. Ray, and W. D. Young, "Mechanized Information Flow Analysis through Inductive Assertions," in *Proceedings of the 8th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2008)*, A. Cimatti and R. B. Jones, Eds. Portland, OR: IEEE Computer Society, Nov. 2008, pp. 227–230.

[52] ISO/IEC, *Programming Languages—C*, International Organization for Standardization Std. ISO/IEC 9899:1990, 1990.

[53] ——, *C Programming Language*, International Committee for Information Technology Standards Std. ISO/IEC 9899:1999, 1999.

[54] ——, *Programming Language—C*, International Committee for Information Technology Standards Std. ISO/IEC 9899:2011, October 2011.

[55] ISO/IEC/ANSI, *Information technology — Programming languages — Ada: Annotated Ada Reference Manual*, Intermetrics, Inc. Std. ISO/IEC/ANSI 8652:1987, December 1995.

[56] S. P. Jones, Ed., *Haskell 98 language and libraries: the Revised Report*. Cambridge University Press, 1999.

[57] G. Kahn, "Natural semantics," in *STACS*, 1987, pp. 22–39.

[58] M. Kaufmann, P. Manolios, and J. S. Moore, *Computer-Aided Reasoning: An Approach*. Norwell, MA: Kluwer Academic Publishishers, 2000.

[59] ——, *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishishers (USA), August 2002.

[60] M. Kaufmann and J. S. Moore, "ACL2 Users Manual (ver. .3.3)," Austin, TX, USA. [Online]. Available: http://www.cs.utexas.edu/users/moore/acl2

[61] ——, "Documentation for ACL2 Version 4.3," Austin, TX, USA, July 2001. [Online]. Available: http://www.cs.utexas.edu/users/moore/acl2/current/acl2-doc-major-topics.html

[62] M. Kaufmann and P. Pecchiari, "Interaction with the Boyer-Moore Theorem Prover: A Tutorial Study Using the Arithmetic-Geometric Mean Theorem," *Journal of Automated Reasoning*, vol. 16, pp. 181–222, 1996.

[63] D. Keaton, T. Plum, R. C. Seacord, D. Svoboda, A. Volkovitsky, and T. Wilson, "As-if infinitely ranged integer model," Software Engineering Institute, Carnegie Mellon University, Tech. Rep. CMU/SEI-2009-TN-023, July 2009. [Online]. Available: http://www.sei.cmu.edu

[64] KernelJanitors (http://janitor.kernelnewbies.org/), "Smatch!!!" [Online]. Available: http://smatch.sourceforge.net

[65] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 2nd ed. Englewood Cliffs, NJ: Prentice Hall, Inc., March 1988.

[66] K. Krause and J. Alves-Foss, "On Designing an ACL2-Based C Integer Type Safety Checking Tool," in *5th NASA Formal Methods Symposium, NFM 2013, NASA Ames Research Center Moffett Field, CA, USA, May 14-16, 2013*, ser. Lecture Notes in Computer Science, G. Brat, N. Rungta, and A. Venet, Eds., vol. 7871. Berlin Heidelberg: Springer-Verlag, 2013, pp. 472–477.

[67] ——, "On Using ACL2 to Verify C Integer Type Safety: A Semi-state Endeavor," in *Cyber Security Symposium: Public-Private Partnerships*. Moscow, ID: Center for Secure and Dependable Systems, April 2013, pp. 11–20.

[68] ——, "On Checking C Source Code for Integer Type Safety: A Schema for Modeling Data Objects," 2014, Preprint submitted to *Journal of Symbolic Computation*. Available on request: krau1931@vandals.uidaho.edu.

[69] R. Kurzweil, *The Singularity is Near: When Humans Transcend Biology.* New York, NY: Penguin Group (USA), 2005.

[70] D. LeBlanc, "SafeInt," 2003. [Online]. Available: http://www.codeplex.com/SafeInt

[71] J. L. Lions, "ARIANE 5 Flight 501 Failure: Report by the Inquiry Board," Paris, France, 1996. [Online]. Available: http://sunnayday.mit.edu/accidents/Ariane5accidentreport.html.

[72] B. H. Liskov and J. M. Wing, "A Behavioral Notion of Subtyping," *ACM Transactions on Programming Languages and Systems*, vol. 16, pp. 1811–1841, 1994.

[73] R. P. Lyons, Jr., "Introduction to Section IV," in *The Avionics Handbook*, C. R. Spitzer, Ed. Boca Raton, FL: CRC Press, 2001.

[74] S. McCamant and G. Morrisett, "Evaluating SFI for a CISC Architecture," in *15th USENIX Security Symposium*, Vancouver, BC, Canada, August 2–4, 2006, pp. 209–224.

[75] J. McCarthy, "Recursive functions as symbolic expressions and their computation by machine (part I)," *CACM*, vol. 3, no. 4, pp. 184–195, 1960.

[76] R. Milner, "A Theory of Type Polymophism in Programming," *Journal of Computer and System Sciences – JCSS*, vol. 17, no. 3, pp. 348–375, July 1978.

[77] R. Milner, M. Tofte, R. Harper, and D. MacQueen, *The definition of Standard ML (revised).* MIT Press, 1997.

[78] MISRA, *Guidelines for the use of the C language in vehicle based software*, Motor Industry Research Association Std., April 1998. [Online]. Available: www.misra.org.uk

[79] ——, *Guidelines for the use of the C language in critical systems*, Motor Industry Research Association Std. MISRA-C:2004, October 2004. [Online]. Available: www.misra-c.com

[80] J. C. Mitchell, “Type inference with simple subtypes,” *Journal of Functional Programming*, vol. 1, no. 3, pp. 245–285, 1991.

[81] MITRE Corp., “Common Vulnerabilities and Exposures: The Standard for Information Security Vulnerability Names.” [Online]. Available: http://cve.mitre.org

[82] E. Moggi, “An Abstract View of Programming Languages,” University of Edinburgh, Laboratory for Foundations of Computer Science, Tech. Rep. ECS-LFCS-90-113, 1990.

[83] G. E. Moore, “Cramming more components onto integrated circuits,” *IEEE Solid-state Circuits Newsletter*, vol. 20, pp. 33–35, 2006, Reprinted from *Electronics*, vol. 38, no. 8, April 19, 1965, pp. 114–118.

[84] J. S. Moore, “Index.” [Online]. Available: http://www.cs.utexas.edu/users/moore/acl2/v3-4/acl2-doc-index.html

[85] ——, “Symbolic Simulation: An ACL2 Approach,” in *Formal Methods in Computer-Aided Design*, ser. Lecture Notes in Computer Science, G. Gopalakrishnan and P. Windley, Eds. Springer Berlin Heidelberg, 1998, vol. 1522, pp. 334–350. [Online]. Available: http://dx.doi.org/10.1007/3-540-49519-3_22

[86] ——, “Proving Theorems About Java-Like Byte Code,” in *Correct System Design, Recent Insight and Advances, (to Hans Langmaack on the occasion of his retirement from his professorship at the University of Kiel).* London, UK, UK: Springer-Verlag, 1999, pp. 139–162. [Online]. Available: http://dl.acm.org/citation.cfm?id=646005.673749

[87] ——, "Mechanized Operational Semantics: The M1 Story," 2008. [Online]. Available: http://www.cs.utexas.edu/users/moore/publications/talks/marktoberdorf-08/papers/main.pdf

[88] Y. Moy, N. Bjørner, and D. Sielaff, "Modular Bug-finding for Integer Overflows in the Large: Sound, Efficient, Bit-precise Static Analysis," Microsoft Research, Microsoft Corporation, One Microsoft Way, Redmond, WA, Tech. Rep. MSR-TR-2009-57, 2009.

[89] National Institute of Standards and Techology - Software Assurance Metrics and Tool Evaluation, "Source Code Analyzers." [Online]. Available: http://samate.nist.gov

[90] ——, "Source Code Analyzers." [Online]. Available: http://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html

[91] P. Naur, "Report on the Algorithmic Language ALGOL 60," *Commun. ACM*, vol. 3, no. 5, pp. 299–314, 1960.

[92] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "CCured: Type-Safe Retrofitting of Legacy Software," *ACM Transactions on Programming Languages and Systems*, vol. 47, no. 3, pp. 477–526, May 2005.

[93] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis: With 51 Tables.* Berlin, Heidelberg: Springer, 1999, 2005.

[94] M. Norrish, "An abstract dynamic semantics for C," Computer Laboratory, University of Cambridge, Tech. Rep., 1997.

[95] N. S. Papaspyrou, "A Formal Semantics for the C Programming Language," Ph.D. dissertation, National Technical University of Athens, Dept. of Electrical and Com-

puter Engineering, Div. of Computer Sci. Software Engineering Laboratory, Athens, Greece, February 1998.

[96] F. Parrennes, "CSUR Project," 2004. [Online]. Available: http://www.lsv.ens-cachan.fr/Software/csur

[97] J. Peterson and K. e. Hammond, "Report on the Programming Language Haskell, version 1.4," March 1997. [Online]. Available: http://www.csee.umbc.edu/courses/graduate/631/Fall2002/haskell.pdf

[98] F. Pfenning, "Lecture Notes on Type Safety, 15-312: Foundations of Programming Languages," Sept. 2004. [Online]. Available: http://www2.cs.cmu.edu/~fp/courses/312/handouts.06-safety.pdf

[99] B. C. Pierce, "A Taste of Category Theory for Computer Scientists," Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU-CS-90-113R, September 1990.

[100] ——, *Basic Category Theory for Computer Scientists*, ser. Foundations of Computing Series. Cambridge, MA: MIT Press, 1991.

[101] G. D. Plotkin, "A structural approach to operational semantics," Computer Science Dept., Aarhus University, Denmark, Tech. Rep. DAMI FN-19, 1981.

[102] "Bjarne Stroustrup Quotes," *Quotes.net*. STANDS4LLC, 2013. [Online]. Available: http://www.quotes.net/quote/9012

[103] Random House, Ed., *Random House Webster's College Dictionary*. Random House, Inc., 2001.

[104] S. Ray and W. A. Hunt, Jr., "Mechanized Certification of Secure Hardware Designs," in *Proceedings of the 8th International Workshop on Microprocessor Test and Verification, Common Challenges and Solutions (MTV 2007)*, M. S. Abadir,

L. Wang, and J. Bhadra, Eds. Austin, TX: IEEE Computer Society, Dec 2007, pp. 25–32.

[105] S. Ray and J. S. Moore, "Proof Styles in Operational Semantics," in *Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2004)*, ser. LNCS, A. J. Hu and A. K. Martin, Eds., vol. 3312. Austin, TX: Springer, Nov. 2004, pp. 67–81.

[106] S. Ray, T. Portlock, and R. Syzdek, "Modeling and Verification of Industrial Flash Memories," in *11th IEEE International Symposium on Quality Electronic Design (ISQED 2010)*, San Jose, CA, March 2010, pp. 705–712.

[107] S. Ray, W. A. Hunt, Jr., J. Matthews, and J. S. Moore, "A mechanical analysis of program verification strategies," *J. Autom. Reason.*, vol. 40, no. 4, pp. 245–269, May 2008. [Online]. Available: http://dx.doi.org/10.1007/s10817-008-9098-1

[108] G. D. Reis and B. Stroustrup, "A formalism for C++," International Committee for Information Technology Standards, American National Standards Institute, Washington, DC, USA, Tech. Rep. N1885=05-0145, ISO/IEC/JTC1/SC22/WG21, Oct 2005.

[109] H. G. Rice, "Classes of Recursively Enumerable Sets and Their Decision Problems," *Transactions of the American Mathematical Society*, vol. 74, no. 2, pp. 358–366, 1953. [Online]. Available: http://www.jstor.org/stable/1990888

[110] D. M. Ritchie, "The Development of the C Language," *ACM SIGPLAN Notices*, vol. 28, no. 3, pp. 201–208, March 1993, preprints of the Second ACM SIGPLAN History of Programming Language (HOPL II).

[111] D. Russionoff, M. Kaufmann, and E. Smith, "Formal Verification of Floating-Point RTL at AMD Using the ACL2 Theorem Prover," *17th IMACS World Congress: Scientific Computation, Applied Mathematics and Simulation*, July 2007.

[112] A. Sabelfeld and A. C. Myers, "Language-Based Information-Flow Security," *IEEE Journal On Selected Areas In Communications*, vol. 21, no. 1, p. 2003, 2003.

[113] D. Schmidt, "Denotational Semantics," 1986, out of print. [Online]. Available: http://people.cis.ksu.edu/~schmidt/text/densem.html

[114] D. Scott and C. Strachey, "Toward a Mathematical Semantics for Computer Languages," in *The Symposium on Computers and Automata*, J. Fox, Ed., vol. XXI, April 2008, pp. 19–46.

[115] M. V. Scovetta, "Yasca." [Online]. Available: http://www.scovetta.com/yasca.html

[116] R. C. Seacord, *Secure Coding in C and C++*. Pearson Education, 2006.

[117] "Rough Auditing Tool for Security (RATS)," Secure Software Inc. [Online]. Available: http://code.google.com/p/rough-auditing-tool-for-security

[118] G. Slabodkin, "Software glitches leave Navy Smart Ship dead in the water," *Government Computer News*, vol. 17, no. 17, July 1998. [Online]. Available: http://www.gcn.com/17_17/news/33727-1.html

[119] M. Spiser, *Introduction to the Theory of Computation*, 2nd ed. Boston, MA: Thomson Course Technology, 2006.

[120] G. L. Steele, Jr., *Common Lisp The Language*. Burlington, MA: Digital Press, 1984.

[121] B. Stroustrup, *The Design and Evolution of C++*. Addison-Wesley, 1996.

[122] R. D. Tennet, *The Principles of Programming Languages*. New York, NY: Prentice-Hall, 1981.

[123] The Open Group, "UNIX." [Online]. Available: http://www.unix.org

[124] J. Trevor, G. Morriset, D. Grossman, M. Hicks, J. Cheney, and Y. Wang, "Cyclone: a safe dialect of C," in *USENIX Annual Technical Conference*, June 2002, pp. 275–288.

[125] J. Von Neumann, "The Principles of Large-Scale Computing Machines," *IEEE Annals of The History of Computing*, vol. 3, pp. 263–273, 1981, originally printed in 1946.

[126] D. A. Wheeler, "Flawfinder." [Online]. Available: http://www.dwheeler.com/flawfinder

[127] A. K. Wright and M. Felleisen, "A Syntactic Approach to Type Soundness," *Information and Computation*, vol. 115, pp. 38–94, 1992.

Appendix A: The Syntax of C Types

This Appendix is a contiguous presentation of the typing specification of C types discussed in Chapter 6.

⟨ *c-type* ⟩ ::= ⟨ *object-type* ⟩
  | ⟨ *function-type* ⟩
  | ⟨ *incomplete-type* ⟩
⟨ *object-type* ⟩ ::= ⟨ *scalar-type* ⟩
  | ⟨ *aggregate-type* ⟩
  | ⟨ *union-type* ⟩
⟨ *scalar-type* ⟩ ::= ⟨ *arithmetic-type* ⟩
  | ⟨ *pointer-type* ⟩
⟨ *arithmetic-type* ⟩ ::= ⟨ *std-integer-type* ⟩
  | ⟨ *floating-type* ⟩
  | ⟨ *enumerated-type* ⟩
⟨ *standard-integer-type* ⟩ ::= `char`
  | ⟨ *standard-signed-integer-type* ⟩
  | ⟨ *standard-unsigned-integer-type* ⟩
⟨ *standard-signed-integer-type* ⟩ ::= `CHAR`
  | `SHORT`
  | `INT`
  | `LONG`
  | `LLONG`
⟨ *standard-unsigned-integer-type* ⟩ ::= `_BOOL`
  | `UCHAR`
  | `USHORT`
  | `UINT`
  | `ULONG`
  | `ULLONG`
⟨ *floating-type* ⟩ ::= ⟨ *real-float-type* ⟩
  | ⟨ *complex-type* ⟩
⟨ *real-float-type* ⟩ ::= `FLOAT`
  | `DOUBLE`
  | `LDOUBLE`
⟨ *complex-type* ⟩ ::= `FLOAT_COMPLEX`
  | `DOUBLE_COMPLEX`
  | `LONG_COMPLEX`
⟨ *enumerated-type* ⟩ | ⟨ *enumeration* ⟩
⟨ *enumeration* ⟩ | ⟨ *standard-integer-type* ⟩ . . .
    ⟨ *standard-integer-type* ⟩
⟨ *pointer-type* ⟩ ::= ⟨ *pointer* ⟩
⟨ *pointer* ⟩ | ⟨ *object-type* ⟩
  | ⟨ *function-type* ⟩
  | ⟨ *incomplete-type* ⟩

⟨ *aggregate-type* ⟩	::=	⟨ *array-type* ⟩
	|	⟨ *structure-type* ⟩
⟨ *array-type* ⟩	::=	⟨ *element-type* ⟩
⟨ *element-type* ⟩	::=	⟨ *object-type* ⟩
⟨ *structure-type* ⟩	::=	⟨ *member-type* ⟩
⟨ *member-type* ⟩	::=	⟨ *object-type* ⟩
⟨ *union-type* ⟩	::=	⟨ *member-type* ⟩
⟨ *incomplete-type* ⟩	::=	`void`
	|	⟨ *array-type* ⟩ (of unknown size)
	|	⟨ *structure-type* ⟩ (of unknown content)
	|	⟨ *union-type* ⟩ (of unknown content)
⟨ *function-type* ⟩	::=	⟨ *return-type* ⟩
⟨ *return-type* ⟩	::=	⟨ *object-type* ⟩
	|	`void`

Appendix B: Required Predicate and Valuation Functions for Expressing Static Typing Semantics

In Chapter 7, the 23 typing constraints imposed by operators on their operands as defined in the standard for C expressions were enumerated. This appendix provides a general overview of the respective predicate functions and other support functions required to formalize the static typing semantics of C expressions and statements.

B.1 Populating the Lookup Table Type Specifier Field

The type safety verification tool presented in this dissertation stores all declared objects in a lookup table (Listing B.1). The lookup table is created at the first instance of declaration statement in the code being analyzed and is appended with each subsequent declaration [68]. Each entity in the lookup table is a tuple containing the fields `ID#`, `NAME`, `TYPE`, `VALUE`, and `ERROR-WARNING`. The table is indexed by the unique integer value identifier assigned to each declared data object by the `c2acl2` translator [3].

The type field contains three sub-fields. The first holds the standardized type name derived from the `type-specifier` keywords used for an object declaration (Fig. 6.5). The second and third hold the `type-qualifier` and `storage-class-specifier` keywords used in the object declaration respectively (Fig. 6.18 and Fig. 6.19).

B.1.1 *getDeclaredArithType*(τ)

The function *getDecaredArithType*(τ) (Equation B.1) is used to populate the `TYPE-SPECIFIER` field in the tools primary lookup table whenever a declaration statement is evaluated. Because one or more type-specifier key words may be used in any order to specify a type, the tool generates a standardized type name for each arithmetic type. For example, the combination of type-specifiers to specify a signed long int is labeled by the analysis tool as `LLONG`:

- `long`
- `long int`
- `signed long`
- `signed long int`

Listing B.1: The four tuple of the `c2acl2` generated lookup table for each data object

```
(ID#
 ("NAME")
 ((TYPE-SPECIFIER)(TYPE-QUALIFIER)(STORAGE-CLASS-SPECIFIER))
 (VALUE)
 (ERROR-WARNING))
```

B.1.1.1 *getDeclaredRealType*(τ) and *getDeclaredIntType*(τ)

Generally, all declared types can be reduced to a single arithmetic-type or a collection of arithmetic-types. Since arithmetic types are real numbers, floats and integers, the function *getDecaredArithType*(τ) calls *getDecaredReal*(τ). If *getDecaredReal*(τ) does not return a floating-type, it calls *getDecaredInt*(τ) (Equation B.3) and if it fails to return a standard-integer-type, it returns `nil`

$$
\begin{aligned}
&\mathit{getDecaredArithType} : \mathbf{TYPE} \rightarrow \mathbf{TYPE} \\
&\mathit{getDecaredArithType} = (\lambda\tau.\ \mathbf{case}\ \tau\ \mathbf{of} \\
&\quad \mathit{getDeclaredReal} \quad \Rightarrow \quad \tau)
\end{aligned} \tag{B.1}
$$

$$
\begin{aligned}
&\mathit{getDecaredReal} : \mathbf{TYPE} \rightarrow \mathbf{TYPE} \\
&\mathit{getDecaredReal} = (\lambda\tau.\ \mathbf{case}\ \tau\ \mathbf{of} \\
&\quad \tau.\text{specifer} = \{\texttt{float}\} \Rightarrow \texttt{FLOAT} \\
&\quad \tau.\text{specifer} = \{\texttt{double}\} \Rightarrow \texttt{DOUBLE} \\
&\quad \tau.\text{specifer} = \{\texttt{long double}\} \Rightarrow \texttt{LDOUBLE} \\
&\quad \tau.\text{specifer} = \{\texttt{float _COMPLEX}\} \Rightarrow \texttt{FLOAT_COMPLEX} \\
&\quad \tau.\text{specifer} = \{\texttt{double _COMPLEX}\} \Rightarrow \texttt{DOUBLE_COMPLEX} \\
&\quad \tau.\text{specifer} = \{\texttt{long double_Complex}\} \Rightarrow \texttt{LDOUBLE_COMPLEX} \\
&\quad \tau.\text{specifer} = \{\texttt{float _Imaginary}\} \Rightarrow \texttt{FLOAT_IMAGINARY} \\
&\quad \tau.\text{specifer} = \{\texttt{double_Imaginary}\} \Rightarrow \texttt{DOUBLE_IMAGINARY} \\
&\quad \tau.\text{specifer} = \{\texttt{long double_Imaginary}\} \Rightarrow \texttt{LDOUBLE_IMAGINARY} \\
&\quad \mathbf{Otherwise} \Rightarrow \mathit{getDeclaredInt})
\end{aligned} \tag{B.2}
$$

$$
\begin{array}{llll}
\multicolumn{4}{l}{\textit{getDecaredInt} : \mathbf{TYPE} \rightarrow \mathbf{TYPE}} \\
\multicolumn{4}{l}{\textit{getDecaredInt} = (\lambda\tau.\ \mathbf{case}\ \tau\ \mathbf{of}} \\
\quad \tau.\text{specifer} & = & \{\texttt{_Bool}\} & \Rightarrow \texttt{_BOOL} \\
\quad \tau.\text{specifer} & = & \{\texttt{signed char}\} \vee \{\texttt{char}\} & \Rightarrow \texttt{SCHAR} \\
\quad \tau.\text{specifer} & = & \{\texttt{unsigned char}\} & \Rightarrow \texttt{UCHAR} \\
\quad \tau.\text{specifer} & = & \{\texttt{signed short int}\} \vee \{\texttt{signed short}\} \vee & \\
 & & \{\texttt{short int}\} \vee \{\texttt{short}\} & \Rightarrow \texttt{SHORT} \\
\quad \tau.\text{specifer} & = & \{\texttt{usigned short int}\} \vee \{\texttt{usigned short}\} & \Rightarrow \texttt{USHORT} \\
\quad \tau.\text{specifer} & = & \{\texttt{signed int}\} \vee \{\texttt{int}\} & \Rightarrow \texttt{INT} \\
\quad \tau.\text{specifer} & = & \{\texttt{unsigned int}\} & \Rightarrow \texttt{UINT} \\
\quad \tau.\text{specifer} & = & \{\texttt{signed long int}\} \vee \{\texttt{signed long}\} \vee & \\
 & & \{\texttt{long int}\} \vee \{\texttt{long}\} & \Rightarrow \texttt{LONG} \\
\quad \tau.\text{specifer} & = & \{\texttt{unsigned long int}\} \vee \{\texttt{unsigned long}\} & \Rightarrow \texttt{ULONG} \\
\quad \tau.\text{specifer} & = & \{\texttt{signed long long int}\} \vee \{\texttt{long long}\} \vee & \\
 & & \{\texttt{signed long long}\} \vee \{\texttt{long long int}\} & \Rightarrow \texttt{LLONG} \\
\quad \tau.\text{specifer} & = & \{\texttt{unsigned long long int}\} \vee & \\
 & & \{\texttt{unsigned long long}\} & \Rightarrow \texttt{ULLONG} \\
\quad \mathbf{Otherwise} & & & \Rightarrow \texttt{nil})
\end{array}
\tag{B.3}
$$

B.2 Truth Returning Functions for Types

Typing predicate functions are applied to data objects cataloged in the lookup table as the tool performs its evaluations. These functions return a truth value (true or false) based upon the declared type of the data object. Predicate functions of sub-types can be used in the construction of the predicate functions of the super-types.

B.2.1 *isInteger*(τ)

The predicate function *isInteger*(τ) (Equation B.4) is used to identify the standard-integer-type (Fig. 6.9) objects. Members of the standard-integer-type include all integer types including the members of the standard-signed-integer-type and of the standard-unsigned-integer-type.

Listing B.2: ACL2 isInteger

```
(defun isInteger (type)
 (let* ((integer_p
        (cond ((or
               (equal (car type) 'INT)
               (equal (car type) 'UINT)
               (equal (car type) 'LONG)
               (equal (car type) 'ULONG)
               (equal (car type) 'LLONG)
               (equal (car type) 'ULLONG)
               (equal (car type) 'CHAR)
               (equal (car type) 'SCHAR)
               (equal (car type) 'UCHAR)
               (equal (car type) 'SHORT)
               (equal (car type) 'USHORT)
               (equal (car type) 'BOOL))
              T)
             (t nil))))
  integer_p))
```

$$
\begin{aligned}
&\mathit{isInteger} : \mathbf{TYPE} \rightarrow \mathbb{B} \\
&\mathit{isInteger} = (\lambda\tau.\ \mathbf{case}\ \tau\ \mathbf{of} \\
&\quad \texttt{_BOOL} \mid \texttt{CHAR} \mid \texttt{SCHAR} \mid \texttt{UCHAR} \mid \texttt{SHORT} \mid \texttt{USHORT} \mid \\
&\quad \texttt{INT} \mid \texttt{UINT} \mid \texttt{LONG} \mid \texttt{ULONG} \mid \texttt{LLONG} \mid \texttt{ULLONG} \quad \Rightarrow \quad \texttt{T} \\
&\quad \mathbf{Otherwise} \quad \Rightarrow \quad \texttt{nil})
\end{aligned} \tag{B.4}
$$

The equivalent *isInteger*(τ) function written in ACL2 is shown in Listing B.2. The parameter `type` for the custom ACL2 function `isInteger` is extracted from the first element of the first type field whose first element is a single type symbol (name). If the extracted type symbol held in the lookup table matches the integer type in the condition statement, the isInteger predicate function returns `T` (true). Otherwise, it returns `nil` (false).

B.2.1.1 *isSignedInt*(τ) and *isUnsignedInt*(τ)

For type safety purposes, it is necessary to differentiate between signed and unsigned integers because of the potential for sign error when these different integer types are used

together in a single expression. The predicate function $\mathit{isSignedInt}(\tau)$ is used to identify the standard-signed-integer-type objects. For purposes of this dissertation, the integer type `CHAR` is considered to be signed.

$$\begin{aligned} &\mathit{isSignedInt} : \textbf{TYPE} \to \mathbb{B} \\ &\mathit{isSignedInt} = (\lambda\tau.\ \textbf{case}\ \tau\ \textbf{of} \\ &\quad \texttt{CHAR} \mid \texttt{SCHAR} \mid \texttt{SHORT} \mid \texttt{INT} \mid \texttt{LONG} \mid \texttt{LLONG} \ \Rightarrow\ \texttt{T} \\ &\quad \textbf{Otherwise} \ \Rightarrow\ \texttt{nil}) \end{aligned} \tag{B.5}$$

Whereas, the predicate function $\mathit{isUnsignedInt}(\tau)$ is used to identify the standard-unsigned-integer-type objects.

$$\begin{aligned} &\mathit{isUnsignedInt} : \textbf{TYPE} \to \mathbb{B} \\ &\mathit{isUnsignedInt} = (\lambda\tau.\ \textbf{case}\ \tau\ \textbf{of} \\ &\quad \texttt{_BOOL} \mid \texttt{UCHAR} \mid \texttt{USHORT} \mid \texttt{UINT} \mid \texttt{ULONG} \mid \texttt{ULLONG} \ \Rightarrow\ \texttt{T} \\ &\quad \textbf{Otherwise} \ \Rightarrow\ \texttt{nil}) \end{aligned} \tag{B.6}$$

Of course, the use of only one of these two functions is required to determine if an integer is signed or not. For example, if $\mathit{isSignedInt}(\tau)$ (Equation B.5) returns false, then the integer type being evaluated is `unsigned`.

B.2.1.2 *isBool*(τ)

At times, it is necessary to identify the standard-unsigned-integer-type `_BOOL` using the predicate function $\mathit{isBool}(\tau)$.

$$\begin{aligned} &\mathit{isBool} : \textbf{TYPE} \to \mathbb{B} \\ &\mathit{isBool} = (\lambda\tau.\ \textbf{case}\ \tau\ \textbf{of} \\ &\quad \texttt{_BOOL} \ \Rightarrow\ \texttt{T} \\ &\quad \textbf{Otherwise} \ \Rightarrow\ \texttt{nil}) \end{aligned} \tag{B.7}$$

B.2.2 *isFloat*(τ)

The predicate function $\mathit{isFloat}(\tau)$ (Equation. B.8) is used to identify floating-type objects. Floating type objects include members of the real-float-type and of the complex-type.

As such, it is constructed from the two predicate functions $isReal(\tau)$ (Equation. B.9) and $isComplex(\tau)$ (Equation. B.10).

$$\begin{aligned} &isFloat : \textbf{TYPE} \rightarrow \mathbb{B} \\ &isFloat = (\lambda\tau.\ \textbf{case}\ \tau\ \textbf{of} \\ &\quad isReal \quad \Rightarrow \quad \texttt{T} \\ &\quad \textbf{Otherwise} \quad \Rightarrow \quad isComplex) \end{aligned} \tag{B.8}$$

B.2.2.1 $isReal(\tau)$

The predicate function $isReal(\tau)$ is used to identify real-float-type (rational numbers) data objects.

$$\begin{aligned} &isReal : \textbf{TYPE} \rightarrow \mathbb{B} \\ &isReal = (\lambda\tau.\ \textbf{case}\ \tau\ \textbf{of} \\ &\quad \texttt{FLOAT} \mid \texttt{DOUBLE} \mid \texttt{LDOUBLE} \quad \Rightarrow \quad \texttt{T} \\ &\quad \textbf{Otherwise} \quad \Rightarrow \quad \texttt{nil}) \end{aligned} \tag{B.9}$$

B.2.2.2 $isComplex(\tau)$

The predicate function $isComplex(\tau)$ is used to identify complex-type (i.e., the numbers comprised of real and imaginary parts) data objects.

$$\begin{aligned} &isComplex : \textbf{TYPE} \rightarrow \mathbb{B} \\ &isComplex = (\lambda\tau.\ \textbf{case}\ \tau\ \textbf{of} \\ &\quad \texttt{FLOAT_COMPLEX} \mid \texttt{DOUBLE_COMPLEX} \mid \texttt{LONG_COMPLEX} \quad \Rightarrow \quad \texttt{T} \\ &\quad \textbf{Otherwise} \quad \Rightarrow \quad \texttt{nil}) \end{aligned} \tag{B.10}$$

B.2.3 $isArithmetic(\tau)$

Since arithmetic-types include the standard-integer-types and the floating-types, the predicate function $isArithmetic(\tau)$ (Equation. B.11) is constructed from $isInteger(\tau)$ (Equation B.4) and $isFloat(\tau)$ (Equation B.8).

Listing B.3: ACL2 isAritmeticType

```
(defun isArithmeticType (type)
 (let* ((isArithmetic_p
        (cond ((or
               (isInteger type)
               (isFloat type))
              T)
             (t nil))))
  isArithmetic_p))
```

$$
\begin{array}{l}
\textit{isArithmetic} : \mathbf{TYPE} \rightarrow \mathbb{B} \\
\textit{isArithmetic} = (\lambda\tau.\ \mathbf{case}\ \tau\ \mathbf{of} \\
\quad \begin{array}{lcl}
\textit{isInteger} & \Rightarrow & \mathtt{T} \\
\textit{isReal} & \Rightarrow & \mathtt{T} \\
\mathbf{Otherwise} & \Rightarrow & \mathtt{nil})
\end{array}
\end{array}
\tag{B.11}
$$

The equivalent *isArithmetic*(τ) function was written in ACL2 and named `isArithmeticType` is defined in Listing B.3.

B.2.4 *isPointer*(τ)

The predicate function *isPointer*(τ) is used to identify pointers to data objects. The C unary address indirection operator (`*`) is used to identify pointers.

$$
\begin{array}{l}
\textit{isPointer} : \mathbf{TYPE} \rightarrow \mathbb{B} \\
\textit{isPointer} = (\lambda\tau.\ \mathbf{case}\ \tau\ \mathbf{of} \\
\quad \begin{array}{lcl}
*\tau & \Rightarrow & \mathtt{T} \\
\mathbf{Otherwise} & \Rightarrow & \mathtt{nil})
\end{array}
\end{array}
\tag{B.12}
$$

The equivalent *isPointer*(τ) function written in ACL2 is named `isPointerType` and is shown in Listing B.4. Except for aggregate (array and `struct`), `union`, and pointer types, the length of the `TYPE-SPECIFIER` field (Listing. B.1) is one. The `TYPE-SPECIFIER` field representing a pointer is $>$ `1`. For example, a pointer-type is designated as `(TYPE-SPECIFIER *)`, `(TYPE-SPECIFIER * *)`, or `(TYPE-SPECIFIER * * *)`.

Listing B.4: ACL2 isPointerType

```
(defun isPointerType (type)
 (let* ((isPointer_p
        (cond ((and (> (length type) '1)
               ((or
                 (equal (second type) '*)
                 (equal (third type) '*)
                 (equal (fourth type) '*))))
               type)
              (t nil))))
  isPointer_p))
```

B.2.5 *isScalar*(τ)

Scalar-types include the arithmetic-types and the pointer-types. As such, the predicate function *isScalar*(τ) is constructed from the predicate functions *isArithmetic*(τ) (Equation B.11) and *isPointer*(τ) (Equation B.12).

$$\begin{aligned}
&\mathit{isScalar} : \textbf{TYPE} \rightarrow \mathbb{B} \\
&\mathit{isScalar} = (\lambda\tau.\ \textbf{case}\ \tau\ \textbf{of} \\
&\quad \mathit{isArithmetic} \quad \Rightarrow \quad \texttt{T} \\
&\quad \mathit{isPointer} \quad \Rightarrow \quad \texttt{T} \\
&\quad \textbf{Otherwise} \quad \Rightarrow \quad \texttt{nil})
\end{aligned} \tag{B.13}$$

B.2.6 *isArray*(τ)

An array is one of the two aggregate-types. The predicate function *isArray*(τ) (Equation B.14) is used to identify objects of array-type.

$$\begin{aligned}
&\mathit{isArray} : \textbf{TYPE} \rightarrow \mathbb{B} \\
&\mathit{isArray} = (\lambda\tau.\ \textbf{case}\ \tau\ \textbf{of} \\
&\quad \tau[\quad \Rightarrow \quad \texttt{T} \\
&\quad \textbf{Otherwise} \quad \Rightarrow \quad \texttt{nil})
\end{aligned} \tag{B.14}$$

B.2.7 *isStruct*(τ)

the other aggregate-type is the structure-type and the predicate function *isStruct*(τ) is used to identify objects of such type. A structure-type may or may not have a tag (identifier) represented by t and has members represented by π.

$$\begin{array}{l} \mathit{isStruct} : \mathbf{TYPE} \rightarrow \mathbb{B} \\ \mathit{isStruct} = (\lambda\tau.\ \mathbf{case}\ \tau\ \mathbf{of} \\ \quad \begin{array}{lcl} \mathtt{STRUCT}[t, \pi] & \Rightarrow & \mathtt{T} \\ \mathbf{Otherwise} & \Rightarrow & \mathtt{nil}) \end{array} \end{array} \tag{B.15}$$

At present, the type safety checking tool does not support structure types.

B.2.8 *isAggregate*(τ)

The predicate function *isAggregate*(τ) is used to identify aggregate-type objects and is built from the functions *isArray*(τ) (Equation B.14) and *isStruct*(τ) (Equation B.15).

$$\begin{array}{l} \mathit{isAggregate} : \mathbf{TYPE} \rightarrow \mathbb{B} \\ \mathit{isAggregate} = (\lambda\tau.\ \mathbf{case}\ \tau\ \mathbf{of} \\ \quad \begin{array}{lcl} \mathit{isArray} & \Rightarrow & \mathtt{T} \\ \mathit{isStruct} & \Rightarrow & \mathtt{T} \\ \mathbf{Otherwise} & \Rightarrow & \mathtt{nil}) \end{array} \end{array} \tag{B.16}$$

B.2.8.1 *isComplete*(τ)

The predicated function *isComplete*(τ) (Equation B.17) determines if an array or a structure is complete. At present, only complete arrays are identified by having size (number of elements or indexes) which either explicitly declared or implicitly declared via initialization. In the type checking tool, array declarations add two additional fields to the `TYPE-SPECIFIER` field of the type tuple, *i.e.,* (`TYPE-SPECIFIER` [n) where the integer n is the declared size.

$$\begin{array}{l} isComplete : \textbf{TYPE} \rightarrow \mathbb{B} \\ isComplete = (\lambda\tau.\ \textbf{case}\ \tau\ \textbf{of} \\ \quad \tau[n \wedge n \neq \texttt{nil} \quad \Rightarrow \quad \texttt{T} \\ \quad \textbf{Otherwise} \quad \Rightarrow \quad \texttt{nil}) \end{array} \tag{B.17}$$

B.2.8.2 *isEqualType*(τ_1, τ_2)

At times, it is necessary to know if two types are equal, such as with structure-types, before performing certain operations. The predicate function *isEqualType*(τ_1, τ_2) is used to check type equality.

$$\begin{array}{l} isEqualType : \textbf{TYPE} \times \textbf{TYPE} \rightarrow \mathbb{B} \\ isEqualType = (\lambda\tau.\ \textbf{case}\ \tau_1\ \textbf{and}\ \tau_2\ \textbf{of} \\ \quad \tau_1 = \tau_2 \quad \Rightarrow \quad \texttt{T} \\ \quad \textbf{Otherwise} \quad \Rightarrow \quad \texttt{nil}) \end{array} \tag{B.18}$$

B.2.9 *isUnion*(τ)

The union-type is the last of the object-types. The predicate function *isUnion*(τ) is used to identify objects of union-type. Like the structure-type, the union-type may or may not have a tag (identifier) represented by t and has members represented by π.

$$\begin{array}{l} isUnion : \textbf{TYPE} \rightarrow \mathbb{B} \\ isUnion = (\lambda\tau.\ \textbf{case}\ \tau\ \textbf{of} \\ \quad \texttt{UNION}[t, \pi] \quad \Rightarrow \quad \texttt{T} \\ \quad \textbf{Otherwise} \quad \Rightarrow \quad \texttt{nil}) \end{array} \tag{B.19}$$

At present, the type safety checking tool does not support `union` types.

B.2.10 *isObject*(τ)

The predicate function *isObject*(τ) (Equation B.20) is used to identify all objects belonging to the object-type category. Because the scalar-types, aggregate-types, and union-

types are members of object-type, functions $isScalar(\tau)$ (Equation B.13), $isAggregate(\tau)$ (Equation B.16), and $isUnion(\tau)$ (Equation B.19) are used to construct $isObject(\tau)$.

$$\begin{array}{l} isObject : \mathbf{TYPE} \rightarrow \mathbb{B} \\ isObject = (\lambda\tau.\ \mathbf{case}\ \tau\ \mathbf{of} \\ \quad \begin{array}{lcl} isScalar & \Rightarrow & \mathtt{T} \\ isAggregate & \Rightarrow & \mathtt{T} \\ isUnion & \Rightarrow & \mathtt{T} \\ \mathbf{Otherwise} & \Rightarrow & \mathtt{nil}) \end{array} \end{array} \tag{B.20}$$

B.2.11 *isVoid*(τ)

The predicate function $isVoid(\tau)$ is used to identify the incomplete-type `void`.

$$\begin{array}{l} isVoid : \mathbf{TYPE} \rightarrow \mathbb{B} \\ isVoid = (\lambda\tau.\ \mathbf{case}\ \tau\ \mathbf{of} \\ \quad \begin{array}{lcl} \mathtt{VOID} & \Rightarrow & \mathtt{T} \\ \mathbf{Otherwise} & \Rightarrow & \mathtt{nil}) \end{array} \end{array} \tag{B.21}$$

B.2.12 *isNull*(τ)

The predicate function $isNull(\tau)$ is used to identify Null pointer constants.

$$\begin{array}{l} isNull : \mathbf{TYPE} \rightarrow \mathbb{B} \\ isNull = (\lambda\tau.\ \mathbf{case}\ \tau\ \mathbf{of} \\ \quad \begin{array}{lcl} *\mathtt{0} & \Rightarrow & \mathtt{T} \\ \mathbf{Otherwise} & \Rightarrow & \mathtt{nil}) \end{array} \end{array} \tag{B.22}$$

B.2.13 *isQualified*(q)

The predicate function $isQualified(q)$ (Equation B.23) is used to identify type-qualifiers when applied to object-types. A type-qualifier is identified by one of the keywords `const`, `volatile`, and `restrict`. A type-qualifier keyword list is represented by q. Instead of returning true, the function returns the first qualifier keyword to represent true. If a qualifier keyword is not present, the function returns `NOQUAL` to represent false. The

comparable function written ACL2 is applied to the second sub-field `(TYPE-QUALIFIER)` of the type field contained in the data object lookup table (Listing. B.1).

$$\begin{array}{l} \mathit{isQualifed} : \mathbf{TYPE} \rightarrow \mathbb{B} \\ \mathit{isQualified} = (\lambda q.\ \mathbf{case}\ q\ \mathbf{of} \\ \quad \begin{array}{lcl} \texttt{CONST} & \Rightarrow & \texttt{CONST} \\ \texttt{VOLATILE} & \Rightarrow & \texttt{VOLATILE} \\ \texttt{RESTRICT} & \Rightarrow & \texttt{RESTRICT} \\ \mathbf{Otherwise} & \Rightarrow & \texttt{NOQUAL}) \end{array} \end{array} \tag{B.23}$$

B.2.13.1 *isCompatibleQualified*$(\tau_1 q_1, \tau_2 q_2)$

At times, it is necessary if two types are compatibly qualified. The predicate function *isCompatibleQualified*$(\tau_1 q_1, \tau_2 q_2)$ checks for compatibility.

$$\begin{array}{l} \mathit{isCompatibleQualified} : \mathbf{TYPE} \times \mathbf{TYPE} \rightarrow \mathbb{B} \\ \mathit{isCompatibleQualified} = (\lambda \langle \tau_1, \tau_2 \rangle.\ \mathbf{case}\ q_1\ \mathbf{and}\ q_2\ \mathbf{of} \\ \quad \begin{array}{lcl} q_1 = q_2 & \Rightarrow & \texttt{T} \\ \mathbf{Otherwise} & \Rightarrow & \texttt{nil}) \end{array} \end{array} \tag{B.24}$$

The same logic can be used for finding equal qualifiers. *isEquallyQualified*$(\tau_1 q_1, \tau_2 q_2)$ is one such function.

B.2.14 *isModifiable*(*identifier*)

The final predicate function with respect to object-types applies to the `lvalue` of individual memory objects. The function *isModifiable*(*identifier*) (Equation B.25) determines if a location value can be changed. This generally pertains to the LHS of an assignment expression expressed as an identifier (I). For example, if the LHS I is declared with the qualifier keyword `CONST` with an initialized value, then I is not modifiable. Let I be viewed as a constant, I.q be the type qualifier associated to I, and I.v be the value associated to I.

$$
\begin{array}{l}
\mathit{isModifiable} : \mathbf{CONST} \rightarrow \mathbb{B} \\
\mathit{isModifiable} = (\lambda I.\ \textbf{case}\ I\ \textbf{of} \\
\quad \begin{array}{lll}
I.\mathrm{q} \neq \mathtt{CONST} & \Rightarrow & \mathtt{T} \\
I.\mathrm{q} = \mathtt{CONST}\ \wedge\ I.\mathrm{v} = \mathit{nil} & \Rightarrow & \mathtt{T} \\
\textbf{Otherwise} & \Rightarrow & \mathtt{nil})
\end{array}
\end{array}
\tag{B.25}
$$

B.3 Type Returning Type Functions

At times, it is necessary to query the type of an object. This is trivial in ACL2 by using a lookup function on the data object lookup table that associates to a data object's unique identifier and returns it's type field.

B.3.1 *intPromote*(τ)

The integer promotion (§6.3.1.1 of the standard) rules state that if an `INT` can represent all values of the original integer type with less precision than that of an `INT`, the value of the original type is converted to an `INT`; otherwise, the original type is converted to an `UINT`. The goal of integer promotions is preserve any sign-age as well as the value (avoiding potential overflow/underflow or sign-age error conditions) of the intermediate results of an expression containing smaller precision integer operands. If the native integer type is 32-bits, then the process is straight forward as the values of all bitwise smaller signed and unsigned integer types are within the range of a signed 32-bit `INT`. The following *intPromote*(τ) function assumes 32-bit integer types.

$$
\begin{array}{l}
\mathit{intPromote} : \mathbf{TYPE} \rightarrow \mathbf{TYPE}' \\
\mathit{intPromote} = (\lambda \tau.\ \textbf{case}\ \tau\ \textbf{of} \\
\quad \begin{array}{lll}
\mathtt{SCHAR} & \Rightarrow & \mathtt{INT} \\
\mathtt{UCHAR} & \Rightarrow & \mathtt{INT} \\
\mathtt{SHORT} & \Rightarrow & \mathtt{INT} \\
\mathtt{USHORT} & \Rightarrow & \mathtt{INT} \\
\textbf{Otherwise} & \Rightarrow & \tau)
\end{array}
\end{array}
\tag{B.26}
$$

If the native integer type is 16-bits, however, then type SHORT is equivalent to type INT and type USHORT is equivalent to type UINT. Although the types USHORT and INT have the same bitwise size, the maximum value that can be held by an USHORT is 65535 and the maximum value can be held by an INT is 32767. Thus, if the value of an USHORT is less than or equal to 32767, then the USHORT is promoted to an INT. Otherwise, the USHORT is promoted to an UINT.

Because of bignums, ACL2 executables do not promote numeric values for computational purposes. While this facilitates value to type value range checking, it provides no notion of the promoted type of a C expression. The contrary, however, can be accomplished through a series of custom functions. The first function, get-type-rank (Listing B.5), written in ACL2 is based on a modified C integer ranking system defined in §6.3.1.1 of the standard.

In particular, the integer ranking system states that no two operands of the same type shall have the same rank. However, it is safe to say that if an expression contains two operands of the same type, then the expression will also be of the same type as its operands. Heuristically, if two operands of the same type are given the same rank, then the expression type has not changed. That is unless the operands are of an integer type less than the rank of type INT, then the expression type is promoted to type INT to accommodate integer promotions. Otherwise, the largest type rank will suffice for the usual arithmetic conversions. That said, the ACL2 promotion functions have included the real-types in ranking system based on bit-wise size such that functions will suffice the usual arithmetic conversions.

The function, get-type-rank, is called by function, get-type-rank-list (Listing B.6) which lists the ranks of all operands contained within an expression. The overall expression type is determined by two additional functions. The first, get-max-type-rank (Listing B.7) inputs the type rank list returned by function get-type-rank -list and returns the largest rank contained in the list. The function, get-dominate-

Listing B.5: ACL2 get-type-rank

```
(defun get-type-rank (type)
 (let*
  ((type-rank
   (cond ((equal type 'BOOL) 0)
         ((or
          (equal type 'CHAR)
          (equal type 'SCHAR)) 1)
         ((equal type 'UCHAR) 2)
         ((equal type 'SHORT) 3)
         ((equal type 'USHORT) 4)
         ((equal type 'INT) 5)
         ((equal type 'UINT) 6)
         ((equal type 'LONG) 7)
         ((equal type 'ULONG) 8)
         ((equal type 'LLONG) 9)
         ((equal type 'ULLONG) 10)
         ((equal type 'FLOAT) 11)
         ((equal type 'DOUBLE) 12)
         ((equal type 'LDOUBLE) 13)
         ((equal type 'FLOAT_COMPLEX) 14)
         ((equal type 'DOUBLE_COMPLEX) 15)
         ((equal type 'LDOUBLE_COMPLEX) 16)
         ((equal type 'FLOAT_IMAGINARY) 17)
         ((equal type 'DOUBLE_IMAGINARY) 18)
         ((equal type 'LDOUBLE_IMAGINARY) 19)
        (t nil))))
  type-rank))
```

`type` (Listing B.8) first determines if the maximum rank of an expression is less than the rank of type `INT`. In this case, the rank of type `INT` is `5`. Then, function `get-dominate-type` calls `get-type-from-type-rank` (Listing B.9) passing a type rank. If the rank is $<$ `5`, then the rank of `5` is passed to `get-type-from-type-rank` which return type `INT`. Otherwise, the rank is passed unaltered. The output of function `get-type-from-type-rank` is the reverse of function `get-type-rank` as it returns the type based on the rank set by `get-type-rank` based on type.

Listing B.6: ACL2 get-type-rank-list

```
(defun get-type-rank-list (type-list)
 (if (endp type-list) nil
  (let*
   ((fst (car type-list))
    (rst (cdr type-list))
    (type-rank-list
     (cons (get-type-rank fst)
           (get-type-rank-list rst))))
  type-rank-list)))
```

Listing B.7: ACL2 get-max-type-rank

```
(defun get-max-type-rank (max-type-rank type-rank-list)
 (if (endp type-rank-list) max-type-rank
  (let*
   ((fst (car type-rank-list))
    (rst (cdr type-rank-list))
    (rank
    (max max-type-rank fst)))
  (get-max-type-rank rank rst))))
```

Listing B.8: ACL2 get-dominate-type

```
(defun get-dominate-type (operand-type-list)
 (if (endp operand-type-list) nil
  (let*
   ((type-rank-list
     (get-type-rank-list operand-type-list))
    (max-type-rank
     (get-max-type-rank
      (car type-rank-list) type-rank-list))
    (dominate-type
     (cond
      ((or
       (equal max-type-rank 0)
       (equal max-type-rank 1)
       (equal max-type-rank 2)
       (equal max-type-rank 3)
       (equal max-type-rank 4))
      (get-type-from-type-rank 5))
     (t
      (get-type-from-type-rank  max-type-rank)))))
  dominate-type)))
```

Listing B.9: ACL2 get-type-from-type-rank

```
(defun get-type-from-type-rank (type-rank)
 (let*
  ((type
   (cond ((equal type-rank 0) 'BOOL)
         ((equal type-rank 1) 'CHAR)
         ((equal type-rank 2) 'UCHAR)
         ((equal type-rank 3) 'SHORT)
         ((equal type-rank 4) 'USHORT)
         ((equal type-rank 5) 'INT)
         ((equal type-rank 6) 'UINT)
         ((equal type-rank 7) 'LONG)
         ((equal type-rank 8) 'ULONG)
         ((equal type-rank 9) 'LLONG)
         ((equal type-rank 10) 'ULLONG)
         ((equal type-rank 11) 'FLOAT)
         ((equal type-rank 12) 'DOUBLE)
         ((equal type-rank 13) 'LDOUBLE)
         ((equal type-rank 14) 'FLOAT_COMPLEX)
         ((equal type-rank 15) 'DOUBLE_COMPLEX)
         ((equal type-rank 16) 'LDOUBLE_COMPLEX)
         ((equal type-rank 17) 'FLOAT_IMAGINARY)
         ((equal type-rank 18) 'DOUBLE_IMAGINARY)
         ((equal type-rank 19) 'LDOUBLE_IMAGINARY))))
  type))
```

B.3.2 $arithConv(\tau_1, \tau_2)$

The goal of the usual arithmetic conversions is to find a common real type for the operands and the result of an expression. The rules comprising the usual arithmetic conversions are described in §6.3.1.8 of the standard and are generally based on the bit-wise size of the types. The output of $arithConv(\tau_1, \tau_2)$ is the result of this type conversion process.

$$
\begin{array}{ll}
\multicolumn{2}{l}{arithConv : \mathbf{TYPE} \times \mathbf{TYPE} \rightarrow \mathbf{TYPE}'} \\
\multicolumn{2}{l}{arithConv = (\lambda\langle\tau_1, \tau_2\rangle.\ \mathbf{case}\ \tau_1\ \mathbf{and}\ \tau_2\ \mathbf{of}} \\
\quad (\tau_1 = \texttt{LDOUBLE})\ \vee\ (\tau_2 = \texttt{LDOUBLE}) & \Rightarrow\ \texttt{LDOUBLE} \\
\quad (\tau_1 = \texttt{DOUBLE})\ \vee\ (\tau_2 = \texttt{DOUBLE}) & \Rightarrow\ \texttt{DOUBLE} \\
\quad (\tau_1 = \texttt{FLOAT})\ \vee\ (\tau_2 = \texttt{FLOAT}) & \Rightarrow\ \texttt{FLOAT} \\
\quad intPromote = (\lambda\langle\tau_1', \tau_2'\rangle.\ \mathbf{case}\ \tau_1'\ \mathbf{and}\ \tau_2'\ \mathbf{of} & \\
\qquad (\tau_1' = \texttt{ULLONG})\ \vee\ (\tau_2' = \texttt{ULLONG}) & \Rightarrow\ \texttt{ULLONG} \\
\qquad ((\tau_1' = \texttt{LLONG})\ \wedge\ (\tau_2' = \texttt{ULONG}))\ \vee & \\
\qquad\ ((\tau_2' = \texttt{LLONG})\ \wedge\ (\tau_1' = \texttt{ULONG})) & \Rightarrow\ ((\texttt{ULONG} \subseteq \texttt{LLONG}) \rightarrow \texttt{LLONG}, \texttt{ULLONG}) \\
\qquad (\tau_1' = \texttt{LLONG})\ \vee\ (\tau_2' = \texttt{LLONG}) & \Rightarrow\ \texttt{LLONG} \\
\qquad (\tau_1' = \texttt{ULONG})\ \vee\ (\tau_2' = \texttt{ULONG}) & \Rightarrow\ \texttt{ULONG} \\
\qquad ((\tau_1' = \texttt{LONG})\ \wedge\ (\tau_2' = \texttt{UINT}))\ \vee & \\
\qquad\ ((\tau_2' = \texttt{LONG})\ \wedge\ (\tau_1' = \texttt{UINT})) & \Rightarrow\ ((\texttt{UINT} \subseteq \texttt{LONG}) \rightarrow \texttt{LONG}, \texttt{ULONG}) \\
\qquad (\tau_1' = \texttt{LONG}) \vee (\tau_2' = \texttt{LONG}) & \Rightarrow\ \texttt{LONG} \\
\qquad (\tau_1' = \texttt{UINT})\ \vee\ (\tau_2' = \texttt{UINT}) & \Rightarrow\ \texttt{UINT} \\
\qquad (\tau_1' = \texttt{INT})\ \wedge\ (\tau_2' = \texttt{INT}) & \Rightarrow\ \texttt{INT}))
\end{array}
\tag{B.27}
$$

B.3.3 $funcArgPromote(\tau_1, \ldots, \tau_n)$

While it is generally accepted that the type of function is that of it's return type, the evaluation of functions will require the static typing semantics of it's parameters (arguments). The function $funcArgPromote(\tau_1, \ldots, \tau_n)$ (Equation B.28) represents the default argument promotions defined in §6.5.2.2 of the standard and is based on the usual arithmetic conversions. Each τ_1 parameter of $funcArgPromote(\tau_1, \ldots, \tau_n)$ represents the type of the corresponding arguments ($\tau_1,\ \ldots,\ \tau_n$) for the function being evaluated.

$$
\begin{aligned}
&\mathit{funcArgPromote} : \textbf{TYPE} \times \textbf{TYPE} \rightarrow \textbf{TYPE}' \\
&\mathit{funcArgPromote} = (\lambda\tau.\ \textbf{case}\ \tau_i\ \textbf{of} \\
&\quad \texttt{LDOUBLE} \quad \Rightarrow \quad \texttt{LDOUBLE} \\
&\quad \texttt{DOUBLE} \quad \Rightarrow \quad \texttt{LDOUBLE} \\
&\quad \texttt{FLOAT} \quad \Rightarrow \quad \texttt{DOUBLE} \\
&\quad \textbf{Otherwise} \quad \Rightarrow \quad \mathit{arithConv}(\mathit{intPromote}))
\end{aligned} \tag{B.28}
$$

B.4 Truth Returning Functions for Literals

Several predicate and value adding functions are required to identify and give meaning to the literal values contained within a program. For example, when data objects are initialized while being declared, the initialization value is generally a literal value. If there is an initialization, that value is the first value assigned to the `(VALUE)` field of the data object lookup table (Listing. B.1).

B.4.1 *isDecimal*(*lit*)

The predicate function *isDecimal*(*lit*) returns `T` if the literal is presented as a decimal number n composed of one or more decimal digits: `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, and `9`. Otherwise, it returns `nil`. Contained within *isDecimal*(*lit*) is the function *prefix*(*lit*) (Equation B.31) which is explained in Section B.4.2 of this appendix.

$$
\begin{aligned}
&\mathit{isDecimal} : \textbf{LITERAL} \rightarrow \mathbb{B} \\
&\mathit{isDecimal} = (\lambda l.\ \textbf{case}\ l\ \textbf{of} \\
&\quad \mathit{prefix} = \emptyset \wedge n \quad \Rightarrow \quad \texttt{T} \\
&\quad \textbf{Otherwise} \quad \Rightarrow \quad \texttt{nil})
\end{aligned} \tag{B.29}
$$

The same logic is applicable to determine if a floating point number represented by f is expressed in decimal. f is comprised of the same decimal digits used to express n with the addition of a single decimal point (`.`). At present, the analysis tool does not reason about literals expressed as f.

Listing B.10: ACL2 example usage of `RATIONALP` to find a decimal numeric literal

```
(defun literal-value (lit)
 (cond ((rationalp lit) lit)
       (t nil)))
```

ACL2 has a collection of built in (standard Lisp) functions to recognize numbers. For example, `RATIONALP` recognizes real numbers and `INTEGERP` recognizes whole numbers. If `RATIONALP` is applied to the number 10,

$$(\texttt{RATIONALP}\ 10), \tag{B.30}$$

it returns `T` for true because *integers* $\subset$ *rationals*. It should be noted that `RATIONALP` does not work on floating point values, such as `0.5`. However, it can reason about fractional numbers, such as `1/2`. The C integer type safety checking tool distinguishes between decimal and hexadecimal numeric literals using `RATIONALP` in a larger function that extracts the decimal values of both numeric and character literals (Listing B.10).

The `RATIONALP` function, however, does not work on decimal numeric literals that have character suffix to indicate an intended type. For example, the numeric decimal literal value `10u` represented as `(LIT 10u)` has the suffix `u` to designate that the literal is an unsigned integer (`UINT`) and if applied to `RATIONALP`, the result would be `NIL` for false.

B.4.2 *prefix*(*lit*)

At times, a numeric literal has a prefix to that the value is represented in other than decimal (base 10). Likewise, a character or string literal may be prefixed the a single `L` to indicate the character or string is wide. The function *prefix*(*lit*) (Equation B.31) returns the set of alpha characters that prefix a numeric literal to represent `T`. The valid prefix

characters are `L` and `0`; and, the two valid prefix strings are `0x` and `0X`. If none of these characters are present, $\emptyset$ is returned to represent `nil`.

$$\begin{array}{l} \textit{prefix} : \textbf{LITERAL} \rightarrow \mathbb{B} \\ \textit{prefix} = (\lambda l.\ \textbf{case}\ l\ \textbf{of} \\ \quad \begin{array}{lcl} \texttt{0} & \Rightarrow & \texttt{0} \\ \texttt{l} \mid \texttt{L} & \Rightarrow & \texttt{L} \\ \texttt{0x} \mid \texttt{0X} & \Rightarrow & \texttt{0X} \\ \textbf{Otherwise} & \Rightarrow & \emptyset) \end{array} \end{array} \tag{B.31}$$

B.4.3 *isOctal*(*lit*)

The predicate function $isOctal(lit)$ returns `T` if the numeric literal is presented as an octal number. Otherwise, it returns `nil`. A octal number begins with the prefix `0` (a zero) and is followed by one or more octal digits: `0`, `1`, `2`, `3`, `4`, `5`, `6`, and `7`. Let o represent an octal number.

$$\begin{array}{l} \textit{isOctal} : \textbf{LITERAL} \rightarrow \mathbb{B} \\ \textit{isOctal} = (\lambda l.\ \textbf{case}\ l\ \textbf{of} \\ \quad \begin{array}{lcl} \textit{prefix} = \texttt{0} \wedge o & \Rightarrow & \texttt{T} \\ \textbf{Otherwise} & \Rightarrow & \texttt{nil}) \end{array} \end{array} \tag{B.32}$$

At present, the type safety analysis tool does not reason about octal numeric literals. This is because ACL2 removes leading zeros from numeric values. This can be corrected by altering the `c2acl2` translator to generate the alphabet character `O` instead of the numeric character `0` to prefix an octal literal.

B.4.4 *isHexadecimal*(*lit*)

The predicate function $isHexadecimal(lit)$ (Equation B.33) returns `T` if the numeric literal is presented as a hexadecimal number. Otherwise, it returns `nil`. A hexadecimal number begins with the prefix “`0x`” or “`0X`” and is followed by one or more hexadecimal digits: `0`,

1, 2, 3, 4, 5, 6, 7, 8, 9, a, A, b, B, c, C, d, D, e, E, f, and F. Let h represent a hexadecimal number.

$$
\begin{array}{lll}
\mathit{isHexadecimal} : \mathbf{LITERAL} \rightarrow \mathbb{B} & & \\
\mathit{isHexadecimal} = (\lambda l.\ \mathbf{case}\ l\ \mathbf{of} & & \\
\quad \mathit{prefix} = \texttt{0X} \ \wedge \ h & \Rightarrow & \texttt{T} \\
\quad \mathbf{Otherwise} & \Rightarrow & \texttt{nil})
\end{array} \tag{B.33}
$$

B.4.5 *isChar(lit)*

Character literals are either a singleton character enclosed in single quote marks (' ') or a string of one or more characters enclosed in double quotation marks (" "). The function $\mathit{isChar}(\mathit{lit})$ returns T if the literal is a single printable American Standard Code for Information Interchange (ASCII) [5] character. Otherwise, it returns nil|.

$$
\begin{array}{lll}
\mathit{isChar} : \mathbf{LITERAL} \rightarrow \mathbb{B} & & \\
\mathit{isChar} = (\lambda l.\ \mathbf{case}\ l\ \mathbf{of} & & \\
\quad ! \mid " \mid \# \mid \$ \mid \% \mid \& \mid ' \mid (\mid) \mid * \mid + \mid , \mid - \mid . \mid / \mid & & \\
\quad \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \mid : \mid ; \mid < \mid = \mid > \mid & & \\
\quad ? \mid @ \mid \texttt{A} \mid \texttt{B} \mid \texttt{C} \mid \texttt{D} \mid \texttt{E} \mid \texttt{F} \mid \texttt{G} \mid \texttt{H} \mid \texttt{I} \mid \texttt{J} \mid \texttt{K} \mid \texttt{L} \mid \texttt{M} \mid & & \\
\quad \texttt{N} \mid \texttt{O} \mid \texttt{P} \mid \texttt{Q} \mid \texttt{R} \mid \texttt{S} \mid \texttt{T} \mid \texttt{U} \mid \texttt{V} \mid \texttt{W} \mid \texttt{X} \mid \texttt{Y} \mid \texttt{Z} \mid [\mid \backslash \mid & & \\
\quad] \mid \hat{} \mid _ \mid \grave{} \mid \texttt{a} \mid \texttt{b} \mid \texttt{c} \mid \texttt{d} \mid \texttt{e} \mid \texttt{f} \mid \texttt{g} \mid \texttt{h} \mid \texttt{i} \mid \texttt{j} \mid \texttt{k} \mid \texttt{l} \mid & & \\
\quad \texttt{m} \mid \texttt{n} \mid \texttt{o} \mid \texttt{p} \mid \texttt{q} \mid \texttt{r} \mid \texttt{s} \mid \texttt{t} \mid \texttt{u} \mid \texttt{v} \mid \texttt{w} \mid \texttt{x} \mid \texttt{y} \mid \texttt{z} \mid & & \\
\quad \{ \mid | \mid \} \mid \tilde{} \mid \mathit{space} & \Rightarrow & \texttt{T} \\
\quad \mathbf{Otherwise} & \Rightarrow & \texttt{nil})
\end{array} \tag{B.34}
$$

The C2ACL2 translator translates character literals into a character string where a single character enclosed by single quotes is enclosed by double quotes, *e.g.,* the literal 'a' is represented as " 'a' " by the output of C2ACL2. Thus, the function written in ACL2 to identify character literals is part of a condition (Listing B.11) based on four known facts about the C2ACL2 translated character literal: it is a string, the length of the string is three, the first character of the string is a single quotation mark, and the last character of the string is a single quotation mark.

Listing B.11: ACL2 conditional identifying character literals to get ASCII decimal value

```
(cond ((and (stringp lit)
            (equal (length lit) 3)
            (equal "'" (subseq lit 0
                               (- (length lit) 2)))
            (equal "'" (subseq lit (- (length lit) 1)
                               (length lit))))
       (get-ascii-dec-val lit)))
```

B.4.6 *isWideChar*(*lit*)

The predicate function *isWideChar*(*lit*) returns `T` if the character literal is prefixed with `L`. Otherwise, it returns `nil`. Let c represent a character literal.

$$\begin{array}{l} \mathit{isWideChar} : \mathbf{LITERAL} \rightarrow \mathbb{B} \\ \mathit{isWideChar} = (\lambda l.\ \mathbf{case}\ l\ \mathbf{of} \\ \quad \mathit{prefix} = \mathtt{L}\ \wedge\ c \quad \Rightarrow \quad \mathtt{T} \\ \quad \mathbf{Otherwise} \qquad \Rightarrow \quad \mathtt{nil}) \end{array} \tag{B.35}$$

B.4.7 *isStringLit*(*lit*)

The predicate function *isStringLit*(*lit*) returns `T` if the literal is enclosed in double quotation marks. Otherwise, it returns `nil`. Let l represent a literal.

$$\begin{array}{l} \mathit{isStringLit} : \mathbf{LITERAL} \rightarrow \mathbb{B} \\ \mathit{isStringLit} = (\lambda l.\ \mathbf{case}\ l\ \mathbf{of} \\ \quad \texttt{"}l\texttt{"} \qquad\quad \Rightarrow \quad \mathtt{T} \\ \quad \mathbf{Otherwise} \quad \Rightarrow \quad \mathtt{nil}) \end{array} \tag{B.36}$$

ACL2 has the built in function `STRINGP` to recognize strings. For example,

$$(\texttt{STRINGP "hello world"}) \tag{B.37}$$

returns `T` because an ACL2 string object is enclosed in double quotation marks.

B.4.8 *isWideString*(*lit*)

The predicate function *isWideString*(*lit*) returns `T` if the string literal is prefixed with a `L`. Otherwise, it returns `nil`. Let s represent a string literal.

$$\begin{array}{l} \mathit{isWideString} : \textbf{LITERAL} \rightarrow \mathbb{B} \\ \mathit{isWideString} = (\lambda l.\ \textbf{case}\ l\ \textbf{of} \\ \quad \mathit{prefix} = \texttt{L} \ \wedge\ s \quad \Rightarrow \quad \texttt{T} \\ \quad \textbf{Otherwise} \qquad \Rightarrow \quad \texttt{nil}) \end{array} \tag{B.38}$$

ACL2 has a collection of built in functions to recognize characters and strings. For example,

$$(\texttt{STRINGP "hello world"}) \tag{B.39}$$

returns `T` because an ACL2 string is enclosed in double quotation marks.

B.5 Type Returning Literal Functions

At times, a literal value can indicate its type. The following functions assist the type safety analysis tool to reason about type.

B.5.1 *suffix*(*lit*)

The function *suffix*(`lit`) (Equation B.40) returns the set of valid alpha characters contained in the suffix of a numeric literal to indicate type. Otherwise, it returns `nil`. Valid suffix characters and their meanings are `f` or `F` for `FLOAT`, `l` or `L` for `LONG`, `u` or `U` for `UNSIGNED`, `ll` or `LL` for `LLONG`, `ul` or `UL` for `ULONG`, and `ull` or `ULL` for `ULLONG`.

$$
\begin{aligned}
&\textit{suffix} : \textbf{LITERAL} \rightarrow \textbf{TYPE}\\
&\textit{suffix} = (\lambda l.\ \textbf{case}\ l\ \textbf{of}\\
&\quad \texttt{f} \mid \texttt{F} \quad \Rightarrow \texttt{FLOAT}\\
&\quad \texttt{l} \mid \texttt{L} \quad \Rightarrow \texttt{LONG}\\
&\quad \texttt{u} \mid \texttt{U} \quad \Rightarrow \texttt{UNSIGNED}\\
&\quad \texttt{ll} \mid \texttt{LL} \quad \Rightarrow \texttt{LLONG}\\
&\quad \texttt{ul} \mid \texttt{UL} \quad \Rightarrow \texttt{ULONG}\\
&\quad \texttt{ull} \mid \texttt{ULL} \quad \Rightarrow \texttt{ULLONG}\\
&\quad \textbf{Otherwise} \quad \Rightarrow \texttt{nil})
\end{aligned} \tag{B.40}
$$

A `c2acl2` translation of a suffixed numeric literal is represented as the number with trailing alpha character(s). It is not recognized as a number or a string by any of the ACL2 built in functions. However, ACL2 has a built in function (`STRING`) that coerces any data type into a string type. For example, the output of

$$(\texttt{STRING 10u}) \tag{B.41}$$

is the string

$$\texttt{"10u"}. \tag{B.42}$$

Once coerced to a string type, ACL2 has a number of string [84] functions that employ pattern matching to find the legal numeric typing suffixes. The same logic applies to numeric prefixes.

B.5.2 *firstToRepresent*(*lit*, τ_1, ..., τ_n)

The function *firstToRepresent*$(lit, \tau_1, \ \ldots, \ \tau_n)$ (Equation B.43) Let n represent a numeric integer value and each τ_i represent an constant integer type value { `INT.MAX | LONG.MAX | LLONG.MAX | ULLONG.MAX` }. Let l represent a literal integer value and c represent a τ_1.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{firstToRepresent} : \mathbf{LITERAL} \times \mathbf{CONST} \rightarrow \mathbf{TYPE}} \\
\multicolumn{3}{l}{\mathit{firstToRepresent} = (\lambda l.\ \mathbf{case}\ l\ \mathbf{and}\ c\ \mathbf{of}} \\
\quad l \leq \texttt{INT.MAX} & \Rightarrow & \texttt{INT} \\
\quad l > \texttt{INT.MAX} \ \wedge \ l \ \leq \texttt{LONG.MAX} & \Rightarrow & \texttt{LONG} \\
\quad l > \texttt{LONG.MAX} \ \wedge \ l \leq \texttt{LLONG.MAX} & \Rightarrow & \texttt{LLONG} \\
\quad l > \texttt{LLONG.MAX} \ \wedge \ l \leq \texttt{ULLONG.MAX} & \Rightarrow & \texttt{ULLONG} \\
\quad \mathbf{Otherwise} & \Rightarrow & \texttt{ERROR})
\end{array}
\tag{B.43}
$$

The function $\mathit{firstToRepresentUnsigned}(\mathit{lit}, \tau_1,\ \ldots,\ \tau_n)$ uses the same logic as the signed counterpart $\mathit{firstToRepresent}(\mathit{lit}, \tau_1,\ \ldots,\ \tau_n)$. Let l represent a numeric integer value and c represent each τ_i that represents an constant unsigned integer type value { `UINT.MAX | ULONG.MAX | ULLONG.MAX` }.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{firstToRepresentUnsigned} : \mathbf{LITERAL} \times \mathbf{CONST} \rightarrow \mathbf{TYPE}} \\
\multicolumn{3}{l}{\mathit{firstToRepresentUnsigned} = (\lambda \tau.\ \mathbf{case}\ l\ \mathbf{and}\ c\ \mathbf{of}} \\
\quad l \leq \texttt{UINT.MAX} & \Rightarrow & \texttt{UINT} \\
\quad l > \texttt{UINT.MAX} \ \wedge \ l \leq \texttt{ULONG.MAX} & \Rightarrow & \texttt{ULONG} \\
\quad l > \texttt{ULONG.MAX} \ \wedge \ l \leq \texttt{ULLONG.MAX} & \Rightarrow & \texttt{ULLONG} \\
\quad \mathbf{Otherwise} & \Rightarrow & \texttt{ERROR})
\end{array}
\tag{B.44}
$$

B.6 Value Returning Literal Functions

B.6.1 *lengthOfString(string)*

The function $\mathit{lengthOfString}(s)$ returns the number of characters enclosed in double quotation marks plus `1` to include the null string terminal character.

$$\mathit{lengthOfString} : \mathbf{STRING} \rightarrow \mathbb{Z} \tag{B.45}$$

The $\mathit{lengthOfString}(s)$ function uses the formula

$$1 + \sum_{i=0}^{n} 1 \tag{B.46}$$

where n is the number of string characters. In ACL2, `lengthOfString`(s) can be realized by applying the built in function `LENGTH` to a string literal and adding `1` to its output.

For example,

$$\texttt{(+ (LENGTH "hello") 1)} \tag{B.47}$$

returns `5` accounting for the null string terminal character generally required for C functions applied to character arrays.

B.6.1.1 *lengthOf*(*string*)

The function *lengthOf*(s) is a modified version of *lengthOfString*(s) as it does not add `1` to the output to account for the null string terminal character. This function is used to determine the length of identifiers that should not be built of no more than 32 characters to be type safe.

B.6.2 *charToDecimalVal*(*lit*)

The function *charToDecimalVal*(*lit*) Of the 128 ASCII characters, 34 are non-printable control characters. Obviously, these control characters will not appear as a character literal enclosed in single quotes (`' '`). The presentation of *charToDecimalVal*(*lit*) in Equation B.48) has been abbreviated to preserve space. The ellipsis represent the omitted contiguous character set and their associated ASCII values. The variable c represents a character literal. The equivalent function written in ACL2 is presented in Listing B.12.

Listing B.12: ACL2 get-ascii-dec-val

```
(defun get-ascii-dec-val (lit)
 (cond ((equal '"' '" lit) 32)  ;; the blank space
       ((equal '"'!'" lit) 33)
       ((equal '"'\"'" lit) 34) ;; \ used to escape
                                ;; single double quotation mark
       ((equal '"'#'" lit) 35)
       ((equal '"'$'" lit) 36)
       ((equal '"'%'" lit) 37)
       ((equal '"'&'" lit) 38)
         ...
         ...
       ((equal '"'y'" lit) 121)
       ((equal '"'z'" lit) 122)
       ((equal '"'{'" lit) 123)
       ((equal '"'|'" lit) 124)
       ((equal '"'}'" lit) 125)
       ((equal '"'~'" lit) 126))
       (t nil))) ;; end get-ascii-dec-val
```

$$
\begin{array}{l}
\mathit{charToDecimalVal} : \textbf{LIRERAL} \rightarrow \mathbb{Z} \\
\mathit{charToDecimalVal} = (\lambda c.\ \textbf{case}\ c\ \textbf{of} \\
\quad \begin{array}{lll}
 & \Rightarrow & \mathsf{32}\ \ \texttt{/* the blank space */} \\
! & \Rightarrow & \mathsf{33} \\
\texttt{"} & \Rightarrow & \mathsf{34} \\
\# & \Rightarrow & \mathsf{35} \\
\$ & \Rightarrow & \mathsf{36} \\
\% & \Rightarrow & \mathsf{37} \\
\& & \Rightarrow & \mathsf{38} \\
\ldots & \ldots & \ldots \\
\ldots & \ldots & \ldots \\
\mathsf{y} & \Rightarrow & \mathsf{121} \\
\mathsf{z} & \Rightarrow & \mathsf{121} \\
\mathsf{y} & \Rightarrow & \mathsf{122} \\
\{ & \Rightarrow & \mathsf{123} \\
| & \Rightarrow & \mathsf{124} \\
\} & \Rightarrow & \mathsf{125} \\
\sim & \Rightarrow & \mathsf{126} \\
\textbf{Otherwise} & \Rightarrow & \texttt{nil})
\end{array}
\end{array}
\tag{B.48}
$$

B.6.3 *octToDecVal(lit)*

The *octVal*(*lit*) function returns the decimal value a literal represented in octal.

$$octToDecVal : \mathbf{LITERAL} \rightarrow \mathbb{N} \tag{B.49}$$

The *octToDecVal*(*lit*) function uses the formula

$$k = \sum_{i=0}^{n} (a_i \times 8^i) \tag{B.50}$$

where k is the octal integer literal and a_i is the octal digit being converted where the i is the position of the digit (counting from 0 for the rightmost digit). At present, this function is not used because ACL2 truncates leading zeros of numeric values. Thus, the octal value `01` is interpreted as the decimal integer value `1`. This should be an easy fix and can be accomplished by changing the prefix the `c2acl2` translation of an octal value from a `0` (zero) to either the upper case or lower case alpha character `O`.

B.6.4 *hexToDecVal(lit)*

The *hexToDecVal*(*lit*) function returns the decimal value a literal represented in hexadecimal.

$$hexToDecVal : \mathbf{LITERAL} \rightarrow \mathbb{N} \tag{B.51}$$

The *hexToDecVal*(*lit*) function uses the formula

$$k = \sum_{i=0}^{n} (a_i \times 16^i) \tag{B.52}$$

where k is the hexadecimal integer literal and a_i is the hexadecimal digit being converted where the i is the position of the digit (counting from 0 for the rightmost digit). If the upper bitwise precision of the machine integer is 64-bits, the $n = 15$ due to the fact that each a_i occupies 4-bits.

In ACL2, numeric hexadecimal values are preceded with `#x` and the decimal value can be derived using `LET`. For example,

$$\texttt{(LET ((n \#x3A)) n)} \tag{B.53}$$

will return the decimal value `58` that `n` represents.

B.7 Value to Type Range Functions

The final set of functions are used for type safety purposes. For example, C integer types are likely to enter an error condition when they are assigned a value that is outside of the valid minimum and maximum range for the integer type.

B.7.1 *isValidIntValue*(*val*, τ_1, τ_2)

The predicate function $isValidIntValue(val, \tau_1, \tau_2)$ returns `T` if a value lies within the valid range that can be represented by the integer type. Let v represent a numeric value, τ_1 represent one of the constant minimum values an integer type can represent ({ `CHAR.MIN` | `SCHAR.MIN` | `SHORT.MIN` | `USHORT.MIN` | `INT.MIN` | `UINT.MIN` | `LONG.MIN` | `ULONG.MIN` | `LLONG.MIN` | `ULLONG.MIN` }) and τ_2 represent the constant maximum value the corresponding integer type can represent ({ `CHAR.MAX` | `SCHAR.MAX` | `SHORT.MAX` | `USHORT.MAX` | `INT.MAX` | `UINT.MAX` | `LONG.MAX` | `ULONG.MAX` | `LLONG.MAX` | `ULLONG.MAX` }).

$$\begin{array}{l} isValidIntValue : \textbf{LITERAL} \times \textbf{CONST} \rightarrow \mathbb{B} \\ isValidIntValue = (\lambda v.\ \textbf{case}\ v,\ \tau_1,\ \textbf{and}\ \tau_2\ \textbf{of} \\ \quad v \geq \tau_1\ \wedge\ v \leq \tau_2 \quad \Rightarrow\ \texttt{T} \\ \quad \textbf{Otherwise} \qquad\quad \Rightarrow\ \texttt{nil}) \end{array} \tag{B.54}$$

B.7.2 *isValidRealValue*(*val*, τ_1, τ_2)

The precision of real numbers (floats) is often dependent on the problem being solved, most C programs are compiled with extensive run time checks for the floats. Because of this, the floats are not as susceptible to entering an error condition as are the integers. For example, if a float literal is invalid, an expectation to indicate that the value is not a number (Nan) is thrown. For completeness, however, the tool introduced in this dissertation performs valid range checking for the floats. The function $isValidRealValue(val, \tau)$ employs the same logic used by $isValidIntValue(val, \tau)$. In this case, τ_1 represents one of the constant minimum values an float type can represent ({ `FLOAT.MIN | DOUBLE.MIN | LDOUBLE.MIN | FLOAT_COMPLEX.MIN | DOUBLE_COMPLEX.MIN | LONG_COMPLEX.MIN` }) and τ_2 represent the constant maximum value the corresponding integer type can represent ({ `FLOAT.MAX | DOUBLE.MAX | LDOUBLE.MAX | FLOAT_COMPLEX.MAX | DOUBLE_COMPLEX.MAX | LONG_COMPLEX.MAX` }).

$$
\begin{array}{l}
isValidRealValue : \textbf{LITERAL} \times \textbf{CONST} \rightarrow \mathbb{B} \\
isValidRealValue = (\lambda v.\ \textbf{case}\ v,\ \tau_1, \textbf{and}\ \tau_2\ \textbf{of} \\
\quad n \geq \tau_1\ \wedge\ n \leq \tau_2 \quad \Rightarrow\ \ \texttt{T} \\
\quad \textbf{Otherwise} \qquad\quad \Rightarrow\ \ \texttt{nil})
\end{array} \tag{B.55}
$$

B.7.3 *isValidArrayIndex*(*index*, *size*)

The final example type safety predicate function is $isValidArrayIndex(index, size)$ (Equation B.56) that returns true if an array index is valid to the declared size of the array. This function is useful in detecting potential and actual array boundary errors. Remember, an array index value cannot be less than `0` and because array indexing begins at `0`, the index value cannot be greater than the declared `(size - 1)`. Let i represent an index value and n represent the declared array size.

$$\begin{aligned}
&\mathit{isValidArrayIndex} : \textbf{LITERAL} \times \textbf{LITERAL} \rightarrow \mathbb{B} \\
&\mathit{isValidArrayIndex} = (\lambda i.\ \textbf{case}\ i\ \textbf{and}\ n\ \textbf{of} \\
&\quad i \geq 0\ \wedge\ i \leq (n-1) \quad \Rightarrow \quad \texttt{T} \\
&\quad \textbf{Otherwise} \qquad\qquad\quad \Rightarrow \quad \texttt{nil})
\end{aligned} \tag{B.56}$$