Formal Methods for Secure Software Construction

by

Ben Goodspeed

A Thesis Submitted to
Saint Mary's University, Halifax, Nova Scotia
in Partial Fulfillment of the Requirements for
the Degree of Master of Science in Applied Science.

April, 2016, Halifax, Nova Scotia

Approved:	Dr. Stavros Konstantinidis Supervisor
Approved:	Dr. Diego Rojas Examiner
Approved:	Dr. Cristian Suteanu Examiner
Approved:	Dr. Peter Selinger Examiner
Date:	April 8, 2016

Abstract

Formal Methods for Secure Software Construction

by Ben Goodspeed

The objective of this thesis is to evaluate the state of the art in formal methods usage in secure computing. From this evaluation, we analyze the common components and search for weaknesses within the common workflows of secure software construction. An improved workflow is proposed and appropriate system requirements are discussed. The systems are evaluated and further tools in the form of libraries of functions, data types and proofs are provided to simplify work in the selected system. Future directions include improved program and proof guidance via compiler error messages, and targeted proof steps.

April 8, 2016

"There is no such thing as perfect security, only varying levels of insecurity."

Salman Rushdie

Acknowledgements

Thanks to Dr. Stavros Konstantinidis for supervising this work - providing no end of encouragement and freedom to learn.

Thanks to Drs Rojas, Suteanu, and Selinger for their help in making this thesis what it is today.

Thank you to Mayya Assouad for putting up with me while I did this, and reading countless drafts.

Thanks to my father Dan Goodspeed, my mother Kathy Goodspeed, my aunt Amy Austin and my uncle Wayne Basler for supporting me throughout my education, and always encouraging me.

Thank you to everyone at the writing center for helping with many drafts of this work.

Thanks as well to Rozina Darvesh, Nicole Carter, Colin Kennedy, Dr Laurie Ricker and Jason Colburne for reading drafts and helping me get into and through graduate school!

Contents

List of Figures

List of Tables

Glossary

Branch Coverage	Pattern matching covering all data type values
Build n.	A release of a piece of software
Build v.	To compile and package software sources into an executable package or library
CIC	Calculus of Inductive Constructions
Compile	Convert source code into executable machine code
CVE	Common Vulnerabilities and Exposures Database
Discretionary	A rule that is enforced depending on configuration
Dynamic	Analyses of the system while executing
Data Egress	The act of extracting data from a (possibly compromised) system
Fixed Value Quantification	Assertions of the behavior of a function for finitely many inputs
IDE	Integrated development environment
Hello World	A program that prints "hello world" to the screen
Hole	A gap in a program
Library	A collection of code packaged for re-use
Mandatory	A rule whose enforcement is critical in the system
PDD	Proof-driven development
Refactor	To improv software without changing functionality
SDLC	Software development lifecycle
Static	Analyses performed without executing the system

String	A sequence of characters, such as "string"
Tactic	A method used by proof assistants to complete proofs by transforming targets
Termination	Functions must terminate after a finite number of steps, by forcing recursive arguments to shrink
TDD	Test-driven development

To my mom, whom I miss everyday.

Chapter 1

Introduction

Computer security is a huge industry, one that costs consumers billions of dollars per year [1]. The notion of security risk spans many areas of technology use, and is not limited to the information technology sector. The Symantec study suggests the damages for a single breach can reach nearly 200USD per record, and with some incidents affecting hundreds of thousands of records; it is not a risk that can be ignored. The traditional area of focus in computer security is cryptography in its various forms [2–13]. However, computer security is also deeply connected to software development [14–17] and program/algorithm correctness [18–21].

As a society we place a huge amount of trust in our computer systems [22, 23], and fundamental insufficiencies in our definitions of security (and requirements generated for our software systems) [24] suggest that this is a danger to us.

This leads to the key questions motivating this research: *How are formal methods used in computer security? Where have they been successful, and where have they fallen flat?* We found several patterns and commonalities in a great deal of the works cited herein, and specifically, we find gaps in the common workflow (which we discuss in chapter 4). Those observations lead us to put forward the hypothesis: "**Dependent type systems enable '*Proof-driven development*', which can reduce the efforts required to construct software with provable properties**", which is described in detail in chapter 6.

The workflow (which we refer to as the "common workflow", figure 1.1) with the following stages has been broadly employed by the papers cited here.

1. Define security goals.
2. Produce specification or policy.
3. Produce model.
4. Prove the model meets the goals.
5. Implement a system based on the model.

FIGURE 1.1: The "Common Workflow"

1.1 Contributions and Organization

In this thesis, we provide the following contributions:

1. We introduce an alternative to the common workflow, which we call "Proof driven development" (PDD). Proof-driven development represents an extension to existing test-driven development workflows, combined with new technology made available by proof assistants and dependently typed languages. This contribution is described in chapter 6.
2. We evaluate a collection of existing languages on the basis of their utility within a PDD workflow. This evaluation includes extraction/compilation readiness, practicality (in the form of performance and utility), and expressive power as measured by their ability to universally quantify assertions. The selected language, Idris, as well as these evaluations, are discussed in chapter 5.
3. We implement a practical example of a program from the security domain, namely a login program. Along with a discussion of the design requirements of such a program, we provide an analysis of some typical existing versions of such programs in current use. We develop our solution in the chosen language, Idris, and prove several interesting properties about the behavior of our implementation. The implementation, the properties and their proofs are described in chapter 7.
4. We provide implementations in Idris for several utility functions and benchmark programs. These utilities (such as functions for computing permutations, cyclic shifts, regular expression matchers, and numeric data types) serve as building blocks for larger efforts. The benchmarks serve to justify the performance claims about the selected language. These contributions are presented in chapter 8.
5. We provide several mathematical models and related lemmas to facilitate proofs. For example, an Idris datatype representing homomorphisms as well as proofs of transitivity of composition of these types, is provided and discussed in chapter 9.

The remainder of the thesis is organized as follows:

1. Chapter 2 reviews and synthesizes the relevant topics surrounding formal methods usage in computer security, it serves as the literature review for this work.
2. Chapter 3 discusses the mathematical components and background material required for formal verification, a survey of the use of formal methods in different contexts, and acknowledges fundamental limitations (for example, we do not attempt to prove the hardware in which the software runs is secure).
3. Chapter 4 provides an analysis of the common themes in the works cited herein, a discussion of the remaining issues with the identified themes, and describes the rationale for the contributions summarized above.
4. Chapters 5 through 9 present the contributions of this work.
5. Chapter 10 summarizes the benchmark results and the savings achieved by the contributions given.
6. Chapter 11 discusses related works, indicating where the approaches are similar and where they differ.
7. Chapter 12 reiterates the key contributions, and lays out potential directions for future work.

1.2 Motivation for Formal Methods

Many security problems have subtle nuances that are difficult to explain or model precisely. Even when the problem is restricted to securing a single local system, the issue is complex and intractable. Network configurations permit additional categories of insecurity. Likewise, particular applications also permit new types of flaws (e.g. web servers, database management systems) [25].

The flaws are so numerous that dedicated tracking has become necessary to determine vulnerable versions of operating systems, software libraries, applications, network devices, cryptographic protocols and even programming languages themselves. One such repository is the Common Vulnerabilities and Exposures (CVE) database[1]. At the time of writing, the archive contained over 70,000 documented vulnerabilities going back only as far as 1999.

[1] https://cve.mitre.org/

Loscocco notes that mandatory security[2] – as defined in the Trusted Computer System Evaluation Criteria (the so-called "Orange Book") [26] – is "insufficient to meet the needs of either the Department of Defense or private industry as it ignores critical properties such as intransitivity and dynamic separation of duty" [24]. Loscocco further suggests that mandatory access controls, including cryptographic data protection policies, require support at the operating system (OS) level to be successful. His paper argues that security must exist at every level, from the OS to the application level, through to the network stack.

To quote a motivating posting to a popular encryption forum[3]:

> [I]n the early to mid 1980's, formal verification was clearly a dead end that would never get anywhere. A boss of mine once asserted (circa 1988) that there would never be a verified program that did anything terribly interesting, and at time he seemed right.
>
> Today, there is a formally verified microkernel called seL4[4], a formally verified C compiler called CompCert[5], a formally verified experimental web browser called Quark[6]... [27]

This informal post suggests a renewed interest in formal methods, and new approaches to circumvent a number of the challenges in formal verification of extremely large systems.

1.3 Definitions of security

Security can be defined broadly as "the degree of resistance to, or protection from, harm. It applies to any vulnerable and valuable asset, such as a person, dwelling, community, nation, or organization."[7] However, particular applications and groups require different aspects of security. For example: cryptography goals might include non-repudiation or data integrity. Systems with goals focused on confidentiality [28] require different definitions than those focused on data integrity [29], which further differ from those focused on information-flow [30]. Each of these prompted the need for their own formal definition of security.

[2] See the glossary for working definitions of selected security terms.
[3] gmane.comp.encryption.general
[4] http://ssrg.nicta.com.au/projects/seL4/
[5] http://compcert.inria.fr/
[6] http://goto.ucsd.edu/quark/
[7] http://en.wikipedia.org/wiki/Security

Different notions of the meaning of modeling [31–33] have caused confusion and mismatched expectations. These different definitions permit the precision necessary to make a formal, refutable, and provable argument about the security of the system.

1.4 Trust

Trust is an important concept from all Department of Defense (DoD) publications. Their guidelines published in the "rainbow series" [26, 34–38] are all based on producing/evaluating "trusted computing" systems.

Ken Thompson's Turing award lecture [23] centers on the chains of trust that we rely on in software. Trust is built by layers, and each tool used in the production of a system has its own network of trust assumptions. In the lecture, he discusses a "theoretical" quine (a self-outputting program) to be inserted in a compiler. The quine would detect the compilation of a login program, and insert a backdoor. It would also detect the compilation of a compiler, and insert both the login backdoor logic and the compiler tampering logic. A compiler produced by this bugged compiler would then indefinitely produce the backdoors in both classes of programs, leaving no source-level mechanism able to detect the tampering.

Similarly, Wadlow [22] talks about the network of trust required to operate any system. Every stakeholder, including internal developers/administrators and clients, is necessarily endowed with a certain degree of trust (sometimes explicitly and sometimes implicitly). He goes on to consider the implications of trusting a person or system; you now implicitly trust what or whom they trust. The article discusses certain mitigating techniques and presents case studies where they were not applied (and the issues that arose from missing such mitigating treatment). The popular "pretty good privacy" (PGP)[8] cryptosystem explicity deals with the issue of trust. The system uses trusted "introducers" (other users trusted by the system) to receive unknown keys in a more secure manner[9].

Formal verification permits a level of trust in software systems that was previously unobtainable. The remainder of this work discusses the challenges and opportunities present in the current landscape of formal verification for secure software development.

[8] http://www.pgp.com
[9] http://en.wikipedia.org/wiki/Web_of_trust

Chapter 2

Literature Review

To examine the role that formal methods have had on security, we begin with the history of formal modeling of computer security. Research began in the 1970's with Bell and Lapadula [28] in their seminal paper "Secure Computer Systems: Mathematical Foundations". They were among the first to formalize the issues at the core of the burgeoning security problem of the era. Their formalism is based on finite state machines [39, 40]. The key idea is that starting in a secure state and controlling transitions between states allowed rigorous proofs of security to be achieved. Many adaptations of this model followed and met with similar success [29, 30, 33, 41–43].

Initially the work was heavily concentrated around operating system research [28, 41, 44–47], and it was often funded by the US department of defense [26, 34–38]. Following the success of operating system security research (see section 2.1), we find similar applications of rigor to other specific disciplines and research lines. Particularly, we find many applications in cryptography (see section 2.2), processes for developing software & specification/design modeling (see section 2.3), quality assurance (see section 2.4), offensive security for attack planning strategies (see section 2.5), programming languages (see section 2.6), and secure string handling within languages (see section 2.7).

2.1 Operating systems and Hardware

Bell and Lapadula's original model [28] sufficed for the simpler time sharing systems that were prevalent at the time (predominately based on Multics). For modern operating system standards, such as Posix [48], and applications of secure operating system

design to embedded systems [49] their model was insufficient. This is in some cases due to theoretical limitations [39, 49] where the finiteness requirement of automata-based systems becomes questionable. Similarly, the models broke down for parallel and distributed systems [43] and when networked communications were explicitly modeled [34]. Recent research has seen success with a fully formal-system verified microkernel called seL4 [50].

The requirements and techniques used for security of operating systems are not entirely disparate from those employed in a cryptographic context.

Indeed, the physical tamper proofing requirements demanded by high-level security standards [51] and embedded system security [49] are virtually identical to those postulated by Goldwasser in describing the requirements for cryptographically guaranteed "one-time" (single run) programs [10].

2.2 Cryptography

Goldwasser [2] was among the first to attempt to bridge the challenging gap of bringing formal security proofs to the world of cryptography.

This area is challenging because cryptographic secrecy relies on the currently unproven conjecture that certain problems in NP are not in P, which would imply that $P \neq NP$, where P represents the class of problems requiring only "polynomial time" to decide, and NP represents those that can be decided in "polynomial time" by guessing a certificate.

Many cryptography systems rely on the difficulty of these problems [3–9, 11, 12, 40]. In particular, two such problems, the discrete logarithm problem and the integer factorization problem form the basis of RSA and DES, which are widely deployed. If efficient algorithms were to be found to solve these problems, the security of many cryptographic systems would be compromised. Whether or not such efficient algorithms exist, Goldwasser was able to prove other important aspects of cryptographic systems such as resilience to message replay attacks (sending the same message repeatedly without knowing the plain text contents) [3, 9]. Recent research trends have leveraged the power of proof assistants [15, 52–62] to provide automatic validation of these cryptographic security properties [12, 13].

Across these systems we find two basic approaches. The first is to leverage a declarative system like Prolog wherein facts are stated about the system, then the system searches

for a way to unify the facts with stated goals. We see this approach in several proofs of cryptographic protocol security [6–8]. The alternative approach is based on constructive logic (where the proofs are built by the user rather than searched for within the system via a unification process). The latter approach generally requires a proof assistant or a programming language with a strongly normalizing type system such as Martin-Löf [63] or the Calculus of Inductive Constructions [58], and we see it used in the verification of cryptographic primitives [12, 13].

2.3 Specification/Design

Specifications are the starting point of implementations. In some cases, the specifications alone are published prior to any systems that satisfy them. These specifications include cryptographic protocols and high level system designs [11, 32, 48, 51]. It is from these specifications that the developer (or the person proving properties of the system) builds his or her mental model. From the mental model a formal mathematical model is derived and it is the mathematical model that is referenced during proofs.

This area has been made rigorous by formalizing the way specifications are described and written [20], and how systems are modeled & designed [16, 64–66]. We saw the formalization of the processes used to create software in the late 1990s and early 2000s [17, 67–69].

2.4 Quality Assurance

Quality assurance, for our purposes, is the act of verifying that an implemented system (per the workflow described in section 1.1) meets the goals or policy defined by the specification phase.

This is often done by testing. Fundamentally, there is a divide between manual testing (where a person operates the machine/runs the program) and automated testing (where the machine consults an oracle [70] to determine the correct results). Formally, procedures (such as ISO 9000 [71], clean room [72], and waterfall [73]) have been used to facilitate manual testing and apply rigor to a human endeavor. Semi-recently, agile software development [16, 17, 67, 69] has pushed for new processes like test-driven development (TDD) [68] to shift more of the work onto the computer. This has had a positive effect on defect rates, in some cases reducing the number of defects by 50% [74].

A parallel research track is finding appropriate test cases, usually done by "fuzzing" (testing by generating a large quantity of inputs). Initially, these inputs were generated at random, or guided by rules unrelated to the structure of the program being tested [21]. This approach was improved by Earl et al. [75], and separately by Holler [21] who used the formal grammar of the target language to help generate intelligent test inputs. Historically, these types of analyses fall into the static or dynamic analysis categories, though recent work has blurred the lines between the two. Static analysis uses the structure of the code (or more commonly the structure of the abstract syntax tree the compiler produces) to predict bugs. Typically, these types of bugs are found by analyzing things like uncovered execution paths and focusing on the areas of greatest complexity (on the idea that greater complexity implies the system is more likely to have bugs) [18, 76]. Dynamic analysis focuses on the output of the program at runtime [21, 75, 77].

2.5 Attack Planning

A popular technique in security evaluation, "penetration testing" is to attempt to break into one's own systems [21, 78–80]. Historically, this was performed ad-hoc, or based on individual experience/intuition [81]. However, recent research shows that the task of attack planning can be (partially) reduced to a decision & optimization problem [78, 79]. The key ideas here have been to explicitly model the uncertainty of the "opponent's" state. While loosely based on the finite state machine models from the historic Bell LaPadula [28] formalization, it requires a Markov style decision procedure [82, 83] and the goal is not to provide security but to compute an optimal strategy to break it.

2.6 Programming Languages

At first glance, the programming languages component makes up the last component in the development workflow described in section 1.1. However, a substantial amount of research has gone into building languages and systems that support security features by default [14, 84–88]. Likewise, work has gone into adding support to existing languages (often accomplished by adding annotations) [85, 87]. Finally, new paradigms of programming like "dependent types" [54, 60–63, 88–91], proof by "tacticals" [15, 56, 57, 61, 92], and reasoning via "separation logic" [53, 93, 94] have been leveraged to improve security.

The theoretical framework of a programming language defines the limits of what can be described by the primitives of the language. In the earliest languages (assembly languages), the framework was dictated by the logic gate structures of the hardware on which they ran [95]. Early higher level languages (ones that require translation or compilation to run as machine code) such as LISP and Fortran were based on very early formalisms of the lambda calculus[96] and Turing machines [97].

More recent advances in programming language theory and type theory gave rise to theories like the calculus of inductive constructions (the CIC) [58], and Martin-Löf type theory [63], itself a subset of the CIC. These theories gave rise to dependent types [54, 90, 91], which form the basis of the systems Agda [60], Idris [61], Coq [56], and Cayenne [62].

These latter languages/systems (Coq, Isabelle, Agda, Cayenne and Idris) belong to a category of languages known as *proof assistants*. This is because they are capable of encoding mathematical logic and formal proofs directly within them. In 2008, the famous "4 color theorem" was indeed proven and formally verified with the Coq proof assistant [98]. Originally, these systems were built to codify and assist proofs on mathematical structures. As such, the older variants (such as Coq, Isabelle) tend to provide more support to users attempting to prove properties of things like Peano arithmetic, group theory and higher order logic [15, 56, 99]. In order to accomplish the tasks most large scale programs require (such as input/ouput, or mutable state/data structures), complex additional libraries must be added to these systems including elements like separation logic [53, 93] and monad constructs [100].

Several of these proof assistant languages, including Coq and Idris support dependent types. These are data types that are defined in terms of the values of other types. A classical example of this is the vector data type, wherein the number of elements in the vector is a part of the type itself. More formally, a dependent type is a family of types, each indexed by the dependent value. This is distinct from parameterized types, such as generics in Java and C# (e.g. a vector of real numbers), where the new type depends on another type (e.g. the real numbers), not the value of the other type.

A substantial amount of effort is required to bring software engineering to the level of mathematical certainty. According to Ricketts, in his 2009 paper "Automating Formal Proofs for Reactive Systems" [14]:

> *"Implementing the kernel for the seL4 [50] verified OS took only 2 person months, while verifying those roughly 9,000 lines of C code required over 20 person years."*

The newer systems like Agda, Cayenne and Idris have tried to start with a more "industry friendly" (versus mathematician friendly) basis [60, 61]. As such, they require less development effort to do common monad-based computations [54], including side effects (such as mutable data structures) and input/output.

Despite the effort required, a number of successes in "real world" industry-relevant programs have been achieved such as: seL4, the verified microkernel [50]; CompCert, the verified C compiler [101]; Quark, the verified web browser [84]; and verified cryptographic systems [12, 13].

2.7 String Handling

String handling is an important part of practical programming. Nearly all large scale programs interact with strings in one way or another. A large number of security vulnerabilities[1] have been the direct result of unsafe string handling in the C programming language.

Many attempts have been made to solve the problem, even within the C family of languages. For example C++ allows both the C-style strings and an object oriented version. This causes mismatches of libraries and has caused conversion issues.

There are many representations of strings both in mathematics and in practical implementations. Each one has different strengths and weaknesses. Likewise, some lend themselves more readily to building rigorous libraries than others. All of these representations and properties and libraries suggest a single core of mathematical structure that is available regardless of representation, and it is on this core that we can build reliable string handling.

Strings are used heavily in industry e.g. all web applications render templates, console programs use them to produce output and consume input, etc. However, they are also important theoretically, as automata theory (including deterministic, non-deterministic and regular varieties) essentially deals in words over an alphabet.

[1] https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=string

With the recent advances in dependent type theories and proof assistants, we are now in a position to formalize and automate the verification of string representations and properties. We would expect strings to be a highly developed aspect of proof assistants and of general purpose languages that support dependent types. Surprisingly, this is not the case. Many such proof assistants merely delegate these things to the primitives of the implementation language, and focus their efforts on other aspects of mathematics.

Each programming system represents strings differently. Some languages consider them second class citizens (notably C), others treat them as first class entities. These choices dictate the efficiency of the programs that use strings, the ease with which programs may manipulate strings, and the ability of the user (or the system itself) to reason about the contents or state of strings.

Chapter 3

Background

3.1 Building Blocks

This section describes the mathematical tools used to build and evaluate formal models. Many of the models described in this thesis are composed of many of the following basic components. In more modern formalisms such as proof assistant systems, these building blocks are deeply buried, and to do high level work with the systems deep knowledge of these components is not always required. They are described here for the sake of completeness.

3.1.1 Lattices

A lattice structure frequently arises from the natural ordering occurring in permissions and classification levels. The hierarchy of roles described by Bell and LaPadula in their landmark 1973 paper "Secure Computer Systems: Mathematical Foundations" form a lattice [28].

Similarly, Denning's paper on secure information flow [102] shows that the operation "flow" (the movement of information between objects in the system) forms a bounded lattice, defined as:

> "a set of objects (files etc), processes (programs running on behalf of some agent); clearances; a mechanism to determine the resulting clearance level/-classification of a combination of two objects; and a flow relationship operator (indicating the permission of information to flow from object A to B)." [102]

McLean's Algebra of Security [103] develops a boolean algebra supporting mandatory and discretionary access control. This particular structure forms a "distributive lattice" (in the way that users and their permissions can be combined).

A textbook on lattice theory, "General Lattice Theory" [104] discusses the axioms and properties required of lattices as well as their evolution from boolean algebras. Lattices are useful in formal modeling because they allow a set of non-numeric disjoint elements (for example users and classifications) to be combined and ordered as though they were numeric.

3.1.2 Matrices

General access control rules are often stored in matrices. Typically rows and columns represent subjects (users/processes) and objects (files/database rows), and store a list of capabilities (reading/writing) that are currently permitted. This is used both for mandatory (fundamental to the operation of the system) and discretionary access controls.

Bell and LaPadula's seminal model [28] stores access control lists in matrices. Brewer and Nash's Chinese Wall security protocol [42] stores access lists in a matrix indicating permissions to interact with other conflict of interest classes. On the other hand, Graham and Denning [47] store permissions such as "creation" and "deletion" in a matrix of subjects and objects. Harrison, Ruzzo and Ullman [45] develop a protection scheme for the secure control of the access matrix described by Graham and Denning. Landwehr's survey of formal models [105] discusses and compares some of the above matrix-based implementations.

Sarraute [78] used partially observable Markov decision processes to model unknown state and scanning process for use in attack planning and penetration testing. In these hidden Markov models (and in general Markov systems) the state transition probability is stored in a matrix. Solving the system of matrix equations for the optimal policy is a computationally intensive (cubic) operation. As such, alternative mechanisms such as dynamic programming have been employed [83].

Often a matrix is a natural way to store 2-dimensional data in a software system. However on some occasions (such as described above for the Markov model), the matrix-based formulation permits general results from linear algebra to be used in a new setting. The Markov case above is an example of this crossover. If the system was modeled in another

way, then it is possible the use of iterative methods to find approximate solutions may not have been found.

3.1.3 Finite State Machines

Beginning with Bell and LaPadula's early work [28] focusing on secure states and secure transitions between them, many security models have been formalized as finite state machines.

McLean [32] discusses Bell's model and points out some of the challenges in defining the scope of what is protected/secure, and the limitations of the theorems arising from those definitions.

Biba's work [29] is of much the same structure and formalism as Bell's. Where Bell focused on confidentiality, Biba focused on integrity. However, this change was largely accomplished by inverting the direction of the lattice structure described in section 3.1.1, without changing the overall structure of the state nor the transition function.

Clark and Wilson's model [41] also focuses on transitions between states. These are defined by the process, the user and the data (constrained or unconstrained). The model then ensures the "internal" and "external" integrity and consistency of the system is maintained.

Benson [43] developed a centralized, parallel, and distributed (CPD) model with similar properties to Bell and LaPadula's. His model adds assurance and the required mechanisms to confirm that all interleaving of instruction sequences (this is especially important in the parallel case) result in a secure state throughout the operation of the system. Other variants of the model (specifically the centralized and distributed cases) deal with models very similar to Bell's, save for the number and location of the "reference monitor(s)".

Lotz [49] extends the work begun by Bell and LaPadula into a model capable of proving security properties at the hardware/embedded system level. The formalism is still state and transition based, but requires a slightly more complex state machine. In particular, some standard aspects of automata theory requiring finiteness are relaxed.

A standard reference for automata theory is Sipser's Theory of Computation [40]. A more advanced branch of automata theory, including automata on infinite state spaces as used above, is described by Thomas in "Automata on Infinite Objects" [39].

3.1.4 Zero Knowledge/Interactive Proof

Cryptography and general message exchange protocols have been successfully analyzed using different proof and modeling techniques than most single-system mechanisms.

Goldwasser makes extensive use of zero knowledge (interactive) proofs in his "Search for Provably Secure Cryptosystems" [2] . An interactive proof is one by which a challenger can query the holder of information repeatedly in order to convince himself or herself that the information holder does in fact hold the information. It is important to note that zero knowledge proofs (and interactive proofs in general) are not "proofs" in the mathematically rigorous sense of the word. Because they are based on individual correct "guesses", the number of trials can increase the probability of correctness to any arbitrary confidence level. Zero knowledge proofs are useful because they do not leak any information about the secret to the challenger, only that the holder does indeed possess the secret.

3.1.5 Lambda Calculus and Process Calculi

The lambda calculus is a general mathematical construct used to define and evaluate functions. It is symbolic in its computation, binding values to variables as evaluation occurs.

The typed lambda calculus (a lambda calculus extended with a type system) is used in Hicks' Secure Typed Languages [85]. Hick's paper describes language extensions (focusing on a particular implementation for Java) that enable the encoding of security policy information along with function and method definitions. The Coq proof assistant [15], also makes use of an even more refined variant of the typed lambda calculus (the "Calculus of Constructions" [58]). Building on Coq and making heavy use of dependent types (which are data types defined algebraically on top of other types), Chlipala [54] discusses "certified programming", which in this case a "certificate" means that a program meets its specification. In particular, Chlipala states:

> "For instance, the type of an array might include a program expression giving the size of the array, making it possible to verify absence of out-of-bounds accesses statically. Dependent types can go even further than this, effectively capturing any correctness property in a type." [54]

Abadi [7] describes the SPI calculus, which is based on the π-calculus. The π-calculus is a "process calculus". Such calculi are focused on modeling processes (including single threaded "sequential" programs and their multi-threaded counterparts) and their communication channels. The SPI calculus extends the π-calculus with explicit cryptographic "primitives", such as encryption and decryption. This work eventually merged into the "applied π-calculus", and is used by Blanchet's automated cryptographic protocol verifier research (see section 3.1.6).

Furthermore, Datta [9] uses what he describes as the "cord calculus", based on the applied π and SPI calculi, to produce a composition system for security protocols. Due to the extensive focus on composition of existing protocols, he has further modeled elements in the calculus (specifically objects and processes) as categories; the morphisms (or mappings/transformations) are processes from the calculus. This permits him to use results from category theory, which requires only associativity of composed morphisms[1].

3.1.6 Prolog

Prolog[2] is a programming language based on logic. Many mainstream programming languages are structured as a list of steps to take in a prescription for how to solve a problem. Prolog, however, expects to be given a list of facts and rules, which are then queried to determine if new claims are supported by the previously given (or generated) facts.

Blanchet [8] uses Prolog rules to prove a given cryptographic protocol does not leak secrets. The system[3] makes use of an intermediate representation to model the capabilities of the attacker, the knowledge of the attacker, and the protocol itself. Later versions integrate portions of the SPI calculus (the "applied π calculus") as defined by Abadi [7].

3.1.7 Markov Decision Processes

A Markov process is a process with no "memory". The transitions to future states are determined entirely by the current state. A decision process is a Markov system with rewards for each state, and a valuation function that determines the value (present and discounted future) of being in a given state, the goal being to maximize the value by

[1] As opposed to other more restrictive algebraic structures which often require distributivity, annihilator elements or other constructs his formalism does not guarantee

[2] http://en.wikipedia.org/wiki/Prolog

[3] http://prosecco.gforge.inria.fr/personal/bblanche/proverif/

producing an optimal policy. An optimal policy always exists according to Bellman equations [82].

In a Partially Observable Markov Decision Process (POMDP), the exact state of the system is not known, but is a probability distribution over the state space based on the belief state (the path that was statistically likely to have occurred), and the associated value.

Attack planning problems have been modeled as hidden Markov models [78, 79]. In classical planning, the assumption is that the state (the operating system and other software configuration) of the target machine is known to the attacker. In practice, however, the attacker lacks exact knowledge of the system (or network) to be attacked. This uncertainty can be modeled better as a POMDP than using deterministic or 'classical' planning. Explicitly modeling this uncertainty leads to better decision making in the attack planning process (e.g. "when should a scan for more information take place?", or "which system should be targeted next?").

Details about general Markov processes can be found in Puterman's text [82], and dynamic programming solutions to solving for the optimal policy (set of actions to take given the belief state) can be found in Sondik's paper [83].

3.1.8 Formal Verification and Proof assistants

Formal verification of software is important because testing an arbitrarily large number of examples does not prove a claim. Exhaustive verification for even simple operations becomes infeasible due to the size of the data types being verified. For example, a binary operation on IEEE[4] floating point numbers would require verification of approximately 2^{80} values. Proof assistants can be employed to both generate and machine-verify formal, rigorous proofs. The Metamath archive maintains a list of machine-verified proofs[5], with over 10,000 theorems as of the time of writing.

Pierce [15] describes the Coq proof assistant system, a system based on the typed lambda calculus (specifically the calculus of inductive constructions [58]). Coq uses the Curry-Howard isomorphism where terms in the given type system correspond to formal deductions. Essentially this permits a valid transition from type "A" to type "B" (in the form

[4]IEEE standard number 754: http://grouper.ieee.org/groups/754/
[5]http://us.metamath.org/

of a compilable program in Coq) to be treated as a proof that the proposition associated with "A" implies the proposition associated with "B".

Paulson [6] and Malecha [106] discuss the use of "traces" (message exchanges) and the Isabelle theorem prover [6] to verify cryptographic protocols. In Malecha's case, traces are required to prove input-output (IO) properties of the system. The languages for proof assistants (like Coq and Isabelle) are restricted subsets of pure functional languages, which lack the direct capacity for side-effects (all forms of IO are considered side effects). However, since side effects can be essential to the operation (and security) of a program, these traces formally encode which side effects are permissible (or necessary).

Coq and Isabelle represent two of many theorem provers. Different approaches sometimes use different strategies for model checking, term rewriting, and thoroughness of the included type systems and proof databases. Bishop's text "Computer Security: Art and Science" [107] has a detailed and thorough chapter on automatic verifiers and proof assistants. From the early Boyer-Moore theorem prover (NQTHM)[7] (a LISP based system), to ACL2[8] (an updated version of NQTHM capable of running the system as well as proving properties of the model), to the Prototype Verification System[9], and the Symbolic Model Verifier[10], a number of approaches are compared and discussed, each with the goal of system security verification.

3.1.9 Category Theory

Category theory forms the basis of several formal semantics of programming languages [108, 109]. It is also critical for the formalization of Monads and Monoids (see further discussion in section 3.5), which form the basis of all stateful function handling in functional languages like Haskell [110] and Idris [61]. They are used as a mechanism to isolate the unpredictable effects of methods that interact with the world (for example, reading from the keyboard) by isolating the interaction in a "monadic computational context" [108, 111]. Similarly, they form the basis for Separation logic [53, 93]. Separation logic is a tool used to formally reason about effectful computation. This is accomplished by sequential chaining of commands given matching pre- and post-conditions (a technique pioneered by Hoare [94]).

[6] http://www.cl.cam.ac.uk/research/hvg/Isabelle/
[7] http://www.cs.utexas.edu/users/moore/best-ideas/nqthm/
[8] A Computational Logic for Lisp: http://www.cs.utexas.edu/users/moore/acl2/index.html
[9] http://pvs.csl.sri.com/
[10] http://www.cs.cmu.edu/ modelcheck/smv.html

3.1.10 Secure Typed Languages

A related approach to those described in section 3.1.5 and 3.1.8 is to augment existing languages with constructs to support security policy encoding. Hicks [85] describes extensions to operating systems and languages (particularly those running the Java Virtual Machine) to provide security labels (such as classification levels and sensitive information flow tagging). These augmentations permit automatic verification of policies and information flows. However, because these are designed after the original core language they are attached to, they are generally not supported by the entire toolchain.

3.2 Uses in defensive security

Typically, when a formal model of security is given it is to support a security policy. Only recently were models of security put forth from an attackers point of view. These are discussed in section 3.3.

As different security goals require different mechanisms for protection and control, different models have been deployed in different areas of application.

Some modeling came from military (Department of Defense) sponsored research in trusted systems [26], while others adapted previous works to new environments such as networks [34], and database management systems [35]. Modern reinterpretation of these criteria, merged with similar works developed in Europe and Asia, have been bundled as the Common Criteria [51].

Modern society has certain laws that necessitate the appropriate protection of data. In Canada, the storage of financial data, sensitive personal data, and similar "valuable" data is governed by the Personal Information Protection and Electronic Documents Act[11]. Similarly, medical records and all health services data are protected by the Health Information Protection Act (HIPA)[12].

3.2.1 Operating Systems

The operating system forms the core layer of security in software systems (firmware and embedded systems can be viewed as having an embedded microkernel). Some unique

[11] PIPEDA: http://www.priv.gc.ca/leg_c/leg_c_p_e.asp
[12] http://www.e-laws.gov.on.ca/html/statutes/english/elaws_statutes_04p03_e.htm

aspects of the security of such systems are discussed by Lotz [49], which built on and extended years of work on general operating system security.

Security issues became more evident as systems moved from single-user "batch" systems to "time share" systems with multiple users. While Bell's early work [28] dealt with an abstraction of a system, the work was motivated by the Multics operating system.

Requirements for a trusted system, including mandatory and discretionary access controls, privilege levels, and levels of trusted systems (along with guidelines to evaluate and rate systems) were codified by the Department of Defense in the "Orange Book" [26]. This started their "Rainbow Series" of books on trusted systems, networks, distributed systems, and databases.

The guidelines from the Department of Defense, along with European equivalent agencies, were eventually merged to provide the Common Criteria [51] guidelines. These guidelines indicate the required steps to take in order to certify a system, including steps on modeling the system, proofs, and the requirements to meet each assurance level.

Graham and Denning [47] discuss a matrix based model of an access control list (ACL). ACLs are used by many operating system security models to indicate whether or not subjects (users/processes they run) can interact (create/read/update/delete, "CRUD") with system objects (e.g. files).

Harrison et al [45] take Graham and Denning's ACL work further by showing a protection scheme for the ACL matrix itself. They also make the notable step of demonstrating the undecidabilty of "security" in this context. Specifically they demonstrate that an algorithm cannot decide if a sequence of operations can add a right to a previously empty cell in the matrix.

Linden [46] discusses the mathematical and data structures required by the operating system to provide security in practice. This includes things like "small protection domains", "extended type objects" (supporting encoding classification and security policy data), separation of security policy enforcement from system functionality (to limit the security impacts of subsystem failures), and capability based addressing (which refers to restricting access to memory addresses based on user capabilities). Many of these features have become commonplace in modern operating systems, and many are required by the POSIX [48] specification.

Neumann [44] describes a "provably secure operating system", in which he details the specification, design, implementation of the system, and gives proofs of security properties. An important discussion in section 8.2 is the recurring theme that the success of security proofs is determined by the quality of the specifications, and the description of the desired security properties of the system.

The Isabelle proof assistant system, along with a restricted subset of the C language, were used by Klean et al [50] to develop a fully formally-verified microkernel. Dubbed "seL4", the kernel is based on the L4 microkernel[13]. As a microkernel, all extraneous functionality that can reasonably be moved to a user-space program has been removed, and the kernel supports only address space control, threading, interprocess communication, and a scheduler.

3.2.2 Distributed and Parallel Systems

Since many software systems are delivered via the internet, systems are now expected to cope with a large burden of concurrent users. A common solution to this type of scaling/load balancing problem is to scale the system "horizontally". Horizontal scaling is accomplished by running multiple copies of the application in parallel, with a router in front dispatching requests according to various schedules. Similarly, modern processors are growing in number of cores much more rapidly than in raw speed. These distributed and parallel systems carry their own sets of security concerns. Synchronization and related "race conditions"[14] have over 400 issues reported in the common vulnerability and exporsures database (the CVE). At a lower level, multi-threaded programs lose deterministic instruction ordering, and as a result, certain orderings (instruction/thread interleavings) potentially expose data to corruption, leaks, or other subversion.

Benson et al [43] present a formal protection model specifically designed to cope with centralized, parallel, and distributed (CPD) clusters of systems. Based on a state machine formalism, the paper shows that securing the CPD model can be reduced to the problem of securing a sequential, single system. The model makes a distinction between commands (sequences of atomic instructions) that end in a secure state, and those that never leave a secure state. This permits the non-deterministic front-end portion of the system to enumerate and prove the security of all possible instruction interleavings.

[13] http://en.wikipedia.org/wiki/L4_microkernel_family
[14] http://www.cve.mitre.org/cgi-bin/cvekey.cgi?keyword=race+condition

3.2.3 Databases

Database systems are one of the most common mechanisms for storing data in a manner that can be efficiently queried. The data stored by a system is one of the most typical targets for attackers. As such, the security of the database is equally vital to the overall system's security as is the control of the application code.

The Department of Defense's "Purple book" [36] describes the changes to the trusted computer system evaluation criteria (the "Orange book") that are needed to apply the work to a database system. These changes are largely accomplished by the extension of subjects and objects to include database tables, connections and other additional structures required by databases, but not present in a standard operating system.

3.2.4 Cryptography

Cryptography is a vital aspect of security at all levels. Cryptography can provide secrecy of data in transit over an untrusted network (via encryption), tamper-detection (via hashing), and authentication (via signing). It is also used to store data securely while "at rest" on a given host, such that even with complete control of the host machine, the data remains unreadable to an attacker.

Abadi's work on the SPI calculus [7] (discussed in section 3.1.5) has been used to analyze specific cryptographic protocols, as well as to demonstrate their resilience to attacks. This is accomplished by explicitly modeling "channels" over which processes may communicate data. By making the "channels" a first-class notion in the calculus, it is possible (for example) to prove that a given model or protocol only provides properly encrypted data to a channel that may be eavesdropped upon.

The modeling efforts and specification used in the SPI calculus lends itself well to automation, and has been integrated with several works by Blanchet described in [8] : ProVerif[15] and CryptoVerif[16]. ProVerif is a protocol verifier based on the algebraic Dolev-Yao model [3]. The Dolev-Yao model makes the basic assumption that the attacker is able to intercept, replay, and synthesize any message sent by the system. This adversarial model is an appropriate representation of the risks of sending messages across an untrusted network. The ProVerif system is therefore capable of proving secrecy and

[15] http://prosecco.gforge.inria.fr/personal/bblanche/proverif/
[16] http://prosecco.gforge.inria.fr/personal/bblanche/cryptoverif/

authentication (via the SPI calculus support in the system). On the other hand, CryptoVerif is a protocol verifier focused on verifying computational secrecy.

Goldwasser [2] was one of the first to rigorously prove the secrecy of cryptosystems based on computational secrecy. Computational secrecy is the notion that an attacker recovering a key would require excessive time (often longer than the age of the sun). He notes that Shannon's Information Theory [4] shows all systems with "perfect" secrecy are equivalent to a "one-time pad" cipher system (wherein the key is at least as large as the message). Goldwasser also includes comparisons of information-theoretical security/perfect secrecy (as defined by Shannon in [4]) with semantic and computationally secure systems. Since an arbitrarily large key is difficult to produce, manage and remember, most modern cryptosystems rely on computational security. These computationally secure systems generally utilize problems in the NP complexity class. If efficient algorithms were discovered to solve these problems, the security of the systems would be compromised.

In a later paper [10], Goldwasser discusses one time programs, which can be employed to develop self-destructing mechanisms in software comparable to tamper proofing in hardware.

3.2.5 Networks

According to the "Internet World Stats"[17], approximately 34% of the world's population has access to the internet. A substantial number of these users have access to their own subnetworks (whether at work or at home) as well. As a result, it is no longer reasonable to think about systems (or their security) in isolation. Networks have their own issues in security, especially due to the distributed nature of routing and "bucket brigade"[18] packet handling. It is more often the case that data passes through untrusted hosts (or at least hosts not controlled by the sender nor the receiver) than entirely through a trusted network.

The Department of Defense rainbow series has adapted the Trusted Computing System Evaluation Criteria guidelines to the network in the "red book" [34]. In this adapted view, the network is treated as a single trusted system, and the network requires a single reference monitor (in charge of policy enforcement). The notion of subjects is

[17] http://www.internetworldstats.com/stats.htm

[18] Packet traffic is handled by repeated forward towards a destination, similar to a "bucket brigade" moving water.

extended to include processes that act on behalf of individual systems to create and preserve connections. The extended model supports multiple partitioned systems via a multi-level gateway. This configuration begins to reflect the more modern network architecture of De-Militarized-Zones (DMZs) connected by secure routers.

3.2.6 Hardware

With the increasing affordability of devices like Arduino and Raspberry Pi, the past few years have seen a huge jump in the diversity and ubiquitousness of user controlled embedded systems. Even without 'homebrew' devices like those, there are still many special purpose systems like routers, modems, home automation devices, and cell phones in most homes and businesses. These devices have been abused as vectors for attackers to break into networks[19], or have been taken over for their own purposes[20].

Lotz [49] provides a model for proving security within embedded systems and in hardware, without abstractions at the operating system level. The modeling work was done to support Common Criteria [51] QA level E4 requirements. The model is based on a "state transition automaton on infinite structures". This variant of automata is discussed by Thomas in the Handbook of Theoretical Computer Science [39]. Hardware support for physical self-destruction upon tampering is required to achieve certain confidentiality goals (once the CPU enters a compromised state as modeled explicitly in the automaton).

3.3 Uses in offensive security

Historically, Bell's work in the 1970s [28] was motivated by the success of the "tiger teams"; the attackers trying to disprove the security of then-current operating systems (such as Multics). Their attacks were based on knowledge of the systems they were attacking, but lacked specific formalisms and guidelines.

Modern systems and networks are frequently tested for security in similar ways. The activity of the former "tiger teams" is now generally referred to as "penetration testing". These tests look for vulnerabilities in systems that arise from broken software, misconfigured communication protocols, and any other vector that might allow access

[19] http://phrack.org/issues/60/7.html
[20] http://www.businessinsider.com/97-arrested-for-spying-through-webcams-2014-5

to a targeted system. Generally, once access has been obtained, risks (based on what data can be obtained or damaged) are reported to the organization that requested the penetration test.

3.3.1 Attack planning

Penetration testing involves a series of attacks, each successful compromise potentially makes new targets either available or easier to exploit. Historically, these "pivots" and the ordering of systems to attack were chosen based on intuition or ad-hoc valuations of the contents of the systems.

In his PhD thesis, Sarraute [79] discusses some of the challenges surrounding automated attack planning, historically treated as a "classical" planning problem. This older approach lacked the ability to model the information gathering/scanning phase. He argues that a model for penetration testing that acknowledges the unknown portions of the process is more accurate than just ignoring them. Partially observable Markov decision processes (POMDPs) allow the scanning portion of attack planning to be modeled explicitly. His later paper [78], builds on the work by employing an automated solver to compare costs, scaling, and success of his POMDP based approach to maximizing penetration testing value.

3.3.2 Steganography

Data egress (the act of extracting data from a system) can be a challenging aspect of a successful attack. Steganography[21] has allowed seemingly innocuous data to leak. This data is often embedded in media, such as images, video, or music. A novel approach has been to use the very transport mechanism of the internet (TCP/IP) to hide information [112]. The specification of the size and contents of TCP and IP packets and their headers leave certain gaps, which can be abused to store additional data in an obfuscated manner.

3.4 Well known models

There are many models that have been developed within the last 45 years, with some finding more success and popularity than others.

[21] http://www.garykessler.net/library/steganography.html

The following models are heavily cited, as evidenced by CiteSeer[22]. Some of them are the focus of college courses, others have Wikipedia pages in lay language, and all of these are generally known outside of formal security research.

3.4.1 Bell LaPadula

Bell and LaPadula's "Secure Computer Systems" [28] is one of the most cited formal modeling papers. The work was sponsored by the Mitre corporation to research security for the Multics[23] Operating System.

The following description is from section II in Bell's Secure Computer Systems paper [28]. After the original modeling work, the authors revisited their work twice: first to address certain criticisms about the nature of the model [33], and, later, in a retrospective highlighting the success of the model [113]. The model is as follows.

Given sets of Subjects (processes/programs), Objects (data, files, programs, subjects), Classifications (top secret, classified, etc.), and Need-to-Know categories, Bell's formalism is capable of proving that if the system begins in a secure state, and only follows permissible transitions, then the system always remains in a secure state. Any state that is not a compromise is secure. In other words, a compromise exists when:

1. a subject's clearance is lower than the requested object's classification; or
2. a subject does not have some need-to-know category that is assigned to the object.

Bell's results show that if only secure transitions are followed from a secure state, the system never leaves a secure state. This is Bell's 'basic security theorem' stated informally.

3.4.2 Biba

Where Bell and LaPadula's model focused on data confidentiality (that no unauthorized "reads" can take place), Biba [29] focused on data integrity (that no unauthorized "writes" can take place).

[22] http://citeseerx.ist.psu.edu/index
[23] http://en.wikipedia.org/wiki/Multics

Below is a description of one such integrity policy, called the "Low Water Mark Policy", which focuses on direct (overt) and indirect (covert) threats to data integrity as a reversal of the "High Water Mark" policy of data confidentiality. Similar to Bell's model, Biba's model deals primarily with subjects (processes/programs) and objects (data, files, programs, subjects), but rather than classifications and need-to-know levels, it supports "integrity levels". These form a lattice in the same way that classifications form a lattice in Bell's model.

Biba's formalism is based on two key rules:

1. an *observation* of an object can lower the integrity level of the subject observing the object to the level of the object.
2. a *modification* of an object by a subject is permitted only if the object's integrity level is at or below that of the subject.
3. a subject may *invoke* another subject if the latter subject's integrity level is below that of the invoking subject.

Biba's results are summarized as follows. Rule (1) provides assurance that indirect sabotage by 'contaminated data' is impossible. Rule (2) provides assurance that direct malicious modification is impossible. Rule (3) "insures improper activation of more privileged subjects may not cause indirect damage to higher integrity level objects" [29].

3.4.3 Clark Wilson

The Clark/Wilson model [41] is another integrity model similar to Biba's [29], where the focus is on preventing the tampering of data (as opposed to Bell and LaPadula's model [28] focused on confidentiality).

The Clark/Wilson model is based on data items, and transformations implemented on them. All interactions (transactions) on constrained data items must conform to two sets of rules. Operating on a triple of a user, a process, and some data; each integrity verification process (IVP) and transformation process (TP) consumes constrained or unconstrained data items (UDIs) and produces constrained data items (CDIs). The two rule-sets that enforce this are certification based (responsible for making sure the state of constrained data is valid, i.e. "external integrity") or enforcement based (responsible for the operation of the system, i.e. "internal integrity").

The rules for certifications are summarized as follows.

1. all IVPs must ensure a CDI is in a valid state before proceeding.
2. all TPs must be certified (by a security officer) to be valid.
3. the list of relations in enforcement rule 2 must be certified to meet the separation of duty requirements
4. all TPs must be certified to write an append-only log (CDI) with sufficient detail to reconstruct the command
5. any TP that takes a UDI may only perform valid transformations or none at all.

The rules for enforcement are summarized as follows.

1. the system must maintain the list of relations in certification rule 2, and ensure any manipulation of a CDI is via a TP specified in said list.
2. the system must maintain a list of relations limiting the CDIs which a TP may manipulate given a current user.
3. the system must authenticate each user attempting to invoke a TP
4. "only the agent permitted to certify entities may change the list of such entities associated with other entities. An agent that can certify an entity may not have execution rights with respect to that entity." [41]

It is interesting to note that this model is described with less mathematical rigor than the Biba model. There are no proofs of the assertions that a system constrained in this way maintains the integrity of the system. The burden of proof is then shifted to the "certifying officer(s)" in charge of maintaining the list of valid TPs.

3.4.4 Brewer Nash

In 1989, Brewer and Nash released their "Chinese wall security policy" [42] paper. Motivated by new laws imposed by banking regulations in the United Kingdom, the authors adapted a model similar to Bell and LaPadula's to support additional restrictions surrounding "conflict of interest" classes.

The goal of the policy was to prevent analysts (or other agents) working for banks (or law firms, or any other agency with access to corporate data) from leaking critical data between competing entities (whether deliberately or inadvertently). For example, it would be permissible for the same agent to view the data from a company in the software industry as well as data from a company in the mining industry. This is under the assumption that the companies are not in direct competition, and the agent having the information will not benefit either company unduly. On the other hand, the model would forbid the same agent from subsequently accessing the data of a third company if it operated in the same industry as either of the first two (software or mining). Their policy for conflict classes "cannot be correctly represented by a Bell-LaPadula model" [42]. To resolve this, the Bell LaPadula notion of a subject is extended to represent a user and any program that might act on his behalf (this is similar to the "seteuid" family of functions in POSIX [48]).

The model is described below.

- Objects: individual items of information, each concerning a single company.
- Company Dataset: group of objects which belong to the same corporation.
- Conflict of Interest Class: a group of company datasets whose corporations are in competition.

The model extends the concepts defined by Bell and LaPadula [28], with a new rule; access is only granted if the object requested meets the following criteria:

- it is in the same company dataset as an object already accessed by that subject; or
- it belongs to a different conflict of interest class.

This rule has implications for how many people are required to operate the policy (as a function of the number of entities within a conflict of interest class). The model dictates that such accesses be recorded in an access matrix, which is queried to determine the state of the conflict of interest classes in the system.

Structure	Closure?	Associative?	Identity?	Inverse?
Magma	Yes	No	No	No
Semigroup	Yes	Yes	No	No
Monoid	Yes	Yes	Yes	No
Group	Yes	Yes	Yes	Yes
Partial Monoid	No	Yes	Yes	No

TABLE 3.1: Summary of Mathematical Structures

3.4.5 Harrison Ruzzo Ullman

Harrison et al [45] define a model (HRU) for protection (meaning controlling the integrity of the access control matrix). Their work is similar to Graham/Denning's model [47], which deals primarily with how to add/remove subjects/objects and update the access control matrix accordingly. The HRU model is also based on a matrix (subject $\times$ object) containing the current access rights. Compound operations of simple operations are bundled as a transaction; the whole being successful only if all the components are successful.

The authors note that the general property of security ("can a finite sequence of commands add a right to previously empty cell?") is undecidable.

The notion of matrix-based state control for access is embedded within other models, including the Chinese Wall Security Policy [42] described in section 3.4.4.

3.5 The Mathematics of Strings

Generally the work in this section is based on three sources, Berstel and Perrin [114], Shallit [115], and Bourbaki [116]. Berstel defines a word to be a sequence of characters from a finite alphabet, and we adopt this definition. Many interesting properties of words are given by Shallit, and we provide predicates and decision procedures for a selection of them. The characterization of mathematical structures is based on the taxonomy and hierarchy given by Bourbaki.

The key mathematical structure exhibited by Strings is that of a monoid. We have the following hierarchy of increasing structure: magma, semigroup, monoid, group. The differences are summarized in table 3.1 (operating on a set S with binary operation $* : S \times S \to S$).

Recall the following definitions, for a set S and a (possibly partial) binary operation $* : S \times S \to S$:

- **Closure/Totality**: S is closed under $*$ if $\forall a, b \in S, a * b \in S$;
- **Associative**: $*$ is associative if $\forall a, b, c \in S, a * (b * c) = (a * b) * c$;
- **Identity**: $*$ has an identity $e \in S$ if $\forall a \in S a * e = a = e * a$;
- **Inverse**: $*$ has inverses if $\forall a \in S, \exists a^{-1} \in S, a * a^{-1} = e = a^{-1} * a$, where e is the identity.

In order to make arguments rigorous, we define strings and concatenation formally as follows.

Definition 1. A **string** is a *finite* sequence of symbols from a finite alphabet Σ. We denote the set of all strings with S, and we call it a string monoid. The *empty string* is the unique string of length 0, denoted as either "" or ().

Remark. A string s can therefore be written as $s = a_1 a_2 ... a_n$ where each $a_i \in \Sigma$ for $i = 1...n$, and the length of s (denoted $|s|$) is n.

Definition 2. **Concatenation** (hereafter denoted $*$ for strings, $++$ for lists and vectors) is defined on $x = a_1...a_n$ and $y = b_1...b_m$ as $x * y = a_1...a_n b_1...b_m$.

We present proof outlines that strings (often called **words** in mathematical literature such as [114]) exhibit the structure of a monoid. Recall that monoids require closure, associativity, and an identity element. The proofs that strings have these properties are as follows:

- **Closure**: Let $x = a_1...a_n, y = b_1...b_m \in S$, then $n = |x|$ and $m = |y|$ are both finite. Since $x * y = a_1...a_n b_1...b_m$ by definition, then $|x * y| = m + n$ is finite. Therefore, $x * y \in S$.
- **Associative**: Let $x = a_1...a_n, y = b_1...b_m, z = c_1...c_p \in S$. Then $(x * y) * z = (a_1...a_n b_1...b_m)c_1...c_p = a_1...a_n b_1...b_m c_1...c_p = a_1...a_n(b_1...b_m c_1...c_p) = x * (y * z)$.
- **Identity**: Let $x = a_1...a_n \in S$, and let $e = ()$, the empty string, where $|e| = 0$. Then $x * e = a_1...a_n() = a_1...a_n = x$, and $e * x = ()a_1...a_n = x$.

Therefore we can see that strings, under the operation of concatenation defined as above, form a monoid (and thus also a semigroup and a magma).

Similarly, we can show strings satisfy Functor and Monad laws. We state these properties briefly as follows (full definitions in source code available online[24]):

- **Functor**: for two string monoids A, B, a functor $F : A \to B$ acts as a homomorphism between strings, with the following laws under the operations $* : A \times A \to A$ and $+ : B \times B \to B$:

  identity: if $a \in A$ and $b \in B$ are the identities, then $F(a) = b$

  concatenation: $F(a * b) = F(a) + F(b)$.

- **Monad** supports the bind and return operations, and requires associativity.

- **Foldable** a type that may be accumulated into a single value from a collection of values (in this case a string may be folded starting from an empty string and accumulating a collection of characters).

The structures of strings with (see section 3.5.2) and without negatives (see section 3.5.1) are summarized in table 3.2.

Structure	**Without Negatives**	**With Negatives**
Magma	Yes	Yes
Semigroup	Yes	Yes
Monoid	Yes	Yes
Group	No	Yes
Partial Monoid	Yes	Yes

TABLE 3.2: Summary of mathematical properties of strings.

As shown above, the most structure exists when negatives are permitted, and the least structure is available when they are not. These options are discussed in the following sections.

3.5.1 Standard

As shown in the previous section, basic string types have the structure of a monoid. This structure is available regardless of the representation (see section 8.1.5). Note that there is no notion of inverses or "negative" characters in the standard formalization, and as such, we cannot give the structure of a group.

[24] https://github.com/bgoodspeed/idris-strings

3.5.2 Augmented with Negatives

We find a fundamental split between basic strings (words) and strings that also have negatives. The so-called free-group can always be created given a semigroup. There is a standard mechanism for constructing the "free group" associated with any semigroup (or monoid). The intuition is to add a "negated" version of the alphabet, and then the words are formed from the union of the two alphabets. Furthermore, the definition of concatenation is augmented with the additional condition that "a" concatenated with the string containing the inverse of "a" yields the empty string, "". Full details of the construction is available in Berstel and Perrin [114].

The full implementation in Idris is given online[25], with key aspects discussed in section 8.1.5.1.

Remark. An interesting side effect of this is that it is possible to model edits from $w_1 \in S$ to $w_2 \in S$ by concatenation. The proof sketch is based on the following equation: $w_1 * (-w_1 * w_2) = w_2$. Clearly, this represents an inefficient upper bound (in fact equal to $|w_1| + |w_2|$ because $|w^{-1}| = |w|$) on a more traditional edit distance, such as the Levenshtein distance [117].

Another implication is that concatenation becomes a more complex operation to implement (as it requires a stack to track potentially cancellable elements) than it would be without the possibility of cancellation. This increased complexity comes from the need to continually check if pairs of characters cancel each time the string is "collapsed". The source code is given in figure 3.1.

This more complicated form of concatenation immediately rules out a vector based representation. In brief, the reason for this is as follows: let $v_1 \in C_n$ and $v_2 \in C_m$ where C_n and C_m denote the vector space of n and m copies of the alphabet C, respectively. Then the concatenation, $v_1 * v_2 = v_k \in C_k$ where $0 \leq k \leq n + m$, violates the required closure property. This is because two vectors spaces, V_n and V_m are considered distinct unless $n = m$. At best, we would be able to show such a construction has the structure of a category: non-closure, associative, identity, and non-inverse. See section 3.5 for details of these structures and for a discussion of some implementation issues.

While interesting, it is not representative of strings as they exist "in the wild", so this is useful more for theoretical reasoning than practical use.

[25] https://github.com/bgoodspeed/idris-strings

```
%assert_total
wordCollapseOneLevel : Word -> Word
wordCollapseOneLevel Empty = Empty
wordCollapseOneLevel (x # Empty) = x # Empty
wordCollapseOneLevel (x # (y # z)) = case (x `isInverseOf` y) of
                                True  => wordCollapseOneLevel z
                                False => x # wordCollapseOneLevel (y # z)

%assert_total
wordCollapse : Word -> Word
wordCollapse x = let y = wordCollapseOneLevel x in
                     case (x == y) of
                          True  => x
                          False => wordCollapse y


wordConcatAndCollapse : Word -> Word -> Word
wordConcatAndCollapse w1 w2 = let w = wordConcat w1 w2 in
                                      wordCollapse w
```

FIGURE 3.1: Concatenate and collapse for the free group on strings.

3.5.3 Categories

As categories require less mathematical structure than semigroups and monoids, any representation that is a semigroup or a monoid is automatically a category. However, as certain representations cannot be officially considered semigroups (because of a lack of closure/totality on the binary operation), we must fall back on these simpler structures.

For lists and cons cell representations, we have a monoidal structure, and thus the monoidal category is sufficient. This is described in section 3.5.3.1.

Recall a **category** is a collection of *objects* and of *arrows*. The operation *composition* is imposed on any pair of arrows, $f : x \to y$, $g : y \to z$ such that $g \circ f : x \to z$. All composed arrows must *associate* (i.e. $h \circ (g \circ f) = (h \circ g) \circ f$). Furthermore, each arrow must have a left and right identity as follows: $1_y \circ f = f = f \circ 1_x$.

3.5.3.1 Monoidal Category

There is a category associated with any monoid. In this case, we define it as follows: there is only one *object*; the set of Strings (or words, commonly written as the the Kleene

closure of the alphabet, Σ^*). The *arrows* are defined one per word, and are denoted $\pi_w : \Sigma^* \to \Sigma^*$. Arrow composition is defined as $\pi_v \circ \pi_w = \pi_{wv}$

The required laws, associativity and identity are proven as follows:

- **associativity**: Let $\pi_x, \pi_y, \pi_z : \Sigma^* \to \Sigma^*$ represent the arrows for the words x, y, z, respectively. We denote concatenation of words x and y as xy. Then $\pi_x \circ (\pi_y \circ \pi_z) = \pi_x \circ (zy) = (zy)x = z(yx) = z(\pi_x \circ \pi_y) = (\pi_x \circ \pi_y) \circ \pi_z$.
- **identity**: In this case the left identity and right identity are the same, and it is called $e : \Sigma^* \to \Sigma^*$, which corresponds to the empty word, "". Therefore $e \circ \pi_x = x"" = x = ""x = \pi_x \circ e$.

Thus, due to the associativity of concatenation, and the existence of the empty string, we have a category.

3.6 Fundamental Limitations

It should be clarified at this point that the issues described in this thesis do not deal with fundamental/physical aspects of security. Side channel analyses [118], such as power fluctuation analysis [119] (where the actual power consumption during computation leaks information about the signal being processed), is deliberately excluded from the considerations here.

Chapter 4

Observations and Hypothesis

From examining all of the efforts surrounding formal methods in computer security detailed in sections 2 and 3 a few patterns emerge. One pattern is that each system or paper defines what "secure" means for their own context. Another pattern is that the common workflow for verified software development (described in section 1) has several phases, and each transition between phases presents an opportunity for defects to enter the system. A further source of problems is that industry produces most software in use, and those in industry are mostly interested in being able to produce software quickly and efficiently (often in terms of cost). However, most security theorems are produced by academics and researchers and are produced for other academics and researchers. Due to this disconnect between the two communities, the barrier to entry is very high for new developers to adopt formal verification in their own work.

4.1 Definitions of Security

For some systems, "secure" means that a system permits no unauthorized reading of information at a higher security clearance than that of the user attempting to access the information [26, 28, 51]. Others define security in terms of resilience to certain classes of attack [3], or by statistical properties of messages created by the system [12]. Regardless of how security is defined, to be formally verified, the definition needs to be rigorously encoded in a way that is accessible to the model.

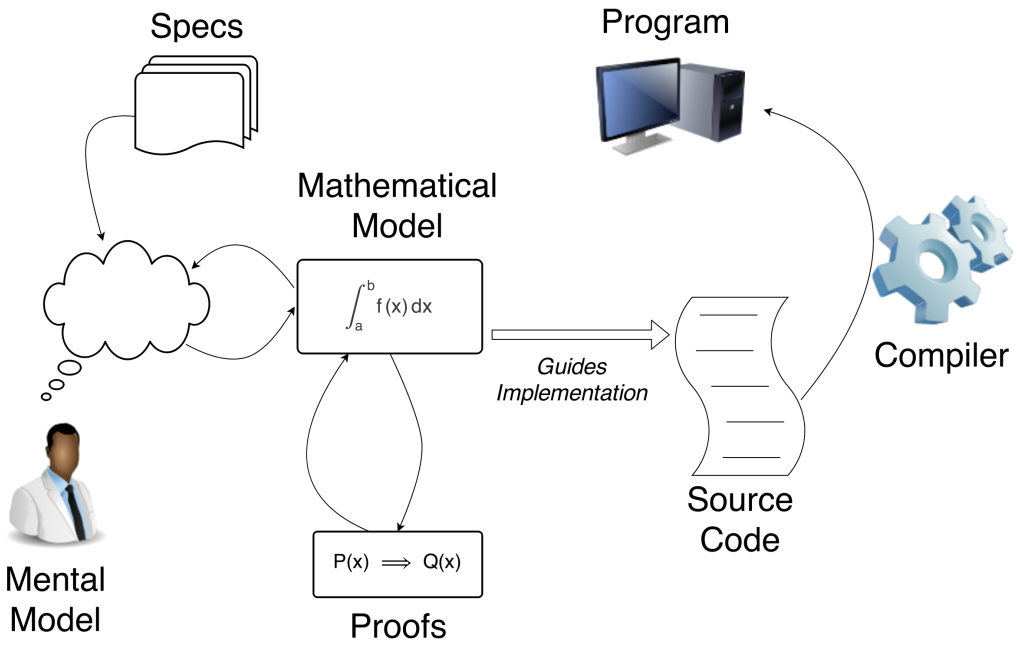

FIGURE 4.1: Existing common workflow

4.2 Gaps

As it has been said, there is a gap between theory and practice (or "in theory, theory is practice; in practice it's not."[1]). In particular, a problematic gap exists where a model (about which proofs exist) is used only as a guide to the implementation. A costly example of this was the recent Heartbleed bug [80] in an extension to the open source cryptography library OpenSSL. While the cryptographic base framework was proven to be sound (at least assuming the utilized problems are not in P), the implementation was not sound. To produce secure systems more reliably, we must reduce the number and size of such gaps.

4.3 Industry vs Academia

There are barriers to entry to the use of formal methods. These barriers prevent large-scale adoption, which ironically slows the progress of making adoption easier. This is due to the lack of mathematical training that is required for formal proofs/modeling.

[1] Variations of this quote have been attributed to DaVinci, Fermat, Pascal and Einstein.

Proof assistant systems are far more "rigorous" in their treatment of programming constructs than popular industry programming languages. This formality and rigor are not familiar to industry programmers. This makes creating even unverified software with these systems more challenging than users might expect.

These barriers imply that while a formally verified program might be safer, it is not likely to be cheaper to produce/maintain. Regrettably, this makes industry less likely to select these more rigorous options in all but the most security-critical applications.

4.4 Hypothesis

We derive our hypothesis, objectives and subsequent predictions and expected outcomes via the following line of reasoning:

- If we appropriately select a starting research language (section 4.5.1), and
- if we guide the software development by proof statements (section 4.5.2), and
- if we produce the correct building blocks for provable software (section 4.5.3), and
- if we produce the correct lemmas and proof tools (section 4.5.4)
- then our software will have provably verifiable features (section 4.5.5), and
- our subsequent proof-driven software will be able to reuse both the proven components and the proof tools (section 4.5.6).

This leads to the following thesis statement and overall hypothesis:

> *Dependent type systems enable a paradigm of programming, "Proof-driven development", which can reduce the efforts required to construct software with provable properties, when there are available libraries of proven building blocks.*

4.5 Methods

The methods are in two parts. The first part is to determine an appropriate starting point. Specifically, we will evaluate systems and determine which option satisfies the conditions specified by the objectives described in section 4.5.1. The second part is to

produce software via proof-driven development, and to measure our savings in time and lines of code as we create new re-usable components.

We must identify a collection of languages or systems suitable for this research line. We must decide whether they can extract an executable program, then identify if the executable program is sufficiently performant to be useful.

We will find a suitable collection of programs to develop (for example a login program and secure string handling functions), and a collection of relevant properties to prove. There are numerous sources for such programs, including standard benchmark programs, common security programs and introductory/tutorial programs used in language instruction.

After that, we will identify properties that can be verified only in a proof environment. Specifically, the properties that are beyond the scope of test-driven development or randomized statistical testing.

As we develop the programs, we will identify commonalities (data types, proofs and useful functions). We will determine a way to package these components for re-use. Finally, we will use the shared components in our benchmark programs and other example programs. We will then measure the difference in lines of code required to solve each problem/complete each program, counting the number of lines of proof and code saved.

4.5.1 Objective: Selecting a Language

We must choose from among a collection of proof assistants (these will be identified and discussed in chapter 5). We will do this based on the language's expressive power as a programming language, the language's power as a theorem prover, the ability to extract executable machine code, and the "acceptable" performance of the language.

There is a spectrum of languages ranging from pure theorem provers to general purpose languages with a powerful enough type system to accomplish proofs. The further toward general purpose languages we get on the spectrum, the more expressive as a programming language the system tends to be. Similarly, the ability to extract a runnable program is essential for our purposes. Not all languages listed can accomplish this; fewer still can extract efficient enough code to have acceptable performance. We still need to be able to exceed the power of fix-point assertions that test-driven development provides, and as such we require certain features that only exist in theorem proving languages.

Finally, the language must be able to produce programs that run efficiently enough to be useful in practice. This means that they efficiently represent data types without wasting unacceptable amounts of memory, or taking an unreasonably long time to complete calculations.

4.5.2 Objective: Proof driven development (PDD)

As an extension of test driven development, we gain the ability to make assertions over mathematically quantified types. That is to say, rather than just asserting that the value of a function for a given input is a fixed value, we can now make assertions about the behaviour with any appropriately typed inputs.

We must adapt test-driven development practices to the broader scope of proof-driven practices. This will allow us to treat the processes we follow formally/rigorously, and make them repeatable. This process permits us to close gaps in the common workflow in figure 4.1. The new workflow with gaps closed is shown in figure 4.2.

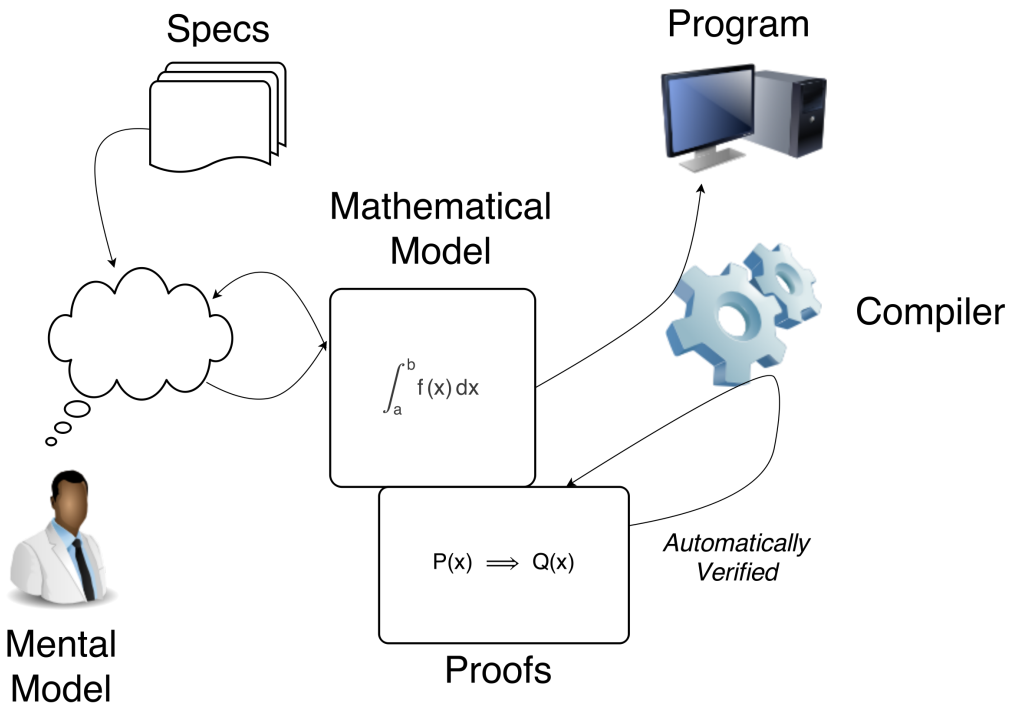

FIGURE 4.2: Improved workflow for PDD.

4.5.3 Objective: Provable Software Building Blocks

We must find appropriate building blocks for provable software. This would include utility methods and data types. The notion of modularizing and packaging code for re-use goes back to the earliest languages. In this case, we are dealing with the dependently typed paradigm, which is an extension of the functional paradigm. These building blocks must be made available to the public in an appropriate way.

4.5.4 Objective: Proof tactics and Lemmas

As with the packaging of dependently typed code, we must find out how to package and re-use the proof tools and proof results we discover. Unlike prior paradigms, where test suites and testing components are rarely considered a directly relevant or mandatory component of the software being packaged, we must consider them as being much more highly coupled.

In a traditional environment we can merely ship the software components. Only developers who desire to do automated testing would note the lack of testing components. Specifically, automated testing calls for convenience methods/objects to help users create their own new tests, and frequently requires the test suite that verifies the library to be included in the new system. With proof driven development (and to a lesser extent dependently typed systems as a whole) we do not have the luxury of not testing the code that uses a verifiable component. The proof portion is central to the data type and any functions that use it. Even calling the function or creating the data type requires the user to supply a proof. It would be an unreasonable burden to leave that entirely to clients of the library. Therefore, we must be able to package proof tactics, useful lemmas and related mathematical tools along with our software libraries.

4.5.5 Prediction: Verifiable Features

We should be able to automatically verify a set of properties every time we alter the software. The set of verifiable features will be strictly larger than those we could verify with fixed value tests.

We must demonstrate that any property that can be verified in a traditional fixed value quantification can also be verified in a proof assistant. We must further demonstrate

there exists properties that can be verified only with a theorem prover and not by fixed value quantification assertions.

4.5.6 Prediction: Subsequent Development

We should be able to successfully re-use both the software and the proof artifacts in subsequent development effort. This re-use will save us time and reduce costs and mental effort. We will be able to show where we used identical code from previous programs, and likewise identical proof components.

4.6 Summary

In this section we have described the commonalities and the gaps we have found in the processes used in prior research. The objectives and predictions described here are elaborated in further sections of this work. The analysis of the language selection and details of the chosen system are presented in section 5. Further details about the new workflow (PDD) are presented in section 6. Our contributions of new datatypes, utility functions, proof tactics and lemmas are presented in sections 8 and 9. The results of cost savings and the evaluation of the predictions are given in section 10.

Chapter 5

Selecting a Language

We have decided to focus on a single language upon which to base further efforts. The selected language is among the hundreds[1] of languages that are in use today. It is not feasible to evaluate every language, so we must consider only the languages that have the possibility of success, given the goals defined in chapter 4.

Computer aided proof, and systems that support this idea are both recent and predominantly academically developed and used. Because of this, even our most popular language option, Haskell [110], accounts for less than 0.2% according to the TIOBE[2] rankings.

Some of the reasons for the minimal use are the relative age of these languages, their complexity (and hence the lack of understanding), and the relative paucity of libraries and frameworks. Furthermore, software developers would not get exposure to these languages in typical undergraduate and trade school curricula.

As proof-related features are somewhat esoteric in the realm of programming languages, we can begin our search with tools/languages that self-identify as proof assistants[3]. The selection is based on the following factors:

- the language's expressive power as a programming language, (section 5.1.1);
- the language's power as a theorem prover (section 5.1.2);
- the ability to extract executable machine code (section 5.1.3), and;

[1] http://en.wikipedia.org/wiki/List_of_programming_languages
[2] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html
[3] http://en.wikipedia.org/wiki/Proof_assistant

- the “acceptable” performance of the language (section 5.1.4).

5.1 Details

In this section we clarify the requirements stated previously. We give metrics for measuring “expressive power” or “proving” capability. Furthermore, we consider various aspects of performance from the perspectives of a developer and an end-user.

5.1.1 Expressive Power/Toolchain

For our purposes, we consider expressive power to be the ease with which a programmer can use the language to solve a problem (often measured by the lines of code required). A part of this power derives not from the language itself, but from the suite of related programs required to construct software with the language – the “toolchain”. The software development lifecycle (SDLC) often concerns itself with the processes surrounding the design of software systems, and the order in which verification and development are done. This is discussed in more detail in chapter 6. Here, we are concerned only with the aspects of the SDLC related to compilation, development, and profiling (performance measurement).

Development in the previous list refers to the interactive workflow of building the software in the system itself. This category includes issues such as the presence and quality of a read-eval-print-loop, the quality and clarity of error messages, and other system specific features (for example Agda and Idris both feature support for “holes”, which can be a convenient way to build programs iteratively).

A related area we must consider is the availability (and ability to construct) quality libraries for common functions and data types. This concept is extended in proof assistants, which unlike previous programming systems, can be helped by having libraries just for proof tactics. These tactic libraries are discussed in chapter 9. The closest mainstream programming construct to a proof library would be an automated testing library such as xUnit frameworks (one example is jUnit for Java [68]).

As noted in depth in section 8.1, string handling is an important part of practical programming. Further, web development tasks are very common, so if there are frameworks to facilitate that, the language is much more likely to be used. The toolchain also includes tools for fetching and packaging these user-libraries, this is a common concept in

mainstream programming languages. In Java, this can be done using 'jars', fetched by the tool 'maven'. In Ruby, they are called 'gems', in Python the tool is called 'pip'.

Aside from the mechanical parts of the toolchain, like the compiler/debugger/profiler and the library format, modern programming systems are often paired with Integrated Development Environments (IDEs). The IDE support typically provides, at a minimum, syntax highlighting and the ability to invoke the toolchain to build the system. More advanced IDEs support graphical breakpoints and expression evaluation during debugging, automated refactoring support, and built-in browser viewing (for web application development).

5.1.2 Utility as a Theorem Prover

To consider a language a theorem prover for our discussions, we need at least universal quantification over a predicate (e.g. $\forall x, P(x) = true$) rather than finite valued quantification (e.g. $\forall x \in \{v_1, \ldots, v_n\}, P(x) = true$). Some of the implications of this broader ability are discussed in chapter 6.

General software systems (in particular, their functions/methods) may be specified only for their behavior given fixed inputs/state. That is to say, we can determine the output value of a function for a given fixed input. So-called "purely functional" languages force users to explicitly model stateful functions. In this case we mean functions f where two subsequent calls with the same input, $f(x)$ need not produce the same output. Purely functional languages provide some ability to reason about pure functions, because any application of the function can be replaced with the once-computed value. This fits nicely with symbolic rewrites and the lambda calculus.

In addition to pure functionality and symbolic rewriting, we need to be confident of two other important properties: we need to be able to determine that all our functions cover all execution paths (complete pattern matching), and we need to be able to determine that our functions terminate. Theorem provers capable of asserting "total correctness" require a termination guarantee to admit proofs [120]. If termination were not required, the system may admit the Curry paradox[4] and thus use the non-terminating proposition to prove any other proposition in the sytem.

All "Turing-complete" programming languages have the same computational power. In this thesis, *expressive power* will refer to the more qualitative aspects of the language.

[4] http://plato.stanford.edu/entries/curry-paradox/

To clarify, a more expressive language might require fewer lines of code to implement the same algorithm, require fewer instances of repetitive"boilerplate" code, or be able to do with the standard/core language what another language can accomplish only extensive third party libraries.

Evaluating systems specifically built to accomplish these goals is a much smaller problem than evaluating the full list of languages. We only need to evaluate proof assistant systems that support the criteria listed above, namely: universal quantification, pure functions with termination and branch coverage properties.

Typically, the theorem provers give up portions of logic (namely the law of the excluded middle/proof by contradiction [58]). This means an intuitionistic or constructivist form of logic must be employed [58, 63]. The most common/popular formalizations are the calculus of constructions (specifically the calculus of inductive constructions, CIC [58]), and Martin-Löf type theory [63]. The CIC formalisms have been extended by the dependently typed formalisms of Martin-Löf, due to the useful algebraic properties of the types supported by such systems [54, 89], and form the basis of several systems like Idris, Agda and Cayenne.

5.1.3 Runnable Code

Fundamentally, a system useful only for verification of mathematical models is not useful as a practical programming language. It does not close the gap between specification/proof and executable code (see figure 4.1). In particular we need to be able to deal with the "messy" aspects of programming such as input/output, mutable data structures and other stateful operations. As a result, if the toolchain provided cannot produce a binary executable (or bytecode that can be interpreted by a virtual machine) or interpreted by an interpreter capable of providing the necessary operations, then the system is not suitable for systems programming.

Even in systems that support the extraction of runnable code, the extraction process is not always built-in. For example, Coq has the ability to extract ML or Haskell code which can then be made executable by the toolchains provided by Haskell and ML. However, the extracted code sometimes loses some efficiency. An example of this problem is shown in the ML code extracted from Coq sources in appendix 6. Further issues related to extraction are described by Swierstra in his experience report [121] on working with ML code extracted from Coq. This is exacerbated if the extracted code needs to be modified.

As some tools do not support either compilation or extraction, these tools are not suitable for end-to-end software construction.

5.1.4 Performance/Usability

Assuming we can extract code or compile the software directly, we must be left with a workable artifact. If the produced artifact is too slow to solve a real-world-scale problem in a reasonable amount of time, then it is not usable. For instance, we need it to run efficiently, that is it should solve real world problems at least as fast as some commonly used language, Haskell for example.

Note that this does not have to be as fast as the fastest languages (e.g. C or assembler). Ruby for many applications runs 2-100x slower for certain algorithms, yet still sees use in the real world.

The toolchain (in C this would be the compiler/linker/debugger/profilers) must not involve manual editing of any artifact in the chain. In terms of usability we need to be able to go from an error (during development, nothing is perfect) in the executable artifact back to the line/function/file in the source code. If we must manually edit any artifact in the build chain, we can no longer be confident in the properties we have proven about an artifact earlier in the production toolchain. This workflow is depicted in figure 4.1. This is similar to the issues posed in chapter 4 where the proofs were about the paper model. In this case, even if the proofs are automated and continually verifiable, they refer to an artifact that is not the one being executed.

5.2 Coq

Formally, Coq implements the Gallina language, based on the Calculus of Inductive Constructions [58]. Unlike standard Hoare triples, all terms in Gallina are strongly normalizing[5], and as such require a proof of termination along with their definition before a definition is admitted. Gallina is a purely functional language, which is to say that side effects (such as changes of state, reading input, writing output, etc) are not permitted. As a result, standard imperative programming paradigms are not available in Coq programs directly. These things need to be 'bolted on'.

[5] that is, each expression can be reduced to an irreducible term and the procedure to reduce them terminates

Coq supports dependent types [54], which are essentially functions whose targets depend on the values of their arguments. This form of type system admits the Curry-Howard isomorphism [109], where logical proofs correspond to terms in the programming language and vice versa. This idea is the core of Coq, and the reason that a program in Coq denotes a proof of the proposition. These dependent types have also been used to provide some of the non-functional/imperative aspects required by most programs. A large collection of such things is provided by the Ynot library [53]. Ynot implements "separation logic" (an extension of Hoare logic) which provides the ability to reason about state via manipulation of heaps. As Chlipala says in his paper describing Ynot, "Coq excludes general recursion, mutable state, exceptions, I/O and concurrency", all of which can be modeled by separation logic and the use of monads (similar to the approach taken by the Haskell language to permit input and output).

The Coq style of interactive proof is not the only approach to formal verification or machine-assisted proof systems. "A competing alternative to the common style of Coq tactics is the declarative style, most frequently associated today with the Isar language. A declarative proof script is very explicit about subgoal structure and introduction of local names, aiming for human readability." [15]

5.2.1 Tactics

"The word 'proof' is rather overloaded and can be used in several different ways. [In Coq], we use 'proof' for a script to be presented to a machine for checking." [57] Furthermore, Coq typically uses an interactive prompt to work with the system towards a proof. This method of proof is described by Bertot as follows:

> "The usual approach to construct proofs is known as goal directed proof, with the following type of scenario:
>
> 1. the user enters a statement that he wants to prove, using the command Theorem or Lemma, at the same time giving a name for later reference,
> 2. the Coq system displays the formula as a formula to be proved, possibly giving a context of local facts that can be used for this proof (the context is displayed above a horizontal line written =====, the goal is displayed under the horizontal line),
> 3. the user enters a command to decompose the goal into simpler ones,
> 4. the Coq system displays a list of formulas that still need to be proved,

5. back to step 3." [52]

The above can be thought of as a manual proof interaction, sometimes known as the "video game" approach. However, Coq also has a tactic library to facilitate automated proofs. The language embedded in Coq to define and extend the proof capabilities is called "Ltac" [57]. The advantages to extending the tactics and techniques available to the automated proof system are that they are more adaptable to changes in the definitions and functions being reasoned about. If the manual version of the proof script is too tightly coupled to the definitions/functions, then any change to the source will perturb the validity of the proof.

5.2.2 Sample Code and Results

A sample, "hello world" program for Coq (using the Ynot [53] library) is shown in figure 5.1.

```
Require Import Ynot.
Require Import Basis.
Require Import String.

Open Local Scope string_scope.
Open Local Scope stsepi_scope.

Definition main : STsep (__) (fun _:unit ⇒ hprop_empty).
  refine (printStringLn "Hello World");
  sep fail auto.
Qed.
```

FIGURE 5.1: Coq "hello world".

Due to the extracted code, the performance issues make the system very problematic at runtime. Representation inefficiencies can be seen in the use of Lists of Ascii sequences in place of character strings in figure 5.2.

To summarize the results of the Coq analysis, we find the results shown in table 5.1.

System	Expressive?	Provable?	Extractable?	Performance?
Coq	Yes	Yes	Yes	No

TABLE 5.1: Coq Results

```
let rec matching_in_field idx token = function
| [] -> []
| x::y ->
  if if prefix token
          (nth idx x ((MlCoq.Ascii (false, true, true, true, false, false,
            true, false))::((MlCoq.Ascii (true, true, true, true, false,
            false, true, false))::((MlCoq.Ascii (false, false, false, false,
            true, false, true, false))::((MlCoq.Ascii (true, false, true,
            false, false, false, true, false))::[])))))
     then prefix
            (nth idx x ((MlCoq.Ascii (false, true, true, true, false, false,
              true, false))::((MlCoq.Ascii (true, false, true, false, false,
              false, true, false))::((MlCoq.Ascii (false, true, true, false,
              true, false, true, false))::((MlCoq.Ascii (true, false, true,
              false, false, false, true, false))::((MlCoq.Ascii (false, true,
              false, false, true, false, true, false))::[]))))))) token
     else false
  then x
  else matching_in_field idx token y
```

FIGURE 5.2: ML code extracted - fragment.

5.3 Agda

Agda [60] is a dependently typed language, based on Martin-Löf type theory [63]. Agda was developed as a proof assistant based on the concept of "holes" that are to be iteratively refined in a new style of development. Rather than creating a complete function from start to finish before completing a program, a missing step (computation) can be left in the form of a hole, denoted *?holeName*. The type signatures (e.g. a function consuming an Integer and returning a Boolean) of the holes can be interactively queried. This workflow is novel and generally unfamiliar to developers using mainstream programming languages.

Agda is focused on mathematics, with most libraries focused on arithmetic and various forms of higher order logic [60].

While it does support direct compilation (unlike Coq), the artifacts produced are not memory efficient. In particular the snippet in figure 5.3 (described in a posting on Stack Overflow [6]), yields a resulting artifact that is nearly 20MB in size[7]. This is substantially larger than the typical binary sizes for typical languages like C (8.3Kb) or even a proof assistant like Idris (89Kb). This is due to a similar data type bloating as seen in Coq extractions to ML (see appendix 6). Agda programs can be compiled using the code shown in figure 5.4.

[6] http://stackoverflow.com/questions/9472488/differences-between-agda-and-idris
[7] Compiled on a macbook air running OSX

```
main : IO Unit main = putStrLn (toCostring “Hello, Agda!”)
```

FIGURE 5.3: Agda “hello world”.

```
save it to “./hello.agda”
download lib-0.6.tar.gz, and unpack it to somewhere, say DIR
cd DIR/ffi && cabal install
agda -i DIR/src -i . -c hello.agda
```

FIGURE 5.4: Agda compilation.

Agda has some advantages over traditional proof assistant languages like Isabelle and Coq, in that is has a programming language “feel”. This is because proofs are just functions, there are no distinctions between “proof mode” and “definition mode”. In Coq, for example, the definition language Gallina [56] is an entirely separate sublanguage from the proof definition language, LTac [57].

These differences are minor compared to other languages described here. Agda could be used for the purposes of this thesis, but it is not built with systems programming as a core function, and as such still seems more math-oriented than programming-oriented. Extracted Agda code is substantially larger than equivalent Idris code.

5.3.1 Sample Code and Results

A sample, “hello world” program for Agda (found online at [8]) is shown in figure 5.5.

```
module hello where
open import IO.Primitive using (IO; putStrLn)
open import Data.String using (toCostring; String)
open import Foreign.Haskell using (Unit)
main : IO Unit
main = putStrLn (toCostring “Hello, Agda!”)
```

FIGURE 5.5: Agda “hello world”.

As stated in the previous section, the code above compiles to an enormous executable of approximately 20MB. This is due to representation issues similar to those shown for the Coq extraction in figure 5.2. As a result, we have to consider the language to be insufficiently performant to be a serious candidate for our purposes.

To summarize the results of the Agda analysis, we find the results shown in table 5.2.

[8] http://stackoverflow.com/questions/9692445/agda-as-a-programming-language

System	Expressive?	Provable?	Extractable?	Performance?
Agda	Yes	Yes	Yes	No

Table 5.2: Agda Results

5.4 Haskell

Haskell [110] is a general purpose language. It has all the features we want from such a language - a rich client library, web framework, executable code/compiler. It is also a pure-functional language, with third-party support for termination checking and branch coverage. Haskell, out of the box, does not support universal quantification or termination checking nor other proof assistant structures. However, we can add on many "dialects" to make Haskell more like the proof assistant types. For example, generalized algebraic data types (GADTs)[9] are one effort to make Haskell's type system closer to a full dependently typed system. However, they are limited to "top level" types, and cannot be used in sub-expressions [122]. This means they are not quite as powerful as a full dependent type system.

While the QuickCheck [77] library adds statistical sampling support (to select and verify behavior for an arbitrarily large number of fixed inputs), we still deal with the limitations of existential and non-universal quantification. Haskell extensions such as "ExistentialQuantification"[10] get us closer to full dependent type flexibility and general universal quantification, but expressions remain that cannot be encoded with Haskell's "forall" statement [122].

Haskell is the most powerful general purpose language discussed here, and is closest to the theorem provers. However, the gaps cited above are sufficient to rule it out as a candidate for our purposes. Despite this, it is an excellent starting point for learning related languages like Idris. In particular, the mathematical side of Haskell can be explored by following along the "Haskell Road to Logic, Maths and Programming" [123].

5.4.1 Sample Code and Results

A sample, "hello world" program for Haskell is shown in figure 5.6.

[9] https://en.wikibooks.org/wiki/Haskell/GADT
[10] https://en.wikibooks.org/wiki/Haskell/Existentially_quantified_types

```
main = putStrLn "hello world"
```

FIGURE 5.6: Haskell "hello world".

While Haskell is performant and expressive, we find ourselves unable to write full proofs. Therefore, we cannot close some important gaps in the common workflow (described in figure 4.1), nor solve the 'existential problem' (see section 6.2.1 for details).

To summarize the results of the Haskell analysis, we find the results shown in table 5.3.

System	Expressive?	Provable?	Extractable?	Performance?
Haskell	Yes	No	Yes	Yes

TABLE 5.3: Haskell Results

5.5 Isabelle

Isabelle [59] is a "generic system for implementing logical formalisms"[11]. It is constructivist language, and is also very focused on the math side of things.

It does not support compilation directly, as the base language does not support Input/Output and related stateful functions[12]. It does support extraction for certain subsets of the language, via a mechanism similar to Coq extraction. Isabelle extractions share the same issues of representation, type safety and performance that Coq extractions exhibit.

The libraries are also focused on math and logic. There is little string-handling support, no web development framework, and like Coq, stateful functions are bolted on via third party libraries. Typically, the proofs associated with bolted-on state functions are based on separation logic [53, 93] or traces [106].

5.5.1 Sample Code and Results

A sample theory (a "hello world" program requires substantial third party libraries and ML support, and is not expressable in basic Isabelle syntax) in Isabelle is shown in figure 5.7.

[11] https://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/Isabelle2015/doc/prog-prove.pdf
[12] https://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/Isabelle2015/doc/codegen.pdf

```
theory Prop
imports Main
begin
theorem A: "A → A ∨ B"
apply (rule impI)
apply (rule disjI1)
apply assumption
done
end
```

FIGURE 5.7: Isabelle basic theory (in lieu of "hello world" program).

While extractions are possible in Isabelle, the inability to do basic state and I/O manipulation makes the expressive power insufficient to be further considered. Due to the extraction to ineffecient forms of ML (similar to Coq), we find the language unsuitable for our purposes.

To summarize the results of the Isabelle analysis, we find the results in table 5.4.

System	Expressive?	Provable?	Extractable?	Performance?
Isabelle	No	Yes	Yes	No

TABLE 5.4: Isabelle Results

5.6 Idris

Idris is a "systems programming language with dependent types" [61]. It can be thought of as a combination of Agda and Haskell. The similarities to Agda are that it supports full proof statements, iterative refinement via "holes" and dependent types based on Martin-Löf type theory. The similarities to Haskell are syntactic, as well as Haskell being the implementation and extension language.

Idris is directly compilable, supports universal quantification and has good libraries for web development. Full universal quantification support is essential to provide proofs. Direct compilation has a number of benefits beyond extraction (the issues with extraction have been discussed in previous sections), including much smaller executables (for example the "hello world" program in Idris compiles to an executable approximately 2000 times smaller than a comparable Agda executabe). Furthermore, direct compilation precludes any manual editing of intermediate artifacts. The toolchain supports debugging, with connection to the line numbers causing compilation issues. The IDE

support, available in both vim and emacs, including syntax highlighting, proof querying (similar to Agda) and interactive proof (similar to Coq).

By design Idris features:

> "easy interoperability with C and high level language constructs to support domain specific language implementation. Idris emphasises general-purpose programming, rather than theorem proving, and as such includes higher level programming constructs such as type classes and do notation. Idris also supports tactic based theorem proving, and has a lightweight Hugs/GHCI style interface." [13]

5.6.1 Sample Code and Results

A "hello world" program for Idris is given in figure 5.8.

```
module Main
main : IO ()
main = putStrLn "hello world"
```

FIGURE 5.8: Idris "hello world".

Unlike proof assistants based on extraction, Idris is directly compilable, and as such produces much more efficient runtime code than the competitors in this section. Further details and benchmarks to prove this claim are mentioned in sections 8.3 and 10.1.

To summarize the results of the Idris analysis, we find the results in table 5.5.

System	Expressive?	Provable?	Extractable?	Performance?
Idris	Yes	Yes	Yes	Yes

TABLE 5.5: Idris Results

5.7 Summary

None of the languages have every feature we might want, some can be ruled out directly. Agda and Isabelle are too focused on math. Haskell is too general purpose,

[13] Idris FAQ: http://docs.idris-lang.org/en/latest/faq/faq.html

without enough support for proofs. The remaining options are Coq and Idris. They represent two competing sides: the CIC and Martin-Löf type theories. However, Coq's runtime/extraction is inefficient and hard to reconcile errors in the extracted side and the original source code.

The criteria for selecting a language for this work were expressive power, theorem proving ability (sufficient to perform universal quantification), extraction/compilation, and performance. Idris has sufficient expressive power to be used as a general purpose language (by design) and has library support for many common tasks (including web development). It supports machine verified proof and universal quantification over its datatypes and can be directly compiled to produce efficiently sized executables with reasonable performance (see section 10.1 for details). Because of these characteristics, we have chosen Idris as the basis for our further work.

A summary of the evaluation results for the languages discussed in this section according to the criteria above is presented in table 5.6.

System	Expressive?	Provable?	Extractable?	Performance?
Coq	Yes	Yes	Yes	No
Agda	Yes	Yes	Yes	No
Haskell	Yes	No	Yes	Yes
Isabelle	No	Yes	Yes	No
Idris	Yes	Yes	Yes	Yes

Table 5.6: Proof Assistant Options

Chapter 6

Proof Driven Development (PDD)

The software development lifecycle (SDLC), for both "normal" software, and "mission critical" software requiring the utmost security includes: specification, design, implementation and testing. However, there are many schools of thought concerning the "best" way to organize the progress between these states during the construction of software. In waterfall development (described in detail in section 6.1), the states are not revisited, but progressed through linearly. In so-called "Agile" processes (see section 6.2), such as "eXtreme Programming" (XP) [69], these states are visited repeatedly during the construction of software, in a spiral pattern. These two schools of thought about the construction of software tend to apply at different times. Waterfall is well-suited to problem domains where the issues are well-understood, and new unknowns are not likely to arise during construction. Agile is better suited to exploration and research, as it does not make the assumption that all the facts are in before implementation begins, and permits requirements to change during the construction process.

This workflow is pictured in figure 6.4. The numbers for each stage in the figure refer to the artifacts, showing clearly the flow from specifications to mental model to mathematical model to source code. The source code is then compiled into the program, and finally the proofs are written.

In this section an argument is made for a mixed approach to secure software construction, as there are important lessons to be learned from both approaches.

6.1 Waterfall

Waterfall assumes that all design decisions can be made before the software/system is constructed. This tends to apply in well-understood areas. Waterfall grew out of very early attempts to formalize software construction processes [73]. These early approaches were based on electrical and hardware engineering techniques. The authors admitted that it was not an ideal approach due to the differences between these engineering disciplines (being well understood) and the mathematically/algorithmically oriented software engineering discipline which was only just emerging.

Many Agile adherents [16, 17, 67, 69] argue that waterfall is costly and unrealistic. However, pre-loading the design and specification phases is very much what tends to happen with secure software systems, especially considering the academic sources of many designs/specifications. Many of the specifications/systems have been proven on paper and published without the original authors intending to implement the systems themselves. This allows for a great deal of caution and review from many expert eyes. Indeed, the most common workflow is described in section 1 and figure 4.1, and is based on this idea.

However, the criticisms of waterfall in terms of emerging issues, and the speed with which the final system is available are valid, and it is clear that the waterfall approach by itself is not sufficient.

6.2 Agile/TDD

Agile itself is a blanket term for many software development processes including Clean Room [72] – which incorporates the use of formal methods such as model checking, process algebras, and testing), XP [69], and SCRUM [17]. The core values of Agile systems are embodied by their manifesto:

> We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
>
> - Individuals and interactions over processes and tools
> - Working software over comprehensive documentation
> - Customer collaboration over contract negotiation

- Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.[1]

Clearly, there is a large variety of so-called Agile methods, and they all vary from one to the next. However, one key discipline is notable in a great deal of these methods, called "test-driven development" or TDD. Test-driven development was described in great detail by Kent Beck in his introductory guide to TDD [68], and it is described in a great deal of related texts [16, 17, 67, 74, 124].

Test-driven development refers to the reversal of the "typical" workflow for creating software. Instead of writing a piece of software, and then writing a test to determine if the software is working correctly, the test is created first, to "drive" the creation of the software required to satisfy the assertions in the test. This is important for several reasons, described in detail by Beck and Martin in their works on software craftsmanship[16, 68, 69]. First, budget and timeline pressure often forces things on the end of the spectrum to be dropped (that is: the tests never get written). Second, the tests "driving" (or "pulling", in Toyota Production [125] or Lean [126] terminology) the functionality helps to keep the system minimal ("lean"), with no unnecessary functions. Third, the creation of tests up-front force the developers to think of the users of the code (be it other systems or end-users). Finally, the tests form a suite of properties that can be automatically verified, thus helping to avoid regression errors (re-introducing bugs fixed before into new releases).

While TDD has gained substantial traction in recent years, with dozens of publications extolling its virtues, the idea is not new. Anecdotally[2], the Mercury Space Program at NASA used a variant of TDD with their punch-card programming system: punching the cards with the expected output, then programming the system until the output punchcards matched. What has changed recently is the availability and quality of tools built to support this new workflow. Many frameworks are based on early work by Beck [68], who created the first "xUnit" unit testing framework, SUnit for Smalltalk. While SUnit (and Smalltalk itself) did not gain a huge marketshare (below 0.21% according to TIOBE [3]), JUnit, a Java saw a great deal of success - with many books discussing it, including Beck's and Martin's well known volumes on practical software construction

[1] http://agilemanifesto.org/
[2] http://c2.com/cgi/wiki?TenYearsOfTestDrivenDevelopment
[3] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[16, 68]. These frameworks made TDD approachable to industry, and promoted the "red-green-refactor" workflow of creating a failing test case (red), making the test case pass by creating the simplest code possible (green), then refactoring the code to make it clean [124]. The workflow has had a remarkable effect on defect rates, with a study at IBM showing a 50% reduction over ad-hoc testing [74].

Testing and test coverage (the proportion of code-paths covered by tests) are influenced by the source language, and the complexity [76] of the methods/classes under test. The limiting factor is that fundamentally all tests are instances of fixed value quantification assertions[4]. That is, for a function f, we can specify the output, y, for any fixed input $y = f(x)$. We can do this for arbitrarily many values of x, but it is still finite instances of fixed value quantification. We are saying there is an x for which this function behaves correctly. Extending this idea with statistical sampling we arrive at the work of Claessen and Hughes, QuickCheck [77]. Their work is based on the idea that any value from a range of inputs should produce valid output within another well-defined range (and not crash). The system uses these ranges as inputs and statistically samples values within them (the number of which is configurable) and synthesizes new assertions/test cases. This is analogous to how a human would extend the same tests, but without the tedium of creating them manually.

Even with such statistical sampling, we cannot say the function is correct for any input. In formal terms we cannot state "$\forall x, f(x)$ is valid." Section 6.2.1 shows why this is still a source of defects.

6.2.1 Example Fixed Value Quantification Problems

To illustrate why testing based on "arbitrarily many" fixed inputs is not sufficient to cover all error cases that a universally quantified assertion would cover, we present the following example. In mathematical terms, we define a function $f(x) = x/x$ where the inputs and outputs of f are the Real numbers. In code, this could be implemented as shown in figure 6.1.

Plotting this function, we would see the graph in figure 6.2.

For arbitrarily many values this method gives the correct and expected return value, 1. However, for the input 0, this program causes a division by zero error. So, not only

[4]This is distinct from "fixed point" values, a mathematical concept denoting a value x such that $f(x) = x$ for some function $f : X \to X$.

```
double f(double x) {
    return x/x;
}
```

FIGURE 6.1: C program with bug.

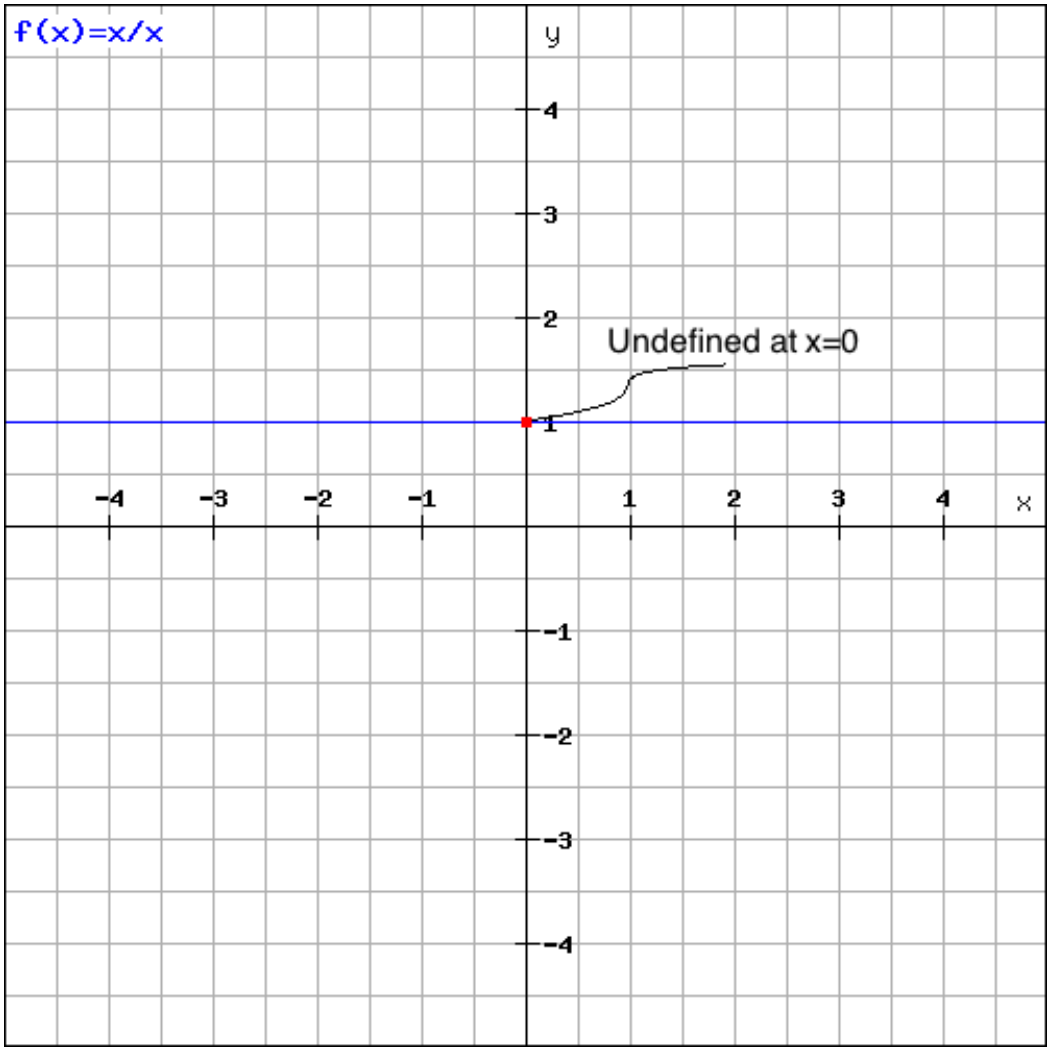

FIGURE 6.2: Graph with hole.

does one value (just one, out of as many distinct values as the data type supports) give an unexpected result, this input actually crashes the program. While this is a contrived example, the issue exposed here is very real.

It is not only numerical issues at play here, that can cause unexpected bugs during translation from symbolic mathematics into computational datatypes and functions.

Take for example the Java method in figure 6.3.

```
String convertBoolean(Boolean b) {
       return b.toString();
}
```

FIGURE 6.3: Java program with bug.

It would seem that by testing the function with the values *true* and *false*, we would exhaustively test this function. However, the Boolean data type in Java is actually tri-state. It is possible to crash this program by passing *null*, a Boolean (distinct from the primitive boolean data type) value that is neither true nor false. To be truly confident in our reasoning about the behavior of functions we need to be able to assert behavior for all possible values of our data types.

6.3 PDD

In order to close gaps in the workflow described in chapter 1 (figure 1.1), as shown in the diagram 6.4, we extend the notion of test driven development and Clean Room software engineering with what we dub "proof-driven development", PDD. The term has been used before by Bird to describe a similar technique used to iteratively refine an algorithm to solve Sudoku puzzles [127]. However, the proofs described therein are not machine verified (essentially Bird is talking more about proof-backed refactoring). Using proof-driven development techniques without automation leaves the system (and proofs) vulnerable to regression errors. Regression errors are a common problem in areas that require change or evolution (as with nearly all software systems).

Using the power of dependent types in a proof assistant language chosen in chapter 5, we are able to express specifications with universal quantification. Furthermore, by selecting a language with extraction/compilation (again, see section 5), we close an entire problematic gap in the workflow diagram. The improved workflow is shown in figure 6.5. The first two steps, creating specifications and synthesizing a mental model, remain the same as in the original workflow. However, the next step is to create proof statements of desired properties with the syntax of the proof assistant. These proofs (and the output from the compiler/verifier) guide the creation of the mathematical model. The mathematical model takes the place of the extracted/derived source code, and the compiler directly produces a program.

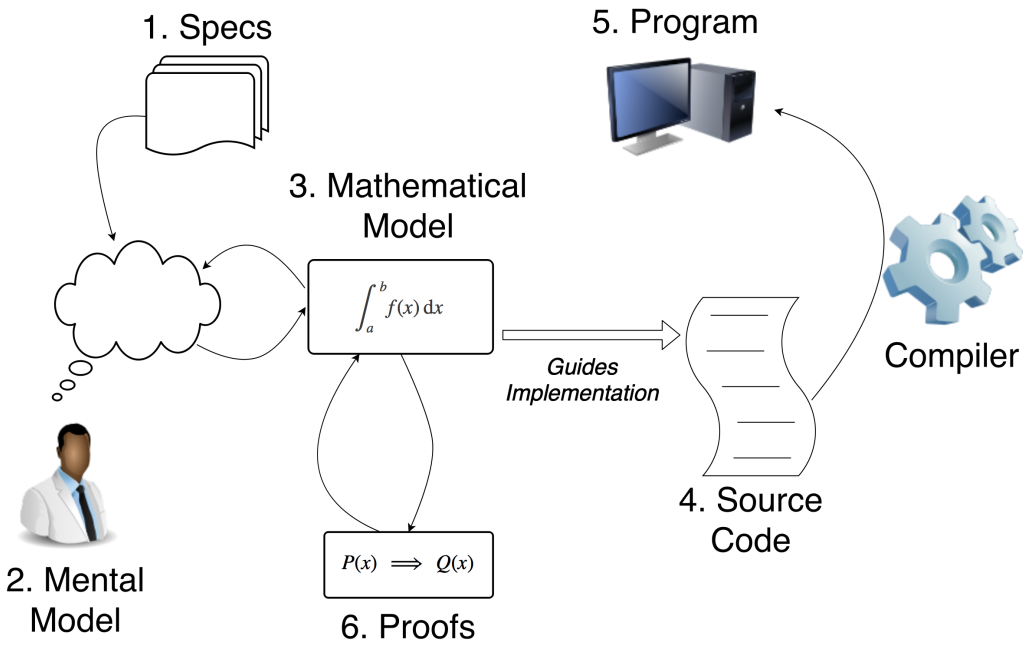

FIGURE 6.4: Original workflow (numbered).

As with TDD, success will depend on the quality of the framework and language upon which the system is based. When we make a proof-based, universally quantified assertion, such as "$\forall x, f(x)$ does not crash", we need our system to tell us if in fact we have changed f such that it is not longer true. Ideally during development, the system will guide us/the user to the "correct" data types to complete the proof of correctness.

Systems like Agda and Idris that support "holes", where the compiler can tell what type of computation is needed to get from state A to state B, are close. Similarly, systems with proof search such as Coq, Agda, and Idris, can make use of the compiler information about the hole in the proof.

To illustrate this, consider the line of reasoning about transitivity of homomorphisms in Idris in figure 6.6. This is a statement and partial proof of transitivity of semigroup homomorphisms (see section 9 for details of this contribution). It states that two homomorphisms h and h' can be evaluated in either order. The proof steps are elided by the holes "?prf1" and "?prf2". The system can be queried as to the exact type required for each hole (which in this case is a complex expression due to the dictionary tracking, necessary for distinguishing which semigroup operation is being referenced).

This signature can then be passed to the proof search engine. For example, if we needed a transformation from Nat (the type denoting natural numbers in Idris) to Bool (the

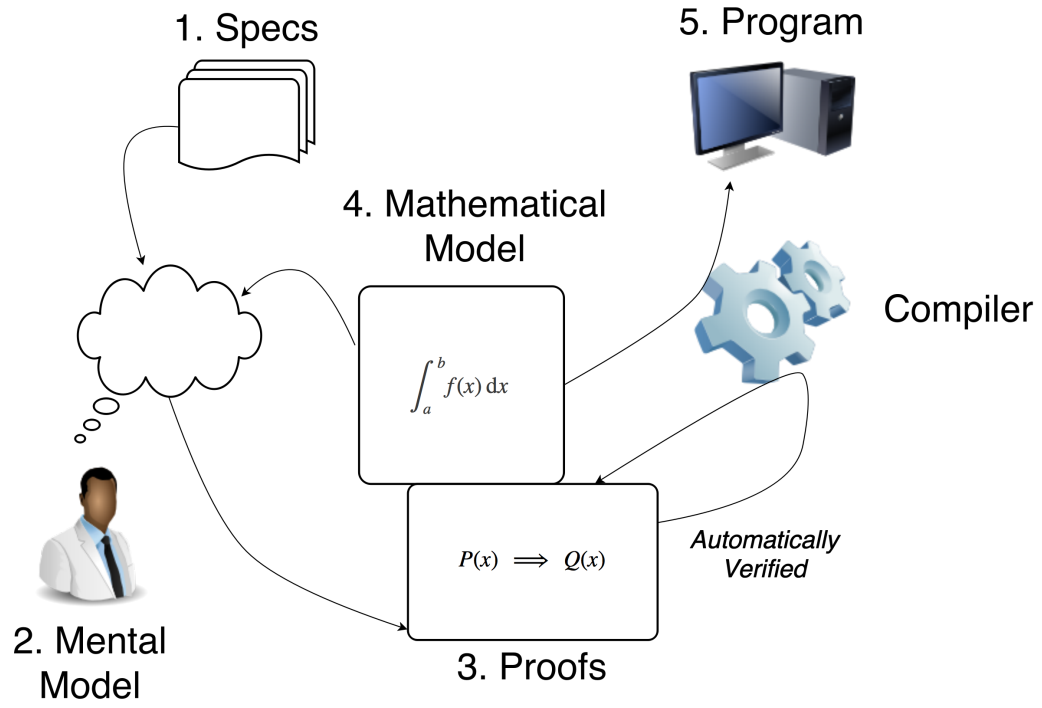

FIGURE 6.5: Improved workflow (numbered).

type denoting boolean values in Idris), we could search as shown in figure 6.7. The results of which shows us that the system knows two such predicates, isSucc (a decision property to determine if a given natural number is a successor form (greater than zero), and isZero, which matches if the natural number is zero.

```
homTrans : (adict : Semigroup a, bdict : Semigroup b,
            cdict : Semigroup c) => Hom a b adict bdict ->
                    Hom b c bdict cdict -> Hom a c adict cdict
homTrans @{adict} @{bdict} @{cdict} (MkHom h preservesGroup)
         (MkHom h' preservesGroup') =
         MkHom @{adict} @{cdict} (\x => h' (h x))
               (\something, another =>
                    (h' (h (something <+> another)))   ={ ?prf1 }=
                    (h' (h something  <+> h another))  ={ ?prf2 }=
                    (h' (h something) <+> h' (h another)) QED)
```

FIGURE 6.6: Idris equational reasoning.

These two features, together with broader type libraries, utility lemmas and improved error reporting should go a long to way to making PDD a viable secure software construction method.

```
> :search Nat -> Bool
= Prelude.Nat.isSucc : Nat -> Bool
= Prelude.Nat.isZero : Nat -> Bool
```

FIGURE 6.7: Idris proof search.

6.3.1 Design By Contract

Meyer, in his pioneering work on Design By Contract (DBC) [128], was one of the first to realize this logical extension of Hoare logic (preconditions, invariants and postconditions). Proof driven development extends this idea further. The limitations of the language's type system define the limits of the assertions that can be made within the contracts. These limitations include being limited to existential assertions, and the capabilities of statically typed systems with weaker foundations than dependent types. If previous systems like Java, C#, C++, C, Python, Haskell, etc. supported dependent types, design by contact frameworks would be capable of encoding very nearly the same types of assertions (both existentially and universally quantified).

This class of limitations applies regardless of the "level" at which the predicate is tested. Specifically, in DBC, three levels are available: preconditions, postconditions and (loop-) invariants. At each level the same limitations of what properties can be asserted exist. For example: Java types cannot define functional types, so a method cannot take another method as an argument; also, Haskell's type system (including algebraic data types) can only parameterize based on existing top-level types (the full details of this are available in Weirich's paper "Depending on Types" [122]).

6.4 Examples

In this section, we show examples of the issues that arise following a "standard" approach. We illustrate the changes to design and how we can utilize the compiler and error messages in a "PDD" approach.

The challenges that arise from a standard approach are discussed in section 6.4.1. The alternative, PDD, approach is discussed in section 6.4.2. In order to provide a valid comparison, we will show issues with creating a function to parse characters into integers. For example, the character '1' represents the integer 1, and should successfully parse. On the other hand, the character 'N' does not represent a valid integer and should not parse.

6.4.1 Standard Approach

In this section we will work through the creation of a function and associated proof in the "standard" workflow.

In Idris, the notion of parsing a character that may or may not be valid can be captured with the definition in figure 6.8. We look for the 10 valid cases, and return Nothing if it does not match any of them. (Clearly this code can be improved, but for illustration purposes it is clear that it is correct).

```
parseInteger : Char -> Maybe Int
parseInteger '0' = Just 0
parseInteger '1' = Just 1
parseInteger '2' = Just 2
parseInteger '3' = Just 3
parseInteger '4' = Just 4
parseInteger '5' = Just 5
parseInteger '6' = Just 6
parseInteger '7' = Just 7
parseInteger '8' = Just 8
parseInteger '9' = Just 9
parseInteger _   = Nothing
```

FIGURE 6.8: Idris type declaration for parsing integers.

We would like to prove, then, that any character not between '0' and '9' yields Nothing. Suppose we have a function "isNumeric" which takes a Char and returns a Boolean (indicating whether the character is between '0' and '9'). Then, we can define the statement of such a proof as shown in figure 6.9.

```
invalidCharAlwaysNothing : (c : Char) -> (pf : isNumeric c = False) ->
                           parseInteger c = Nothing
invalidCharAlwaysNothing c pf = ?invalidCharAlwaysNothing_rhs
```

FIGURE 6.9: Idris proof statement.

When we attempt to prove the statement, the problem becomes intractable (figure 6.10). This is because there is nothing to connect the implementation of "parseInteger" to the stipulated portion of the proof, specifically that the character is numeric. To move forward, we must examine the definition of our parse function, and change it to use our numeric query function, or similarly sever the conditional logic. This happens after

```
parse> :p invalidCharAlwaysNothing_rhs
----------              Assumptions:              ----------
c : Char
pf : isNumeric c = False
----------                  Goal:                  ----------
{hole3} : parseInteger c = Nothing
```

FIGURE 6.10: Idris proof status.

10 lines of very obvious implementation code. The problem grows with larger code segments.

As an example of a larger code segment, during string concatention, if we allow for "negative characters" (such that 'a' concatenated with its negative would yield the empty string) proving even a simple property such as that a string concatenated with the empty string becomes extremely involved. The proof state corresponding to this is shown in figure 6.11. (The full details of such string manipulation are available online[5]).

```
----------              Assumptions:              ----------
 c : SignedChar
 w : Word
 inductiveHypothesis : wordConcatAndCollapse w Empty = w
----------                  Goal:                  ----------
{hole3} : case block in
   wordCollapse (c # wordConcat w Empty)
                (wordCollapseOneLevel (c # wordConcat w Empty))
   (Words.Word instance of Prelude.Classes.Eq,
       method == (c # wordConcat w Empty)
                 (wordCollapseOneLevel
                    (c # wordConcat w Empty))) = c # w
```

FIGURE 6.11: Idris complex proof status.

6.4.2 PDD Approach

As shown in figure 6.9, if we approach the same problem with a proof statement first but before creating the code, the necessary connection between the qualifier (isNumeric) and the implementation is more obvious. In fact, we can now encode the relationship directly into the datatype (figure 6.12).

[5] https://github.com/bgoodspeed/idris-strings

```
data NumChar : Type  where
  MkNumChar : (c : Char) -> (ok: isNumeric c = True) -> NumChar
```

FIGURE 6.12: Idris data type declaration.

```
parseInteger : NumChar -> Maybe Int
```

FIGURE 6.13: Idris improved parseInteger declaration (A).

This means we can declare the signature of our "parseInteger" function using the new datatype (figure 6.13). Using this, we realize that we cannot even construct a call to "parseInteger" using a non-numeric character. The compiler refuses to unify "False = True" (figure 6.14). This means we can remove the possibility of failure from the method signature, removing the Maybe clause (figure 6.15).

```
parse> parseInteger (MkNumChar 'c' Refl)
(input):1:25:When elaborating argument ok to constructor Main.MkNumChar:
        Can't unify
                x = x
        with
                isNumeric 'c' = True

        Specifically:
                Can't unify
                        False
                with
                        True
```

FIGURE 6.14: Idris compilation error for bad invocation.

```
parseInteger : NumChar -> Int
```

FIGURE 6.15: Idris improved parseInteger declaration (B).

This small change in workflow for a trivial function has many improvements to the resulting design. This is true not only to the design of the code developed here, but also for any client code making use of the function. Now, rather than using a parse routine and checking afterwards to see if something went wrong, it is up to the client to prove nothing will go wrong *before it is allowed to make the call.*

6.5 Impacts

As shown in the previous section, the use of PDD extends TDD (it is a strict superset) and permits universal quantification of system properties. The use of proof assistants in general close an important security gap, as shown in figures 4.1 and 4.2.

Just as TDD was shown to alter the structure of the code under test/development (by making hooks for tests to assert intermediate results), so too does PDD. In this case, we have intermediate placeholders for assertions, as well as intermediate dependent data types (which carry their properties with them, similar to invariants in Hoare logic [94]).

It is possible for a proof script itself to contain errors, thus "locking in" erroneous behavior in the system. Of course, this is true of ink and paper proofs as well. The traditional remedy for such paper proofs is peer review. The corresponding software quality technique is code review, which has been shown to positively impact quality [129]. This is sometimes done during development, as in pair programming used by the XP [69] process, and sometimes after the system is completed. The benefits arising from code reviews happening after construction is that the system is viewed by someone not involved in its creation. This has had a substantial impact on the quality of systems as a whole [129, 130].

It should be noted that this is not a 'silver bullet' [131]. It should be easier to prove properties about code created with proofs first than trying to prove properties post-hoc. It is worth noting the extreme cost and difficulties associated with machine verified proofs. Substantial work is required for making this easier for industrial users as well as expert academics before this can become mainstream.

Chapter 7

Secure Programming

To provide a meaningful discussion of practical programming in a proof-assistant environment, we present a discussion of a secure login program. Login programs solve the "authentication" problem, thereby confirming a user is who he or she claims to be.

Implementing a login program exposes several key issues in secure programming in Idris: string handling, cryptography, input/output and error handling. Furthermore, the development of the program in the common workflow revealed several difficulties associated with post-hoc proofs of correctness. These challenges helped motivate the improved workflow described in section 6, several of the string handling functions in section 8 and some of the mathematical tools described in section 9.

7.1 Authentication

Many authentication schemes exist. For example, HTTP defines seven mechanisms supported by the web server Internet Information Server (IIS)[1]. All of these mechanisms are based on "knowledge factors" which are authentication based on something the user knows. Historically authentication systems have been based on this type of system, including systems based on passwords, one-time hashes, etc.

However, two other categories exist[2]: "ownership factors" (something the user should have, e.g. an ID card) and "inherence factors" (something fundamental to the user, e.g. a fingerprint). When a security system uses "two-factor" authentication, it refers to the

[1] http://msdn.microsoft.com/en-us/library/ms789031(v=vs.110).aspx
[2] http://en.wikipedia.org/wiki/Authentication

demand that authentication is provided from more than one category. These systems and combined mechanisms are beyond the scope of this thesis.

Focusing on knowledge based authentication, passwords are by far the most common type of authentication system. The default implementation is to store a cryptographic hash [3] of user passwords in a database of varying types.

Unix based systems include the Linux family (Debian, Ubuntu, Redhat, etc), the BSD family (Mac OS X, FreeBSD, OpenBSD, etc), and "traditional" Unix systems (Solaris, AIX, etc). Many of these, including the BSD and Linux families support a variant of the password-database-driven mechanism described herein. Two very common authentication systems are BSD_AUTH [132] and Pluggable Authentication Modules (PAM)[4]. The password database, typically stored in /etc/passwd, or /etc/master.passwd[5] is supported by both and is the default configuration for many systems, including OpenBSD.

Since Unix based open source systems provide a convenient and popular platform these days[6] we've chosen to focus on BSD Auth, specifically OpenBSD. A reduced C implementation showing the workflow is in figure 7.1. The system can also alter the hash function to be used for password storage, by editing the appropriate configuration file as described in section 7.1.2.

```
1  ...
2  as=auth_open();
3  auth_setoption(as,"login","yes");
4  auth_setitem(as,AUTHV_NAME,username);
5  pwd=getpwnam(username);
6  auth_setpwd(as,pwd);
7
8  auth_verify(as,style,NULL,lc->lc_class, NULL);
9  authok=auth_getstate(as);
10 auth_close(as);
11
12 if(authok)
13 ...
```

FIGURE 7.1: Reduced C Login Program Logic.

[3] http://en.wikipedia.org/wiki/Cryptographic_hash_function
[4] http://www.opengroup.org/rfc/rfc86.0.html
[5] http://www.openbsd.org/cgi-bin/man.cgi?query=passwd&sektion=5
[6] http://w3techs.com/technologies/overview/operating_system/all

7.1.1 Implementation in practice

The OpenBSD system is well regarded as a secure open source operating system[7], and the project has strong ties with the popular OpenSSL cryptographic software project.

The implementation workflow shown in figure 7.1 is actually spread across several files (assuming the MD5 hash algorithm is the configured digest): login.c[8], auth_subr.c[9], and md5_dgst.c[10] This makes the information/code flow particularly difficult to reason about. Proving theorems about the system would be exceedingly difficult.

Due to the interaction with sensitive information (specifically user passwords), and the need to allocate system resources (such as spawning a shell, assigning a TTY to said shell etc), the login program necessarily runs at a high privilege level (specifically it is a seteuid root program, as described by the POSIX specification [48]). A search for seteuid or setuid on the CVE shows how frequently this high privilege execution causes security issues.

Since the password database must be readable by the login program, it is essential that passwords not be stored in cleartext. As cryptographic methods are notoriously difficult to implement correctly[11], the authentication system (be it PAM or BSD_AUTH), delegates the calculation of hashes (which is what is actually stored in the password database) to a full fledged cryptographic library. In the case of OpenBSD, using BSD_AUTH, it is delegated to the OpenSSL library[12]. This usage is discussed in the cryptography section (7.1.2).

7.1.2 Cryptography

The standard configuration for BSD and related systems is for passwords to be stored after being digested by a hash function. In BSD_AUTH this is done in the login.conf[13] file, and looks like this[14]:

[7] http://www.openbsd.org/security.html
[8] http://www.openbsd.org/cgi-bin/cvsweb/checkout/src/usr.bin/login/login.c
[9] http://www.openbsd.org/cgi-bin/cvsweb/checkout/src/lib/libc/gen/auth_subr.c
[10] http://git.openssl.org/gitweb/?p=openssl.git;a=blob;f=crypto/md5/md5_dgst.c
[11] http://en.wikipedia.org/wiki/MD5#Collision_vulnerabilities
[12] https://www.openssl.org/docs/apps/passwd.html
[13] http://www.openbsd.org/cgi-bin/man.cgi?query=login.conf&sektion=5
[14] http://www.openbsd.org/cgi-bin/cvsweb/checkout/src/etc/etc.i386/login.conf

```
default:
:passwd_format=md5:
```

FIGURE 7.2: Unix password database configuration.

Password hashes can be calculated on the fly using the command "openssl passwd -1 new_password". Per the man page on openssl passwd[15], the default is md5.

Many hash functions suffice for calculating password hashes, and can be configured within the BSD auth system. Historically MD2/5 was used, and modern systems use SHA256/512 or Blowfish[16].

However, many hash functions have had issues with birthday attacks [17], where two messages that hash to the same value can be interchanged. Likewise, hash functions using smaller keyspaces have been shown to be vulnerable.

Since a strong cryptographic hash function SHA256 has been proven correct and implemented in Coq by Appel [13], we do not prove the security of our hash function. He demonstrated the implementation fulfills the specification FIPS-180 [11]. Additionally he claims: "We know there's no heartbleed in SHA."

We elide the complexity of implementing and proving security properties of the hash function in this implementation of a login/auth system. Since there are many options for hash functions, and the choice doesn't matter so much except for the property that for any pair of messages m_1, m_2, and hash function h we have $m_1 \neq m_2 \Rightarrow h(m_1) \neq h(m_2)$ with some strong probability, we employ a trivial function with only the above property, the function that reverses the given string. The reversing function has no collisions and is therefore suitable according to that criterion.

7.2 Formally Verified

Some work has been done on proving/verifying semantics of C code, notably the CompCert compiler that has been proven to preserve semantics as it translates from Clight (a subset of the C language) to PowerPC assembler [101]. However an alternative approach is to use a language with designed-in support for formal logic. There are several

[15] https://www.openssl.org/docs/apps/passwd.html
[16] http://openbsd.cs.toronto.edu/cgi-bin/cvsweb/src/etc/etc.i386/login.conf
[17] http://en.wikipedia.org/wiki/Birthday_attack

machine-assisted proof tools (several have been discussed in chapter 5). These languages are often based on a pure-functional core. This means certain standard programming languages features (mutable state, IO, etc) require some awkwardness. Often these are addressed by the introduction of Monads [108].

Successful projects like CompCert C compiler [101], the seL4 verified microkernel [50], the Quark web browser [84], and a verified implementation of SHA-256 [13] have been implemented in Coq, and we have selected a related system, Idris (for more details see chapter 5). The challenges in implementing and proving properties about the login program described herein motivated several of the contributions noted in chapters 8 and 9.

The general process for building software within formal verification frameworks (including Idris) is to develop definitions and functions in the formal language of the framework, prove properties about the definitions and functions, then extract runnable or compilable code from the formal language artifacts.

7.3 Implementation

The key to the authentication problem is the comparison of an encrypted password to one stored for that user. We would like to perform formal verification against an implementation which is as close as possible to what exists "in the wild". While it is not possible to prove something completely "secure", it is possible to show certain classes of errors cannot occur.

As discussed in previous sections, we elide several parts of the process (such as formal cryptographic hashing, signal trapping, re-try policies and other details) to simplify the workflow for discussion and proof.

In particular, we show that our implementation has the following properties:

1. improperly formatted password configuration lines (such as those without enough data) cannot be added to the database. In this case, a properly formatted configuration line contains exactly three colon (':') delimited fields. This is shown in part in figure 7.8,
2. an empty password database always leads to a rejected login regardless of username and password applied. This is shown in part in figure 7.7,

3. any login attempt with an incorrect username or password with either one or two entries in the password database which do not match the credentials will be rejected[18] . This is shown in part in figure 7.6.

Section 7.3.1 contains high level pseudocode of the authentication problem, as well as examples of the interesting cases the code needs to handle.

7.3.1 Examples and Pseudocode

Here is a snippet of a typical password database (typically stored in /etc/passwd):

```
nobody:C2/feOyBslbOzH//dJP4B1:-2:-2::0:0:Nobody
:/var/empty:/usr/bin/false
root:4t0I/EI66gdmLd02PALYB1:0:0::0:0:System Administra-
tor:/var/root:/bin/sh
daemon:wdy3JW/OPU6.sUnfFwPmV1:1:1::0:0:System Ser-
vices:/var/root:/usr/bin/false
```

FIGURE 7.3: Unix password database sample.

There are 3 important cases an authentication system needs to handle:

1. no such user (username: "bob", supplied password: "somepassword"): in this case the supplied username is matched against the value in field one, when the database is exhausted, failure is returned.

2. bad password (username: "root", supplied password: "bad password"): as above, except the line with the matching username is parsed and field two is compared to the hash of the supplied password, when it does not match failure is returned

3. success (username "root", supplied password: "password"): as above except the hashed password matches field two and success is returned.

In figure 7.4 we present the pseudocode for this logic.

In case 1, the first clause of the if statement on line 7 fails every time, as there is no user "bob".

[18]The logic for the proofs of these claims could be extended to an arbitrary number of password database entries

```
1 username=read_input()
2 password=read_input()
3 password_db=read_file()
4 for entry in password_db
5   fields=split_string(':',entry)
6   hashed=compute_hash(password)
7   if (fields[0]==username) and (fields[1]==hashed) return(OK)
8 return(BAD)
```

FIGURE 7.4: Pseudocode for authentication.

In case 2, the first clause matches on line two of the password file, indicating the "root" user does exist. However the hash of the supplied password evaluates to "qtweE/C4z2x48o5sE.uc.", which does not match the stored hash: "4t0I/EI66gdmLd02PALYB1", so the second clause on line 7 fails.

Finally in case 3, the first clause matches as in case two, and the supplied password hashes to the matching string in the database. Both clauses in the if statement are satisfied and the loop terminates early with a code indicating a successful login.

Due to the types developed, we know that malformed entries in the password file do not lead to records in the database, incorrect credentials will not yield a valid login.

7.3.2 Idris

Our implementation of the login workflow described previously is completed in Idris, as selected in chapter 5.

The core of the implementation is the "validateLogin" function (figure 7.5), which searches for the matching user record in the password database. It does this by delegating the check of each individual record to the "matchesLoginRecord" function, eventually returning true if any record matches the given username and password hash.

This implementation is the basis about which we proved several specific properties. In particular, this includes the fact that for any given entry in the database, and for any invalid username or password (ones for which the "matchesLoginRecord" function returns false), the system will not grant a login request. This proposition ("allBadLoginRecordsMeanFalseForAllUserAndPasswords") and the proof are given in figure 7.6. A related proposition ("allBadLoginRecordsMeanFalseForAllUserAndPasswords2"), namely that the same is true for any configuration with two login records will also reject bad logins, is also shown in figure 7.6, the difference being the need for more complex rewrites

(specifically, we must use each proof twice, once for each "side" of the equation) of the given assumptions in order to satisfy the proof checker. This type of logic can be extended to prove an arbitrary number of records in the database will maintain the same behaviour.

In figure 7.7, we show that given an empty password database, any login attempt (regardless of the supplied username or password) will always reject the login.

In figure 7.8, we verify that all candidate lines must have the appropriate number of entries. We do this by converting the split string into a vector, and then successively pattern matching against the number of elements in the vector. Only if the appropriate size vector is obtained (in this case a 3 element vector) can we create a login record.

The proofs supplied are not proofs of the total security of the program. However, they do illustrate a universally quantification proof statement surrounding a runnable program. Despite the limited size of the password database, we have demonstrated universal quantification over both the username and password choice, as well as the contents of the first two lines of the password database.

It is also worth re-iterating that the choice of a toy "hash" function (merely reversing a string) is not representative of real-world security. A real world version of this function would need to make use of a cryptographic hash function, and would need to use the salt values from the password database to defend against dictionary attacks.

The full details of our implementation , including source code and proofs as well as our comparison implementation in Coq are available online[19].

```
matchesLoginRecord : String -> String -> LoginRecord -> Bool
matchesLoginRecord u p lr = (u == (usernameFrom lr)) &&
                            (compute_hash p) == (hashedPasswordFrom lr)

validateLogin : String -> String -> PasswordDatabase -> Bool
validateLogin u p (MkPasswordDatabase list) =
  any (matchesLoginRecord u p) list
```

FIGURE 7.5: Idris code snippet showing part of a login program.

[19] https://github.com/bgoodspeed/idris-secure

```
allBadLoginRecordsMeanFalseForAllUserAndPasswords :
   (u : String) -> (p : String) -> (lr : LoginRecord) ->
   matchesLoginRecord u p lr = False ->
   validateLogin u p (MkPasswordDatabase [lr]) = False
allBadLoginRecordsMeanFalseForAllUserAndPasswords u p lr prf = ?pfHole


Main.pfHole = proof
  intros
  rewrite prf
  trivial


allBadLoginRecordsMeanFalseForAllUserAndPasswords2 :
   (u : String) -> (p : String) -> (lr : LoginRecord) ->
   (lr2 : LoginRecord) -> matchesLoginRecord u p lr = False ->
   matchesLoginRecord u p lr2 = False ->
   validateLogin u p (MkPasswordDatabase [lr, lr2 ]) = False
allBadLoginRecordsMeanFalseForAllUserAndPasswords2 u p lr lr2 prf prf1 =
  ?allBadLoginRecordsMeanFalseForAllUserAndPasswords2_rhs


Main.allBadLoginRecordsMeanFalseForAllUserAndPasswords2_rhs = proof
  intros
  rewrite prf
  rewrite sym prf
  rewrite prf1
  rewrite sym prf1
  trivial
```

FIGURE 7.6: Idris code snippet showing proofs how incorrect logins are handled.

```
emptyPasswdDB : PasswordDatabase
emptyPasswdDB = MkPasswordDatabase []

emptyPasswordDBMeansNoLogin : (u : String) ->
     (p : String) -> validateLogin u p emptyPasswdDB = False
emptyPasswordDBMeansNoLogin u p = Refl
```

FIGURE 7.7: Idris code snippet showing login program behavior of empty passwords.

```
data LoginRecord : Type where
    MkLoginRecord : Vect 3 String -> LoginRecord


splitv : (s : String) -> Vect (length (split (== ':') s)) String
splitv x = let xs = split (== ':') x
               vs = fromList xs in
                  vs

vectToLoginRecord : {n : Nat} -> Vect n String -> Maybe LoginRecord
vectToLoginRecord {n} xs = case natToFin 0 n of
    Nothing  => Nothing
    Just  z  => let un = index z xs in
      case natToFin 1 n of
        Nothing => Nothing
        Just o  => let ep = index o xs in
          case natToFin 2 n of
            Nothing => Nothing
            Just t  => Just (MkLoginRecord [un, ep, (index t xs)])
```

FIGURE 7.8: Idris code snippet showing invalid configuration handling.

Chapter 8

Provable Software Building Blocks

Now that we have selected a language (Idris) and are armed with a new process for creating code in a language that supports proof, we turn our attention to the problem of cost (in terms of time and mental effort). It has been established that "it is easier to learn new facts that are comprised of more familiar elements" [133]. As software construction can be viewed as the composition of ideas, this finding suggests familiar libraries are an extremely valuable tool for learning new languages and techniques. To this end, we use a standard approach in software engineering. Specifically, we provide solutions to common problems and package the resulting software components for re-use.

8.1 String Handling

String-handling is an important tool for both practical programming and theoretical research. In this section, we examine the implementation of strings in several programming languages. We extract the common features, describe the mathematical structures they exhibit, and present alternatives for formal representations of strings in proof assistant languages (particularly in Idris).

8.1.1 General purpose language string representations

C is still the most used language[1], and is the basis for a number of other programming languages (including C++, C#, Python, Ruby and others). As a result, many languages have inherited C's representation of Strings either directly (by exposing C string primitives) or indirectly (by modeling their own representation in the same style as C). C uses arrays (contiguous memory blocks) to store character data in sequence. C strings need to be null-terminated (by a sentinel value, denoted '\0'). This usage causes a special case of the semipredicate problem [134] (where a legitimate return value must be reserved for signaling errors in a function).

Object oriented languages tend to use either a linked list behind the scenes, similar to Lisp [135], or a structure containing the array as well as the length [136]. Both of these approaches eliminate the sentinel value/null termination issue, and thus resolve the semipredicate problem. Other data structures, such as trees and heaps, can be used to represent strings, but these tend not to be used often.

Regardless of the representation, all mainstream programming languages support strings either natively or in a standard library. Generally, the internal representation is not accessible to programmers short of compiler/interpreter extensions [137].

8.1.2 Using Proof Assistants to Formalize Strings

Most general purpose languages permit mutable strings (a notable exception is Haskell [110]). Mutable data structures and mutable state in general are a problem for formal reasoning. Purely functional languages like Haskell do not support changes to state or side effects in functions, and as a result it is possible to automatically reason about their behavior. To alleviate this problem, proof assistants require either side-effect-free functions (as in Haskell), or a type system that supports quantification over values and types, such as a Martin-Löf type theory [63]. Therefore, to formalize strings in a programming language context we require a proof assistant (for example one based on the calculus of constructions [58] such as Coq [56]), or a dependently typed language, such as Agda [60] or Idris [61].

Without the power of a proof assistant or dependently typed language, all of the properties of strings can only be demonstrated at the fixed value quantification scale

[1] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

i.e. for a finite (albeit arbitrarily large) number of inputs, we can automatically verify the output. In other words we can only provide fixed value quantification ($\forall x \in \{v_1, \ldots, v_n\} | P(f(x))$), not universal quantification ($\forall x P(f(x))$). This distinction is discussed in detail in chapter 6. Without the ability to quantify types over values, a system is unable to automatically validate properties of arguments to functions (e.g. a function that asks for the Nth value of a sequence with potentially fewer than N values requires a sentinel value like Null). Similarly, we cannot be sure that the user has done the appropriate check at runtime [e.g. if (str.length $<$ N) handle_error();]. With a dependently typed language, we can build these checks directly into the type signature, either at the data type level or the function declaration level.

Once we have a system capable of fixed value and universal quantification, we can make statements using higher order connectives, e.g. $\forall s : String, P(s)$ for some predicate P. This is a step beyond conventional testing/correctness as it can be verified at the time of program construction/compilation. This leads to the main idea of this section: **make a good string library (such as Ruby or Haskell) formally rigorous**.

8.1.3 API

All mainstream string implementations share a similar core application programming interface (API), including general purpose languages, proof assistants, and dependently typed languages. This API includes several methods and predicates. These are available independently of the representation of the strings. The representation can have an effect on the performance (in terms of time and memory usage) of these methods and predicates.

We examined C [138], C++ [136], Ruby [137], Haskell [110], Coq [56], Isabelle [59], Agda [60] and Idris [61]. We found, among others, the following core methods:

- **equals**: decides if two strings are equal;
- **length**: calculate the number of characters in the string;
- **concatenate**: adds either a character or another string onto a string;
- **any**: returns true if any character satisfies the predicate;
- **all**: returns true if all characters satisfy the predicate;
- **match**: return true if the supplied regular expression matches the string;

- **split**: breaks a string into a list of strings according to a supplied rule;
- **nth**: returns the nth symbol in the string, if one exists.

We also found a core collection of predicates operating on the characters in the alphabets composing strings, namely:

- **upper**: returns true if the character is upper case;
- **lower**: returns true if the character is lower case;
- **digit**: returns true if the character is numeric;
- **space**: returns true if the character is a white-space character (including spaces, newlines, and tab characters).

8.1.4 Primitive Delegation

Since proof assistants are built atop general purpose languages (Coq is built in OCaml, Idris is built in Haskell, etc), it is sometimes the case that the proof assistants inherit their implementation language's string primitives. In Agda and Idris strings are delegated to their primitive types, proofs about them tend to use assertions (for example Idris uses the assertion "really_believe_me", which always satisfies the type checker). In Agda, we can see this in the standard library[2]. In Coq, a String is defined as a List of Ascii characters [52]. Similarly, Isabelle declares strings to be synonymous[3] with character lists as shown in figure 8.1. Idris acknowledges these problems, and provides

```
type_synonym string = ‘char list’
```

FIGURE 8.1: Isabelle Type Synonym for Strings

alternative representations, namely StrM (see section 8.1.5.1), and List Char.

Primitive delegation makes it difficult to reason about their state. Given this, we have chosen to model our strings manually, making them first class citizens.

[2] https://github.com/agda/agda-stdlib/blob/master/src/Data/String/Core.agda
[3] http://isabelle.in.tum.de/website-Isabelle2013/dist/library/HOL/String.html

8.1.5 Representation

As described in section 8.1.1, we have several basic and popular choices for representation of strings, all of which support the standard string API described in section 8.1.3. To provide a meaningful and grounded context for discussion, we will implement alternatives with and without negatives in Idris [61]. Due to the dependent type system in Idris, we can prove properties of strings, and verify them with the compiler. Likewise, we can encode conditions on arguments directly into function signatures and datatypes. We build our own version of strings in order to control the behavior of concatenation (which differs when negatives are permitted) and to allow parameterization over arbitrary alphabets (StrM is restricted to the ASCII character set type).

8.1.5.1 Cons Cells

The name "Cons Cells" stems from Lisp [135], and it derives from the operation "cons" used to *cons*truct a list, based on a node data structure called a "cell", which contains a place to hold data and a pointer to the next cell. Cons cells and lists are very similar, and have similar properties. Many implementations of linked lists (as opposed to array lists and others) are based on a nearly identical node/cell data structure. In Idris, such a pattern matching structure is stored internally as an abstract syntax tree (AST) representation. An AST representation maintains the order of operations, which is not required in this case because concatenation is associative (see chapter 3 for more details).

Figure 8.2 shows the built-in version of a cons cell representation. Note the delegation to the primitive (base-language) operations to perform the actual concatenation ("prim__strCon"). The data type and the function both rely on this delegation.

```
strCons : Char -> String -> String
strCons = prim__strCon

data StrM : String -> Type where
  StrNil : StrM ""
  StrCons : (x : Char) -> (xs : String) -> StrM (strCons x xs)
```

FIGURE 8.2: Idris StrM cons cell built-in definition.

Our version, called "Word" (shown in figure 8.3) does not rely on delegation to the primitive types, and permits parameterization over other alphabets (the built-in StrM

supports only the ASCII character datatype, Char). Furthermore, defining our own datatype allows us control of the behaviour of concatenation (which is necessary if "negative" characters are permitted). The full implementation with proofs and API methods is in appendix A.10.

```
data Word t = Empty | (#) t (Word t)
```

FIGURE 8.3: Our Idris Cons Cell definition parameterized.

An additional benefit to maintaining control of our implementation and to use a cons cell representation (rather than relying on a list-based formalism), is we can now implement a customized induction proof structure. The statement and proof are shown in figure 8.4.

```
wordInduction : (P : (Word t) -> Type)  -> -- Property to show
            (P Empty)                   -> -- Base case
            ((c : t) -> (w : (Word t)) ->  P w -> P (c # w)) -> -- Step
            ( a : (Word t)) ->          -- Show for all a
            P a
wordInduction P p_Empty p_Concat Empty   = p_Empty
wordInduction P p_Empty p_Concat (c # w) = p_Concat c w
                        (wordInduction P p_Empty p_Concat w)
```

FIGURE 8.4: Our Idris Cons Cell induction principle.

8.1.6 Printf/Scanf

Two very common mechanisms in procedural programming to handle string-based input and output are the functions "printf" and "scanf" (denoting "print formatted" and "scan formatted", respectively). These are used extensively in the C programming language [138]. The work presented here is an effort to apply dependent types to each of these. Fortunately, McKenna discussed the output half (the "printf" function) of these functions on his Youtube channel[4], with a code snippet related to it[5]. The Idris implementation he provides is similar structurally to that given in a paper by Augustsson on Cayenne (another dependently typed research language) [62].

The code style is an informal implementation of dependent type providers [139]. Essentially, this means that the code is structured such that return types of the core functions

[4] https://www.youtube.com/watch?v=fVBck2Zngjo
[5] https://gist.github.com/puffnfresh/11202637

(in this case "printf" and "scanf") are determined by function calls. Unlike the approach by Weitz et al [140] for Java, we can take full advantage of the dependent type system.

The full implementation, based on McKenna's work is in appendix A.11, and our contribution of a scanf function (technically "sscanf", a string scanning function that can be used to build the input-driven "scanf" version) is in figure 8.5.

Compared to the versions of "scanf" and "printf" employed in the C standard library, the dependently typed versions given here are much safer, and the type safety is verified by the compiler. Specifically, a call to the function in C as in figure 8.6 will compile (although fail at runtime due to a datatype mismatch), just as a correct call as in 8.7 will compile.

The Idris implementation can catch the data type error at compile time, as it is capable of predicating the types of the remaining arguments based on the contents of the format string.

8.2 Utilities

As discussed in section 6, the costs and effort required to build verified software are quite high. In order to reduce this burden for future development, we discuss a number of additional datatypes, mathematical utilities, and useful functions to extend the Idris system.

8.2.1 Rational

Idris supports Natural numbers (the positive integers) and Floating point numbers (double precision floating points, to represent the Reals in finite memory). However, Idris lacks a numerically exact rational (Integer quotient) datatype. For our purposes, we define a Rational as a pair consisting of the numerator and the denominator. Part of the source code is shown in figure 8.8, with the full source code available online[6].

[6] https://github.com/bgoodspeed/idris-misc

8.2.2 Monadic State Helpers

Several state-based monads were used while porting the source code from the excellent Haskell reference "The Haskell Road to Logic Maths and Programming" [123], and while developing some of the benchmark programs in section 8.3. The monad structure is well used in Idris as well, but the state monad was lacking a few utility methods: to run and evaluate state. Their source code is shown in figure 8.9, with the full source code available online[7].

8.2.3 NFA

Finite automata and regular expressions are an enormously popular tool in theoretical computer science [40, 141–143]. Idris does not have regular expression support as part of the standard library. Inspired by a Haskell implementation[8], we present an Idris regular expression matcher, based on the non-deterministic finite automata construction (NFA). Portions of the source code are shown in figure 8.10, and the full source is available online[9].

8.2.4 Power List

The power list is the list of all of sublists of a list. For example, the powerlist of [1,2,3] is [[1,2,3], [1,2], [1,3], [2,3], [1], [2], [3]]. This construct is useful for reasoning about relationships as mappings between sets. An Idris representation is shown in figure 8.11, and is available online[10].

8.2.5 Permutations

Permutations represent all re-arrangements of a list. For example, the permutations of [1,2,3] are [[1,2,3], [1,3,2], [2, 1,3], [2, 3,1], [3, 1,2], [3, 2,1]]. These are very useful for combinatorics research and string handling functions. An Idris representation is shown in figure 8.11, and is available online[11].

[7] https://github.com/bgoodspeed/idris-misc
[8] http://pbrisbin.com/posts/regular_expression_evaluation_via_finite_automata/
[9] https://github.com/bgoodspeed/idris-misc
[10] https://github.com/bgoodspeed/idris-misc
[11] https://github.com/bgoodspeed/idris-misc

8.2.6 Cyclic Shifts

Cyclic shifts are rotations of a given list. For example, the list of all cyclic shifts of [1,2,3] are [[1,2,3], [3,1,2], [2,3,1]]. These are useful for theoretical computer science, specifically in dealing with automata theory and formal languages [115]. An Idris representation is shown in figure 8.11, and is available online[12].

8.3 Benchmarks

Benchmarks are programs used to measure the performance of languages. An excellent benchmarking resource is the "Computer Language Benchmarks Game" [13]. The benchmark programs are solutions to specific problems with fixed inputs. Each language must implement the solution with the same algorithm, because they are not seeking to measure relative algorithm performance but language performance. The results are then compared according to the solution's runtime, memory usage and lines of code required. The code presented here is ported from either the Haskell or ML implementations available for the Benchmarks Game.

The benchmarks main repository[14] and the Idris versions[15] are both available to the public online. The results of the benchmarks are presented in chapter 10.

8.3.1 Fib

The Fibonacci sequence is obtained by taking the sum of the two prior elements in the sequence, beginning with [1,1, ...]. In mathematical terms this is $F(n) = F(n-1) + F(n-2)$. This function is useful for benchmarking a program with rapid stack growth. This is implemented as shown in figure 8.12.

[12] https://github.com/bgoodspeed/idris-misc
[13] http://benchmarksgame.alioth.debian.org/
[14] http://benchmarksgame.alioth.debian.org/
[15] https://github.com/bgoodspeed/idris-benchmarks

8.3.2 Ackermann

The Ackermann function[16] is a non-primitive recursive function which grows faster than exponential and factorial functions. This function is useful for benchmarking numerical computation of a fast-growing function. This is implemented as shown in figure 8.13.

8.3.3 Sieve

The sieve function produces the sequence of values that are all relatively co-prime to the rest of the list. That is to say, all multiples of previous elements are removed from the remainder of the list. This function is useful for benchmarking filtering of a large list in memory. The implementation is shown in figure 8.14. Note that this particular program makes use of the incomplete pattern matching feature of Idris (see "strictlast", which has undefined behavior in the event of an empty list).

8.3.4 Binary Trees

The binary trees program produces a series of balanced binary trees. This function is designed to benchmark memory usage and traversal of structures in memory. Key portions of the implementation are shown in figure 8.15, with the complete program available online[17].

8.4 Summary

In this section, we presented several useful building blocks for the Idris language. These components are reusable, and can be employed for a broad selection of problems. The reuse savings and benchmark results are discussed in chapter 10. The contributions dealing with the mathematical and proof-oriented portions of Idris are detailed in chapter 9.

[16] https://en.wikipedia.org/wiki/Ackermann_function
[17] https://github.com/bgoodspeed/idris-benchmarks

```
-- Printf code based on:
-- https://gist.github.com/puffnfresh/11202637
-- written by:
-- Brian McKenna, https://www.youtube.com/watch?v=fVBck2Zngjo
-- Scanf code by Ben Goodspeed

data SFormat = SFInt SFormat
             | SFOther Char SFormat
             | SFString SFormat
             | SFEnd

data Result : Type where
  MkIntResult : Int -> Result
  MkCharResult : Char -> Result
  MkStringResult : String -> Result

sformat : List Char -> SFormat
sformat ('%' :: 'd' :: cs) = SFInt (sformat cs)
sformat ('%' :: 's' :: cs) = SFString (sformat cs)
sformat (c :: cs) = SFOther c (sformat cs)
sformat [] = SFEnd

interpSFormat : SFormat -> Type
interpSFormat (SFInt f) =  Int -> interpSFormat f
interpSFormat (SFOther _ f) =  interpSFormat f
interpSFormat (SFString f) =  String -> interpSFormat f
interpSFormat SFEnd = List Result

sformatString : String -> SFormat
sformatString x = sformat (unpack x)

toSFunction : (fmt : SFormat) -> (List Result) -> interpSFormat fmt
toSFunction (SFInt f) acc = \i => toSFunction f ((MkIntResult i) :: acc)
toSFunction (SFOther c f) acc = toSFunction f ((MkCharResult c) :: acc)
toSFunction (SFString f) acc  = \s => toSFunction f
                                        ((MkStringResult s) :: acc)
toSFunction SFEnd  acc  = acc

sscanf : (s : String) -> interpSFormat (sformatString s)
sscanf s = toSFunction (sformatString s) []
```

FIGURE 8.5: Idris definition of dependently typed scanf.

```
char v;
sscanf("512", "%d", &v);
```

FIGURE 8.6: C version of an invalid sscanf call.

```c
int v;
sscanf("512", "%d", &v);
```

FIGURE 8.7: C version of a valid sscanf call.

```
module Rational

data Rational : Type where
  MkRational : Nat -> (d : Nat) -> GT d Z -> Rational

quotRem : Nat -> Nat -> (Nat, Nat)
quotRem n d = (div n d, mod n d)

numerator : Rational -> Nat
numerator (MkRational x d prf) = x


denominator : Rational -> Nat
denominator (MkRational x d prf) = d

recip : Rational -> Rational
recip (MkRational Z d prf) = MkRational Z (S Z) (LTESucc LTEZero)
recip (MkRational (S k) d prf) = MkRational d (S k) (LTESucc LTEZero)

rationalMult : Rational -> Rational -> Rational
rationalMult (MkRational x Z prf) (MkRational y Z w) = absurd prf
rationalMult (MkRational x Z prf) (MkRational y (S k) w) = absurd prf
rationalMult (MkRational x (S k) prf) (MkRational y Z w) = absurd w
rationalMult (MkRational x (S k) prf) (MkRational y (S l) w) =
                MkRational (x * y) ((S k) * (S l)) (LTESucc LTEZero)

rationalAdd : Rational -> Rational -> Rational
rationalAdd (MkRational x Z p1) (MkRational n Z p2) = absurd p1
rationalAdd (MkRational x Z p1) (MkRational n (S k) p2) = absurd p1
rationalAdd (MkRational x (S k) p1) (MkRational n Z p2) = absurd p2
rationalAdd (MkRational x (S k) p1) (MkRational n (S l) p2) =
  MkRational ((x * (S l)) + (n * (S k))) ((S k) * (S l)) (LTESucc LTEZero)
```

FIGURE 8.8: Idris definition of a Rational data type.

```
module StateUtils

import Control.Monad.Identity
import Control.Monad.State


runState : State s a -> s -> (a, s)
runState m = runIdentity . runStateT m


evalState : State s a -> s -> a
evalState m s = fst (runState m s)
```

FIGURE 8.9: Idris definition of state monad helper functions.

```
module RegexNFA

import Control.Monad.Identity
import Control.Monad.State
import StateUtils

SID : Type
SID = Int -- State Identifier

data Pattern
   = Empty                    -- ""
   | Literal Char             -- "a"
   | Concat Pattern Pattern   -- "ab"
   | Choose Pattern Pattern   -- "a|b"
   | Repeat Pattern           -- "a*"

record Rule : Type where
  MkRule : (fromState  : SID) -> (inputChar  : Maybe Char) ->
  (nextStates : List SID) -> Rule

record NFA : Type where
  MkNFA : (rules : List Rule) -> (currentStates : List SID) ->
  (acceptStates : List SID) -> NFA

...

accepted : NFA -> Bool
accepted nfa = any (\x => x `elem` (acceptStates nfa))
(currentStates nfa ++ freeStates nfa)

matches : String -> Pattern -> Bool
matches s = (\x => x `accepts` (unpack s)) . toNFA

main : IO ()
main = do
    -- This AST represents the pattern /ab|cd*/:
    let p = Choose
            (Concat (Literal 'a') (Literal 'b'))
            (Concat (Literal 'c') (Repeat (Literal 'd')))
    print $ "xyz" `matches` p
    -- => False
    print $ "cddd" `matches` p
    -- => True
```

FIGURE 8.10: Idris definition of a regular expression matcher.

```
module Utilities

cyclicShift : List a -> List a
cyclicShift [] = []
cyclicShift (x :: xs) = xs ++ [x]

allCyclicShiftsOf : List a -> List (List a)
allCyclicShiftsOf xs = iterateN (length xs) cyclicShift xs

permutations : (Eq a) => List a -> List (List a)
permutations [] = [[]]
permutations xs = [x::ys | x <- xs, ys <- permutations (delete x xs)]

powerList  : List a -> List (List a)
powerList  [] = [[]]
powerList  (x::xs) = (powerList xs) ++ (map (x::) (powerList xs))
```

FIGURE 8.11: Idris definition of various utilty functions.

```
-- £Id: fibo.ghc,v 1.5 2005-04-25 19:01:38 igouy-guest Exp £
-- http://www.bagley.org/~doug/shootout/
-- ported to Idris by Ben Goodspeed

import System

fibI : Int -> Int
fibI n = case n < 2 of
             True => 1
             _    => fibI (n-2) + fibI (n-1)

fibonacci : Int -> Int
fibonacci n = if n < 2 then 1 else fibonacci (n-1) + fibonacci (n-2)



main : IO ()
main = do args <- getArgs
          case args of
               [self] => putStrLn ("usage: " ++ self ++ " <n>")
               [_, n] => putStrLn (show (fibI (cast n)))
```

FIGURE 8.12: Idris definition of the fibonacci sequence.

```
-- http://shootout.alioth.debian.org/
-- shortened by Bryn Keller, Einar Karttunen and Don Stewart
-- Ported to Idris by Ben Goodspeed
module Main

import System

ack : Int -> Int -> Int
ack  0 n = n+1
ack  m n = ack (m-1) $ if n == 0 then 1 else ack m (n-1)

main : IO()
main = do
    args <- getArgs
    case args of
      [self ] => putStrLn ("usage: " ++ self ++ " <n>")
      [_, ns] => putStrLn ("Ack(3," ++ ns ++ "): " ++
                     show (ack 3 (cast ns)))
```

FIGURE 8.13: Idris definition of the Ackermann function.

```
-- £Id: sieve.ghc,v 1.2 2004-11-23 08:08:45 bfulgham Exp £
-- http://www.bagley.org/~doug/shootout/
-- from Roland Dowdeswell
-- adjusted by Aaron Denney, borrowing strictness attempt
-- from Malcom Wallace's matrix multiplication
-- ported to Idris by Ben Goodspeed

module Main

import System

force : List a -> Bool
force [] = True
force (x::xs) = force xs

strictlast : List (List a) -> List a
strictlast [x] = x
strictlast (x::xs) = let f =  force x in  strictlast xs

sieve : List Int -> List Int
sieve [] = []
sieve (h::t) = h :: sieve [x| x<-t, (x `mod` h) /= 0]

mytest : Int -> Int
mytest n = cast $ length $ strictlast $ map sieve $ replicate
              (cast n) ([2..8192])

main : IO ()
main = do
  args <- getArgs
  case args of
       [self] => putStrLn ("usage: " ++ self ++ " <n>")
       [_, n] => putStrLn . ("Count: "++) . show . mytest $ cast n
```

FIGURE 8.14: Idris definition of a numeric sieve function.

```
-- Contributed by Don Stewart in Haskell
-- Ported to Idris by Ben Goodspeed
import System
import Data.Bits

data Tree = Nil | Node Int Tree Tree

-- ...
make : Int -> Int -> Tree
make i 0 = Node i Nil Nil
make i d = let i2 = 2*i in
               let d2 = d - 1 in
                  Node i (make (i2-1) d2) (make i2 d2)

-- ...

createTrees : Int -> IO ()
createTrees n = let maxN     = max (minN + 2) n
  stretchN = maxN + 1
  c        = check (make 0 stretchN)
  vs       = depth minN maxN
  long   = make 0 maxN in
    do
      io "stretch tree" stretchN c
      mapM_ (\((m,d,i)) => io (show m ++ "\t trees") d i) vs
      io "long lived tree" maxN (check long)

-- ...
```

FIGURE 8.15: Idris definition of binary tree creation and traversal.

Chapter 9

Proofs and Tactics

Formal verification is difficult and time consuming. This is because software systems tend to exhibit "complex" behaviors. Here, we borrow Simon's definition of complexity, "given the properties of the parts and the laws of their interaction, it is not a trivial matter to infer the properties of the whole" [144]. While constructing formally verified software itself is often not substantially more costly than producing the same software with a mainstream language, the formal verification can certainly be very expensive and protracted.

We know the structure of code has changed as a result of TDD, and we have seen the changes that PDD has on code. However, small changes to the structure of proven pieces of software are likely to break a proof of a property of that code (because they are often structural proofs). Because of these challenges, similar care to the proof side of things must be given when developing libraries for re-use as the code itself. In addition to the traditional libraries of building blocks described in chapter 8, we also provide building blocks for proofs and proof-tactics (named as such for the LTac proof tactic library, [57]) as employed by proof search system within Coq [15, 52, 56] and Idris [61, 89, 139].

In this section, we show a list of lemmas and mathematical structures we've modeled for use as building blocks in Idris.

9.1 Lists

Lists are well-supported in Idris, but we add several quantifiers and procedures. In Idris a list is a linked list, with a cons cell making up each node. Each cons cell holds a value

and a reference to the remainder of the list. The only constant is the empty list, denoted "Nil" or "[]".

As in LISP [135], lists are used in place of arrays, whose properties (such as numeric indices) can be emulated. Similarly, they form the building blocks of other data types such as Strings (lists of Characters). Details about this specific type are available in section 8.1.

9.1.1 Quantifiers

The quantifiers presented in figure 9.1 (such as 'Any', 'All', etc) are based on similar quantifiers available for Vectors[1] (which are conceptually a List type parameterized by a Natural number representing the number of elements). They define proof types for 'Any', which is satisfied if a predicate holds for at least one element in the list. Likewise the 'All' type is satisfied if the predicate holds for every element in the list. Since the list itself is defined by the head and the tail (the first element and the "rest"), both proofs of 'Any' and 'All are case-based, called 'Here', meaning the Head is valid at least, and 'There', meaning the Tail is valid at least.

To clarify these quantifiers, the function "isUpper" (which has the type Character → Boolean) is a predicate, as it returns a boolean value depending on whether the character is upper case. When applied to the list ['A', 'b'], the 'All' quantifier could not be satisfied, but the 'Any' quantifier could be satisfied using the fact that the head of the list contains an upper case character.

This is implemented as shown in part in figure 9.1. The full implementation is available online[2].

9.1.2 Decision Procedures

We define several decision procedures for Lists (along with types associated with the decision, denoted with the suffix "T"). In particular, we define predicates based on *prefixes* (which are true if the first part of the list matches the supplied value), *suffixes* (which hold when the end of the list matches the supplied value); and, *palindromes* (which are lists which remain identical after reversing the order of the elements). The

[1] https://github.com/idris-lang/Idris-dev/blob/master/libs/base/Data/Vect/Quantifiers.idr
[2] https://github.com/bgoodspeed/idris-misc

two versions (those types ending with a "T"), and the decidability data type are given because of the tactics available in Idris [3] Conceptually they model the same relationships, so the choice of which to use is at the discretion of the user. This is implemented as shown in part in figure 9.2. The full implementation is available online[4].

9.2 Boolean Values and Lazy Evaluation

Booleans, logical connectives and related control structures are supported by Idris. However, Idris is an eagerly evaluated language. Some of these features require a lazier evaulation scheme.

In order to provide the expected semantics for logical connectives, in particular "AND" (denoted "&&") and "OR" (denoted "||"), the right hand side is forced to be lazily evaluated (using the *Delay* directive). This allows "short-circuiting" of logical expressions. In other words, for the expressions " True || X" and " False && X", the expression "X" does not need to be evaluated. The same laziness is necessary for "if ... then ... else ..." expressions. It is an error to evaluate the *else* branch when the condition is true, and an error to evaluate the *then* branch when the condition is false.

Both of these are modeled in Idris using the *Delay* monad for lazy evaluation. This makes proofs less straightforward, as the logical connective is not the expected type. It would be reasonable to expect "A && B" to have the type signature " Bool → Bool → Bool". The actual type signature is " Bool → Lazy Bool → Bool", which means logical proof steps can fail.

Fortunately these difficulties can be resolved with the appropriate lemmas. For example, the expression "$\forall b \in$ Bool, b && Lazy False = False" holds. Specifically, this shows that if we know the sub-expression will eventually evaluate to False (Lazy False), we can conclude the whole expression is False. This relationship can be proven in Idris as shown in figure 9.3, encoded first as direct proof in the type "boolAndLazyFalseIsFalse", and later interactively in the type "boolAndLazyFalseIsFalse2".

We have implemented these lemmas in Idris and made them available freely online[5].

[3] For example, searching with ":apropos Dec" at the Idris prompt will show tactics usable with decision types.

[4] https://github.com/bgoodspeed/idris-misc

[5] https://github.com/bgoodspeed/idris-misc

9.3 Mathematical Structures

Isomorphisms are supported in the Idris standard library[6]. We added the notion of a semigroup and monoid homomorphism (see figure 9.4).

Recall that a homomorphism, h from $(S, *)$ to $(T, +)$, is a mapping such that $h(a * b) = h(a) + h(b) \forall a, b \in S$. In addition to datatypes representing homormorphisms between semigroups and monoids, the reflexivity (for a mapping this refers to the identity function) and transitivity (for mappings this refers to composition) of homomorphisms is proven, as shown in figure 9.5. These proofs use the technique of filling in "holes" (called "?prf1" and "?prf2" in this case), and equational reasoning, as shown by the "={ ... }= ... QED" syntax.

To provide clarity and consistency between "paper" proofs and Idris proofs, we will illustrate the proof of transitivity of semigroup homomorphisms in both formats.

Theorem 9.1. *If A, B, C are semigroups, and $h : A \to B$, $h' : B \to C$ are semigroup homomorphisms, then $h \circ h' : A \to C$ is also a semigroup homomorphism.*

Proof:

Let $a_1, a_2 \in A$, and let $*_A, *_B$ and $*_C$ denote the semigroup operations for A, B and C, respectively.

$h\prime(h(a_1 *_A a_2))$

$= h\prime(h(a_1) *_B h(a_2))$ *as h preserves $*_A$ in $*_B$ (in Idris, preservesGroup, figure 9.5)*

$= h\prime(h(a_1)) *_C h\prime(h(a_2))$ *as $h\prime$ preserves $*_B$ in $*_C$ (in Idris, preservesGroup', 9.5)*

$= (h \circ h\prime)(a_1) *_C (h \circ h\prime)(a_2)$ *by definition of composition.*

$\therefore (h \circ h\prime) : A \to B$ *preserves $*_A$ in $*_C$, and is transitive.*

As shown in the paper proof above, the transitivity proof of the composition relies on the fact that the mapping is in fact a homomorphism. This proof must be given when the "Hom" datatype is created in Idris (see figure 9.4), and is shown in the argument named "preservesGroup". A very similar argument holds (and thus similar datatypes exist) for monoid homomorphisms, which additionally require the correct handling of the identity element.

[6] https://github.com/idris-lang/Idris-dev/blob/master/libs/base/Control/Isomorphism.idr

The Idris code for a semigroup homomorphism datatype is shown in figure 9.4, and is available along with the related monoid homomorphism definitions and proofs online[7]. Isomorphisms are part of the standard library, and are declared as shown in figure 9.6.

9.4 Summary

In this section we presented several mathematical structures, encoded in the Idris type system. These structures can serve as the basis for constructing more specific proofs and arguments for the correctness of future programs. In traditional software development, reusable libraries of functionality are crucial to the development of programs in a timely fashion. This need is just as strong in dependently typed systems. In fact this requirement is broader, as the entire category of proof statements (including tactics and types) requires support. There is no need and thus no support for such structures in traditional system and library development. The results of this work, including cost savings and re-use potential are discussed in section 10.

[7] https://github.com/bgoodspeed/idris-misc

```
data Any : (P : a -> Type)-> List a -> Type where
  Here  : {P : a -> Type} -> {xs : List a} -> P x -> Any P (x :: xs)
  There : {P : a -> Type} -> {xs : List a} -> Any P xs -> Any P (x :: xs)

anyNilAbsurd : {P : a -> Type} -> Any P Nil -> Void
anyNilAbsurd (Here _) impossible
anyNilAbsurd (There _) impossible

anyElim : {xs : List a} -> {P : a -> Type} -> (Any P xs -> b) ->
          (P x -> b) -> Any P (x :: xs) -> b
anyElim _ g (Here p)  = g p
anyElim f _ (There p) = f p

any : {P : a -> Type} -> (dec : (x : a) -> Dec (P x)) ->
                         (xs : List a) -> Dec (Any P xs)
any _ [] = No anyNilAbsurd
any p (x :: xs) with (p x)
  | Yes prf = Yes (Here prf)
  | No prf  = case any p xs of
                  Yes prf' => Yes (There prf')
                  No  prf' => No  (anyElim prf' prf)

data All : (P : a -> Type) -> List a -> Type where
  Nil : {P : a -> Type} -> All P Nil
  (::) : {P : a -> Type} -> {xs : List a} -> P x ->
         All P xs -> All P (x :: xs)

notAllHere : {P : a -> Type} -> {xs : List a} -> Not (P x) ->
             All P (x :: xs) -> Void
notAllHere _ Nil impossible
notAllHere np (p :: _) = np p

notAllThere : {P : a -> Type} -> {xs : List a} -> Not (All P xs) ->
              All P (x :: xs) -> Void
notAllThere _ Nil impossible
notAllThere np (_ :: ps) = np ps

all : {P : a -> Type} -> (dec : (x : a) -> Dec (P x)) ->
                         (xs : List a) -> Dec (All P xs)
all _ Nil = Yes Nil
all d (x :: xs) with (d x)
  | No prf = No (notAllHere prf)
  | Yes prf = case all d xs of
                  Yes prf' => Yes (prf :: prf')
                  No  prf' => No (notAllThere prf')
```

FIGURE 9.1: Idris definition of various List Quantifiers.

```
module ListDecisions

import Decidable.Equality
import DecHelper

isPrefixOfT : (Eq a) => List a -> List a -> Type
isPrefixOfT xs ys = isPrefixOf xs ys = True

isPrefixOfDec : (Eq a) => (xs : List a) -> (ys : List a) ->
                Dec (isPrefixOfT xs ys)
isPrefixOfDec xs ys with (isPrefixOf xs ys)
  | True  = Yes Refl
  | False = No  falseNotTrue


isSuffixOfT : (Eq a) => List a -> List a -> Type
isSuffixOfT xs ys = isSuffixOf xs ys = True

isSuffixOfDec : (Eq a) => (xs : List a) -> (ys : List a) ->
                Dec (isSuffixOfT xs ys)
isSuffixOfDec xs ys with (isSuffixOf xs ys)
  | True  = Yes Refl
  | False = No  falseNotTrue

isPalindrome : (Eq a) => List a -> Bool
isPalindrome xs = (reverse xs) == xs

isPalindromeT : (Eq a) => List a -> Type
isPalindromeT xs = isPalindrome xs = True

isPalindromeDec : (Eq a) => (xs : List a) -> Dec (isPalindromeT xs)
isPalindromeDec xs with (isPalindrome xs)
  | True  = Yes Refl
  | False = No falseNotTrue
```

FIGURE 9.2: Idris definition of various List Decision Predicates.

```
boolAndLazyFalseIsFalse : (b : Bool) -> b && False = False
boolAndLazyFalseIsFalse b = case b of
                                 True => Refl
                                 False => Refl

boolAndLazyFalseIsFalse2 : (b : Bool) -> b && False = False
boolAndLazyFalseIsFalse2 b = ?boolAndLazyFalseIsFalse2Proof



---------- Proofs ----------

Main.boolAndLazyFalseIsFalse2Proof = proof
  intros
  case b
  trivial
  trivial
```

FIGURE 9.3: Idris lemma for dispatching lazy evaluation of booleans.

```
data Hom : (a, b : Type) -> Semigroup a -> Semigroup b -> Type where
    MkHom : (actxt : Semigroup a, bctxt : Semigroup b) => (h : a -> b) ->
            (preservesGroup : (a1 : a) -> (a2 : a) ->
              h ((<+>) @{actxt} a1 a2) = (<+>) @{bctxt} (h a1) (h a2)) ->
              Hom a b actxt bctxt
```

FIGURE 9.4: Idris definition of a homomorphism.

```
homRefl : (as : Semigroup a) => Hom a a as as
homRefl = MkHom id (\x,y => Refl)

homTrans : (adict : Semigroup a, bdict : Semigroup b,
            cdict : Semigroup c) =>
            Hom a b adict bdict -> Hom b c bdict cdict ->
            Hom a c adict cdict
homTrans @{adict} @{bdict} @{cdict} (MkHom h preservesGroup)
  (MkHom h' preservesGroup') =
  MkHom @{adict} @{cdict} (\x => h' (h x))
            (\something, another =>
               (h' (h (something <+> another))) ={ ?prf1 }=
               (h' (h something <+> h another)) ={ ?prf2 }=
               (h' (h something) <+> h' (h another)) QED)



Homomorphism.prf2 = proof
  intros
  rewrite preservesGroup' (h something) (h another)
  trivial


Homomorphism.prf1 = proof
  intros
  rewrite preservesGroup something another
  trivial
```

FIGURE 9.5: Idris proofs of Homomorphism properties.

```
data Iso : Type -> Type -> Type where
    MkIso : (to : a -> b) ->
            (from : b -> a) ->
            (toFrom : (y : b) -> to (from y) = y) ->
            (fromTo : (x : a) -> from (to x) = x) ->
            Iso a b
```

FIGURE 9.6: Idris definition of an isomorphism.

Chapter 10

Code Savings and Benchmark Results

After identifying the five proof assistants, we evaluated them according to the features required in table 5.6 in section 5. The selected proof assistant, Idris, was then used in the development of several programs and proofs. A total of 25 programs were developed. We selected two secure programs, three benchmark programs, 20 tutorial programs. These comprise approximately 11,000 lines of code, of which we were able to extract about 1200 lines (roughly 350 lines of proofs and proof tactics, and 850 lines of utility functions and types), and reuse them, saving about 2400 lines. The numerical breakdown of these savings is shown in table 10.1. Concrete examples of the software before and after are shown in figures 10.1 and 10.2 respectively.

These libraries were developed using a mixture of the "common" workflow and the newly proposed "proof-driven development" workflow. This division was to provide a context in which to compare the two workflows. The differences in quality and time-to-completion were discussed in chapter 6.

We find that about 1200 lines of functions, types and proofs were able to be reused, leading to a savings of 2400 lines in about 11,000 lines of code. In table 10.2 we see that certain programs are more amenable to reuse, and that frequently proof-oriented code is clustered together. This table shows the number of lines of code, lines of proof, and the number of lines saved (this quantity includes both lines of code and proof saved, if applicable) by importing the functionality from the reusable libraries that we created. This segregation of verification and implementation code is typical of code produced

```
data Rational : Type where
    MkRational : Nat -> (d : Nat) -> GT d Z -> Rational
-- Definitions for: rational multiplication
--                  rational additional
--                  reciprocal
-- ...
-- elided

iterateN : Nat -> (f : a -> a) -> (x : a) -> List a
iterateN Z     f x = []
iterateN (S n) f x = x :: iterateN n f (f x)


mechanicsRule : Rational -> Rational -> Rational
mechanicsRule p x = rationalMult (rationalAdd
                         x (rationalMult p (recip x))) (1,2)

mechanics : Rational -> Rational -> Stream Rational
mechanics p x = iterateN 100 (mechanicsRule p) x
```

FIGURE 10.1: Idris Code Before

```
import Rational
import Utilities

mechanicsRule : Rational -> Rational -> Rational
mechanicsRule p x = rationalMult (rationalAdd
                         x (rationalMult p (recip x))) (1,2)

mechanics : Rational -> Rational -> Stream Rational
mechanics p x = iterateN 100 (mechanicsRule p) x
```

FIGURE 10.2: Idris Code After

under test driven development as well: for example, the default configuration for Java unit testing is for the unit test source code to exist in a separate directory which is then excluded from release builds[1].

[1] http://junit.org

Utility	Lines of Code	Times Used	Lines Saved
CharDec	94	2	188
DecHelper	4	9	36
GS	174	2	348
Homomorphism	68	2	136
LTE	44	1	44
ListCharDec	69	1	69
ListDecisions	35	1	35
ListQuantifiers	53	3	159
Polynomials	76	3	228
PowerSeries	28	1	28
Rational	57	1	57
RegexNFA	175	1	175
STAL	186	2	372
SetEq	75	1	75
StateUtils	14	1	14
Utilities	75	6	450
Total			2414

Table 10.1: Code reuse

Program	Lines of Code	Lines of Proof	Lines Saved
ackerman	20	0	0
binarytree	94	0	0
fibo	23	0	0
hello	5	0	0
sieve	41	0	0
CharDec	94	94	4
ConsCellWord	251	67	94
ListCharDec	69	69	147
ListDecisions	53	53	79
ListQuantifiers	53	53	4
WordCombinatorics	156	44	74

Table 10.2: Code savings

10.1 Benchmark Results

Here we present a comparison of Idris code with a similar but less proof oriented language, Haskell. The details of the algorithms described in section 8 are based on benchmarks from the excellent Computer Language Benchmarks Game[2].

In order to provide an up-to-date comparison, the Haskell (7.10.1) and Idris (0.9.16) programs were run on the same MacBook Air, using the Unix "time" utility with the

[2]http://benchmarksgame.alioth.debian.org/

results shown below (averaged over 5 runs). Both programs were compiled with default compiler settings and no optimizations, with Haskell using "ghc PROG.hs -o PROG", and Idris using "idris PROG.idr -o PROG".

The functions include: *hello world* (which benchmarks initialization and simple IO), *ackermann* (which benchmarks a very fast growing mathematical function), *fibonacci* (which benchmarks the performance as the stack grows during recursion), *binary trees* (which benchmarks garbage collection) and *sieve* (which benchmarks a large list filtering in memory). The results are summarized in table 10.3.

Program	Argument	Haskell Runtime	Idris Runtime
hello world	-	0.007s	0.007s
ackermann	10	2.595s	0.888s
ackermann	8	0.141s	0.061s
binarytrees	20	2.083s	9.057s
binarytrees	14	0.038s	0.019s
fibonacci	35	1.873s	4.287s
fibonacci	28	0.049s	0.113s
sieve	13	0.489s	17.351s
sieve	10	0.383s	13.563s

TABLE 10.3: Benchmark Results

As with all benchmarks, these results should be not interpreted as a guarantee of future performance. The early results suggest for programs using a lot of heap memory or stack space, Haskell still runs substantially faster. However, for raw numeric computations Haskell and Idris are comparable in performance.

10.2 Contributions Revisited

As stated in section 4, we sought to make contributions in five key areas. We address these contributions as follows:

- an **evaluation** of potential systems to support the goals stated, the details of this evaluation of Coq, Idris, Agda, Haskell and Isabelle along with the rationale for selecting **Idris** are in section 5;
- a **security-focused program**, along with proofs of security properties, described in section 4.5 and are available online[3], including both Idris and Coq sources;

[3]https://github.com/bgoodspeed/idris-secure

- a **new workflow**, "proof-driven development", a superset of test-driven development, described in detail in section 6;
- a number of **new data types and libraries** of reusable functions and proofs. The sources for string handling functions[4] and general utility functions and types[5] are both available online;
- a collection of **benchmark programs** are available online[6] and the runtime results are in section 10.1.

[4]https://github.com/bgoodspeed/idris-strings
[5]https://github.com/bgoodspeed/idris-misc
[6]https://github.com/bgoodspeed/idris-misc

Chapter 11

Related Work

As this work spans a variety of areas, there is a great deal of related work. In this chapter, we look at this work from the point of view of the PDD workflow elaborated in this thesis. The key areas we will revisit are: formalism in security, cryptographic protocol verification, secure software development, dependently typed programming & proof assistants for software development and test-driven development & software development workflows.

Formalism in computer security has a long history by the standards of computer science. The seminal work on security modeling by Bell and LaPadula from 1973 [28] can be seen as the most important early work in the area. This formed the basis of many other modeling efforts [29, 30, 33, 41–43, 45, 102, 113]. These models found their way into specifications and guidelines for secure systems construction, including the Department of Defense "Rainbow Series" [26, 34–38], and more recent versions like the Common Criteria [51] and Posix guidelines [48]. These models are generally mathematical and descriptive, and while groundbreaking and exquisitely detailed, they are not machine verifiable, and thus suffer from the issues identified in the workflow described in section 1 and figure 4.1.

The same issues (due to gaps in the workflow) have been addressed by various authors in different ways. In particular the cryptographic community has tended towards process calculi for verification of protocols [2, 6–9, 12, 13]. In some cases, these protocol descriptions have been machine verified (and thus regression issues can be avoided). However, despite this success, there is still the problematic gap of the implemented system not being the artifact about which the proofs argue.

The issues surrounding implementation of the models described by the formalisms have also been studied, often with restricted subsets of existing general purpose languages [85, 101]. There have been several successful projects using this technique, perhaps most notable the SEL4 verified microkernel [50]. However, the formalized semantics of these sub-languages are not as expressive as proof assistants [15, 55, 59, 93]. Recent efforts have been made to use these systems as the basis for further development (as we have done) [13, 14, 84]. These authors were able to close gaps in the secure software development process, but still followed a more common specification, design, implement, prove workflow.

New studies into dependently typed programming [54, 89, 91, 111, 122, 139, 145] have shown that dependently typed programs (the basis of several proof assistants, such as Coq [56], Agda [60] and Idris [61]) have levels of power beyond what any prior system can exhibit [122]. The issues with these systems are that the best practices for programming in them are unknown, and the costs for developing software as a result is very high (both due to limited available expertise and time required). This work seeks to help ameliorate some of the learning curve associated with transitioning into this paradigm of programming. We do this by introducing a new but familiar workflow, PDD, by providing many utilities and proof types.

The study of workflow in software engineering has grown substantially in recent memory. From early descriptions of Waterfall [73], to newer Agile processes [17, 69], many changes have been implemented resulting in higher quality code [16, 68, 74]. This work is also not the first to attempt to bridge the gap between "efficiency" oriented processes and "security" oriented ones [67]. As with the cryptographic studies, these have often focused on protocols [6, 8] and message passing semantics and traces [6, 53, 86, 93, 106]. Our workflow combining the power of proof assistants, dependent types and test-driven development represents a new way of combining a number of the most effective ideas of these authors.

Chapter 12

Conclusion

We showed that our new workflow, PDD, is a strict superset of TDD, which itself had desirable properties in the software development lifecycle. We provided a number of new data types and utility functions to help facilitate the use of a particular dependently typed language (Idris), and discussions of where difficulties lie. We then analyzed the available dependently typed systems, and justified our choice of Idris for this research. We have used the lessons of other recent developments with related languages and for related goals to avoid re-inventing the wheel.

Though we were able to demonstrate proofs that some classes of errors cannot occur, we have not been able to show that *unexpected* classes of error are prevented. We have shown only the ability to be more robust in our handling and verification of *expected* error classes. This is a fundamental and unresolved challenge in security.

While several steps have been taken, some important issues of difficulty and utility of these systems remain unsolved before widespread industrial use is likely. We discuss several avenues of research work below in the hopes that some of these can be addressed in the future.

12.1 Future Work

This is a young discipline, and the paradigm of dependently typed programming is not well understood. The implications of design decisions are not clear, nor are best practices. Much experimentation will be required before we can tell the long term ramifications of our type designs.

12.1.1 Error Messages

As with TDD, the quality of error messages determines the utility of the workflow - if we want the process to guide us towards correct code, we need to know where we have gone wrong. With the availability of proof search, and the compiler's ability to query for the type required to fill a hole in a program, it should be possible to generate suggestions within the system library. This would give: a) the exact signature required of a method to complete the program; and, b) and known methods that have the required signature. A more flexible search to return results that might be a likely interim result/datatype would be useful as well.

When proofs fail to unify (as when assertions fail to pass), it would be extremely helpful to have a return value saying why and exactly where it failed. In particular, Idris does not inform the user during interactive proofs that a rewrite has failed. The compiler silently fails to change the target expression.

12.1.2 Proof Targeting

It would be useful to support pre-order reasoning (as in equational reasoning in mathematical proof), so that intermediate transformations can happen. This would more closely mimic the way mathematics are done without computer assistance, and would therefore be a more familiar and gentler introduction to mathematical verification.

12.2 Summary

The new workflow described in this thesis, combined with solutions to the issues posed in the previous section, can help to bring rigor to a discipline in need of security. Fully utilizing the newest technology and tools will allow us further trust our ever-present software and hardware.

Bibliography

[1] Ponemon Institute. 2013 cost of data breach study: Global analysis. *Symantec Whitepapers*, 2013. URL https://www4.symantec.com/mktginfo/whitepaper/053013_GL_NA_WP_Ponemon-2013-Cost-of-a-Data-Breach-Report_daiNA_cta72382.pdf.

[2] S. Goldwasser. The search for provably secure cryptosystems. *Proceeding, Symposium of Applied Math, Cryptology and Computational Number Theory*, 42:89–113, 1990.

[3] D. Dolev and A. Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 2(29):198, 1983. URL http://www.cs.huji.ac.il/~dolev/pubs/dolev-yao-ieee-01056650.pdf.

[4] Claude Shannon. Communication theory of secrecy systems. *Bell Systems Technical Journal*, 1949. URL http://en.wikipedia.org/wiki/One-time_pad.

[5] Ueli Maurer. Conditionally-perfect secrecy and a provably secure randomized cipher. *Journal of Cryptology*, 5(1):53–66, 1992.

[6] L. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.

[7] Martin Abadi and Andrew Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148:36–47, 1999.

[8] Bruno Blanchet. An efficient cryptographic protocol verifier based on prolog rules. *IEEE Computer Security Foundations Workshop*, page 82, 2001. URL http://prosecco.gforge.inria.fr/personal/bblanche/publications/BlanchetCSFW01.ps.gz.

[9] Anupam Datta, Ante Derek, John Mitchell, and Dusko Pavlovic. A derivation system and compositional logic for security protocols. *IOS Press, Journal of*

Computer Security, page 423, 2005. URL http://seclab.stanford.edu/pcl/papers/ddmp-jcs05.pdf.

[10] S. Goldwasser. One-time programs. *CRYPTO LNCS 5157*, pages 39–56, 2008. URL CRYPTO2008,LNCS5157,pp.3956,2008.

[11] NIST. Fips pub 180-4 secure hash standard. *NIST*, 2012. URL http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf.

[12] J. Almeida. Certified computer-aided cryptography: efficient provably secure machine code from high-level implementations. *Proceedings of the 2013 ACM SIGSAC conference on Computer and communications security*, page 1217, 2013. URL https://eprint.iacr.org/2013/316.pdf.

[13] A. Appel. Verification of a cryptographic primitive: Sha-256. *ACM Transactions on Programming Languages and Systems*, 2015. URL http://www.cs.princeton.edu/~appel/papers.

[14] Daniel Ricketts. Automating formal proofs for reactive systems. *PLDI 14*, 2014. URL https://www.youtube.com/watch?v=phwVo66aChc.

[15] Benjamin Pierce. Software foundations. *Course notes, online at http://www.cis.upenn.edu/ bcpierce/sf*, 2013. URL http://www.cis.upenn.edu/~bcpierce/sf/.

[16] Robert Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice-Hall, 2008.

[17] Ken Schwaber and Mike Beedle. *Agile Software Development with SCRUM*. Prentice Hall, 1st edition, 2001.

[18] Danijel Radjenovic. Software fault prediction metrics: A systematic literature review. *Information and Software Technology*, 55(8):1397, 2013.

[19] P. Hofner, R. Khedri, and B. Moller. An algebra of product families. *Software and Systems Modeling*, 10(2):161, 2011.

[20] Charles Haley. A framework for security requirements engineering. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 31(1):133–153, 2008. URL http://charles.the-haleys.org/papers/Haley-SESS06-p35.pdf.

[21] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. *In Proc. USENIX Security*, pages 445–458, 2012. URL https://www.youtube.com/watch?v=mCIog3FaGco.

[22] Thomas Wadlow. Who must you trust? *ACM Queue*, 12(5), 2014. URL http://queue.acm.org/detail.cfm?id=2630691.

[23] K. Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8), 1984. URL http://cm.bell-labs.com/who/ken/trust.html.

[24] Peter Loscocco and Stephen Smalley. The inevitability of failure: The flawed assumption of security in modern computing environments. In *Usenix*, 2001. URL http://www.nsa.gov/research/_files/publications/inevitability.pdf.

[25] Carl Landwehr. A taxonomy of computer program security flaws, with examples. *ACM Computing Surveys*, 26(3), 1994.

[26] Department of Defense. *Trusted Computer System Evaluation Criteria*. DoD 5200.28-STD, CSC-STD-001-83 [Orange Book], 1985. URL http://www.fas.org/irp/nsa/rainbow/std001.htm.

[27] Perry Metzger. The case for formal verification, 'gmane.comp.encryption.general' forum posting, 2013.

[28] D. Bell and L. LaPadula. Secure computer systems: Mathematical foundations. *MITRE Technical Report*, I(2547), 1973. URL http://www.albany.edu/acc/courses/ia/classics/belllapadula1.pdf.

[29] Kenneth Biba. Integrity considerations for secure computer systems. *MITRE Technical Report*, (3153), 1975. URL http://seclab.cs.ucdavis.edu/projects/history/papers/biba75.pdf.

[30] Dorothy Denning and Peter Denning. *Data Security*. Addison-Wesley, 1982. URL http://www.gtnoise.net/papers/library/denning-data.pdf.

[31] J. McLean. Reasoning about security models. *Proc. IEEE Symp. Security and Privacy*, pages 123–131, 1987.

[32] J. McLean. The specification and modeling of computer security. *IEEE Computer*, 23, 1990.

[33] D. Bell. Concerning modelling of computer security. *Proc. IEEE Symp. Security and Privacy*, pages 8–13, 1988.

[34] Department of Defense. *Trusted Network Interpretation of the Trusted Computer System Evaluation Criteria*. DoD 5200.28-STD, 1987, NCSC-TG-005 [Red book], 1987. URL http://www.fas.org/irp/nsa/rainbow/tg005.htm.

[35] Department of Defense. *A Guide to Understanding Identification and Authentication in Trusted Systems*. DoD NCSC-TG-017 [Light Blue Book], 1991. URL http://www.fas.org/irp/nsa/rainbow/tg017.htm.

[36] Department of Defense. *Trusted Database Management System Interpretation of the Trusted Computer System Evaluation Criteria*. DoD 5200.28-STD, 1991, NCSC-TG-021 [Lavender/Purple book], 1991. URL http://www.fas.org/irp/nsa/rainbow/tg021.htm.

[37] Department of Defense. *A Guide to Understanding Security Modeling in Trusted Systems*. DoD NCSC-TG-010 [Teal book], 1992. URL http://www.fas.org/irp/nsa/rainbow/tg010.pdf.

[38] Department of Defense. *A Guide to Understanding Covert Channel Analysis of Trusted Systems*. DoD NCSC-TG-030 [Light Pink Book], 1993. URL http://www.fas.org/irp/nsa/rainbow/tg030.htm.

[39] W. Thomas. Automata on infinite objects. *Handbook of Theoretical Computer Science*, B, 1990.

[40] Michael Sipser. *Theory of Computation*. CEngage Learning, India, 2007.

[41] David Clark and David Wilson. A comparison of commercial and military computer security policies. *IEEE*, 1989. URL http://theory.stanford.edu/~ninghui/courses/Fall03/papers/clark_wilson.pdf.

[42] David Brewer and Michael Nash. The chinese wall security policy. *Proceedings of IEEE Symposium on Security and Privacy*, page 206, 1989. URL http://www.cs.purdue.edu/homes/ninghui/readings/AccessControl/brewer_nash_89.pdf.

[43] G. Benson, I. Akyildiz, and W. Appelbe. A formal protection model of security in centralized, parallel, and distributed systems. *ACM Transactions on Computing Systems*, 8(3), 1990.

[44] Peter Neumann. *A Provably secure operating system : the system, its applications, and proofs*. SRI International, CSL-116, 2nd edition, 1980.

[45] Michael Harrison, Walter Ruzzo, and Jeffrey Ullman. Protection in operating systems. *ACM*, 19(8), 1976.

[46] Theodore Linden. Operating system structures to support security and reliable software. *National Bureau of Standards Technical Report 919*, 1976. URL http://csrc.nist.gov/publications/history/lind76.pdf.

[47] G. Graham and Peter Denning. Protection - principles and practice. *AFIPS Conf. Proceedings*, 40:417–429, 1972.

[48] Open Group. Single unix specification, version 3. *IEEE Std 1003.1*, 2008. URL http://www.unix.org/version3/ieee_std.html.

[49] Volkmar Lotz, Volker Kessler, and Georg Walter. A formal security model for microprocessor hardware. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 26(8), 2000.

[50] G. Klein. sel4: Formal verification of an os kernel. *22nd ACM Symposium on Operating System Principles*, 2009. URL http://www.sigops.org/sosp/sosp09/papers/klein-sosp09.pdf.

[51] CCRA. Common criteria for information technology security evaluation. *CCRA, V3.R4*, 2012. URL http://www.commoncriteriaportal.org/cc/.

[52] Y. Bertot. Coq in a hurry. *Technical Report, MARELLE - INRIA Sophia Antipolis*, 2010.

[53] Adam Chlipala. Effective interactive proofs for higher-order imperative programs. *ACM ICFP*, 2009. URL http://ynot.cs.harvard.edu/papers/icfp09.pdf.

[54] Adam Chlipala. *Certified Programming with Dependent Types*. The MIT Press, 2008. URL http://adam.chlipala.net/cpdt/cpdt.pdf.

[55] Leonardo de Moura. Proofs and refutations, and z3. *Proceedings of the LPAR Workshops, Knowledge Exchange: Automated Provers and Proof Assistants, and the 7th International Workshop on the Implementation of Logics*, 418, 2008.

[56] Y. Bertot and P. Casteran. *Interactive theorem proving and program development : Coq'Art : the calculus of inductive constructions*. Springer, Berlin; New York, 2004. ISBN 3540208542 9783540208549. ID: 55514299.

[57] David Delahaye. A tactic language for the system Coq. *In Proceedings Logic for Programming and Automated Reasoning*, 2000. URL http://cedric.cnam.fr/~delahaye/papers/ltac%20(LPAR'00).pdf.

[58] Thierry Coquand and G. Huet. The calculus of construction. *Technical Report 530 (INRIA, Centre de Rocquencourt)*, 1986. URL http://hal.inria.fr/docs/00/07/60/24/PDF/RR-0530.pdf.

[59] L. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363, 1989.

[60] Ulf Norell. Dependently typed programming in Agda. *Advanced Functional Programming Lecture Notes in Computer Science*, 5832:230, 2009.

[61] Edwin Brady. Idris - systems programming meets full dependent types. *ACM Programming Languages and Program Verification*, 2011.

[62] Lennart Augustsson. Cayenne - a language with dependent types. *Advanced Functional Programming Lecture Notes in Computer Science*, 1608:240, 1999.

[63] Per Martin-Lof. Intuitionistic type theory. *Bibliopolis.*, 1984.

[64] Gail-Joon Ahn, Seung-Phil Hong, and Michael Shin. Reconstructing a formal security model. *Information and Software Technology*, 44(11):649–657, 2002.

[65] Charles Pfleeger. *Security in Computing, 4th ed.* Prentice Hall, 4 edition, 2006. URL http://rvrjcce.ac.in/ksm/sc.pdf.

[66] F. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30, 2000. URL https://www.cs.cornell.edu/fbs/publications/EnfSecPols.pdf.

[67] Konstantin Beznosov and Phillippe Kruchten. Towards agile security assurance. *Proceedings of the workshop on New security paradigms, Nova Scotia, Canada*, pages 47–54, 2004. URL http://lersse-dl.ece.ubc.ca/record/87/files/87.pdf.

[68] Kent Beck. *Test-Driven Development by Example.* Addison Wesley, 2003.

[69] Kent Beck. *XP Explained, 1st Edition.* Addison-Wesley Professional, 1999.

[70] Edward Miller and William Howden. Software testing and validation techniques. *IEEE Computer Society Press*, page 16, 1978.

[71] Y. Tsim. An adaptation to iso 9001:2000 for certified organisations. *Managerial Auditing Journal*, 17(5):245, 2002.

[72] Stacy Prowell. Cleanroom software engineering: Technology and process. *Addison-Wesley*, 1999.

[73] Herbert Benington. Production of large computer programs. *IEEE Annals of the History of Computing*, 5(4):350, 1983.

[74] E Michael Maximilien and Laurie Williams. Assessing test-driven development at ibm. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 564–569. IEEE, 2003.

[75] C. Earl and M. Might. Introspective pushdown analysis of higher-order programs. *ICFP*, 2012. URL http://arxiv.org/pdf/1207.1813.pdf;https://www.youtube.com/watch?v=HaPsYmOmgcI.

[76] Thomas McCabe. A complexity measure. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, page 308, 1976.

[77] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. *ACM SIGPLAN Notices*, 46(4):53, 2011.

[78] Carlos Sarraute. Penetration testing == pomdp solving? *Core Security*, 2013. URL http://arxiv.org/pdf/1306.4714.pdf.

[79] Carlos Sarraute. Automated attack planning. *Instituto Tecnologico de Buenos Aires*, 2012. URL http://arxiv.org/pdf/1307.7808.pdf.

[80] David Wheeler. Preventing heartbleed. *IEEE Computer*, 47(8):80, 2014.

[81] Rick Wash. Folk models of home computer security. *Symposium on Usable Privacy and Security (SOUPS)*, 2010.

[82] Martin Puterman. *Markov Decision Processes Discrete Stochastic Dynamic Programming*. Wiley Interscience, 2005.

[83] E. Sondik. The optimal control of partially observable Markov processes. *STANFORD UNIVERSITY TECHNICAL REPORT*, 1971.

[84] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. Establishing browser security guarantees through formal shim verification. *USENIX Security*, 2012. URL http://goto.ucsd.edu/quark/usenix12.pdf.

[85] Boniface Hicks. Secure systems development using security-typed languages. *Pennsylvania State University PhD dissertation*, 2007. URL https://etda.libraries.psu.edu/paper/8013/3303.

[86] M. Clarkson. Hyperproperties. *Journal of Computer Security*, 18: 1157–1210, 2010. URL https://www.cs.cornell.edu/fbs/publications/Hyperproperties.pdf.

[87] A. Sabelfeld. Language-based information-flow security. *IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS*, 21(1), 2003. URL http://www.cse.chalmers.se/~andrei/jsac.pdf.

[88] J. Morgenstern and D. Licata. Security-typed programming within dependently typed programming. *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, page 169, 2010.

[89] Edwin Brady. Programming and reasoning with algebraic effects and dependent types. *ICFP*, 2013.

[90] James McKinna. Why dependent types matter. *ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2006.

[91] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. *ACM Symposium on Principles of Programming Languages*, 1999.

[92] Amy Felty. Implementing tactics and tacticals in a higher-order logic programming language. *Journal of Automated Reasoning*, 11(1):43, 1993.

[93] J. Reynolds. Separation logic: A logic for shared mutable data structures. *IEE Computer Society, Logic in Computer Science*, 2002. URL http://www.cs.cmu.edu/~jcr/seplogic.pdf.

[94] C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10), 1969. URL http://www.spatial.maine.edu/~worboys/processes/hoare%20axiomatic.pdf.

[95] Donald Knuth and Luis Pardo. Early development of programming languages. *Encyclopedia of Computer Science and Technology*, page 419, 2009.

[96] Alonzo Church. The calculi of lambda-conversion. *Princeton University Press*, 1941.

[97] A. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230, 1937.

[98] Georges Gonthier. Formal proof-the four-color theorem. *Notices of the American Mathematical Society*, 55(11):1382, 2008.

[99] J. Paris. A mathematical incompleteness in peano arithmetic. *Studies in Logic and the Foundations of Mathematics*, 1977.

[100] Eugenio Moggi. Notions of computation and monads. *Information and computation*, 93(1):55, 1991.

[101] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107, 2009.

[102] Dorothy Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[103] J. McLean. The algebra of security. *IEEE Symposium on Security and Privacy*, page 18, 1988. URL http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8092&isnumber=427.

[104] G. Gratzer. *General lattice theory*. New York: Academic Press, 1978.

[105] Carl Landwehr. Formal models for computer security. *ACM Computing Surveys*, 13(3), 1981. URL http://winlab.rutgers.edu/~trappe/Courses/AdvSec05/LandwehrSecModels.pdf.

[106] Gregory Malecha, Greg Morrisett, and Ryan Wisnesky. Trace based verification of imperative programs with i/o. *Journal of Symbolic Computation. Vol 46, issue 2.*, 46(2), 2009. URL http://ynot.cs.harvard.edu/papers/jsc-wwv-10.pdf.

[107] Matt Bishop. *Computer security : art and science*. Addison-Wesley, 2003.

[108] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 1993.

[109] W. Howard. The formulae-as-types notion of construction. *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism. Academic Press Limited, 1980.*, page 479, 1980.

[110] Simon L Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[111] Philip Wadler. Propositions as sessions. *ACM SIGPLAN Notices.*, 47(9), 2012.

[112] Craig Rowland. Covert channels in the tcp/ip protocol suite. *First Monday*, 2(5), 1997. URL http://firstmonday.org/ojs/index.php/fm/article/view/528/449.

[113] D. Bell. Looking back at the bell lapadula model. *ACSAC*, 2005. URL http://www.acsac.org/2005/papers/Bell.pdf.

[114] Jean Berstel and Dominique Perrin. *Theory of codes*, volume 117. Academic Press, 1985.

[115] Jeffrey Outlaw Shallit. *A second course in formal languages and automata theory*, volume 179. Cambridge University Press Cambridge, 2009.

[116] Nicolas Bourbaki. *Algebra I: chapters 1-3*. Springer Science & Business Media, 1998.

[117] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics doklady*, 10(8):707–710, 1966.

[118] Marc Joye and Francis Olivier. Side-channel analysis. *Encyclopedia of Cryptography and Security*, pages 1198–1204, 2011.

[119] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In *Cryptographic Hardware and Embedded Systems-CHES 2004*, pages 16–29. Springer, 2004.

[120] Zohar Manna and Amir Pnueli. Axiomatic approach to total correctness of programs. *Acta Informatica*, 3(3):243–263, 1974.

[121] Wouter Swierstra. Xmonad in coq (experience report): Programming a window manager in a proof assistant. In *ACM SIGPLAN Notices*, volume 47, pages 131–136. ACM, 2012.

[122] Stephanie Weirich. Depending on types. *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, 2014.

[123] Kees Doets. *The Haskell Road to Logic, Maths and Programming*. King's College Publications, 1 edition, 2004.

[124] Martin Fowler. *Refactoring. Improving the Design of Existing Code.* Addison-Wesley, 1999.

[125] Taiichi Ōno. *Toyota production system: beyond large-scale production*. Productivity press, 1988.

[126] Mary Poppendieck and Tom Poppendieck. *Lean software development: an agile toolkit*. Addison-Wesley Professional, 2003.

[127] Richard Bird. Functional pearl: A program to solve sudoku. 16(6):671–679, 2006. URL http://www.cs.tufts.edu/~nr/cs257/archive/richard-bird/sudoku.pdf.

[128] Bertrand Meyer. Applying "design by contract. *IEEE Computer*, 25(10):40–51, 1992. URL http://se.ethz.ch/~meyer/publications/computer/contract.pdf.

[129] C.F. Kemerer and M.C. Paulk. The impact of design and code reviews on software quality: An empirical study based on psp data. *Software Engineering, IEEE Transactions on*, 35(4):534–550, July 2009. ISSN 0098-5589. doi: 10.1109/TSE.2009.27. URL http://www.pitt.edu/~ckemerer/PSP_Data.pdf.

[130] M.V. Mantyla and C. Lassenius. What types of defects are really discovered in code reviews? *Software Engineering, IEEE Transactions on*, 35(3):430–448, May 2009. ISSN 0098-5589. doi: 10.1109/TSE.2008.71.

[131] Frederick P Brooks. *The mythical man-month*, volume 1995. Addison-Wesley Reading, MA, 1975.

[132] Niels Provos. Preventing privilege escalation. *Proceedings of the 12th USENIX Security Symposium*, page 231, 2003.

[133] Lynne M Reder, Xiaonan L Liu, Alexander Keinath, and Vencislav Popov. Building knowledge requires bricks, not sand: The critical role of familiar constituents in learning. *Psychonomic bulletin & review*, pages 1–7, 2015.

[134] Peter Norvig. *The General Problem Solver*, volume 3. Morgan Kaufmann, 1992.

[135] Paul Graham. *On lisp*. Prentice Hall, 1994.

[136] Bjarne Stroustrup. *The C++ programming language*. Pearson Education India, 1986.

[137] Yukio Matsumoto and K Ishituka. *Ruby programming language*. Addison Wesley Publishing Company, 2002.

[138] Brian W Kernighan and Dennis M Ritchie. *The C programming language*, volume 2. Prentice-Hall Englewood Cliffs, 1988.

[139] David Christiansen. Dependent type providers. *Proceedings of the 9th ACM SIGPLAN Workshop on Generic Programming*, page 25, 2013.

[140] Konstantin Weitz, Gene Kim, Siwakorn Srisakaokul, and Michael D Ernst. A type system for format strings. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 127–137. ACM, 2014.

[141] Eric Spishak, Werner Dietl, and Michael Ernst. A type system for regular expressions. *Formal Techniques for Java-like Programs*, 2012.

[142] S. Konstantinidis. Computing the edit distance of a regular language. *Information and Computation*, 205:1307, 2007.

[143] Christian Doczkal. A constructive theory of regular languages in coq. *Lecture Notes in Computer Science*, 8307:82–97, 2013.

[144] Herbert A Simon. *The architecture of complexity*. Springer, 1991.

[145] Neelakantan R. Krishnaswami, Pierre Pradic, and Nick Benton. Integrating linear and dependent types. 2015.

Appendix - Idris Syntax Guide

This section provides a concise guide to Idris syntax and references to resources to learn more about the language.

The official web page[1] contains links to documentation[2], the wiki[3], and the source code repository on github[4]. The tutorial[5] is an excellent starting point.

Figure 1 shows examples of some of the most common syntactic elements in Idris.

- **:** the colon operator defines a type.
- **::** double colons denote concatenation.
- **->** the arrow operator separates arguments.
- _ the underscore is a placeholder for pattern matching (matches anything).
- () parenthesis group arguments for pattern matching and control order of evaluation. Empty parentheses denote the empty type, "void".
- **'** single quotes enclose character literals.
- **"** double quotes enclose strings.
- **case ... of** introduces pattern matching.
- **?name** denotes a "hole" in the program.
- **<-** is shorthand for reading from a monad.

[1] idris-lang.org
[2] http://www.idris-lang.org/documentation/
[3] https://github.com/idris-lang/Idris-dev/wiki
[4] https://github.com/idris-lang/Idris-dev
[5] http://docs.idris-lang.org/en/latest/tutorial/index.html#tutorial-index

```
funcName : Nat -> Char
funcName Z     = 'Z'
funcName (S Z) = 'S'
funcName _     = 'N'

otherFunc : (c : Char) -> (pf : isUpper c = True) -> Char
otherFunc c pf = ?holeNameWithProofAccessible

ArgType : Type
ArgType = Type

data SomeType : Type where
  MkSomeType : (t : ArgType) -> SomeType


oneMore : String -> Bool
oneMore x = case length x of
                0   =>  let v = 123 in
                        False
                _   => True

main : IO ()
main = do
  args <- getArgs
  case args of
       [self]  => putStrLn "usage: ..."
       [_, f]  => do { contents <- readFile f
                       -- More functions
                       {- long comment
                       -}
                     }
```

FIGURE 1: Idris syntax samples.

- **do { ... }** do-notation, used for blocks of repeated monad usage and sequential operations.
- **let ... in** declares a new variable for the following scope.
- => denotes the body of a matched pattern. Also denotes the body of a lambda expression.
- **data ... where** introduces a new type with the constructors given after the "where" keyword
- **Type** the type of the top-level type.

- – used for single line comments.
- **{- ... -}** used for multiple line comments.

Appendix - Code

This code may be periodically updated, and the latest versions are available online in the following repositories:

- https://github.com/bgoodspeed/idris-strings
- https://github.com/bgoodspeed/idris-misc
- https://github.com/bgoodspeed/idris-secure
- https://github.com/bgoodspeed/idris-benchmarks

A.1 Extracted Code

This is the extracted code in OCaml for the modules StringHelper, AuthenticationSystem and Login, per the extraction settings in Extract.

```
(** val app : 'a1 list -> 'a1 list -> 'a1 list **)

let rec app l m =
  match l with
  | [] -> m
  | a::l1 -> a::(app l1 m)

(** val nth : MlCoq.nat -> 'a1 list -> 'a1 -> 'a1 **)

let rec nth n l default =
  match n with
  | MlCoq.O ->
    (match l with
```

```
    | [] -> default
    | x::l' -> x)
  | MICoq.S m ->
    (match l with
    | [] -> default
    | x::t -> nth m t default)

(** val rev : 'a1 list -> 'a1 list **)

let rec rev = function
| [] -> []
| x::l' -> app (rev l') (x::[])

(** val map : ('a1 -> 'a2) -> 'a1 list -> 'a2 list **)

let rec map f = function
| [] -> []
| a::t -> (f a)::(map f t)

(** val append : MICoq.ascii list -> MICoq.ascii list -> MICoq.ascii list **)

let rec append s1 s2 =
  match s1 with
  | [] -> s2
  | c::s1' -> c::(append s1' s2)

(** val substring :
    MICoq.nat -> MICoq.nat -> MICoq.ascii list -> MICoq.ascii list **)

let rec substring n m s =
  match n with
  | MICoq.O ->
    (match m with
    | MICoq.O -> []
    | MICoq.S m' ->
      (match s with
      | [] -> s
      | c::s' -> c::(substring MICoq.O m' s')))
  | MICoq.S n' ->
    (match s with
    | [] -> s
```

```
55      | c::s' -> substring n' m s')
56
57  (** val prefix : MICoq.ascii list -> MICoq.ascii list -> bool **)
58
59  let rec prefix s1 s2 =
60    match s1 with
61    | [] -> true
62    | a::s1' ->
63      (match s2 with
64       | [] -> false
65       | b::s2' -> if (=) a b then prefix s1' s2' else false)
66
67  type 't sT =  't STImpl.axiom_ST
68
69  type 't sTsep = 't sT
70
71  (** val printStringLn : MICoq.ascii list -> unit sTsep **)
72
73  let printStringLn = BasisImpl.axiom_printStringLn
74
75  (** val split_string_r :
76      MICoq.ascii -> MICoq.ascii list -> MICoq.ascii list -> MICoq.ascii list
77      list -> MICoq.ascii list list **)
78
79  let rec split_string_r delimiter str current_word accumulated_words =
80    match str with
81    | [] -> current_word::accumulated_words
82    | a::b ->
83      if prefix (delimiter::[]) str
84      then split_string_r delimiter b [] (current_word::accumulated_words)
85      else split_string_r delimiter b
86             (append current_word (substring MICoq.O (MICoq.S MICoq.O) str))
87             accumulated_words
88
89  (** val split_string :
90      MICoq.ascii -> MICoq.ascii list -> MICoq.ascii list -> MICoq.ascii list
91      list -> MICoq.ascii list list **)
92
93  let split_string delimiter str current_word accumulated_words =
94    rev (split_string_r delimiter str current_word accumulated_words)
95
```

```
(** val matching_in_field :
    MICoq.nat -> MICoq.ascii list -> MICoq.ascii list list list ->
    MICoq.ascii list list **)

let rec matching_in_field idx token = function
| [] -> []
| x::y ->
  if if prefix token
        (nth idx x ((MICoq.Ascii (false, true, true, true, false, false,
          true, false))::((MICoq.Ascii (true, true, true, true, false,
          false, true, false))::((MICoq.Ascii (false, false, false, false,
          true, false, true, false))::((MICoq.Ascii (true, false, true,
          false, false, false, true, false))::[])))))
     then prefix
        (nth idx x ((MICoq.Ascii (false, true, true, true, false, false,
          true, false))::((MICoq.Ascii (true, false, true, false, false,
          false, true, false))::((MICoq.Ascii (false, true, true, false,
          true, false, true, false))::((MICoq.Ascii (true, false, true,
          false, false, false, true, false))::((MICoq.Ascii (false, true,
          false, false, true, false, true, false))::[]))))))) token
     else false
  then x
  else matching_in_field idx token y

(** val string_rev : MICoq.ascii list -> MICoq.ascii list **)

let rec string_rev = function
| [] -> []
| c::rest -> append (string_rev rest) (c::[])

(** val compute_hash : MICoq.ascii list -> MICoq.ascii list **)

let rec compute_hash input =
  string_rev input

(** val check_login :
    MICoq.ascii list -> MICoq.ascii list -> MICoq.ascii list list ->
    MICoq.ascii list **)

let rec check_login user pass auth_db =
  let entry =
```

```
137     matching_in_field MICoq.O user
138       (map (fun str ->
139         split_string (MICoq.Ascii (false, true, false, true, true, true,
140           false, false)) str [] []) auth_db)
141   in
142   let passwd_crypted = compute_hash pass in
143   let db_passwd =
144     nth (MICoq.S MICoq.O) entry ((MICoq.Ascii (false, true, false, false,
145       false, false, true, false))::((MICoq.Ascii (true, false, false, false,
146       false, false, true, false))::((MICoq.Ascii (false, false, true, false,
147       false, false, true, false))::((MICoq.Ascii (false, false, false, false,
148       true, false, true, false))::((MICoq.Ascii (true, false, false, false,
149       false, false, true, false))::((MICoq.Ascii (true, true, false, false,
150       true, false, true, false))::((MICoq.Ascii (true, true, false, false,
151       true, false, true, false))::((MICoq.Ascii (true, true, true, false,
152       true, false, true, false))::((MICoq.Ascii (true, true, true, true,
153       false, false, true, false))::((MICoq.Ascii (false, true, false, false,
154       true, false, true, false))::((MICoq.Ascii (false, false, true, false,
155       false, false, true, false))::[]))))))))))))
156   in
157   if if prefix passwd_crypted db_passwd
158     then prefix db_passwd passwd_crypted
159     else false
160   then (MICoq.Ascii (true, true, true, true, false, true, true,
161         false))::((MICoq.Ascii (true, true, false, true, false, true, true,
162         false))::[])
163   else (MICoq.Ascii (false, true, true, true, false, true, true,
164         false))::((MICoq.Ascii (true, true, true, true, false, true, true,
165         false))::[])
166
167 (** val main : unit sTsep **)
168
169 let main =
170   printStringLn
171     (check_login ((MICoq.Ascii (true, false, true, false, true, true, true,
172       false))::((MICoq.Ascii (true, true, false, false, true, true, true,
173       false))::((MICoq.Ascii (true, false, true, false, false, true, true,
174       false))::((MICoq.Ascii (false, true, false, false, true, true, true,
175       false))::[])))) ((MICoq.Ascii (false, false, false, false, true, true,
176       true, false))::((MICoq.Ascii (true, false, false, false, false, true,
177       true, false))::((MICoq.Ascii (true, true, false, false, true, true,
```

```
true, false))::((MlCoq.Ascii (true, true, false, false, true, true,
true, false))::((MlCoq.Ascii (true, true, true, false, true, true,
true, false))::((MlCoq.Ascii (false, false, true, false, false, true,
true, false))::[]))))))) (((MlCoq.Ascii (false, false, true, false,
false, true, true, false))::((MlCoq.Ascii (true, false, false, true,
false, true, true, false))::((MlCoq.Ascii (false, true, true, false,
false, true, true, false))::((MlCoq.Ascii (false, true, true, false,
false, true, true, false))::((MlCoq.Ascii (true, false, true, false,
false, true, true, false))::((MlCoq.Ascii (false, true, false, false,
true, true, true, false))::((MlCoq.Ascii (true, false, true, false,
false, true, true, false))::((MlCoq.Ascii (false, true, true, true,
false, true, true, false))::((MlCoq.Ascii (false, false, true, false,
true, true, true, false))::((MlCoq.Ascii (false, true, false, true,
true, true, false, false))::((MlCoq.Ascii (true, false, false, false,
false, true, true, false))::((MlCoq.Ascii (true, true, false, false,
true, true, true, false))::((MlCoq.Ascii (false, false, true, false,
false, true, true, false))::((MlCoq.Ascii (false, true, true, false,
false, true, true, false))::((MlCoq.Ascii (false, true, false, true,
true, true, false, false))::((MlCoq.Ascii (true, true, true, false,
true, true, true, false))::((MlCoq.Ascii (false, false, false, true,
false, true, true, false))::((MlCoq.Ascii (true, false, false, false,
false, true, true, false))::((MlCoq.Ascii (false, false, true, false,
true, true, true, false))::((MlCoq.Ascii (true, false, true, false,
false, true, true, false))::((MlCoq.Ascii (false, true, true, false,
true, true, true, false))::((MlCoq.Ascii (true, false, true, false,
false, true, true, false))::((MlCoq.Ascii (false, true, false, false,
true, true, true, false))::((MlCoq.Ascii (false, true, false, true,
true, true, false, false))::((MlCoq.Ascii (true, false, false, true,
false, true, true, false))::((MlCoq.Ascii (false, true, false, false,
true, true, true, false))::((MlCoq.Ascii (false, true, false, false,
true, true, true, false))::((MlCoq.Ascii (true, false, true, false,
false, true, true, false))::((MlCoq.Ascii (false, false, true, true,
false, true, true, false))::((MlCoq.Ascii (true, false, true, false,
false, true, true, false))::((MlCoq.Ascii (false, true, true, false,
true, true, true, false))::((MlCoq.Ascii (true, false, true, false,
false, true, true, false))::((MlCoq.Ascii (false, true, true, true,
false, true, true, false))::((MlCoq.Ascii (false, false, true, false,
true, true, true,
false))::[])))))))))))))))))))))))))))))))))))::(((MlCoq.Ascii (true,
false, false, false, false, true, true, false))::((MlCoq.Ascii (false,
true, true, true, false, true, true, false))::((MlCoq.Ascii (true,
```

```
219     true, true, true, false, true, true, false))::((MICoq.Ascii (false,
220     false, true, false, true, true, true, false))::((MICoq.Ascii (false,
221     false, false, true, false, true, true, false))::((MICoq.Ascii (true,
222     false, true, false, false, true, true, false))::((MICoq.Ascii (false,
223     true, false, false, true, true, true, false))::((MICoq.Ascii (false,
224     true, false, true, true, true, false, false))::((MICoq.Ascii (true,
225     true, true, false, true, true, true, false))::((MICoq.Ascii (false,
226     false, false, true, false, true, true, false))::((MICoq.Ascii (true,
227     false, false, false, false, true, true, false))::((MICoq.Ascii (false,
228     false, true, false, true, true, true, false))::((MICoq.Ascii (true,
229     false, true, false, false, true, true, false))::((MICoq.Ascii (false,
230     true, true, false, true, true, true, false))::((MICoq.Ascii (true,
231     false, true, false, false, true, true, false))::((MICoq.Ascii (false,
232     true, false, false, true, true, true, false))::((MICoq.Ascii (false,
233     true, false, true, true, true, false, false))::((MICoq.Ascii (true,
234     false, false, false, false, true, true, false))::((MICoq.Ascii (true,
235     true, false, false, true, true, true, false))::((MICoq.Ascii (false,
236     false, true, false, false, true, true, false))::((MICoq.Ascii (false,
237     true, true, false, false, true, true,
238     false))::[])))))))))))))))))))))))::(((MICoq.Ascii (true, false, true,
239     false, true, true, true, false))::((MICoq.Ascii (true, true, false,
240     false, true, true, true, false))::((MICoq.Ascii (true, false, true,
241     false, false, true, true, false))::((MICoq.Ascii (false, true, false,
242     false, true, true, true, false))::((MICoq.Ascii (false, true, false,
243     true, true, true, false, false))::((MICoq.Ascii (false, false, true,
244     false, false, true, true, false))::((MICoq.Ascii (true, true, true,
245     false, true, true, true, false))::((MICoq.Ascii (true, true, false,
246     false, true, true, true, false))::((MICoq.Ascii (true, true, false,
247     false, true, true, true, false))::((MICoq.Ascii (true, false, false,
248     false, false, true, true, false))::((MICoq.Ascii (false, false, false,
249     false, true, true, true, false))::((MICoq.Ascii (false, true, false,
250     true, true, true, false, false))::((MICoq.Ascii (false, true, true,
251     false, false, true, true, false))::((MICoq.Ascii (true, true, true,
252     true, false, true, true, false))::((MICoq.Ascii (true, true, true,
253     true, false, true, true, false))::[])))))))))))))))::(((MICoq.Ascii
254     (true, false, true, true, false, true, true, false))::((MICoq.Ascii
255     (true, true, true, true, false, true, true, false))::((MICoq.Ascii
256     (false, true, true, true, false, true, true, false))::((MICoq.Ascii
257     (true, true, false, true, false, true, true, false))::((MICoq.Ascii
258     (true, false, true, false, false, true, true, false))::((MICoq.Ascii
259     (true, false, false, true, true, true, true, false))::((MICoq.Ascii
```

```
260     (false, true, false, true, true, true, false, false))::((MICoq.Ascii
261     (false, true, false, false, false, true, true, false))::((MICoq.Ascii
262     (true, false, true, false, true, true, true, false))::((MICoq.Ascii
263     (false, false, true, false, true, true, true, false))::((MICoq.Ascii
264     (false, false, true, true, false, true, true, false))::((MICoq.Ascii
265     (true, false, true, false, false, true, true, false))::((MICoq.Ascii
266     (false, true, false, false, true, true, true, false))::((MICoq.Ascii
267     (false, true, false, true, true, true, false, false))::((MICoq.Ascii
268     (true, false, false, true, false, true, true, false))::((MICoq.Ascii
269     (true, false, false, true, false, true, true, false))::((MICoq.Ascii
270     (true, false, false, true, false, true, true, false))::((MICoq.Ascii
271     (true, false, false, true, false, true, true,
272     false))::[])))))))))))))))))))::[])))))
```

Appendix - String Handling Code

A.2 Homomorphism

```
module Homomorphism

import Syntax.PreorderReasoning

--Thanks to David Christiansen from the Idris mailing list for fixing this.

data Hom : (a, b : Type) -> Semigroup a -> Semigroup b -> Type where
  MkHom : (actxt : Semigroup a, bctxt : Semigroup b) => (h : a -> b) ->
          (preservesGroup : (a1 : a) -> (a2 : a) ->
          h ((<+>) @{actxt} a1 a2) = (<+>) @{bctxt} (h a1) (h a2)) ->
                                      Hom a b actxt bctxt



homRefl : (as : Semigroup a) => Hom a a as as
homRefl = MkHom id (\x,y => Refl)

homTrans : (adict : Semigroup a, bdict : Semigroup b,
            cdict : Semigroup c) =>
            Hom a b adict bdict -> Hom b c bdict cdict ->
            Hom a c adict cdict
homTrans @{adict} @{bdict} @{cdict} (MkHom h preservesGroup)
          (MkHom h' preservesGroup') =
  MkHom @{adict} @{cdict} (\x => h' (h x))
            (\something, another =>
```

```
                (h' (h (something <+> another))) ={ ?prf1 }=
                (h' (h something <+> h another)) ={ ?prf2 }=
                (h' (h something) <+> h' (h another)) QED)


data MonoidHom : (a, b : Type) -> Semigroup a -> Semigroup b ->
                 Monoid a -> Monoid b -> Type where
  MkMonoidHom : (actxt : Semigroup a, bctxt : Semigroup b,
                actxtM : Monoid a, bctxtM : Monoid b) =>
                (h : a -> b) ->
                (preservesGroup : (a1 : a) -> (a2 : a) ->
                h ((<+>) @{actxt} a1 a2) =
                    (<+>) @{bctxt} (h a1) (h a2)) ->
                    (preservesNeutral : h (neutral @{actxtM}) =
                      neutral @{bctxtM}) ->
                        MonoidHom a b actxt bctxt actxtM bctxtM


monoidHomRefl : (as : Semigroup a, am : Monoid a) =>
                MonoidHom a a as as am am
monoidHomRefl = MkMonoidHom id (\x,y => Refl) Refl

monoidHomTrans : (as : Semigroup a, bs : Semigroup b,
                  cs : Semigroup c, am : Monoid a,
                  bm : Monoid b, cm : Monoid c) =>
                      MonoidHom a b as bs am bm ->
                      MonoidHom b c bs cs bm cm ->
                      MonoidHom a c as cs am cm
monoidHomTrans @{actxt} @{bctxt} @{cctxt} @{am} @{bm} @{cm}
  (MkMonoidHom h preservesGroup preservesNeutral)
  (MkMonoidHom h' preservesGroup' preservesNeutral') =
      MkMonoidHom @{actxt} @{cctxt} @{am} @{cm}
         (\x => h' (h x))
            (\p,q =>
                (h' (h ((<+>) @{actxt} p q)))
                  ={ cong $ preservesGroup p q }=
```

```
                (h' ((<+>) @{bctxt} (h p) (h q)))
                  ={ preservesGroup' (h p) (h q) }=
                ((<+>) @{cctxt}  (h' (h p)) (h' (h q))) QED)
                        ?monoidIdentProof

---------- Proofs ----------

Homomorphism.monoidIdentProof = proof
  intros
  compute
  rewrite sym preservesNeutral
  rewrite sym preservesNeutral'
  trivial


Homomorphism.prf2 = proof
  intros
  rewrite preservesGroup' (h something) (h another)
  trivial


Homomorphism.prf1 = proof
  intros
  rewrite preservesGroup something another
  trivial
```

A.3 ListQuantifiers

```
module ListQuantifiers

import DecHelper
-- This is identical to the Data.Vect.Quantifiers but for lists.

%default total
```

```
data Any : (P : a -> Type)-> List a -> Type where
  Here  : {P : a -> Type} -> {xs : List a} -> P x -> Any P (x :: xs)
  There : {P : a -> Type} -> {xs : List a} -> Any P xs -> Any P (x :: xs)

anyNilAbsurd : {P : a -> Type} -> Any P Nil -> Void
anyNilAbsurd (Here _) impossible
anyNilAbsurd (There _) impossible

anyElim : {xs : List a} -> {P : a -> Type} -> (Any P xs -> b) -> (P x -> b)
   -> Any P (x :: xs) -> b
anyElim _ g (Here p)  = g p
anyElim f _ (There p) = f p

any : {P : a -> Type} -> (dec : (x : a) -> Dec (P x)) -> (xs : List a) ->
      Dec (Any P xs)
any _ [] = No anyNilAbsurd
any p (x :: xs) with (p x)
  | Yes prf = Yes (Here prf)
  | No prf  = case any p xs of
                  Yes prf' => Yes (There prf')
                  No  prf' => No  (anyElim prf' prf)


data All : (P : a -> Type) -> List a -> Type where
  Nil : {P : a -> Type} -> All P Nil
  (::) : {P : a -> Type} -> {xs : List a} -> P x
         -> All P xs -> All P (x :: xs)


negAnyAll : {P : a -> Type} -> {xs : List a} -> Not (Any P xs) ->
            All (\x => Not (P x)) xs
negAnyAll {xs=Nil} _ = Nil
negAnyAll {xs=(x::xs)} f = (\x => f (Here x)) ::
                           negAnyAll (\x => f (There x))

notAllHere : {P : a -> Type} -> {xs : List a} -> Not (P x) ->
```

```
             All P (x :: xs) -> Void
notAllHere _ Nil impossible
notAllHere np (p :: _) = np p

notAllThere : {P : a -> Type} -> {xs : List a} -> Not (All P xs) ->
              All P (x :: xs) -> Void
notAllThere _ Nil impossible
notAllThere np (_ :: ps) = np ps

all : {P : a -> Type} -> (dec : (x : a) -> Dec (P x)) -> (xs : List a) ->
      Dec (All P xs)
all _ Nil = Yes Nil
all d (x :: xs) with (d x)
  | No prf = No (notAllHere prf)
  | Yes prf = case all d xs of
                  Yes prf' => Yes (prf :: prf')
                  No  prf' => No (notAllThere prf')
```

A.4 ListCharDec

```
module ListCharDec

import CharDec
import ListQuantifiers


%default total

allUpperDec : (cs : List Char) -> Dec (All isUpperT cs)
allUpperDec cs = all isUpperDec cs

anyUpperDec : (cs : List Char) -> Dec (Any isUpperT cs)
anyUpperDec cs = any isUpperDec cs

allLowerDec : (cs : List Char) -> Dec (All isLowerT cs)
```

```
allLowerDec cs = all isLowerDec cs

anyLowerDec : (cs : List Char) -> Dec (Any isLowerT cs)
anyLowerDec cs = any isLowerDec cs

allAlphaDec : (cs : List Char) -> Dec (All isAlphaT cs)
allAlphaDec cs = all isAlphaDec cs

anyAlphaDec : (cs : List Char) -> Dec (Any isAlphaT cs)
anyAlphaDec cs = any isAlphaDec cs


allDigitDec : (cs : List Char) -> Dec (All isDigitT cs)
allDigitDec cs = all isDigitDec cs

anyDigitDec : (cs : List Char) -> Dec (Any isDigitT cs)
anyDigitDec cs = any isDigitDec cs

allAlphaNumDec : (cs : List Char) -> Dec (All isAlphaNumT cs)
allAlphaNumDec cs = all isAlphaNumDec cs

anyAlphaNumDec : (cs : List Char) -> Dec (Any isAlphaNumT cs)
anyAlphaNumDec cs = any isAlphaNumDec cs


allNLDec : (cs : List Char) -> Dec (All isNLT cs)
allNLDec cs = all isNLDec cs

anyNLDec : (cs : List Char) -> Dec (Any isNLT cs)
anyNLDec cs = any isNLDec cs



allHexDigitDec : (cs : List Char) -> Dec (All isHexDigitT cs)
allHexDigitDec cs = all isHexDigitDec cs
```

```
anyHexDigitDec : (cs : List Char) -> Dec (Any isHexDigitT cs)
anyHexDigitDec cs = any isHexDigitDec cs

allOctDigitDec : (cs : List Char) -> Dec (All isOctDigitT cs)
allOctDigitDec cs = all isOctDigitDec cs

anyOctDigitDec : (cs : List Char) -> Dec (Any isOctDigitT cs)
anyOctDigitDec cs = any isOctDigitDec cs


allSpaceDec : (cs : List Char) -> Dec (All isSpaceT cs)
allSpaceDec cs = all isSpaceDec cs

anySpaceDec : (cs : List Char) -> Dec (Any isSpaceT cs)
anySpaceDec cs = any isSpaceDec cs
```

A.5 ListDecisions

```
module ListDecisions

import Decidable.Equality
import DecHelper
import Utilities

isPrefixOfT : (Eq a) => List a -> List a -> Type
isPrefixOfT xs ys = isPrefixOf xs ys = True

isPrefixOfDec : (Eq a) => (xs : List a) -> (ys : List a) ->
                Dec (isPrefixOfT xs ys)
isPrefixOfDec xs ys with (isPrefixOf xs ys)
  | True  = Yes Refl
  | False = No  falseNotTrue


isSuffixOfT : (Eq a) => List a -> List a -> Type
```

```
isSuffixOfT xs ys = isSuffixOf xs ys = True

isSuffixOfDec : (Eq a) => (xs : List a) -> (ys : List a) ->
                Dec (isSuffixOfT xs ys)
isSuffixOfDec xs ys with (isSuffixOf xs ys)
  | True  = Yes Refl
  | False = No  falseNotTrue

isPalindrome : (Eq a) => List a -> Bool
isPalindrome xs = (reverse xs) == xs

isPalindromeT : (Eq a) => List a -> Type
isPalindromeT xs = isPalindrome xs = True

isPalindromeDec : (Eq a) => (xs : List a) -> Dec (isPalindromeT xs)
isPalindromeDec xs with (isPalindrome xs)
  | True  = Yes Refl
  | False = No falseNotTrue


possibleBordersFor : (Eq a) =>  List a -> List (List a)
possibleBordersFor xs = let prefixes = inits xs
                            lxs = (div (length xs) 2) in
                            filter (\x => (length x > 0) &&
                                          (length x <= lxs)) prefixes

isBordered : (Eq a) => List a -> Bool
isBordered xs = let valid_prefixes = possibleBordersFor xs in
                    any (\pfx => isSuffixOf pfx xs) valid_prefixes


isBorderedT : (Eq a) => List a -> Type
isBorderedT xs = isBordered xs = True

isBorderedDec : (Eq a) => (xs : List a) -> Dec (isBorderedT xs)
isBorderedDec xs with (isBordered xs)
```

```
  | True  = Yes Refl
  | False = No falseNotTrue
```

A.6 ListWord

```
module ListWord

%default total

Str : Type
Str = List Char

strLength : Str -> Nat
strLength [] = Z
strLength (x :: xs) = S( strLength xs )

listAppendIsAssociative : (l : List Char) -> (c : List Char) ->
                          (r : List Char) ->
                          (l ++ (c ++ r)) = ((l ++ c) ++ r)
listAppendIsAssociative [] c r = Refl
listAppendIsAssociative (x :: xs) c r =
  let inductiveHypothesis = listAppendIsAssociative xs c r in
     ?listAppendIsAssociativeStepCase

-- list a already gives us Eq, Semigroup, Monoid
instance [verifiedSemigroupWord] VerifiedSemigroup Str where
  semigroupOpIsAssociative = listAppendIsAssociative


listAppendNeutralIsNeutralL : (l : List Char) -> (l ++ []) = l
listAppendNeutralIsNeutralL [] = Refl
listAppendNeutralIsNeutralL (x :: xs) =
  let inductiveHypothesis = listAppendNeutralIsNeutralL xs in
     ?listAppendNeutralIsNeutralLStepCase
```

```
listAppendNeutralIsNeutralR : (r : List Char) -> [] ++ r = r
listAppendNeutralIsNeutralR [] = Refl
listAppendNeutralIsNeutralR (x :: xs) =
  let inductiveHypothesis = listAppendNeutralIsNeutralR xs in
     ?listAppendNeutralIsNeutralRStepCase

instance [verifiedMonoidWord] VerifiedMonoid Str where
  monoidNeutralIsNeutralL = listAppendNeutralIsNeutralL
  monoidNeutralIsNeutralR = listAppendNeutralIsNeutralR


concatIsInvInjective : (c : Char) -> (s1 : Str) -> (s2 : Str) ->
                       (pf : s1 = s2) -> c :: s1 = c :: s2
concatIsInvInjective c s1 s2 pf = ?concatIsInvInjectivePf


concatIsInjective : (c : Char) -> (s1 : Str)  -> (s2 : Str) ->
                    {auto pf : c :: s1 = c :: s2 } -> s1 = s2
concatIsInjective _ _ _ {pf = Refl} = Refl

---------- Proofs ----------

ListWord.concatIsInvInjectivePf = proof
  intros
  rewrite pf
  trivial


ListWord.listAppendNeutralIsNeutralRStepCase = proof
  intros
  trivial


ListWord.listAppendNeutralIsNeutralLStepCase = proof
  intros
  rewrite inductiveHypothesis
```

```
  trivial


ListWord.listAppendIsAssociativeStepCase = proof
  intros
  rewrite inductiveHypothesis
  trivial
```

A.7 VectorWord

```
module VectorWord


import Data.Vect
%default total

Str : Nat -> Type -> Type
Str n t = Vect n t

strConcat : (Str n t) -> (Str m t)-> (Str (n + m) t)
strConcat x x1 = x ++ x1



instance VerifiedMonoid (Str n) where
  monoidNeutralIsNeutralL = ?listAppendNeutralIsNeutralL
  monoidNeutralIsNeutralR = ?listAppendNeutralIsNeutralR
```

A.8 WordCombinatorics

```
-- Inspired by A second course in formal languages and automata by Shallit

module WordCombinatorics
```

```
import DecHelper

Word : Type
Word = String

%default total


intDivCeil2 : Nat -> Nat
intDivCeil2 k = (div k 2) + (mod k 2)


prefixLengthRange : Word -> List Nat
prefixLengthRange w = [0..(intDivCeil2 (length w))]

prefixesOf : Word -> List Word
prefixesOf w with (unpack w)
  | [] = []
  | xs = let segs = inits xs in
             map pack segs


primitivePrefixCandidatesOf : Word -> List Word
primitivePrefixCandidatesOf w = let prefixes = prefixesOf w in
    filter (\x => (length x > 0) &&
                  (length x <= intDivCeil2 (length w)) &&
                  (mod (length w) (length x) == 0 ) ) prefixes

primitivePrefixCandidateLengthPairsOf : Word -> List (Word, Nat)
primitivePrefixCandidateLengthPairsOf w =
  let prefixes = primitivePrefixCandidatesOf w in
      map (\x => (x, div (length w) (length x))) prefixes


wordMult : Word -> Nat -> Word
wordMult x Z = ""
```

```
wordMult x (S k) = x ++ (wordMult x k)

computeCandidates : List (Word, Nat) -> List (Word)
computeCandidates xs = map (\(w,n) => wordMult w n) xs


prefixPowerOf : Word -> (Word, Nat)
prefixPowerOf x =
  let candidatePairs = primitivePrefixCandidateLengthPairsOf x
      validCandidates = filter (\(w,n) =>
                          (wordMult w n) == x) candidatePairs in
                          case validCandidates of
                             []               => (x, 1)
                             (apair :: pairs) => apair

isPrimitive : Word -> Bool
isPrimitive w = let (p,n) = prefixPowerOf w in
                    p == w

isPower : Word -> Bool
isPower w = not (isPrimitive w)

isPrimitiveP1 : isPrimitive "dodo" = False
isPrimitiveP1 = Refl


isPrimitiveP2 : isPrimitive "door" = True
isPrimitiveP2 = Refl

--TODONOTE could make a circular shift dat structures

cyclicShift : List a -> List a
cyclicShift [] = []
cyclicShift (x :: xs) = xs ++ [x]

cyclicShiftWord : Word -> Word
```

```
cyclicShiftWord w = pack . cyclicShift $ unpack w

--TODONOTE TODO iterateN is a handy utility function
iterateN : Nat -> (f : a -> a) -> (x : a) -> List a
iterateN Z     f x = []
iterateN (S n) f x = x :: iterateN n f (f x)

allCyclicShiftsOf : Word -> List Word
allCyclicShiftsOf w = iterateN (length w) cyclicShiftWord w

isConjugateOf : Word -> Word -> Bool
isConjugateOf w1 w2 = any (== w2) (allCyclicShiftsOf w1)

isConjugateOfP1 : isConjugateOf "listen" "enlist" = True
isConjugateOfP1 = Refl




isPrimitiveT : Word -> Type
isPrimitiveT x = isPrimitive x = True


isPrimitiveDec : (w : Word) -> Dec (isPrimitiveT w)
isPrimitiveDec w with (isPrimitive w)
  | True  = Yes Refl
  | False = No falseNotTrue

isPowerT : Word -> Type
isPowerT x = isPower x = True


isPowerDec : (w : Word) -> Dec (isPowerT w)
isPowerDec w with (isPower w)
  | True  = Yes Refl
  | False = No falseNotTrue
```

```
isConjugateOfT : Word -> Word -> Type
isConjugateOfT x x1 = isConjugateOf x x1 = True

isConjugateOfDec : (w1 : Word) -> (w2 : Word) -> Dec (isConjugateOfT w1 w2)
isConjugateOfDec w1 w2 with (isConjugateOf w1 w2)
  | True  = Yes Refl
  | False = No falseNotTrue


nthConjugateOf : Nat -> Word -> Word
nthConjugateOf Z x = x
nthConjugateOf (S k) x = nthConjugateOf k (cyclicShiftWord x)

cyclicShiftIsConjugate : (w1 : Word) -> (w2 : Word) ->
                         { auto ok: cyclicShiftWord w1 = w2 } ->
                                    isConjugateOfT w1 w2
cyclicShiftIsConjugate w1 w2 = ?cyclicShiftIsConjugate_pf

data Power : Word -> Word -> Nat -> Type where
  MkPower : (w : Word) -> (p : Word) -> (n : Nat) ->
            { auto ok: wordMult p n = w } -> Power w p n


data Conjugate : Word -> Word -> Nat -> Type where
  MkConjugate : (w : Word) -> (x : Word) -> (n : Nat) ->
                { auto ok: nthConjugateOf n x = w } -> Conjugate w x n


splitConjugate : (Conjugate w x n) -> {u : Word} -> {v : Word} ->
                        ((Word, Word) , ((w = u ++ v), (x = v ++ u)))
splitConjugate (MkConjugate w x n ) = ?splitConjugateProof
```

```
theorem2_4_2DT : (w : Word) -> (x : Word) -> (Conjugate w x n) ->
                 (Power w pw kw) -> wordMult px kx = x
theorem2_4_2DT w x (MkConjugate w x n) (MkPower w pw kw) = ?thm242Proof
```

A.9 SignedConsCellWord

```
module SignedConsCellWord

%default total

-- Data Types
%elim
data SignedChar : Type where
  Pos : Char -> SignedChar
  Neg : Char -> SignedChar


%name SignedChar c,x,y,z


infixr 8 #
%elim
data Word = Empty | (#) SignedChar Word

%name Word w, w2, w3, w4

-- Equivalence

instance Eq SignedChar where
  (Pos x) == (Neg y) = False
  (Neg x) == (Pos y) = False
  (Pos x) == (Pos y) = x == y
  (Neg x) == (Neg y) = x == y

  x /= y = not (x == y)
```

```
instance Eq Word where
  (a # b) == Empty   = False
  Empty   == (a # b) = False
  Empty   == Empty   = True
  (a # b) == (c # d) = (a == c) && (b == d)

  x /= y             = not (x == y)

-- Methods
signedCharInverse : SignedChar -> SignedChar
signedCharInverse (Pos x) = Neg x
signedCharInverse (Neg x) = Pos x


wordLength : Word -> Nat
wordLength Empty = Z
wordLength (x # y) = S (wordLength y)


wordSignedFromCharList : (Char -> SignedChar) -> List Char -> Word
wordSignedFromCharList f [] = Empty
wordSignedFromCharList f (x :: xs) = (f x) #
                                (wordSignedFromCharList f xs)

wordFromCharList : List Char -> Word
wordFromCharList xs = wordSignedFromCharList Pos xs

wordInverseFromCharList : List Char -> Word
wordInverseFromCharList xs = wordSignedFromCharList Neg xs


wordFromString : String -> Word
wordFromString x = wordFromCharList (unpack x)

wordInverseFromString : String -> Word
```

```
wordInverseFromString x = wordInverseFromCharList (unpack (reverse x))

stringFromWord : Word -> String
stringFromWord Empty = ""
stringFromWord ((Pos x) # y) = (singleton x) ++ stringFromWord y
stringFromWord ((Neg x) # y) = (singleton x) ++ stringFromWord y

wordConcat : Word -> Word -> Word
wordConcat Empty x1 = x1
wordConcat (x # y) x1 = x # (wordConcat y x1)

isInverseOf : SignedChar -> SignedChar -> Bool
isInverseOf (Pos _) (Pos _) = False
isInverseOf (Neg _) (Neg _) = False
isInverseOf (Neg x) (Pos y) = x == y
isInverseOf (Pos x) (Neg y) = x == y

%assert_total
wordCollapseOneLevel : Word -> Word
wordCollapseOneLevel Empty = Empty
wordCollapseOneLevel (x # Empty) = x # Empty
wordCollapseOneLevel (x # (y # z)) = case (x `isInverseOf` y) of
  True  => wordCollapseOneLevel z
  False => x # wordCollapseOneLevel (y # z)

%assert_total
wordCollapse : Word -> Word
wordCollapse x = let y = wordCollapseOneLevel x in
                     case (x == y) of
                          True  => x
                          False => wordCollapse y


wordConcatAndCollapse : Word -> Word -> Word
wordConcatAndCollapse w1 w2 = let w = wordConcat w1 w2 in
                                  wordCollapse w
```

```
wordSignedCharAtMaybe : Nat -> Word -> Maybe SignedChar
wordSignedCharAtMaybe Z Empty    = Nothing
wordSignedCharAtMaybe Z (x # xs) = Just x
wordSignedCharAtMaybe (S k) Empty = Nothing
wordSignedCharAtMaybe (S k) (x # xs) = wordSignedCharAtMaybe k xs

wordInverse : Word -> Word
wordInverse Empty = Empty
wordInverse (x # y) = wordConcat (wordInverse y)
  (signedCharInverse (x) # Empty)

isPrefixOf : Word -> Word -> Bool
isPrefixOf Empty w2 = True
isPrefixOf (x # y) Empty = False
isPrefixOf (x # y) (z # w) = (x == z) && (isPrefixOf y w)

-- Semigroup Properties

instance Semigroup Word where
  x <+> y   =  wordConcatAndCollapse x y

wordInduction : (P : Word -> Type)  -> -- Property to show
    (P Empty)           -> -- Base case
    ((c : SignedChar) -> (w : Word) ->  P w -> P (c # w)) -> -- Step
    ( a : Word ) ->        -- Show for all a
    P a
wordInduction P p_Empty p_Concat Empty   = p_Empty
wordInduction P p_Empty p_Concat (c # w) =
  p_Concat c w (wordInduction P p_Empty p_Concat w)

instance Monoid Word where
  neutral = Empty
```

```
wordConcatEmptyRightNeutral : (w : Word) -> wordConcat w Empty = w
wordConcatEmptyRightNeutral Empty = Refl
wordConcatEmptyRightNeutral (c # w) =
  let inductiveHypothesis = wordConcatEmptyRightNeutral w in
      ?wordConcatEmptyRightNeutralStepCase


wordConcatAndCollapseEmptyRightNeutral : (w : Word) ->
                        wordConcatAndCollapse w Empty = w
wordConcatAndCollapseEmptyRightNeutral Empty = Refl
wordConcatAndCollapseEmptyRightNeutral (c # w) =
  let inductiveHypothesis = wordConcatAndCollapseEmptyRightNeutral w in
                        ?wordConcatAndCollapseEmptyRightNeutralStepCase

instance Group Word where
  inverse = wordInverse

wordInverseIsGroupInverseL : (l : Word) ->
                             wordConcat l (wordInverse l) = Empty
wordInverseIsGroupInverseL Empty = Refl
wordInverseIsGroupInverseL (c # w) =
  let inductiveHypothesis = wordInverseIsGroupInverseL w in
      ?wordInverseIsGroupInverseLStepCase

instance VerifiedGroup Word where
  groupInverseIsInverseL = wordInverseIsGroupInverseL
  groupInverseIsInverseR = ?wordInverseIsGroupInverseR


---------- Proofs ----------

SignedConsCellWord.wordConcatEmptyRightNeutralStepCase = proof
  intros
  rewrite inductiveHypothesis
  trivial
```

A.10 ConsCellWord

```
module ConsCellWord

import Prelude.Functor
import CharDec

%default total
-- Data types
infixr 8 #
data Word t = Empty | (#) t (Word t)

infixr 2 ##
(##) : Word t -> Word t -> Word t
(##) Empty w     = w
(##) (c # w) w2 = c # (w ## w2)



-- Instances
instance (Eq t) =>  Eq (Word t) where
  Empty   == Empty   = True
  (a # x) == Empty   = False
  Empty   == (b # y) = False
  (a # x) == (b # y) = (a == b) && (x == y)


instance Functor Word where
  map f Empty   = Empty
  map f (c # w) = (f c) # (map f w)


instance Semigroup (Word t) where
  (<+>) = (##)

instance Monoid (Word t) where
```

```
  neutral = Empty

instance (Show t) => Show (Word t) where
  show xs = "(" ++ show' "" xs ++ ")" where
    show' acc Empty       = acc
    show' acc (c # Empty) = acc ++ show c
    show' acc (c # w)     = show' (acc ++ show c ++ " # ") w

||| A tail recursive right fold on Words
total foldrImpl : (t -> acc -> acc) -> acc -> (acc -> acc) ->
                  Word t -> acc
foldrImpl f e go Empty = go e
foldrImpl f e go (c # w) = foldrImpl f e (go . (f c)) w

instance Foldable Word where
    foldr f e xs = foldrImpl f e id xs



instance Applicative Word where
  pure x = x # Empty

  fs <$> vs = concatMap (\f => map f vs) fs


instance Traversable Word where
  traverse f Empty = pure Empty
  traverse f (c # w) = [| (#) (f c) (traverse f w) |]


instance Monad Word where
   m >>= f = concatMap f m


instance Alternative Word where
  empty = Empty
```

```
  (<|>) = (##)


-- Proofs and properties
wordInduction : (P : (Word t) -> Type)  -> -- Property to show
  (P Empty)               -> -- Base case
  ((c : t) -> (w : (Word t)) ->  P w -> P (c # w)) -> -- Step
  ( a : (Word t)) ->        -- Show for all a
  P a
wordInduction P p_Empty p_Concat Empty   = p_Empty
wordInduction P p_Empty p_Concat (c # w) =
  p_Concat c w (wordInduction P p_Empty p_Concat w)


wordConcatIsAssociative : (l : Word t) -> (c : Word t) ->
          (r : Word t) -> (l ## (c ## r)) = ((l ## c) ## r)
wordConcatIsAssociative Empty c r = Refl
wordConcatIsAssociative (x # y) c r =
  let inductiveHypothesis = wordConcatIsAssociative y c r in
      ?wordConcatIsAssociativeStepCase

instance VerifiedSemigroup (Word t) where
  semigroupOpIsAssociative = wordConcatIsAssociative


wcN : (w : Word t) -> (w ## Empty) = w
wcN Empty = Refl
wcN (x # y) = let iH = wcN y in
                  ?theRest1


wordConcatNeutralIsNeutralL : (w : (Word t)) -> (w ## Empty) = w
wordConcatNeutralIsNeutralL Empty = Refl
wordConcatNeutralIsNeutralL (x # y) =
  let inductiveHypothesis = wordConcatNeutralIsNeutralL y in
      ?wordConcatNeutralIsNeutralLStepCase
```

```
wordConcatNeutralIsNeutralR : (w : (Word t)) -> (Empty ## w) = w
wordConcatNeutralIsNeutralR Empty = Refl
wordConcatNeutralIsNeutralR (x # y) =
  let inductiveHypothesis = wordConcatNeutralIsNeutralR y in
      ?wordConcatNeutralIsNeutralRStepCase


instance VerifiedMonoid (Word t) where
  monoidNeutralIsNeutralL = wordConcatNeutralIsNeutralL
  monoidNeutralIsNeutralR = wordConcatNeutralIsNeutralR


wordConcatIsInvInjective : (c : t) -> (w1 : (Word t)) ->
                (w2 : (Word t)) -> (pf : w1 = w2 ) -> c # w1 = c # w2
wordConcatIsInvInjective c w1 w2 pf = ?wordConcatIsInvInjective_rhs

wordConcatIsInjective : (c : t) -> (w1 : (Word t)) ->
    (w2 : (Word t)) -> {auto pf : c # w1 = c # w2 } -> w1 = w2
wordConcatIsInjective _ _ _ {pf = Refl} = Refl


functorIdentityProof :  (x : Word t) -> map id x = x
functorIdentityProof Empty = Refl
functorIdentityProof (x # y) =
  let inductiveHypothesis = functorIdentityProof y in
      ?functorIdentityProofStepCase

functorCompositionProof : (x : Word a) -> (g1 : a -> b) ->
    (g2 : b -> c) -> map (g2 . g1) x = map g2 (map g1 x)
functorCompositionProof Empty g1 g2 = Refl
functorCompositionProof (x # y) g1 g2 =
  let inductiveHypothesis = functorCompositionProof y g1 g2 in
      ?functorCompositionProofStepCase

instance VerifiedFunctor Word where
```

```
  functorIdentity = functorIdentityProof
  functorComposition = functorCompositionProof


applicativeMapProof : (x : Word a) -> (g : a -> b) ->
                      map g x = ((map g x) ## Empty)
applicativeMapProof Empty g = Refl
applicativeMapProof (x # y) g =
  let inductiveHypothesis = applicativeMapProof y g in
      ?applicativeMapProofStepCase

applicativeIdentityProof : (x : Word a) ->  ((map id x) ## Empty) = x
applicativeIdentityProof Empty = Refl
applicativeIdentityProof (x # y) =
  let inductiveHypothesis = applicativeIdentityProof y in
      ?applicativeIdentityProofStepCase



applicativeCompositionProof : (x : Word a) -> (g1 : Word (a -> b)) ->
    (g2 : Word (b -> c)) ->
    ((pure (.) <$> g2) <$> g1) <$> x = g2 <$> (g1 <$> x)
applicativeCompositionProof Empty g1 g2 =
  ?applicativeCompositionProofBaseCase
applicativeCompositionProof (x # y) g1 g2 =
  ?applicativeCompositionProof_rhs_2

applicativeHomomorphismProof : (x : a) -> (g : a -> b) ->
                               ((g x) # Empty) = ((g x) # Empty)
applicativeHomomorphismProof x g = Refl


applicativeInterchangeProof : (x : a) -> (g : Word (a -> b)) ->
              g <$> pure x = pure (\g' : a -> b => g' x) <$> g
applicativeInterchangeProof x g = ?applicativeInterchangeProof_rhs_1
```

```
instance VerifiedApplicative Word where
  applicativeMap = applicativeMapProof
  applicativeIdentity = applicativeIdentityProof
  applicativeComposition = applicativeCompositionProof
  applicativeHomomorphism = applicativeHomomorphismProof
  applicativeInterchange = applicativeInterchangeProof

instance VerifiedMonad Word where
  monadApplicative = ?monadApplicativeProof
  monadLeftIdentity = ?monadLeftIdentityProof
  monadRightIdentity = ?monadRightIdentityProof
  monadAssociativity = ?monadAssociativity

---------- Proofs ----------

ConsCellWord.theRest1 = proof
  intros
  rewrite iH
  trivial


ConsCellWord.wordConcatNeutralIsNeutralLStepCase = proof
  intros
  rewrite inductiveHypothesis
  trivial


ConsCellWord.applicativeIdentityProofStepCase = proof
  intros
  rewrite inductiveHypothesis
  trivial


ConsCellWord.applicativeMapProofStepCase = proof
  intros
  rewrite inductiveHypothesis
```

```
  trivial


ConsCellWord.functorCompositionProofStepCase = proof
  intros
  rewrite inductiveHypothesis
  trivial


ConsCellWord.functorIdentityProofStepCase = proof
  intros
  rewrite inductiveHypothesis
  trivial



ConsCellWord.wordConcatIsInvInjective_rhs = proof
  intros
  rewrite pf
  trivial


ConsCellWord.wordConcatNeutralIsNeutralRStepCase = proof
  intros
  trivial


ConsCellWord.wordConcatIsAssociativeStepCase = proof
  intros
  rewrite inductiveHypothesis
  trivial
```

A.11 Printf and Scanf

```
-- Printf code based on:
-- https://gist.github.com/puffnfresh/11202637
-- written by:
-- Brian McKenna, https://www.youtube.com/watch?v=fVBck2Zngjo

-- Scanf code by Ben Goodspeed
module Printf

%default total

data Format = FInt Format -- %d
            | FString Format -- %s
            | FOther Char Format -- [a-zA-Z0-9]
            | FEnd --

format : List Char -> Format
format ('%'::'d'::cs) = FInt (format cs)
format ('%'::'s'::cs) = FString (format cs)
format (c::cs) = FOther c (format cs)
format [] = FEnd

interpFormat : Format -> Type
interpFormat (FInt f) = Int -> interpFormat f
interpFormat (FString f) = String -> interpFormat f
interpFormat (FOther _ f) = interpFormat f
interpFormat FEnd = String

formatString : String -> Format
formatString s = format (unpack s)

toFunction : (fmt : Format) -> String -> interpFormat fmt
toFunction (FInt f) a = \i => toFunction f (a ++ show i)
toFunction (FString f) a = \s => toFunction f (a ++ s)
toFunction (FOther c f) a = toFunction f (a ++ singleton c)
```

```
toFunction FEnd a = a

printf : (s : String) -> interpFormat (formatString s)
printf s = toFunction (formatString s) ""

data SFormat = SFInt SFormat
             | SFOther Char SFormat
             | SFString SFormat
             | SFEnd

data Result : Type where
  MkIntResult : Int -> Result
  MkCharResult : Char -> Result
  MkStringResult : String -> Result

sformat : List Char -> SFormat
sformat ('%' :: 'd' :: cs) = SFInt (sformat cs)
sformat ('%' :: 's' :: cs) = SFString (sformat cs)
sformat (c :: cs) = SFOther c (sformat cs)
sformat [] = SFEnd

interpSFormat : SFormat -> Type
interpSFormat (SFInt f) =  Int -> interpSFormat f
interpSFormat (SFOther _ f) =  interpSFormat f
interpSFormat (SFString f) =  String -> interpSFormat f
interpSFormat SFEnd = List Result

sformatString : String -> SFormat
sformatString x = sformat (unpack x)


toSFunction : (fmt : SFormat) -> (List Result) ->
              interpSFormat fmt
toSFunction (SFInt f)  acc i =
  \i => toSFunction f  ((MkIntResult i) :: acc)
toSFunction (SFOther c f)  acc =
```

```
  toSFunction f  ((MkCharResult c) :: acc)
toSFunction (SFString f)  acc  =
  \s => toSFunction f  ((MkStringResult s) :: acc)
toSFunction SFEnd  acc  = acc

sscanf : (s : String) -> interpSFormat (sformatString s)
sscanf s = toSFunction (sformatString s) []
```