Verification Techniques for Low-Level Programs

Samuel D. Pollard

February 21, 2019

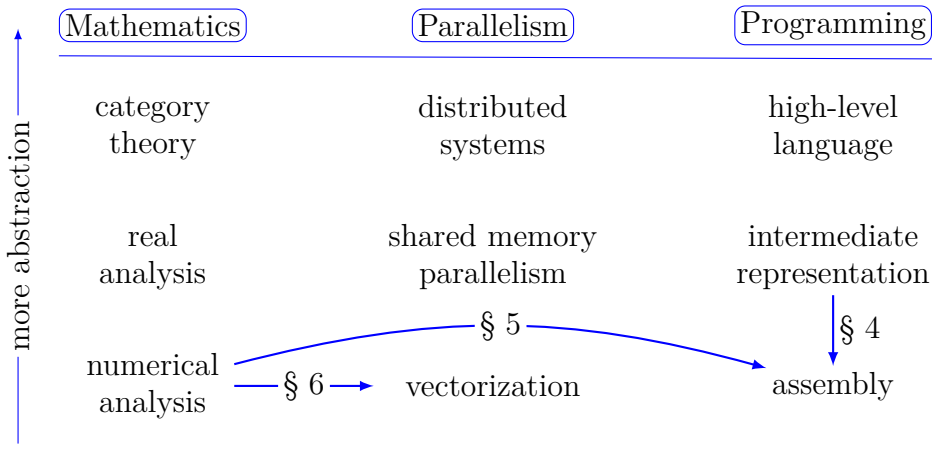

Figure 1: An overview of proposed contributions. $x \to y$ means we use x to reason about y.

With a sudden kiss or a fiery stare
Don't mess up my programs of love
I've input hellos and goodbyes so neatly
Everything comes to an end in due time

Plastic Love, Mariya Takeuchi, 1984

but also a proof assistant's type-checker, probably

Abstract

We explore the application of highly expressive logical and automated reasoning techniques to the analysis of computer programs. We begin with an introduction to formal methods by describing different approaches and the strength of the properties they can guarantee. These range from static analyzers, SMT solvers, deductive program provers, and proof assistants. We then explore applications of formal methods to the analysis of intermediate representations, verification of floating point arithmetic, and fine-grained parallelism such as vectorization. Throughout, we focus on verification techniques applied to programs written at the lowest level of abstraction including binary, bytecode, and assembly languages.

1 Verification of Computation

Verification of the behavior of a computation predates electronic computers. One reasonable starting point is 1940 with Alonzo Church's invention of simply typed lambda calculus (STLC). At first, STLC was invented to prevent paradoxes like Curry's paradox. Further study of STLC inspired computer scientists and mathematicians to develop increasingly powerful type theory. Programming languages with these features not only help avoid logical paradoxes and prevent common programming errors but also have deep connections with formal proofs. By the Curry-Howard isomorphism, proof systems in intuitionistic logic—a logic in which all proofs are constructive and thus computable—correspond exactly to typed models of computation [132].

Alongside the exponential increase in the complexity of computers, two interesting phenomena occurred. The first is the increasing abstraction computer scientists developed to manage this complexity. The second is the development of sophisticated type systems, logics, and proof systems such as the Calculus of Inductive Constructions (CIC), separation logic, and sequent calculus [143]. These allow incredibly expressive logical specifications of program behavior and machine-checked proofs that implementations match these specifications.

Despite this move towards the highly abstract and expressive, we cannot escape our roots: the low-level machine languages upon which computers operate. Ultimately, there is still need for computer programs which are written at the lowest level of abstraction: bootloaders, performance-critical code, and compiler optimizations are performed on languages which interact directly with microprocessor hardware. Additionally, the increasing het-

erogeneity of hardware with the advent of GPUs, FPGAs, and other accelerators brings with it the need for more low-level code. This hardware often has vastly different behavior and performance characteristics which makes correct and well-performing code difficult to write—and correct, well-performing, portable code nearly impossible.

This survey presents the most common formal methods techniques and how they are used to verify low-level programs. We emphasize that verifying any program hinges on a description of what we wish to verify—the *specification*. In practice, these specifications range from informal English descriptions (for example, "This program shall not crash on any input") to more formal English specifications (such as IETF RFCs or ISO standards) to machine-readable and machine-checkable specifications.

Ultimately, all of formal methods fall into two domains. These are so central to the field we distinguish them typographically:

Definition 1. *Formal specification: writing precise, unambiguous descriptions of what we want a program to do.*

Definition 2. *Formal verification: how we prove the program is correct with respect to this specification.*

Verifying any property of a program requires both domains. We can think of specification as the "front end" or user-facing part of formal methods while verification is the "back end."

Another bifurcation of formal methods is *model checking* versus *correct by construction programming* (CBCP). With model checking, the desired program behavior is described using a specification language such as TLA+[97], F* [134]. Correct by construction programming begins with the user developing in a language with features that allow the generation and dispatch of proofs of correctness. Examples include Coq [136], HOL/Isabelle [113], and Agda [114]. We typically get a higher level of assurance with CBCP at the cost of rewriting an application in a different language. In research, nothing is ever so cut and dried so some approaches a blend of model checking and CBCP.

To understand how one may go about verifying computation, this section continues with a few canonical papers by Hoare [87], Dijkstra [61], and Reynolds [122] which lay the foundations of proving the correctness of computer programs. Next is a discussion on proof assistants (§ 2). We then describe a workflow for formal methods and some tools and languages for model checking and CBCP in § 3.

The main portion of this survey describes how these verification techniques are applied to intermediate representations (IRs), assembly language programs, floating point arithmetic, and low-level parallelism such as Single-Instruction, Multiple Data (SIMD) parallelism (§ 4–6).

We conclude the survey with some references to formal methods' applications and adoption (§ 7) and with a brief description of potential future research directions.

1.1 Logical Notation

We provide a brief introduction to logic chiefly to establish a consistent notation. An excellent, more comprehensive introduction is given in the Stanford Encyclopedia of Philosophy [128].

A *predicate* or *formula* is a function mapping from some domain into the set {True, False}. An example of a domain is the set of all pairs of integers ($\mathbb{Z} \times \mathbb{Z}$) which an accompanying predicate may be $P(x, y) = x + y > 0$. Here, x and y are *variables* and may only be assigned values from their appropriate sets.

Now that we have defined predicates, the next natural step is reasoning about how to combine and modify them. A branch of logic which accomplishes this is *propositional logic*. Predicates in propositional logic may only contain variables and the following logical operators. These are also called *connectives* because they connect one or two objects.

- and ($\wedge$)
- or ($\vee$)
- exclusive or (xor)
- implies ($\implies$)
- if and only if ($\iff$)
- not ($\neg$)

First-order logic contains all of propositional logic with the addition of quantifiers:

- existential ($\exists$)
- universal ($\forall$)

These quantifiers may only be applied to objects in the domain. For example, $\forall x, y \in \mathbb{R},\ x + y = y + x$ is a valid first-order predicate. We sometimes use a period to separate the statements of an existential quantifier which reads as "such that."

Second-order logic allows these quantifiers to be applied not just to objects in the domain but to properties (that is, formulas consisting only of logical connectives, quantifiers, and objects). For example, suppose our domain is the set of pairs of real numbers ($\mathbb{R} \times \mathbb{R}$) and we wish to say there is some property that holds regardless of how you pass in the arguments.

$$\exists P(x, y).\ \forall x, y\ P(x, y) \iff P(y, x) \tag{1}$$

where the first $\exists$ is the second-order quantifier. An example of such a P is the statement we can always add two reals to get a third:

$$P(x, y) := \forall x, y\ \exists z.\ x + y = z.$$

We know P is true by commutativity of $\mathbb{R}$, and thus we can use P to prove (1).

A non-example is division; $0/1$ is defined but $1/0$ is not. Namely,

$$Q(x, y) := \forall x, y\ \exists z.\ x/y = z$$

is false.

Higher-order logic includes second-order logic, third order logic, fourth, etc. Third order logic, for example, would allow quantifiers to be applied to properties about properties.

With propositional, first-order, and higher-order logic expressiveness increases while the ability to reason about the logic decreases. For example, specifying the rules of chess using first-order logic requires one page of definitions, whereas using propositional logic requires 100,000 pages [125]. Conversely, we can decide the satisfiability of any propositional logic formula, but in general, this is impossible for first-order logic.

The logics we see next form the basis of formal methods and all works throughout use one or several of them.

1.2 Hoare Logic

Hoare logic [87] is the basis for deductive verification. Hoare logic lays out a scheme for specifying programs via the notation

$$P\{Q\}R$$

where P is a predicate called a *precondition*, Q is a program, and R is a predicate called the *postcondition*. These are referred to as *Hoare triples*. For example, the axiom which allows sequencing of statements (a semicolon in many languages) is described as

$$\text{If } \vdash P\{Q_1\}R_1 \text{ and } \vdash R_1\{Q_2\}R \text{ then } P\{Q_1; Q_2\}R.$$

Here $\vdash$ is read as "entails." The fact that nothing precedes $\vdash$ means the Hoare triples require no prior assumptions (such as no initialization of variables). In this fashion, we may specify transformer semantics for all of a given language's features.

Building upon Hoare logic, Dijkstra provided a scheme of predicate transformation so given a program Q and a postcondition R, one could compute its *weakest precondition* $wp(Q, R)$ such that any other precondition P implies $wp(Q, R)$ [61]. For example, the sequence of two statments (separated by a semicolon) are defined as $wp(S_1; S_2, R) = wp(S_1, wp(S_2, R))$.

1.3 Separation Logic

Reynolds' work describes *Separation Logic*, an extension of Hoare Logic which allows succinct proofs and descriptions of correctness properties [122]. The primary ideas in separation logic are separating connectives which describe the heap memory of a program (as opposed to stack memory).

Separation logic is based on three primary concepts. Throughout, p and q represent assertions which may describe things such as the value at a given memory location.

1. Pointers into the heap, $x \mapsto \text{val}$. Dereferencing is denoted by $[x]$ and null pointers by $\emptyset$.[1]
2. Separating conjunction, $p * q$. This behaves similarly, but not equivalently, to logical conjunction $\wedge$.
3. Separating implication, $p \mathrel{-\!\!*} q$. This behaves similarly to logical implication $\implies$.

The separating conjunction says for some assertions p and q, $p * q$ means we can split the heap into two disjoint parts such that p holds for one and q holds for the other. The separating implication means if p holds for some heap and we extend the heap with a disjoint part (think `malloc`), then q holds for that extended heap.

One can think of $*$ as decomposing the existing heap whereas $\mathrel{-\!\!*}$ relates to adding new resources to the heap. For example, suppose we have some assertion which holds given a heap with a single variable x which points to the value 12. This is expressed

$$(x \mapsto 12) \mathrel{-\!\!*} p.$$

Examples of valid p are $x \neq \emptyset$ or $x = 12$. Now, we can begin to see the usefulness of such a logic by considering that a heap in a previous, different state can "become valid" by assigning x to the correct value:

$$x \mapsto 9 * ((x \mapsto 12) \mathrel{-\!\!*} p).$$

A Hoare triple for such a statement might be

$$\{x \mapsto 9 * ((x \mapsto 12) \mathrel{-\!\!*} p)\}\ [x] := 12\ \{p\}.$$

This just scratches the surface of separation logic. Importantly, this logic has been proven sound and has been widely extended. One example is higher-order separation logic [35]. Additionally, researchers at INRIA encoded subsets of separation logic in first-order logic to allow integration with existing specification languages [21].

1.4 Abstract Interpretation

Abstract interpretation was invented by Radhia and Patrick Cousot [49] in 1977. Abstract interpretation at its core describes a family of Galois connections between partially ordered sets (posets). Some examples of Galois connections are crammed into a paper by Cousot & Cousot [50]. The theoretical specifics are not necessary for our discussion; instead, we can think of abstract interpretation as "sound static analysis." By static analysis, we mean discovering properties of a program without executing it. Sound means that any property discovered by the static analyzer will also hold when executing the program—no false positives.

The key to abstract interpretation is choosing an *abstract domain* over which to analyze. An example of an

[1] The original separation logic paper denotes a null pointer as $-$ but this is confusing to read alongside $\mathrel{-\!\!*}$ and $*$

abstract domain over the floating point numbers would be to consider every floating point value as either zero, nonzero, infinite, or Not-a-Number (NaN). Thus the abstract domain is the set

$$\mathcal{A} = \{0,\ \text{nnz},\ \pm\infty,\ \text{NaN},\ \text{flt}\}. \tag{2}$$

We require the last element "flt" to encompass "any floating point value" because for some operations we cannot know for certain what the value will be. For example, $f(x) = 1.0/x$ will not return finite values for all x.

With this abstract domain, we could check if a floating point operation "behaves nicely," depending on our definition of nice. One example might be a division operation which always returns a finite value. Semantically, there may not be much use for such a function but that is a different issue.

To use (2) as our abstract domain, we must calculate the abstract valuation for the operations we care about. For this survey, we choose only division and show the valuation in Table 1.

Table 1: Abstract valuation for division. The column is the numerator and the row is the denominator. We bold the column used in Figure 2.

/	0	**nnz**	NaN	$\pm\infty$	flt
0	NaN	$\pm\infty$	NaN	$\pm\infty$	flt
nnz	flt	**flt**[1]	NaN	$\pm\infty$	flt
NaN	NaN	**NaN**	NaN	NaN	NaN
$\pm\infty$	0	**0**	NaN	NaN	flt
flt	flt	**flt**	NaN	flt	flt

[1] For a numerator too close to 0, underflow to 0 can occur. For a denominator too close to 0, overflow to $\pm\infty$ can occur.

To see the use of the abstract domain $\mathcal{A}$, consider the functions in Listing 2. Each is attempting to compute $f(x) = 1.0/x$ with the additional goal that f returns either $1.0/x$ if finite, or 0.0 in any other case. We notice only the second column, nnz, is necessary since the numerator is always 1.0. Therefore, we must consider several cases to ensure f returns a finite value. The first, `unsafe`, only considers $x = 0.0$. An abstract interpreter explores both branches of the conditional and notices in the `else` branch it cannot narrow down the scope of x sufficiently, thus will state that `unsafe` will return some flt.

For `mostlysafe`, we consider the three edge cases for x: 0, NaN, $\pm\infty$. These all behave correctly, but for some nnz we still cannot ensure that $1.0/x$ is finite because the reciprocal of sufficiently small floating point numbers can overflow to $\pm\infty$. Therefore, `mostlysafe` will also return flt.

This is somewhat of an unsatisfying result. We present this example because, despite its relative simplicity, we still require a more complex abstract domain to prove correctness. We discuss the subtleties of floating point arithmetic in further detail in § 5. But fear not, to get

```c
#include <math.h>
float unsafe(float x) {
  if (x==0.)
    return 0.;
  else
    return 1. / x;
}
```

```c
#include <math.h>
float mostlysafe(float x) {
  if (x==0. ||
      isnan(x) || isinf(x))
    return 0.;
  else
    return 1. / x;
}
```

```c
#include <math.h>
float reallysafe(float x) {
  // Cast to int without changing bits
  unsigned long c = *(unsigned long*)&x;
  if (isnan(x) || isinf(x) ||
     (0x80000000 <= c && c <= 0x80200000) ||
     (0x00000000 <= c && c <= 0x00200000))
    return 0.;
  else
    return 1. / x;
}
```

Figure 2: Three functions over which an abstract domain may be applied. However, `reallysafe` is the only one guaranteed to always return a finite value.

some closure we include `reallysafe` which always returns a finite value and returns the floating point approximation $1/x$ for exactly the set of x which do not overflow. We found these tight bounds through an exhaustive search of the input space.

There is interest using abstract interpretation over floating point operations for reproducible BLAS [60] in applications such as medical diagnosis and legal cases.

1.5 Symbolic Execution

Symbolic execution is based on the idea that in a program variables can represent logical formulas rather than concrete values [27, 40]. Slightly more formally, symbolic execution for a variable of type τ generates some expression representing how values of τ are transformed by a given program or function. Symbolic execution is a form of static analysis which generates formulas in propositional logic to be dispatched to SMT solvers.

Symbolic execution is a popular technique with dozens of packages implementing techniques ranging from pure symbolic execution to a mix of concrete and symbolic (concolic) execution to generate test cases for a given program. A survey of symbolic execution techniques is given by Baldoni et al. [9].

Some popular symbolic execution engines include Angr [129], primarily designed for security, and KLEE [34], primarily designed for finding bugs.

Figure 3 shows a conceptual example of how symbolic execution proceeds. We begin by mapping x and y to symbolic values A and B, which represent any possible

input. At the `if` statement at line 2, there is a branch in the symbolic executor which splits the execution into two possibilities which it must then explore completely. The right branch is easy because we are done with the program and no assertions have failed. The left branch is more interesting. Consider line 4, the statement `y = x - y;`. Symbolically, our previous state is

$$x \mapsto A + B,\ y \mapsto B$$

so subtracting $x - y$ is symbolically equivalent to $A + B - B = A$. This sort of transformation is the core of symbolic execution. Proceeding down the tree and likewise the program, we reach the single assertion on line 6. Substituting the symbolic values in for x and y we get an infeasible (unreachable) path for $B - A > 0$ and feasible (reachable) otherwise. This allows us to verify the assertion will never fail because the only state in which it does fail is unreachable since $A > B$ being true at the higher branch implies $B - A > 0$ is false at the lower branch.

One limitation of symbolic execution is the *state space explosion* problem. This arises from the fact that symbolic execution explores all potential execution paths. This is simple in Figure 3 but for programs with unbounded loops it is impossible to completely explore all paths. Even for relatively simple codes, full symbolic execution may be intractable. Thus, researchers may maximize code coverage without exploring all paths or transforming programs to reduce the number of branches.

1.6 SMT: Satisfiability Modulo Theories

SMT solvers are to formal methods as matrix multiplication is to scientific computing: many problems can be expressed with respect to these algorithms, much work has gone into optimizing them, and they are often the most computationally expensive part of a larger algorithm.

Before we get too far ahead of ourselves, we will break down SMT. *Satisfiability* relates to the fact that the output of an SMT solver is essentially "yes" or "no" with the caveat that if "yes" the user may want to know *how* to satisfy the formulas. *Modulo Theories* means the question of satisfiability is identical but different theories can be substituted in terms. For example, typical SMT solvers have theories for integers, lists, arrays, and even floating point numbers.

For example, if our SMT solver understands the theory of integer arithmetic then

$$x + 2y = 20 \wedge x - y = 2$$

is satisfiable because both equations are true for $x = 8$ and $y = 6$. A non-example is

$$x > 0 \wedge y > 0 \wedge x + y < 0.$$

A good overview of SMT is given by Barrett [16].

Once a problem is reduced to a question of satisfiability, there is often a standardized API called SMTLIB [15] upon which a solver can take over and compute the satisfiability of a given formula.

Some popular SMT Solvers include CVC4 [14], Z3 [59], Alt-Ergo [94], and OpenSMT [32].

1.7 Summary

Hoare logic, separation logic, abstract interpretation, symbolic execution, and SMT solvers are the building blocks of formal methods. Every modern paper regarding verification is influenced by at least one of these methods. In the following section we see computer scientists' efforts towards a more complete formalization of mathematics via proof assistants.

2 Proof Assistants

Attempting to cover all of type theory in a couple pages is a silly endeavor; we instead focus on a few type theory concepts commonly used in proof assistants. This section is a compilation of material from Robert Harper's [81] and Benjamin Pierce's [118] books on type theory and Barendregt's taxonomy of lambda calculi [11]. Namely, we discuss the λ cube (a.k.a. Berendregt's cube) with special attention on dependent types. We mention particular proof assistants but provide a more comprehensive taxonomy in § 3.2.4.

The λ cube shown in Figure 4 describes three dimensions of the many levels of abstraction in typed programming languages. Additionally, most proof assistants are total; they do not allow nonterminating programs and thus are a useful design space to ensure computable proofs. We describe the bottom of the cube as the edge representing our old friend the simply typed lambda calculus, denoted STLC or $\lambda^{\rightarrow}$. At the top is the Calculus of Constructions (CoC), the language upon which Coq is based.

In the STLC, values and types are separate and are never interchangeable. For example, a function which returns its second argument is written

$$\lambda(x : \tau_1).\ \lambda(y : \tau_2).\ y \tag{3}$$

We use cyan to denote types and as we move higher on the cube types and values become increasingly connected.

A grammar for types could be

$$\tau ::= \text{int} \mid \text{bool} \mid \tau \rightarrow \tau. \tag{4}$$

The two types int and bool are the *base types* of our grammar and in practice there are more than two.

The following three subsections describe the three axes starting from $\lambda^{\rightarrow}$.

2.1 λF: System F

Languages based on System F support polymorphism, which is to say a value can depend on a type. For ex-

```
1  void f(int x, int y) {
2    if (x > y) {
3      x = x + y;
4      y = x − y;
5      x = x − y;
6      assert(x − y > 0);
7    }
8    printf("(%d,%d)\n",x,y);
9  }
```

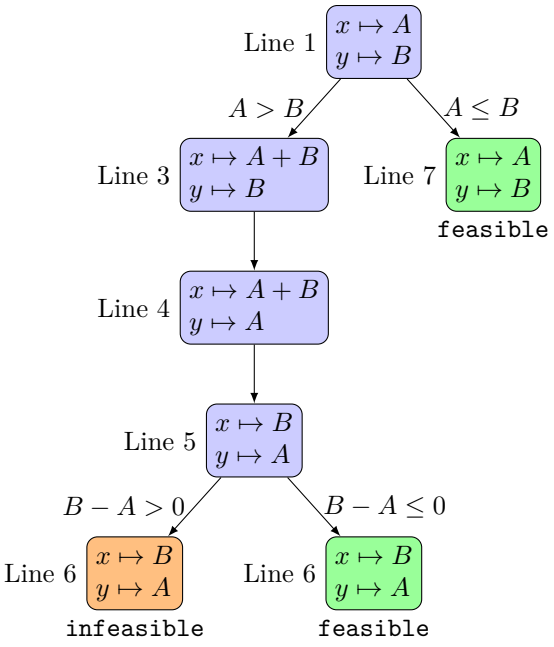

Figure 3: Example to illustrate symbolic execution taken from Emina Torlak [138]. Line numbers in the right graph indicate the state of the program *after* the given line. Symbolic execution can determine there are no values of `x` and `y` that make the assertion fail.

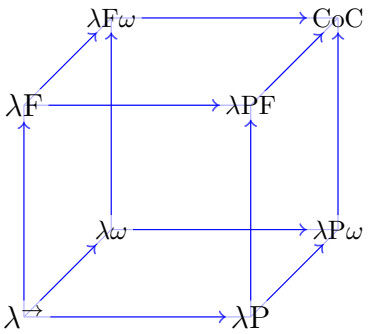

Figure 4: Three of the many dimensions of type theory. An arrow $x \to y$ indicates y includes the features of x.

ample, the polymorphic identity function is

$$\forall\tau\ \lambda(x : \tau).\ x.$$

In English, this value means "to have a well-formed identity function, we may simply drop in any τ". One subtlety here is we *must* input a type to get a valid value. Many modern languages allow *type inference* where the context in which a value is used can cause the typechecker to insert the correct type for the polymorphic function. This allows for cleaner programs and is supported in languages like Haskell and OCaml.

Other literature may refer to λF as λ2 because, theoretically, λF corresponds to second-order intuitionistic logic via the Curry-Howard correspondence [118, p. 341].

2.2 λω: Type Operators

Type operators can be described as "types depending on types." That is, one can write functions which take as input types and return types. One example would be a function which takes as input two types and creates a pair out of them:

$$\lambda\tau_1.\ \lambda\tau_2.\ (\tau_1, \tau_2)$$

This is completely cyan for a reason; this is a different λ which cannot be interchanged with black λs. One valid mixing would be a function which takes as input a pair, then returns the first of the pairs.

$$\lambda(x : \lambda\tau_1.\ \lambda\tau_2.\ (\tau_1, \tau_2)).\ \text{fst}\ x$$

where fst has type $(\tau_1, \tau_2) \to \tau_1$.

In a more approachable setting, a list is a type operator. Using Haskell as an example, first consider a regular function which always returns 0.

```
zero :: a -> Int
zero _ = 0
```

Here, `a` is a *type variable*. This can stand in for any type, but the function still takes some *value* as input. When creating type operators, there must be a distinction between functions which take values and functions which take types as inputs for it does not make sense to write `zero Bool`.

Type operators are denoted by a function signature with special syntax. To give the signature for the list type operator, we write

```
[] :: * -> *
```

Specifically, `[]` has *kind* `* -> *`; the list constructor takes a type and returns a different type. This means `[] Int` is a perfectly valid type, though this is typically written `[Int]`. Similar to type variables, `*` is called a *kind* variable and can stand for any type.

2.3 λFω: the Basis for HOL

The combination of type operators and polymorphism give rise to higher-order logic. This is the basis for, and the naming of, HOL. One important distinction here is HOL is classical higher-order logic rather than intuitionistic.

2.4 λP: Dependent Types

Broadly speaking, dependent types allow a program to "compute types." Slightly more formally, in dependently-typed languages, we may write type definitions that take as input not just other types but *values*. Other literature may refer to dependent types as the λΠ-calculus and use Π to refer to lambda abstractions which take as input values and return types. Symbolically,

$$\Pi(x : \tau_1).\ \tau_2.$$

Here is where λ-calculus notation gets confusing. Previously, we had a clean separation of types and values. In λω we had functions which accepted *kinds* and functions which accepted values. Now, we have yet another function denoted Π which takes values and returns types.

The canonical example of dependent typing is a vector, or a list of a fixed length. We provide the definition of a vector in an imaginary language similar to Agda.

```
data Vec {A} (A : *) : ℕ → *
```

`Vec` is a type constructor which takes as input some type and some natural number `n` and returns the type of lists of length `n`. The questions of *if* and *how* the typechecker can know a given value is of type `Vec` is a more complex question. Type inference is undecidable for dependently typed languages [62] so a typechecker only in some cases can infer the correct type `A` for polymorphism.

While not strictly required, dependently typed languages play an important role in formal methods because they allow strong specifications and theorems to be checked alongside software development.

Some examples of actively maintained dependently typed languages are Agda [26], Idris [28], F* [134], and Twelf [117].

The discussion of dependent types brings up the question of what is essential for a proof assistant? By the Curry-Howard isomorphism, technically any Turing-complete language will suffice. However, suggest using a language like Bash to compute a proof and a type theorist will laugh (the author has tested this). The reason dependent types are popular for proving properties of a program is twofold: there is strong theoretical research put into reasoning about dependent types' properties and practicality. Also, providing sophisticated typing systems allows the programmer to specify the properties completely and statically. Using a dynamically typed language to specify theorems is sort of like trying to prove a theorem without writing down what you're trying to prove. This is such a misuse of Definitions 1 and 2 as to be almost absurd.

2.5 CoC: the Basis for Coq

Dependent types, polymorphism, and type operators together yield the Calculus of Constructions. Early versions of Coq were based on CoC (this, paired with its inventor's surname Coquand may help explain the name). The current Coq's core logic includes inductive types and so is referred to as the Calculus of Inductive Constructions (CIC) [48].

To describe the CoC, we must first expand on this distinction between types and values brought up in § 2.4. With type operators, we distinguished between types which describe values and kinds which describe type operators. For example, in Haskell `5` has *type* `Int` and `Int` has *kind* `*`. But this raises the question, what is the "type" of `*`? We do not have the language to express such a statement in Haskell, but in the CoC we do. To describe meta-labeling of types, we use *type universes*. Each type universe is distinct from the others to avoid self-referential type signatures. If we recall, STLC was created to prevent such self-reference in order to avoid logical paradoxes. It turns out if you allow something like `* :: *` in Haskell (`Type : Type` in Coq), this leads to a paradox called Girard's Paradox [47]. One can think of this as the type-theoretic equivalent to Russel's Paradox. Russel's Paradox constructs a set

$$R = \{x \mid x \text{ is a set}, x \notin x\}$$

then asks the question "Is $R \in R$?" The paradox is if R contains itself, $R \notin R$. But since $R \notin R$, this fits the definition of R so $R \in R$. Coq gets around this issue by enforcing universes; under the hood, `Type_0 : Type_1` and so on.

If a logic allows such self-reference it is called *impredicative*. Formally, an object is *predicative* if it is defined as a quantification over a type which does not contain that object [48]. Impredicativity is closely tied with inconsistency but is also more expressive. Coq has a single impredicative type called `Prop`. Below is a listing of some of the type universes:

```
(∀ A B : Prop, A ∧ B -> B ∨ B) : Prop
Prop : Type_0
Type_0 : Type_1
Type_1 : Type_2
(* And so on... *)
```

All the other universes are predicative. Through this careful distinction, CoC, and therefore Coq, remain consistent [37, Universes Chapter].

2.6 Tactics

Automation of formal methods is limited by three seemingly insurmountable problems. First of all, discovering a proof of a given proposition is undecidable in general. Additionally, for programs of reasonable scale, there tends to be a combinatorial explosion of the state space. Lastly, the specifications of what is considered a correct program may not even be written down (i.e. it only exists in the mind of the developer). However, languages such as Coq support the creation of user-specified tactics which guide a proof assistant into finding proofs of given theorems [37].

A *tactic* is a program that finds a proof. The nice thing about Coq is all tactics and proofs compile to a simple, well-understood kernel. That means an incorrect tactic or proof will not allow false results to be proven; instead, generating an incorrect proof will succeed but the checking of a proof will fail. This property of Coq is known as the *de Bruijn* criterion. As Chlipala writes, "To believe a proof, we can ignore the possibility of bugs during *search* and just rely on a (relatively small) proof-checking kernel that we apply to the *result* of the search." [37, Introduction Chapter]. An overview and history of proof assistants is given by Geuvers [71], who describes tactics as a necessary feature for a proof assistant.

Tactics are useful when the structure of a proof of correctness is dissimilar from the program itself. It is these cases where languages without some proof automation are difficult to use because dependent types do not suffice when describing a valid specification. For example, verifying the correctness of a compiler has very little to do with the structure of the compiler itself. Namely, a compiler's proof of correctness must refer to the semantics of the source and target languages which have nothing to do with how the intermediate representations are transformed during each compiler pass.

2.7 Foundations of Mathematics

Philosophers, mathematicians, and computer scientists have thought deeply about type theory have developed it as an alternative foundation of mathematics in contrast with, for example, Zermelo-Fraenkel set theory. Additionally, all proof assistants are based on intuitionistic logic [108] which has the additional requirement of constructivism; that all proofs must be computable. To see the distinction, let us consider a non-constructive proof to the following theorem.

Theorem 1. *In a class with some number of students, there exists a student such that if s/he gets an A in the class then everyone in the class gets an A.*

Proof. We know at the end of the term one of two possibilities occur: either everyone gets an A or at least one person does not. If everyone gets an A, then we may choose anyone in the class arbitrarily to complete the proof. Otherwise, that person who did not get an A is the desired student. □

If I'm in a class, my first goal is: I should find that special student! But given a classroom we cannot find such a student, the proof is somehow less satisfying than a constructive proof. Beyond this aesthetic shortcoming, the nonconstructive proof of Theorem 1 relies on the *law of excluded middle* which states for any proposition, either A or $\neg A$ must be true; the problem with a proof using this law is we cannot know which is true! This has deep consequences with respect to computability. Without getting into the details, removing the law of excluded middle and the law of double negation ($\neg(\neg A) \iff A$) allows a logic to be intuitionistic and therefore contains only constructive proofs. Programming languages which are based on constructivism are said to have *computational type theory* or CTT [45].

One feature of proof assistants is their ability to formalize a given field of mathematics. Certain combinations of proof assistants and theories may be intractable. For example, Sylvie Boldo et al. give a survey of practical proof assistants for real analysis which includes HOL, PVS, and Coq but states ACL2 is not suitable for real analysis and languages like Agda are not even mentioned [23]. As a rule of thumb, the more strongly a language is designed for a particular application, the more pleasant it is to design in that particular space. Conversely, that language is less practical for other domains.

For example, Coq has been used to formalize a large number of theories [52]. However, programming proofs in Coq may be unwieldy or infeasible for large programs thus software abstractions are built on top of it such as Bedrock [36]. Conversely, Twelf is specifically designed for proofs relating to programming languages but has not been expanded to domains outside of its use as a metalanguage for programming language properties [130].

2.8 Summary

In this section we described the λ cube (Figure 4) and how some corners of it have been used as the basis for various proof assistants. Additionally, we described proof finding programs, or tactics, and how they contribute to a proof assistant's practicality.

The following section describes in further detail the workflow of a verification researcher as well as provides a more complete list of state of the art tools.

3 Formal Methods in Practice

Given that a user wants to verify a piece of software, s/he must make a few design decisions. These begin intentionally vague.

1. To what degree of confidence must the software be guaranteed?

2. What tools can be used to accomplish #1?

3. How much time can a human spend on #2? How much time can a computer spend on #2?

Each has important considerations we address in the following subsections, respectively.

3.1 Degree of Confidence

Given the task of verification, at what point do we become confident enough that a piece of software is correct? Donald Knuth's famous quote hints at this challenge: "Beware of bugs in the above code; I have only proved it correct, not tried it." Verily, bugs in software are ubiquitous. Additionally, verification is only as good as the specification. One example is a security vulnerability found in the WPA2 WiFi standard which had previously been proven secure [142]. The exploit did not violate the verified safety properties but instead exploited a temporal property, of which the WPA2 specification was ambiguous. While temporal properties are beyond the scope of this survey, we must always remember a bug in any part of the software toolchain, from handwritten specifications all the way down to hardware implementations, can be the cause for failure. For example, in 1962 NASA's Mariner I rocket crashed because of a mistake in the handwritten formula for a radar timing system [112]. Even worse, the accumulation of errors from representing 0.1 seconds in binary using a fixed-point format resulted in the deaths of 28 soldiers because of a timing error of a Patriot missile [141].

Furthermore, one must also know if one can trust the verifier; a bug in the verification software could cause false positives or false negatives. The approach to solving this problem taken by some proof assistants is to create a small, well-understood kernel upon which all else is built. The trust is further strengthened by proving existing theorems using this kernel, proving the kernel's correctness by hand, and checking the results on many computer systems. This is known as the de Bruijn criterion.

Put another way, we write a proof (which by the Curry-Howard correspondence is equivalent to the evaluation of a function), but we must believe the program which performs the evaluation is correct as well. This is easier the simpler the evaluator is. We discussed the de Bruijn criterion in § 2 and Table 3 lists whether various theorem provers satisfy this criterion.

Epistemology aside, a more practical approach would be to determine what sort of properties you may want to prove about a system. For example, consider Figure 5 which attempts to find solutions to $a^3 + b^3 = c^3$.[2]

[2]We use $n = 3$ to keep the algorithm simple, but it is important to note a proof for $n = 3$ has been known for hundreds of years.

Now, there are several questions we may want to ask about this program. Some are basic safety guarantees. For example, will the function `ipow` correctly handle all integers? In this case, no because $n < 0$ will most likely cause a stack overflow. The next question we may ask is will the program ever pass a negative value into `ipow`? We say yes because eventually the integers will overflow into negative values and cause an error. These sorts of questions can be answered with symbolic execution then dispatched to SMT solvers.

A more interesting question is will this program test every valid combination of a, b, and c? Proving this requires some loop invariants which requires annotating the source code, but otherwise manageable. Once we've done this, we've proven partial correctness of this code. That is, if the program terminates, then it gives us the right answer.

The final question we may ask is: Does this program terminate? In order to answer this question, we need a proof of Fermat's Last Theorem, a feat which was left unanswered until Wiles' proof in 1995 [146].

Thus, we see the property we wish to prove about a program ranges from trivial to almost impossible.

```
int ipow(int x, int n) {
  if (n==0) return 1;
  return x * ipow(x,n-1);
}
int main() {
  int a,b,c,n;
  n = 3; c = 0;
  while (1) {
    c++;
    for (a = 1; a < c; a++) {
      for (b = 1; b < c; b++) {
        if (ipow(a,n)+ipow(b,n)==ipow(c,n))
          return 0;
} } } }
```

Figure 5: A naïve attempt to find a counterexample to Fermat's Last Theorem for $n = 3$.

3.2 Methods for... Formal Methods

When selecting a strategy for verification, one must know what sort of properties may be proven using that strategy.

In his 2016 Milner Award Lecture, Xavier Leroy provides an illustrative image describing the tradeoff between interactive and automatic verification [102]. We render this image in Figure 6.

We provide an overview in Table 2 and follow up with a more detailed description of each.

Many formal methods concepts rely on these automated reasoning backends. This is a vast area beyond the scope of this paper but an excellent reference on automated reasoning is provided by Harrison [83].

Table 2: Categories of formal methods strategies

Method	Input	Output	Notable Example(s)
Static Analyzer	source code	safe array accesses	Astreé [51], Clang [68]
SMT Solvers	propositional, 1st-order logic	satisfiable/unsatisfiable	Z3 [59]
Model Checkers	petri nets, büchi automata	concurrency safety	SPIN [88], Helena [63]
Deductive Program Provers	annotated source code	partial correctness	Why3 [65]
Proof Assistants	proof assistant program	proof certificate	See Table 3

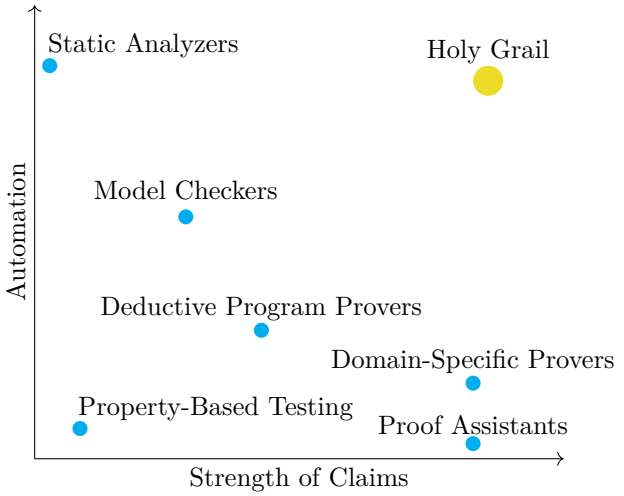

Figure 6: The landscape of formal verification, inspired by Leroy [102]. Up and to the right is good, but the holy grail is unattainable.

3.2.1 Static Analyzers

Corporations such as Google [126] and Facebook develop its own static analyzers. FBInfer is an open-source static analyzer for C, C++, Objective-C, and Java supported by Facebook [64]. Google started using FindBugs [8] to analyze Java programs then developed their own static analyzer. France's primary static analyzer is Astreé [19]. One concern of static analyzers such as Clang or FBInfer is they can produce false positives (valid code is labeled as invalid) or false negatives (invalid code is labeled as valid). Tools which faithfully implement abstract interpretation such as Astreé are sound—no false negatives. Of course, we must temper our expectations because static analysis cannot detect all possible errors. However, a large class of errors such as division by zero, arithmetic overflow, array out of bounds, and data races can be checked using static analyzers.

Additionally, the ROSE project from Lawrence Livermore National Labs [127] is a source-to-source compiler which facilitates static analysis of source code and binaries.

Predicate abstraction is a form of abstract interpretation wherein the abstract domain is constructed by a user-provided collection of predicates regarding program variables. This is a more robust method of abstraction than loop invariants and these abstractions can generate loop invariants. However, deductive program provers can ultimately prove stronger properties.

The original paper on predicate abstraction uses the PVS theorem provers [77]. However, more modern approaches apply predicate abstraction to analyze languages like Java [66]. The Berkeley Lazy Abstraction Software Verification Tool (BLAST) [86]. and Microsoft's SLAM project used to verify code used in device drivers [10], are more examples of predicate abstraction which both take as input C programs.

One prominent static analyzer is Frama-C [53]. This suite of program analyzers (Frama-C calls them plug-ins) takes as input annotated C code. These annotations are written in a language called ANSI/ISO C Specification Language (ACSL). The program is then run through Frama-C which uses various methods including abstract interpretation, Hoare Logic, as well as deductive verification to analyze various properties of the program.

3.2.2 Model Checkers

Model checking is typically separate from the actual program. One common use case of model checking is a machine-readable specification which supplements both a written description and source code implementation. Then, the model checker verifies the behavior. To go with the duality of Definitions 1 and 2, model checkers often consist of a specification language and checking engine. The following examples are for modeling and verifying concurrent and distributed software:

- Modeling language PROMELA and the SPIN model checker [88].
- Modeling language TLA+ and its checker TLC [97].
- Petri Nets [116] and the Helena [63] model checker.
- NuSMV [38] in its own language which is used to express finite state machines.

3.2.3 Deductive Program Provers

Deductive Program provers seek to combine static analysis, SMT solvers, and model checkers to prove strong properties of a program without requiring full proof assistants. These tools work by annotating a program with

a more complete specification. These annotations are typically embedded in comments in the code. The program with annotations is used to generate *verification conditions* which then are dispatched to various backends such as SMT solvers.

An example of this approach is Why3 [22] from INRIA. With Why3, specifications are written in WhyML, an ML-style language extending first-order logic with some common programming bells and whistles such as algebraic data types, polymorphism, and pattern matching. Why3 aims to be a front end for theorem provers and can also dispatch verification conditions to Coq.

Additionally, Frama-C is not just a static analyzer because of its collection of plugins. It fits the description of deductive program prover because of its reliance on annotation followed by proof dispatch. Another static analyzer which goes beyond just static analysis is SPARK [12]. SPARK is a formally defined subset of the Ada programming language which includes code annotation and can dispatch verification conditions to the SPADE proof checker, whether these are proven automatically or manually.

Yet another project from Microsoft Research is Dafny [99]. which dispatches to other Microsoft projects like Boogie [13] and Z3 [59].

All these layers of specification languages, intermediate representations, and proof checkers can get confusing, even for professionals in the field. To this end, workflows have been developed in order to manage these levels of abstraction and decidability [42].

3.2.4 Proof Assistants

In § 2 we provided an overview of proof assistant concepts while Table 3 taxonomizes several of the most mature or popular proof assistants. Many of these draw inspiration from what could probably be considered the first proof assistant, AUTOMATH, way back in 1968 [55]. Further developments came from the logic of computable functions (LCF) [75] from which HOL [76], Coq [70], and NuPRL [46] are descendants. Strictly speaking, HOL is a family of theorem provers which includes its canonical implementation HOL4 [131], its most active implementation Isabelle [113], and HOL Light, an implementation with a simpler logical kernel [84].

As per Table 3, not all proof assistants satisfy the de Bruijn criterion. However, this does not mean other automated theorem provers are untrustworthy per se, just there are more places in which mistakes in software can happen. In Coq, any user-generated tactic doesn't change the underlying kernel, but a system like ACL2 does not satisfy the de Bruijn criterion, so extensions must be much more carefully checked.

A popular and mature proof assistant is Coq [119]. Coq is regarded as well-understood by the formal methods community because of the large number of proofs which have been written in it, most famously a proof of the four-color theorem [73]. Programs in Coq are written using the specification language Gallina.

LEAN is a new theorem prover from Microsoft Research [58] which aims to be easily integrated with Microsoft's specification language F*.

3.3 Human and Computer Time

Randomized property-based testing such as QuickCheck [39] requires relatively little human time but randomized inputs allow an arbitrary amount of computer time to be spent. Conversely, thinking of specific, known edge cases will take more human time but less computer time. This may be useful for Continuous Integration (CI) scenarios where long-running unit tests can disrupt the workflow of an active project.

The strongest guarantees such as termination typically require proof assistants. On the other hand, unambiguous specifications can also be time-consuming to define. We also note proof assistants are not necessarily fast; a user may have written tactics which take an unreasonable amount of time and thus will require more human guidance to have the proof terminate in a reasonable amount of time. The simplest example of this is writing a tactic which exhaustively checks all proofs consisting of n terms; this is akin to attempting to prove a theorem by picking words out of a hat.

Further automation can be achieved via domain-specific heuristics. For example, Gappa [56] can search through a large number of lemmas in real analysis such as $x + y = y + x$ to guide proofs. However, these lemmas are completely useless when reasoning about heap memory so our quest toward the Holy Grail in Figure 6 is limited.

3.4 Summary

We outlined three questions a researcher must answer when verifying a program. The first is: what is the strength of the property we wish to verify? We used the word strength here to mean to what degree can a program misbehave while still maintaining our verified property. For example, a nonterminating program is still partially correct but we would probably not consider it correct. In general, the stronger the claim, the more human effort is required to prove this claim. We then listed and briefly described popular tools for verification. Lastly, we emphasize this tradeoff between human time and computer time. This section is in some sense just an explanation of Figure 6.

The next three sections describe applications of these verification techniques.

4 Intermediate Representations

Before discussing verification techniques for intermediate representations (IRs) we answer the question: why would

Table 3: Characterization of specification and proof languages. Versions and release dates as of January 2, 2019

System	de Bruijn Criterion	Dependent types	Tactics	version	Date	Code generation
Coq	yes	yes	yes	8.8.2	9/'18	OCaml, Scheme, Haskell[1]
HOL/Isabelle	yes	no	no	2018	8/'18	Scala, OCaml, Haskell, SML[2]
ACL2	no	no	yes	8.1	9/'18	N/A
PVS	yes[3]	partial[4]	yes	7.0	11/'15	Java [100]
Twelf	no	partial[5]	no	1.7.1	3/'11	N/A
F*	yes[6]	yes[7]	yes	0.9.6	5/'18	OCaml, F#
NuPRL	no	yes	yes	5[8]	12/'02	N/A
Agda	no	yes	no	2.5.4.2	10/'18	Haskell
Lean	yes	yes	yes	3.4.1	4/'18	N/A

[1] Code generators are not verified; OCaml is much better supported than others.
[2] HOL/Isabelle generates code only for certain classes of programs. See Isabelle's documentation [79] for details.
[3] Depends on your definition of simple; Coq and HOL are relatively simple, PVS involves propositional logic solvers
[4] Dependent types must be a normal type and a predicate over that type. For example, *list* may be the normal type, and the predicate may be the last has nonzero length.
[5] Types only compute on a monomorphic λ-calculus
[6] If you trust their translation from a λ-calculus extension into the CiC [134].
[7] F* is functional but has imperative aspects. Imperative terms are not dependently typed.
[8] Metatheory version. NuPRL is in the process of being implemented in Coq [5].

we not just reason about high-level languages or binaries instead? For one, IRs may have nicer properties than assembly languages; for example, LLVM's IR is typed and has a formal semantics. The translation from IRs to binary is straightforward so the cost of proving correct translation is not prohibitively difficult. Despite being straightforward, this difficulty is non-negligible. However, one IR can have several different front ends (high-level languages) and back ends (ISAs). Thus, we get a unified format to reason about a program which generalizes easily to multiple programming languages and architectures, depending on the IR used. Additionally, compiler transformations often operate on source code or an IR itself so we can more easily verify code transformations and compilers by verifying the IR.

4.1 Reasoning about Assembly Language

One important contribution to the design of assembly languages which allows better reasoning about programs is *static single assignment* form (SSA), first proposed by Rosen et al. [123]. SSA is widely used in intermediate representations (IR) and most of the IRs we describe throughout this report are in SSA, notably GCC and LLVM [98]. The defining characteristic of SSA is each variable is only written to once. SSA facilitates many optimizations such as common subexpression elimination. This is accomplished by introducing ϕ-nodes in the control flow graph (CFG). Wherever two or more vertices in the CFG join, we insert a ϕ node by replacing the assignment of a variable with ϕ to indicate the assigned variable can take one of several values. For example, a join point can be at the end of an if-then-else statement shown in Figure 7.

Mechanically verified assembly languages have been researched since 1989 with the Piton assembly language [106] [107]. However, the most popular IR is the language used by LLVM IR [98], which surprisingly, doesn't have a name other than "LLVM IR."

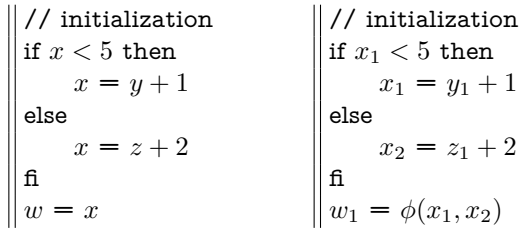

Figure 7: SSA example

After the success of LLVM, Zhao et al. formalized LLVM's IR with Vellvm in Coq (Verified LLVM) [149]. Key contributions of Vellvm were formalizing the nondeterminism which arises from the fact that LLVM can be underspecified (using the **undef** keyword) as well as allowing for arbitrary memory models.

4.2 Compilers

As early as 1967, compilation algorithms have been verified with respect to the source semantics are translated into a target semantics [91]. However, there is still the concern whether a programmer's implementation matches the algorithm written on paper. In 2000, verified transformations were applied to GCC [110]. However, it wasn't until several years later that a more complete compiler verification story was undertaken by Xavier Leroy.

The most prominent project striving for complete formal verification of a compiler is CompCert, written in Coq [101]. In addition, there is an ambitious project led by Andrew Appel called the Verified Software Toolchain [6] (VST), of which CompCert is one step. In

the VST, the goal is the formal verification of the input code (viz. static analysis), the compilation (viz. CompCert), and a runtime system (viz. operating system) to verify a program's behavior. The VST ultimately also aims to support concurrent behavior via shared-memory semaphores.

CompCert and the VST assume the same exact compiler is used on all parts of the program. However, modern software simply is not developed this way because of the heavy reliance on standard libraries coded in multiple languages. Amal Ahmed's research group seeks to address these issues [115].

Other verification efforts for compiler optimizations include the Alive project from Microsoft Research [103], more optimization passes in CompCert [109], and local optimizations using SMT solvers [33].

4.3 IRs in Practice

Microsoft Research focuses heavily on verification through domain-specific languages and IRs. For example, CIVL is a concurrent intermediate verification language which extends the Boogie intermediate language for concurrent programs [13]. CIVL looks similar to a markup language for assembly IRs [85].

A project from CMU called the Binary Analysis Platform (BAP) [31] consists of an intermediate language which makes all side effects of a given assembly instruction (such as setting condition codes) explicit. BAP supports subsets of Intel x86 and ARM ISAs and can generate verification conditions in a weakest-precondition style specification. BAP can also be used to mine general information about a binary such as instruction mixes. Once the desired postcondition is described, BAP dispatches the verification of the weakest precondition to an SMT solver.

One drawback to BAP is its limited support of ISAs. The complete x86-64 along with vectorized instructions consist of thousands of instructions so it is difficult for projects to keep up.

A feature of many binary analyzers is their use of intermediate representations or intermediate languages (IR/IL). For example, BAP has its own IL, Valgrind has VEX. These ILs/IRs are often formally specified (as with BAP and Vellvm) to remove any ambiguities. Another benefit of ILs/IRs are the increased portability; support for a new ISA requires only adding a new parser.

A common theme throughout verification tools is that the workflow consists of three steps:

1. Annotation of a program, either hand-written or automatically generated
2. Transformation of this program into an IR via some semantics such as Hoare logic-style predicate transformers
3. Transform this IR into a common format such as SMTLib then dispatching to an SMT

In between these steps is often optimization to mitigate the state space explosion problem.

Other work includes verification of an extension of MIPS assembly language [2]. Recently, WebAssembly has been formalized and its type system proved sound in Isabelle [144]. WebAssembly is meant to be a target for high-level languages which can run on web clients and servers while also using the local machine's hardware directly to improve performance. In the trend of hardware corporations adopting formal methods much better than software corporations, ARM is also in the process of converting their semantics to a machine-checkable representation [121].

Microsoft Research has a recent project to combine its specification language $F^\star$ and a subset of the Intel x86-64 assembly language to verify hand-optimized code [69].

Last but not least, we give special mention to Valgrind [111], a tool which consistently surprises people with its features. Valgrind also has static and dynamic analysis, concurrency error detection, memory leak checking, and profiling.

5 Floating Point Arithmetic

After the previous discussion on formal methods and verification, this section may seem out of place. We go into great detail describing floating point representations which may seem pedantic. Floating point arithmetic is a remarkably successful and popular abstraction for good reason; floating point numbers behave as expected most of the time. However, situations where they do not behave like the real line can be disastrous; floating point errors have been the cause of some of the most expensive bugs in engineering history [89]. It is only at a high level of detail can we see some of the subtleties which make floating point arithmetic difficult to reason about completely.

5.1 Floats, Bits, and the Real Line

Floating point arithmetic is the workhorse of scientific computation yet is fundamentally an approximation since it represents an uncountably infinite set, the real numbers (denoted $\mathbb{R}$), in a fixed number of bits. While this approximation is usually good enough, we wish to describe when and how the approximation fails.

The most common encoding of $\mathbb{R}$ is the IEEE 754 standard. We refer to its most recent specification, the 2008 revision [148]. The IEEE 754 standard for floating point arithmetic is the most ubiquitous standard by far, but we wish to understand other formats in a unified way. IEEE 754 was first standardized in 1985, so before then floating point arithmetic varied across hardware. We describe these in § 5.4.

While popular, IEEE floats certainly have their quirks. For example, having the same bit pattern is neither necessary nor sufficient for two IEEE floats to be considered

equal. For example, the bit pattern $0\ldots0$ and $10\ldots0$ represent -0 and $+0$, respectively, and are equal but whose bit patterns are not the same. Conversely, Not-a-Numbers (NaNs) may have the same bit pattern but are defined to be not equal to every other float, including the same NaN bit pattern.

A good introduction to the concerns when representing floating point numbers is described in 1991 by Goldberg [72]. From Goldberg, we describe the accuracy of a floating point operation in terms of *units in the last place* or *ulp*. For example, if the true value of an operation is 1.1 and the floating point representation is 1.125, then the error is 0.25 ulp. We also use *relative error* which in the previous example is

$$1.125 - 1.1/1.124 \approx 0.147.$$

When reasoning about floats, naïvely treating floating point properties as instances of a Satisfiability Modulo Theories (SMT) can lead to pitfalls [43]. For example, (5) gives an unsatisfiable proposition; proving this proposition unsatisfiable equates to proving if $x \leq y$ then it is impossible for $y + z < x + z$ for some small z:

$$\begin{aligned} &(-2 \leq x \leq 2) \\ \wedge&(-2 \leq y \leq 2) \\ \wedge&(-1 \leq z \leq 1) \\ \wedge&(x \leq y) \\ \wedge&(y + z < x + z). \end{aligned} \tag{5}$$

Treating these variables as vectors of bits ("bit blasting") takes an unreasonable amount of time with even sophisticated SAT solvers. This example will be discussed further in § 5.5

Broadly speaking, formalizations of floating point arithmetic either treat floating point numbers as subsets of real numbers or as a collection of bits. While not strictly true (we shall give special attention to a Coq library called Flocq [24]), this provides a good characterization.

Using the "floats as bits" interpretation, bit pattern quirks of IEEE floats can be easily handled but properties of real numbers can be intractable to specify.

Using the "floats as reals" interpretation, one can express any theorem of real analysis and determine how floating point operations map to these theorems. Libraries instead focus on operations on $\mathbb{R}$ and how well floats approximate these operations via the semantics of rounding. Conversely, real analysis is not understood by the lowly transistor and real-world code may do strange things to the bits of a float while still being correct.

5.2 Floating Point Representations

A floating point number is represented as

$$\pm k\beta^p$$

where $\beta > 1$ is the radix, p is the exponent, and k is the mantissa. To make notation a bit[3] clearer we use $\beta = 2$, though $\beta = 10$ is also widely used.

While we could pick any range for k, in reality either $0 \leq k < 1$ or $1 \leq k < 2$. This means we have $k = (1.k_1k_2k_3\cdots k_n)_b$ or $k = (1.k_1k_2k_3\cdots k_n)_b$. Here the subscript $_b$ indicates the number is expressed using a binary radix. Since we know the first bit is either 0 or 1 these are omitted with a binary representation and we say a number is *normalized* with an implied 1 and *subnormal* with an implied 0.

5.3 IEEE 754

The IEEE 754-2008 allows both normalized and subnormal values, though subnormal behave more predictably because they sacrifice lower precision to represent numbers closer to 0.

With IEEE floats, the exponent is an e-bit number. Values of e are shown in Table 4. Given some bit pattern representing an unsigned integer p, we compute the actual exponent as $p - bias$, where $bias = 2^e - 1$.

One would think this means we have representable exponents in the range $\{-2^{e'-1} - 1, \ldots, 2^{e'-1}\}$ (for 32 bits this is $\{-127, \ldots, 128\}$) but in reality this range is 2 smaller because an exponent pattern of all 1's signals Infinity (Inf) or Not a Number (NaN) and an exponent pattern of all 0's signals a subnormal number.

For example, with IEEE 754 half precision, the following indicates the smallest normalized float and the largest subnormal float.

$$\begin{aligned} &0\ 00001\ 0000000000 \mapsto 2^{-14} \times 1.0_b \\ &0\ 00000\ 1111111111 \mapsto 2^{-14} \times 0.1111111111_b. \end{aligned}$$

IEEE 754 is implemented such that the result of a floating point operation should be the result of the operation applied to real numbers (i.e. with infinite precision), then rounded to fit in the correct number of bits. Practically, the rounding aspect is the most difficult to reason about and is the cause of the most insidious floating point bugs.

The IEEE standard defines five exceptions:

1. Invalid Operation, e.g. $0.0 \times \infty$
2. Division by Zero
3. Overflow, i.e. to either $+\infty$ or $-\infty$
4. Underflow, i.e. to $+0$ or -0
5. Inexact, e.g. $\sqrt{2}$

The GNU C library manual describes how they handle this [137]. Typically if one of the five exceptions occur, a status word is set and the program continues as normal. However, one can override this to throw a `SIGFPE` exception which can be trapped.

[3] pun intended

Another subtle feature of IEEE 754 is the difference between *quiet* and *signaling* NaNs, (qNaN and sNaN). Distinguishing between these types of NaNs is hardware-dependent but are typically differentiated by the most significant bit of the mantissa. For example, the canonical sNaN and qNaN for RISC-V are:

```
#define qNaNf 0x7fc00000
#define sNaNf 0x7f800001
```

Thus, there is a wide range of valid signaling and quiet NaNs ($2^{52} - 1$ different bit patterns are valid NaNs for double precision floats).

While this is an IEEE specification, not all implementations (for example, Java) make this distinction between quiet and signaling because it requires hardware support and the five exceptions paired with traps sort of already handle this. Additionally, the behavior of sNaNs is quite messy because the compiler may optimize away instructions which would generate these sNaNs, thus not throwing an exception.

Because of this, one might ask the question "Why would I even bother with signaling NaNs?" One reason would be to catch floating point errors earlier on. For example, one could set an uninitialized floating point value to a sNaN to raise an exception in all cases rather than potentially propagating a garbage value. Signaling NaNs allows catching of some operations in which qNaNs do not propagate; for example

$$\min(x, \text{NaN}) = \min(\text{NaN}, x) = x$$

for all x which aren't NaNs (including ∞!).

The last and most important concept with IEEE floats is the rounding mode. IEEE 754 specifies four rounding modes:

1. Round to nearest, ties break to even
2. Round to zero
3. Round to positive infinity
4. Round to negative infinity

The default behavior is typically round to nearest.

5.4 Other Floating Point Formats

We briefly describe other floating point formats which seek to either address issues with IEEE 754 floats, simplify their implementation for performance, or were invented before IEEE 754's standardization in 1985.

5.4.1 Posits

Posits are a new floating point format invented by John Gustafson with strong claims for better performance and accuracy without actually being implemented in hardware nor much empirical evidence [78]. However, it is an interesting format which if the claims are correct could usurp IEEE 754 as the dominant floating point format.

5.4.2 MIL-STD-1750A

MIL-STD-1750A is an open instruction set architecture describing an instruction set architecture including two floating point formats [140]. One interesting feature of the floats of MIL-STD-1750A is since all floating point values are subnormal, to prevent multiple representations of the same number all mantissa must be in the range $[-0.5, 0.5]$. This means a large number of bit patterns are invalid floats.

5.4.3 Modifications to IEEE 754

Hardware manufacturers in the interest of performance may simplify aspects of IEEE floats. For example, NVIDIA allows flags which do not give full accuracy for division and square root as well as flushing subnormal numbers to zero [145]. Additionally, on a GPU traps for floating point exceptions are typically not supported.

Machine learning hardware may implement a nonstandard 16-bit floating point format called a *bfloat* (brain float) which has 8 bits for the exponent and 7 bits for the mantissa (contrasted with 5 and 10 for IEEE half precision) [135].

5.5 Formalizations of Floats

One early formal description of floating point arithmetic was introduced in 1989 [17] defined using the specification language Z [133] but this formalization is not maintained.

Another formalization of IEEE 754-2008 floats is an extension to the SMT-Lib Standard done by Brain, Rümmer and Wahl [124, 30]. This extension supports exceptions, the usual mathematical operations, and rounding modes. However, SMT representations are limited because they essentially only think of floats as their bit-level representations and not how they relate to $\mathbb{R}$. This means an SMT solver when determining bounds of some operation has no good way to estimate them and in many cases must exhaustively check all floating point values. For example, it is difficult to represent a property like $x + y = y + x$ for floating point arithmetic. Another example that SMT solvers struggle with was previously shown in Equation (5).

To address some of these limitations, Brain et al. developed an Abstract Conflict-Driven Clause Learning (ACDCL) [29]. This method is based on choosing an abstract domain over which to apply the CDCL algorithm [104]. ACDCL is one part of an effort to unify abstract interpretation with SAT techniques. As it applies to floating point arithmetic, ACDCL creates abstract domains for interval arithmetic. So for example, the formula $x = y + z$ may restrict $y, z \in [0, 10]$. Thus, interval arithmetic states $x \in [0, 20]$. This also cannot solve even a simple problem such as (5) easily because the abstract domain is not precise enough. Thus, the community requires more expressive theories about floating point operations.

Table 4: Some important values for floating point representations

Name	Bits	Radix	e	m	$e_{\min}$	$e_{\max}$
IEEE	n	2	e	m	$-2^{e-1}+2$	$2^{e-1}-1$
half	16	2	5	10	-14	15
single	32	2	8	23	-126	127
double	64	2	11	52	-1022	1023
quad	128	2	15	113	$-16,382$	$16,383$
posit	n	2	e	$n-3-e$[1]	$-(n-2)\times 2^e$	$(n-2)\times 2^e$
posit	32	2	3	26[1]	-240	240
posit	64	2	4	57[1]	-992	992
1750A	32	2	8	16	-128	127
1750A	48	2	8	32	-128	127

[1] Tapered precision; maximum is shown here

Formal verification of floating point arithmetic and numerical algorithms primarily comes from the Toccata research group [1]. One example is creating a specification of floating point numbers in Coq, called Flocq [24]. Boldo & Melquiond also wrote a book describing verification of some numerical algorithms using Flocq [25]. Flocq allows reasoning about arbitrary rounding modes and is generic enough to define other floating point formats. However, there is no concept of overflow since exponents can be of arbitrary size in Flocq. The authors acknowledge this and mention it is easier to reason about overflow separately from correct rounding and error analysis. However, this still distinguishes Flocq as a library on the side of "floats as reals."

In addition to Flocq, there are PFF, Gappa, and Coq.Interval which accomplish related goals. PFF focuses on arithmetic without rounding errors and has been subsumed by Flocq [54]. Gappa more closely resembles numerical code written in a C-like language and is designed to be easily machine-checkable [56, 57]. Coq.Interval is a library to formalize interval arithmetic and is also mostly subsumed by Flocq.

The limitations of SMT solvers when reasoning about real numbers caused Conchon et al. [44] to develop a method which uses both Gappa and the Alt-Ergo SMT solver [94]. However, this efficiency comes at the cost of automation; while Gappa can reason about high-level properties of floating point arithmetic (e.g. $x+y=y+x$), Gappa typically requires source code annotation.

While the most mature package for reasoning about floats is Flocq, hardware manufacturers have been early adopters of formal methods and floating point programs are no exception. Work at AMD [4] using the ACL2 prover focuses on register transfer logic. Additionally at Intel, Harrison worked on floating point verification using HOL Light [82]. However, these works are not publicly available so we do not know if their projects are still active.

5.6 Notation

It may also be useful to distinguish between interpretations of floating point values and $\mathbb{R}$. To the best of our knowledge, the following set describes all possible floating point values. We use some notation from Flocq [24] such as f2r but most of this formalization is novel in an attempt to unify "floats as bits" and "floats as reals."

$$\begin{aligned}\mathbb{F} =&\{\text{NaN}, 0, -0, -\infty, \infty, \pm\infty\} \\ &\cup \{\beta^e \times k_0.k_1k_2\cdots k_n \mid n, e, \beta \in \mathbb{Z}, k_i \in \mathbb{Z}_\beta, \beta > 1\}.\end{aligned} \tag{6}$$

Note that both mathematically and practically speaking $\mathbb{F} \not\subseteq \mathbb{R}$. For any floating point format f which is represented using n bits, we write a floating point number x as a string of n bits and so $0 \le x < 2^n$. Henceforth we write $x \in \mathbb{Z}_{2^n}$.

The second portion of (6) is a subset of the rational numbers. This precludes the possibility of irrational numbers like $\sqrt{2}$, or π from being represented exactly. This is an issue, because one would wish floating point arithmetic in some sense remember its rationality, such as $\sqrt{x}\times\sqrt{x} = x$. Alas, this does not in general hold. As we have seen with the division example in Table 1 we cannot even guarantee $\frac{1.0}{x}\times x = 1.0$ for all finite nonzero $x \in \mathbb{F}$. However, approximations must suffice both for efficiency and decidability. That not all real numbers are not computable has been proven more generally by a relatively distinguished computer scientist [139].

We now introduce some mathematical formalizations similar to Flocq. We begin with some floating point representation f. While Flocq does not concern itself with bitwise representation, we roll this into f. One motivation for this is hardware; specifications such as MIL-STD-1750A specify floating point operations with respect to masking, shifting, and adding bits. Similarly, built-in hardware operations such as tests for equality, absolute value, or negation could be used on floating point values as potential optimizations. We present a list of the floating point representations we consider (the set of f's) in Table 5.

We also need three more pieces of notation to under-

stand f. Notice in (6) we do not specify an n. We do this to ensure a float *can* be represented by some finite number of bits. This is to allow for the development of rounding schemes which may need extra bits to compute the correct rounding but also for generality.

If we must specify exactly the set of representable floats for a given f we write $\mathbb{F}_f$. For example, with 32-bit posits

$$\mathbb{F}_{\text{posit32}} = \{\pm\infty, 0\} \cup \{2^e \times 1.k_1k_2\cdots k_{26} \mid k_i \in \mathbb{Z}_2, e \in \{-240, \ldots, 240\}\}$$

because there is no NaN, only a single value for infinity, and mantissas have at most 26 bits. In reality, posits have *tapered precision* which means for larger e the mantissa is shorter (thus requiring the a portion of the least significant k_i to be 0) but this is a sufficient description for illustrative purposes.

Next, we need some function interpreting binary as a float:

$$\text{b2f}_f : \mathbb{Z}_{2^n} \to \mathbb{F}$$

and a function interpreting a float as a real

$$\text{f2r}_f : \mathbb{F} \to \mathbb{R}.$$

These may be quite complicated and messy. For example, b2f_f may be neither total nor injective. In IEEE 754, many bit strings map to NaN and in MIL-STD-1750A there are many bit patterns which are not valid floats. On the other hand, f2r_f is typically neither total (NaN does not map to a real) nor injective ($+0$ and -0 map to $0 \in \mathbb{R}$).

Along with f we have operators on floats denoted with a subscript: $<_f, =_f, \times_f$, etc. We expect these to behave similarly to their corresponding operators on real numbers but there are important differences.

Lastly, one may wish to have inverses of b2f and f2r. This is in general impossible because they are usually neither injective nor surjective. However, analysis of floating point arithmetic correctness requires us to think of how $\mathbb{R}$ embed into floats. This is done through the various *rounding modes*. For some rounding mode r associated with a floating point format we define

$$\text{round}_r : \mathbb{R} \to \mathbb{F}.$$

In contrast with f2r, all rounding functions are total. For practical reasons, round_r has a canonical binary representation for each float to which it rounds. To keep notation cleaner, we overload round_r to map both into $\mathbb{F}$ and $\mathbb{Z}_{2^n}$; the former codomain is preferred when reasoning about floats more theoretically and the latter when you actually need to push some bits around.

5.7 Properties of Floating Point Representations

We present some examples of properties that help with reasoning about floating point numbers. A small piece of notation is $2C$ means "two's complement." Additionally, while we typically write $+$ these properties are also desired for multiplication.

Table 5: Floating Point Representations

Name	NaN	Infinite	Subnormal	Zero(s)
IEEE	Yes	$+\infty$, $-\infty$	Yes	$+0, -0$
Posit	No	$\pm\infty$	No	0
1750A	No	No	Always	0

1. Monotonicity: if $x, y \in \mathbb{Z}_{2^n}$,

   $$x <_{2C} y \implies \text{f2r}(\text{b2f}(x)) <_{\mathbb{R}} \text{f2r}(\text{b2f}(y)).$$

   Intuitively, we wish typical two's-complement integer comparison operators to work with floating point values.

2. Correct Rounding: this is the lofty goal that round-to-nearest paired with some arithmetic operation, rounding error doesn't accumulate. This is in general false, but knowing the situations in which rounding error accumulates, and what is the bound on that error, is incredibly useful for verification.

3. Sterbenz Lemma: informally, if two numbers are sufficiently close then we can exactly represent their difference (without rounding error).

   Formally, Sterbenz Lemma holds for a given floating point scheme f if

   $$\forall x, y \in \mathbb{R},\ \frac{y}{2} \le x \le 2y \implies \text{round}(x - y) = \text{round}(x) -_f \text{round}(y).$$

4. Associativity: if $x, y, z \in \mathbb{F}_f$,

   $$(x + y) + z = x + (y + z)$$

   This does not hold in general, but we might want to know what further conditions must hold on x, y, z to ensure associativity, either with addition or multiplication.

5.8 Summary

Reasoning about floating point arithmetic beyond just the well-behaved cases is difficult. There exists no complete unification of the dual interpretation of "floats as bits" and "floats as reals." However, packages such as Flocq are a step in the right direction and we expanded on some formalizations given in Flocq as a starting point to future work.

Subsequently, we mention a concept which goes hand-in-hand with floating point arithmetic on the newest processors: SIMD parallelism.

6 SIMD Parallelism

We describe efforts toward verification of Single-Instruction, Multiple-Data parallelism from Flynn's taxonomy of parallel computing [67]. We choose SIMD because of its increasing prevalence in modern ISAs.

The first approach to describing parallelism formally was by Robert Keller in 1976 [92]. He described a program as a set of states Q and a transition relation $\rightarrow$, a binary relation on Q. Importantly, for some $q \in Q$, there may be many q' such that $q \rightarrow q'$; these represent the interleavings of execution traces. The groundwork laid by Robert Keller has influenced verification techniques such as TLA+ [97]. However, the verification of only SIMD parallelism is much more limited and most research is done with the KLEE symbolic execution engine.

Existing symbolic execution tools such as KLEE have been used to verify the correctness of vectorized code using a technique called symbolic crosschecking [41]. With this approach, the vectorized code is expressed as symbolic operations, then checked against a serial equivalent. This approach is conservative because it ensures the exact same operations are performed with SIMD code and symbolic code. Interestingly, Collingbourne et al. claim there exists no floating point constraint solvers available. However, in 2001 Michel et al. presented a floating point constraint solver [105]. Collingbourne may not have noticed or considered the work by Michel because of its lack of mechanization and focus on test pattern generation. Another related project is called FloPSy, a floating-point constraint solver for symbolic execution, the underlying technique of KLEE [96]. Regardless, there is not a large amount of work being done in this area.

Instead, researchers interested in performance focus more on higher level error analysis rather than efficient translation of SIMD operations. For examples of such efforts, we direct the reader to the Correctness Workshop at Supercomputing [95] and the US Department of Energy's summit on Correctness for HPC [74].

A common strategy for SIMD parallelization is through compiler directives such as OpenMP or Intel Thread Building Blocks (TBB). Blom, Darabi, and Huisman create a separation logic-based approach to prove independence between loop iterations [20].

Intermediate representations (IRs) are at a surface level attractive for handling verification of parallel programs: heterogeneity of hardware and programming models should hopefully not require different verification techniques when the underlying concepts (happens-before relations, synchronization, atomic operations) are the same. In practice, IRs are often less mature than source programs. Additionally, the implementer must know yet another programming model.

One attempt at using IRs is a symbolic execution-based model checker called CIVL [150]. This compiles various parallel programming paradigms into an intermediate representation, CIVL-C, as long as the input program is written in a C-like language such as CUDA-C, MPI, OpenMP, and can handle hybrid parallelism. This should not be confused with the aforementioned CIVL project from Microsoft Research used to reason about the correctness of concurrent programs [85].

7 Conclusion

We presented a survey of verification techniques for low-level programs. By low-level, we mean code written in binary, assembly, intermediate representations and programs which interact directly (or nearly-directly) with hardware. We began by describing the logics and canonical verification techniques used in all of formal methods (§ 1). In § 2, we discussed proof assistants and some type-theoretical bases for them. In § 3, we listed formal methods tools and how they are used. These provided the background information to explain three applications of formal methods, each to important abstractions in computer science. The first was intermediate representations § 4, the second was floating point arithmetic § 5, and the final was SIMD parallelism § 6. Key abstractions such as high-level programming languages, compilers, and floating point arithmetic have allowed programmers to manage the incredible complexity of modern hardware and software. In this survey, we provided a more complete picture of how computer scientists reason about the correctness of programmers on programs which do not use these critical abstractions.

7.1 Other Formal Methods Surveys

Formal verification encompasses a wide range of disciplines including mathematical logic, type theory, software engineering, and programming languages. A survey of formal methods is described by Woodcock et al. in 2009 [147]. A brief introduction to formal proofs is given by Hales [80]. A previous area exam by Johnson-Freyd focuses on formal methods using temporal logic [90]. Temporal logic refers to logical systems whose propositions are temporally dependent and properties. For example, a temporal invariant for a job scheduler would be, "for all n less than the total nodes of a cluster, a job of size n in the queue will eventually be scheduled."

With large codebases, it often becomes intractable to formerly verify behavior. In these cases, hybrid techniques such as automatic test pattern generation can be employed. A survey of these approaches is provided by Bhadra et al [18].

Additionally, Sandia National Laboratories has published a survey of existing verification tools for both hardware and software [7].

7.2 Formal Methods in the Wild

One complaint about formal methods in general is it is too expensive, either in computer or human time [93]. One

factor in this complaint is many approaches to formal methods are either exponential, such as checking satisfiability, or undecidable, such as proving total correctness. While not considered formal methods, even the effect of statically versus dynamically-typed languages on productivity and code quality is unclear [120] and has a limited amount of research.[4]

Software engineering and formal methods are ultimately human endeavors but to the best of our knowledge, there is no academic study relating code quality, human effort, and the degree of formal methods used. Evidence of its efficacy can be found in the avionics industry which is heavily dependent on formal methods, both in practice and by regulation [102]. That huge companies such as Airbus, AMD, and Intel as well as the French and United States [3] governments support formal methods hints not only at its profitability but of its value to society.

Slowly but steadily, the percolation of computing into our lives is causing computer scientists and software engineers alike to realize the importance of writing correct software and formal methods' central role in achieving this goal.

7.3 Future Work

One thing which is lacking with floating point verification is the unification of "floats as bits" and "floats as reals." Flocq seems like a nice formalization but also has a lot of overhead in order to get the nice theorems we want; the binary representation of IEEE 754 is 2,100 lines of Coq, describing when something is NaN, infinity, finite, etc. but also providing semantics and proving correctness of square root, negation, absolute value, comparison, truncation, rounding, multiplication, division, addition, and subtraction. For strictly "floats as bits," SMT solvers may suffice but must be reconciled with the real analysis of Flocq. Additionally, I'm interested in applying Ltac, Coq's tactic language, to make the generation of these proofs less painful for the MIL-STD-1750A. Ultimately, I aim to formalize floating point numbers such that IEEE 754, MIL-STD-1750A, and Posit representations can all be described using the same representation.

My summer research at Sandia in 2018 focused on a project called Quameleon which translates assembly and binary programs into its own intermediate representation QIL, then runs analyses using various backends such as symbolic executors or Hoare logic-style predicate transformers. As of January 2019, Quameleon supports to various degrees the M6800, MIL-STD-1705A, and MIPS32 instruction set architectures.

In the longer term, my goal is to look at floating point SIMD compiler transformations which are error-aware since existing approaches are either too conservative by enforcing identical ordering, or are too liberal by assuming associative floating point arithmetic [41]. We direct the reader once again to Figure 1 and its three arrows, each roughly corresponding to a chapter of my proposed dissertation. Quameleon will be a tool and IR used to reason about assembly language programs in a hardware-agnostic fashion. Formalization of floating point arithmetic will use numerical analysis and formal methods to reason about assembly language programs. Vectorization techniques require both numerical analysis and knowledge of assembly language semantics to reason about the correctness of parallel programs while also maintaining the best efficiency. These together will provide computer scientists with better techniques to verify low-level programs.

[4] A collection of other surveys are given at `https://danluu.com/empirical-pl/`

Acknowledgements

I am grateful to Philip Johnson-Freyd and Geoff Hulette for taking a chance hiring me as an intern, funding me, and for first getting me hooked on formal methods. Of course, thanks to my advisor Boyana Norris for supporting me while I veer (mostly) away from HPC. Thanks also Zena Ariola and Hank Childs for reviewing my work.

References

[1] Toccata: Formally verified programs, certified tools and numerical computations. Website at `http://toccata.lri.fr/fp.en.html`.

[2] Affeldt, R., and Marti, N. An approach to formal verification of arithmetic functions in assembly. In *11th Asian Computing Science Conference* (Tokyo, Japan, Dec. 2006), M. Okada and I. Satoh, Eds., Advances in Computer Science — ASIAN 2006. Secure Software and Related Issues (LNCS 4435), Springer, pp. 346–360.

[3] Amla, N., Banerjee, A., Chaudhary, V., Cosley, D. R., Donlon, J., Fisher, D. L., Greenspan, S., and Khan, S. Formal methods in the field (FMitF): Program solicitation. Tech. rep., Alexandria, VA, USA, Sept. 2018.

[4] an Arthur Flatau, D. M. R. Mechanical verification of register-transfer logic: A floating-point multiplier. In *Computer-Aided Reasoning: ACL2 Case Studies*, M. Kaufmann, P. Manolios, and J. S. Moore, Eds. Kluwer Academic Publishers, Dordrecht, Netherlands, June 2000.

[5] Anand, A., and Rahli, V. Towards a formally verified proof assistant. In *Interactive Theorem Proving* (Vienna, Austria, July 2014), G. Klein and R. Gamboa, Eds., ITP (LNCS

8558), Springer International Publishing, pp. 27–44. Repository available at `https://github.com/vrahli/NuprlInCoq`.

[6] Appel, A. W. Verified software toolchain. In *Proceedings of the 20th European Conference on Programming Languages and Systems* (Saarbrücken, Germany, Mar. 2011), ESOP/ETAPS (LNCS 6602), Springer, pp. 1–17.

[7] Armstrong, R. C., Punnoose, R. J., Wong, M. H., and Mayo, J. R. Survey of existing tools for formal verification. Tech. rep., Sandia National Laboratories, Albuquerque, NM, USA, Dec. 2014.

[8] Ayewah, N., Hovermeyer, D., Morgenthaler, J. D., Penix, J., and Pugh, W. Using static analysis to find bugs. *IEEE Software 25*, 5 (Sept. 2008), 22–29.

[9] Baldoni, R., Coppa, E., D'Elia, D. C., Demetrescu, C., and Finocchi, I. A survey of symbolic execution techniques. *ACM Computing Surveys 51*, 3 (July 2018), 50:1–50:39.

[10] Ball, T., Majumdar, R., Millstein, T., and Rajamani, S. K. Automatic predicate abstraction of C programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (Snowbird, UT, USA, June 2001), PLDI '01, ACM, pp. 203–213.

[11] Barendregt, H. P. Lambda calculi with types. In *Handbook of Logic in Computer Science*, S. Abramsky, D. M. Gabbay, and S. E. Maibaum, Eds., vol. 2. Oxford University Press, Inc., New York, NY, USA, 1992, pp. 117–309.

[12] Barnes, J. *High Integrity Software: The SPARK Approach to Safety and Security.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[13] Barnett, M., Chang, B.-Y. E., DeLine, R., Jacobs, B., and Leino, K. R. M. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects* (Amsterdam, Netherlands, Nov. 2006), F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, Eds., FMCO (LNCS 4111), Springer, pp. 364–387.

[14] Barrett, C., Conway, C. L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., and Tinelli, C. Cvc4. In *23rd International Conference on Computer Aided Verification* (Snowbird, UT, USA, 2011), G. Gopalakrishnan and S. Qadeer, Eds., CAV (LNCS 6806), Springer, pp. 171–177.

[15] Barrett, C., Fontaine, P., and Tinelli, C. The SMT-LIB standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories* (Edinburgh, UK, July 2010), CAV 2010 and SAT 2010, pp. 1–14.

[16] Barrett, C., and Tinelli, C. Satisfiability modulo theories. In *Handbook of Model Checking*, E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds. Springer International Publishing AG, Basel, Switzerland, 2018, pp. 305–343.

[17] Barrett, G. Formal methods applied to a floating-point number system. *IEEE Transactions on Software Engineering 15*, 5 (May 1989), 611–621.

[18] Bhadra, J., Abadir, M. S., Wang, L.-C., and Ray, S. A survey of hybrid techniques for functional verification. *IEEE Design Test of Computers 24*, 2 (Mar. 2007), 112–122.

[19] Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. *Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software.* LNCS 2566. Springer, Berlin, Heidelberg, 2002, pp. 85–108.

[20] Blom, S., Darabi, S., and Huisman, M. Verification of loop parallelisations. In *Fundamental Approaches to Software Engineering* (London, UK, 2015), A. Egyed and I. Schaefer, Eds., FACE (LNCS 9033), Springer, pp. 202–217.

[21] Bobot, F., and Filliâtre, J.-C. Separation predicates: A taste of separation logic in first-order logic. In *Formal Methods and Software Engineering* (Kyoto, Japan, Nov. 2012), T. Aoki and K. Taguchi, Eds., ICFEM (LNCS 7635), Springer, pp. 167–181.

[22] Bobot, F., Filliâtre, J.-C., Marché, C., and Paskevich, A. Why3: Shepherd your herd of provers. In *23rd International Conference on Automated Deduction* (Wroclaw, Poland, 2011), First International Workshop on Intermediate Verification Languages: Boogie, pp. 53–64.

[23] Boldo, S., Lelay, C., and Melquiond, G. Formalization of real analysis: A survey of proof assistants and libraries. Tech. rep., Laboratoire de Recherche en Informatique, Université Paris-Sud, Apr. 2013. Working paper.

[24] Boldo, S., and Melquiond, G. Flocq: A unified library for proving floating-point algorithms in coq. In *Proceedings of the 20th IEEE Symposium on Computer Arithmetic* (Tübingen, Germany, July 2011), E. Antelo, D. Hough, and P. Ienne, Eds., ARITH '11, IEEE Computer Society, pp. 243–252.

[25] Boldo, S., and Melquiond, G. *Computer Arithmetic and Formal Proofs: Verifying Floating-Point Algorithms with the Coq System*, 1st ed. ISTE Press - Elsevier, UK, Nov. 2017.

[26] Bove, A., Dybjer, P., and Norell, U. A brief overview of agda – a functional language with dependent types. In *Theorem Proving in Higher Order Logics* (Munich, Germany, Aug. 2009), S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, Eds., TPHOLs (LNCS 5674), Springer, pp. 73–78.

[27] Boyer, R. S., Elspas, B., and Levitt, K. N. SELECT–a formal system for testing and debugging programs by symbolic execution. In *Proceedings of the International Conference on Reliable Software* (Los Angeles, CA, USA, Apr. 1975), ACM, pp. 234–245.

[28] Brady, E. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming 23*, 5 (2013), 552âĂŞ–593.

[29] Brain, M., D'Silva, V., Griggio, A., Haller, L., and Kroening, D. Deciding floating-point logic with abstract conflict driven clause learning. *Formal Methods in System Design 45*, 2 (Oct. 2014), 213–245.

[30] Brain, M., Tinelli, C., Ruemmer, P., and Wahl, T. An automatable formal semantics for IEEE-754 floating-point arithmetic. In *IEEE 22nd Symposium on Computer Arithmetic* (Lyon, France, June 2015), ARITH 22, pp. 160–167.

[31] Brumley, D., Jager, I., Avgerinos, T., and Schwartz, E. J. BAP: A binary analysis platform. In *23rd International Conference on Computer Aided Verification* (Snowbird, UT, USA, July 2011), CAV (LNCS 6806), Springer, pp. 463–469.

[32] Bruttomesso, R., Pek, E., Sharygina, N., and Tsitovich, A. The OpenSMT solver. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Paphos, Cyprus, Mar. 2010), TACAS '10, Springer-Verlag, pp. 150–153.

[33] Buchwald, S. Optgen: A generator for local optimizations. In *Proceedings of the 24th International Conference on Compiler Construction* (Apr. 2015), B. Franke, Ed., CC (LNCS 9031), Springer, pp. 171–189.

[34] Cadar, C., Dunbar, D., and Engler, D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, CA, USA, Dec. 2008), OSDI '08, USENIX Association, pp. 209–224.

[35] Chlipala, A. Mostly-automated verification of low-level programs in computational separation logic. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, CA, USA, June 2011), PLDI '11, ACM, pp. 234–245.

[36] Chlipala, A. The bedrock structured programming system: Combining generative metaprogramming and hoare logic in an extensible program verifier. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (Boston, MA, USA, Sept. 2013), ICFP '13, ACM.

[37] Chlipala, A. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press, 2013.

[38] Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., and Tacchella, A. Nusmv 2: An opensource tool for symbolic model checking. In *Computer Aided Verification* (Copenhagen, Denmark, July 2002), E. Brinksma and K. G. Larsen, Eds., CAV (LNCS 2404), Springer, pp. 359–364.

[39] Claessen, K., and Hughes, J. Quickcheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming* (Montreal, Canada, Sept. 2000), ICFP '00, ACM, pp. 268–279.

[40] Clarke, L. A. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering 2*, 3 (Sept. 1976), 215–222.

[41] Collingbourne, P., Cadar, C., and Kelly, P. H. J. Symbolic crosschecking of floating-point and simd code. In *Proceedings of the Sixth Conference on Computer Systems* (Salzburg, Austria, Apr. 2011), EuroSys '11, ACM, pp. 315–328.

[42] Conchon, S., Contejean, E., Kanig, J., and Lescuyer, S. Lightweight integration of the ergo theorem prover inside a proof assistant. In *Proceedings of the Second Workshop on Automated Formal Methods* (Atlanta, GA, USA, 2007), AFM '07, ACM, pp. 55–59.

[43] Conchon, S., Iguernelala, M., Ji, K., Melquiond, G., and Fumex, C. A three-tier strategy for reasoning about floating-point numbers in smt. In *29th International Conference*

on Computer Aided Verification (Heidelberg, Germany, July 2017), V. Kuncak and R. Majumdar, Eds., CAV (LNCS 10426, 10427), Springer, pp. 419–435.

[44] Conchon, S., Melquiond, G., Roux, C., and Iguernelala, M. Built-in treatment of an axiomatic floating-point theory for smt solvers. In *10th International Workshop on Satisfiability Modulo Theories* (Manchester, United Kingdom, June 2012), P. Fontaine and A. Goel, Eds., SMT '12, pp. 12–21.

[45] Constable, R. L. Computational type theory. *Scholarpedia 4*, 2 (2009), 7618. revision #130876.

[46] Constable, R. L., Allen, S. F., Bromley, H. M., Cleaveland, W. R., Cremer, J. F., Harper, R. W., Howe, D. J., Knoblock, T. B., Mendler, N. P., Panangaden, P., Sasaki, J. T., and Smith, S. F. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

[47] Coquand, T. An analysis of girard's paradox. Tech. Rep. RR-0531, INRIA, May 1986.

[48] Coquand, T., and Huet, G. The calculus of constructions. *Information and Computation 76*, 2 (1988), 95–120.

[49] Cousot, P., and Cousot, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Los Angeles, CA, USA, Jan. 1977), POPL '77, ACM, pp. 238–252.

[50] Cousot, P., and Cousot, R. A galois connection calculus for abstract interpretation. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2014), POPL '14, ACM, pp. 3–4.

[51] Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. The ASTREÉ analyzer. In *European Symposium on Programming Languages and Systems* (Edinburgh, UK, Apr. 2005), M. Sagiv, Ed., ESOP (LNCS 3444), Springer, pp. 21–30.

[52] Cruz-Filipe, L., Geuvers, H., and Wiedijk, F. C-CoRN, the constructive coq repository at nijmegen. In *Proceedings of the 3rd International Conference on Mathematical Knowledge Management* (Bialowieza, Poland, Sept. 2004), MKM, pp. 88–103. Available at `http://corn.cs.ru.nl`.

[53] Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., and Yakobowski, B. Frama-c. In *Software Engineering and Formal Methods* (Thessaloniki, Greece, Oct. 2012), G. Eleftherakis, M. Hinchey, and M. Holcombe, Eds., SFEM (LNCS 7504), Springer, pp. 233–247.

[54] Daumas, M., Rideau, L., and Théry, L. A generic library for floating-point numbers and its application to exact computing. In *14th International Conference on Theorem Proving in Higher Order Logics* (Edinburgh, United Kingdom, Sept. 2001), R. J. Boulton and P. B. Jackson, Eds., TPHOLs (LNCS 2152), Springer, pp. 169–184.

[55] de Bruijn, N. G. *AUTOMATH, a Language for Mathematics*, vol. 2, 1967–1970 of *Automation of Reasoning: Classical Papers on Computational Logic*. Springer, Berlin, Heidelberg, 1983, pp. 159–200.

[56] de Dinechin, F., Lauter, C., and Melquiond, G. Certifying the floating-point implementation of an elementary function using gappa. *IEEE Transactions on Computers 60*, 2 (Feb. 2011), 242–253.

[57] de Dinechin, F., Lauter, C. Q., and Melquiond, G. Assisted verification of elementary functions using gappa. In *Proceedings of the ACM Symposium on Applied Computing* (Dijon, France, 2006), SAC '06, ACM, pp. 1318–1322.

[58] de Moura, L., Kong, S., Avigad, J., Van Doorn, F., and von Raumer, J. The lean theorem prover (system description). In *International Conference on Automated Deduction* (Berlin, Germany, Aug. 2015), CADE-25, Springer, pp. 378–388.

[59] de Moura, L. M., and Bjørner, N. Proofs and refutations, and Z3. In *7th International Workshop on the Implementation of Logics at the 15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning* (Doha, Qatar, Nov. 2008), IWIL LPAR 2008.

[60] Demmel, J., Ahrens, P., and Nguyen, H. D. Efficient reproducible floating point summation and blas. Tech. Rep. UCB/EECS-2016-121, EECS Department, University of California, Berkeley, June 2016.

[61] Dijkstra, E. W. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM 18*, 8 (Aug. 1975), 453–457.

[62] Dowek, G. The undecidability of typability in the lambda-pi-calculus. In *Typed Lambda Calculi and Applications* (Utrecht, Netherlands, Mar. 1993), M. Bezem and J. F. Groote, Eds., TLCA (LNCS 664), Springer, pp. 139–145.

[63] Evangelista, S. High level petri nets analysis with helena. In *Proceedings of the 26th International Conference on Applications and Theory of Petri Nets* (Miami, FL, USA, June 2005), ICATPN '05, Springer-Verlag, pp. 455–464.

[64] Facebook Open Source Community. Infer: A tool to detect bugs in java and c/c++/objective-c code before it ships. Available at `https://fbinfer.com`, 2019.

[65] Filliâtre, J.-C., and Paskevich, A. Why3 — where programs meet provers. In *22nd European Symposium on Programming: Programming Languages and Systems* (Rome, Italy, Mar. 2013), M. Felleisen and P. Gardner, Eds., ESOP (LNCS 7792), Springer, pp. 125–128.

[66] Flanagan, C., and Qadeer, S. Predicate abstraction for software verification. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, OR, USA, 2002), POPL '02, ACM, pp. 191–202.

[67] Flynn, M. J. Some computer organizations and their effectiveness. *IEEE Transactions on Computers C-21*, 9 (Sept. 1972), 948–960.

[68] Foundation, L. Clang static analyzer. `https://clang-analyzer.llvm.org`. Accessed 15 Jan. 2018.

[69] Fromherz, A., Giannarakis, N., Hawblitzel, C., Parno, B., Rastogi, A., and Swamy, N. A verified, efficient embedding of a verifiable assembly language. *Proceedings of the ACM on Programming Languages 3*, POPL (Jan. 2019), 63:1–63:30.

[70] Gérard Huet, G. K., and Paulin-Mohring, C. *The Coq Proof Assistant A Tutorial*. Inria, Rocquencourt, France, Sept. 2018.

[71] Geuvers, H. Proof assistants: History, ideas and future. *Sādhanā: Academy Proceedings in Engineering Sciences 34*, 1 (Feb. 2009), 3–25.

[72] Goldberg, D. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys 23*, 1 (Mar. 1991), 5–48.

[73] Gonthier, G. Formal proof — the four-color theorem. *Notices of the AMS 55*, 11 (2008), 1382–1393.

[74] Gopalakrishnan, G., Hovland, P. D., Iancu, C., Krishnamoorthy, S., Laguna, I., Lethin, R. A., Sen, K., Siegel, S. F., and Solar-Lezama, A. Report of the HPC correctness summit, jan 25-26, 2017, washington DC. Tech. rep., Department of Energy, 2017. Available at `https://arxiv.org/abs/1705.07478`.

[75] Gordon, M., Milner, R., and Wadsworth, C. *Edinburgh LCF*. Springer, Berlin, Heidelberg, 1979.

[76] Gordon, M. J. C. *VLSI Specification, Verification and Synthesis*, vol. 35 of *The Kluwer International Series in Engineering and Computer Science*. Springer, Boston, MA, USA, 1988, ch. HOL: A Proof Generating System for Higher-Order Logic, pp. 73–128.

[77] Graf, S., and Saidi, H. Construction of abstract state graphs with PVS. In *Computer Aided Verification* (Haifa, Israel, June 1997), O. Grumberg, Ed., CAV (LNCS 1254), Springer, pp. 72–83.

[78] Gustafson, J., and Yonemoto, I. Beating floating point at its own game: Posit arithmetic. *Supercomputing Frontiers and Innovations 4*, 2 (2017), 71–86.

[79] Haftmann, F., and Bulwahn, L. Code generation from isabelle/hol theories. Available at `https://isabelle.in.tum.de/doc/codegen.pdf`, Dec. 2013.

[80] Hales, T. C. Formal proof. *Notices of the AMS 55*, 11 (2008), 1370–1380.

[81] Harper, R. *Practical Foundations for Programming Languages*, 2nd ed. Cambridge University Press, New York, NY, USA, 2016.

[82] Harrison, J. Floating-point verification using theorem proving. In *Formal Methods for Hardware Verification* (Bertinoro, Italy, May 2006), M. Bernardo and A. Cimatti, Eds., SFM (LNCS 3965), Springer, pp. 211–242.

[83] Harrison, J. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, Cambridge, UK, 2009.

[84] Harrison, J. Hol light: An overview. In *Theorem Proving in Higher Order Logics* (Munich, Germany, Aug. 2009), S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, Eds., TPHOLs, (LNCS 5674), Springer, pp. 60–66.

[85] Hawblitzel, C., Petrank, E., Qadeer, S., and Tasiran, S. Automated and modular refinement reasoning for concurrent programs. In *27th International Conference on Computer Aided Verification* (San Francisco, CA, USA, July 2015), D. Kroening and C. S. Păsăreanu, Eds., CAV (LNCS 9207), Springer, pp. 449–465.

[86] Henzinger, T. A., Jhala, R., Majumdar, R., and Sutre, G. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, OR, USA, 2002), POPL '02, ACM, pp. 58–70.

[87] Hoare, C. A. R. An axiomatic basis for computer programming. *Communications of the ACM 12*, 10 (Oct. 1969), 576–580.

[88] Holzman, G. J. The model checker SPIN. *IEEE Transactions on Software Engineering 23*, 5 (May 1997), 279–295.

[89] Intel Corporation. Statistical analysis of floating point flaw: Intel white paper. Tech. rep., Intel, July 2004.

[90] Johnson-Freyd, P. A. Refinement and composition in formal modeling of temporal systems. Available at `http://www.cs.uoregon.edu/Reports/AREA-201511-Johnson-Freyd.pdf`, 2015. Area Exam.

[91] Kaplan, D. M. Correctness of a compiler for algol-like programs. Tech. rep., Stanford University, 1967. Stanford Artificial Intelligence Memo No. 48.

[92] Keller, R. M. Formal verification of parallel programs. *Communications of the ACM 19*, 7 (July 1976), 371–384.

[93] Kneuper, R. Limits of formal methods. *Formal Aspects of Computing 9*, 4 (July 1997), 379–394.

[94] Laboratoire de Recherche en Informatique and Inria Saclay Ile-de-France. The alt-ergo automated theorem prover, version 2.2.0, Apr. 2006–2018. `https://alt-ergo.ocamlpro.com`.

[95] Laguna, I., and Rubio-González, C., Eds. *International Workshop on Software Correctness for HPC Applications* (Denver, CO, USA, Nov. 2017), ACM.

[96] Lakhotia, K., Tillmann, N., Harman, M., and de Halleux, J. FloPSy — search-based floating point constraint solving for symbolic execution. In *IFIP International Conference on Testing Software and Systems* (Natal, Brazil, Nov. 2010), A. Petrenko, A. Simão, and J. C. Maldonado, Eds., ICTSS (LNCS 6435), Springer, pp. 142–157.

[97] Lamport, L. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[98] Lattner, C., and Adve, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization* (Palo Alto, CA, USA, Mar. 2004), CGO '04, IEEE Computer Society.

[99] Leino, K. R. M. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning* (Dakar, Senegal, Apr. 2010), E. M. Clarke and A. Voronkov, Eds., LPAR (LNCS 6355), Springer, pp. 348–370.

[100] Lensink, L., Smetsers, S., and van Eekelen, M. Generating verifiable java code from verified pvs specifications. In *Proceedings of the 4th International Conference on NASA Formal Methods* (Norfolk, VA, USA, Apr. 2012), NFM'12, Springer, pp. 310–325.

[101] Leroy, X. Formal verification of a realistic compiler. *Communications of the ACM 52*, 7 (July 2009), 107–115.

[102] Leroy, X. In search of software perfection. Available at `https://youtu.be/lAU5hx_3xRc`, Nov. 2016.

[103] Lopes, N. P., Menendez, D., Nagarakatte, S., and Regehr, J. Provably correct peephole optimizations with alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA, 2015), PLDI '15, ACM, pp. 22–32.

[104] Marques-Silva, J., Lynce, I., and Malik, S. Conflict-driven clause learning sat solvers. In *Handbook of Satisfiability*, A. Biere, M. Heule, H. van Maaren, and T. Walsch, Eds. IOS Press, Amsterdam, Netherlands, 2008, ch. 4, pp. 127–148.

[105] Michel, C., Rueher, M., and Lebbah, Y. Solving constraints over floating-point numbers. In *Principles and Practice of Constraint Programming* (Paphos, Cyprus, 2001), T. Walsh, Ed., CP (LNCS 2239), Springer, pp. 524–538.

[106] Moore, J. S. A mechanically verified language implementation. *Journal of Automated Reasoning 5*, 4 (Dec. 1989), 461–492.

[107] Moore, J. S. *Piton: A Mechanically Verified Assembly-Level Language*, 1st ed., vol. 3 of *Automated Reasoning Series*. Springer Netherlands, 1996.

[108] Moschovakis, J. Intuitionistic logic. In *The Stanford Encyclopedia of Philosophy*, E. N. Zalta, Ed., winter 2018 ed. Metaphysics Research Lab, Stanford University, 2018.

[109] Mullen, E., Zuniga, D., Tatlock, Z., and Grossman, D. Verified peephole optimizations for compcert. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA, June 2016), PLDI '16, ACM, pp. 448–461.

[110] Necula, G. C. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (Vancouver, BC, Canada, 2000), PLDI '00, ACM, pp. 83–94.

[111] Nethercote, N., and Seward, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, CA, USA, June 2007), PLDI '07, ACM, pp. 89–100.

[112] Neumann, P. Mariner i — no holds BARred. *Forum on Risks to the Public in Computers and Related Systems 8*, 75 (May 1989).

[113] Nipkow, T., Wenzel, M., and Paulson, L. C. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, Berlin, Heidelberg, 2002.

[114] Norell, U. *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden, September 2007.

[115] Perconti, J. T., and Ahmed, A. Verifying an open compiler using multi-language semantics. In *European Symposium on Programming Languages and Systems* (Grenoble, France, Apr. 2014), Z. Shao, Ed., ESOP (LNCS 8410), Springer, pp. 128–148.

[116] Petri, C. A., and Reisig, W. Petri net. *Scholarpedia 3*, 4 (2008), 6477. revision #91647.

[117] Pfenning, F., and Schürmann, C. System description: Twelf — a meta-logical framework for deductive systems. In *16th International Conference on Automated Deduction* (Trento, Italy, July 1999), CADE (LNCS 1632), Springer, pp. 202–206.

[118] Pierce, B. C. *Types and Programming Languages*, 1st ed. The MIT Press, 2002.

[119] Pierce, B. C., de Amorim, A. A., Casinghino, C., Gaboardi, M., Greenberg, M., Hriţcu, C., Sjöberg, V., and Yorgey, B. *Software Foundations Volume 1: Logical Foundations*. University of Pennsylvania, 2018. Updated 25 Aug 2018.

[120] Ray, B., Posnett, D., Filkov, V., and Devanbu, P. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China, Nov. 2014), FSE 2014, ACM, pp. 155–165.

[121] Reid, A. Trustworthy specifications of ARM v8-A and v8-M system level architecture. In *Proceedings of Formal Methods in Computer-Aided Design* (Mountain View, CA, USA, Oct. 2016), FMCAD, FMCAD Inc, pp. 161–168.

[122] Reynolds, J. C. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science* (Copenhagen, Denmark, July 2002), LICS '02, IEEE Computer Society, pp. 55–74.

[123] Rosen, B. K., Wegman, M. N., and Zadeck, F. K. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, CA, USA, Jan. 1988), POPL '88, pp. 12–27.

[124] Rümmer, P., and Wahl, T. An smt-lib theory of binary floating-point arithmetic. In *8th International Workshop on Satisfiability Modulo Theories (SMT)* (Edinburgh, UK, July 2010), pp. 151–165.

[125] Russell, S. Unifying logic and probability. *Communications of the ACM 58*, 7 (July 2015), 88–97.

[126] Sadowski, C., Aftandilian, E., Eagle, A., Miller-Cushon, L., and Jaspan, C. Lessons from building static analysis tools at google. *Communications of the ACM 61*, 4 (Mar. 2018), 58–66.

[127] Schordan, M., and Quinlan, D. A source-to-source architecture for user-defined optimizations. In *Modular Programming Languages* (Klagenfurt, Austria, Aug. 2003), L. Böszörményi and P. Schojer, Eds., JMLC (LNCS 2789), Springer, pp. 214–223.

[128] Shapiro, S., and Kouri Kissel, T. Classical logic. In *The Stanford Encyclopedia of Philosophy*, E. N. Zalta, Ed., spring 2018 ed. Metaphysics Research Lab, Stanford University, 2018.

[129] Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., and Vigna, G. SoK: (state of) the art of war: Offensive techniques in binary analysis. In *37th IEEE Symposium on Security and Privacy* (San Jose, CA, USA, May 2016), SP, IEEE Computer Society.

[130] Simmons, R. J. Research projects using twelf. Available at `http://twelf.org/wiki/Research_projects_using_Twelf`, Mar. 2015.

[131] Slind, K., and Norrish, M. A brief overview of hol4. In *Theorem Proving in Higher Order Logics* (Montreal, Canada, Aug. 2008), O. A. Mohamed,

C. Muñoz, and S. Tahar, Eds., TPHOLs (LNCS 5170), Springer, pp. 28–32.

[132] Sørensen, M. H., and Urzyczyn, P. *Lectures on the Curry-Howard Isomorphism*, vol. 149 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, July 2006.

[133] Spivey, J. M. *Understanding Z: a Specification Language and Its Formal Semantics*. Cambridge Tracts in Theoretical Computer Science 3. Cambridge University Press, Cambridge, UK, 1988.

[134] Swamy, N., Chen, J., Fournet, C., Strub, P., Bhargavan, K., and Yang, J. Secure distributed programming with value-dependent types. In *Proceeding of the 16th ACM SIGPLAN International Conference on Functional Programming* (Tokyo, Japan, Sept. 2011), M. M. T. Chakravarty, Z. Hu, and O. Danvy, Eds., ACM, pp. 266–278.

[135] Tagliavini, G., Mach, S., Rossi, D., Marongiu, A., and Benin, L. A transprecision floating-point platform for ultra-low power computing. In *Design, Automation Test in Europe Conference Exhibition* (Dresden, Germany, Mar. 2018), DATE, pp. 1051–1056.

[136] The Coq Development Team. The coq proof assistant, version 8.8.0, Apr. 2018.

[137] The Free Software Foundation. The GNU C library manual. Tech. rep., Aug. 2018.

[138] Torlak, E. Symbolic execution. Lecture Slides at the University of Washington, 2016. Available at `https://courses.cs.washington.edu/courses/cse403/16au/lectures/L16.pdf`.

[139] Turing, A. M. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society s2-42*, 1 (1937), 230–265.

[140] United States Department of Defense. Military standard: Sixteen-bit computer instruction set architecture. Tech. Rep. MIL-STD-1750A, 1980.

[141] U.S. Government Accountability Office. Patriot missile defense: Software problem led to system failure at dhahran, saudi arabia. Tech. Rep. IMTEC-92-26, Feb. 1992.

[142] Vanhoef, M., and Piessens, F. Key reinstallation attacks: Forcing nonce reuse in wpa2. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, TX, USA, Oct. 2017), CCS '17, ACM, pp. 1313–1328.

[143] von Plato, J. The development of proof theory. In *The Stanford Encyclopedia of Philosophy*, E. N. Zalta, Ed., winter 2018 ed. Metaphysics Research Lab, Stanford University, 2018.

[144] Watt, C. Mechanising and verifying the webassembly specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Los Angeles, CA, USA, Jan. 2018), CPP, ACM, pp. 53–65.

[145] Whitehead, N., and Fit-Florea, A. Precision and performance: Floating point and IEEE 754 compliance for NVIDIA GPUs. Tech. rep., NVIDIA Corporation, Oct. 2018. Available at `https://docs.nvidia.com/pdf/Floating_Point_on_NVIDIA_GPU.pdf`.

[146] Wiles, A. Modular elliptic curves and fermat's last theorem. *Annals of Mathematics 141*, 3 (1995), 443–551.

[147] Woodcock, J., Larsen, P. G., Bicarregui, J., and Fitzgerald, J. Formal methods: Practice and experience. *ACM Computing Surveys 41*, 4 (Oct. 2009), 19:1–19:36.

[148] Working Group for Floating-Point Arithmetic. Ieee standard for floating-point arithmetic. *IEEE Std 754-2008* (Aug. 2008), 1–70.

[149] Zhao, J., Nagarakatte, S., Martin, M. M. K., and Zdancewic, S. Formalizing the LLVM intermediate representation for verified program transformations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA, Jan. 2012), POPL '12, ACM, pp. 427–440.

[150] Zheng, M., Rogers, M. S., Luo, Z., Dwyer, M. B., and Siegel, S. F. CIVL: Formal verification of parallel programs. In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering* (Washington, DC, USA, 2015), ASE '15, IEEE Computer Society, pp. 830–835.