QED at Large: A Survey of Engineering of Formally Verified Software

Talia Ringer[1], Karl Palmskog[2], Ilya Sergey[3], Milos Gligoric[4] and Zachary Tatlock[5]

[1] *University of Washington; tringer@cs.washington.edu*
[2] *University of Texas at Austin; palmskog@utexas.edu*
[3] *Yale-NUS College; ilya.sergey@yale-nus.edu.sg*
[4] *University of Texas at Austin; gligoric@utexas.edu*
[5] *University of Washington; ztatlock@cs.washington.edu*

ABSTRACT

Development of formal proofs of correctness of programs can increase actual and perceived reliability and facilitate better understanding of program specifications and their underlying assumptions. Tools supporting such development have been available for over 40 years, but have only recently seen wide practical use. Projects based on construction of machine-checked formal proofs are now reaching an unprecedented scale, comparable to large software projects, which leads to new challenges in proof development and maintenance. Despite its increasing importance, the field of proof engineering is seldom considered in its own right; related theories, techniques, and tools span many fields and venues. This survey of the literature presents a holistic understanding of proof engineering for program correctness, covering impact in practice, foundations, proof automation, proof organization, and practical proof development.

Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric and Zachary Tatlock (2019), "QED at Large: A Survey of Engineering of Formally Verified Software", Foundations and Trends® in Programming Languages: Vol. 5, No. 2-3, pp 102–281. DOI: 10.1561/2500000045.

The version of record is available at: http://dx.doi.org/10.1561/2500000045

1

Introduction

A formal proof of program correctness can show that for all possible inputs, the program behaves as expected. This theoretical guarantee can provide practical benefits. For example, the formally verified optimizing C compiler CompCert (Leroy, 2006) is empirically more reliable than GCC and LLVM: the test generation tool Csmith (Yang *et al.*, 2011) found 79 bugs in GCC and 202 bugs in LLVM, but was unable to find any bugs in the verified parts of CompCert.

Methodologies for developing proofs of program correctness are as old as the proofs themselves (Turing, 1949; Floyd, 1967; Hoare, 1971). These proofs were on paper and of simple programs; tools to support their development followed soon after (Milner, 1972) and have continued to evolve for over 40 years (Bjørner and Havelund, 2014). Projects based on construction of formal, machine-checked proofs using these tools are now reaching a scale comparable to that of large software engineering projects. For example, the initial correctness proofs for an operating system kernel took around 20 person years to develop (Klein *et al.*, 2009), and as of 2014 consisted of 480,000 lines of specifications and proof scripts (Klein *et al.*, 2014).

This survey covers the timeline and research literature concerning proof development for program verification, including theories, languages, and tools. It emphasizes challenges and breakthroughs at each stage in history and highlights challenges that are currently present due to the increasing scale of proof developments.

1.1 Challenges at Scale

Scaling up leads to new challenges and additional demand for tool support in proof development and maintenance. For example, users may have to reformulate properties to facilitate library reuse (Hales *et al.*, 2017), or to encode data structures in specific ways to aid in automation of proofs about them (Gonthier, 2008). Proof development environments need to allow users to efficiently write, check, and share proofs (Faithfull *et al.*, 2018); proof libraries need to allow easy search and seamless integration of results into local developments (Gauthier and Kaliszyk, 2015). Evolving projects face the possibility of previous proofs breaking due to seemingly unrelated changes, justifying design principles (Woos *et al.*, 2016) as well as support for quick error detection (Celik *et al.*, 2017) and repair (Ringer *et al.*, 2018).

The research community has answered these challenges with theories, techniques, and tools for proofs of program correctness that scale—all of which fall under the umbrella of *proof engineering*, or software engineering for proofs. Many of these techniques draw inspiration from work in software engineering on large-scale development practices and tools (Klein, 2014). However, even with close conceptual ties between construction of programs and proofs, research in software engineering requires careful translation to the world of formal proofs. For example, proof engineers can benefit from regression testing techniques by considering lemmas and their proofs in place of tests, as in *regression proving* (Celik *et al.*, 2017); yet, the standard metric used to prioritize regression tests—statement coverage—has no clear analogue for lemmas with complex conditions and quantification.

This survey serves to gather these theories, techniques, and tools into a single place, drawing parallels to software engineering, and pointing out challenges that are especially pronounced in proof development.

It discusses the problems engineers encounter when verifying large systems, existing solutions to these problems, and future opportunities for research to address underserved problems.

1.2 Scope: Domain and Literature

We consider proof engineering research in the context of interactive theorem provers (ITPs) or *proof assistants* (used interchangeably with ITPs in this survey) that satisfy the *de Bruijn criterion* (Barendregt and Barendsen, 2002; Barendregt and Wiedijk, 2005), which requires that they produce proof objects that a small proof-checking kernel can verify; the general workflow of such tools is illustrated in Figure 1.1. That is, we consider proof assistants such as Coq (Coq Development Team, 1989-2019), Isabelle/HOL (Isabelle Development Team, 1994-2019), HOL Light (HOL Light Development Team, 1996-2019), and Agda (Agda Development Team, 2007-2019); we do not consider program verifiers, theorem provers, and constraint solvers such as Dafny (Leino, 2010), ACL2 (ACL2 Development Team, 1990-2019), and Z3 (Z3 Development Team, 2008-2019) except when contributions carry over. We focus on proof engineering for software verification, but consider contributions from mathematics and other domains when relevant.

Sometimes, the key design principles in *engineering* a large program verification effort are not the focus of the most well-known publications on the effort. Instead, they can be in less standard references such as workshop papers (Komendantskaya *et al.*, 2012; Paulson and Blanchette, 2012; Pit-Claudel and Courtieu, 2016; Mulhern, 2006), invited talks (Wenzel, 2017a), blog posts (*Verified cryptography for Firefox 57* n.d.), and online documents (Leroy, 2017; Wenzel, 2017b). One purpose of this survey is to bring such design principles front-and-center. Naturally, we shall aim to survey the relevant literature with our best effort to provide accurate and thorough citations. To that end, we will not hesitate to cite both traditional research papers in well-known venues and relevant discussions in less traditional forms, without further distinction among them.

1.3 Overview

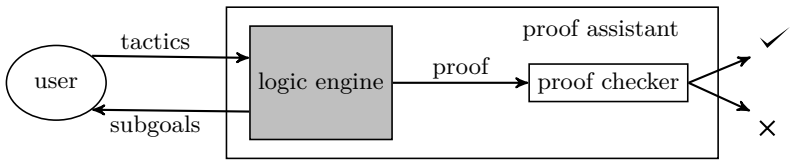

Figure 1.1: Typical proof assistant workflow, adapted from Geuvers (2009).

After motivating chapters (Chapters 2 and 3), this survey discusses the history and foundations of proof assistants (Chapter 4). It then surveys proof engineering research under three headings: languages and automation (Chapter 5), proof organization and scalability (Chapter 6), and practical proof development and evolution (Chapter 7). At a glance, Chapter 5 concerns proof automation approaches and languages, Chapter 6 concerns methods to express and organize programs and proofs, and Chapter 7 concerns development processes and tools. Each of these three chapters is divided into sections; each section surveys a more granular area of proof engineering research, then concludes with a discussion of opportunities for future work within that area when applicable. The survey concludes (Chapter 8) with a discussion of opportunities for future work within proof engineering more broadly. In the case of factual errors, an errata may be found on https://proofengineering.org.

1.4 Reading Guide

This survey aims to reach a broad audience of researchers, proof engineers, and community members who are interested in understanding, using, or contributing to proof engineering research. Readers need not be deterred for lack of background knowledge. It is not always necessary to understand previous chapters in order to understand later chapters; readers should feel free to skip sections or chapters, or to consult later chapters or cited resources for more information. This guide lists topics with which basic familiarity is helpful in order to get the most out of the referenced chapters (all chapters unless otherwise specified), along with resources (cited next to ⊠ icons) for the interested reader:

- Programming languages, type systems, and metatheory ✉ (Pierce, 2002; Harper, 2016), including:
  - ITPs ✉ (Geuvers, 2009; Harrison *et al.*, 2014), especially:
    - Coq ✉ (Chlipala, 2013a; Pierce *et al.*, 2014; Bertot and Casteran, 2004)
    - Isabelle/HOL ✉ (Wenzel *et al.*, 2004; Nipkow and Klein, 2014)
  - Automated reasoning ✉ (Bradley and Manna, 2007; Kroening and Strichman, 2008) (Chapters 3 and 5)
  - The Curry-Howard correspondence ✉ (Pfenning, 2010; Sørensen and Urzyczyn, 2006)
  - Dependent and inductive types ✉ (Chlipala, 2013a)
  - Equality ✉ (Chlipala, 2013a; nLab authors, 2019a) (Chapters 3 and 4)
  - Compilers ✉ (Cooper and Torczon, 2011) (Chapters 3, 4, and 6)
- Software engineering ✉ (Shaw *et al.*, 2015)
- Systems ✉ (Anderson and Dahlin, 2014; Cachin *et al.*, 2011) (Chapters 3 and 6)
- Formalized mathematics ✉ (American Mathematical Society, 2008; nLab authors, 2019b) (Chapter 3)

The version of record is available at: http://dx.doi.org/10.1561/2500000045

2

Proof Engineering by Example

To make our notion of engineering of machine-checked proofs of program correctness more concrete, we consider the classic problem of constructing a program that decides whether a string (say, "*aaab*") matches a regular expression (say, a^*b). This problem has been used to demonstrate the usefulness of rigorous proofs of program properties by induction (Harper, 1999; Yi, 2006).

Specification A necessary step towards a correct program is to obtain a suitable *specification* (Section 6.1) of *functional correctness*, in this case, what it means for a string to match a regular expression. We adopt the usual meaning in terms of the theory of regular languages. Consider an enumerable alphabet Σ of characters c, where it is decidable whether two characters are equal. A string s is an element of the set Σ^*, consisting of 0 or more characters from the alphabet. We call any such set of strings (subset of Σ^*) a *regular language*, and refer to the special *empty string* of 0 letters as ϵ. We then define regular expression syntax:

$$r ::= \mathbf{0} \mid \mathbf{1} \mid c \mid r_1 + r_2 \mid r_1 \cdot r_2 \mid r^*$$

We next define a matching relation between regular expressions and strings informally using inductive rules:

$$\frac{}{\epsilon \lhd \mathbf{1}} \qquad \frac{}{c \lhd c} \qquad \frac{s \lhd r_1}{s \lhd r_1 + r_2} \qquad \frac{s \lhd r_2}{s \lhd r_1 + r_2}$$

$$\frac{s \lhd r_1 \quad s' \lhd r_2}{s\,s' \lhd r_1 \cdot r_2} \qquad \frac{}{\epsilon \lhd r^*} \qquad \frac{s \lhd r \quad s' \lhd r^*}{s\,s' \lhd r^*}$$

The intended interpretation of $s \lhd r$, "s matches r," is that s belongs to the regular language defined by r. One way to convince ourselves that $\lhd$ is adequate is by proving that it holds for some cases where we expect s to be in the language of r (e.g., "$aaab$" and a^*b), and that it does not hold for cases where we do not expect this (e.g., "ab" and b^*). Yi (2006) defines equationally a function which purports to decide whether $s \lhd r$. We write $r!s$ for the application of this function to given r and s, omitting cases that include the Kleene star (r^*):

$$\mathbf{0}!s \stackrel{\text{def}}{=} \mathit{false} \qquad \mathbf{1}!s \stackrel{\text{def}}{=} s = \epsilon \qquad c!s \stackrel{\text{def}}{=} s = c \qquad r_1 + r_2!s \stackrel{\text{def}}{=} r_1!s \vee r_2!s$$

$$\epsilon \cdot r_2!cs \stackrel{\text{def}}{=} r_2!cs \qquad c \cdot r_2!cs \stackrel{\text{def}}{=} r_2!s \qquad (r_1' \cdot r_2') \cdot r_2!cs \stackrel{\text{def}}{=} r_1' \cdot (r_2' \cdot r_2)!cs$$

$$r_1 \cdot r_2!\epsilon \stackrel{\text{def}}{=} r_1!\epsilon \wedge r_2!\epsilon \qquad (r_1' + r_2') \cdot r_2!cs \stackrel{\text{def}}{=} r_1' \cdot r_2!cs \vee r_2' \cdot r_2!cs$$

Encoding Our approach to specifying and verifying this function in an ITP is to make a *deep embedding* (Section 6.2.5) of the language of regular expressions. Specifically, we encode the language as an inductive datatype (Section 4.3.4) in the Coq proof assistant, parameterized by an arbitrary type `char` of characters:

```
Inductive r : Type := r_zero : r | r_unit : r | r_char : char → r
| r_plus : r → r → r | r_times : r → r → r | r_star : r → r.
```

We consider strings to be simple lists with elements in `char`, i.e., to have type `list char`. This allows us to define s' appended to s ($s\,s'$) as regular list concatenation ($s +\!\!+ s'$). The matching relation becomes:

```
Inductive r_match : list char → r → Prop :=
| r_match_unit  : r_match [] r_unit
| r_match_char  : ∀ c, r_match (c :: []) (r_char c)
| r_match_plus' : ∀ s r1 r2, r_match s r1 → r_match s (r_plus r1 r2)
| r_match_plus2 : ∀ s r1 r2, r_match s r2 → r_match s (r_plus r1 r2)
| r_match_times : ∀ s s' r1 r2,
    r_match s r1 → r_match s' r2 → r_match (s ++ s') (r_times r1 r2)
| r_match_star1 : ∀ r0, r_match [] (r_star r0)
| r_match_star2 : ∀ s s' r0,
    r_match s' (r_star r0) → r_match (s ++  s') (r_star r0).
```

We can now define the specification (type) `acc_t` of a function `acc` that decides the matching relation:

```
Definition acc_p (rs : r char * list char) := r_match (snd rs) (fst rs).
Definition acc_t (rs : r char * list char) := {acc_p rs}+{¬ acc_p rs}.
```

Verified Implementation Directly writing a computable function with this specification (with the type `acc_t`) requires us to immediately establish termination (Section 4.3.5). However, we more conveniently use a *definitional extension* (Section 4.3.4) to encode the function and prove partial correctness and termination separately (Sozeau, 2010) using a well-founded relation `acc_lt` whose definition we omit:

```
Equations acc (rs : re char * list char) : acc_t rs by wf rs acc_lt :=
 acc (re_zero, _) := right _;
 acc (re_unit, []) := left _; acc (re_unit, _ :: _) := right _;
 acc (re_char c, [c']) := match char_eq_dec c c' with
  left _ ⇒ left _ | right _ ⇒ right _ end;
 acc (re_char _, []) := right _; acc (re_char _, _ :: _) := right _;
 acc (re_plus r1 r2, s) :=
  match acc (r1, s) with left _⇒ left _ | right _ ⇒
   match acc (r2, s) with left _ ⇒ left _ | right _ ⇒ right _ end end;
 (* ... re_times/re_star cases and proof scripts omitted ... *)
```

While our certified matching function follows the equational definition provided by Yi quite closely, we may have made some mistakes. One way to ensure the adequacy of our proof (Section 4.4.1) is to connect it to a Coq theory of regular languages (Doczkal and Smolka, 2018). We use this theory's notion of language given by a regular expression (`re_lang`) to build a matcher that uses our original `acc` function, by converting from the theory's notion of a regular expression:

```
Program Definition acc' (r : regexp char) (w : list char) :
 {w ∈ re_lang r}+{w ∉ re_lang r} :=
  match acc (@eq_comparable char) (regexp2re r, w) with
  left _ ⇒ left _ | right _ ⇒ right _ end.
```

Executable Code Finally, we can extract (Section 4.4.2) the Coq code for the function `acc` to a practical programming language such as OCaml and run it on example regexp-string pairs where we have an intuition whether they should match or not. We could also refine (Section 6.1.2) the code to more performant Coq code, or to an imperative language (Section 6.2.5) embedded in Coq, making it possible to use certified compilation (Section 3.1.1) to assembly language.

The version of record is available at: http://dx.doi.org/10.1561/2500000045

3

Why Proof Engineering Matters

Formal verification of a program can improve actual and perceived reliability. It can help the programmer think about the desired and actual behavior of the program, perhaps finding and fixing bugs in the process (Murray and Oorschot, 2018). It can make explicit which parts of the system are trusted, and further decrease the burden of trust as more of the system is verified.

One noteworthy program verification success story is the CompCert (Leroy, 2006; Leroy, 2009) verified optimizing C compiler. Both the back-end and front-end compilation passes of CompCert have been verified, ensuring the correctness of their composition (Kästner *et al.*, 2017). CompCert has stood up to the trials of human trust: it has been used, for example, to compile code for safety-critical flight control software (França *et al.*, 2011). It has also stood up to rigorous testing: while the test generation tool Csmith (Yang *et al.*, 2011) found 79 bugs in GCC and 202 bugs in LLVM, it was unable to find any bugs in the verified parts of CompCert.

CompCert, however, was not a simple endeavor: The original development comprised of approximately 35,000 lines of Coq code; functionality accounted for only 13% of this, while specifications and proofs accounted

for the other 87%. This is not unusual for large proof developments. The initial correctness proofs for an operating system (OS) kernel in Isabelle/HOL, for example, consisted of 480,000 lines of specifications and proofs (Klein *et al.*, 2014).

Much like software engineering theories, techniques, and tools help software engineers deal with the challenges of programming for scale, so proof engineering helps proof engineers. To highlight the importance of proof engineering, we start by surveying a sample of domains in which proof engineering for program verification has been particularly influential (Section 3.1). We then discuss proof engineering for other domains (Section 3.2), as the lessons learned from those domains transfer to program verification as well. We conclude by describing some examples of practical impact from program verification (Section 3.3).

3.1 Proof Engineering for Program Verification

We discuss a sample of the domains in which proof engineering has had a large impact: certified compilers (Section 3.1.1), low-level systems software (Section 3.1.2), concurrent and distributed systems (Section 3.1.3), and certified solvers and checkers (Section 3.1.4).

3.1.1 Certified Compilers

Development of certified compilers is a classic application of proof assistants (Section 4.2). In spite of their long history, however, certified compilers for practical and widely used programming languages have only started to appear in the last decade. This is mainly due to the sheer size and complexity of the semantics of these languages—challenges that have necessitated developments in proof engineering.

The inertia for those developments came in 2006 with the help of CompCert. This project was a pivotal moment in the history of program verification. Leroy (2006) received the POPL test of time award in 2016 (ACM SIGPLAN, 2016), with the release noting its pivotal role:

> The paper was (and still is) groundbreaking in that it demonstrates the feasibility of using an interactive theorem prover—specifically, Coq—to both program and formally verify a

> realistic compiler ... [it] made a convincing case that theorem-proving technology is mature enough to be applied to the full functional verification of realistic systems, and in so doing heralded a new age of "big verification."

Many projects built on CompCert by, for example, using its C semantics, or by targeting C or any of its intermediate representations. Appel (2011b) developed a program logic based on CompCert's C semantics, allowing Coq users to prove properties of deeply embedded C programs in Coq. These properties then hold for machine code generated by CompCert, via CompCert's main correctness result (Appel *et al.*, 2014). Kästner *et al.* (2017) described many extensions and enhancements to the basic compilation toolchain of CompCert, e.g., translation validation of the process of *linking* machine code to produce object files and executable files. Cao *et al.* (2018) presented a CompCert-based C program verification environment called VST-Floyd in Coq based on separation logic, simplifying the process of specifying and verifying properties that hold down to machine code.

Certified compilers now span broad applications: For example, the COGENT language for system programming and verification is accompanied a certifying compiler (O'Connor *et al.*, 2016) in Isabelle/HOL that produces a proof that the generated C code is correct. The compiler uses a refinement framework (Section 6.1.2) to automate relating the COGENT semantics to the generated code (Rizkallah *et al.*, 2016). The Standard ML language variant CakeML (Kumar *et al.*, 2014) has a verified compiler in HOL4 with a certified machine-code implementation produced by bootstrapping (applying the compiler to itself). Using the Vellvm (Zhao *et al.*, 2012) framework in Coq, proof engineers can reason about transformations on the LLVM intermediate language representation.

Certified compilers have also covered new ground with respect to modularity and compositionality. The IMM (Podkopaev *et al.*, 2019) memory model modularizes certified compilation from high-level concurrent programming languages to different hardware models. The Bedrock (Chlipala, 2013b) intermediate language and verification environment in Coq for low-level programming contains a notion of macros that can

be reasoned about modularly. The Pilsner (Neis *et al.*, 2015) certified compiler in Coq from a higher-order ML-like language to machine code supports programs that can, in contrast to CompCert, be compositionally verified. Compositional CompCert (Stewart *et al.*, 2015) is a variant of CompCert with a correctness theorem that can be applied compositionally.

Proof engineering for certified compilers has had applications directly to the languages these compilers are verified in. For example, there are certified compilers both from HOL4 (Myreen and Owens, 2012) and from Isabelle/HOL (Hupel and Nipkow, 2018) to CakeML; it is possible to produce machine code by composing these compilers with the certified CakeML compiler. Both CertiCoq (Anand *et al.*, 2017) and Œuf (Mullen *et al.*, 2018) describe certified compilers for Coq's specification language Gallina. CertiCoq is an ongoing project to build a certified compiler from Gallina to machine code, using a hierarchy of custom intermediate languages. Œuf presents a certified compiler from a subset of Gallina to assembly code. Both compilers target CompCert intermediate languages: CertiCoq targets Clight, while Œuf targets Cminor. Each of these compilers provide an alternative to untrusted program extraction (Section 4.4.2).

3.1.2 Low-Level Systems Software

In addition to compilers, low-level software such as operating systems, file systems, and network stacks are building blocks of large software systems. In turn, such software relies on interfaces to hardware, and on hardware behavior. Through considerable effort, proof engineers have specified and verified important pieces of systems software and their hardware bases.

In pioneering work, Klein *et al.* (2009) and Klein *et al.* (2014) developed a small general-purpose OS kernel, called seL4, in the C programming language, with correctness proofs in Isabelle/HOL. The proven properties include correctness of interprocess communication, access-control enforcement, and information-flow noninterference. Many verified extensions to seL4 have been proposed since its inception, e.g.,

to ensure hard deadlines are met for system calls, which is required for applications in real-time systems (Sewell *et al.*, 2017).

Verification of OS kernels has brought new developments in proof engineering. For example, as part of the CertiKOS project, Gu *et al.* (2015) presented a framework in Coq for specifying and verifying abstraction layers, which they used to develop several certified OS kernels. In doing so, they introduced the idea of a deep specification (Section 6.2.3). Gu *et al.* (2016) designed and verified a concurrent kernel for the x86 architecture with fine-grained locking.

An Instruction Set Architecture (ISA) provides an important interface to hardware for, .e.g., compilers. Fox and Myreen (2010) developed formal specifications (semantics) of the ISA for ARMv7 in HOL4. Armstrong *et al.* (2019) later used a domain-specific language to provide ISA specifications for Isabelle/HOL, HOL4, and Coq for the ARMv8, RISC-V, and CHERI-MIPS architectures. Morrisett *et al.* (2012) modeled a subset of the x86 ISA in Coq, and used it to build a verified checker for a sandbox policy.

Security policies of systems software are apt targets for verification due to the importance of them being correct. Along these lines, Dam *et al.* (2013) proposed a separation kernel (hypervisor) based on the ARMv7 processor architecture and its formalization by Fox and Myreen, and proved, in HOL4, an information flow security property that ensures OS instances can only communicate via explicit channels. Guanciale *et al.* (2016) proved memory virtualization security in HOL4 for ARMv7.

OS kernel subsystems and formats are additional formalization targets: Bishop *et al.* (2006) and Bishop *et al.* (2018) defined and validated executable specifications in HOL4 for the TCP/IP stack, and Kell *et al.* (2016) formalized the ELF binary format in HOL4 for executables used in Unix-like operating systems such as Linux.

Verification has also reached file systems, such as FSCQ (Chen *et al.*, 2015), a file system with guarantees about crash safety that have been verified in Coq, and DFSCQ (Chen *et al.*, 2017), an efficient crash-safe file system with several verified optimizations. Using the COGENT language and its certifying compiler (Section 3.1.1), Amani *et al.* (2016) developed a file system called BilbyFS in Isabelle/HOL with executable code in C; they also implemented and verified the legacy Linux file

system ext2. Ridge *et al.* (2015) developed a specification of POSIX file systems in HOL4, which they tested against real-world file system behavior.

3.1.3 Concurrent and Distributed Systems

Concurrent and distributed systems can be difficult to develop, understand, and debug. Researchers have developed many different theories and frameworks for reasoning about such systems in proof assistants. For example, FCSL (Sergey *et al.*, 2015) is a framework in Coq for reasoning about fine-grained concurrency, based on a shallow embedding of concurrent programs. The DISEL (Sergey *et al.*, 2017) framework for distributed separation logic in Coq builds on this, using the shallow embedding approach from FCSL. The formalization is extracted to executable OCaml code and run on real hardware. Two different frameworks (Zeller *et al.*, 2014; Gomes *et al.*, 2017) in Isabelle/HOL exist for verifying two different models of CRDTs, replicated datatypes which provide strong eventual consistency guarantees.

Progress in proof engineering for verification of concurrent and distributed systems has had implications for the formalization of practical programming languages and protocols. This is illustrated by the Jung *et al.* (2017) formalization of the imperative and threaded Rust programming language using the Iris (Jung *et al.*, 2018) framework for concurrent separation logic in Coq, and by the Woos *et al.* (2016) implementation and verification of the key correctness property of the Raft consensus algorithm using the Verdi (Wilcox *et al.*, 2015) framework for verification of asynchronous message-passing distributed systems.

3.1.4 Certified Solvers and Checkers

Proof engineers have formalized and proven correct automated solvers for first-order logic and other more restricted logics. Blanchette *et al.* (2018) verified a SAT solver in Isabelle/HOL with conventional features such as clause learning. Schlichtkrull *et al.* (2019) verified a purely functional first-order superposition-based solver in Isabelle/HOL and obtained executable code in Standard ML.

Another line of work in Isabelle/HOL formalizes various model checkers, for example for Linear Temporal Logic (Esparza *et al.*, 2013) and for timed automata (Wimmer and Lammich, 2018).

3.2 Proof Engineering for Other Domains

While this survey focuses on proof engineering for program verification, domains outside of program verification encounter similar proof engineering challenges. The solutions that these communities develop have implications for proof engineering for program verification. We discuss these implications for two domains: mathematics (Section 3.2.1) and programming languages metatheory (Section 3.2.2).

3.2.1 Mathematics

Mathematics is a natural application domain for proof assistants. *Formalized mathematics* is the attempt to formalize mathematical theories in part or in whole using proof assistants, so that proofs can be mechanically checked. Formalized mathematics was one of the first major application domains for proof assistants; several early ITPs and their predecessors were designed with this domain in mind (Section 4.2).

Early formal developments in mathematics arose in the 1990s (Zucker, 1994; Benthem Jutting, 1994; Bancerek, 1990; Boyer, 1994). Since then, there have been many notable developments for formalized mathematics, including the Four Color Theorem (Gonthier, 2008), the Kepler conjecture (Hales *et al.*, 2011; Hales *et al.*, 2017), the fundamental theorem of algebra (Geuvers *et al.*, 2002b), Gödel-Rosser incompleteness (O'Connor, 2005), and the Jorden curve theorem (Hales, 2007). Other interesting mathematical proofs can be found in a comparative overview of different proof assistants for mathematics (Wiedijk, 2006). The QED manifesto (Boyer, 1994) called for a complete database of formalized mathematics. The UniMath (Voevodsky *et al.*, 2011-2019) library is an ongoing attempt to formalize foundations of mathematics (Voevodsky, 2015) in Coq, using homotopy type theory (Section 4.3.2).

This section samples tools and design principles for formalized mathematics, and discusses how they are relevant to program verification.

Design Principles Many proof developments in mathematics are mature and involve a large community of contributors. Several of these developments have style guides for contributors. For example, UniMath library has a style guide that serves to make proofs rigorous, easy to port to other proof assistants, and less fragile, and to standardize and improve appearance and readability. Among other things, the style guide prohibits the addition of new axioms, and encourages the use of tactics whose semantics are well-defined. Similarly, the HoTT library (Bauer *et al.*, 2017) for homotopy type theory contains a style guide which, among other things, encourages uniform naming principles, outlines methods for defining equivalences, describes how to use axioms uniformly, and encourages the use of tactics that have well-defined relationships with the terms they produce.

In addition to style guides, proof engineers have developed design principles to handle certain kinds of problems common in mathematics. Gonthier *et al.* (2013), for example, outlines a number of techniques used in the proof of the Odd Order Theorem.

Wiedijk (2006) compares the styles of seventeen different proof assistants for mathematics. The book is a collection of proofs of the irrationality of $\sqrt{2}$ from users of each of the proof assistants, and a discussion of each of the proof assistants and the proofs in those proof assistants. This comparison can be useful for understanding the tradeoffs of and design considerations in each of the proof assistants.

Tooling Formalized mathematics has also seen the development of tooling to support entire classes of proofs. Notable examples include autarkic computations for algebraic reasoning (Barendregt and Barendsen, 2002), special support for equational reasoning in proof checkers (Barthe *et al.*, 1996), decision procedures for fragments of arithmetic (Section 5.2.1), techniques for reasoning modulo associativity and commutativity (Braibant and Pous, 2011), transport methods (Sections 6.4.2 and 6.4.3), and theory exploration (Section 5.2.1).

Beyond Mathematics In formalized mathematics, user communities of certain frameworks or libraries adhere to style guides and design

principles. These style guides and design principles have different emphases. Proof engineers in communities outside of mathematics may similarly benefit from standardizing some elements of style depending on the desired outcome. On a project-by-project basis, this may make collaboration between proof engineers easier, and limit the accidental introduction of untrusted code.

Style guides and design principles also have implications for proof understanding. Mathematicians are the original proof checkers, so it's perhaps unsurprising that many mathematics communities emphasize proof understanding by humans. Beyond mathematics, human understanding of proofs communicates information to the reader beyond the theorem statement itself. Furthermore, just like in software engineering, effective collaboration between proof engineers hinges on mutual understanding of the underlying code.

Standardizing style may also make automation easier. For example, by limiting the set of tactics used within a community, the produced proofs are more clearly defined, which may make higher-level automation such as refactoring and repair tools (Section 7.2) less challenging.

Proof engineers in communities outside of mathematics may also benefit from comparative studies across different proof assistants, similar to Wiedijk (2006) but for other domains.

Finally, much of the tooling developed for mathematics addresses problems that occur in proof developments outside of mathematics. For example, dealing with equivalences and isomorphisms is a problem that is not exclusive to the domain of mathematics. Many of the techniques and tooling from mathematics that solve this problem may be useful for proof engineers who encounter this same problem in other domains.

3.2.2 Programming Language Metatheory

One large domain of focus is *mechanized metatheory*: proofs about programming languages. The desire to mechanize theory led to the introduction of the Edinburgh Logical Framework (LF) by Harper *et al.* (1987) (and later in more detail, Harper *et al.* (1993)), building on ideas from Automath. LF defined a methodology for encoding and reasoning about a simpler programming language from within the

higher-order dependently typed lambda calculus. Mechanizations of metatheory followed shortly after, both in Nuprl (for example, Howe (1988)) and in LF (see Harper and Licata (2007) for an overview of mechanized metatheory in LF, and Harrison (1995) for an early history of mechanized metatheory more broadly).

Since then, the domain has grown to reach practical languages: The mechanization (Crary and Harper, 2014) of Standard ML in Twelf formalized the metatheory of a practical language in its entirety. WebAssembly has had a formal semantics from the very beginning, which has been mechanized (Watt, 2018) in Isabelle/HOL. Simplified languages representing the core underlying theories of Scala (Rompf and Amin, 2016; Amin and Rompf, 2017) and of OCaml (Owens, 2008) have been verified. Results from mechanized metatheory have also influenced verification of real compilers, like CakeML (Section 3.1.1).

The success of metatheory has brought with it benchmark suites and design principles. In addition, mechanized metatheory has influenced new additions to the core languages of ITPs. This section describes a small sample of those benchmark suites, design principles, and language features, and discusses how the lessons learned from mechanized metatheory generalize beyond this domain.

Benchmark Suites Some of the success of mechanized metatheory is attributable to benchmark suites that have clearly established the importance of the domain and set out to define how to measure progress within it. The POPLMark challenge (Aydemir *et al.*, 2005) has been particularly influential.

The benchmarks in the POPLMark challenge are proofs of properties of the language System F-Sub (Cardelli *et al.*, 1994), which has parametric polymorphism and subtyping. POPLMark highlights specific problems in proof engineering for metatheory, and outlines criteria for evaluating the success of technology that addresses these problems.

15 solutions to the POPLMark challenge remain accessible online (Aydemir *et al.*, 2005-2019). Of these solutions, 8 are in Coq, 2 are in Isabelle/HOL, and the remaining 5 are spread across 5 other ITPs. The solutions cover 8 different ways to represent variable *binders*, one of the problems that POPLMark highlights.

Personal communications (Harper and Crary, 2019; Pierce, 2019) suggest that the solution using Twelf (Pfenning and Schürmann, 1999) (an LF implementation) was the first solution to solve all of the difficult parts of the challenge. The website notes that this solution demonstrated the benefits of using that framework, including the style of binders it supports, while also sparking an interesting discussion on different ways of specifying the problem across different frameworks. Only Arthur Charguéraud attempted the same solution in the same proof assistant with different styles of binders; while his solutions were inconclusive, they inspired later work on making binders easier to represent (Aydemir *et al.*, 2008; Charguéraud, 2011).

The POPLMark solutions may be thought of as a springboard for later work. They provided information about what the state-of-the-art was at the time, which enabled later researchers to measure progress. Over 300 papers have cited POPLMark since its introduction in 2005.

Still, there is some dissatisfaction with the outcomes of POPLMark. For example, mst papers that cite POPLMark focus on the difficulty of dealing with binders, which is just one challenge that proof engineers face when mechanizing metatheory (Pierce, 2017). The List-machine Benchmark (Appel *et al.*, 2012), developed in parallel with POPLMark, deemphasizes binders and instead emphasizes connections between proofs and real implementations of compilers. POPLMark Reloaded (Abel *et al.*, 2017a) emphasizes logical relations proofs. The ORBI (Felty *et al.*, 2018; Felty *et al.*, 2015) benchmarks focus on the tradeoffs of design decisions of different systems for mechanizing metatheory, rather than on different approaches using a given system.

Design Principles Many papers that cite POPLMark focus on difficulties of dealing with binders. Paper proofs typically use a *named* representation, where variables are represented by names. These are easy for humans to reason about, but make it difficult for tools to reason about alpha-equivalence. Nominal logics (Aydemir *et al.*, 2006; Urban, 2008) encode names such that equivalent terms are alpha-equivalent. These representations have the benefits of named representations, but without the cost of difficulty reasoning about alpha-equivalence. While

the nominal approach is common in Isabelle (Urban, 2008), only preliminary approaches exist in Coq (Aydemir *et al.*, 2006). Developing such a tool in Coq may be difficult due to the differences in logics between Coq and Isabelle/HOL, as Nominal Isabelle makes use (Urban and Kaliszyk, 2011) of quotient types (Homeier, 2005) in HOL.

In the absence of a nominal tool for Coq, proof engineers may explore tradeoffs between *de Bruijn indexes* (introduced by de Bruijn (1972) and explored by Owens (2008) and Schäfer *et al.* (2015), among others) and a *locally nameless* representation (introduced by Gordon (1994) and explored by Leroy (2007), Aydemir *et al.* (2008), and Charguéraud (2011), among others). De Bruijn indexes make reasoning about alpha-equivalence simple because alpha-equivalence is definitional equality (Section 4.3.2), but they require shifting operations in the code, which complicates human understanding; Berghofer and Urban (2007) detail the tradeoffs between de Bruijn indices and names. Locally nameless attempts to capture the best of both worlds: it uses names for free variables and de Bruijn indexes for bound variables. Consequentially, locally nameless requires reasoning about indexes and shifting operations only for locally closed terms.

LF-based systems like Twelf (Pfenning and Schürmann, 1999), Delphin (Poswolsky and Schürmann, 2009), Beluga (Pientka and Dunfield, 2008) use *higher-order abstract syntax* (HOAS) (Pfenning and Elliott, 1988) to simplify reasoning about binders. HOAS gives an encoding of binders in the object language (the language that is reasoned about) as binders in the meta-language (the language of reasoning—in the case of LF, the higher-order dependently typed lambda calculus). Beluga uses ideas from contextual modal type theory (Nanevski *et al.*, 2008b) to further simplify reasoning about HOAS encodings. The Hybrid (Ambler *et al.*, 2002) tool makes it possible to use HOAS within Isabelle/HOL; Capretta and Felty (2007) describe a version of Hybrid for Coq, and Felty and Momigliano (2012) describe how to use Hybrid to reason about an object language in a manner similar to Twelf. Felty and Pientka (2010) compare Twelf, Beluga, and Hybrid on case studies of metatheory using HOAS, and presents a set of challenge problems which highlights the differences between these systems. Variants of HOAS such as weak HOAS (Ciaffaglione and Scagnetto, 2012) and *parametric higher-order*

abstract syntax (PHOAS) (Chlipala, 2008) make HOAS more tractable in general-purpose ITPs like Coq.

There is a small amount of work on design principles that address the concerns of POPLMark beyond dealing with binders. For example, Engineering Formal Metatheory (Aydemir *et al.*, 2008) identifies specific lemmas that are useful and discusses the organization of theorems, proofs, and automation. It also introduces *cofinite quantification* of free variables in inductive relations—defining relations that hold on all but finitely many variables, rather than for some fresh variable. This strengthens the premise of the relations, which in turn strengthens inductive hypotheses for proofs.

Design principles for mechanized metatheory often go hand-in-hand with high-level frameworks such as 3MT (Section 6.3.2), or with domain-specific languages such as Ott and Lem (Section 6.1). Other work in design principles for mechanized metatheory includes an overview of different ways of formalizing language semantics in an ITP for the same language (Bertot, 2009), and the use of the coinductive partiality monad (Capretta, 2005) in Agda to define denotational semantics (Danielsson, 2012).

Beyond Metatheory Few domains have seen as much movement in the development of design principles for proof engineering as mechanized metatheory. Opinions on the role that POPLMark played in this are mixed (Pierce, 2017; Appel, 2017). There is little disagreement that POPLMark was timely: Proof assistants were becoming more usable, and the ongoing development of CompCert (Section 3.1.1) inspired confidence in their usefulness. At the same time, the properties that researchers wanted to prove about their languages were becoming larger and more complex. It was becoming difficult to know that these properties were actually correct, and to maintain confidence in correctness in the face of changes.

POPLMark gave a common platform for experimentation and offered a concrete criteria for success in a timely domain. The benchmarks were difficult enough to stress technology, but simple enough that they were easy to understand and that experts could prove them in a few weeks. The work in metatheory that followed POPLMark

demonstrated that general-purpose proof assistants really were usable to prove these properties that researchers cared about.

Work to this day continues to use POPLMark as an evaluation metric. Amin and Rompf (2017), for example, introduce a proof technique using definitional interpreters that addresses the open challenge of scaling type soundness proofs to realistic languages, and evaluate the success of this technique using the F-Sub language from the POPLMark benchmarks. This highlights that sometimes, proving a slightly different property and then showing how that relates to the original property can be much simpler than proving the original property directly.

POPLMark suggests that timely benchmark suites are instrumental in bringing the challenges of design for proof engineering to the attention of the research community; in doing so, however, they can narrow the focus to one particularly difficult problem, sometimes to the exclusion of the bigger picture. Domains outside of metatheory can take this into consideration.

In addition, the success of experts using LF and Twelf on the POPLMark benchmarks and in mechanizing a practical programming language suggests that it is worth weighing carefully the tradeoffs of using different ITPs, including ITPs with special support for a given domain. Along those lines, Miller (2018) argues that handling of variable bindings should be built into ITPs. It is also worth considering the barriers to adoption by non-experts of tools with which experts have demonstrated success within a domain, and how to overcome those barriers. SASyLF (Aldrich *et al.*, 2008), for example, is one attempt to make LF-based ITPs more accessible to students.

3.3 Practical Impact

Proof engineering has already had a large impact on program verification in many domains, including those from Section 3.1. Proof engineers have in recent years verified operating system (Klein *et al.*, 2009) and web browser (Jang *et al.*, 2012) kernels, machine learning systems (Selsam *et al.*, 2017), distributed systems (Woos *et al.*, 2016), quantum circuits (Rand *et al.*, 2017), constraint solvers (Blanchette *et al.*, 2018;

Schlichtkrull *et al.*, 2019), compilers (Leroy, 2006; Kumar *et al.*, 2014), and file systems (Chen *et al.*, 2015; Amani *et al.*, 2016).

So far, proof assistants have had the strongest practical impact in systems software. The CompCert verified compiler, sold as a commercial product, is finding applications in embedded systems, such as those used in aviation (Kästner *et al.*, 2018). The BoringSSL library, used in the popular Google Chrome Web browser, recently started to include high-performance cryptographic code in C verified in Coq (Erbsen *et al.*, 2019). The seL4 verified operating system kernel is used in SCADA systems, and aviation and automotive systems (Klein *et al.*, 2018).

The version of record is available at: http://dx.doi.org/10.1561/2500000045

4

Foundations and Trusted Bases

We give a short overview of the pre-history (Section 4.1) and early history (Section 4.2) of proof assistants, and then discuss their foundations (Section 4.3) and trusted bases (Section 4.4).

4.1 Proof Assistant Pre-History

Specification and verification of software systems can be viewed as reducing human, informal notions and reasoning to systematic application of logical principles and axioms. From this perspective, Aristotle's systematization of the principles of correct reasoning (Aristotle, 1926; Smith, 2018) is arguably the oldest precursor. The proposal of Leibniz (1685) to reduce human reasoning to mathematical calculation is a second important step. Leibniz also laid the foundations for symbolic propositional logic, although this was also done independently by, e.g., Boole.

A later important development was the introduction of predicate logic (or *predicate calculus*) by Frege in the late 19th century (Frege, 1893; Zalta, 2018b). Two key innovations in Frege's logical system were (1) the introduction of *quantifiers* of expressions in propositions, and (2) a notion of proof (sequences of valid inferences) for propositions with

quantifiers. Frege was able to capture and prove concepts from number theory in his system from first principles. However, his system included an axiom later shown by Russell to make the system inconsistent (Zalta, 2018a). Nevertheless, the logics of ITPs based on higher-order logic are reminiscent of Frege's logic (which included second-order quantification), and his notion of proof is similar to the modern conception.

In the early 20th century, Russell and Whitehead continued Frege's work of putting mathematics on a firm logical basis. Crucially, this included developing methods for avoiding inconsistencies, e.g., due to unrestricted formation of sets of entities (Russell, 1918). In the end, they proved many significant theorems of arithmetic and set theory in their logical system by rigorous inference (Whitehead and Russell, 1997), but relied on axioms that were considered questionable at the time (Irvine, 2016); this is echoed in more recent concerns for philosophical justification of the basis for the logical system underpinning a proof assistant (Barras, 2010). Gödel (1930) established the connection between truth and provability for *first-order* predicate logic, showing that proofs of true propositions can always be constructed, in principle (systems of inference rules can be made *complete*). However, he then subsequently established that even modest extensions of expressibility in first-order logic lead to *incompleteness* (Gödel, 1931): there can be no system that allows constructing proofs for all true propositions. Together with other negative results, e.g., by Tarski (1936), this ended the search for a single universal logical system as a foundation for mathematics and all mathematical endeavors.

At roughly the same time, a theoretical basis for computation and computer programs was given by Church (1936) in the λ-calculus, and computers were developed more practically by von Neumann and others in the 1940s (von Neumann, 1993). As pointed out by Backus (1978), the λ-calculus and computers as described by von Neumann gave rise to two distinct program styles: the functional style is characterized by computational steps as reductions of *expressions* and an absence of state, and the imperative style is characterized by computation as transitions between complex states and *statements* that effect such transitions.

Also around that time, Curry (1934) observed a connection between axioms and type systems. This and later observations culminated in 1969

(published in Howard (1980)) with the principle of *formulae-as-types*, also known as *propositions as types*, the Curry-Howard correspondence, or the Curry-Howard isomorphism. This principle established the connection between programs and proofs, which provided groundwork for the later development of ITPs.

Turing (1949) first considered the problem of correctness for an imperative program that computes the factorial of its input by repeated additions. He described how full correctness could be decomposed into verifying assertions associated with certain points in the code (today called *invariants*), and how to ensure program termination by finding a consistently decreasing quantity (today called a *variant* or *ranking function*). However, this work remained obscure, and more systematic approaches for reasoning about imperative programming languages were presented only late in the 1960s (Floyd, 1967; Hoare, 1969).

McCarthy (1960) proposed a practical realization of the functional style of programming in the form of the Lisp language. McCarthy (1963) also highlighted the problem of putting computing and programs on a formal foundation. To this end, he proposed several formalisms for capturing different classes of functions, and showed how to reason about the equivalence of such functions. He also described how datatypes could be constructed recursively and be subject to inductive reasoning. Burstall (1969) showed how to reason about more practical programs in the functional style using the principle of structural induction.

Research on logical reasoning using computers initially took two main forms (Warden and Biancuzzi, 2009): (1) fully automated proofs of propositions in simple proof systems such as Robinson's resolution system (Robinson, 1965), and (2) computer checking of the validity of single steps in human-constructed mathematical proofs, as in the Automath system by de Bruijn (de Bruijn, 1970; de Bruijn, 1994); Automath is notable for representing both propositions and proofs in the same formal system (a variant of the λ-calculus). The former approach is limited by the difficulty (and resulting long machine time) of finding proofs algorithmically and its bounds on expressiveness of propositions, while the latter is limited by the ingenuity (and labor supply) of the humans that construct the proofs that the system checks. The legacy of Automath includes the *de Bruijn principle* (Barendregt and Barendsen,

2002), which states that proof-checking programs should be as small and simple as possible to facilitate high assurance and trustworthiness.

4.2 Proof Assistant Early History

In the early 1970s, Milner proposed an approach to computer proofs in between full automation and basic inference checking. One of his insights was that fine-grained automation can be directed by human ingenuity through so-called *proof tactics* (Section 5.1.1), alleviating the burden on users in Automath-style systems. He also chose an underlying formal system (Scott's logic of computable functions (Scott, 1993)) that could represent concepts familiar to computer scientists and programmers, such as integers, lists, and computer programs themselves (Gordon, 2000). The first implementation of his approach, called Stanford LCF (Milner, 1972), provided a workflow still used in several modern proof assistants, where the user inputs a command (e.g., a single tactic to apply to attempt to reach the current proof goal) and the system executes the command, resulting in a complete proof or in a number of subgoals. Although tactics could be complex, the system guaranteed that a proof reported as finished could be exported and verified independently by an Automath-style checker (Warden and Biancuzzi, 2009).

Limitations on the flexibility and scalability of Stanford LCF prompted Milner to develop ML (Meta Language), a programming language for use in a new version of LCF. ML was a typed language, and Milner defined a theorem in LCF as an abstract data type whose predefined values were instances of axioms and whose operations were inference rules. This technique, which persists in some proof assistants today, is usually referred to as the "LCF approach," and it in effect reduces the soundness of inferences in an embedded logical system to the soundness of the type system (and type checking mechanism) of the host language. Adventurous and flexible tactics could be implemented in ML and applied without concern for affecting soundness, although, e.g., termination was not guaranteed.

The resulting implementation of LCF in ML was called Edinburgh LCF (Gordon *et al.*, 1979), and was further developed mainly by Paulson and Huet, who enhanced its reasoning capabilities and wrote a

compiler for ML to avoid the overhead of interpretation (Gordon, 2000). Paulson then went on to develop the Isabelle proof assistant framework (Paulson, 1994; Paulson, 1993), and Huet to develop, with Coquand, the first version of the Coq proof assistant (Coquand and Huet, 1985). Gordon used the last version of the LCF system, called Cambridge LCF, as a basis for the HOL proof assistant (Gordon and Melham, 1993). The Nuprl proof assistant (Constable *et al.*, 1986) also followed in the LCF tradition. Together, these proof assistants comprise the *LCF family*, and their recent incarnations are now widely used in the research community. Recently developed proof assistants such as RedPRL (RedPRL Development Team, 2015-2018) and Lean (de Moura *et al.*, 2015) have also joined the LCF family. ML was standardized as Standard ML (Milner *et al.*, 1997), and it and its dialects are widely used as implementation languages for proof assistants.

While Automath targeted mathematics, the initial applications of LCF-style systems for verification was in the area of programming languages and compilers. Stanford LCF had case studies for verified compilation of an imperative language to a stack-based language (Milner and Weyhrauch, 1972), and equational theories on integers and lists (Newey, 1973). Edinburgh LCF had case studies for verified programming language implementations (Cohn, 1983), and an important use case of HOL was hardware verification (Boulton *et al.*, 1992).

Geuvers (2009) and Harrison *et al.* (2014) provide a more comprehensive description of the history of ITPs.

4.3 Proof Assistant Foundations

The foundational theories of many ITPs are based on some variation of the theory of *types*, which goes back to Russell and his attempt in the early 1900s to avoid inconsistency in formal systems by forbidding pathological cases such as the set of all sets with some arbitrary property (Irvine and Deutsch, 2016). Specifically, Church (1940) introduced the *simply typed* λ-calculus to avoid inconsistencies in the original λ-calculus. This typed calculus, also referred to as Higher-Order Logic (HOL), is the basis of the proof assistants HOL4 and HOL Light.

While λ-calculus gives an account of computation, the principle of propositions-as-types was a later development, related to the conception of *intuitionistic* mathematics by Brouwer, Heyting, Kolmogorov, and others (Heyting, 1956). A basic tenet of intuitionism (or *constructivism*) is to only admit mathematical objects that can be mentally construed from basic principles; postulation of existence or axiomatization is not enough. At the level of logical reasoning, this leads to intuitionists rejecting certain proofs established by an appeal to the law of excluded middle (LEM)—that all propositions are either true or false. Moreover, intuitionists interpret functions as effective methods of computation rather than, say, relations that satisfy some set of equations. Consequently, many classical mathematical theorems do not hold as typically formulated with such a restricted logic. However, similar theorems turn out to be possible to prove in many cases, as shown, e.g., by Bishop and Bridges (1985). Widely used proof assistants based on intuitionistic type theories, following the tradition of Martin-Löf (1984) and Martin-Löf (1982), include Coq, Agda, and Lean.

Logical frameworks (Harper *et al.*, 1993) support reasoning about many different logics from within a single system. Automath is a logical framework, as is LF (Section 3.2.2). The popular general-purpose proof assistant Isabelle similarly supports many logics, as long as they can be made to conform to underlying framework for simply-typed higher-order natural deduction. While HOL is the most commonly used logic for Isabelle, other bundled logics include first-order logic with Zermelo-Fraenkel set theory, and constructive type theory (Paulson, 1993).

4.3.1 Proof Objects

Barendregt (2013) characterizes proof assistants according to how they deal with *proof objects*, i.e., the certificates that some property is true according to the underlying logic. In proof assistants closely related to LCF such as Isabelle, HOL4, and HOL Light, proof objects are normally not represented in full, but constructed and checked piece by piece, i.e., they are *ephemeral*. In contrast, Coq and Agda produce complete proof objects, although such objects are usually not kept in memory once constructed, but are stored on disk.

The time to construct and validate proof objects (piecemeal) in Isabelle, HOL4, and HOL Light is directly proportional to the object's size. However, checking proofs in Coq, Agda, and other proof assistants that support *reflection* (Boutin, 1997), i.e., computational steps in proofs, may not be proportional to proof object size. In effect, one computational step can take as long as all conventional steps combined.

Barendregt (2007) uses a formula of the following kind to illustrate the usefulness of reflection and its relation to proof object sizes:

$$A \stackrel{\text{def}}{=} p \leftrightarrow (p \leftrightarrow (p \leftrightarrow (p \leftrightarrow (p \leftrightarrow (p \leftrightarrow (p \leftrightarrow (p \leftrightarrow p)))))))$$

Proving A directly requires repeated and tedious use of basic derivation rules. In addition, even if rule application is automated, the proof object will be large. Instead, it is possible to perform the proof indirectly by using computation. To this end, we define:

$$B(1) \stackrel{\text{def}}{=} p \qquad B(n+1) \stackrel{\text{def}}{=} p \leftrightarrow B(n)$$

We then prove by induction on n that whenever $n \geq 1$, we have $B(2 \times n)$. We conclude the proof of A by rewriting using two equalities, $A = B(10)$ and $10 = 2 \times 5$, and apply the fact about B that we just proved. In proof assistants that support reflection, the final proof object contains no trace of the proofs of the two equalities, since they are established using reductions in the logic engine. In proof assistants without reflection, full proofs must be provided for the equalities before they can be used for rewriting, resulting in large proof objects. For example, proof objects in Isabelle/HOL are typically large, but this does not necessarily mean that proof checking is slower overall, since they are ephemeral (Wenzel, 2015). In effect, large proof objects in Isabelle can be viewed as a consequence of deliberate design decisions, e.g., concerning how to perform rewriting, computation, and proof checking.

4.3.2 Equality

In logical systems and type theories, there is a conceptual difference between *definitional equality*, used for proof checking (type checking), and *propositional equality*, used in expressing statements to prove.

In *intensional* type theories, such as the early intuitionistic type theory by Martin-Löf (1984) and the Calculus of Constructions (Coquand

and Huet, 1988) (CoC), these concepts are completely distinct, which can limit what can conveniently be proven to be equal. For example, in Coq, $0 + n = n$ follows by definitional equality, while $n + 0 = n$ requires inductive reasoning. In contrast, in *extensional* type theories, such as that implemented in Nuprl (Constable *et al.*, 1986), definitional and propositional equality coincide. However, this means that proof checking is inherently undecidable, since propositional equality can be used to specify undecidable problems. Intuitively, intensionally equal entities are such that they are "constructed in the same way," while extensionally equal entities "behave in the same way." Proofs in extensional systems can sometimes be translated into intensional theories after adding a few axioms (Oury, 2005).

Even within intensional type theories, not all notions of propositional equality are created equal. Homotopy type theory (Univalent Foundations Program, 2013) (HoTT), for example, is an intensional type theory (nLab authors, 2019c) in which the notion of propositional equality corresponds to type equivalence. A *type equivalence* between types `A` and `B` is a function:

```
f : A → B.
```

for which there exists some function:

```
g: B → A.
```

that is a mutual inverse:

```
section    : ∀ (a : A), g (f a) = a.
retraction : ∀ (b : B), f (g b) = b.
```

Univalence in HoTT states that propositional equality between types is equivalent to type equivalence between those types. Consequentially, in HoTT, it is possible to treat equivalent types as being the same.

Both CoC and HoTT are intensional. In both of these type theories, propositional equality corresponds to inhabitance of the identity type. However, in HoTT, univalence provides a means of constructing a term of the identity type (Escardó, 2018) that is not present in CoC. This has implications for other properties of these intensional type theories. For example, in HoTT, as a consequence of univalence, *functional extensionality* holds: functions can be proven equal merely from the fact

that they always return the same values for the same arguments. This is not true in CoC, though it may be consistently assumed as an axiom.

There are many other weaker equalities than propositional equality that can be useful for reasoning about programs and systems. McBride (2002) proposes a heterogenous equality relation for type theories where terms can be considered equal despite having different types. As Chlipala (2013a) remarks, researchers are continually discovering new ways for entities such as functions and data to be equal.

4.3.3 Predicativity

The term *predicative* was first used by Russell (1906) to describe so-called propositional functions $\phi(x)$ that define a class, i.e., for which the class $\{x : \phi(x)\}$ actually exists. He distinguished such functions from *impredicative* functions for which no such class exists (Feferman, 2005). For example, the propositional function specifying that a class has itself as member does not define a class, and is thus impredicative. In modern logical systems, enforcing predicativity means that when objects are defined using quantifiers, no such quantifier may be instantiated with the object itself (see Chapter 12 of Chlipala (2013a)).

Isabelle's meta-logic stays within predicative simple type theory (Paulson, 1993). In Coq, the `Type` universe (including `Set`) is predicative, but `Prop` is impredicative. Including both of these provides a balance to users of consistency with common axioms and expressivity: Impredicative `Type` with *large elimination* (pattern matching that returns terms of type `Type`) would not be consistent with LEM, which can be added as an axiom in Coq (cody, 2015). On the other hand, there is an informal consensus that the impredicativity of `Prop` adds expressivity which is useful for expressing most mathematical proofs (cody, 2015). Otherwise, in predicative logic, some proofs are more complex (Avigad, 2004), though it is not known to what extent this has practical implications on what it is possible to express in each logic. Thus, Coq includes an impredicative `Prop` universe in which large elimination is disabled.

4.3.4 Definitional Mechanisms

Programs of interest to computer scientists and engineers often involve classic data structures such as lists, trees, and natural numbers. These data structures can be described, e.g., by initial algebras in category theory or in fixpoint theory (Scott, 1970). Most proof assistants provide mechanisms for defining such data structures; these mechanisms take one of three forms (Berghofer and Wenzel, 1999): (1) axiomatic, (2) inherent, and (3) definitional.

In the first approach, taken by early users of the LCF system, datatype constructors are defined by introducing new axioms, from which induction principles are proved (Paulson, 1984). In the second approach, the underlying logic is extended to support custom datatypes, which requires metatheoretic investigation, e.g., as carried out by Coquand and Paulin-Mohring (1990) for inductive types in CoC, and then implemented in Coq by Paulin-Mohring (1993). In the third approach, datatype support is added on top of already existing mechanisms; this is done by Pfenning and Paulin-Mohring (1990) for the CoC and by Berghofer and Wenzel (1999) for HOL. Church's classic encoding of numbers as functions repeatedly applying an argument function in the λ-calculus may be considered an example of the definitional approach.

Initial support for datatypes in proof assistants only included inductive datatypes, i.e., the minimal solutions to fixpoint equations. Inductive datatypes are arguably the most important, since they facilitate proofs by the fundamental technique of structural induction (Burstall, 1969; Harper, 2016). However, some applications require coinductive datatypes (maximal solutions), which can be accounted for in most type theories (Coquand, 1994). Giménez (1995) initially implemented support for coinductive and corecursive functions in Coq, while Paulson (1997) did the same for Isabelle/HOL.

A long-standing issue in proof assistants is developing mechanisms for *quotient types*, which are defined by dividing members of an existing type into equivalence classes. Quotients are widely used in mathematical reasoning, in particular in algebra. An initial approach to quotients in Isabelle/HOL was proposed by Slotosch (1997), with later alternatives by Paulson (2006) and Huffman and Kunčar (2013). Cohen (2013) proposed

an approach to quotient types in Coq which elides the conventional approach using *setoids* (Geuvers *et al.*, 2002a) that significantly restricts the scope of rewriting tactics.

The dependently-typed language Cedille (Stump, 2017) makes it possible to define induction principles in a language based on Church-encodings (Church, 1941), and encodes datatypes in terms of induction principles. This allows for, among other things, zero-cost reuse of functions and proofs across certain datatypes (Section 6.4).

Research on definitional mechanisms is still an active topic. Sozeau (2010) designed and implemented a Coq extension for defining functions equationally which compiles definitions down to eliminators for inductive types; this extensions was used for the function `acc` in Chapter 2. Biendarra *et al.* (2017) presented a redesigned Isabelle/HOL library, following the definitional approach, for writing and reasoning about inductive and coinductive datatypes. While Coq and Agda inherently allow *nonuniform* datatypes, i.e., recursive types whose arguments vary recursively, HOL systems did not support them until the advent of this library, which reduces such definitions to uniform counterparts.

4.3.5 Totality of Functions and Termination

The logic of Church's simply-typed λ-calculus, HOL, is a logic of total functions. This means that partial functions cannot be directly described in proof assistants based on HOL, such as Isabelle/HOL. Similarly, the Calculus of Inductive Constructions (CIC), which Coq is based on, supports only total functions. Partial functions can still be indirectly encoded in CIC and HOL, for example by (a) returning values in a monad (McBride, 2015), such as the coinductive delay monad described by Capretta (2005), (b) requiring proofs of argument value subset membership as function arguments (Bertot and Casteran, 2004), (c) letting functions return values in the option type, or (d) capturing functions as inductive relations between input and output.

Functions in proof assistants based on intuitionistic type theories like CIC need to be terminating for the sake of consistency. When a function is defined in Coq, for example, termination is automatically proven for cases where functions recurse on a subterm of the input and in

other simple cases; in more advanced cases, users must manually prove termination or rely on approaches that use, e.g., *sizes* of argument terms (Abel *et al.*, 2017b). In contrast, functions in HOL are not required to be computable at the outset, and thus do not need to be accompanied by termination witnesses. On the other hand, the uses of functions with unproven termination are somewhat limited.

Requirements for totality and termination are two hurdles that new users of ITPs face. They are constraints even to users familiar with functional programming languages, where no such requirements are typically imposed. For certain functions, arguing and formally proving termination may not even be a key concern. In that spirit, Zombie (Casinghino *et al.*, 2014) separates out a logical, terminating fragment from a programmatic, possibly non-terminating fragment, that way the programmer can move freely between those fragments.

In other languages, a common technique to encode such functions in a total setting is to define a "fuel" argument, such as a natural number. Either the fuel argument is empty (0) and the function terminates, or there is enough fuel to continue to, e.g., perform recursive calls. This allows for proving termination by a simple structural argument on the fuel type. When calling the function in some other context, passing "infinite fuel" may be possible, which implicitly trusts that the function always terminates. Jourdan *et al.* (2012) provide a detailed description of the fuel technique in the context of a verified parsing function on a potentially infinite stream of tokens in Coq.

4.4 Trusted Computing Bases of Proofs and Programs

The concept of a Trusted Computing Base (TCB) was introduced by Rushby (1981) in the context of security of computer systems. The basic idea is that the security of a system may be reduced to the security of a proper subset of all system components. If these components behave as expected, the system as a whole is secure. For verified software, security is replaced with correctness, e.g., functional correctness. Kumar (2015) divides the TCB into the following categories:

- formal models of system components (e.g., model of a processor);

- system components for which there are no explicit formal models (e.g., linker or operating system);
- tools used to check proofs about the system (e.g., proof assistant).

4.4.1 TCB of Proofs

Pollack (1998) considers the question of how to trust specific machine-checked proofs, and by extension, programs that such proofs pertain to. He divides the question into a purely formal part—whether a provided proof is derivable in a given formal system—and an informal part that asks whether the proof has a purported meaning as expressed outside any formal system.

As to the formal part, trusting the proof can be reduced, by computer, to trusting the (implementation of) the proof checker of the formal system; the source code for such proof checkers can be compact and readable. However, Pollack argues that a complicated semantics of the checker's implementation language can still provide serious obstacle to trust, and proposes that the language itself should be a logical framework designed to represent formal systems, such as LF (Harper *et al.*, 1993) or Isabelle (Paulson, 1994). As to the informal part, Pollack points to that understanding specific pieces of mathematics relies on acceptance of previous mathematics, whose trust may be partly due to its wide acceptance.

Based on Pollack's investigation, Wiedijk (2012) defined the notion of *Pollack-inconsistency*, which is expressed in terms of the mechanisms a proof checker uses to print and parse its formulas (which are what the user ultimately must interpret informally). In particular, Wiedijk argues that a system should always be able to parse formulas it outputs. He then demonstrates that current proof assistants are Pollack-inconsistent to some extent, but outlines how this can be addressed by modifying the implementations of printing and parsing.

With the goal of determining how small a trusted proof checker can be for a practical application, Appel *et al.* (2003) attempted to minimize the size of a proof checker for proof-carrying machine code. The result was less than 2700 lines of code. The Lean theorem prover

attempted to minimize the size of the proof-checking kernel from the start (de Moura *et al.*, 2015).

Barras and Werner (1997) encoded a limited version of the formal system underlying Coq in Coq itself, and proved strong normalization of its type system. Barras (2010) addressed the problem of providing set-theoretical models of CoC, the logic that underpins Coq, with the ultimate goal of ensuring that Coq's theory is consistent with the theory implicitly or explicitly assumed by most mathematicians. Anand and Rahli (2014) encoded and verified the foundations of the Nuprl proof assistant in Coq. Davis and Myreen (2015) certified the Milawa theorem prover. Kunčar and Popescu (2018) proved the relative consistency of extensions made to the foundations of Isabelle/HOL. Anand *et al.* (2018) encoded Coq's internal data structures in Coq itself and gave a semantics for type checking, leading up to the MetaCoq project (Sozeau *et al.*, 2019) for building verified checking and extraction for Coq.

4.4.2 TCB of Programs

Coq and other similar proof assistants contain logic engines that can execute functions that have been verified. However, execution inside such a logic engine is generally slow compared to execution of native functional programs (Leroy, 2015), and does not directly support handling of input and output. Instead, to obtain practical verified programs, proof assistant users rely on mechanisms such as *program extraction* to produce programs that can be integrated into larger systems or executed in conventional runtime environments. However, these mechanisms may increase the trusted base of verified programs.

Program Extraction Paulin-Mohring (1989b) and Paulin-Mohring (1989a) proposed realizability for CoC to F_ω. To obtain practical executable programs from Coq functions, Paulin-Mohring and Werner (1993) extended the realizability from F_ω to ML. Letouzey (2003) and Letouzey (2004) later introduced a new extraction mechanism for Coq which removed several restrictions. This introduced an intermediate language called MiniML, which can be translated to OCaml, Haskell, and Scheme. The new mechanism, argued correct by a conventional proof,

was evaluated on several large projects (Berger *et al.*, 2005; Cruz-Filipe and Letouzey, 2006; Letouzey, 2008).

Berghofer and Nipkow (2002) identified a subset of HOL that can be translated to practical functional languages and implemented code generation for Isabelle/HOL. Haftmann and Nipkow (2010) proposed a redesign of the code generation mechanism in Isabelle/HOL; their approach is based on translating HOL to an intermediate language called Mini-Haskell, and then further to Standard ML, Haskell, and OCaml. The correctness argument is reminiscent of that for Coq's extraction mechanism. Haftmann *et al.* (2013) proposed a data refinement (Section 6.1.2) framework which replaces abstract datatypes with concrete ones, which widens the scope of code generation.

Beyond Extraction and Code Generation Practical functional programming languages such as OCaml and Haskell lack a fully formal (and machine-checked) semantics. Proof assistant users who want to avoid trusting extraction may use *deep embeddings* (Section 6.2.5) of target practical programming languages along with language semantics. These embeddings and semantics can then be used in certified compilers (Section 3.1.1), which may include formal models of system components such as processors, to produce verified machine code. However, these approaches may have more restrictions and inconveniences than extraction. Removing or circumventing these restrictions may be fruitful. Continuing to develop and improve certified compilers for Coq like Œuf and CertiCoq, for example, may help proof engineers circumvent extraction to OCaml and Haskell altogether. Instead, proof engineers may be able to directly compile certified programs to machine or assembly code and run those programs directly.

The version of record is available at: http://dx.doi.org/10.1561/2500000045

5

Between the Engineer and the Kernel: Languages and Automation

At the core of every ITP that meets the de Bruijn criterion is a proof object that a small kernel can check. Directly constructing this proof object may be too low-level to provide for a positive user experience, and in some cases the proof object may not be exposed to the end-user at all. Typically, several layers of languages and automation act as an interface between the proof engineer and the kernel.

Automation is all about finding a *proof*—what the proof object (Section 4.3.1) is can differ by ITP. Because of this, and because of the different philosophies and needs of different communities, ITPs have different approaches to automation. We consider two commonly used ITPs as examples: Coq and Isabelle/HOL.

In Coq, a proof is a term in the language Gallina, which the kernel type-checks. The Coq proof engineer can write proofs in Gallina directly, but it is common to write proofs using high-level *tactics*, or proof search procedures. In Coq, these tactics search for and ultimately produce a Gallina term, which the kernel then type-checks. Users can combine existing tactics, or write their own, either in the general-purpose programming language OCaml or in the tactic language Ltac.

In contrast, in Isabelle/HOL, a proof is a term in the language ML with a specific type, combined with the guarantee that this term must have been produced using the axioms of the logic.[1] It is not possible to directly write a term of this type, since correctness hinges on the guarantees about its construction. Instead, users commonly write proofs in Isabelle/Isar, which is a high-level *proof language*: a language for structuring and composing propositions, facts, and proof goals.

Ltac and Isabelle/Isar are examples of two languages which embody different styles of automation. This chapter discusses these and other styles of automation (Section 5.1), then concludes with a discussion of automation in practice (Section 5.2) built in these different styles.

5.1 Styles of Automation

Proof engineers can construct proofs of theorems in a wide variety of ways. There are three common styles of proof automation: writing sequences of proof search *tactics* (which can be defined either using a *metalanguage* like Standard ML or a specialized *tactic language* like Ltac), writing high-level programs in a structured *proof language*, and using *reflection* to write proof-checking procedures within the host language itself. When executed, all expressions or commands reduce to primitive inference rule applications in the proof-checking kernel.

Ltac and Isabelle/Isar are examples of a tactic language and a proof language, respectively. Some proof assistants (for example, both Coq and Isabelle) have support for all three styles of automation, either natively or through extensions. However, not all do; Agda, for example, takes a minimilistic approach, supporting only reflection (it is possible to imitate tactics using reflection). Table 5.1 references examples of supported styles of automation for a sample of major proof assistants.

These styles often merge, and the lines between them can be blurry. It is possible, for example, to write proofs in the high-level proof language SSReflect in Coq, or to combine this language with Ltac tactics. It is possible to write ML tactics in Isabelle/HOL, and to write tactic-style proofs in Isabelle/HOL that look similar to Ltac proofs in Coq.

[1] It is possible to explicitly produce proof terms that can be checked with a small kernel using Isabelle/HOL-Proofs, but it is not common to explicitly do so.

	Tactics			
	Metalanguage	**Tactic Lang.**	**Proof Lang.**	**Reflection**
Agda				Possible
Coq	OCaml	Ltac	SSReflect	Possible
Isabelle	ML	Eisbach	Isar	Possible
Nuprl	ML			Possible

Table 5.1: Examples of styles of automation a sample of major proof assistants support, including some external developments.

This section explores the design space and uses in common proof assistants of languages for different styles of automation: Tactics and tactic languages (Section 5.1.1), proof languages (Section 5.1.2), and reflection (Section 5.1.3). It then concludes with a discussion of future styles of automation (Section 5.1.4).

5.1.1 Tactics & Tactic Languages

LCF introduced the language ML (*metalanguage*) to let users write high-level proof automation (Gordon *et al.*, 1978). In LCF, theorems are represented using the abstract type `thm` in ML; the only way to construct an inhabitant of `thm` is using the axioms and inference rules of the logic. A proof in LCF has the following type in ML:

```
type proof = thm list → thm
```

In other words, a proof is a function that takes a list of hypotheses and, from them, proves the conclusion.

A basic unit in LCF proof automation is the `tactic`:

```
type tactic = goal → (goal list × proof)
```

The `goal` type (not shown) represents proof goals. Thus, a tactic is a function that takes a proof goal and then produces a list of new goals which the goal reduces to; such goals are conventionally called *subgoals*. When no more goals remain, the tactic produces a value of type `proof`.

Based on this tactic definition, it is possible to define higher-order functions that take tactics as arguments and return new tactics. Milner called such functions *tacticals*. For example, a collection of tactic combinators may include the tactical `repeat` with type `tactic → tactic`, which repeatedly applies its argument tactic to the proof goal. In Coq,

the composition tactical `t1; t2` runs the first tactic `t1`, then runs the second tactic on all goals produced by the first tactic `t2`.

However, the LCF representation of tactics means that, by definition, the outputted goals are mutually independent—a proof for one is unrelated to proofs for others. In practice, constraints during proof search can apply across subgoals (Spiwack, 2016). For this reason, Coq previously used, up to at least version 8.3 in 2010 (Spiwack, 2010), a tactic type definition along the following lines:

```
type proof = thm list → thm
type tactic = goal × state → (goal list × state × proof)
```

Here, the state returned from a tactic call can be used to figure out dependencies between subgoals, such as shared variables.

In the early days of proof assistants, users combined custom tactics written in an ML dialect with built-in tactics and tactical combinators to write custom automation (Constable *et al.*, 1986; Cornes *et al.*, 1995; Paulson, 1988; Paulson, 1983). This tradition for programming proof automation is the default workflow in the HOL4 and HOL Light proof assistants, and remains a possibility in Coq and Isabelle. However, Coq and Isabelle also support writing tactics in tactic languages rather than in the tool's implementation language.

Tactic-Based Proofs

The original proof development workflow in LCF was to write sequences of tactic calls until no proof goals were left. This style is still prevalent in modern proof assistants. Consider an inductive proof of the theorem `app_nil_r` in Coq, which states that appending the empty list to any list produces the original list. We can write this using tactics and tacticals:

```
Theorem app_nil_r : ∀ (A : Type) (l : list A), l ++ [] = l.
Proof.
  intros. induction l; auto. simpl. rewrite IHl. auto.
Qed.
```

Executing these tactics produces a Gallina proof term:

```
(fun (A : Type) (l : list A) ⇒ (* hypotheses *)
  list_ind (* induction principle for lists *)
    (fun (l₀ : list A) ⇒ l₀ ++ [] = l₀) (* motive to prove *)
```

```
    eq_refl (* base case (by reflexivity) *)
    (fun (a : A) (l₀ : list A) (IHl : l₀ ++ [] = l₀) ⇒
      (* inductive case (by rewriting) *)
      eq_ind_r (fun (l₁ : list A) ⇒ a :: l₁ = a :: l₀) eq_refl IHl)
    l) (* argument to induction *)
: ∀ (A : Type) (l : list A), l ++ [] = l. (* theorem type *)
```

There is an analogous in the Isabelle/HOL standard library:

```
lemma append_Nil2 : "append xs [] = xs"
by (induct xs) auto
```

While it is not typical to do so, using Isabelle/HOL-Proofs, it is possible to reconstruct and inspect a proof object for this proof.

Tactic Languages

Tactic languages allow proof engineers to write custom tactics alongside specifications and proofs, rather than in an implementation language such as Standard ML. We describe two tactic languages—Ltac for Coq and Eisbach for Isabelle—in detail, then conclude with a brief discussion of other tactic languages.

Ltac Nearly 20 years ago, Coq introduced the Ltac tactic language (Delahaye, 2000), which has since become the standard for tactic development in Coq. Ltac is an untyped domain-specific language with support for pattern matching on terms and goals, as well as writing custom tactics and tacticals. The Ltac manual can be found in the Coq documentation (Inria, 1999-2018).

To understand Ltac, consider the `break_match` tactic from the StructTact (StructTact Development Team, 2016-2019) library:

```
Ltac break_match := break_match_goal || break_match_hyp.
```

This tactic breaks down match statements, both in goals:

```
Ltac break_match_goal := match goal with
  | [ |- context [ match ?X with _ ⇒ _ end ] ] ⇒
    match type of X with
      | sumbool _ _ ⇒ destruct X
      | _ ⇒ destruct X eqn:?
    end
end.
```

The version of record is available at: http://dx.doi.org/10.1561/2500000045

and in hypotheses:

```
Ltac break_match_hyp := match goal with
  | [ H : context [ match ?X with _ ⇒ _ end ] |- _] ⇒
    match type of X with
      | sumbool _ _ ⇒ destruct X
      | _ ⇒ destruct X eqn:?
    end
end.
```

Both tactics perform syntactic pattern matching over the goal, using the syntax `name : cpattern |- cpattern`, where the left `cpattern` represents hypotheses and the right `cpattern` represents the conclusion. In `break_match_goal`, pattern matching looks only in the conclusion; in `break_match_hyp`, pattern matching looks only in the hypotheses. Both tactics use the `context` syntax to find all subterms of the term that are `match` statements, then pattern match on the type of the result, using the `destruct` tactic to break down those `match` statements.

The effect of `break_match` is to simplify tedious but conceptually simple proofs by case analysis, and to do so without relying on the names of hypotheses, which can make proofs likely to break as specifications change (Woos *et al.*, 2016). Consider, for example, an interpreter correctness proof:

```
Lemma interp_eval : ∀ op v v', interp op v = Some v' → eval op v v'.
Proof.
  unfold interp. intros. destruct op; destruct v;
  try discriminate; inversion H; constructor.
Qed.
```

Here, the `intros` tactic introduces hypotheses named `op`, `v`, `v'`, and `H`. The `destruct op; destruct v` sequence of tactics then does case analysis on `op` and `v`. We can use `repeat break_match` instead of `destruct op; destruct v` to simplify this proof:

```
Proof.
  unfold interp. intros. repeat break_match;
  try discriminate; inversion H; constructor.
Qed.
```

The resulting proof is more concise. It is also less likely to break as specifications change, since it does not depend on the automatically generated names `op` and `v`, which may later change (Section 6.2.3).

Ltac was designed to achieve a balance between the flexibility of writing tactics in a powerful language like OCaml, and the ease of flexibility from using built-in combinators to write custom tactics directly inside of the Coq proof assistant. Like ML and OCaml, it is Turing-complete (see Chapter 16 of Chlipala (2013a)). However, it gives proof engineers limited access to underlying features like environment management. This means that proof engineers do not have to deal with low-level issues in OCaml such as managing de Bruijn indexes; proof engineers who want that level of control may write plugins in OCaml.

Ltac2 (Pédrot, 2019), the next generation of Ltac, is in development. It comes full circle, returning to the ML family of languages.

Eisbach The tactic language Eisbach (Matichuk *et al.*, 2015b) for Isabelle was inspired by Ltac. Eisbach is tactic language that is integrated into the proof language Isabelle/Isar. Using Eisbach, proof engineers write tactics (called *proof methods*) directly in Isar syntax. The Eisbach manual can be found in Matichuk *et al.* (2015c).

To understand Eisbach, consider an example proof method from the Eisbach manual for solving existentials:

```
method solve_ex =
  (match conclusion in ∃ x. Q x for Q ⇒
    <match premises in U : Q y for y ⇒
      <rule exI [where P = Q and x = y, OF U]>>)
```

This matches the current conclusion with the `Q` in `∃ x. Q x`, then looks in the hypotheses for a term that matches `Q y` for the matched `Q` and some `y`, and then calls the introduction rule for existentials with that hypothesis. In other words, if some hypothesis is `Q y` and the goal is `∃ x. Q x`, then `solve_ex` will automatically prove the goal from the hypothesis. The manual shows an example of calling this proof method to solve a proof:

```
lemma halts p ⇒ ∃ x . halts x
  by solve_ex
```

Like Ltac, Eisbach provides limited access to low-level details. Isabelle proof engineers who want access to these details may write proof methods directly in Isabelle/ML.

Other Tactic Languages The untyped nature of languages like Ltac may make it difficult to debug custom automation and to provide strong guarantees on custom tactics. Typed tactic languages such as those of the Delphin (Poswolsky and Schürmann, 2009) and Beluga (Pientka and Dunfield, 2008) logical frameworks, the tactic language for VeriML (Stampoulis and Shao, 2010), and Mtac (Ziliani *et al.*, 2015) and Mtac2 (Kaiser *et al.*, 2018) in Coq address these problems.

PSGraph (Lin *et al.*, 2016) is a graphical tactic language which aims to make debugging and refactoring of tactics easier. In PSGraph, tactics are flow graphs for proof subgoals, and executing tactics amounts to following the flow graph directly.

The Matita (Asperti *et al.*, 2007) proof assistant introduces a language of *tinycals* to address some challenges that tacticals pose for user interaction. In particular, proof assistants traditionally execute tacticals atomically. This makes it difficult to communicate how the tactical is executed to the user, as well as to debug tacticals and provide useful error messaging when they fail. Tinycals, in contrast, act like traditional tacticals, except that they allow for more fine-grained execution.

Some tactic languages merge the tactic approach with metaprogramming constructs, so that users can write tactics in the proof language itself. In Idris (Brady, 2013), it is possible to implement tactics using the elaboration monad, which exposes elaboration to users and enables metaprogramming within Idris (Christiansen and Brady, 2016). Agda recently replaced its reflection mechanism with one based on Idris' elaborator reflection (Norell, 2016).

Similarly, Lean exposes several metaprogramming constructs (including a tactic monad), which enable users to write tactics in Lean itself, to access proof state, and to access internal methods in the underlying C++ codebase (Ebner *et al.*, 2017); this enables proof engineers to write powerful procedures.

Some tools merge the approach of proof by reflection with a tactic language; we discuss these in Section 5.1.3.

5.1.2 Proof Languages

A proof language is a mechanism for structuring and composing propositions, facts, and proof goals. Proof languages generally allow both *backward* reasoning, going from the proof goal to a new set of goals, and *forward* reasoning, where new facts are added to the proof context but the goal remains the same. When formulating reasoning steps in a proof language, procedures in lower-level languages, such as tactics, can be invoked explicitly or implicitly. In turn, these procedures can invoke specialized external proof search programs. In contrast to plain unstructured sequences of commands ("tactic soups"), proofs written in proof languages are usually meant to convey key proof ideas, i.e., to be understandable by humans. To ensure readability, proof languages take inspiration from traditional mathematical vernacular. We consider two proof languages—Coq/SSReflect and Isabelle/Isar—in detail, and then briefly discuss other proof languages.

Coq/SSReflect Coq/SSReflect is a proof language for Coq that emphasizes reasoning by rewriting using equalities and proofs by computation via small-scale reflection (Gonthier and Mahboubi, 2010). It was originally developed by Gonthier in the context of his proof of the four-color theorem (Gonthier, 2008). Idiomatic proofs in Coq/SSReflect make use of "bullets" (`-`, `*`, or `+`) to structure proofs similarly to how Isabelle/Isar uses indentation of blocks to indicate structure.

The basis of Coq/SSReflect is that a step in a proof is one of the following:

- a *deduction step* that directly constructs parts of a proof, either by backwards or forwards reasoning;
- a *bookkeeping step* that performs a management operation on the proof context, e.g., introducing or renaming assumptions;
- a *rewriting step* that changes parts of the proof goal or some assumption, either by way of some equality lemma or by computation.

In idiomatic proofs, these kinds of steps are interleaved and tend to be used in equal proportions.

The *reflect* part of SSReflect refers to the convention of performing deduction steps that translate between symbolic representations, such as expressions involving boolean functions, and logical representations, such as inductive predicates. For example, when a proof goal can be solved by reasoning in propositional logic, we can convert the proof goal to boolean form and perform computation in Coq's logic engine instead of applying multiple propositional derivations manually. In contrast to general proofs by reflection (Section 5.1.3), which focus on computational efficiency and managing the sizes of proof objects, SSReflect leverages reflection for convenience and user productivity. For example, a conjunct can be reflected to a boolean value that directly computes to `true`, saving the manual effort of applying tactics.

The proof language contains special syntax for translating between representations, which is called application of *view lemmas*. Moreover, Coq users traditionally use different commands for rewriting, definition expansion, and partial evaluation. In Coq/SSReflect, all of these tasks are performed via parameters to the `rewrite` tactic. Rewriting operations can pinpoint specific subterms in the current proof goal through the use of *pattern expressions* (Gonthier and Tassi, 2012) that may mention some constant names for disambiguation but leave others implicit.

Consider the following lemma from the Mathematical Components project whose proof is written in idiomatic SSReflect:

```
Lemma edivnP : ∀ m d, edivn_spec m d (edivn m d).
Proof.
rewrite /edivn ⇒ m [|d] //=; rewrite -{1}[m]/(0 * d.+1 + m).
elim: m {-2}m 0 (leqnn m) ⇒ [|n IHn] [|m] q //=; rewrite ltnS ⇒ le_mn.
rewrite subn_if_gt; case: (ltnP m d) ⇒ [// | le_dm].
rewrite -{1}(subnK le_dm) -addSn addnA -mulSnr; apply: IHn.
apply: leq_trans le_mn; exact: leq_subr.
Qed.
```

Here, the first rewrite unfolds the `edivn` definition, while the last rewrite performs chained rewriting using facts arithmetic, such as that addition is associative (`addnA`). Explicit names for quantified variables are given after the operator ⇒, which reduces the chance of brittleness due to

reliance of machine-generated variable names. The tactic `elim` performs induction on the natural number `m`.

Isabelle/Isar Isabelle/Isar is a proof language for Isabelle that aims for human readability while retaining some symbolism of formal deduction systems (Wenzel, 2007). It is built on the Isabelle/Pure logic, which is an intuitionistic fragment of HOL. Isabelle/Isar can be understood as an interpreter for block-structured syntax capturing the flow of facts and proof goals (Wenzel, 2006).

As an example, the Isabelle/Isar manual (Wenzel *et al.*, 2004) contains a definition of a group that assumes only a left identity element, along with the following proof that for any group, the identity element of the group is a right identity (we expand the `term` ... notation from the version in the reference manual for clarity):

```
theorem right_unit : x ∘ 1 = x
proof -
  have 1 = x⁻¹ ∘ x by (rule left_inv [symmetric])
  also have x ∘ (x⁻¹ ∘ x) = (x ∘ x⁻¹) ∘ x by (rule assoc [symmetric])
  also have x ∘ x⁻¹ = 1 by (rule right_inv)
  also have 1 ∘ x = x by (rule left_unit)
  finally show x ∘ 1 = x.
qed
```

Translated directly into English, we can think of this as the following proof (with implicit symmetry of equality):

Theorem 5.1 (Right unit). $x \circ 1 = x$

Proof. By left inverse, $1 = x^{-1} \circ x$. By associativity, $x \circ (x^{-1} \circ x) = (x \circ x^{-1}) \circ x$. By right inverse, $x \circ x^{-1} = 1$. By left unit, $1 \circ x = x$. Then by the above, $x \circ 1 = x$. □

Isabelle/Isar completes and checks this proof much like a human reader would, by making all of the appropriate substitutions. We could render an alternate English proof with all of these substitutions explicit, rather than leaving them to the reader:

Proof. We can write $x \circ 1 = x$ as $x \circ (x^{-1} \circ x) = x$ by left inverse, which is $(x \circ x^{-1}) \circ x = x$ by associativity, which is $1 \circ x = x$ by right inverse, which holds by left unit. Thus, $x \circ 1 = x$. □

At this point, we are much closer to proof by a sequence of commands. The key difference for readability is that the intermediate goals are explicit in the proof language, so to reconstruct an English proof, the reader does not need to step through tactics one-by-one and track the transformation of the goal.

Other Proof Languages Mathematically-inclined proof languages such as Isar for Isabelle and Czar for Coq (Corbineau, 2008) were influenced by the language of the Mizar proof system (Trybulec and Blair, 1985), which tilts more towards natural language than logical symbolism for representing deduction steps. Many proof assistants following the LCF tradition now have Mizar modes; for example, Mizar modes have been implemented in HOL (Harrison, 1996) and HOL light (Wiedijk, 2001), as well as in Coq (Giero *et al.*, 2003).

Building a proof system specifically for human-readability means that there is less of a barrier for humans and computers to check the same proofs. Following in this spirit, the Formalized Mathematics journal consists entirely of mathematical properties and proofs in Mizar that are automatically translated into English and generated as PDFs, such as the properties of sets (Darmochwał, 1990), naturals (Bancerek, 1990), and reals (Kaliszyk and O'Connor, 2009).

While most proof languages were designed with mathematics in mind, their use has not been confined to mathematics. For example, using Isar is recommended style for submission to the Isabelle Archive of Formal Proofs (Klein *et al.*, 2004-2019), which consists of more computer science than mathematics formalizations (Blanchette *et al.*, 2015).

The language PSL (Nagashima and Kumar, 2017) for Isabelle/HOL allows expressing high-level *proof strategies*. PSL generates efficient Isar proof scripts from user-written strategies.

5.1.3 Proofs by Reflection

Writing proofs by reflection, that is, calling certified procedures within the host language itself (Allen *et al.*, 1990), can be viewed as an alternative to writing proofs in a proof language or using tactics.

Chapter 15 of CPDT (Chlipala, 2013a) illustrates this style of proof in Coq and demonstrates its benefits on a proof that a natural number is even; we present a slightly modified version of that example that is self-contained. Given some inductive predicate for evenness:

```
Inductive isEven : nat → Prop :=
| Even_0 : isEven 0
| Even_SS : ∀ n, isEven n → isEven (S (S n)).
```

we construct a verified function to check evenness:

```
Fixpoint check_even (n : nat) : option (isEven n) := match n with
| 0 ⇒ Some Even_0
| 1 ⇒ None
| S (S n') ⇒
    match check_even n' with
    | Some p ⇒ Some (Even_SS n' p)
    | _ ⇒ None
    end
end.
```

For a given `n`, `check_even` returns an optional a proof of `isEven n` (`None` when `n` is not even). As CPDT notes, this type signature guarantees that it only returns a proof when `n` actually is even.

Our goal is to write a tactic that uses `check_even` to prove evenness. To write this tactic, we need to extract the proof from the `option` type above when possible. We define a dependently-typed function `optionOut` that does this:

```
Definition optionOutType (P : Prop) (o : option P) :=
  match o with
  | Some _ ⇒ P
  | _ ⇒ True
  end.

Definition optionOut (P : Prop) (o : option P) : optionOutType P o :=
  match o with
  | Some pf ⇒ pf
  | _ ⇒ I
  end.
```

We then write a tactic that extracts the proof that `check_even` returns:

```
Ltac prove_even_reflective :=
  match goal with
  | [ |- isEven ?N] ⇒ exact (optionOut (isEven N) (check_even N))
  end.
```

With this, the following proof goes through:

```
Theorem even_256 : isEven 256.
Proof.
  prove_even_reflective.
Qed.
```

and similarly for any other even number; the tactic fails (as expected) for odd numbers. As CPDT notes, the size of the resulting proof term is manageable even for large numbers. This is a particular advantage of this style of proof.

The concept of computational reflection predates ITPs; an early history of reflection can be found in Demers and Malenfant (1995), and an early history of its use in theorem proving can be found in Harrison (1995). Accordingly, it is one of the oldest styles of proof automation. Its use in modern proof assistants with support for higher-order logics can be traced back to the 1990s, starting with a proof of the existence of this class of proofs in Nuprl (Allen *et al.*, 1990), and following soon after in other ITPs such as LEGO (Pollack, 1995) and Coq (Boutin, 1997); in Coq, this approach predates Ltac (Delahaye, 2000).

Idris recently replaced its specialized tactic language with a mechanism for reflection called *elaborator reflection* (Christiansen and Brady, 2016). This mechanism exposes Idris' elaborator directly to the programmer, which allows for powerful proof automation. For example, Christiansen and Brady (2016) demonstrates how to use elaborator reflection to write a `mush` tactic, which can be used to dispatch many goals in the Idris standard library:

```
mush : Elab ()
mush =
  do attack
    x <- gensym "x"
    intro x
    try intros
    induction (Var x) `andThen` auto
    solve
```

Proof by reflection is also the dominant style of proof automation in Agda, which does not support tactics. Van Der Walt and Swierstra (2012) demonstrate the `isEven` example from earlier using Agda's old mechanism for reflection. This mechanism was replaced in 2016 with a

reflection mechanism based on Idris' elaborator reflection. Other proof assistants that support proofs by reflection include HOL4 (Fallenstein and Kumar, 2015), Isabelle/HOL (Chaieb and Nipkow, 2008), and Milawa (Davis and Myreen, 2015).

Nowadays, there is a trend of integrating the approach of proof by reflection with tactic languages. For example, Cybele (Claret *et al.*, 2013) is a plugin for writing reflective tactics in Coq, with support for effects and non-termination. Rtac (Malecha and Bengtson, 2016) is a reflective tactic language for Coq, which contains specialized automation to make it simpler to write soundness proofs of decision procedures when writing reflective tactics. These mixed approaches enable proof engineers to take advantage of the benefits of both approaches to more easily build efficient automation.

5.1.4 Future Styles of Automation

One drawback of using tactics is that they can sometimes impede proof understanding. In the future, we expect more tools for proof understanding (in addition to existing structured proof languages). For example, a tool could use tactics to find proofs, then simplify the result, or otherwise output a format that is easier to understand.

Along those lines, debugging tactics and tacticals can be difficult, since the execution of tactics and tacticals often is not conducive to fine-grained debugging, and since fully informative debugging of tactics sometimes requires interfacing with multiple languages (such as Ltac and OCaml). Future tactic languages should better support debugging. The continued development of alternative tactic execution models as well as typed and graphical tactic languages may help with both of these problems.

Another opportunity for improvement with existing automation is improved performance of tactics and tactic languages. We expect more exploration of improving tactic performance, both by writing tactics differently and by improving the performance of the underlying engine.

The continued development of tools that integrate several styles of automation may help proof engineers better take advantage of the benefits of each of these approaches.

5.2 Automation in Practice

Both specialized and general-purpose automation help move the burden of proof away from the proof engineer and toward the tooling with which the proof engineer interacts. This section briefly discusses a non-exhaustive sample of automation procedures (Section 5.2.1). It then concludes with a discussion of the future of automation (Section 5.2.2).

5.2.1 Automation Procedures

Automation can be built using any of the various styles of automation (Section 5.1); since these styles of automation overlap, we consider automation by what it achieves, rather than by the style of automation that it utilizes.

Domain-Specific Automation Domain-specific automation automates proofs within particular domains. For example, the `omega` (Crégut, 1999-2018) tactic in Coq implements a decision procedure for quantifier-free Presburger arithmetic based on the Omega Test (Pugh, 1991), an integer programming algorithm. It can automatically prove mathematical statements that can be difficult for Coq users to prove by hand.

Some domains include verifying programs within specific languages (Cao *et al.*, 2015; Ricketts *et al.*, 2014), writing mathematical proofs (Nipkow, 1990; Slind, 1994; Braibant and Pous, 2011; Narboux, 2004; Grégoire and Mahboubi, 2005; Pouillard, 2012), deciding regular expressions (Braibant and Pous, 2012), and reasoning about embedded logics such as separation logic (Appel, 2006; McCreight, 2009; Krebbers *et al.*, 2017).

General-Purpose Automation General-purpose automation is machinery that is useful across many domains. For example, the `break_match` tactic that we used as an example for the Ltac tactic language (Section 5.1.1) contains useful machinery to make proofs by case analysis simpler and more robust.

Many proof assistants ship with useful general-purpose automation; third-party tools may build on these. For example, many proof assistants come with automation for inversion and induction (Nipkow, 1989; McBride, 1996; Cornes and Terrasse, 1995). Isabelle/Isar has special support for performing induction proofs (Wenzel, 2006); besides specifying the variable to perform induction on, and the induction principle (rule) to use, a user can indicate that certain variables are to be *arbitrary*, i.e., that they are not bound in the resulting assumptions and proof goals. For example, the following clause opens a proof by strong induction for natural numbers on the expression $x - y$, where x and some other variable z from the previous context are arbitrary:

```
proof (induct "x - y" arbitrary: z x rule:less_induct)
```

In Coq, a similar proof requires building a custom induction principle as a separate lemma.

Hint databases (Coq Development Team, 1999-2018c) in Coq store theorems that its other tactics (Coq Development Team, 1999-2018b) such as `auto` and `rewrite` can use as hints. For example, the tactic `auto with arith` tells `auto` to use the arithmetic theorems defined in the `arith` database when it tries to solve the goal. Hints can help make proofs not only simpler, but more robust (see Chapter 3.8 of CPDT), though they may negatively impact proof search performance or even cause it not to terminate (see Chapter 13 of CPDT).

Some third-party libraries such the StructTact library (StructTact Development Team, 2016-2019) and the code distributed with CPDT and FRAP (Chlipala, 2017) ship a variety of general-purpose automation that builds on the standard library packaged in one place. Automation from these libraries ranges from machinery to better handle induction such as `prep_induction` (StructTact Development Team, 2016-2019) and `induct` (Chlipala, 2017) to powerful tactics like `crush` (Chlipala, 2013a), which can dispatch many proof obligations automatically. The `agda-prelude` (Norell, 2015-2019) library provides efficient alternatives to automation in the Agda standard library.

Theory exploration—the automatic discovery and sometimes proof of theorems for a given theory—is a form of general-purpose automation

that first arose in the context of automated theorem proving for mathematics (Buchberger, 2000). This style of automation aims to mimic the way that mathematicians explore theories when writing proofs by hand. While theory exploration tooling began with the development of specialized tooling, specialized tools can be used in tandem with an ITP; Dramnesc *et al.* (2015), for example, uses the theory exploration tool *Theorema* (Buchberger *et al.*, 2006) in combination with Coq to explore the theory of binary trees. The tool Hipster (Johansson *et al.*, 2014; Valbuena and Johansson, 2015; Johansson, 2017) for Isabelle/HOL integrates theory exploration directly with an ITP.

Other examples of useful general-purpose automation include simple general-purpose proof automation (Coq Development Team, 1999-2018b; Zhan, 2016; Lindblad and Benke, 2006), rewriting (Coq Development Team, 1999-2018b; Nipkow, 1989), and solving logical fragments (Paulson, 1999; Lescuyer and Conchon, 2009; Hurd, 2003; Kumar *et al.*, 1991; Busch, 1994; Dahn *et al.*, 1997; Hurd, 1999), and techniques for reasoning about executable specifications (Barthe and Courtieu, 2002), as well as an implementation of a generalization of congruence closure to dependent type theory (Selsam and de Moura, 2017). In addition, Chapter 6 describes general-purpose automation and tooling for proof reuse (Section 6.4.3), as well as general-purpose automation built on type classes and canonical structures (Section 6.2.1).

Hammers Hammers are systems for general reasoning over large libraries of formal proofs (Blanchette *et al.*, 2016b). Like the verification language F* (Swamy *et al.*, 2016) or the congruence closure algorithm (Selsam and de Moura, 2017) in Lean, hammers leverage automated theorem provers (ATPs) from within an ITP. Hammers leverage ATPs while preserving the small trusted bases of ITPs. They are able to learn from previous proof efforts. In proof assistants, a hammer is exposed as a collection of tactics that in effect comprise a brute-force method for discharging a proof goal.

A hammer for a proof assistant typically has three components:

1. a *premise selector* that selects facts (axioms) to be used by ATPs from the large library available to the proof assistant;

2. a *translator* that converts the selected facts and proof goal to the restricted logics of the ATPs;

3. a *proof reconstructor* that builds proofs accepted by the proof assistants from the evidence provided by the ATPs.

The premise selector arguably has the most challenging task, since the database of facts can be large, and it is difficult to determine whether a fact is relevant to the given proof goal. The standard approach is to leverage machine learning techniques, such as naive Bayes and k-nearest neighbors (Blanchette *et al.*, 2016b; Czajka and Kaliszyk, 2018).

The translator must take into account the particular foundations and features of the proof assistant, such as polymorphic or dependent types, and provide faithful representation in target logics, which may have no types or only monomorphic types.

The proof reconstructor, like the translator, is highly specific to the proof assistant. Reconstructors can use many different approaches of varying robustness, such as ATP proof replay, reflection, or proof assistant source generation, augmented by various heuristics. As a result, reconstruction may sometimes fail.

Implementations of hammers include Sledgehammer for Isabelle/HOL (Paulson and Blanchette, 2012), HOL(y)Hammer for HOL Light and HOL4 (Kaliszyk and Urban, 2014), and CoqHammer for Coq (Czajka and Kaliszyk, 2018; Czajka *et al.*, 2018). Invocations of hammers typically spawn many parallel instances of different ATPs, such as Z3, Vampire, the E theorem prover, and CVC4. Hammer services to proof assistants can also be provided remotely, overcoming local limitations on processing power and memory (Kaliszyk and Urban, 2015).

As an example of applying a hammer, consider a Coq lemma about lists from the StructTact library:

```
Lemma app_cons_singleton_inv : ∀ A xs (y : A) zs w,
 xs ++ y :: zs = [w] → xs = [] ∧ y = w ∧ zs = [].
```

Invoking the CoqHammer `hammer` tactic finds a proof via Z3:

```
Extracting features...
Running provers (using 8 threads)...
Z3 (nbayes-32) succeeded
- dependencies: List.app_eq_unit
```

The output also gives the following tactic call to replace the `hammer` invocation, yielding a proof of the lemma without ATPs:

```
Proof. Reconstr.rcrush List.app_eq_unit Reconstr.Empty. Qed.
```

While CoqHammer generates sequences of calls to custom tactics for reconstruction, Isabelle's Sledgehammer typically results in calls to the built-in superposition prover `metis` (Blanchette *et al.*, 2011), which works similarly to ATPs such as Vampire.

The usual way to evaluate the effectiveness of a hammer for a particular proof assistant is to apply the hammer on a standard library by replacing proof scripts with invocations of hammer tactics. For example, CoqHammer was able to reprove 44.5% of all results in the Coq standard library (Czajka and Kaliszyk, 2018), which is in line with success rates for HOL Light benchmarks (40%). Success rates for benchmarks in Isabelle/HOL can be as high as 70%, for databases with upwards of 100,000 facts (Blanchette *et al.*, 2016b). However, these rates do not reflect practical application of hammers in evolving projects, where proof goals may be reformulated based on manual exploration using certain proof strategies.

In contrast to property-based testing (Paraskevopoulou *et al.*, 2015) and counterexample generators (Blanchette and Nipkow, 2010), hammers do not give feedback when ATPs are unable to discharge a proof goal. Consequently, applying hammers does not necessarily lead to progress. On the other hand, hammers do not require decidable properties, generation of datatype instances, or domain knowledge.

Augmentation of hammer components to increase effectiveness and success rates is an active research topic (Blanchette *et al.*, 2016a; Wang *et al.*, 2017; Peng and Ma, 2017).

5.2.2 Future of Automation in Practice

Hammers have been around in Isabelle for a long time, but until recently, it was not known if a hammer could be implemented for a dependent type theory. We expect more development to follow in the lines of CoqHammer.

The existence of third-party libraries for general-purpose proof automation in many ways mirrors the rise of third-party libraries for

other programming languages which supplement the standard library; we expect more of these libraries to come into existence, and we expect existing libraries to grow in popularity. We also expect domain-specific tactics for common domains to continue to develop, and to cover new domains as they arise.

The version of record is available at: http://dx.doi.org/10.1561/2500000045

6

Proof Organization and Scalability

As verification projects have grown in size, proof engineers have increasingly stressed strategies and tools for organizing large proof developments and for dealing with the challenges of scale. This section describes some of these strategies and tools: constructs for property specification and encodings (Section 6.1), proof design principles (Section 6.2), high-level frameworks (Section 6.3), and constructs and tools for proof reuse (Section 6.4).

6.1 Property Specification and Encodings

Proof engineers leverage many constructs and notations to express programs and their specifications. For example, Coq offers a single basic language called Gallina for both logical formulas and (computable) functions, while Isabelle offers both an object language (e.g., HOL) and a metalogic with different operators and quantifiers. The specification languages can be extended inside proof assistants by using *notations*, which provides new syntax for existing concepts; this is crucial for emulating mathematical vernacular, which can aid understanding of formal definitions.

On top of basic specification constructs, sophisticated properties can be expressed using inductive predicates via familiar definitional mechanisms (Section 4.3.4). These predicates can be interpreted as higher-order Prolog programs (Bertot and Casteran, 2004). For generality and reuse, collections of such specifications can be abstracted over using mechanisms such as parametric polymorphism (Strachey, 2000), modules, and type classes (Section 6.2).

6.1.1 Domain-Specific Specification Languages

Sewell *et al.* (2010) presented a domain-specific language called Ott for expressing inductive definitions and inductive properties over such definitions, suited in particular for formalizing programming language semantics. Ott files can be exported to Isabelle/HOL, Coq, and HOL4, using specific annotations for each proof assistant. For example, the regular expression datatype from Chapter 2 can be expressed in Ott as

```
regexp :: regexp_ ::= {{ com regexp }} {{ coq-universe Type }}
 | 0 :: :: zero | 1 :: :: unit | c :: :: char
 | r + r' :: :: plus | r r' :: :: times | r * :: :: star
```

while the last matching rule for the Kleene star becomes

```
s in L ( r ) s' in L ( r * )
---------------------------- :: star_2
s s' in L ( r * )
```

Note that the extra spacing is necessary for Ott's parser to properly disambiguate the syntax.

The more general proof assistant-agnostic specification language Lem (Mulligan *et al.*, 2014) also includes definition of recursive functions and other programming language constructs, as well as a standard library useful for semantic definitions. Ott files can be exported to Lem format and thus incorporated into larger definitions.

6.1.2 Refinement of Programs, Data, and Proofs

Stepwise *program refinement* is the construction of a program by a sequence of refinement steps, where each refinement step breaks the original problem into a subproblem (Wirth, 1971); these steps can be verified in an ITP. Each refinement can be a refinement of a program

without changing the datatypes, or a refinement of the datatypes themselves (*data refinement* (De Roever *et al.*, 1998)). Via the principle of propositions-as-types, similar approaches can be used to develop proofs by stepwise *proof refinement* of an existing specification.

Program Refinement A proof of refinement formally relates an abstract program to a concrete, refined version of that program. It establishes that all of the behaviors of the concrete program are contained in the set of behaviors of the abstract program (De Roever *et al.*, 1998). This relation can also be stated and proven in terms of the program specifications (as in the *refinement calculus* (Back, 1988)) or in terms of a simulation relation (Section 6.2.4). Chlipala (2017) contains an overview of using program refinement to derive verified correct programs from their specifications.

Back (1991) formalized the refinement calculus in HOL. von Wright (1994) presented a tool for verified program refinement using the refinement calculus in HOL. Since then, there have been a number of refinement tools in Isabelle/HOL with support for logic (Hemer *et al.*, 2001), object-oriented (Liu *et al.*, 2011), functional (Lammich, 2013), and imperative (Lammich, 2015) programs. Cohen *et al.* (2013) developed a framework for Coq called CoqEAL which automates key steps of data refinement. Delaware *et al.* (2015) presented Fiat, a refinement framework for deductive synthesis of abstract data types in Coq.

Proof engineers use proofs of program refinement to break down large proof developments or to compose modular proof developments, for example for the verification of storage systems (Chajed *et al.*, 2019), compilers (Leroy, 2009; Rizkallah *et al.*, 2016; Kumar *et al.*, 2014), and OS kernels (Klein *et al.*, 2014; Gu *et al.*, 2015; Gu *et al.*, 2016). Refinement proofs can also help make proof developments robust to changes (Section 6.2.3).

Proof Refinement Reasoning backwards in proof assistants, from goals to premises, can be viewed as a form of proof refinement (Bates, 1979; Krafft, 1981), where the proof is the refinement of the specification. The idea of proof refinement is to refine the goal to proofs of subgoals,

then refine those subgoals further. Bates (1979), for example, describes the rule for refining a conjunction `A ∧ B` given hypotheses `S`:

```
S pr A ∧ B by
  S pr A
  S pr B
```

In other words, `A ∧ B` follows from `S` if each of `A` and `B` follow from `S`. Each of `A` and `B` follow from `S` if they can be refined using other rules.

Refinement logics such as Nuprl (Constable *et al.*, 1986) and RedPRL (Angiuli *et al.*, 2018) as well as other proof assistants following in the LCF tradition (Section 4.2) encourage this style of reasoning. Sterling and Harper (2017) contains an overview of proof refinement.

6.2 Proof Design Principles

Good design principles can make proofs easier to develop and maintain. These design principles mirror software engineering design principles in many ways, but also address challenges unique to proof engineering.

Consider an example in Coq from Woos *et al.* (2016), which demonstrates a design principle that addresses challenges unique to proof engineering. In this example, we have a proof `eg_proof` of a theorem `eg`, which shows that if two functions map equal inputs to equal outputs, then any proposition that holds on all outputs of `g` must also hold on all outputs of `f`:

```
Definition eg : Prop :=
  ∀ (A B : Type) (f g : A → B) (P : B → Prop),
    (∀ x, P (g x)) →
    (∀ x, f x = g x) →
    (∀ x, P (f x)).

Lemma eg_proof : eg.
Proof.
  unfold eg. intros. rewrite H0. auto.
Qed.
```

Suppose we later change `eg` (using orange to show changes):

```
Definition eg : Prop :=
  ∀ (A B : Type) (f g : A → B) (P Q : B → Prop),
    (∀ x, P (g x)) →
    (forall x, P (g x) -> Q (g x)) →
```

```
  (∀ x, f x = g x) →
  (∀ x, P (f x) ∧ Q (f x)).
```

As the authors note, our proof `eg_proof` no longer holds, since the automatically generated hypothesis name `H0` is now called `H1`. One way to address this is to change the hypothesis name in the proof as well:

```
Proof.
  unfold eg. intros. rewrite H1. auto.
Qed.
```

But if we continue changing `eg`, then we will need to keep making these kinds of changes. Instead, the authors advocate for using the tactic `find_rewrite`, since it does not depend on hypothesis names:

```
Proof.
  unfold eg. intros. find_rewrite. auto.
Qed.
```

This proof goes through for both definitions of `eg`.

The design principle from this example addresses a challenge unique to proof engineering, since it deals with the consequences of proof automation. Other proof engineering design principles mirror software engineering design principles. This section provides an overview of design principles for proof engineering, drawing parallels to software engineering when appropriate. It focuses on general-purpose design principles, and discusses domain-specific design principles (beyond those from Chapter 3) only when relevant more broadly.

6.2.1 Design Principles for Abstraction

As in software engineering, design principles for proof engineers prevent changes in implementation from breaking dependencies that ought to rely only on specifications. For example, much like a software engineer may write an interface for a collection of functions so that he can switch out implementation details such as the underlying data structure without breaking functionality that depends on those functions, so a proof engineer may write an interface for a collection of lemmas so that changes to the proofs of those lemmas do not break other lemmas and theorems that depend on those lemmas (Woos *et al.*, 2016).

There are many ways to achieve this sort of abstraction in ITPs, some of which have different implications for proof automation. For example, in Coq, it is possible to write interfaces using modules, type classes, or canonical structures; Coq has special support for proof search for type classes (Sozeau and Oury, 2008) and canonical structures (Saibi, 1999). This section describes some means of abstraction in an ITP.

Modules Modules, as manifested in languages such as Standard ML (MacQueen, 1986) and proof assistants such as Coq (Chrząszcz, 2003), are collections of named components which may be types, values, or nested modules. A central property is the separation of module interfaces (signatures or module types) and module implementations (structures). The interface-implementation relation is many-to-many; one signature can be implemented by several structures, and one structure can implement several signatures. A structure can choose to hide all information not specified in the signatures it implements.

Parametric modules, called functors, take structures that implement certain signatures as arguments. In proof assistants, functors can provide abstraction and reuse of both functions and proofs. This approach is taken to implement finite sets and maps in Coq using AVL trees (Filliâtre and Letouzey, 2004), and later to implement balanced binary search trees using red-black trees (Appel, 2011a).

Type Classes Type classes were first implemented in the Haskell programming language (Wadler and Blott, 1989). In a proof assistant context, type classes have notably been implemented for Coq (Sozeau and Oury, 2008) and Isabelle/HOL (Haftmann and Wenzel, 2007); instance arguments (Devriese and Piessens, 2011) are a similar feature in Agda. Type classes can be viewed as a particular use of a module system as in Standard ML, and type classes can coexist with such a module system (Dreyer *et al.*, 2007).

A type class can be viewed as an abstract data type that defines a collection of functions by their parameter types, while not fixing function implementations. The abstract data type can then be implemented in different ways for different parameter types. For example, Volume 4 of

Software Foundations (Pierce *et al.*, 2014) describes an equality type class with a single function `eqb` which, when provided two arguments of the same type, returns a boolean:

```
Class Eq A :=
{
  eqb: A → A → bool;
}.
```

Different implementations (*instances*) of this class can then be provided for different types; Software Foundations describes one for booleans:

```
Instance eqBool : Eq bool :=
{
  eqb := fun (b c : bool) ⇒
    match b, c with
      | true, true ⇒ true
      | true, false ⇒ false
      | false, true ⇒ false
      | false, false ⇒ true
    end
}.
```

and one for natural numbers:

```
Instance eqNat : Eq nat :=
{
  eqb := Nat.eqb
}.
```

A compiler can translate programs that use functions defined for type classes to programs that do not by looking up and applying the appropriate function instances, using information about function invocation types.

A key use of type classes in Haskell programs is as a way to structure programs by abstracting certain code over appropriate type classes, and concretizing the abstracted code with appropriate type instances elsewhere, avoiding duplication and facilitating reuse; proof engineers can use type classes similarly to achieve both code and proof reuse. However, type classes in ITPs also provide additional benefits beyond those that type classes in other languages such as Haskell provide. For example, one drawback of using only type classes for structuring programs in Haskell is that a type can implement a type class in exactly one way (Harper, 2011); it may be useful to define different type class

instances for sorting of integers depending on the size of the input data, or use different orders on integers for sorting. Unlike in Haskell, type classes in Coq can support multiple instances.

The Coq implementation of type classes is *first-class*, meaning that it is a thin layer on top of existing functionality (specifically, implicit arguments and dependent records). In addition, Sozeau and Oury (2008) added specific support for type class resolution into Coq's proof search mechanism. In contrast to Coq's type classes, the type classes of Isabelle/HOL are restricted to one type variable, and are not first-class. One particular advantage of type classes in proof assistants (as opposed to type classes in Haskell) is that propositions can be type class members, e.g., a type class for a monad can require witnesses (proofs) for the monad laws along with monad operations. By extension, this means that proofs in one type class instance can be derived partly from proofs in other type class instances (e.g., of some more general class). In contrast with Haskell, the type class instance resolution system in Coq can always be elided by manually passing implicit type class instances.

Type classes have been used for abstraction and reuse in many proof developments. For example, Spitters and Weegen (2011) used type classes to represent a standard algebraic hierarchy in Coq, along with parts of category theory. Woos *et al.* (2016) used type classes to organize the correctness proof of the Raft consensus protocol in Coq, and for abstracting the Raft protocol implementation for replication over arbitrary state machines.

Type classes are closely tied to other language features for abstraction. General parametrization of theories in Isabelle can be achieved via *locales* (Kammüller *et al.*, 1999; Ballarin, 2006), which is the mechanism used to provide type class support (Haftmann and Wenzel, 2009); a locale can be viewed as a persistent proof context that includes arbitrary variables and assumptions, and which can be instantiated in other proofs.

Canonical Structures In Coq, canonical structures (Mahboubi and Tassi, 2013; Saibi, 1999) provide an alternative to type classes. Canonical structures are a mechanism to provide theory-specific dictionaries to

datatypes, allowing for more flexible resolution strategy than more the more widely used type classes.

To show their typical use, consider partial commutative monoids (PCMs); an algebraic structure which recurs in our current ongoing work on the verification of stateful and concurrent programs (Nanevski *et al.*, 2014). We implement PCMs using two of the Coq's native constructs: *dependent records* and *canonical structures*. We follow the established SSReflect design pattern of defining algebraic data structures by means of *mix-in* composition (Garillot, 2011), whereby different dependent records formalize different algebraic properties, which can be combined using *packed classes* mechanism. The latter also defines the field resolution strategy (Garillot *et al.*, 2009) in a case of overlapping names. For instance, in Coq the *mix-in* defining PCMs is represented by the following dependent record:

```
Record mixin_of (T : Type) := Mixin {
  valid : T → bool;
  join : T → T → T;
  unit : T;
  _ : commutative join;
  _ : associative join;
  _ : left_id unit join;
  _ : ∀ x y, valid (join x y) → valid x;
  _ : valid unit }.
```

The type `T` is the *carrier type* of the structure. The field `valid` selects a subset of `T`, standing for the "defined" elements. The invalid (or "undefined") elements help model partiality: a partial function over `T` will return some invalid element on an input on which it is mathematically undefined. `join` is the binary operation of the PCM, and `unit` is the unit element. The remaining five unnamed fields enumerate the axioms that have to be satisfied by each PCM instance.

Next, the mix-in "interface" is packaged with a carrier type, into a dependent record type, which represents PCMs. We also introduce a coercion from the package to the underlying carrier type, so that the two can be conflated. This coercion essentially accounts for the delegation hierarchy from object-oriented languages.

```
Structure pcm : Type := Pack {type : Type; _ : mixin_of type}.
Coercion type : pcm >→ Sortclass.
```

Next, we explain the mechanism of packaging all necessary definitions along with lemmas about data structures (such as *join*'s commutativity and associativity in the case of PCMs) into a single module that should be imported by the clients of the algebraic structure. For example, we introduce appropriate notation for the join operation, and specifically name and prove the lemmas that correspond to the PCM properties that we left unnamed in the mixin.

```
Notation x • y := (join x y).
Lemma joinC (U : pcm) (x y : U) : x • y = y • x.
Lemma joinA (U : pcm) (x y z : U) : x • y • z = x • (y • z).
```

The lemmas such as `joinC` and `joinA` are proved by destructing the package `U`, but notice how the coercion allows conflating `U` with its carrier type. Also notice how the notation • allows the PCM `U` to be omitted from the equations themselves, as the typechecker can infer it from the context.

Algebraic structures can inherit the properties of other, more basic structures. Thus, we also require an analogue of object-oriented *inheritance*. We illustrate how this can be done in Coq, by defining an interface for a *cancellative* PCM, which inherits from an ordinary PCM. The cancellative PCM is defined as the following mix-in record:

```
Record mixin_of (U : pcm) := Mixin {
  _ : ∀ a b c: U, valid (a • b) → a • b = a • c → b = c
}.
```

Notice that the dependent record `mixin_of` in this case is parametrized via the carrier PCM `U`, which is used as a target for a coercion whenever an instance of a plain PCM or a carrier type `U` is required, since coercions a transitive.

Let us now *instantiate* the definition of abstract structure with concrete datatypes. It turns out that it is insufficient to merely prove that a datatype satisfies the PCM axioms. To work comfortably with an algebraic structure in practice, one has to explicitly "register" the structure with the type inference engine.

We first show what goes wrong if one doesn't perform the "registration." For instance, assume we first define an instance of a PCM for `nat` with addition, by proving that `+` with `0` satisfies the PCM axioms. Then the following lemma which uses the generic notation • for the

PCM operation, is considered ill-formed by Coq. The reason is that Coq cannot figure that there is a PCM associated with `nat`, and that the generic notation • should be resolved with addition. Indeed, we could have defined the PCM for `nat` via multiplication (×) with 1, in which case • should be resolved by ×.

```
Lemma add_perm (a b c : nat) : a • (b • c) = c • (b • a).
```

In the above case, once a structure is registered as the default PCM for `nat`, the `add_perm` lemma can be proved by selective rewriting using the standard PCM properties.

Some notable uses of canonical structures include telescopes (Garillot *et al.*, 2009) and higher-order tactics for separation logic (Gonthier *et al.*, 2011).

6.2.2 Design Principles for Programming with Dependent Types

In order to use dependent types to their full extent, proof engineers have developed many paradigms to deal with the challenges they present. For example, using dependent types, we can define heterogenous lists and a selection function over heterogenous lists. To write the selection function, however, we must first define the *type* that it has, which depends on the case. Chlipala (2013a) accomplishes this using a membership predicate:

```
Section hlist.
  Variable A : Type.
  Variable B : A → Type.

  Inductive hlist {A : Type} {B : A → Type} : list A → Type :=
  | HNil : hlist nil
  | HCons : ∀ (x : A) (ls : list A), B x → hlist ls → hlist (x :: ls).

  Variable elm : A.

  Inductive member : list A → Type :=
  | HFirst : ∀ ls, member (elm :: ls)
  | HNext : ∀ x ls, member ls → member (x :: ls).

  Fixpoint hget ls (mls : hlist ls) : member ls → B elm :=
    (* ... *)
End hlist.
```

In other words, the type of `hget` states that whenever `elm` is a member of some list `ls`, then we can select some element of type `B elm` from any `mls : hlist ls`. This is one example of a common style of writing functions and proofs using dependent types, wherein the proof engineer first defines the type of the function or proof inductively, and then defines the function or proof that has that type. It is a powerful style that makes it possible to define very expressive types.

Chlipala (2013a) provides a comprehensive overview of dependently-typed programming in Coq with many more examples. Tanter and Tabareau (2015) outlines design principles for gradual verification in Coq, which may help reduce the burden of verification with dependent types and increase adoption.

6.2.3 Design Principles for Scale

The scale of programs verified in ITPs has increased over the years. In recent years, proof engineers have begun to look at how to address the challenges that come with this increase in scale. Proof engineering in the large (Kaivola and Kohatsu, 2003), for example, describes a methodology for verifying large-scale circuits; it is among the earliest work noting that proof design for large verification projects is important. This section describes design principles dealing with the challenges of scale such as robustness in the face of changes, compositionality of components, and efficiency of code and proofs. In addition, Section 6.4 describes design principles for proof reuse.

Design Principles for Robustness A major source of inefficiency in verification is *proof brittleness*: Even a minor change to a single theorem or definition can break many dependent proofs. This makes proofs difficult to maintain (Woos *et al.*, 2016; Aydemir *et al.*, 2008; Murphy-Hill and Grossman, 2014; Delaware *et al.*, 2013a). Design principles help make proofs robust in the face of changes. This is one approach to proof evolution (Section 7.2).

One approach to building robust proofs is to make use of proof automation (Chapter 5) to dispatch similar goals. Proof engineers who use the default tactics included in many proof assistants already take

advantage of this, since the same tactic can discharge different goals. For example, a proof engineer who uses `omega` in Coq or `presburger` in Isabelle/HOL need not change the proof script as the goal changes, so long as the goal stays within the same fragment of arithmetic solved by those tactics.

The degree to which proof engineers rely on automation varies by style. CPDT (Chlipala, 2013a), for example, advocates for the heavy use of program-specific automation, noting that this makes proofs more robust; the tagless interpreter proofs from Chapter 8.3 of CPDT contain an example of automation of this kind. This style of development localizes the burden of change to the automation itself as opposed to the many proofs that use the automation.

While automation can make proofs more robust, it can also be brittle in itself. For example, some Coq tactics automatically generate hypothesis names; small changes in specifications can cause proofs that rely on those names to break. One approach to this problem is to always explicitly specify hypothesis names, so that Coq never generates hypothesis names automatically; the IDE Company-Coq (Pit-Claudel and Courtieu, 2016) provides some built-in support for this approach. Planning for Change (Woos *et al.*, 2016) notes that, while this approach helps, it is still necessary to update those explicit names as specifications change. Instead, the authors advocate for the use of *structural tactics*, or tactics that do not depend on hypothesis names and hypothesis ordering; many tactics of this style can be found in the associated StructTact (StructTact Development Team, 2016-2019) library.

Planning for Change addresses design for robustness not only at the level of automation, but also at the level of specifications and proof objects. It presents a methodology for writing robust proofs independently of any domain or framework. This methodology is informed by a large proof engineering effort verifying the Raft consensus protocol. It is a set of five recommendations. Some of these recommendations draw on software engineering design principles. For example, the authors recommend using information hiding techniques similar to those used in software engineering to hide definitions. That way, the burden of change is localized to interface changes, and changes in only implementation do not cause breaking changes in dependencies. Other recommendations

tackle challenges that are unique to proof engineering. For example, the authors advocate for the use of custom induction principles to capture common patterns in inductive proofs.

Refinement (Section 6.1.2) can help make proofs robust to changes. For example, the proofs of the seL4 microkernel in Isabelle/HOL have evolved alongside the implementation for over eight years (Klein *et al.*, 2014). The proof development makes use of two layers of specifications: an *abstract specification* which describes only behavior of the system, and an *executable specification* which includes implementation details. These two layers are connected by a refinement proof. Using this approach, the authors found that both making low-level changes and adding new simple features were not very costly, though more complex changes that interacted with other parts of the code significantly were still costly.

Design Principles for Compositionality CompCert (Section 3.1.1) employs a compositional design for describing the different intermediate languages and how they interact with each other. Affinity lemmas from Planning for Change also capture this concept. The CertiKOS project introduces the idea of a *deep specification* (Gu *et al.*, 2015) that makes compositional verification more tractable. DeepSpec (DeepSpec Team, 2013-2019), an ongoing project, is addressing this problem more generally. Section 6.3 describes frameworks for compositional verification.

Design Principles for Efficiency Some proof assistants like Coq work by extraction (Section 4.4.2) from the core language into an executable language. The resulting extracted code can be slow, which can be a barrier for verifying a realistic system. Cruz-Filipe and Spitters (2003) and Cruz-Filipe and Letouzey (2006) describe proof design principles for optimizing the efficiency of extracted code.

6.2.4 Style Guides and Proof Techniques

Style guides and proof techniques help guide proof engineers in dealing with common patterns to address common challenges.

Style Guides Section 3.2.1 described style guides in mathematics. A few general-purpose style guides exist. Gerwin's style guide (Klein, 2015) for Isabelle, for example, is a set of guidelines that are used within Isabelle itself and in several large developments. The Isabelle Archive of Formal Proofs requires that submitted proofs follow some of these guidelines, and recommends others (Klein *et al.*, 2004-2019). The CoqStyle (Coq Development Team, 2017) style guide is a set of guidelines for Coq in the main Coq repository which is used within the standard library.

Proof Techniques Proof techniques are techniques that handle common classes of proofs, or that make it easier to write proofs in a particular style. For example, one widely-used proof technique is *simulation* (Lynch and Vaandrager, 1994); an overview of this technique can be found in Chlipala (2017). This technique helps proof engineers prove that systems preserve liveness and safety properties. Refinement (Section 6.1.2) reduces to simulation (Klein *et al.*, 2014).

One application of simulation is to show compiler correctness. CompCert (Leroy, 2009), for example, uses this technique to show that the program transformations that the compiler makes are semantics-preserving. In the case of compiler correctness as in CompCert, both directions of simulation (*forward simulation* and *backward simulation*) start with a source program s and a target program t that are related along some relation r. The forward simulation (Figure 6.1, left) states that if s steps to s', then t can step to some t', where s' and t' are related by r. Similarly, the backward simulation (Figure 6.1, right) states that if t steps to t', then s can step some s', where t' and s' are related by r. Intuitively, a forward simulation shows that "anything the source program could do, the target program could do too," and a backward simulation shows that "anything the target program could do, the source program could do too." Together, a forward and a backward simulation establish *indistinguishability*, any entity restricted to only observe "visible" program transitions (e.g., input and output) will never be able to determine if they are interacting with the source or target program (Sangiorgi, 2011). If the source language is deterministic, then the forward simulation follows from the backward simulation; if the

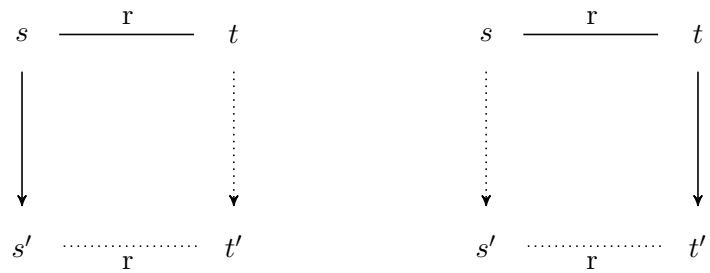

Figure 6.1: Forward (left) and backward (right) simulation for compiler correctness. Premises are show as solid lines and goals are shown as dashed lines.

target language is deterministic, then the backward simulation follows from the forward simulation (Leroy, 2009). CompCert takes advantage of this to show backward simulation from only forward simulation and determinism of the target language.

Section 5.1 discusses some techniques for interacting with automation. Proof techniques can also help proof engineers reason within certain domains. Bahr and Hutton (2015), for example, describes a technique for deriving correct compilers from specifications in Coq. Section 6.2.5 describes techniques for reasoning about imperative programs.

6.2.5 Design Principles for Reasoning about Imperative Programs

When designing a verification framework for imperative programs based on a dependently-typed proof assistant (*e.g.*, Coq), the most common approach is to implement a version of a Floyd-Hoare style program logic (Floyd, 1967; Hoare, 1969) in it. When doing so, the framework designer is faced with the following choices:

- How to embed, into a proof assistant, the language with the features, which the *host* language does not have (*e.g.*, mutable state and concurrency)?
- How to encode verification conditions for imperative programs specified in Floyd-Hoare style, and implement the corresponding reasoning principles in a proof assistant?

Below, we elaborate on these two design choices and provide a survey of the most prominent approaches implementing them, both for sequential and concurrent reasoning about imperative programs.

On Shallow and Deep Embedding

An important design decision to take when designing a framework for verification of effectful (i.e., heap-manipulating or concurrent) programs on top of a general-purpose proof assistant is its use of shallow or deep embedding the language to be verified.

Shallow embedding is an approach of implementing programming languages, characterized by representation of the language of interest (usually called a *domain-specific language* or DSL) as a subset of another general-purpose host language, so the programs in the former one are simply the programs in the latter one. The idea of shallow embedding originates at early '60s with the beginning of era of the Lisp programming language (Graham, 1996), which, thanks to its macro-expansion system, serves as a powerful platform to implement DSLs by means of shallow embedding (such DSLs are sometimes called internal or embedded).

Shallow embedding in the world of practical programming is advocated for a high speed of language prototyping and the ability to reuse most of the host language infrastructure. An alternative approach of implementing and encoding programming languages is called *deep embedding*, and amounts to the implementation of a DSL from scratch, essentially, writing its parser, interpreter and type-checker in a general-purpose language. Deep embedding is preferable when the overall performance of the implemented language runtime is of more interest than the speed of DSL implementation, since then a lot of intermediate abstractions, which are artifacts of the host language, can be avoided.

In the world of mechanized program verification, both deep and shallow embeddings have their own strengths and weaknesses. Although implementations of deeply embedded languages and calculi naturally tend to be more verbose, design choices in them are usually simpler to explain and motivate. Moreover, the deep embedding approach makes the problem of name binding to be explicit, so it would be appreciated

as an important aspect in the design and reasoning about programming languages (Aydemir *et al.*, 2008; Weirich *et al.*, 2011; Charguéraud, 2011). We believe that these are the reasons why this approach is typically chosen as a preferable one when teaching program specification and verification in Coq (Pierce *et al.*, 2014).

Importantly, deep embedding gives the programming language implementor full control over its syntax and semantics. In particular, the expressivity limits of a defined logic or a type system are not limited by expressivity of the host language's type system. Deep embedding makes it much more straightforward to reason about pairs of programs by means of defining the relations as propositions on pairs of syntactic trees, which are implemented as elements of corresponding datatypes. This point becomes crucial when one needs to reason about the correctness of program transformations and optimizing compilers (Appel *et al.*, 2014).

In contrast, the choice of shallow embedding, while sparing one the labor of implementing the parser, name binder and type checker, may limit the expressivity of the logical calculus or a type system to be defied. In the case of Hoare Type Theory (Nanevski *et al.*, 2010), for instance, it amounts to the impossibility to specify programs that store effectful functions and their specifications into a heap.[1]

In the past decade Coq has been used in a large number of projects targeting formalization of logics and type systems of various programming languages and proving their soundness, with most of them preferring the deep embedding approach to the shallow one. We believe that the explanation of this phenomenon is the fact that it is much more straightforward to define semantics of a deeply-embedded "featherweight" calculus (Igarashi *et al.*, 2001) and prove soundness of its type system or program logic, given that it is the ultimate goal of the research project. However, in order to use the implemented framework to specify and verify realistic programs, a significant implementation effort is required to extend the deep implementation beyond the "core language," which makes shallow embedding more preferable.

[1] This limitation can be, however, overcome by postulating necessary axioms.

Encoding Verification Conditions

In a Floyd-Hoare style logic, specification of a program c is given in a form of a triple $\{P\}\ c\ \{Q\}$, where the assertions P and Q are referred to as the *precondition* and the *postcondition*, respectively. The standard semantics of the triple imposes that for any state, satisfying P, the final state, after c terminates, satisfies Q. This definition corresponds to termination-insensitive *partial correctness* (*i.e.*, c is allowing to not terminate at all, so any postcondition would hold). Some program logics impose a stronger semantics of *total correctness*, requiring c to terminate, in addition to the above (Dockins and Hobor, 2010).

This treatment of a Floyd-Hoare triple allows for verifying the programs by means of following the inference rules of a program logic, allowing to decompose the proof of $\{P\}\ c\ \{Q\}$ into the proofs about c's sub-programs (Hoare, 1969). While this style of reasoning seems natural and relatively easy to implement in a proof assistant, and is advocated by the most widely used tutorials (Pierce *et al.*, 2014), it is not the most convenient to conduct the proofs in, due to the need of constantly discharge the *weakening* obligations, required for "massaging" the verification goal, and represented by the following inference rule:

$$\frac{P \Rightarrow P' \qquad \{P'\}\ c\ \{Q'\} \qquad Q' \Rightarrow Q}{\{P\}\ c\ \{Q\}}\ (\textsc{Weaken})$$

A more proof-assistant-friendly way to encode the Floyd-Hoare-style verification conditions, "compressing" the necessary applications of the weakening rule, is to use the idea of predicate transformer by (Dijkstra, 1975) that can be used to compute a pre-condition for a computation, for any context in which that computation may be used. This approach, dubbed the *weakest precondition (WP) calculus* allows one to encode the meaning of a Floyd-Hoare triple (roughly) as follows:

$$\{P\}\ c\ \{Q\} \triangleq P \Rightarrow \mathsf{wp}\ c\ \{Q\},$$

where $\mathsf{wp}\ c\ \{Q\}$ is the program c's *weakest* precondition *wrt.* the imposed postcondition Q, expressed as a logical formula. Therefore, what is left to the designer of the mechanised program logic to do is to provide the implementation of the primitive wp, which would "compile" a

program and its postcondition to the logical assertion, which can be later discharged using the host proof assistant's machinery.

The weakest precondition approach to encoding verification conditions for imperative programs is amazingly versatile, and, to the best our knowledge, has been adopted in most of the major implementations of program logics embedded into proof assistants (Nanevski *et al.*, 2008a; McCreight, 2009; Nanevski *et al.*, 2010; Charguéraud, 2010; Chlipala, 2011; Swamy *et al.*, 2013; Appel *et al.*, 2014; Krebbers *et al.*, 2017).

Verifying sequential heap-manipulating programs

The main success in a program logic-based verification of heap-manipulating programs has been achieved with the discovery of Separation Logic (O'Hearn *et al.*, 2001; Reynolds, 2002). It did not take long for Separation Logic to be mechanised in an ITP (Nanevski *et al.*, 2008a; Mehta and Nipkow, 2003; McCreight, 2009), using both deep and shallow embedding.

One of the most successful formalizations in Coq by means of shallow embedding is the series of work on Hoare Type Theory (HTT) by Nanevski *et al.* (2008a), known as YNot. YNot has been used, among other things, in verifying a relational database system (Malecha *et al.*, 2010) and a secure browser kernel (Jang *et al.*, 2012).

In addition to adopting the WP-calculus for expressing verification conditions in an embedding of Separation Logic into Coq, HTT first made active use of *binary postconditions*, enabling a straightforward treatment of *logical* variables, whose scope spans both pre- and postconditions of a Floyd-Hoare triple. Specifically, this has been achieved by making a postcondition Q to be not of type $\mathsf{state} \to \mathsf{Prop}$ (*i.e.*, unary, as suggested by the textbook expositions of program logics), but rather of type $\mathsf{state} \to \mathsf{state} \to \mathsf{Prop}$, *i.e.*, constraining both the pre- and the post-state. This style of specification has later been adopted by multiple other verification frameworks (Swierstra, 2009; Swamy *et al.*, 2013).

The Sepref (Lammich, 2015) tool for verifying imperative programs in Isabelle/HOL includes a separation logic framework built on top of Imperative HOL, which is built on top of Isabelle/HOL. Sepref uses

refinement (Section 6.1.2) to derive an imperative heap-based program and correctness proof from a functional program and correctness proof.

YNot (Nanevski *et al.*, 2008a) has implemented the *heap disjointness*, inherent to Separation Logic, by means of a deep embedding of logic reasoning principles. Such an embedding of a domain-specific logic required a later development of a number of tactics for making large mechanised proofs tractable (Chlipala *et al.*, 2009). In a later work, Nanevski *et al.* (2010) have shown how to achieve almost the same expressivity with very little domain-specific automation, by making reasoning about finite heaps decidable and leveraging the machinery of small-scale reflection (Gonthier and Mahboubi, 2010).

Various successful deep embeddings of Floyd-Hoare style reasoning into Coq have been demonstrated viable for the sake of reasoning about low-level programs using different versions of Separation Logic (Chlipala *et al.*, 2009; Chlipala, 2011; Chlipala, 2013b; Chen *et al.*, 2015; Cao *et al.*, 2018). All those efforts came supplied with tailored libraries of domain-specific tactics, with those tactics automatically applying Separation Logic's FRAME rule and thus progressively reducing the size of the verification goal.

6.2.6 Future of Design

While we think that domain-specific design principles will always be important, we expect that there is a lot of potential for general-purpose design principles that frame proof engineering in the context of software engineering and make novel use of what we already know. Planning for Change investigates where proof engineering diverges from software engineering and where it calls for specialized techniques; continuing along these lines should drive more useful proof design techniques.

Compared to design principles for mathematics, current general-purpose design principles place little emphasis on proof understanding. While this is an understandable difference in emphasis, his can inhibit collaboration for proof engineers as well. We expect more work on proof understanding to become common as collaboration between proof engineers increases with the growth in large-scale verification projects.

Automation-heavy styles can help prevent breaking changes, but have drawbacks. Some of these drawbacks may be avoidable. For example, one limitation is that proof checking of the large and complex terms these procedures produce can be slow. Developments in proof checking such as term simplification could make this style more tractable. Debugging is also difficult; alleviating this concern could be as simple as better debugging tooling for tactics.

6.3 High-Level Verification Frameworks

In the context of software engineering, a *framework* is distinguished from a library or domain-specific language in that the client relinquishes control of execution to the framework. In practice, the concept of a framework often refers to some combination of design principles, libraries, and tooling that together give structure to code, often within a certain domain, regardless of control of execution. We use the latter term, as it is what is used most often in proof engineering papers, and as the concept of control of execution does not always make sense in the context of proof development.

Several of the libraries and languages we have already discussed (for example, Bedrock (Chlipala, 2013b)) fit this definition of a framework. This section extends that discussion to cover frameworks for two common domains: concurrent applications (Section 6.3.1) and language design and metatheory (Section 6.3.2). It then discusses frameworks for a few other domains (Section 6.3.3), and concludes with a discussion of the future of frameworks (Section 6.3.4) for proof engineering.

6.3.1 Frameworks for Verifying Concurrent Applications

Reasoning about concurrent programs brings new challenges into mechanising reasoning: due to the excessively large state-space of possible interactions between simultaneously executing processes or threads, simply enumerating them is no longer tractable. However, since in most of the practical applications the interaction between processes on some sort of shared state happens only at dedicate program points, via specific programming primitives, a plausible way to reduce this complexity

is to reduce *concurrent* reasoning to a *sequential* one. This idea has been pioneered in the work on *Concurrent Separation Logic* by O'Hearn (2007), which provided a series of inference rules for compositional sequential and concurrent reasoning for shared-memory concurrency.

Similarly to plain Separation Logic, variants of CSL have been implemented as both shallow and deep embedding with the corresponding benefits and drawbacks.

The first *shallow embedding* of Subjective Concurrent Separation Logic, a CSL-like logic for concurrency, was due to Ley-Wild and Nanevski (2013), who implemented it using Coq's indexed types. Unlike the prior work on Hoare Type Theory (Nanevski *et al.*, 2010), in which Coq's dependent types were only capturing the effect of an imperative program on a state, in SCSL, the types were also carrying information about *resource invariants*, capturing the contract of a concurrent interaction between threads. That work has been later extended to a more expressive *Fine-Grained Concurrent Separation Logic* (FCSL) (Nanevski *et al.*, 2014; Sergey *et al.*, 2015), which provided a more general treatment of concurrent resources, incorporating ideas from both CSL and Rely-Guarantee-based verification methodologies (Jones, 1983; Feng, 2009), and implementing them in a form of a shallowly-embedded type theory for state.

The main shortcoming of both SCSL and FCSL, both being shallowly-embedded type theories for state, are the limitations due to the limitations of Coq's model *wrt.* impredicativity. At the time of this writing, FCSL did not support higher-order heaps (*i.e.*, the possibility to reason about arbitrary storable effectful procedures). It was conjectured by FCSL's authors that this obstacle could be overcome by relying on the universe polymorphism feature introduced in Coq version 8.5 (Sozeau and Tabareau, 2014). An approach based on *Rely-Guarantee references*, similar to FCSL in spirit, employed Coq as a host framework for implementing DSL (but not proving its soundness *wrt.* some semantics) for streamlining reasoning about certain concurrency patterns (Gordon *et al.*, 2017), allowed by considering Rely-Guarantee contracts, but without CSL-enabled proof modularity.

Implementation of concurrent imperative programs in Coq by means of *deep embedding* has been first considered in the context of verifying

low-level code with dynamic thread creations in CAP and CCAP program logics (Yu and Shao, 2004; Feng and Shao, 2005). Targeting real architectures, those formal verification efforts required astonishingly high proof efforts and have been eventually superseded by a mechanized proof methodology based on *certified abstraction layers*, not grounded in any specific Hoare-style program logic (Gu *et al.*, 2015; Kim *et al.*, 2017; Gu *et al.*, 2018).

Iris is another CSL-inspired mechanised verification framework that has been in development in parallel with FCSL, with an aim to provide more uniform foundations for reasoning about concurrency (Jung *et al.*, 2015). Due to the chosen semantic foundations, allowing for impredicativity in the presence of mutable state (and hence, storable higher-order procedures) (Svendsen and Birkedal, 2014), Iris could not have been implemented as a shallow embedding and, hence, has been encoded as a deeply-embedded logic.

While that initially has been considered an significant obstacle for verifying large concurrent programs in Iris, due to a large proof overhead, the later introduction of Iris Proof Mode (IPM) (Krebbers *et al.*, 2017) fixed this shortcoming, significantly lowering the entrance threshold for conducting mechanised Iris proofs (Birkedal and Bizjak, 2018). This has been achieved in IPM by effectively leveraging Coq's extensible parsing and proof-by-reflection, and introducing a library of domain-specific tactics, mimicking, for the sake of an end user of the framework, standard CSL-style inference rules. Due to its success, IPM itself has been later generalised to MoSeL—an extensible proof mode allowing for reasoning not just with Iris but with any separation-style program logics (Krebbers *et al.*, 2018).

As frameworks implementing Hoare-style reasoning about concurrency with pre/postconditions, both FCSL and IPM follow the encoding style with Dijkstra-style weakest preconditions.

6.3.2 Frameworks for Language Design and Metatheory

Several frameworks deal with the challenge of component reuse, one of the challenges from POPLMark (Section 3.2.2). Meta-Theory à la Carte (MTC) (Delaware *et al.*, 2013b) is a framework and Coq

library that builds on Data Types à la Carte (Swierstra, 2008) to address challenges in reuse: *extensibility* of definitions and proofs through algebraic properties that provide control over the evaluation order, and *modular reasoning* about partial definitions and proofs through algebraic combinators. Using MTC, the proof engineer can assemble a language from existing components.

MTC does not address extensibility of languages with effects: adding new effects breaks existing proofs. Modular Monadic Meta-Theory (3MT) (Delaware *et al.*, 2013a) extends MTC with a methodology and monad library that includes monads for effects as well as algebraic laws. The methodology and library make proofs resilient to the addition of new effects to a language. MTC and 3MT both use algebraic properties to address difficulties with component reuse—algebra in many ways offers natural abstractions, and those techniques can apply more broadly outside of formal metatheory.

Other notable frameworks for language design include the Fiat (Chlipala *et al.*, 2017; Delaware *et al.*, 2015) framework for Coq, as well as the Hybrid (Felty and Momigliano, 2012) framework for Isabelle/HOL and Coq, which addresses the difficulties of using HOAS with inductive and coinductive proofs.

6.3.3 Frameworks for Other Domains

Concurrent applications and language metatheory are just two domains for which verification frameworks are useful. Frameworks assist proof engineers in many other domains. For example, a few frameworks exist for verifying distributed systems. Verdi (Wilcox *et al.*, 2015) is a framework for building verified distributed systems in Coq; it has been used to build and verify an implementation of the Raft consensus protocol (Woos *et al.*, 2016). Disel (Sergey *et al.*, 2017) is a framework for compositional verification of distributed protocols.

6.3.4 Future Frameworks

The expressiveness of the underlying logics of common proof assistants combined with their interactive natures makes it possible to develop useful frameworks for a variety of domains. We expect that proof

engineers will continue to develop and improve on frameworks that tackle challenges associated with common domains, as well as build new frameworks to handle challenges associated with new domains for verification as they arise. In addition, we expect that frameworks will address challenges that current frameworks do not fully address, such as language extension and component reuse in metatheory.

While it is natural to apply frameworks to challenges within common domains, we also expect the development of more general-purpose frameworks building on common proof assistants to address challenges that proof engineers face independently of domain, or when following specific design principles.

6.4 Proof Reuse

Large proof developments may involve redundant efforts that can be time-consuming. Proof reuse addresses this by repurposing existing proofs as much as possible, minimizing the amount of redundant work that proof engineers must do.' Early examples of proof reuse include *proof by analogy* (Curien, 1995), the technique of adapting a proof of a theorem to a proof of a related theorem, and *proof generalization* (Hasker and Reddy, 1992), the technique of adapting a proof of a theorem to prove a more general theorem.

Proof reuse is the proof engineering analogue to software reuse. Like software reuse, proof reuse leverages design principles (Section 6.4.1) and language constructs (Section 6.4.2). In addition, the interactive nature of proof assistants naturally leads to a class of proof reuse technologies less explored in the software engineering world: automated tooling (Section 6.4.3). This section samples these approaches.

6.4.1 Design Principles for Modularity and Reuse

Good design principles can help maximize the reusability of existing proofs. Some of these design principles are natural generalizations of design principles for software reuse more generally, such as *aspect-oriented software development* (AOSD), a programming approach that optimizes for separation of concern (Filman *et al.*, 2004). Others, like

the affinity lemmas from Planning for Change (Section 6.2.3) are unique to proof engineering.

Design Principles from Software Engineering In software engineering, encapsulating behavior can help not only protect against future changes, but can also help with reusing multiple implementations of interfaces with the same behavior. Likewise, the interfaces and information hiding recommendations form Planning for Change are useful not only to protect proofs against future changes, but also to switch between different datastructure implementations with the same high-level behavior.

The work by Delaware *et al.* (2011) attacks the problem of language metatheory extension, along with the corresponding formalization in a proof assistant and changing the corresponding type safety proofs (i.e., progress and preservation theorems), from the perspective of Software Product Lines (SPL). SPL is an approach to AOSD that opportunistically reuses software by deriving many different pieces of software from a common producer. Delaware *et al.* (2011) starts from formalizing a core language, taking a "core" Featherweight Java (cFJ), and considering all further extensions to the language (casts, interfaces, generics) as features.

What is inherent for the SPL approach is reasoning about composition and possible interaction between features, expressed by means of an algebra of feature operators: $\cdot$, $\#$, and $\times$. Introducing multiple features can lead to an exponential explosion of pairwise interaction, which, however, is rarely observed in practice, as most of the features are mutually independent.

In order to enable feature-based decomposition of a language, all its components (syntax, dynamic semantics, safety proofs etc.) are written is specific languages, amenable for feature compositions. For instance, the language syntax and its semantic/typing rules can be extended by introducing the mechanism of *variation points* (VPs) into the corresponding grammar productions, premises and conclusions, of the rules, reusing the intuition of SPL design.

On the implementation side, the modularity of extensions is achieved by means of reusing Coq's capabilities for higher-order parametrization:

language component definitions are parameterized by the corresponding variation point contexts. The crux of the technique is identifying the effect of the VPs to the safety proofs, which are conducted in a way, parametric with respect to the inductive cases to be considered. For each specific combination of the features, the top-level proof dispatches to the proofs from the corresponding feature module.

The shortcoming of the approach is the requirement, for a core language, to have a significant foresight when identifying the appropriate VPs, which provide the opportunity for feature extensions. While the paper demonstrates how to do it in the context of a language, whose safety is formalised via the syntactic approach, it provides little guidance with respect to other ways of stating type soundness (e.g., via logical relations), neither does it consider other domains beyond PL design. Overall, the approach seems to be a bit ad-hoc, which is why further advances in this direction lead to the creation of the monadic MTC (Delaware *et al.*, 2013b) and 3MT (Delaware *et al.*, 2013a) frameworks we have already discussed.

Beyond Software Engineering Proof assistants in the LCF family are complex systems with multiple languages at different levels. Accordingly, reuse in these systems happens not only at the term level, but also at the tactic level. Designing powerful tactics can maximize reuse of proof scripts to prove different goals (Section 6.2.3).

Among the recommendations that Planning for Change makes is the use of *affinity lemmas* that describe relationships between components. These lemmas show that properties that hold over one component also hold over another, which facilitates reuse of proofs across components.

6.4.2 Language Constructs for Organization and Reuse

As in software engineering, proof assistants often provide support for reuse at the language level. These range from entire languages optimized for reuse to useful constructs built on existing languages that make reuse easier. This section describes a sample of languages and language constructs for proof reuse.

Languages for Reuse Some languages are designed with the goal of optimizing for proof reuse. For example, Felty and Howe (1994) describes an ITP that is optimized for reuse at the tactic level.

In this system, reuse works by replaying tactics in a new proof setting. To make this possible, the system automatically generalizes proofs using metavariables. In contrast, the logical framework *PR* (Caplan and Harandi, 1995) optimizes for reuse at the level of the type theory. The framework builds on an embedded Hoare logic, adding constructs to the logic that aid in abstraction and reuse of proof terms.

HoTT (introduced in Section 4.3.2) has practical proof engineering applications. HoTT's univalence axiom gives rise to automatic *transport* of functions and proofs across type equivalences: to write the same function or proof about two equivalent types, the proof engineer needs only to write the function or proof over one of these two types, and then show the equivalence between them. Cubical type theory (Cohen *et al.*, 2018) provides a computational interpretation of HoTT's univalence, so that it is no longer an axiom.

Proof assistants or extensions to proof assistants built on HoTT or cubical type theory include Cubical Agda (Agda Development Team, 2005-2018), CoqHoTT (CoqHoTT Development Team, 2015-2019), and RedPRL (RedPRL Development Team, 2015-2018). While these ITPs are relatively new, we expect that reuse will be easier in these proof assistants. However, univalence is incompatible with the popular axiom UIP (Uniqueness of Identity Proofs, which states that all proofs of equality at a given type are equal), and univalent ITPs present their own difficulties, so these are not a catch-all solution. We discuss tooling for transport that does not rely on univalence at the level of the type theory in Section 6.4.3.

Language Constructs for Reuse Even in languages that are not designed with the goal of proof reuse in mind, certain language features can help make proof reuse more tractable. The modules, type classes, and canonical structures discussed in Section 6.2.1 are examples of these features, as are other mechanisms for inheritance. In addition, many proof assistants implement subtyping or type coercions (Barthe, 1995; Aspinall and Compagnoni, 2001; Saibi, 1997; Luo, 1999; Asperti *et al.*,

2007; Callaghan and Luo, 2001; de Moura *et al.*, 2015) in various forms, and these can also help make proof reuse more tractable.

One recent development is the notion of an *ornament* (McBride, 2011), a programming mechanism for describing relationships between inductive types that preserve inductive structure. That is, there is an ornament between natural numbers and lists, and between lists and length-indexed vectors; there is no ornament between lists and trees, since these types have different inductive structures. Ornaments allow for the derivation of new types from existing types, and for the automatic lifting of functions and proofs from each existing type to the corresponding new type. Lifting functions and proofs necessitates some additional automation beyond the addition of ornaments to a language. So far, ornaments exist in various forms as deep embeddings in Agda (Dagand, 2017; Williams *et al.*, 2014; Ko and Gibbons, 2016), and as tooling for proof reuse (Section 6.4.3) in Coq.

The language Cedille makes it possible to define combinators that allow for reuse of functions and proofs across certain related datatypes without any performance penalty (Diehl *et al.*, 2018). Like ornaments, these combinators facilitate reuse between unindexed and indexed versions of types like lists and vectors. They do not support incompletely determined relations that ornaments support, such as the ornament between natural numbers and lists (lists have a new element in the inductive case). Applications of these combinators definitionally reduce in such a way as to facilitate efficient reuse thanks to properties of the underlying type theory of Cedille.

6.4.3 Automated Tooling for Proof Reuse

Since proof assistants typically involve a heavily interactive workflow like the REPL, they lend themselves naturally to automation. As such, in addition to the design principles and language features for proof reuse found in typical software engineering projects, there is a body of work that uses automated tooling to repurpose existing proofs. Section 6.3.2 describes some frameworks for component reuse, a kind of proof reuse, in mechanized metatheory. This section describes other tooling for proof reuse.

Adapting Inductive Proofs Boite (2004) describes a tactic to adapt proof obligations to changes in inductive types. This technique constructs and analyzes a dependency graph to determine when reuse of existing proofs is possible, then reuses existing proofs when possible and generates new proof obligations for new branches of the proof. Mulhern (2006) provides a high-level description of a possible method to synthesize missing proofs for those new obligations using a type reconstruction algorithm, though it is not currently implemented.

Proof Planning *Proof planning* (Bundy, 1988) is a proof search technique that uses plans to guide search for proofs with similar structures. Proof planning can involve the use of *critics* (Ireland, 1996), which reuse information from failing proofs to guide search for correct proofs. While it was originally designed for use with automated theorem provers, it has also reached interactive theorem provers. For example, IsaPlanner (Dixon and Fleuriot, 2003) is a proof planner for Isabelle with support for rippling (Shah, 2005), a technique for automatic induction. Rippling has also been implemented in an induction automation tool for Coq (Wilson *et al.*, 2010).

Proof Generalization Proof generalization tools generalize proofs of a theorem to obtain proofs of more general theorems. Proof generalization first arose in the 1990s (Hasker and Reddy, 1992; Kolbe and Walther, 1998; Pons, 1999). A simple example of proof generalization is the Coq `generalize` tactic, which does basic syntactic generalization. The Coq documentation demonstrates this tactic on the following proof state (Coq Development Team, 1999-2018b):

```
x, y : nat
============================
0 <= x + y + y
```

Running `generalize (x + y + y)` on this goal produces the following proof state:

```
x, y : nat
============================
∀ n : nat, 0 <= n
```

The final generated proof term proves the original goal by specialization of this generalized goal.

The generalization technique implemented in Coq's `generalize` tactic can handle only simple syntactic substitution. A few tools can handle more complex transformations. For example, Johnsen and Lüth (2004) presents a proof generalization tool for Isabelle with proof terms which can handle generalizing over dependencies on other theorems, as well as generalization over functions and types. The Coq proof repair tool PUMPKIN PATCH (Section 7.2.3) includes an abstraction component which does not just syntactic generalization, but also type-driven generalization.

Transport *Transport* (also known as *transfer*) methods automatically adapt proofs along relations. These tools aim to mimic the experience of mathematical proofs on paper, in which simply stating a relation between two structures (such as an equivalence) can be enough use theorems about one structure as theorems about the other.

This idea developed both as an extension to the language and as an approach to automation: Barthe and Pons (2001) introduced an extension to dependent type theory with a computational interpretation of isomorphisms using rewrites. Around the same time, Magaud and Bertot (2002) introduced an automatic method for adapting proofs along binary and unary representations of the natural numbers.

Since then, there have been many more transport tools handling more than just those two types, including the *Transfer* and *Lifting* packages (Huffman and Kunčar, 2013) for Isabelle/HOL, and a prototype Coq plugin for transporting proofs across isomorphisms and implications (Zimmermann and Herbelin, 2015).

Univalent transport is the particular kind of transport across type equivalences that arises from HoTT's univalence axiom (see Sections 4.3.2 and 6.4.2). Equivalences for Free! (Tabareau *et al.*, 2018) uses insights from HoTT to develop and formalize a powerful tool for transporting proofs across equivalences in Coq. In many cases, it is possible to use this tool to port functions and proofs without any axiomatic dependencies; in some cases, the tool relies on the functional extensionality axiom. Thus far, the primary barriers to usability of the tool are the proof

burden on the user to configure the automation, and the inefficiency of the generated functions. Nonetheless, this is a significant step toward a robust tool for automatic transport in a proof assistant that does not depend on univalence.

The DEVOID (Ringer *et al.*, 2019) Coq plugin automates transport across certain equivalences that correspond to algebraic ornaments, a particular class of ornaments (Section 6.4.2). DEVOID automatically discovers and proves the equivalences that correspond to these ornaments, and then transports functions and proofs across those equivalences using a program transformation. DEVOID handles a narrow class of equivalences relative to Equivalences for Free!, but the functions and proofs that it produces for the cases it can handle are small and efficient in comparison.

6.4.4 Packaging and Distributing Programs and Proofs

Proof assistants projects are software artifacts, and can thus be packaged and distributed in a similar way. For Isabelle/HOL, the venue for distribution is the Archive of Formal Proofs (Klein *et al.*, 2004-2019; Blanchette *et al.*, 2015). Coq uses the OCaml infrastructure around the OPAM package manager to provide a similar collection of packages (Coq development team, 2018). In principle, executable verified software can also be distributed on these platforms, but can also use conventional channels, which may raise issues of trust.

6.4.5 Future of Reuse

Component reuse, a form of proof reuse, is an underaddressed tenant POPLMark. The solutions which do exist are able to take advantage of common proof structure within the domain of metatheory. We expect that similar common structure exists for proofs in other domains, and this area is ripe for the development of design principles, frameworks, and automated tooling to maximize reuse.

More generally, we expect to see mainstream proof assistants continue to integrate language reuse constructs. For example, ornaments are a promising feature designed specifically for reuse in a dependently typed language, but most existing implementations require the user

to write programs and proofs in a domain-specific deeply embedded logic. DEVOID takes some steps toward integrating ornaments into an existing ITP without an embedding, but it handles only a small class of ornaments and makes some additional restrictions beyond those that the original ornaments work assumes. We expect ornaments to integrate more smoothly with existing ITPs in the future.

We expect that recent developments in HoTT will fundamentally change how people view proof reuse, and that concepts from HoTT will continue to influence the design of proof reuse tools for other languages. Approaches like Equivalences for Free! have the benefit of principled design of automation with guaranteed properties, but do not introduce univalence, and so are not incompatible with other assumptions that programmers may want in their type theories. These two views of univalent transport can continue to evolve alongside one another.

The version of record is available at: http://dx.doi.org/10.1561/2500000045

7

Practical Proof Development and Evolution

As the scale of proof development grows, priorities change. Project management becomes more important, as does dealing gracefully with changes over time. These demands mirror the concerns the software community has addressed as program development has scaled. The proof engineering community has responded similarly, with interfaces, environments, processes, and tools for effective development that scales.

This section discusses developments in user interfaces and tooling (Section 7.1), proof evolution (Section 7.2), user productivity and cost estimation (Section 7.3), and mining proof repositories (Section 7.4) that address these concerns.

7.1 User Interfaces and Tooling for User Support

Most early proof assistants shipped with a very simple user interface: the Read-Eval-Print Loop (REPL). This interface reads in user-written expressions in the proof assistant language, evaluates those expressions, then prints a result or error for the user.

User interfaces for proof assistants have come a long way from the REPL. The support that these REPLs provide users is minimal, and so soon after their development, many techniques to ease interaction with

REPLs arose. While the interfaces from earlier eras still see common use, we are now entering an era of interaction that emphasizes full integrated development environments (IDEs), with support for project management and for asynchronous development.

In parallel to this evolution of user interfaces (Section 7.1.1), we are seeing an increase in specialized interfaces (Section 7.1.2), usability analysis of user interfaces (Section 7.1.3), and advanced tooling for user support (Section 7.1.4). We expect these traditions will merge and drive the future of interaction (Section 7.1.5) with proof assistants.

7.1.1 The Evolution of User Interfaces

We can think of proof assistant user interfaces as evolving in three generations:

1. Generation I: The REPL
2. Generation II: Separation of Concerns
3. Generation III: Full IDEs

Generation I: The REPL The REPL was the earliest form of interaction with the proof assistant. For example, the description of Stanford LCF (Milner, 1972) calls the proof process a "conversation between the user and the computer." The LCF user writes commands, which the computer evaluates and replies to with feedback such as new goals. In part of the example from the LCF description, the user cuts an inline lemma:

```
*****GOAL f ⊂ g;
```

The computer then responds acknowledging the new goal:

```
NEWGOAL #1 f ⊂ g
```

The user tells the computer to prove this goal inductively:

```
*****TRY 1 INDUCT 1;
```

The computer responds with two intermediate goals, a base case and an inductive case:

```
NEWGOAL #1#1 UU ⊂ g
NEWGOAL #1#2 fun(f1) ⊂ g ASSUME f1⊂g
```

The user then proves those subgoals, then uses the inline lemma to prove the original result.

Many ITPs followed in this tradition and introduced command line REPLs. Examples of command line REPLs include the `coqtop` (Coq Development Team, 1999-2018c) command for Coq and the `hol` (HOL Development Team, 2016-2018) command for HOL. Some of these tools are still accessible even when graphical interfaces exist. For example, Coq still exposes its `coqtop` command, in spite of the existence of the graphical interfaces CoqIDE (Coq Development Team, 1999-2018a) and Proof General (Aspinall, 2000b).

However, not all ITPs followed in this tradition. Nuprl, for example, was distributed with a graphical interface from the start (Constable *et al.*, 1986). Even those ITPs that followed in this tradition sometimes later diverged. For example, Agda's `--interactive` option to interact with the REPL directly is no longer supported; Agda interaction happens through an Emacs (GNU Project, 1985-2019) or Atom (GitHub, 2014-2019) mode which calls out directly to the backend theorem prover (Agda Development Team, 2005-2017). Figure 7.1 shows an examples of the Agda Emacs mode. Isabelle/HOL has recently done away with its REPL; the default interface is now Isabelle/jEdit (Wenzel, 2012), which instead builds on Isabelle/PIDE (Wenzel, 2014).

Generation II: Separation of Concerns The 1990s saw a surge in the release of interfaces for ITPs decoupled from the proof checker, typically communicating with the system through a protocol. In some ways, this was a natural path of evolution from the way that users typically interacted with the REPLs for existing ITPs. While the REPL was typically exposed through a command line tool, it was common to instead use multiple Emacs buffers, one for development and for the proof assistant top-level, and to copy definitions between the two. This approach is still used in some modern proof assistants such as HOL (HOL Development Team, 2018).

Figure 7.1: Agda Emacs mode (from Barret (2018))

This mode of interaction naturally led to the development of Emacs modes which interact with the REPL or theorem prover backend. For example, Isamode (Aspinall, 2000a) for Isabelle99 was an Emacs mode for Isabelle which smoothed interaction with the REPL. The HOL4 Emacs mode (Myreen, 2008-2018) is still used to this day. The Agda Emacs mode, which interacts with the Agda backend, is similar in spirit but contains more advanced functionality; it allows the user to, for example, define holes in terms and fill those holes in later in development. Idris includes an Emacs mode (Mehnert and Christiansen, 2014) for interacting with the REPL, which inspired by both the Agda Emacs mode and Proof General.

Other interfaces beyond Emacs modes communicate with the backend theorem prover or REPL in this style. For example, ALF, a predecessor to Agda, included a window-based interface for communicating with the backend (Altenkirch *et al.*, 1994). The lightweight interface TkHOL (Syme, 1995) for HOL also follows in this style. Bertot and

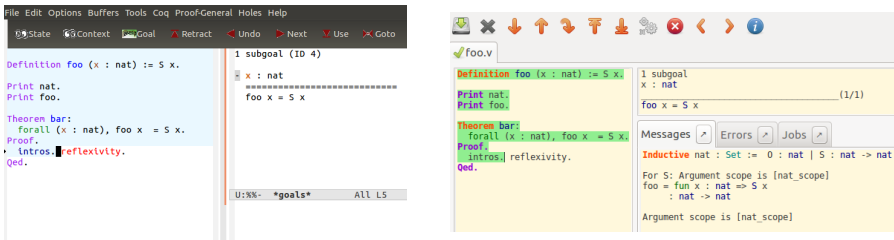

Figure 7.2: Proof General (left) and CoqIDE (right) for Coq

Théry (1998) describes a generic approach for building an interface that communicates with the ITP using a protocol, inspired by the early Coq user interface CtCoq.

In some cases, these interfaces were entirely independent of the underlying proof assistant. One notable example of such an interface from this generation is the Emacs extension Proof General (Aspinall, 2000b), an interface for proof development that supports multiple proof assistants. Proof General has seen widespread use, especially within the Coq community. While Proof General best supports Coq, it also has support for LEGO, PhoX, and an old version of Isabelle, as well as experimental support for other proof assistants (PG development team, 2016). It is simple yet easily extensible, both to support new proof assistants and to add new functionality for existing proof assistants. Company-Coq (Pit-Claudel and Courtieu, 2016) for example, extends Proof General with many new features for Coq, including improved autocompletion, and integration of documentation.

Following the success of Proof General, Coq released the lightweight interface CoqIDE (Coq Development Team, 1999-2018a) as part of Coq 8.0 (Coq Development Team, 2003). Its main selling point was speed: It claimed to be faster than Proof General. In addition, CoqIDE's native support for Coq means that it is always maintained and distributed with new versions of Coq, imposing minimal overhead on users. Figure 7.2 shows CoqIDE and Proof General side-by-side for Coq.

Both third-party interfaces and native interfaces from this generation continue to be popular to this day. This separation of concerns has also inspired a new generation of specialized interfaces (Section 7.1.2) for proof assistants.

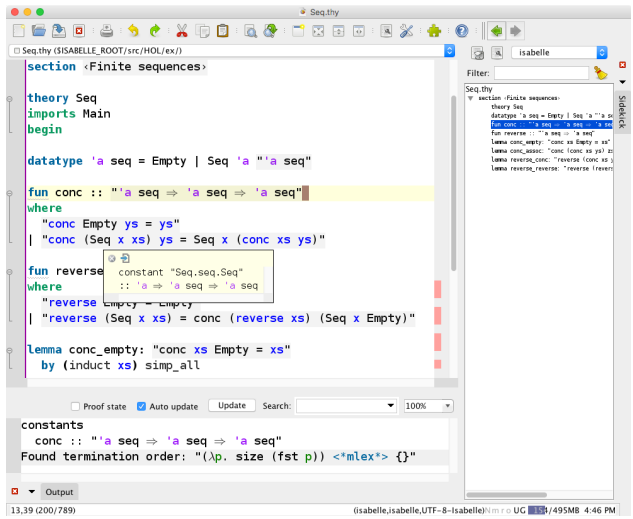

Figure 7.3: Isabelle/jEdit (from Wenzel (2018b))

Generation III: Full IDEs The third generation of user interfaces coincides with the rise of proof development of large projects and the corresponding increase in concern for good proof engineering support. Interfaces from this generation focus on scaling to large developments. For example, early user interfaces did not support *asynchronous development*: they did not allow the user to run the proof checker on some proofs while modifying others. Early user interfaces also did not have support for project management, and so were not truly full-scale IDEs.

Many IDEs in the latest wave of development address these concerns. Coqoon (Faithfull *et al.*, 2018), for example, is an IDE for Coq built on Eclipse with support for both project management and asynchrony. The PIDE framework, originally developed for use with the Isabelle IDE Isabelle/jEdit (Wenzel, 2014), also supports asynchronous development; Isabelle/jEdit is shown in Figure 7.3.

PIDE is ultimately indifferent to the backend theorem prover; Wenzel (2013a) and Barras *et al.* (2015) describe interfaces built on PIDE for Coq. PIDE also has additional interfaces in Isabelle aside from the default Isabelle/jEdit, including Isabelle/VSCode (Wenzel, 2017c), an

Isabelle plugin for Visual Studio (Microsoft, 1997-2019). PIDE has seen enough success that Isabelle has done away with its REPL entirely.

Like Isabelle, Lean also has an IDE implemented as a Visual Studio plugin (Lean Development Team, 2016-2019). This IDE communicates with the Lean server, and supports incremental compilation and proof checking, debugging, documentation, and batch execution.

Many existing proof assistant interfaces have integrated features from this generation. For example, CoqIDE now supports asynchrony (Coq Development Team, 1999-2018a). It remains to be seen to what extent full-scale IDEs for proof assistants will continue to evolve and to grow in popularity.

7.1.2 Specialized Interfaces

The separation of concern from Generation II interfaces for proof assistants inspired the development of specialized interfaces. For example, web-based interfaces require minimal setup and installation, and so are thought to be less intimidating to new users, especially students. Many web-based interfaces are built with students as the key audience to address concerns students have about installing and using heavyweight IDEs. Examples of web-based interfaces for proof development include ProofWeb (Kaliszyk, 2007), jsCoq (Gallego Arias *et al.*, 2017), and PeaCoq (Robert and Lerner, 2014-2016). The Lean 2 tutorial (Lean Development Team, 2014-2017) uses the Lean.JS (Lean Development Team, 2017-2018) web interface for Lean to provide an interactive learning experience directly in the browser.

Proof assistant users sometimes note that the experience of writing proofs has game-like elements. The interactive nature of a proof assistant, for example, is similar to interacting with an adversary in a game. There is some work on gamification of proofs that reifies this intuition into the interface itself. In these games, players can generate program annotations (Dietl *et al.*, 2012), write natural deduction proofs (Lerner *et al.*, 2015), and identify inductive invariants (Bounov *et al.*, 2018), all the while having low-level details of these proofs abstracted away from them. While these games are not interfaces for well-known ITPs like Coq and Isabelle, they may help with tasks that can assist users in writing proofs,

such as finding inductive invariants. Applying this same intuition to build new interfaces for commonly-used proof assistants may help make them more accessible to non-experts in the future.

7.1.3 Interface Usability Analysis

Traditional software engineering tools and interfaces are often subject to usability analyses according to the conventions in human-computer interaction (HCI). There are some similar analyses related to proof assistants. Aitken *et al.* (1998) propose a three-layer model to account for user interaction with a proof assistant, and perform an empirical study which concludes that there is support for the view of "proof as programming" for proof assistant interaction, rather than "proof by pointing" (Bertot *et al.*, 1994) and "proof as structure editing". Kadoda *et al.* (1999) analyze the usability of theorem provers in a cognitive framework by using questionnaires. Aitken and Melham (2000) analyze errors in proof attempts. Beckert *et al.* (2014) use focus groups to evaluate usability of proof assistants, finding that users prefer proof assistants that produce intuitive proofs, can present comprehensible proof steps, and provides a convenient interface.

7.1.4 Tooling for User Support

In parallel with the evolution of user interfaces, recent years have seen an emphasis on tooling to help users with proof development. These features are more useful than ever because of the advent of Generation II user interfaces that are not tightly tied to the REPL.

Many of the user support features that are now arising for proof engineering echo similar features that already exist in languages with more mature IDEs. For example, languages with more mature IDEs often integrate refactoring tools into those IDEs; now that proof assistant interfaces are maturing, interfaces with refactoring support such the Coq interface CoqPIE (Roe and Smith, 2016) are beginning to emerge. We discuss more refactoring tools in Section 7.2.2.

In addition, new techniques are extending the reach of user support features to support the challenges particular to proof development. For example, one common challenge in proof engineering is efficiently finding

relevant datatypes and proofs Many proof assistants distribute tools for this by default. For example, Coq includes the `Search` command, which SSReflect (Gonthier and Mahboubi, 2010) extends; Isabelle includes the `find_theorems` and `find_consts` commands. This challenge has also inspired several external tools, including the web-based tool Whelp (Asperti *et al.*, 2004) for Coq, upon which Matita builds (Asperti *et al.*, 2007).

Machine-learning techniques can also help with challenges in proof developments, for example by suggesting hints to users. Recent tooling of this flavor includes the ML4PG (Komendantskaya *et al.*, 2012) extension to Proof General, which uses machine learning to suggest hints during proof development, and ACL2(ml) (Heras *et al.*, 2013), which uses machine learning to suggest auxiliary lemmas for ACL2 development. Nagashima and He propose a proof method recommendation system for Isabelle/HOL based on machine learning, which is trained on large proof corpora (Nagashima and He, 2018).

Unlike traditional software development, proof development with ITPs often involves significant interaction with automation. Accordingly, one question that many tools explore is the ideal user experience for interacting with automation. The web-based IDE PeaCoq (Robert and Lerner, 2014-2016), for example, has extra support for tactic previews and context management. Matita (Asperti *et al.*, 2007) includes special support for contextual term manipulation, and for understanding the execution of tactical-like chains of tactics.

Another common problem in proof development is that the proof engineer may accidentally state a false theorem, or may be unsure if a stated theorem is true. When a stated theorem is false, it can be difficult to determine that the theorem is actually false; the proof engineer may instead think his inability to prove the theorem is due to his own shortcomings. Hammers and other general-purpose automation (Section 5.2.1) can help a proof engineer discharge simple proof obligations and quickly determine that a theorem is true; the proof engineer can then reprove the theorem in a different way if desired. Property-based testing tools like Quickcheck for Isabelle (Bulwahn, 2012) and QuickChick (Lampropoulos *et al.*, 2017; Paraskevopoulou *et al.*, 2015) for Coq can help users identify counterexamples to false properties.

7.1.5 Future of User Interfaces

Many of the Generation I interfaces still exist today. We expect some of these will continue to exist, since they are useful when resources are limited. However, there is a growing trend of moving away from the REPL in some ITPs such as Isabelle/HOL; perhaps more ITPs will move in that direction.

The interactive nature of the REPL makes it simple to collect fine-grained data on how proof engineers develop code. For proof assistants that are backed by a REPL, collecting this data could help with the development of better tooling to support proof engineers during development. Similarly, while there is some empirical information on how proof engineers interact with different user interfaces, this is still a lot more ground to cover. Even collecting simple information like the number of users of each interface for each proof assistant over time may help gauge the impacts of different design decisions. More user studies on interacting with proof assistants could also help pave the way for more useful interfaces.

We expect that the separation of concerns emphasized with Generation II interfaces has had a strong influence and will likely continue to have a strong influence. Separation of concerns and extensibility may be part of why Proof General has continued to be successful after so many years. In many ways, this mirrors the success that is seen in successful IDEs for software engineers, such as Eclipse (Eclipse Foundation, 2001-2019) or IntelliJ IDEA (JetBrains, 2001-2019). We expect that future developments will continue to work on separation of concerns and extensibility, with better plugin systems for IDEs to support more features with minimal effort for the interface developer and for the proof engineer.

Emacs has played a crucial role in the history of the development of IDEs for ITPs, with many early interfaces implemented as Emacs modes. Recent years have seen the development of IDEs as Visual Studio plugins for Isabelle/HOL and for Lean. In the future, perhaps more proof assistants will implement IDEs as plugins for existing IDEs such as Visual Studio and Eclipse, and perhaps these existing IDEs will play

a similar role to Emacs in the continued development of proof assistant IDEs.

We also expect that project management will continue to grow in importance as large proof developments become more common. Few interfaces for proof assistants currently have strong support for project management; this is an area ripe for improvement. Better integration of build tools and continuous integration tools can greatly improve development experience.

Finally, we expect more productivity tools to emerge, like the refactoring tools that already exist, and for these tools to be integrated into IDEs. For example, most mature IDEs for existing languages have strong support for debugging. Debugging tools for proof assistants, on the other hand, are few and far between. Better plugin systems for interfaces could help minimize the friction in supporting these features at the IDE level.

7.2 Proof Evolution

Programs change over time, and so proofs about programs must change with those programs. This concern is raised in the Social Processes (DeMillo *et al.*, 1977) critique of program verification as a barrier for the verification of real programs. This barrier has been realized in real developments; a review (Elphinstone and Heiser, 2013) of the evolution of the seL4 verified OS microkernel (Klein *et al.*, 2009), for example, notes that while customizing the kernel to different environments may be desirable, "the formal verification of seL4 creates a powerful disincentive to changing the kernel." Leroy *et al.* (2012) motivates and describes updates to the initial CompCert memory model that include changes in specifications, automation, and proofs (CompCert Development Team, 2010).

Changes in programs and proofs are not always in the proof engineer's control— updating a standard library, for example, can lead to proofs in client code failing during regression proof checking (Section 7.2.1). *Reactive* approaches to proof evolution address changes that occur outside of the proof engineer's control. These approaches contrast with and are complementary to *proactive* approaches that address

brittleness ahead of time, such as the design principles discussed in Section 6.2.

Consider, for example, a Coq proof that uses the `intros` tactic. If the user does not pass identifiers to `intros`, then Coq automatically chooses hypothesis names. Small changes to the theorem statement or to the proof can change the names of the hypotheses that Coq chooses, which can make proofs that refer to those hypotheses brittle. We briefly discussed two proactive approaches to this problem in Section 6.2.3: explicitly choosing identifiers to pass to `intros`, and writing tactics that do not refer to these hypotheses at all. In contrast with these proactive approaches, the IDE CoqPIE (Roe and Smith, 2016) automatically renames references to hypotheses in proofs to work around this problem reactively.

The renaming functionality of CoqPIE is an example of proof refactoring (Section 7.2.2), a reactive approach to proof evolution. Proof repair (Section 7.2.3) is a similar reactive approach to proof evolution. The main distinction between these two approaches is that proof refactoring is semantics-preserving, while proof repair need not be. Nonetheless, these technologies often overlap.

7.2.1 Regression Proving

Regression proving is the process of rechecking proofs after a change to a verification project, mirroring *regression testing* for software projects. For large-scale projects, regression proving may require considerable machine time—from tens of minutes and hours up to several days. This can negatively affect the productivity of proof engineers. Absent domain- and context-specific knowledge, as in proof refactoring, the two main techniques to speed up regression proving are proof-checking parallelization (Wenzel, 2013b; Barras *et al.*, 2013) and proof selection (Celik *et al.*, 2017).

Support for parallelization varies in degree and kind among proof assistants. Isabelle leverages the support for threads in its host compiler, Poly/ML, to spawn proof checking tasks processed by parallel workers. Using a notion of *proof promises*, proofs that require previous unfinished result can proceed normally and become finalized when extant tasks

terminate (Wenzel, 2013a). Isabelle also includes a build system with integrated support for checking of proofs and management of parallel workers. The lack of native threads in OCaml prevents similar low-cost fine-grained parallelism for Coq. However, spawning parallel operating system processes is still possible, and such processes can be leveraged for both file-level parallelism and to check fine-grained proof tasks (Barras *et al.*, 2015). Lean supports fine-grained parallel proof checking (de Moura *et al.*, 2015). Compared to parallelization of test execution for software projects, checking a proof is deterministic and has no side-effects detrimental to checking other proofs.

A *regression proof selection* (RPS) technique limits the scope of regression proving to those proofs that are affected by a change to a project. While selection at the file level (modulo file dependencies) is broadly supported via build systems such as `make`, only some proof assistants such as Isabelle and Coq supports selection of individual proofs; this is made possible by support for *asynchronous* proof checking (Wenzel, 2014; Barras *et al.*, 2015). Celik et al. proposed an RPS technique for Coq that combines dependency analysis at the file and proof levels (Celik *et al.*, 2017); their tool implementation, dubbed iCoq, compares checksums of files, terms, and proof scripts to locate and run affected proofs sequentially. In an evaluation on the revision histories of several large-scale Coq projects, iCoq was up to 3 times faster than using conventional `make`-style checking with a persistent store, and up to 10 times faster than conventional checking when each revision is checked from a clean slate.

Palmskog *et al.* (2018) defined a taxonomy of regression proving techniques for proof assistants that include both parallelism and selection. Along one axis, they consider parallelization at the file and proof granularity. Along the other axis, they consider selection of files and proofs. Their most sophisticated technique combines proof selection and fine-grained parallelization, and consistently outperforms other techniques on the revision histories of several Coq large-scale projects.

Wenzel (2017b) and Wenzel (2018a) outlined how to scale Isabelle for large projects using both parallelism and other techniques.

7.2.2 Proof Refactoring

Refactoring is the restructuring of code in a way that preserves semantics (Opdyke, 1992); *proof refactoring* is the refactoring of proofs (Whiteside, 2013). Proof refactoring tools help automate this process, propagating a single change throughout the proof development. Like program refactoring tools, proof refactoring tools can help keep developments maintainable as they change over time (Bourke *et al.*, 2012). In that way, it is possible to consider refactoring tools as both proactive and reactive approaches to proof evolution, though we consider them here through a reactive lens.

Some proof assistants expose tactics (Section 5.1.1) or proof languages (Section 5.1.2) in which the proof engineer can write high-level proof scripts to guide proof search. Some proof refactoring tools refactor these proof scripts directly. One such tool is POLAR (Dietrich *et al.*, 2013), a generic framework for proof script refactoring. POLAR is instantiated with two languages, both of which are based on Isabelle/Isar (Wenzel, 2007): Hiscript (Whiteside, 2013), a language with support for refactoring, and ΩSCRIPT (Dietrich, 2011), a language with support for proof planning (Section 6.4.3). Refactoring in POLAR works through a combination of rewrite rules that operate over a graph representation of the underlying language. POLAR implements ten kinds of refactorings by default, and also supports custom refactorings. It guarantees that all lemmas that go through before the refactoring continue to go though after the refactoring.

Some proof refactoring tools focus on specific refactoring tasks that are common in proof development. For example, Levity (Bourke *et al.*, 2012) is a proof refactoring tool for an old version of Isabelle/HOL that automatically moves lemmas to maximize reuse. The design of Levity is informed by experiences with two large proof developments. Levity addresses problems that are especially pronounced in the domain of proof refactoring, such as the context-sensitivity of proof scripts. Tactician (Adams, 2015) is a refactoring tool for proof scripts in HOL Light that focuses on refactoring proofs between sequences of tactics and tacticals.

There is little work on refactoring proof terms (Section 4.3.1) directly. This is the main focus of Chick (Robert, 2018), which refactors terms in a dependently-typed functional language similar to Gallina. To use Chick, the proof engineer applies some refactorings. Chick then uses a program differencing algorithm to determine the changes to make elsewhere in the program, then makes those changes. Chick supports insertion, deletion, modification, and permutation of subterms. Similarly, RefactorAgda (Wibergh, 2019) is a refactoring tool for a subset of Agda that operates directly over Agda terms. RefactorAgda supports many changes, including changing indentation, renaming terms, moving terms, converting between implicit and explicit arguments, reordering subterms, and adding or removing constructors to or from types; it also documents ideas for supporting other refactorings, such as adding and removing arguments and indices to and from types.

For both Chick and RefactorAgda, only some of these changes are semantics-preserving. Adding a new index to a type, for example, does not preserve the semantics of the original program. Accordingly, these tools can be viewed as both refactoring and repair tools, though the algorithms that they use are syntactic.

A natural integration point for a proof refactoring tool is at the level of a platform or an IDE. The Coq IDE CoqPIE (Roe and Smith, 2016) for Coq takes this approach for refactoring proof scripts. CoqPIE includes a *Replay* button which steps through the proof while renaming any changed hypothesis names. CoqPIE can also automatically split out intermediate goals from a proof into separate lemmas. There are plans to support more refactoring functionality in CoqPIE in the future.

7.2.3 Proof Repair

Program repair (Monperrus, 2018) is the automatic patching of programs to fix bugs; *proof repair* is program repair for proofs. Proof repair tools automatically fix broken proofs. Recent lessons from a review of a certain class of program repair tools (Qi *et al.*, 2015) highlight why proof repair is a particularly good domain of program repair. The review demonstrates that many existing tools produce incorrect patches. Among the recommendations the authors make to remedy this is the

suggestion that program repair tools make use of extra information such as specifications, code from other applications, or example patches when generating patches.

In proof repair, a specification is always available: the theorem the repaired proof ought to prove. Some proof repair tools take this a step further and make use of additional extra information, such as examples patches. One such tool is PUMPKIN PATCH (Ringer *et al.*, 2018), a proof repair tool for Coq that generalizes example patches. PUMPKIN PATCH takes as inputs an old proof and a new proof that addresses some change in specification. From those, it identifies a reusable patch that describes the change in specification; for the kinds of changes PUMPKIN PATCH can currently handle, this patch is a Gallina function. The proof engineer can then use this patch to patch other proofs broken by the change in specification. PUMPKIN PATCH has only preliminary tooling (Ringer and Yazdani, 2018) for applying patches automatically, and currently handles only simple changes.

Chick (Robert, 2018) was developed in parallel to PUMPKIN PATCH, and has a similar workflow: Chick takes a set of example changes supplied by the programmer, and uses a program differencing algorithm to determine the changes to make elsewhere. Unlike PUMPKIN PATCH, Chick also applies the changes it finds. However, Chick does this using a syntactic algorithm that handles only simple transformations; for this reason, it presents itself primarily as a refactoring tool, even though the changes it makes may not preserve semantics. The refactoring tool RefactorAgda (Wibergh, 2019) similarly describes some semantics-changing repairs for a subset of Agda.

While proof repair is analogous to program repair, it was born out of traditional proof reuse (Section 6.4). For example, PUMPKIN PATCH discovers patches which help adapt a proof of a theorem to a proof of a related theorem, and can so be thought of as a tool to assist in proof by analogy (Curien, 1995). Similarly, the proposed proof weaving (Mulhern, 2006) method to automatically satisfy new obligations generated in response to changes in inductive types can be viewed as a proof repair technique. Proof planning critics (Ireland, 1996) can also be viewed as a technique for proof repair.

New technologies continue to make proof repair more feasible. GALILEO (Chan *et al.*, 2011) is a tool build on Isabelle for identifying and repairing faulty ontologies in response to contradictory evidence; it has been applied to repair faulty physics ontologies, and may have applications more generally for mathematical proofs. GALILEO uses repair plans to determine when to trigger a repair, as well as how to repair the ontology.

Knowledge sharing methods (Gauthier and Kaliszyk, 2014) match concepts across different proof assistants with similar logics and identify isomorphic types, and may have implications for proof repair. Later work uses these methods in combination with HOL(y)Hammer to reprove parts of the standard library of HOL4 and HOL Light using combined knowledge from the two proof assistants (Gauthier and Kaliszyk, 2015). More recently, this approach has been used to identify similar concepts across libraries in proof assistants with different logics (Gauthier and Kaliszyk, 2019). These methods combined with automation like hammers may help the proof engineer adapt proofs between isomorphic types, and may have applications when repairing proofs even within the same logic, using information from different libraries, different commits, or different representations of similar types.

7.2.4 Future of Proof Evolution

There is a lot of room for work in proof evolution—only a few techniques exist so far, many of which emerged in parallel. We expect these reactive approaches to continue to evolve alongside proactive approaches like design principles, as the two approaches are complementary. Proof evolution can help with changes that occur outside of the programmer's control, such as changes in dependencies (examples of this can be found in Ringer *et al.* (2018)) and changes that are difficult to protect against even with informed design (examples of this can be found in Klein (2014)).

Ideally, proof evolution tools ought to integrate naturally with the workflows of proof engineers, for example through integration with existing tactic or proof languages, or through IDE or continuous integration support. While some proof evolution tools focus on this already and

can offer useful insights, this can be challenging. For example, refactoring Ltac proof scripts can be difficult, since the semantics of Ltac are not well-defined; Ltac2 (Pédrot, 2019) may simplify this in the future. Robert (2018) discusses the challenges involved in refactoring proof scripts in more detail. Ringer *et al.* (2018) also discusses the challenges of workflow integration, along with other open problems in proof repair. We expect to see more emphasis on addressing these challenges in the future.

There is only preliminary work exploring how much of the work from existing refactoring and repair tools for programming carries over to the domain of proof assistants. It is worth exploring in more detail which challenges are unique to this domain. For example, Qi *et al.* (2015) provides several recommendations for how program repair tools can make use of extra information such as examples to make searching for patches more feasible; PUMPKIN PATCH and machine learning tools use examples for this purpose already. Future proof refactoring and repair tools can similarly learn from those recommendations.

One tempting use case for proof refactoring and repair tools is when a library changes a specification that breaks proofs in client code that uses those libraries. Current refactoring and repair tools, however, rely each individual client to determine the appropriate refactors and repairs to make to fix those proofs. To better address this problem, future refactoring and repair tools can provide support at the level of library design. A library designer may, for example, specify how something has changed to a tool; the tool may then apply this information in client code automatically. Some program repair tools already support library-provided patches (Monperrus, 2018); we expect to see this extend to proof refactoring and repair tools in the future.

One barrier to useful refactoring and repair tools for proof engineers is the lack of information on the kinds of changes that proof engineers make in practice. Collecting data on the changes that proof engineers make and classifying it could help guide refactoring and repair tools to handle classes of changes that matter in practice, and could also help machine learning tools gather both positive and negative examples. Similarly, collecting the benchmarks and examples from both proactive and reactive approaches to proof evolution such as Planning for Change,

seL4, iCoq, and PUMPKIN PATCH can help drive the development of future proof evolution tools and measure their success meaningfully.

7.3 User Productivity and Cost Estimation

Bourke *et al.* (2012) outline challenges in large-scale verification projects using proof assistants: (1) new proof engineers joining the project, (2) expert proof engineering during main development, (3) proof maintenance, and (4) social and management aspects. They highlight three lessons: (1) proof automation is crucial, (2) using introspective tools for quickly finding facts in large databases gain importance for productivity, and (3) tools that shorten the edit-check cycle increase productivity, even when sacrificing soundness.

Zhang *et al.* (2012) present a simulation model of the process of verifying the operating system kernel seL4. Their model is expressed as a software process using the tool Vensim. Andronick *et al.* (2012) describe the development process and management issues in verifying seL4. They conclude that formal verification, and re-verification, for systems requiring in the order of 10,000 LOC is feasible using a proof assistant. Staples *et al.* (2013) studied the relationships between sizes of artifacts in seL4. They find that the formal specifications have a significant relationship with the size of the verified executable code. Staples *et al.* (2014) study the *proof productivity* problem in the context of seL4; they find that effort is correlated linearly with proof size. Matichuk *et al.* (2015a) analyze the Isabelle/HOL specifications and proof scripts from the seL4 project, and find a quadratic relationship between the size of a formal property and the proof script required to prove it.

Jeffery *et al.* (2015) identify 30 research questions about productivity in application of formal methods, such as verification using proof assistants. Klein *et al.* (2017) outline the benefits of trustworthy systems.

7.4 Mining and Learning from Proof Repositories

Mining software repositories is an emerging field that analyzes software repositories to yield actionable information about software systems and

their development and evolution. We describe similar forms of analysis that have been carried out for repositories with proof assistant code.

Wiedijk (2009) compared statistics for standard libraries of several proof assistants for versions available around 2009, including Isabelle/HOL, Coq, and HOL Light. For each library, he reports the number of lines of comments, proofs, definitions, etc. Despite foundational differences, the numbers are similar, with HOL Light having the smallest number of lines for definitions. For example, the LOC shares of theorem statements, definitions, and proof in the Coq version 8.1 standard library were 11%, 8%, and 53%, respectively. Wiedijk argues informally that fewer definitions per proof means higher trustworthiness, since having proofs of relevant properties yield higher confidence in the adequacy of definitions.

Blanchette *et al.* (2015) investigated Isabelle's Archive of Formal Proofs (AFP), analyzing among other properties the number and sizes of proofs, interdependencies between projects, and number of authors. For the AFP in aggregate, the LOC shares of theorem statements, definitions, and proofs were 19%, 8%, and 58%, respectively. They found that the Isabelle Sledgehammer tool for proof automation (Paulson and Blanchette, 2012) could prove about 60% of all theorems in the AFP.

Software metrics provide quantitative ways to describe software artifacts and processes and discover new properties. Aspinall and Kaliszyk (2016a) first considered analogous metrics for formal proofs. More specifically, they define an abstract model of formal proofs and a set of proof metrics for this model, which they implement for three different proof assistants (Isabelle, Mizar, and HOL Light) and apply to several large proof corpora.

Komendantskaya *et al.* (2012) used machine learning with clustering algorithms to identify patterns in large collections of Coq tactic sequences and proof trees, e.g., to find structural similarities between lemmas, and Heras and Komendantskaya (2013) and Heras and Komendantskaya (2014) highlighted how statistical patterns in proofs can be leveraged during interactive proof development. Aspinall and Kaliszyk (2016b) used machine learning, in the form of a k-nearest-neighbor classifier, to learn and suggest theorem names in HOL Light projects that accurately reflect their property definitions.

Müller *et al.* (2017) proposed a format and database for capturing and leveraging *alignments* between concepts in different proof assistants, e.g., between natural numbers in Coq on one hand and Isabelle/HOL on the other. One of the basic assumptions in alignment is that concepts have syntactic and semantic similarities across environments, consistent with repetitiveness assumptions in naturalness. Gauthier and Kaliszyk (2019) proposed an algorithm based on heuristics for generating alignments given two proof assistant libraries, and evaluated it on libraries from six proof assistants. For example, by evaluating a library against itself for alignment, duplicated concepts can be found.

Kaliszyk *et al.* (2017b) leveraged statistical machine learning techniques in a tool that automatically translates ("formalizes") mathematical texts to proof assistant code. Their approach and evaluation is based on learning and cross-validation using a corpus with established alignments between English texts and HOL Light documents, based on the Flyspeck project (Hales *et al.*, 2017). They find that the number of correct translations among the top 20 is 64%.

There are many recent lines of work that learn from large proof assistant corpora to directly perform various automated reasoning tasks (Kühlwein *et al.*, 2012; Kühlwein *et al.*, 2013; Kaliszyk and Urban, 2014; Irving *et al.*, 2016; Loos *et al.*, 2017; Peng and Ma, 2017); the HOLStep dataset (Kaliszyk *et al.*, 2017a) is designed as a benchmark for training and evaluating such techniques in a proof assistant context. Gauthier *et al.* (2017) and Gauthier *et al.* (2018) proposed a technique for learning from HOL4 tactic sequences and proof states and automatically suggest tactic-based proofs of theorems. They achieved around 66% success rate on the HOL4 standard library, and by also incorporating the automated E prover into the toolchain, they raised the success rate to 69%. Huang *et al.* (2019) similarly learn from tactics and proof states, but in the context of Coq and for a limited set of algebraic proof goals. Yang and Deng (2019) proposed a more general tactic-based approach for learning and automatic proof suggestion for Coq, which achieved around 12% success rate on proofs from a large dataset of 123 Coq projects. Nagashima and He (2018) used custom encodings of proof state in Isabelle/HOL for learning in order to predict suitable proof methods

(essentially powerful domain-specific proof tactics) to apply. Bansal *et al.* (2019) presented a learning environment for HOL Light.

The version of record is available at: http://dx.doi.org/10.1561/2500000045

8

Conclusion

Proof engineering has come far since its infancy in the 1970s (Milner, 1972; Milner and Weyhrauch, 1972), drawing on hundreds of years of foundational ideas in mathematics and logic. Researchers and proof engineers have used proof assistants to build software artifacts spanning hundreds of thousands of lines of code (Leroy, 2009; Klein *et al.*, 2009). A growing fraction of these artifacts are executable on real hardware, and of these, some are verified down to machine code for verified hardware. Verified artifacts have shown themselves to be more reliable (Yang *et al.*, 2011), and are beginning to see industrial applications (Kästner *et al.*, 2018; Erbsen *et al.*, 2019).

Compared to most research software, widely used proof assistants such as Coq and Isabelle/HOL are mature and well-maintained tools, with large communities, large software ecosystems, wide selections of support tools, and sophisticated interfaces. Interest in verification across academia and industry builds additional momentum for proof assistant development. Interest in formal proofs among mathematicians, for example, results in rich libraries (Voevodsky *et al.*, 2011-2019; Bauer *et al.*, 2017), foundational advances (Univalent Foundations Program,

2013), and tooling (Braibant and Pous, 2011) useful for verifying software. Advances in automated theorem proving reach proof engineers through tools like hammers (Blanchette *et al.*, 2016b), which allow them to benefit from cutting-edge research without increasing the trusted computing base. Adaptations of research on programming practices and developer support tools and systems from software engineering help proof engineers continually increase productivity.

We are living in the age of "big verification" (ACM SIGPLAN, 2016). A new generation of computer science students are learning to use proof assistants in undergraduate and graduate courses (Coq Development Team, 2017-2019), entering the workforce equipped with the skills to verify software. As the scale of that verified software continues to grow, the challenges of proof engineering will continue to grow more significant and salient. This survey concludes with a discussion of five of the opportunities to address high-level challenges that remain.

Opportunity 1: Adapting Tools and Ideas from Software Engineering Development processes, build workflows, and support infrastructure for proof assistants are far behind those for traditional software development, and proof engineer productivity is consequently far from its potential. Communities and ecosystems are small compared to those for traditional software. Many features of and tools for proof assistants are not adequately documented, and proof engineers struggle to develop and maintain verified software in the face of evolving proof assistants, libraries, and requirements (Bourke *et al.*, 2012). Domains may lack usable frameworks and libraries to build on, forcing time-consuming development from scratch. Interfaces may lack in usability and key features, and may become unmaintained and obsolete. Results in one proof assistant cannot be readily used in another, except in special cases. Proof engineering is particularly far behind software engineering with respect to maintenance, disincentivizing experts (Elphinstone and Heiser, 2013) from changing the system once it has been verified.

Proof engineering has already benefited from traditional software engineering, for example through many of the design principles discussed in Chapter 6. It can continue to draw on tools and ideas from software engineering when applicable. The surveyed work suggests that it is

sometimes necessary to adapt these tools and ideas, both to address challenges unique to or especially pronounced in proof engineering (such as brittleness), and to fully take advantage of the opportunities that proof engineering presents (such as the availability of full specifications). Continuing to transfer ideas and tools from software engineering to proof engineering with these differences in mind may help close the gap between the two disciplines. Ideas and tools ripe for transfer include improved continuous integration systems, package systems, source code hosting, graphical interfaces, error messaging, debugging tools, and development processes.

Opportunity 2: Making Proof Assistants More Accessible Compared to traditional programming languages, systems, and environments, proof assistants can be hard for non-experts to understand. Their foundational bases in logic, mathematics, and type theory, which have served to maintain trust, may also deter potential users due to perceived complexity. Omissions in and misunderstandings of specifications may lead to lowered expectations and negative perceptions of formally verified software. An overwhelming majority of large successful software verification projects using proof assistants are carried out and maintained by small teams of highly specialized and trained researchers, frequently with close ties to the institutions where the corresponding proof assistant is developed (Gonthier *et al.*, 2013).

Some of this inaccessibility may dissipate over time, as students become more familiar with concepts like inductive or dependent types through the use of interactive theorem provers in computer science courses, or through the availability of online books and tutorials. Some may inspire new abstractions around concepts that are not accessible to the average programmer. For example, new abstractions may help non-experts more easily interface with unification algorithms and existential variables. The interactive theorem proving community can continue to draw on automated theorem proving to help make interactive theorem proving more accessible and increase its reach and impact, not just through hammers, but also as part of counterexample generators and similar tools (Blanchette and Nipkow, 2010). Techniques from the

broader formal methods community can even be certified in proof assistants, and then applied inside them.

Opportunity 3: Understanding and Evaluating Development Processes It is difficult to improve the state of the art in proof engineering without understanding the status quo. The surveyed work suggests that little work has been done to understand and assess the current development processes of proof engineers, and that some of the work that has been done in this direction has been inconclusive.

Thousands of proof assistant software verification projects are publicly available on platforms such as GitHub for study, reuse, and as research subjects; curated collections such as Isabelle's Archive of Formal Proofs (Klein *et al.*, 2004-2019) and Coq's OPAM package index (Coq development team, 2018) provide high-quality projects with extensive metadata, guaranteed to work with certain proof assistant versions. These codebases can be investigated empirically, learned from, and used to construct benchmarks when evaluating new proof engineering techniques. Collecting and analyzing data from other points in the development process, such as interaction with the IDE, may further facilitate in these processes, as may user studies and community-wide retrospective discussions of influential work.

Opportunity 4: End-to-End Verification Bugs affecting the soundness of a proof assistant with respect to its foundations, although rare, threaten to undermine trustworthiness. Coq Development Team (2018-2019), for example, contains preliminary documentation of 23 (now fixed) critical bugs in the history of stable releases of Coq. Of the bugs listed, only 1 (fixed in 2015) was assessed as likely to be exploited by chance, but the risk of others was not determined. Extensions to proof assistant kernels may further cloud understanding of the implemented metatheory.

The wide availability of verified systems software provides the basis of a fully verified software ecosystem, with all-encompassing end-to-end guarantees for both functional and non-functional properties for a wide range of practical software. This could rule out many recently discovered

critical bugs and even many prevalent security flaws (Chlipala, 2018). Proof assistant self-verification projects and certified compilers for proof assisants, although still emerging and used by few, promise to eventually make the trusted computing base of verified ecosystems minimal. One important step towards a verified ecosystem is to formalize additional practical programming languages and their semantics and runtime environments.

Opportunity 5: Looking to New Applications In the Social Processes critique of program verification, DeMillo *et al.* (1977) wrote that:

> We believe that, in the end, it is a social process that determines whether mathematicians feel confident about a theorem–and we believe that, because no comparable social process can take place among program verifiers, program verification is bound to fail. We can't see how it's going to be able to affect anyone's confidence about programs.

The authors of this survey find it surprising with this in mind that Leroy (2006) and Leroy (2009) put years of effort into verifying an optimizing C compiler, with little precedent to suggest that such a project could succeed. But it did succeed, and in so doing increased confidence in both the compiler itself (Yang *et al.*, 2011) and in the compiled software (Kästner *et al.*, 2018).

As proof engineers and researchers like Leroy (2006) stretch the boundaries of current proof engineering techniques on new and interesting domains, proof engineering research can continue to grow to support the new needs that arise. This cycle can help build a world with, for example, safer transportation systems or more reliable medical devices. It can help build a world that is a bit more trustworthy.

The version of record is available at: http://dx.doi.org/10.1561/2500000045

Acknowledgements

We would like to thank Dan Grossman, Derek Dreyer, Xavier Leroy, Benjamin Pierce, Andrew Appel, Bob Harper, Jonathan Aldrich, Karl Crary, Adam Chlipala, Chris Martens, Joachim Breitner, Christine Rizkallah, Giuliano Losa, Thomas Tuerk, Roberto Guanciale, Doug Woos, James R. Wilcox, Ryan Doenges, Jared Roesch, Sorin Lerner, Leslie Lamport, Fred Schneider, John Leo, Bob Atkey, Lars Hupel, Buday Gergely, Makarius Wenzel, Jasmin Christian Blanchette, Matthieu Sozeau, Cyril Cohen, David Thrane Christiansen, Sam Tobin-Hochstadt, Mario Alvarez, Joomy Korkut, Robert Rand, Taylor Blau, Anna Kornfeld Simpson, Jonathan Sterling, Colin Barret, Toby Murray, Gerwin Klein, Graydon Hoare, Emilio Jésus Gallego Arias, and Brendan Zabarauskas. We are grateful to the anonymous reviewers for their detailed comments and valuable suggestions. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE-1256082, the National Science Foundation under Grant Nos. CCF-1652517, CCF-1836813, and CCF-1749570, and by a Research Award from Facebook. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

Abel, A., A. Momigliano, and B. Pientka. 2017a. "POPLMark Reloaded". In: *Proceedings of the Logical Frameworks and Meta-Languages: Theory and Practice Workshop*.

Abel, A., A. Vezzosi, and T. Winterhalter. 2017b. "Normalization by Evaluation for Sized Dependent Types". *Proc. ACM Program. Lang.* 1(ICFP): 33:1–33:30. DOI: 10.1145/3110277.

ACL2 Development Team. 1990-2019. "ACL2". URL: http://www.cs.utexas.edu/users/moore/acl2.

ACM SIGPLAN. 2016. "Most Influential POPL Paper Award". URL: http://www.sigplan.org/Awards/POPL.

Adams, M. 2015. "Refactoring Proofs with Tactician". In: *Software Engineering and Formal Methods*. Berlin, Heidelberg: Springer. 53–67. DOI: 10.1007/978-3-662-49224-6_6.

Agda Development Team. 2005-2017. "Quick Guide to Editing, Type Checking and Compiling Agda Code – Agda 2.5.4.2 Documentation". URL: http://agda.readthedocs.io/en/v2.5.4.2/getting-started/quick-guide.html#quick-guide-introduction.

Agda Development Team. 2005-2018. "Cubical Type Theory in Agda – Agda 2.6.0 Documentation". URL: http://agda.readthedocs.io/en/latest/language/cubical.html.

Agda Development Team. 2007-2019. "The Agda Wiki". URL: http://wiki.portal.chalmers.se/agda/pmwiki.php.

Aitken, J., P. Gray, T. Melham, and M. Thomas. 1998. "Interactive Theorem Proving: An Empirical Study of User Activity". *Journal of Symbolic Computation*. 25(2): 263–284. DOI: 10.1006/jsco.1997.0175.

Aitken, S. and T. Melham. 2000. "An analysis of errors in interactive proof attempts". *Interacting with Computers*. 12(6): 565–586. DOI: 10.1016/S0953-5438(99)00023-5.

Aldrich, J., R. J. Simmons, and K. Shin. 2008. "SASyLF: An Educational Proof Assistant for Language Theory". In: *Proceedings of the 2008 International Workshop on Functional and Declarative Programming in Education. FDPE '08*. Victoria, BC, Canada: ACM. 31–40. DOI: 10.1145/1411260.1411266.

Allen, S. F., R. L. Constable, D. J. Howe, and W. E. Aitken. 1990. "The semantics of reflected proof". In: *Proceedings. Fifth Annual IEEE Symposium on Logic in Computer Science*. 95–105. DOI: 10.1109/LICS.1990.113737.

Altenkirch, T., V. Gaspes, B. Nordström, and B. von Sydow. 1994. "A user's guide to ALF". URL: http://www.cse.chalmers.se/~bengt/papers/usersguidetoalf.pdf.

Amani, S., A. Hixon, Z. Chen, C. Rizkallah, P. Chubb, L. O'Connor, J. Beeren, Y. Nagashima, J. Lim, T. Sewell, J. Tuong, G. Keller, T. Murray, G. Klein, and G. Heiser. 2016. "Cogent: Verifying High-Assurance File System Implementations". In: *International Conference on Architectural Support for Programming Languages and Operating Systems*. Atlanta, GA, USA. 175–188. DOI: 10.1145/2872362.2872404.

Ambler, S. J., R. L. Crole, and A. Momigliano. 2002. "Combining Higher Order Abstract Syntax with Tactical Theorem Proving and (Co)Induction". In: *Theorem Proving in Higher Order Logics*. Berlin, Heidelberg: Springer. 13–30. DOI: 10.1007/3-540-45685-6_3.

Notices of the American Mathematical Society. 2008: *A Special Issue on Formal Proof*. URL: http://www.ams.org/notices/200811/.

Amin, N. and T. Rompf. 2017. "Type Soundness Proofs with Definitional Interpreters". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. POPL 2017*. Paris, France: ACM. 666–679. DOI: 10.1145/3009837.3009866.

Anand, A., A. W. Appel, G. Morrisett, M. Weaver, M. Sozeau, O. Savary Belanger, R. Pollack, and Z. Paraskevopoulou. 2017. "CertiCoq: A verified compiler for Coq". In: *CoqPL*. URL: http://www.cs.princeton.edu/~appel/papers/certicoq-coqpl.pdf.

Anand, A., S. Boulier, C. Cohen, M. Sozeau, and N. Tabareau. 2018. "Towards Certified Meta-Programming with Typed Template-Coq". In: *Interactive Theorem Proving*. Cham: Springer International Publishing. 20–39. DOI: 10.1007/978-3-319-94821-8_2.

Anand, A. and V. Rahli. 2014. "Towards a Formally Verified Proof Assistant". In: *Interactive Theorem Proving*. Cham: Springer International Publishing. 27–44. DOI: 10.1007/978-3-319-08970-6_3.

Anderson, T. and M. Dahlin. 2014. *Operating Systems: Principles and Practice*. 2nd. Recursive books.

Andronick, J., R. Jeffery, G. Klein, R. Kolanski, M. Staples, H. Zhang, and L. Zhu. 2012. "Large-Scale Formal Verification in Practice: A Process Perspective". In: *International Conference on Software Engineering*. Zurich, Switzerland: ACM. 1002–1011. DOI: 10.1109/ICSE.2012.6227120.

Angiuli, C., E. Cavallo, K. Hou, R. Harper, and J. Sterling. 2018. "The RedPRL Proof Assistant (Invited Paper)". In: *Proceedings of the 13th International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, LFMTP@FSCD 2018, Oxford, UK, 7th July 2018*. 1–10. DOI: 10.4204/EPTCS.274.1.

Appel, A. W. 2006. "Tactics for separation logic". *INRIA Rocquencourt and Princeton University, Early Draft*. URL: https://www.cs.princeton.edu/~appel/papers/septacs.pdf.

Appel, A. W. 2017. Personal communication.

Appel, A. W. 2011a. "Efficient Verified Red-Black Trees". URL: https://www.cs.princeton.edu/~appel/papers/redblack.pdf.

Appel, A. W. 2011b. "Verified Software Toolchain". In: *Programming Languages and Systems: 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26–April 3, 2011. Proceedings*. Berlin, Heidelberg: Springer. 1–17. DOI: 10.1007/978-3-642-19718-5_1.

Appel, A. W., R. Dockins, A. Hobor, L. Beringer, J. Dodds, G. Stewart, S. Blazy, and X. Leroy. 2014. *Program Logics for Certified Compilers*. Cambridge University Press. DOI: 10.1017/CBO9781107256552.

Appel, A. W., N. G. Michael, A. Stump, and R. Virga. 2003. "A Trustworthy Proof Checker". *J. Autom. Reasoning*. 31(3-4): 231–260. DOI: 10.1023/B:JARS.0000021013.61329.58.

Appel, A. W., R. Dockins, and X. Leroy. 2012. "A list-machine benchmark for mechanized metatheory". *Journal of Automated Reasoning*. 49(3): 453–491. DOI: 10.1007/s10817-011-9226-1.

Aristotle. 1926. "Prior Analytics". In: *The Works of Aristotle*. Translated by A. J. Jenkinson. Oxford.

Armstrong, A., T. Bauereiss, B. Campbell, A. Reid, K. E. Gray, R. M. Norton, P. Mundkur, M. Wassell, J. French, C. Pulte, S. Flur, I. Stark, N. Krishnaswami, and P. Sewell. 2019. "ISA Semantics for ARMv8-a, RISC-v, and CHERI-MIPS". *Proc. ACM Program. Lang.* 3(POPL): 71:1–71:31. DOI: 10.1145/3290384.

Asperti, A., C. S. Coen, E. Tassi, and S. Zacchiroli. 2007. "User interaction with the Matita proof assistant". *Journal of Automated Reasoning*. 39(2): 109–139. DOI: 10.1007/s10817-007-9070-5.

Asperti, A., F. Guidi, C. S. Coen, E. Tassi, and S. Zacchiroli. 2004. "A content based mathematical search engine: Whelp". In: *International Workshop on Types for Proofs and Programs*. Springer. 17–32. DOI: 10.1007/11617990_2.

Aspinall, D. 2000a. "Isamode". URL: https://homepages.inf.ed.ac.uk/da/Isamode/doc/Isamode_toc.html.

Aspinall, D. 2000b. "Proof General: A Generic Tool for Proof Development". In: *Tools and Algorithms for the Construction and Analysis of Systems: 6th International Conference, TACAS 2000 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2000 Berlin, Germany, March 25 – April 2, 2000 Proceedings*. Berlin, Heidelberg: Springer. 38–43. DOI: 10.1007/3-540-46419-0_3.

Aspinall, D. and A. Compagnoni. 2001. "Subtyping dependent types". *Theoretical Computer Science*. 266(1): 273–309. DOI: 10.1016/S0304-3975(00)00175-4.

Aspinall, D. and C. Kaliszyk. 2016a. "Towards Formal Proof Metrics". In: *Fundamental Approaches to Software Engineering*. Berlin, Heidelberg: Springer. 325–341. DOI: 10.1007/978-3-662-49665-7_19.

Aspinall, D. and C. Kaliszyk. 2016b. "What's in a Theorem Name?" In: *Interactive Theorem Proving*. Cham: Springer International Publishing. 459–465. DOI: 10.1007/978-3-319-43144-4_28.

Avigad, J. 2004. "Proof Mining". URL: http://www.andrew.cmu.edu/user/avigad/Talks/asl04.pdf.

Aydemir, B. E., A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. 2005. "Mechanized Metatheory for the Masses: The POPLMark Challenge". In: *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics. TPHOLs'05*. Oxford, UK: Springer-Verlag. 50–65. DOI: 10.1007/11541868_4.

Aydemir, B. E., A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. 2005-2019. "The POPLMark Challenge". URL: http://www.seas.upenn.edu/~plclub/poplmark/.

Aydemir, B., A. Bohannon, and S. Weirich. 2006. "Nominal Reasoning Techniques in Coq". In: *International Workshop on Logical Frameworks and Meta-Languages:Theory and Practice (LFMTP)*. Seattle, WA, USA.

Aydemir, B., A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. 2008. "Engineering Formal Metatheory". In: *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '08*. San Francisco, CA, USA: ACM. 3–15. DOI: 10.1145/1328438.1328443.

Back, R. J. R. 1991. "Refinement Diagrams". In: *4th Refinement Workshop*. London: Springer London. 125–137. DOI: 10.1007/978-1-4471-3756-6_7.

Back, R.-J. 1988. "A calculus of refinements for program derivations". *Acta Informatica*. 25(6): 593–624. DOI: 10.1007/BF00291051.

Backus, J. 1978. "Can Programming Be Liberated from the von Neumann Style?: A Functional Style and Its Algebra of Programs". *Commun. ACM*. 21(8): 613–641. DOI: 10.1145/359576.359579.

Bahr, P. and G. Hutton. 2015. "Calculating correct compilers". *J. Funct. Program.* 25. DOI: 10.1017/S0956796815000180.

Ballarin, C. 2006. "Interpretation of Locales in Isabelle: Theories and Proof Contexts". In: *Mathematical Knowledge Management*. Berlin, Heidelberg: Springer. 31–43. DOI: 10.1007/11812289_4.

Bancerek, G. 1990. "The fundamental properties of natural numbers". *Formalized Mathematics*. 1(1): 41–46.

Bansal, K., S. M. Loos, M. N. Rabe, C. Szegedy, and S. Wilcox. 2019. "HOList: An Environment for Machine Learning of Higher-Order Theorem Proving (extended version)". *CoRR*. abs/1904.03241.

Barendregt, H. 2007. "Proofs of Correctness in Mathematics and Industry". In: *Wiley Encyclopedia of Computer Science and Engineering*. John Wiley & Sons. DOI: 10.1002/9780470050118.ecse579.

Barendregt, H. 2013. "Foundations of Mathematics from the Perspective of Computer Verification". In: *Mathematics, Computer Science and Logic - A Never Ending Story: The Bruno Buchberger Festschrift*. Cham: Springer International Publishing. 1–49. DOI: 10.1007/978-3-319-00966-7_1.

Barendregt, H. and E. Barendsen. 2002. "Autarkic Computations in Formal Proofs". *Journal of Automated Reasoning*. 28(3): 321–336. DOI: 10.1023/A:1015761529444.

Barendregt, H. and F. Wiedijk. 2005. "The challenge of computer mathematics". *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*. 363(1835): 2351–2375. DOI: 10.1098/rsta.2005.1650.

Barras, B. 2010. "Sets in Coq, Coq in Sets". *Journal of Formalized Reasoning*. 3(1): 29–48. DOI: 10.6092/issn.1972-5787/1695.

Barras, B., L. del Carmen González Huesca, H. Herbelin, Y. Régis-Gianas, E. Tassi, M. Wenzel, and B. Wolff. 2013. "Pervasive Parallelism in Highly-Trustable Interactive Theorem Proving Systems". In: *Intelligent Computer Mathematics: MKM, Calculemus, DML, and Systems and Projects 2013, Held as Part of CICM 2013, Bath, UK, July 8-12, 2013. Proceedings*. Berlin, Heidelberg: Springer. 359–363. DOI: 10.1007/978-3-642-39320-4_29.

Barras, B., C. Tankink, and E. Tassi. 2015. "Asynchronous Processing of Coq Documents: From the Kernel up to the User Interface". In: *Interactive Theorem Proving: 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*. Cham: Springer International Publishing. 51–66. DOI: 10.1007/978-3-319-22102-1_4.

Barras, B. and B. Werner. 1997. "Coq in Coq". URL: http://www.lix.polytechnique.fr/Labo/Bruno.Barras/publi/coqincoq.pdf.

Barret, C. 2018. "Twitter Response". URL: http://twitter.com/cbarrett/status/1059888137026068482.

Barthe, G. 1995. "Implicit coercions in type systems". In: *International Workshop on Types for Proofs and Programs*. Springer. 1–15. DOI: 10.1007/3-540-61780-9_58.

Barthe, G. and P. Courtieu. 2002. "Efficient reasoning about executable specifications in Coq". In: *International Conference on Theorem Proving in Higher Order Logics*. Springer. 31–46. DOI: 10.1007/3-540-45685-6_4.

Barthe, G. and O. Pons. 2001. "Type Isomorphisms and Proof Reuse in Dependent Type Theory". In: *Foundations of Software Science and Computation Structures*. Berlin, Heidelberg: Springer. 57–71. DOI: 10.1007/3-540-45315-6_4.

Barthe, G., M. Ruys, and H. Barendregt. 1996. "A two-level approach towards lean proof-checking". In: *Types for Proofs and Programs: International Workshop, TYPES '95 Torino, Italy, June 5–8, 1995 Selected Papers*. Berlin, Heidelberg: Springer. 16–35. DOI: 10.1007/3-540-61780-9_59.

Bates, J. L. 1979. "A Logic for Correct Program Development". AAI8003896. *PhD thesis*. Ithaca, NY, USA: Cornell University.

Bauer, A., J. Gross, P. L. Lumsdaine, M. Shulman, M. Sozeau, and B. Spitters. 2017. "The HoTT Library: A Formalization of Homotopy Type Theory in Coq". In: *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs. CPP 2017*. Paris, France: ACM. 164–172. DOI: 10.1145/3018610.3018615.

Beckert, B., S. Grebing, and F. Böhl. 2014. "A Usability Evaluation of Interactive Theorem Provers Using Focus Groups". In: *12th International Conference on Software Engineering and Formal Methods (SEFM 2014) – Collocated Workshops: Human-Oriented Formal Methods (HOFM 2014)*. Vol. 8938. *Lecture Notes in Computer Science*. Springer. 3–19. DOI: 10.1007/978-3-319-15201-1_1.

Benthem Jutting, L. S. van. 1994. "Checking Landau's Grundlagen in the Automath System: Parts of Chapters 0, 1 and 2 (Introduction, Prepration, Translation)". In: *Studies in Logic and the Foundations of Mathematics*. Vol. 133. Elsevier. 701–720. DOI: 10.1016/S0049-237X(08)70222-2.

Berger, U., S. Berghofer, P. Letouzey, and H. Schwichtenberg. 2005. "Program extraction from normalization proofs". *Studia Logica*. 82. DOI: 10.1007/s11225-006-6604-5.

Berghofer, S. and T. Nipkow. 2002. "Executing Higher Order Logic". In: *Types for Proofs and Programs: International Workshop, TYPES 2000 Durham, UK, December 8–12, 2000 Selected Papers*. Berlin, Heidelberg: Springer. 24–40. DOI: 10.1007/3-540-45842-5_2.

Berghofer, S. and C. Urban. 2007. "A Head-to-Head Comparison of de Bruijn Indices and Names". *Electronic Notes in Theoretical Computer Science*. 174(5): 53–67. DOI: 10.1016/j.entcs.2007.01.018.

Berghofer, S. and M. Wenzel. 1999. "Inductive Datatypes in HOL — Lessons Learned in Formal-Logic Engineering". In: *Theorem Proving in Higher Order Logics*. Berlin, Heidelberg: Springer. 19–36. DOI: 10.1007/3-540-48256-3_3.

Bertot, Y. 2009. "Theorem proving support in programming language semantics". In: *From Semantics to Computer Science: Essays in Honour of Gilles Kahn*. Ed. by Y. Bertot, G. Huet, J.-J. Lévy, and G. Plotkin. Cambridge University Press. 337–361. URL: http://hal.inria.fr/inria-00160309/.

Bertot, Y. and P. Casteran. 2004. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science An EATCS Series*. Berlin, Heidelberg: Springer. DOI: 10.1007/978-3-662-07964-5.

Bertot, Y., G. Kahn, and L. Théry. 1994. "Proof by pointing". In: *Theoretical Aspects of Computer Software*. Berlin, Heidelberg: Springer. 141–160. DOI: 10.1007/3-540-57887-0_94.

Bertot, Y. and L. Théry. 1998. "A Generic Approach to Building User Interfaces for Theorem Provers". *Journal of Symbolic Computation*. 25(2): 161–194. DOI: 10.1006/jsco.1997.0171.

Biendarra, J., J. C. Blanchette, A. Bouzy, M. Desharnais, M. Fleury, J. Hölzl, O. Kunčar, A. Lochbihler, F. Meier, L. Panny, A. Popescu, C. Sternagel, R. Thiemann, and D. Traytel. 2017. "Foundational (Co)datatypes and (Co)recursion for Higher-Order Logic". In: *Frontiers of Combining Systems*. Cham: Springer International Publishing. 3–21. DOI: 10.1007/978-3-319-66167-4_1.

Birkedal, L. and A. Bizjak. 2018. "Lecture Notes on Iris: Higher-Order Concurrent Separation Logic". URL: https://iris-project.org/tutorial-pdfs/iris-lecture-notes.pdf.

Bishop, E. and D. Bridges. 1985. *Constructive Analysis*. Springer. DOI: 10.1007/978-3-642-61667-9.

Bishop, S., M. Fairbairn, H. Mehnert, M. Norrish, T. Ridge, P. Sewell, M. Smith, and K. Wansbrough. 2018. "Engineering with Logic: Rigorous Test-Oracle Specification and Validation for TCP/IP and the Sockets API". *J. ACM*. 66(1): 1:1–1:77. DOI: 10.1145/3243650.

Bishop, S., M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough. 2006. "Engineering with Logic: HOL Specification and Symbolic-evaluation Testing for TCP Implementations". In: *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '06*. Charleston, SC, USA: ACM. 55–66. DOI: 10.1145/1111037.1111043.

Bjørner, D. and K. Havelund. 2014. "40 Years of Formal Methods". In: *FM 2014: Formal Methods: 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*. Cham: Springer International Publishing. 42–61. DOI: 10.1007/978-3-319-06410-9_4.

Blanchette, J. C., L. Bulwahn, and T. Nipkow. 2011. "Automatic Proof and Disproof in Isabelle/HOL". In: *Frontiers of Combining Systems*. Berlin, Heidelberg: Springer. 12–27. DOI: 10.1007/978-3-642-24364-6_2.

Blanchette, J. C., M. Fleury, P. Lammich, and C. Weidenbach. 2018. "A verified SAT solver framework with learn, forget, restart, and incrementality". *Journal of Automated Reasoning*. 61(1-4): 333–365.

Blanchette, J. C., D. Greenaway, C. Kaliszyk, D. Kühlwein, and J. Urban. 2016a. "A Learning-Based Fact Selector for Isabelle/HOL". *Journal of Automated Reasoning*. 57(3): 219–244. DOI: 10.1007/s10817-016-9362-8.

Blanchette, J. C., M. Haslbeck, D. Matichuk, and T. Nipkow. 2015. "Mining the Archive of Formal Proofs". In: *Intelligent Computer Mathematics: International Conference, CICM 2015, Washington, DC, USA, July 13-17, 2015, Proceedings*. Cham: Springer International Publishing. 3–17. DOI: 10.1007/978-3-319-20615-8_1.

Blanchette, J. C., C. Kaliszyk, L. C. Paulson, and J. Urban. 2016b. "Hammering towards QED". *Journal of Formalized Reasoning*. 9(1): 101–148. DOI: 10.6092/issn.1972-5787/4593.

Blanchette, J. C. and T. Nipkow. 2010. "Nitpick: A Counterexample Generator for Higher-order Logic Based on a Relational Model Finder". In: *Proceedings of the First International Conference on Interactive Theorem Proving. ITP'10*. Edinburgh, UK: Springer-Verlag. 131–146. DOI: 10.1007/978-3-642-14052-5_11.

Boite, O. 2004. "Proof Reuse with Extended Inductive Types". In: *Theorem Proving in Higher Order Logics: 17th International Conference, TPHOLs 2004, Park City, Utah, USA, September 14-17, 2004. Proceedings*. Berlin, Heidelberg: Springer. 50–65. DOI: 10.1007/978-3-540-30142-4_4.

Boulton, R. J., A. Gordon, M. J. C. Gordon, J. Harrison, J. Herbert, and J. V. Tassel. 1992. "Experience with Embedding Hardware Description Languages in HOL". In: *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*. Amsterdam, The Netherlands: North-Holland Publishing Co. 129–156.

Bounov, D., A. DeRossi, M. Menarini, W. G. Griswold, and S. Lerner. 2018. "Inferring Loop Invariants through Gamification". In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM. 231. DOI: 10.1145/3173574.3173805.

Bourke, T., M. Daum, G. Klein, and R. Kolanski. 2012. "Challenges and Experiences in Managing Large-Scale Proofs". In: *Conferences on Intelligent Computer Mathematics (CICM) / Mathematical Knowledge Management*. Bremen, Germany: Springer. 32–48. DOI: 10.1007/978-3-642-31374-5_3.

Boutin, S. 1997. "Using reflection to build efficient and certified decision procedures". In: *Theoretical Aspects of Computer Software: Third International Symposium, TACS'97 Sendai, Japan, September 23–26, 1997 Proceedings*. Berlin, Heidelberg: Springer. 515–529. DOI: 10.1007/BFb0014565.

Boyer, R. S. 1994. "A mechanically proof-checked encyclopedia of mathematics: Should we build one? Can we?" In: *Automated Deduction — CADE-12: 12th International Conference on Automated Deduction Nancy, France, June 26 – July 1, 1994 Proceedings*. Berlin, Heidelberg: Springer. 237–251. DOI: 10.1007/3-540-58156-1_17.

Bradley, A. R. and Z. Manna. 2007. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Berlin, Heidelberg: Springer-Verlag.

Brady, E. 2013. "Idris, a general-purpose dependently typed programming language: Design and implementation". *Journal of Functional Programming*. 23(5): 552–593. DOI: 10.1017/S095679681300018X.

Braibant, T. and D. Pous. 2011. "Tactics for Reasoning Modulo AC in Coq". In: *Certified Programs and Proofs: First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings*. Berlin, Heidelberg: Springer. 167–182. DOI: 10.1007/978-3-642-25379-9_14.

Braibant, T. and D. Pous. 2012. "Deciding Kleene Algebras in Coq". *Logical Methods in Computer Science*. Volume 8, Issue 1. DOI: 10.2168/LMCS-8(1:16)2012.

Buchberger, B. 2000. "Theory exploration with Theorema". *Analele Universitatii Din Timisoara, ser. Matematica-Informatica*. 38(2): 9–32.

Buchberger, B., A. Craciun, T. Jebelean, L. Kovács, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkranz, *et al.* 2006. "Theorema: Towards computer-aided mathematical theory exploration". *Journal of Applied Logic*. 4(4): 470–504. DOI: 10.1016/j.jal.2005.10.006.

Bulwahn, L. 2012. "The New Quickcheck for Isabelle: Random, Exhaustive and Symbolic Testing Under One Roof". In: *Proceedings of the Second International Conference on Certified Programs and Proofs. CPP'12*. Kyoto, Japan: Springer-Verlag. 92–108. DOI: 10.1007/978-3-642-35308-6_10.

Bundy, A. 1988. "The Use of Explicit Plans to Guide Inductive Proofs". In: *Proceedings of the 9th International Conference on Automated Deduction*. Berlin, Heidelberg: Springer. 111–120. DOI: 10.1007/BFb0012826.

Burstall, R. M. 1969. "Proving Properties of Programs by Structural Induction". *The Computer Journal*. 12(1): 41–48. DOI: 10.1093/comjnl/12.1.41.

Busch, H. 1994. "First-order automation for higher-order-logic theorem proving". In: *Higher Order Logic Theorem Proving and Its Applications*. Springer. 97–112. DOI: 10.1007/3-540-58450-1_37.

Cachin, C., R. Guerraoui, and L. Rodrigues. 2011. *Introduction to Reliable and Secure Distributed Programming*. 2nd. Springer. DOI: 10.1007/978-3-642-15260-3.

Callaghan, P. and Z. Luo. 2001. "An implementation of LF with coercive subtyping & universes". *Journal of Automated Reasoning*. 27(1): 3–27. DOI: 10.1023/A:1010648911114.

Cao, J., M. Fu, and X. Feng. 2015. "Practical tactics for verifying C programs in Coq". In: *Proceedings of the 2015 Conference on Certified Programs and Proofs*. ACM. 97–108. DOI: 10.1145/2676724.2693162.

Cao, Q., L. Beringer, S. Gruetter, J. Dodds, and A. W. Appel. 2018. "VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs". *Journal of Automated Reasoning*. 61(1): 367–422. DOI: 10.1007/s10817-018-9457-5.

Caplan, J. E. and M. T. Harandi. 1995. "A Logical Framework for Software Proof Reuse". *SIGSOFT Softw. Eng. Notes.* 20(SI): 106–113. DOI: 10.1145/223427.211821.

Capretta, V. 2005. "General Recursion via Coinductive Types". *Logical Methods in Computer Science.* Volume 1, Issue 2(July). DOI: 10.2168/LMCS-1(2:1)2005.

Capretta, V. and A. P. Felty. 2007. "Combining de Bruijn Indices and Higher-Order Abstract Syntax in Coq". In: *Types for Proofs and Programs.* Berlin, Heidelberg: Springer. 63–77. DOI: 10.1007/978-3-540-74464-1_5.

Cardelli, L., S. Martini, J. C. Mitchell, and A. Scedrov. 1994. "An Extension of System F with Subtyping". *Inf. Comput.* 109(1-2): 4–56. DOI: 10.1006/inco.1994.1013.

Casinghino, C., V. Sjöberg, and S. Weirich. 2014. "Combining proofs and programs in a dependently typed language". In: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014.* 33–46. DOI: 10.1145/2535838.2535883.

Celik, A., K. Palmskog, and M. Gligoric. 2017. "iCoq: Regression Proof Selection for Large-Scale Verification Projects". In: *32nd IEEE/ACM International Conference on Automated Software Engineering. ASE 2017.* 171–182. DOI: 10.1109/ASE.2017.8115630.

Chaieb, A. and T. Nipkow. 2008. "Proof synthesis and reflection for linear arithmetic". *Journal of Automated Reasoning.* 41(1): 33.

Chajed, T., J. Tassarotti, M. F. Kaashoek, and N. Zeldovich. 2019. "Argosy: Verifying Layered Storage Systems with Recovery Refinement". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI 2019.* Phoenix, AZ, USA: ACM. 1054–1068. DOI: 10.1145/3314221.3314585.

Chan, M., J. Lehmann, and A. Bundy. 2011. "GALILEO: A system for automating ontology evolution". *ARCOE-11*: 46.

Charguéraud, A. 2010. "Program Verification Through Characteristic Formulae". In: *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming. ICFP '10.* Baltimore, MD, USA: ACM. 321–332. DOI: 10.1145/1863543.1863590.

Charguéraud, A. 2011. "The Locally Nameless Representation". *Journal of Automated Reasoning*: 1–46. DOI: 10.1007/s10817-011-9225-2.

Chen, H., T. Chajed, A. Konradi, S. Wang, A. İleri, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. 2017. "Verifying a High-performance Crash-safe File System Using a Tree Specification". In: *Proceedings of the Symposium on Operating Systems Principles*. 270–286. DOI: 10.1145/3132747.3132776.

Chen, H., D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. 2015. "Using Crash Hoare Logic for Certifying the FSCQ File System". In: *Proceedings of the 25th Symposium on Operating Systems Principles. SOSP '15*. Monterey, California: ACM. 18–37. DOI: 10.1145/2815400.2815402.

Chlipala, A. 2008. "Parametric Higher-order Abstract Syntax for Mechanized Semantics". In: *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming. ICFP '08*. Victoria, BC, Canada: ACM. 143–156. DOI: 10.1145/1411204.1411226.

Chlipala, A. 2011. "Mostly-automated verification of low-level programs in computational separation logic". In: *PLDI*. ACM. 234–245. DOI: 10.1145/1993498.1993526.

Chlipala, A. 2013a. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. Cambridge, MA, USA: MIT Press.

Chlipala, A. 2013b. "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. ICFP '13*. Boston, MA, USA: ACM. 391–402. DOI: 10.1145/2500365.2500592.

Chlipala, A. 2017. "Formal Reasoning About Programs". URL: http://adam.chlipala.net/frap/.

Chlipala, A. 2018. "The Surprising Security Benefits of End-to-End Formal Proofs". URL: https://www.cccblog.org/2018/06/13/the-surprising-security-benefits-of-end-to-end-formal-proofs/.

Chlipala, A., B. Delaware, S. Duchovni, J. Gross, C. Pit-Claudel, S. Suriyakarn, P. Wang, and K. Ye. 2017. "The End of History: Using a Proof Assistant to Replace Language Design with Library Design". In: *2nd Summit on Advances in Programming Languages (SNAPL 2017)*. Vol. 71. *Leibniz International Proceedings in Informatics (LIPIcs)*. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. 3:1–3:15. DOI: 10.4230/LIPIcs.SNAPL.2017.3.

Chlipala, A., J. G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. 2009. "Effective interactive proofs for higher-order imperative programs". In: *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming (ICFP 2009)*. ACM. 79–90. DOI: 10.1145/1596550.1596565.

Christiansen, D. and E. Brady. 2016. "Elaborator Reflection: Extending Idris in Idris". *SIGPLAN Not.* 51(9): 284–297. DOI: 10.1145/3022670.2951932.

Chrząszcz, J. 2003. "Implementing Modules in the Coq System". In: *Theorem Proving in Higher Order Logics*. Berlin, Heidelberg: Springer. 270–286. DOI: 10.1007/10930755_18.

Church, A. 1936. "An Unsolvable Problem of Elementary Number Theory". *American Journal of Mathematics*. 58(2): 345–363. DOI: 10.2307/2371054.

Church, A. 1940. "A formulation of the simple theory of types". *Journal of Symbolic Logic*. 5(2): 56–68. DOI: 10.2307/2266170.

Church, A. 1941. *The calculi of lambda-conversion*. Princeton University Press.

Ciaffaglione, A. and I. Scagnetto. 2012. "A weak HOAS approach to the POPLMark Challenge". In: *Proceedings Seventh Workshop on Logical and Semantic Frameworks, with Applications, LSFA 2012, Rio de Janeiro, Brazil, September 29-30, 2012*. 109–124. DOI: 10.4204/EPTCS.113.11.

Claret, G., L. d. C. G. Huesca, Y. Régis-Gianas, and B. Ziliani. 2013. "Lightweight proof by reflection using a posteriori simulation of effectful computation". In: *International Conference on Interactive Theorem Proving*. Springer. 67–83. DOI: 10.1007/978-3-642-39634-2_8.

cody. 2015. "Why does Coq have Prop?" Theoretical Computer Science Stack Exchange. URL: http://cstheory.stackexchange.com/questions/21836/why-does-coq-have-prop.

Cohen, C. 2013. "Pragmatic Quotient Types in Coq". In: *Interactive Theorem Proving*. Berlin, Heidelberg: Springer. 213–228. DOI: 10.1007/978-3-642-39634-2_17.

Cohen, C., T. Coquand, S. Huber, and A. Mörtberg. 2018. "Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom". In: *21st International Conference on Types for Proofs and Programs (TYPES 2015)*. Vol. 69. Schloss Dagstuhl. 5:1–5:34. DOI: 10.4230/LIPIcs.TYPES.2015.5.

Cohen, C., M. Dénès, and A. Mörtberg. 2013. "Refinements for Free!" In: *Certified Programs and Proofs*. Cham: Springer International Publishing. 147–162. DOI: 10.1007/978-3-319-03545-1_10.

Cohn, A. 1983. "The Equivalence of Two Semantic Definitions: A Case Study in LCF". *SIAM Journal on Computing*. 12(2): 267–285. DOI: 10.1137/0212016.

CompCert Development Team. 2010. "Merge of the newmem and newextcalls branches". URL: http://github.com/AbsInt/CompCert/commit/a74f6b45d72834b5b8417297017bd81424123d98.

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

Cooper, K. and L. Torczon. 2011. *Engineering a compiler*. Elsevier. DOI: 10.1016/C2009-0-27982-7.

Coq Development Team. 2003. "interface GTK2 experimentale". URL: http://github.com/coq/coq/commit/3fe1d8e4287.

Coq Development Team. 2017. "CoqStyle". URL: https://github.com/coq/coq/wiki/CoqStyle.

Coq Development Team. 2017-2019. "CoqInTheClassroom". URL: http://github.com/coq/coq/wiki/CoqInTheClassroom.

Coq Development Team. 2018-2019. "Preliminary compilation of critical bugs in stable releases of Coq". URL: http://github.com/coq/coq/blob/master/dev/doc/critical-bugs.

Coq Development Team. 1999-2018a. "Coq Integrated Development Environment". URL: http://coq.inria.fr/refman/practical-tools/coqide.html.

Coq Development Team. 1999-2018b. "Tactics". URL: http://coq.inria.fr/refman/proof-engine/tactics.html.

Coq Development Team. 1999-2018c. "The Coq Commands". URL: http://coq.inria.fr/refman/practical-tools/coq-commands.html.

Coq Development Team. 1989-2019. "The Coq Proof Assistant". URL: http://coq.inria.fr.

Coq development team. 2018. "Coq OPAM Package Index". URL: https://coq.inria.fr/opam/www/.

CoqHoTT Development Team. 2015-2019. "The CoqHoTT Project". URL: http://coqhott.gforge.inria.fr/.

Coquand, T. 1994. "Infinite objects in type theory". In: *Types for Proofs and Programs*. Berlin, Heidelberg: Springer. 62–78. DOI: 10.1007/3-540-58085-9_72.

Coquand, T. and G. Huet. 1985. "Constructions: A higher order proof system for mechanizing mathematics". In: *EUROCAL '85*. Berlin, Heidelberg: Springer. 151–184. DOI: 10.1007/3-540-15983-5_13.

Coquand, T. and G. Huet. 1988. "The calculus of constructions". *Information and Computation*. 76(2): 95–120. DOI: 10.1016/0890-5401(88)90005-3.

Coquand, T. and C. Paulin-Mohring. 1990. "Inductively defined types". In: *COLOG-88*. Berlin, Heidelberg: Springer. 50–66. DOI: 10.1007/3-540-52335-9_47.

Corbineau, P. 2008. "A Declarative Language for the Coq Proof Assistant". In: *Types for Proofs and Programs: International Conference, TYPES 2007, Cividale des Friuli, Italy, May 2-5, 2007 Revised Selected Papers*. Berlin, Heidelberg: Springer. 69–84. DOI: 10.1007/978-3-540-68103-8_5.

Cornes, C., J. Courant, J.-C. Filliâtre, G. Huet, P. Manoury, C. Munoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saibi, *et al.* 1995. "The Coq Proof assistant, Reference Manual, Version 5.10".

Cornes, C. and D. Terrasse. 1995. "Automating inversion of inductive predicates in Coq". In: *International Workshop on Types for Proofs and Programs*. Springer. 85–104. DOI: 10.1007/3-540-61780-9_64.

Crary, K. and R. Harper. 2014. "The Mechanization of Standard ML". URL: http://github.com/SMLFamily/The-Mechanization-of-Standard-ML.

Crégut, P. 1999-2018. "Omega: A solver for quantifier-free problems in Presburger Arithmetic". URL: http://coq.inria.fr/refman/addendum/omega.html.

Cruz-Filipe, L. and P. Letouzey. 2006. "A Large-Scale Experiment in Executing Extracted Programs". *Electronic Notes in Theoretical Computer Science*. 151(1): 75–91. DOI: 10.1016/j.entcs.2005.11.024.

Cruz-Filipe, L. and B. Spitters. 2003. "Program Extraction from Large Proof Developments". In: *Theorem Proving in Higher Order Logics: 16th International Conference, TPHOLs 2003, Rome, Italy, September 8-12, 2003. Proceedings*. Berlin, Heidelberg: Springer. 205–220. DOI: 10.1007/10930755_14.

Curien, R. 1995. "Tools for proof by analogy". *Theses*. Université Henri Poincaré - Nancy 1. URL: https://hal.univ-lorraine.fr/tel-01748604.

Curry, H. B. 1934. "Functionality in Combinatory Logic". *Proceedings of the National Academy of Sciences of the United States of America*. 20(11): 584–590. URL: http://www.jstor.org/stable/86796.

Czajka, Ł. and C. Kaliszyk. 2018. "Hammer for Coq: Automation for Dependent Type Theory". *Journal of Automated Reasoning*. 61(1): 423–453. DOI: 10.1007/s10817-018-9458-4.

Czajka, Ł., C. Kaliszyk, and B. Ekici. 2018. "CoqHammer: Automation for Dependent Type Theory". URL: http://cl-informatik.uibk.ac.at/cek/coqhammer/.

Dagand, P. 2017. "The essence of ornaments". *J. Funct. Program.* 27: e9. DOI: 10.1017/S0956796816000356.

Dahn, B. I., J. Gehne, T. Honigmann, and A. Wolf. 1997. "Integration of automated and interactive theorem proving in ILF". In: *International Conference on Automated Deduction*. Springer. 57–60. DOI: 10.1007/3-540-63104-6_7.

Dam, M., R. Guanciale, N. Khakpour, H. Nemati, and O. Schwarz. 2013. "Formal Verification of Information Flow Security for a Simple ARM-based Separation Kernel". In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security. CCS '13*. Berlin, Germany: ACM. 223–234. DOI: 10.1145/2508859.2516702.

Danielsson, N. A. 2012. "Operational Semantics Using the Partiality Monad". In: *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming. ICFP '12*. Copenhagen, Denmark: ACM. 127–138. DOI: 10.1145/2364527.2364546.

Darmochwał, A. 1990. "Finite sets". *Formalized Mathematics*. 1(1): 165–167.

Davis, J. and M. O. Myreen. 2015. "The Reflective Milawa Theorem Prover is Sound (Down to the Machine Code that Runs it)". *Journal of Automated Reasoning*. 55(2): 117–183. DOI: 10.1007/s10817-015-9324-6.

de Bruijn, N. G. 1970. "The mathematical language Automath, its usage, and some of its extensions". In: *Symposium on Automatic Demonstration*. Berlin, Heidelberg: Springer. 29–61. DOI: 10.1007/BFb0060623.

de Bruijn, N. G. 1972. "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem". *Indagationes Mathematicae (Proceedings)*. 75(5): 381–392. DOI: 10.1016/1385-7258(72)90034-0.

de Bruijn, N. G. 1994. "A Survey of the Project Automath". In: *Selected Papers on Automath*. Ed. by R. Nederpelt, J. Geuvers, and R. de Vrijer. Vol. 133. *Studies in Logic and the Foundations of Mathematics*. Elsevier. 141–161. DOI: 10.1016/S0049-237X(08)70203-9.

de Moura, L., S. Kong, J. Avigad, F. van Doorn, and J. von Raumer. 2015. "The Lean Theorem Prover (System Description)". In: *Automated Deduction - CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*. Cham: Springer International Publishing. 378–388. DOI: 10.1007/978-3-319-21401-6_26.

De Roever, W.-P., K. Engelhardt, and K.-H. Buth. 1998. *Data refinement: model-oriented proof methods and their comparison*. Vol. 47. Cambridge University Press.

DeepSpec Team. 2013-2019. "DeepSpec Project". URL: https://deepspec.org.

Delahaye, D. 2000. "A Tactic Language for the System Coq". In: *Logic for Programming and Automated Reasoning: 7th International Conference, LPAR 2000 Reunion Island, France, November 6–10, 2000 Proceedings*. Berlin, Heidelberg: Springer. 85–95. DOI: 10.1007/3-540-44404-1_7.

Delaware, B., W. Cook, and D. Batory. 2011. "Product Lines of Theorems". In: *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications. OOPSLA '11*. Portland, OR, USA: ACM. 595–608. DOI: 10.1145/2048066.2048113.

Delaware, B., S. Keuchel, T. Schrijvers, and B. C. Oliveira. 2013a. "Modular Monadic Meta-theory". In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming. ICFP '13*. Boston, MA, USA: ACM. 319–330. DOI: 10.1145/2500365.2500587.

Delaware, B., C. Pit-Claudel, J. Gross, and A. Chlipala. 2015. "Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant". In: *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '15*. Mumbai, India: ACM. 689–700. DOI: 10.1145/2676726.2677006.

Delaware, B., B. C. d. S. Oliveira, and T. Schrijvers. 2013b. "Meta-theory à La Carte". In: *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '13*. Rome, Italy: ACM. 207–218. DOI: 10.1145/2429069.2429094.

Demers, F.-N. and J. Malenfant. 1995. "Reflection in logic, functional and object-oriented programming: A Short Comparative Study". In: *In IJCAI '95 Workshop on Reflection and Metalevel Architectures and their Applications in AI*. 29–38.

DeMillo, R. A., R. J. Lipton, and A. J. Perlis. 1977. "Social Processes and Proofs of Theorems and Programs". In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. POPL '77*. Los Angeles, California: ACM. 206–214. DOI: 10.1145/512950.512970.

Devriese, D. and F. Piessens. 2011. "On the Bright Side of Type Classes: Instance Arguments in Agda". In: *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming. ICFP '11*. Tokyo, Japan: ACM. 143–155. DOI: 10.1145/2034773.2034796.

Diehl, L., D. Firsov, and A. Stump. 2018. "Generic Zero-cost Reuse for Dependent Types". *Proc. ACM Program. Lang.* 2(ICFP): 104:1–104:30. DOI: 10.1145/3236799.

Dietl, W., S. Dietzel, M. D. Ernst, N. Mote, B. Walker, S. Cooper, T. Pavlik, and Z. Popović. 2012. "Verification games: Making verification fun". In: *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*. ACM. 42–49. DOI: 10.1145/2318202.2318210.

Dietrich, D. 2011. *Assertion level proof planning with compiled strategies*.

Dietrich, D., I. Whiteside, and D. Aspinall. 2013. "Polar: A Framework for Proof Refactoring". In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Berlin, Heidelberg: Springer. 776–791. DOI: 10.1007/978-3-642-45221-5_52.

Dijkstra, E. W. 1975. "Guarded Commands, Nondeterminacy and Formal Derivation of Programs". *Commun. ACM*. 18(8): 453–457. DOI: 10.1145/360933.360975.

Dixon, L. and J. Fleuriot. 2003. "IsaPlanner: A Prototype Proof Planner in Isabelle". In: *Automated Deduction – CADE-19*. Berlin, Heidelberg: Springer. 279–283. DOI: 10.1007/978-3-540-45085-6_22.

Dockins, R. and A. Hobor. 2010. "A theory of termination via indirection". In: *Proceedings of the Dagstuhl Seminar on Modelling, Controlling and Reasoning about State*. Vol. 10351. Dagstuhl, Germany. 166–177.

Doczkal, C. and G. Smolka. 2018. "Regular Language Representations in the Constructive Type Theory of Coq". *Journal of Automated Reasoning*. 61(1): 521–553. DOI: 10.1007/s10817-018-9460-x.

Dramnesc, I., T. Jebelean, and S. Stratulat. 2015. "Theory exploration of binary trees". In: *Intelligent Systems and Informatics (SISY), 2015 IEEE 13th International Symposium on*. IEEE. 139–144. DOI: 10.1109/SISY.2015.7325367.

Dreyer, D., R. Harper, M. M. T. Chakravarty, and G. Keller. 2007. "Modular Type Classes". In: *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '07*. Nice, France: ACM. 63–70. DOI: 10.1145/1190216.1190229.

Ebner, G., S. Ullrich, J. Roesch, J. Avigad, and L. de Moura. 2017. "A Metaprogramming Framework for Formal Verification". *Proc. ACM Program. Lang.* 1(ICFP): 34:1–34:29. DOI: 10.1145/3110278.

Eclipse Foundation. 2001-2019. "Eclipse". URL: http://www.eclipse.org/ide/.

Elphinstone, K. and G. Heiser. 2013. "From L3 to seL4 What Have We Learnt in 20 Years of L4 Microkernels?" In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. SOSP '13*. Farminton, PA: ACM. 133–150. DOI: 10.1145/2517349.2522720.

Erbsen, A., J. Philipoom, J. Gross, R. Sloan, and A. Chlipala. 2019. "Simple High-Level Code For Cryptographic Arithmetic – With Proofs, Without Compromises". In: *IEEE Symposium on Security and Privacy*. DOI: 10.1109/SP.2019.00005.

Escardó, M. H. 2018. "A self-contained, brief and complete formulation of Voevodsky's Univalence Axiom". *CoRR*. abs/1803.02294.

Esparza, J., P. Lammich, R. Neumann, T. Nipkow, A. Schimpf, and J.-G. Smaus. 2013. "A fully verified executable LTL model checker". In: *International Conference on Computer Aided Verification*. Springer. 463–478.

Faithfull, A., J. Bengtson, E. Tassi, and C. Tankink. 2018. "Coqoon: An IDE for interactive proof development in Coq". *International Journal on Software Tools for Technology Transfer*. 20(2): 125–137. DOI: 10.1007/s10009-017-0457-2.

Fallenstein, B. and R. Kumar. 2015. "Proof-producing reflection for HOL". In: *International Conference on Interactive Theorem Proving*. Springer. 170–186. DOI: 10.1007/978-3-319-22102-1_11.

Feferman, S. 2005. "Predicativity". In: *Oxford Handbook of Philosophy of Mathematics and Logic*. Ed. by S. Shapiro. Oxford University Press. 590–624.

Felty, A. P., A. Momigliano, and B. Pientka. 2015. "The Next 700 Challenge Problems for Reasoning with Higher-Order Abstract Syntax Representations: Part 1-A Common Infrastructure for Benchmarks". *CoRR*. abs/1503.06095.

Felty, A. and D. Howe. 1994. "Generalization and reuse of tactic proofs". In: *Logic Programming and Automated Reasoning: 5th International Conference. LPAR '94*. Berlin, Heidelberg: Springer. 1–15. DOI: 10.1007/3-540-58216-9_25.

Felty, A. and A. Momigliano. 2012. "Hybrid: A Definitional Two-Level Approach to Reasoning with Higher-Order Abstract Syntax". *Journal of Automated Reasoning*. 48(1): 43–105. DOI: 10.1007/s10817-010-9194-x.

Felty, A., A. Momigliano, and B. Pientka. 2018. "Benchmarks for reasoning with syntax trees containing binders and contexts of assumptions". *Mathematical Structures in Computer Science*. 28(9): 1507–1540. DOI: 10.1017/S0960129517000093.

Felty, A. and B. Pientka. 2010. "Reasoning with Higher-Order Abstract Syntax and Contexts: A Comparison". In: *Interactive Theorem Proving*. Berlin, Heidelberg: Springer. 227–242. DOI: 10.1007/978-3-642-14052-5_17.

Feng, X. 2009. "Local rely-guarantee reasoning". In: *POPL*. ACM. 315–327. DOI: 10.1145/1480881.1480922.

Feng, X. and Z. Shao. 2005. "Modular verification of concurrent assembly code with dynamic thread creation and termination". In: *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming (ICFP 2005)*. ACM. 254–267. DOI: 10.1145/1086365.1086399.

Filliâtre, J.-C. and P. Letouzey. 2004. "Functors for Proofs and Programs". In: *Programming Languages and Systems*. Berlin, Heidelberg: Springer. 370–384. DOI: 10.1007/978-3-540-24725-8_26.

Filman, R., T. Elrad, S. Clarke, and M. Akşit. 2004. *Aspect-oriented Software Development*. First. Addison-Wesley Professional.

Floyd, R. W. 1967. "Assigning Meanings to Programs". *Proceedings of Symposium on Applied Mathematics*. 19.

Fox, A. and M. O. Myreen. 2010. "A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture". In: *Interactive Theorem Proving*. Berlin, Heidelberg: Springer. 243–258. DOI: 10.1007/978-3-642-14052-5_18.

França, R. B., D. Favre-Felix, X. Leroy, M. Pantel, and J. Souyris. 2011. "Towards Formally Verified Optimizing Compilation in Flight Control Software". In: *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*. Vol. 18. *OpenAccess Series in Informatics (OASIcs)*. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. 59–68. DOI: 10.4230/OASIcs.PPES.2011.59.

Frege, G. 1893. *Grundgesetze der Arithmetik*. Jena: Verlag Hermann Pohle.

Gallego Arias, E. J., B. Pin, and P. Jouvelot. 2017. "jsCoq: Towards Hybrid Theorem Proving Interfaces". In: *Proceedings of the 12th Workshop on User Interfaces for Theorem Provers, Coimbra, Portugal, 2nd July 2016*. Vol. 239. *Electronic Proceedings in Theoretical Computer Science*. Open Publishing Association. 15–27. DOI: 10.4204/EPTCS.239.2.

Garillot, F. 2011. "Generic Proof Tools and Finite Group Theory". *PhD thesis*. Palaiseau, France: École Polytechnique.

Garillot, F., G. Gonthier, A. Mahboubi, and L. Rideau. 2009. "Packaging Mathematical Structures". In: *TPHOL*. Vol. 5674. *LNCS*. Springer. 327–342. DOI: 10.1007/978-3-642-03359-9_23.

Gauthier, T. and C. Kaliszyk. 2014. "Matching concepts across HOL libraries". In: *Intelligent Computer Mathematics*. Springer. 267–281. DOI: 10.1007/978-3-319-08434-3_20.

Gauthier, T. and C. Kaliszyk. 2015. "Sharing HOL4 and HOL Light Proof Knowledge". In: *Logic for Programming, Artificial Intelligence, and Reasoning: 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*. Berlin, Heidelberg: Springer. 372–386. DOI: 10.1007/978-3-662-48899-7_26.

Gauthier, T. and C. Kaliszyk. 2019. "Aligning concepts across proof assistant libraries". *Journal of Symbolic Computation*. 90: 89–123. Symbolic Computation in Software Science. DOI: 10.1016/j.jsc.2018.04.005.

Gauthier, T., C. Kaliszyk, and J. Urban. 2017. "TacticToe: Learning to Reason with HOL4 Tactics". In: *LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning*. Vol. 46. *EPiC Series in Computing*. EasyChair. 125–143. DOI: 10.29007/ntlb.

Gauthier, T., C. Kaliszyk, J. Urban, R. Kumar, and M. Norrish. 2018. "Learning to Prove with Tactics". *CoRR*. abs/1804.00596. arXiv: 1804.00596.

Geuvers, H. 2009. "Proof assistants: History, ideas and future". *Sadhana*. 34(1): 3–25. DOI: 10.1007/s12046-009-0001-5.

Geuvers, H., R. Pollack, F. Wiedijk, and J. Zwanenburg. 2002a. "A Constructive Algebraic Hierarchy in Coq". *Journal of Symbolic Computation*. 34(4): 271–286. DOI: 10.1006/jsco.2002.0552.

Geuvers, H., F. Wiedijk, and J. Zwanenburg. 2002b. "A Constructive Proof of the Fundamental Theorem of Algebra Without Using the Rationals". In: *Types for Proofs and Programs. TYPES '00*. Berlin, Heidelberg: Springer. 96–111. DOI: 10.1007/3-540-45842-5_7.

Giero, M., F. Wiedijk, M. Giero, and F. Wiedijk. 2003. "MMode, a Mizar mode for the proof assistant Coq". URL: http://www.cs.kun.nl/~freek/mmode/mmode.pdf.

Giménez, E. 1995. "Codifying guarded definitions with recursive schemes". In: *Types for Proofs and Programs*. Berlin, Heidelberg: Springer. 39–59. DOI: 10.1007/3-540-60579-7_3.

GitHub. 2014-2019. "Atom". URL: http://atom.io/.

GNU Project. 1985-2019. "GNU Emacs". URL: http://www.gnu.org/software/emacs/.

Gödel, K. 1930. "Die Vollständigkeit der Axiome des logischen Funktionenkalküls". *Monatshefte für Mathematik und Physik*. 37(1): 349–360. DOI: 10.1007/BF01696781.

Gödel, K. 1931. "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I". *Monatshefte für Mathematik und Physik*. 38(1): 173–198. DOI: 10.1007/BF01700692.

Gomes, V. B., M. Kleppmann, D. P. Mulligan, and A. R. Beresford. 2017. "Verifying strong eventual consistency in distributed systems". *Proceedings of the ACM on Programming Languages*. 1(OOPSLA): 109. DOI: 10.1145/3133933.

Gonthier, G. 2008. "Formal proof—the four-color theorem". *Notices of the American Mathematical Society*. 55(11): 1382–1393. URL: http://www.ams.org/notices/200811/tx081101382p.pdf.

Gonthier, G., A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. Le Roux, A. Mahboubi, R. O'Connor, S. Ould Biha, I. Pasca, L. Rideau, A. Solovyev, E. Tassi, and L. Théry. 2013. "A Machine-Checked Proof of the Odd Order Theorem". In: *Interactive Theorem Proving: 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*. Berlin, Heidelberg: Springer. 163–179. DOI: 10.1007/978-3-642-39634-2_14.

Gonthier, G. and A. Mahboubi. 2010. "An introduction to small scale reflection in Coq". *Journal of Formalized Reasoning*. 3(2): 95–152. DOI: 10.6092/issn.1972-5787/1979.

Gonthier, G. and E. Tassi. 2012. "A Language of Patterns for Subterm Selection". In: *Interactive Theorem Proving*. Berlin, Heidelberg: Springer. 361–376. DOI: 10.1007/978-3-642-32347-8_25.

Gonthier, G., B. Ziliani, A. Nanevski, and D. Dreyer. 2011. "How to Make Ad Hoc Proof Automation Less Ad Hoc". In: *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming. ICFP '11*. Tokyo, Japan: ACM. 163–175. DOI: 10.1145/2034773.2034798.

Gordon, A. D. 1994. "A mechanisation of name-carrying syntax up to alpha-conversion". In: *Higher Order Logic Theorem Proving and Its Applications*. Berlin, Heidelberg: Springer. 413–425. DOI: 10.1007/3-540-57826-9_152.

Gordon, C. S., M. D. Ernst, D. Grossman, and M. J. Parkinson. 2017. "Verifying Invariants of Lock-Free Data Structures with Rely-Guarantee and Refinement Types". *ACM Trans. Program. Lang. Syst.* 39(3): 11:1–11:54. DOI: 10.1145/3064850.

Gordon, M. J. C. 2000. "Proof, Language, and Interaction". In: ed. by G. Plotkin, C. Stirling, and M. Tofte. Cambridge, MA, USA: MIT Press. Chap. From LCF to HOL: A Short History. 169–185.

Gordon, M. J. C. and T. F. Melham, eds. 1993. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. New York, NY, USA: Cambridge University Press.

Gordon, M. J. C., R. Milner, L. Morris, M. C. Newey, and C. P. Wadsworth. 1978. "A Metalanguage for Interactive Proof in LCF". In: *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. POPL '78*. Tucson, Arizona: ACM. 119–130. DOI: 10.1145/512760.512773.

Gordon, M. J. C., R. Milner, and C. P. Wadsworth. 1979. "Edinburgh LCF: A Mechanised Logic of Computation". In: *Lecture Notes in Computer Science*. Vol. 78. DOI: 10.1007/3-540-09724-4.

Graham, P. 1996. *ANSI Common Lisp*. Prentice Hall Press.

Grégoire, B. and A. Mahboubi. 2005. "Proving equalities in a commutative ring done right in Coq". In: *International Conference on Theorem Proving in Higher Order Logics*. Springer. 98–113. DOI: 10.1007/11541868_7.

Gu, R., J. Koenig, T. Ramananandro, Z. Shao, X. Wu, S. Weng, H. Zhang, and Y. Guo. 2015. "Deep Specifications and Certified Abstraction Layers". In: *POPL*. ACM. 595–608. DOI: 10.1145/2676726.2676975.

Gu, R., Z. Shao, H. Chen, X. (Wu, J. Kim, V. Sjöberg, and D. Costanzo. 2016. "CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels". In: *OSDI*. USENIX Association. 653–669.

Gu, R., Z. Shao, J. Kim, X. Wu, J. Koenig, V. Sjöberg, H. Chen, D. Costanzo, and T. Ramananandro. 2018. "Certified concurrent abstraction layers". In: *PLDI*. ACM. 646–661. DOI: 10.1145/3192366.3192381.

Guanciale, R., H. Nemati, M. Dam, and C. Baumann. 2016. "Provably secure memory isolation for Linux on ARM". *Journal of Computer Security*. 24(6): 793–837. DOI: 10.3233/JCS-160558.

Haftmann, F., A. Krauss, O. Kunčar, and T. Nipkow. 2013. "Data Refinement in Isabelle/HOL". In: *Interactive Theorem Proving*. Berlin, Heidelberg: Springer. 100–115. DOI: 10.1007/978-3-642-39634-2_10.

Haftmann, F. and T. Nipkow. 2010. "Code Generation via Higher-order Rewrite Systems". In: *Proceedings of the 10th International Conference on Functional and Logic Programming. FLOPS'10*. Sendai, Japan: Springer-Verlag. 103–117. DOI: 10.1007/978-3-642-12251-4_9.

Haftmann, F. and M. Wenzel. 2007. "Constructive Type Classes in Isabelle". In: *Types for Proofs and Programs*. Ed. by T. Altenkirch and C. McBride. Berlin, Heidelberg: Springer. 160–174. DOI: 10.1007/978-3-540-74464-1_11.

Haftmann, F. and M. Wenzel. 2009. "Local Theory Specifications in Isabelle/Isar". In: *Types for Proofs and Programs*. Berlin, Heidelberg: Springer. 153–168. DOI: 10.1007/978-3-642-02444-3_10.

Hales, T. C. 2007. "The Jordan curve theorem, formally and informally". *American Mathematical Monthly*. 114(10): 882–894.

Hales, T. C., J. Harrison, S. McLaughlin, T. Nipkow, S. Obua, and R. Zumkeller. 2011. "A Revision of the Proof of the Kepler Conjecture". In: *The Kepler Conjecture: The Hales-Ferguson Proof*. New York, NY: Springer New York. 341–376. DOI: 10.1007/978-1-4614-1129-1_9.

Hales, T., M. Adams, G. Bauer, T. D. Dang, J. Harrison, L. T. Hoang, C. Kaliszyk, V. Magron, S. McLaughlin, T. T. Nguyen, Q. T. Nguyen, T. Nipkow, S. Obua, J. Pleso, J. Rute, A. Solovyev, T. H. A. Ta, N. T. Tran, T. D. Trieu, J. Urban, K. Vu, and R. Zumkeller. 2017. "A Formal Proof of the Kepler Conjecture". *Forum of Mathematics, Pi*. 5. DOI: 10.1017/fmp.2017.1.

Harper, R. 1999. "Proof-directed Debugging". *J. Funct. Program.* 9(4): 463–469. DOI: 10.1017/S0956796899003378.

Harper, R. 2011. "Modules Matter Most". URL: https://existentialtype.wordpress.com/2011/04/16/modules-matter-most/.

Harper, R. 2016. *Practical foundations for programming languages*. 2nd. Cambridge, UK: Cambridge University Press. DOI: 10.1017/CBO9781316576892.

Harper, R. and K. Crary. 2019. Personal communication.

Harper, R., F. Honsell, and G. Plotkin. 1993. "A Framework for Defining Logics". *J. ACM*. 40(1): 143–184. DOI: 10.1145/138027.138060.

Harper, R., F. Honsell, and G. D. Plotkin. 1987. "A Framework for Defining Logics". In: *Proceedings of the Second Annual IEEE Symposium on Logic in Computer Science (LICS 1987)*. Ithaca, NY, USA: IEEE Computer Society Press. 194–204.

Harper, R. and D. R. Licata. 2007. "Mechanizing metatheory in a logical framework". *Journal of Functional Programming*. 17(4-5): 613–673. DOI: 10.1017/S0956796807006430.

Harrison, J. 1995. "Metatheory and Reflection in Theorem Proving: A Survey and Critique". *Technical Report* No. CRC-053. Millers Yard, Cambridge, UK: SRI Cambridge. URL: http://www.cl.cam.ac.uk/~jrh13/papers/reflect.dvi.gz.

Harrison, J. 1996. "A Mizar Mode for HOL". In: *Theorem Proving in Higher Order Logics*. Vol. 1125. Turku, Finland: Springer. 203–220. DOI: 10.1007/BFb0105406.

Harrison, J., J. Urban, and F. Wiedijk. 2014. "History of Interactive Theorem Proving". In: *Computational Logic*. Ed. by J. H. Siekmann. Vol. 9. *Handbook of the History of Logic*. No. Supplement C. North-Holland. 135–214. DOI: 10.1016/B978-0-444-51624-4.50004-6.

Hasker, R. W. and U. S. Reddy. 1992. "Generalization at higher types". In: *Proceedings of the Workshop on the λProlog Programming Language*. 257–271.

Hemer, D., I. Hayes, and P. Strooper. 2001. "Refinement Calculus for Logic Programming in Isabelle/HOL". In: *Theorem Proving in Higher Order Logics*. Berlin, Heidelberg: Springer. 249–260. DOI: 10.1007/3-540-44755-5_18.

Heras, J. and E. Komendantskaya. 2013. "ML4PG in Computer Algebra Verification". In: *Intelligent Computer Mathematics*. Berlin, Heidelberg: Springer. 354–358. DOI: 10.1007/978-3-642-39320-4_28.

Heras, J. and E. Komendantskaya. 2014. "Recycling Proof Patterns in Coq: Case Studies". *Mathematics in Computer Science*. 8(1): 99–116. DOI: 10.1007/s11786-014-0173-1.

Heras, J., E. Komendantskaya, M. Johansson, and E. Maclean. 2013. "Proof-Pattern Recognition and Lemma Discovery in ACL2". In: *Logic for Programming, Artificial Intelligence, and Reasoning: 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings*. Berlin, Heidelberg: Springer. 389–406. DOI: 10.1007/978-3-642-45221-5_27.

Heyting, A. 1956. "Intuitionism. An introduction".

Hoare, C. A. R. 1969. "An Axiomatic Basis for Computer Programming". *Commun. ACM*. 12(10): 576–580. DOI: 10.1145/363235.363259.

Hoare, C. A. R. 1971. "Proof of a Program: FIND". *Commun. ACM*. 14(1): 39–45. DOI: 10.1145/362452.362489.

HOL Development Team. 2018. "The HOL System TUTORIAL". URL: http://sourceforge.net/projects/hol/files/hol/kananaskis-12/kananaskis-12-tutorial.pdf?download.

HOL Development Team. 2016-2018. "Running hol". URL: https://hol-theorem-prover.org/guidebook/#running-hol.

HOL Light Development Team. 1996-2019. "HOL Light". URL: http://www.cl.cam.ac.uk/~jrh13/hol-light.

Homeier, P. V. 2005. "A Design Structure for Higher Order Quotients". In: *In Proc. of the 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs), volume 3603 of LNCS*. 130–146. DOI: 10.1007/11541868_9.

Howard, W. A. 1980. "The formulae-as-types notion of constructions". In: *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Ed. by J. P. Seldin and J. R. Hindley. Academic Press.

Howe, D. J. 1988. "Computational metatheory in Nuprl". In: *International Conference on Automated Deduction*. Springer. 238–257. DOI: 10.1007/BFb0012835.

Huang, D., P. Dhariwal, D. Song, and I. Sutskever. 2019. "GamePad: A Learning Environment for Theorem Proving". In: *International Conference on Learning Representations*. URL: https://openreview.net/forum?id=r1xwKoR9Y7.

Huffman, B. and O. Kunčar. 2013. "Lifting and Transfer: A Modular Design for Quotients in Isabelle/HOL". In: *Certified Programs and Proofs: Third International Conference. CPP 2013*. Cham: Springer International Publishing. 131–146. DOI: 10.1007/978-3-319-03545-1_9.

Hupel, L. and T. Nipkow. 2018. "A Verified Compiler from Isabelle/HOL to CakeML". In: *Programming Languages and Systems*. Cham: Springer International Publishing. 999–1026. DOI: 10.1007/978-3-319-89884-1_35.

Hurd, J. 1999. "Integrating Gandalf and HOL". In: *International Conference on Theorem Proving in Higher Order Logics*. Springer. 311–321. DOI: 10.1007/3-540-48256-3_21.

Hurd, J. 2003. "First-order proof tactics in higher-order logic theorem provers". *Design and Application of Strategies/Tactics in Higher Order Logics, number NASA/CP-2003-212448 in NASA Technical Reports*: 56–68.

Igarashi, A., B. C. Pierce, and P. Wadler. 2001. "Featherweight Java: A minimal core calculus for Java and GJ". *ACM Trans. Program. Lang. Syst.* 23(3): 396–450. DOI: 10.1145/503502.503505.

Inria. 1999-2018. "The Tactic Language". URL: https://coq.inria.fr/refman/proof-engine/ltac.html.

Ireland, A. 1996. "Productive Use of Failure in Inductive Proof". *J. Autom. Reasoning.* 16(1-2): 79–111. DOI: 10.1007/BF00244460.

Irvine, A. D. 2016. "Principia Mathematica". In: *The Stanford Encyclopedia of Philosophy.* Ed. by E. N. Zalta. Winter 2016. Metaphysics Research Lab, Stanford University. URL: https://plato.stanford.edu/archives/win2016/entries/principia-mathematica/.

Irvine, A. D. and H. Deutsch. 2016. "Russell's Paradox". In: *The Stanford Encyclopedia of Philosophy.* Ed. by E. N. Zalta. Winter 2016. Metaphysics Research Lab, Stanford University. URL: https://plato.stanford.edu/archives/win2016/entries/russell-paradox/.

Irving, G., C. Szegedy, A. A. Alemi, N. Een, F. Chollet, and J. Urban. 2016. "DeepMath - Deep Sequence Models for Premise Selection". In: *Advances in Neural Information Processing Systems 29.* Curran Associates, Inc. 2235–2243. URL: http://papers.nips.cc/paper/6280-deepmath-deep-sequence-models-for-premise-selection.pdf.

Isabelle Development Team. 1994-2019. "Isabelle". URL: http://isabelle.in.tum.de.

Jang, D., Z. Tatlock, and S. Lerner. 2012. "Establishing Browser Security Guarantees Through Formal Shim Verification". In: *Proceedings of the 21st USENIX Conference on Security Symposium. Security'12.* Bellevue, WA: USENIX Association. 8–8.

Jeffery, R., M. Staples, J. Andronick, G. Klein, and T. Murray. 2015. "An empirical research agenda for understanding formal methods productivity". *Information and Software Technology.* 60: 102–112. DOI: 10.1016/j.infsof.2014.11.005.

JetBrains. 2001-2019. "IntelliJ IDEA". URL: http://www.jetbrains.com/idea/.

Johansson, M. 2017. "Automated Theory Exploration for Interactive Theorem Proving". In: *International Conference on Interactive Theorem Proving*. Springer. 1–11. DOI: 10.1007/978-3-319-66107-0_1.

Johansson, M., D. Rosén, N. Smallbone, and K. Claessen. 2014. "Hipster: Integrating Theory Exploration in a Proof Assistant". *CoRR*. abs/1405.3426.

Johnsen, E. B. and C. Lüth. 2004. "Theorem Reuse by Proof Term Transformation". In: *Theorem Proving in Higher Order Logics: 17th International Conference, TPHOLs 2004, Park City, Utah, USA, September 14-17, 2004. Proceedings*. Berlin, Heidelberg: Springer. 152–167. DOI: 10.1007/978-3-540-30142-4_12.

Jones, C. B. 1983. "Specification and Design of (Parallel) Programs". In: *IFIP Congress*. 321–332.

Jourdan, J.-H., F. Pottier, and X. Leroy. 2012. "Validating LR(1) Parsers". In: *Programming Languages and Systems*. Berlin, Heidelberg: Springer. 397–416. DOI: 10.1007/978-3-642-28869-2_20.

Jung, R., J.-H. Jourdan, R. Krebbers, and D. Dreyer. 2017. "RustBelt: Securing the Foundations of the Rust Programming Language". *Proc. ACM Program. Lang.* 2(POPL): 66:1–66:34. DOI: 10.1145/3158154.

Jung, R., R. Krebbers, J.-H. Jourdan, A. Bizjak, L. Birkedal, and D. Dreyer. 2018. "Iris from the ground up: A modular foundation for higher-order concurrent separation logic". *Journal of Functional Programming*. 28: e20. DOI: 10.1017/S0956796818000151.

Jung, R., D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer. 2015. "Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning". In: *POPL*. ACM. 637–650. DOI: 10.1145/2676726.2676980.

Kadoda, G. F., R. G. Stone, and D. Diaper. 1999. "Desirable features of educational theorem provers - a cognitive dimensions viewpoint". In: *PPIG*. Psychology of Programming Interest Group. 4.

Kaiser, J.-O., B. Ziliani, R. Krebbers, Y. Régis-Gianas, and D. Dreyer. 2018. "Mtac2: Typed Tactics for Backward Reasoning in Coq". *Proc. ACM Program. Lang.* 2(ICFP): 78:1–78:31. DOI: 10.1145/3236773.

Kaivola, R. and K. Kohatsu. 2003. "Proof engineering in the large: formal verification of Pentium 4 floating-point divider". *International Journal on Software Tools for Technology Transfer*. 4(3): 323–334. DOI: 10.1007/s10009-002-0081-6.

Kaliszyk, C. 2007. "Web interfaces for proof assistants". *Electronic Notes in Theoretical Computer Science*. 174(2): 49–61. DOI: 10.1016/j.entcs.2006.09.021.

Kaliszyk, C., F. Chollet, and C. Szegedy. 2017a. "HolStep: A Machine Learning Dataset for Higher-order Logic Theorem Proving". In: *ICLR*. URL: https://openreview.net/forum?id=ryuxYmvel.

Kaliszyk, C. and R. O'Connor. 2009. "Computing with Classical Real Numbers". *Journal of Formalized Reasoning*. 2(1): 27–39. DOI: 10.6092/issn.1972-5787/1411.

Kaliszyk, C. and J. Urban. 2014. "Learning-Assisted Automated Reasoning with Flyspeck". *Journal of Automated Reasoning*. 53(2): 173–213. DOI: 10.1007/s10817-014-9303-3.

Kaliszyk, C. and J. Urban. 2015. "HOL(y)Hammer: Online ATP Service for HOL Light". *Mathematics in Computer Science*. 9(1): 5–22. DOI: 10.1007/s11786-014-0182-0.

Kaliszyk, C., J. Urban, and J. Vyskočil. 2017b. "Automating Formalization by Statistical and Semantic Parsing of Mathematics". In: *Interactive Theorem Proving*. Cham: Springer International Publishing. 12–27. DOI: 10.1007/978-3-319-66107-0_2.

Kammüller, F., M. Wenzel, and L. C. Paulson. 1999. "Locales A Sectioning Concept for Isabelle". In: *Theorem Proving in Higher Order Logics*. Berlin, Heidelberg: Springer. 149–165. DOI: 10.1007/3-540-48256-3_11.

Kästner, D., X. Leroy, S. Blazy, B. Schommer, M. Schmidt, and C. Ferdinand. 2017. "Closing the Gap – The Formally Verified Optimizing Compiler CompCert". In: *SSS'17: Safety-critical Systems Symposium 2017. Developments in System Safety Engineering: Proceedings of the Twenty-fifth Safety-critical Systems Symposium*. Bristol, UK: CreateSpace. 163–180. URL: https://hal.inria.fr/hal-01399482.

Kästner, D., U. Wünsche, J. Barrho, M. Schlickling, B. Schommer, M. Schmidt, C. Ferdinand, X. Leroy, and S. Blazy. 2018. "CompCert: Practical experience on integrating and qualifying a formally verified optimizing compiler". In: *ERTS 2018: Embedded Real Time Software and Systems*. SEE. URL: https://hal.inria.fr/hal-01643290.

Kell, S., D. P. Mulligan, and P. Sewell. 2016. "The Missing Link: Explaining ELF Static Linking, Semantically". In: *OOPSLA*. Amsterdam, Netherlands: ACM. 607–623. DOI: 10.1145/2983990.2983996.

Kim, J., V. Sjöberg, R. Gu, and Z. Shao. 2017. "Safety and Liveness of MCS Lock - Layer by Layer". In: *APLAS*. Vol. 10695. *LNCS*. Springer. 273–297. DOI: 10.1007/978-3-319-71237-6_14.

Klein, G. 2014. "Proof Engineering Considered Essential". In: *FM 2014: Formal Methods: 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*. Cham: Springer International Publishing. 16–21. DOI: 10.1007/978-3-319-06410-9_2.

Klein, G. 2015. "Gerwin's Style Guide for Isabelle/HOL". URL: https://proofcraft.org/blog/isabelle-style.html.

Klein, G., J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser. 2014. "Comprehensive Formal Verification of an OS Microkernel". *ACM Trans. Comput. Syst.* 32(1): 2:1–2:70. DOI: 10.1145/2560537.

Klein, G., J. Andronick, M. Fernandez, I. Kuz, T. Murray, and G. Heiser. 2018. "Formally Verified Software in the Real World". *Commun. ACM*. 61(10): 68–77. DOI: 10.1145/3230627.

Klein, G., J. Andronick, G. Keller, D. Matichuk, T. Murray, and L. O'Connor. 2017. "Provably trustworthy systems". *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*. 375(2104). DOI: 10.1098/rsta.2015.0404.

Klein, G., K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. 2009. "seL4: Formal Verification of an OS Kernel". In: *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles. SOSP '09*. Big Sky, MT, USA: ACM. 207–220. DOI: 10.1145/1629575.1629596.

Klein, G., T. Nipkow, L. Paulson, and R. Thiemann. 2004-2019. "Archive of Formal Proofs: Submission Guidelines". URL: https://www.isa-afp.org/submitting.html.

Ko, H.-S. and J. Gibbons. 2016. "Programming with ornaments". *Journal of Functional Programming*. 27. DOI: 10.1017/S0956796816000307.

Kolbe, T. and C. Walther. 1998. "Proof analysis, generalization and reuse". In: *Automated Deduction — A Basis for Applications*. Springer. 189–219. DOI: 10.1007/978-94-017-0435-9_8.

Komendantskaya, E., J. Heras, and G. Grov. 2012. "Machine Learning in Proof General: Interfacing Interfaces". In: *Proceedings 10th International Workshop On User Interfaces for Theorem Provers, UITP 2012, Bremen, Germany, July 11th, 2012*. 15–41. DOI: 10.4204/EPTCS.118.2.

Krafft, D. B. 1981. "AVID: A system for the interactive development of verifiably correct programs". *PhD thesis*. Cornell University.

Krebbers, R., J.-H. Jourdan, R. Jung, J. Tassarotti, J.-O. Kaiser, A. Timany, A. Charguéraud, and D. Dreyer. 2018. "MoSeL: A General, Extensible Modal Framework for Interactive Proofs in Separation Logic". *Proc. ACM Program. Lang.* 2(ICFP): 77:1–77:30. DOI: 10.1145/3236772.

Krebbers, R., A. Timany, and L. Birkedal. 2017. "Interactive proofs in higher-order concurrent separation logic". In: *POPL*. ACM. 205–217. DOI: 10.1145/3009837.3009855.

Kroening, D. and O. Strichman. 2008. *Decision Procedures: An Algorithmic Point of View*. 1st ed. Springer Publishing Company, Incorporated. DOI: 10.1007/978-3-540-74105-3.

Kühlwein, D., J. C. Blanchette, C. Kaliszyk, and J. Urban. 2013. "MaSh: Machine Learning for Sledgehammer". In: *Interactive Theorem Proving*. Berlin, Heidelberg: Springer. 35–50. DOI: 10.1007/978-3-642-39634-2_6.

Kühlwein, D., T. van Laarhoven, E. Tsivtsivadze, J. Urban, and T. Heskes. 2012. "Overview and Evaluation of Premise Selection Techniques for Large Theory Mathematics". In: *Automated Reasoning*. Berlin, Heidelberg: Springer. 378–392. DOI: 10.1007/978-3-642-31365-3_30.

Kumar, R. 2015. "Self-compilation and self-verification". *PhD thesis*. University of Cambridge.

Kumar, R., M. O. Myreen, M. Norrish, and S. Owens. 2014. "CakeML: A Verified Implementation of ML". In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '14*. San Diego, CA, USA: ACM. 179–191. DOI: 10.1145/2535838.2535841.

Kumar, R., T. Kropf, and K. Schneider. 1991. "Integrating a first-order automatic prover in the HOL environment". In: *1991 International Workshop on the HOL Theorem Proving System and Its Applications*. IEEE. 170–176. DOI: 10.1109/HOL.1991.596284.

Kunčar, O. and A. Popescu. 2018. "A Consistent Foundation for Isabelle/HOL". *Journal of Automated Reasoning*. Jan. DOI: 10.1007/s10817-018-9454-8.

Lammich, P. 2013. "Automatic Data Refinement". In: *Interactive Theorem Proving*. Berlin, Heidelberg: Springer. 84–99.

Lammich, P. 2015. "Refinement to imperative/HOL". In: *International Conference on Interactive Theorem Proving*. Springer. 253–269.

Lampropoulos, L., Z. Paraskevopoulou, and B. C. Pierce. 2017. "Generating good generators for inductive relations". *Proceedings of the ACM on Programming Languages*. 2(POPL): 45:1–45:30. DOI: 10.1145/3158133.

Lean Development Team. 2014-2017. "Theorem Proving in Lean". URL: http://leanprover.github.io/tutorial/.

Lean Development Team. 2017-2018. "Javascript interface to the Lean server". URL: http://github.com/leanprover/lean-client-js.

Lean Development Team. 2016-2019. "Lean for VS Code". URL: http://github.com/leanprover/vscode-lean.

Leibniz, G. W. 1685. *The Art of Discovery*. Wiener 51.

Leino, K. R. M. 2010. "Dafny: An Automatic Program Verifier for Functional Correctness". In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Berlin, Heidelberg: Springer. 348–370. DOI: 10.1007/978-3-642-17511-4_20.

Lerner, S., S. R. Foster, and W. G. Griswold. 2015. "Polymorphic blocks: Formalism-inspired UI for structured connectors". In: *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. ACM. 3063–3072. DOI: 10.1145/2702123.2702302.

Leroy, X. 2006. "Formal certification of a compiler back-end or: programming a compiler with a proof assistant". In: *POPL*. ACM. 42–54. DOI: 10.1145/1111037.1111042.

Leroy, X. 2007. "A locally nameless solution to the POPLMark challenge". *Research report* No. 6098. INRIA. URL: http://gallium.inria.fr/~xleroy/publi/POPLmark-locally-nameless.pdf.

Leroy, X. 2009. "Formal Verification of a Realistic Compiler". *Commun. ACM*. 52(7): 107–115. DOI: 10.1145/1538788.1538814.

Leroy, X. 2015. "Using Coq's evaluation mechanisms in anger". URL: http://gallium.inria.fr/blog/coq-eval/.

Leroy, X., A. W. Appel, S. Blazy, and G. Stewart. 2012. "The CompCert Memory Model, Version 2". *Research Report* No. RR-7987. INRIA. 26. URL: https://hal.inria.fr/hal-00703441.

Leroy, X. 2017. "The formal verification of compilers". URL: https://deepspec.org/event/dsss17/leroy-dsss17.pdf.

Lescuyer, S. and S. Conchon. 2009. "Improving Coq propositional reasoning using a lazy CNF conversion scheme". In: *International Symposium on Frontiers of Combining Systems*. Springer. 287–303.

Letouzey, P. 2003. "A New Extraction for Coq". In: *Types for Proofs and Programs*. Berlin, Heidelberg: Springer. 200–219. DOI: 10.1007/3-540-39185-1_12.

Letouzey, P. 2004. "Programmation fonctionnelle certifiee – L'extraction de programmes dans l'assistant Coq". *PhD thesis*. Universite Paris-Sud. URL: https://www.irif.fr/~letouzey/download/these_letouzey.pdf.

Letouzey, P. 2008. "Coq Extraction, an Overview". In: *Logic and Theory of Algorithms, Fourth Conference on Computability in Europe, CiE 2008*. Vol. 5028. Springer. DOI: 10.1007/978-3-540-69407-6_39.

Ley-Wild, R. and A. Nanevski. 2013. "Subjective auxiliary state for coarse-grained concurrency". In: *POPL*. ACM. 561–574. DOI: 10.1145/2429069.2429134.

Lin, Y., G. Grov, and R. Arthan. 2016. "Understanding and maintaining tactics graphically OR how we are learning that a diagram can be worth more than 10K LoC". *CoRR*. abs/1610.05593.

Lindblad, F. and M. Benke. 2006. "A Tool for Automated Theorem Proving in Agda". In: *Proceedings of the 2004 International Conference on Types for Proofs and Programs. TYPES'04*. Jouy-en-Josas, France: Springer-Verlag. 154–169. DOI: 10.1007/11617990_10.

Liu, Z., C. Morisset, and S. Wang. 2011. "A Graph-Based Implementation for Mechanized Refinement Calculus of OO Programs". In: *Formal Methods: Foundations and Applications*. Berlin, Heidelberg: Springer. 258–273. DOI: 10.1007/978-3-642-19829-8_17.

Loos, S., G. Irving, C. Szegedy, and C. Kaliszyk. 2017. "Deep Network Guided Proof Search". In: *LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning*. Vol. 46. *EPiC Series in Computing*. EasyChair. 85–105. DOI: 10.29007/8mwc.

Luo, Z. 1999. "Coercive subtyping". *Journal of Logic and Computation*. 9(1): 105–130. DOI: 10.1093/logcom/9.1.105.

Lynch, N. and F. Vaandrager. 1994. "FORWARD AND BACKWARD SIMULATIONS PART I: UNTIMED SYSTEMS (Replaces TM-486)". *Tech. rep.* Cambridge, MA, USA.

MacQueen, D. B. 1986. "Using Dependent Types to Express Modular Structure". In: *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. POPL '86*. St. Petersburg Beach, Florida: ACM. 277–286. DOI: 10.1145/512644.512670.

Magaud, N. and Y. Bertot. 2002. "Changing Data Structures in Type Theory: A Study of Natural Numbers". In: *Types for Proofs and Programs: International Workshop. TYPES 2000*. Berlin, Heidelberg: Springer. 181–196. DOI: 10.1007/3-540-45842-5_12.

Mahboubi, A. and E. Tassi. 2013. "Canonical Structures for the Working Coq User". In: *Interactive Theorem Proving*. Vol. 7998. *LNCS*. Berlin, Heidelberg: Springer. 19–34. DOI: 10.1007/978-3-642-39634-2_5.

Malecha, G. and J. Bengtson. 2016. "Extensible and Efficient Automation Through Reflective Tactics". In: *Programming Languages and Systems: 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2–8, 2016, Proceedings*. Berlin, Heidelberg: Springer. 532–559. DOI: 10.1007/978-3-662-49498-1_21.

Malecha, G., G. Morrisett, A. Shinnar, and R. Wisnesky. 2010. "Toward a Verified Relational Database Management System". In: *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '10*. Madrid, Spain: ACM. 237–248. DOI: 10.1145/1706299.1706329.

Martin-Löf, P. 1982. "Constructive Mathematics and Computer Programming". In: *Logic, Methodology and Philosophy of Science VI*. Vol. 104. *Studies in Logic and the Foundations of Mathematics*. Elsevier. 153–175. DOI: 10.1016/S0049-237X(09)70189-2.

Martin-Löf, P. 1984. *Intuitionistic Type Theory*. Bibliopolis.

Matichuk, D., T. Murray, J. Andronick, R. Jeffery, G. Klein, and M. Staples. 2015a. "Empirical Study Towards a Leading Indicator for Cost of Formal Software Verification". In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. 722–732. DOI: 10.1109/ICSE.2015.85.

Matichuk, D., T. C. Murray, and M. Wenzel. 2015b. "Eisbach: A Proof Method Language for Isabelle". *Journal of Automated Reasoning*. 56: 261–282. DOI: 10.1007/s10817-015-9360-2.

Matichuk, D., M. Wenzel, and T. Murray. 2015c. "The Eisbach user manual". *Isabelle Community*.

McBride, C. 1996. "Inverting inductively defined relations in LEGO". In: *International Workshop on Types for Proofs and Programs*. Springer. 236–253.

McBride, C. 2002. "Elimination with a Motive". In: *Types for Proofs and Programs*. Berlin, Heidelberg: Springer. 197–216. DOI: 10.1007/3-540-45842-5_13.

McBride, C. 2011. "Ornamental algebras, algebraic ornaments". URL: http://plv.mpi-sws.org/plerg/papers/mcbride-ornaments-2up.pdf.

McBride, C. 2015. "Turing-Completeness Totally Free". In: *Mathematics of Program Construction*. Cham: Springer International Publishing. 257–275. DOI: 10.1007/978-3-319-19797-5_13.

McCarthy, J. 1960. "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I". *Commun. ACM*. 3(4): 184–195. DOI: 10.1145/367177.367199.

McCarthy, J. 1963. "A Basis for a Mathematical Theory of Computation". In: *Computer Programming and Formal Systems*. Ed. by P. Braffort and D. Hirschberg. Vol. 35. *Studies in Logic and the Foundations of Mathematics*. Elsevier. 33–70. DOI: 10.1016/S0049-237X(08)72018-4.

McCreight, A. 2009. "Practical tactics for separation logic". In: *International Conference on Theorem Proving in Higher Order Logics*. Springer. 343–358. DOI: 10.1007/978-3-642-03359-9_24.

Mehnert, H. and D. Christiansen. 2014. "Tool Demonstration: An IDE for Programming and Proving in Idris". *Proceedings of Vienna Summer of Logic, VSL*. 14: 2.

Mehta, F. and T. Nipkow. 2003. "Proving Pointer Programs in Higher-Order Logic". In: *CADE*. Vol. 2741. *LNCS*. Springer. 121–135. DOI: 10.1007/978-3-540-45085-6_10.

Microsoft. 1997-2019. "Visual Studio". URL: http://visualstudio.microsoft.com/.

Miller, D. 2018. "Mechanized Metatheory Revisited". *Journal of Automated Reasoning*. Oct. DOI: 10.1007/s10817-018-9483-3.

Milner, R. 1972. "Implementation and Applications of Scott's Logic for Computable Functions". In: *Proceedings of ACM Conference on Proving Assertions About Programs*. Las Cruces, NM, USA: ACM. 1–6. DOI: 10.1145/800235.807067.

Milner, R., M. Tofte, R. Harper, and D. MacQueen. 1997. *The Definition of Standard ML (Revised)*. Cambridge, MA, USA: MIT Press.

Milner, R. and R. Weyhrauch. 1972. "Proving compiler correctness in a mechanized logic". *Machine Intelligence*. 7: 51–70.

Monperrus, M. 2018. "Automatic Software Repair: A Bibliography". *ACM Comput. Surv.* 51(1): 17:1–17:24. DOI: 10.1145/3105906.

Morrisett, G., G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan. 2012. "RockSalt: Better, Faster, Stronger SFI for the x86". In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '12*. Beijing, China: ACM. 395–404. DOI: 10.1145/2254064.2254111.

Mulhern, A. 2006. "Proof Weaving". In: *In Proceedings of the First Informal ACM SIGPLAN Workshop on Mechanizing Metatheory*.

Mullen, E., S. Pernsteiner, J. R. Wilcox, Z. Tatlock, and D. Grossman. 2018. "Oeuf: Minimizing the Coq Extraction TCB". In: *CPP*. Los Angeles, CA, USA: ACM. 172–185. DOI: 10.1145/3167089.

Müller, D., T. Gauthier, C. Kaliszyk, M. Kohlhase, and F. Rabe. 2017. "Classification of Alignments Between Concepts of Formal Mathematical Systems". In: *Intelligent Computer Mathematics*. Cham: Springer International Publishing. 83–98. DOI: 10.1007/978-3-319-62075-6_7.

Mulligan, D. P., S. Owens, K. E. Gray, T. Ridge, and P. Sewell. 2014. "Lem: Reusable Engineering of Real-world Semantics". In: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming. ICFP '14*. Gothenburg, Sweden: ACM. 175–188. DOI: 10.1145/2628136.2628143.

Murphy-Hill, E. and D. Grossman. 2014. "How Programming Languages Will Co-evolve with Software Engineering: A Bright Decade Ahead". In: *Proceedings of the on Future of Software Engineering. FOSE 2014*. Hyderabad, India: ACM. 145–154. DOI: 10.1145/2593882.2593898.

Murray, T. and P. C. van Oorschot. 2018. "BP: Formal Proofs, the Fine Print and Side Effects". In: *IEEE Cybersecurity Development (SecDev)*. 1–10. DOI: 10.1109/SecDev.2018.00009.

Myreen, M. O. 2008-2018. "Guide to HOL4 interaction and basic proofs". URL: http://hol-theorem-prover.org/HOL-interaction.pdf.

Myreen, M. O. and S. Owens. 2012. "Proof-producing Synthesis of ML from Higher-order Logic". In: *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming. ICFP '12*. Copenhagen, Denmark: ACM. 115–126. DOI: 10.1145/2364527.2364545.

Nagashima, Y. and Y. He. 2018. "PaMpeR: Proof Method Recommendation System for Isabelle/HOL". In: *Proceedings of the International Conference on Automated Software Engineering. ASE 2018*. Montpellier, France: ACM. 362–372. DOI: 10.1145/3238147.3238210.

Nagashima, Y. and R. Kumar. 2017. "A Proof Strategy Language and Proof Script Generation for Isabelle/HOL". In: *Automated Deduction – CADE 26*. Cham: Springer International Publishing. 528–545. DOI: 10.1007/978-3-319-63046-5_32.

Nanevski, A., R. Ley-Wild, I. Sergey, and G. A. Delbianco. 2014. "Communicating State Transition Systems for Fine-Grained Concurrent Resources". In: *Programming Languages and Systems*. Ed. by Z. Shao. Berlin, Heidelberg: Springer. 290–310. DOI: 10.1007/978-3-642-54833-8_16.

Nanevski, A., G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. 2008a. "Ynot: Dependent Types for Imperative Programs". In: *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming. ICFP '08*. Victoria, BC, Canada: ACM. 229–240. DOI: 10.1145/1411204.1411237.

Nanevski, A., F. Pfenning, and B. Pientka. 2008b. "Contextual modal type theory". *ACM Transactions on Computational Logic*. 9(3): 23. DOI: 10.1145/1352582.1352591.

Nanevski, A., V. Vafeiadis, and J. Berdine. 2010. "Structuring the Verification of Heap-manipulating Programs". In: *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '10*. Madrid, Spain: ACM. 261–274. DOI: 10.1145/1706299.1706331.

Narboux, J. 2004. "A decision procedure for geometry in Coq". In: *International Conference on Theorem Proving in Higher Order Logics*. Springer. 225–240. DOI: 10.1007/978-3-540-30142-4_17.

Neis, G., C.-K. Hur, J.-O. Kaiser, C. McLaughlin, D. Dreyer, and V. Vafeiadis. 2015. "Pilsner: A Compositionally Verified Compiler for a Higher-order Imperative Language". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming. ICFP 2015*. Vancouver, BC, Canada: ACM. 166–178. DOI: 10.1145/2784731.2784764.

Newey, M. C. 1973. "Axioms and Theorems for Integers, Lists and Finite Sets in LCF". *Tech. rep.* Memo AIM-184. Stanford, CA, USA.

Nipkow, T. 1989. "Term rewriting and beyond–theorem proving in Isabelle". *Formal Aspects of Computing*. 1(1): 320–338. DOI: 10.1007/BF01887212.

Nipkow, T. 1990. "Proof transformations for equational theories". In: *Symposium on Logic in Computer Science*. IEEE. 278–288. DOI: 10.1109/LICS.1990.113754.

Nipkow, T. and G. Klein. 2014. *Concrete Semantics: With Isabelle/HOL*. Springer. DOI: 10.1007/978-3-319-10542-0.

nLab authors. 2019a. "equality". URL: http://ncatlab.org/nlab/show/equality.

nLab authors. 2019b. "foundation of mathematics". URL: http://ncatlab.org/nlab/show/foundation%20of%20mathematics.

nLab authors. 2019c. "intensional type theory". URL: http://ncatlab.org/nlab/show/intensional%20type%20theory.

Norell, U. 2016. "Agda reflection overhaul". URL: http://lists.chalmers.se/pipermail/agda/2016/008414.html.

Norell, U. 2015-2019. "agda-prelude". URL: http://github.com/UlfNorell/agda-prelude.

O'Connor, L., C. Rizkallah, Z. Chen, S. Amani, J. Lim, Y. Nagashima, T. Sewell, A. Hixon, G. Keller, T. C. Murray, and G. Klein. 2016. "COGENT: Certified Compilation for a Functional Systems Language". *CoRR*. abs/1601.05520.

O'Connor, R. 2005. "Essential Incompleteness of Arithmetic Verified by Coq". In: *Theorem Proving in Higher Order Logics: 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005. Proceedings*. Berlin, Heidelberg: Springer. 245–260. DOI: 10.1007/11541868_16.

O'Hearn, P. W. 2007. "Resources, concurrency, and local reasoning". *Theor. Comput. Sci.* 375(1-3): 271–307. DOI: 10.1016/j.tcs.2006.12.035.

O'Hearn, P. W., J. C. Reynolds, and H. Yang. 2001. "Local Reasoning about Programs that Alter Data Structures". In: *CSL*. Vol. 2142. *LNCS*. Springer. 1–19. DOI: 10.1007/3-540-44802-0_1.

Opdyke, W. F. 1992. "Refactoring: A program restructuring aid in designing object-oriented application frameworks". *PhD thesis*. University of Illinois at Urbana-Champaign.

Oury, N. 2005. "Extensionality in the Calculus of Constructions". In: *Theorem Proving in Higher Order Logics*. Berlin, Heidelberg: Springer. 278–293. DOI: 10.1007/11541868_18.

Owens, S. 2008. "A sound semantics for OCaml Light". *Programming Languages and Systems*: 1–15. DOI: 10.1007/978-3-540-78739-6_1.

Palmskog, K., A. Celik, and M. Gligoric. 2018. "piCoq: Parallel Regression Proving for Large-scale Verification Projects". In: *ISSTA*. Amsterdam, Netherlands: ACM. 344–355. DOI: 10.1145/3213846.3213877.

Paraskevopoulou, Z., C. Hritçu, M. Dénès, L. Lampropoulos, and B. C. Pierce. 2015. "Foundational Property-Based Testing". In: *Interactive Theorem Proving: 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*. Cham: Springer International Publishing. 325–343. DOI: 10.1007/978-3-319-22102-1_22.

Paulin-Mohring, C. 1989a. "Extracting F-omega's Programs from Proofs in the Calculus of Constructions". In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '89*. Austin, TX, USA: ACM. 89–104. DOI: 10.1145/75277.75285.

Paulin-Mohring, C. 1989b. "Extraction de programmes dans le Calcul des Constructions". *PhD thesis*. Université Paris 7.

Paulin-Mohring, C. 1993. "Inductive definitions in the system Coq rules and properties". In: *Typed Lambda Calculi and Applications*. Berlin, Heidelberg: Springer. 328–345. DOI: 10.1007/BFb0037116.

Paulin-Mohring, C. and B. Werner. 1993. "Synthesis of ML programs in the system Coq". *Journal of Symbolic Computation*. 15(5): 607–640. DOI: 10.1016/S0747-7171(06)80007-6.

Paulson, L. 1983. "Tactics and tacticals in Cambridge LCF". *Tech. rep.* University of Cambridge, Computer Laboratory.

Paulson, L. 1984. "Deriving structural induction in LCF". In: *Semantics of Data Types*. Berlin, Heidelberg: Springer. 197–214. DOI: 10.1007/3-540-13346-1_10.

Paulson, L. C. 1988. "A preliminary users manual for Isabelle". *Tech. rep.* University of Cambridge, Computer Laboratory.

Paulson, L. C. 1993. "Isabelle: The Next 700 Theorem Provers". *CoRR*. cs.LO/9301106.

Paulson, L. C. 1994. "Isabelle: A Generic Theorem Prover". In: *Lecture Notes in Computer Science*. Vol. 828. DOI: 10.1007/BFb0030541.

Paulson, L. C. 1997. "Mechanizing Coinduction and Corecursion in Higher-order Logic". *Journal of Logic and Computation*. 7(2): 175–204. DOI: 10.1093/logcom/7.2.175.

Paulson, L. C. 1999. "A generic tableau prover and its integration with Isabelle". *Journal of Universal Computer Science*. 5(3): 73–87. DOI: 10.3217/jucs-005-03-0073.

Paulson, L. C. 2006. "Defining Functions on Equivalence Classes". *ACM Trans. Comput. Logic*. 7(4): 658–675. DOI: 10.1145/1183278.1183280.

Paulson, L. C. and J. C. Blanchette. 2012. "Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers". In: *International Workshop on the Implementation of Logics (IWIL 2010)*. Vol. 2. *EPiC Series*. EasyChair. 1–11.

Pédrot, P.-M. 2019. "Ltac2: Tactical Warfare". In: *CoqPL 2019*.

Peng, K. and D. Ma. 2017. "Tree-Structure CNN for Automated Theorem Proving". In: *Neural Information Processing*. Cham: Springer International Publishing. 3–12. DOI: 10.1007/978-3-319-70096-0_1.

Pfenning, F. and C. Elliott. 1988. "Higher-order Abstract Syntax". In: *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation. PLDI '88*. Atlanta, GA, USA: ACM. 199–208. DOI: 10.1145/53990.54010.

Pfenning, F. 2010. "Lecture Notes on Proofs as Programs: Lecture 2". URL: http://www.cs.cmu.edu/~fp/courses/15816-s10/lectures/02-pap.pdf.

Pfenning, F. and C. Paulin-Mohring. 1990. "Inductively defined types in the Calculus of Constructions". In: *Mathematical Foundations of Programming Semantics*. New York, NY: Springer-Verlag. 209–228. DOI: 10.1007/BFb0040259.

Pfenning, F. and C. Schürmann. 1999. "System Description: Twelf — A Meta-Logical Framework for Deductive Systems". In: *Automated Deduction — CADE-16: 16th International Conference on Automated Deduction Trento, Italy, July 7–10, 1999 Proceedings*. Berlin, Heidelberg: Springer. 202–206. DOI: 10.1007/3-540-48660-7_14.

PG development team. 2016. "Proof General 4.4.1 pre Documentation". URL: https://proofgeneral.github.io/doc/userman/.

Pientka, B. and J. Dunfield. 2008. "Programming with proofs and explicit contexts". In: *Proceedings of the 10th international ACM SIGPLAN conference on Principles and practice of declarative programming*. ACM. 163–173. DOI: 10.1145/1389449.1389469.

Pierce, B. C. 2002. *Types and programming languages*. MIT press.

Pierce, B. C. 2017. Personal communication.

Pierce, B. C. 2019. Personal communication.

Pierce, B. C., C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjöberg, and B. Yorgey. 2014. *Software Foundations*. Electronic textbook. URL: http://www.cis.upenn.edu/~bcpierce/sf.

Pit-Claudel, C. and P. Courtieu. 2016. "Company-Coq: Taking Proof General one step closer to a real IDE". In: *CoqPL'16: The Second International Workshop on Coq for PL*. DOI: 10.5281/zenodo.44331.

Podkopaev, A., O. Lahav, and V. Vafeiadis. 2019. "Bridging the gap between programming languages and hardware weak memory models". *PACMPL*. 3(POPL): 69:1–69:31. DOI: 10.1145/3290382.

Pollack, R. 1995. "On extensibility of proof checkers". In: *Types for Proofs and Programs*. Ed. by P. Dybjer, B. Nordström, and J. Smith. Berlin, Heidelberg: Springer. 140–161. DOI: 10.1007/3-540-60579-7_8.

Pollack, R. 1998. "How to Believe a Machine-Checked Proof". In: *Twenty Five Years of Constructive Type Theory*. Ed. by G. Sambin and J. Smith. Oxford University Press.

Pons, O. 1999. "Conception et rÉalisation d'outils d'aide au dÉveloppement de grosses thÉories dans les systèmes de preuves interactifs". *PhD thesis*.

Poswolsky, A. and C. Schürmann. 2009. "System description: Delphin–a functional programming language for deductive systems". *Electronic Notes in Theoretical Computer Science*. 228: 113–120.

Pouillard, N. 2012. "agda-tactics: Semiring". URL: http://github.com/xplat/agda-tactics/blob/master/Tactics/Nat/Semiring.agda.

Pugh, W. 1991. "The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis". In: *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing. Supercomputing '91*. Albuquerque, NM, USA: ACM. 4–13. DOI: 10.1145/125826.125848.

Qi, Z., F. Long, S. Achour, and M. Rinard. 2015. "An Analysis of Patch Plausibility and Correctness for Generate-and-validate Patch Generation Systems". In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis. ISSTA 2015*. Baltimore, MD, USA: ACM. 24–36. DOI: 10.1145/2771783.2771791.

Rand, R., J. Paykin, and S. Zdancewic. 2017. "QWIRE Practice: Formal Verification of Quantum Circuits in Coq". In: *Proceedings 14th International Conference on Quantum Physics and Logic, QPL 2017, Nijmegen, The Netherlands, 3-7 July 2017*. 119–132. DOI: 10.4204/EPTCS.266.8.

RedPRL Development Team. 2015-2018. "The RedPRL Proof Assistant". URL: http://www.redprl.org.

Reynolds, J. C. 2002. "Separation Logic: A Logic for Shared Mutable Data Structures". In: *LICS*. IEEE Computer Society. 55–74. DOI: 10.1109/LICS.2002.1029817.

Ricketts, D., V. Robert, D. Jang, Z. Tatlock, and S. Lerner. 2014. "Automating Formal Proofs for Reactive Systems". In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '14*. Edinburgh, UK: ACM. 452–462. DOI: 10.1145/2594291.2594338.

Ridge, T., D. Sheets, T. Tuerk, A. Giugliano, A. Madhavapeddy, and P. Sewell. 2015. "SibylFS: Formal Specification and Oracle-based Testing for POSIX and Real-world File Systems". In: *Proceedings of the 25th Symposium on Operating Systems Principles. SOSP '15*. Monterey, California: ACM. 38–53. DOI: 10.1145/2815400.2815411.

Ringer, T. and N. Yazdani. 2018. "PUMPKIN-git". URL: http://github.com/uwplse/PUMPKIN-git.

Ringer, T., N. Yazdani, J. Leo, and D. Grossman. 2018. "Adapting Proof Automation to Adapt Proofs". In: *Proceedings of the 7th ACM SIGPLAN Conference on Certified Programs and Proofs. CPP 2018*. DOI: 10.1145/3167094.

Ringer, T., N. Yazdani, J. Leo, and D. Grossman. 2019. "Ornaments for Proof Reuse in Coq". In: *Interactive Theorem Proving*.

Rizkallah, C., J. Lim, Y. Nagashima, T. Sewell, Z. Chen, L. O'Connor, T. Murray, G. Keller, and G. Klein. 2016. "A Framework for the Automatic Formal Verification of Refinement from Cogent to C". In: *International Conference on Interactive Theorem Proving*. Nancy, France. DOI: 10.1007/978-3-319-43144-4_20.

Robert, V. 2018. "Front-end tooling for building and maintaining dependently-typed functional programs". *PhD thesis*. UC San Diego.

Robert, V. and S. Lerner. 2014-2016. "PeaCoq". URL: http://goto.ucsd.edu/peacoq/.

Robinson, J. A. 1965. "A Machine-Oriented Logic Based on the Resolution Principle". *J. ACM*. 12(1): 23–41. DOI: 10.1145/321250.321253.

Roe, K. and S. Smith. 2016. "CoqPIE: An IDE Aimed at Improving Proof Development Productivity". In: *Interactive Theorem Proving: 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings*. Cham: Springer International Publishing. 491–499. DOI: 10.1007/978-3-319-43144-4_32.

Rompf, T. and N. Amin. 2016. "Type Soundness for Dependent Object Types". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. OOPSLA 2016*. Amsterdam, Netherlands: ACM. 624–641. DOI: 10.1145/2983990.2984008.

Rushby, J. M. 1981. "Design and Verification of Secure Systems". In: *Proceedings of the Eighth ACM Symposium on Operating Systems Principles. SOSP '81*. Pacific Grove, CA, USA: ACM. 12–21. DOI: 10.1145/800216.806586.

Russell, B. 1906. "On Some Difficulties in the Theory of Transfinite Numbers and Order Types". *Proceedings of the London Mathematical Society*. s2-4(1): 29–53. DOI: 10.1112/plms/s2-4.1.29.

Russell, B. 1918. *Introduction to Mathematical Philosophy*. George Allen and Unwin.

Saibi, A. 1997. "Typing Algorithm in Type Theory with Inheritance". In: *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '97*. Paris, France: ACM. 292–301. DOI: 10.1145/263699.263742.

Saibi, A. 1999. "Outils Génériques de Modélisation et de Démonstration pour la Formalisation des Mathématiques en Théorie des Types: application à la Théorie des Catégories". *PhD thesis*. Paris, France: Université Paris VI.

Sangiorgi, D. 2011. *Introduction to Bisimulation and Coinduction*. New York, NY, USA: Cambridge University Press. DOI: 10.1017/CBO9780511777110.

Schäfer, S., T. Tebbi, and G. Smolka. 2015. "Autosubst: Reasoning with de Bruijn terms and parallel substitutions". In: *International Conference on Interactive Theorem Proving*. Springer. 359–374. DOI: 10.1007/978-3-319-22102-1_24.

Schlichtkrull, A., J. C. Blanchette, and D. Traytel. 2019. "A verified prover based on ordered resolution". In: *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM. 152–165.

Scott, D. 1970. "Constructive validity". In: *Symposium on Automatic Demonstration*. Berlin, Heidelberg: Springer. 237–275. DOI: 10.1007/BFb0060636.

Scott, D. S. 1993. "A type-theoretical alternative to ISWIM, CUCH, OWHY". *Theoretical Computer Science*. 121(1): 411–440. DOI: 10.1016/0304-3975(93)90095-B.

Selsam, D. and L. de Moura. 2017. "Congruence Closure in Intensional Type Theory". *CoRR*. abs/1701.04391.

Selsam, D., P. Liang, and D. L. Dill. 2017. "Developing Bug-free Machine Learning Systems with Formal Mathematics". In: *Proceedings of the 34th International Conference on Machine Learning - Volume 70. ICML'17*. Sydney, NSW, Australia: JMLR.org. 3047–3056.

Sergey, I., A. Nanevski, and A. Banerjee. 2015. "Mechanized Verification of Fine-grained Concurrent Programs". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '15*. Portland, OR, USA: ACM. 77–87. DOI: 10.1145/2737924.2737964.

Sergey, I., J. R. Wilcox, and Z. Tatlock. 2017. "Programming and Proving with Distributed Protocols". *Proc. ACM Program. Lang.* 2(POPL): 28:1–28:30. DOI: 10.1145/3158116.

Sewell, P., F. Zappa Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. StrniŠa. 2010. "Ott: Effective tool support for the working semanticist". *Journal of Functional Programming.* 20(1): 71–122. DOI: 10.1017/S0956796809990293.

Sewell, T., F. Kam, and G. Heiser. 2017. "High-assurance timing analysis for a high-assurance real-time operating system". *Real-Time Systems.* 53(5): 812–853. DOI: 10.1007/s11241-017-9286-3.

Shah, N. 2005. ""Rippling: Meta-Level Guidance for Mathematical Reasoning, " by Alan Bundy, David Basin, Dieter Hutter, and Andrew Ireland, Cambridge University Press, 2005". *J. Autom. Reasoning.* 35(4): 429–431. DOI: 10.1007/s10817-006-9027-0.

Shaw, M., J. Aldrich, T. D. Breaux, D. Garlan, C. Kästner, C. Le Goues, and W. L. Scherlis. 2015. "Seminal Papers in Software Engineering: The Carnegie Mellon Canonical Collection". URL: http://reports-archive.adm.cs.cmu.edu/anon/isr2015/CMU-ISR-15-107.pdf.

Slind, K. 1994. "AC unification in HOL90". In: *Higher Order Logic Theorem Proving and Its Applications.* Springer. 437–449. DOI: 10.1007/3-540-57826-9_154.

Slotosch, O. 1997. "Higher order quotients and their implementation in Isabelle/HOL". In: *Theorem Proving in Higher Order Logics.* Berlin, Heidelberg: Springer. 291–306. DOI: 10.1007/BFb0028401.

Smith, R. 2018. "Aristotle's Logic". In: *The Stanford Encyclopedia of Philosophy.* Ed. by E. N. Zalta. Spring 2018. Metaphysics Research Lab, Stanford University. URL: https://plato.stanford.edu/archives/spr2018/entries/aristotle-logic/.

Sørensen, M. H. and P. Urzyczyn. 2006. *Lectures on the Curry-Howard isomorphism.* Vol. 149. Elsevier.

Sozeau, M. 2010. "Equations: A Dependent Pattern-Matching Compiler". In: *Interactive Theorem Proving.* Berlin, Heidelberg: Springer. 419–434. DOI: 10.1007/978-3-642-14052-5_29.

Sozeau, M., A. Anand, S. Boulier, C. Cohen, Y. Forster, F. Kunze, G. Malecha, N. Tabareau, and T. Winterhalter. 2019. "The MetaCoq Project". *Tech. rep.* INRIA. URL: https://hal.inria.fr/hal-02167423.

Sozeau, M. and N. Oury. 2008. "First-Class Type Classes". In: *Theorem Proving in Higher Order Logics: 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*. Berlin, Heidelberg: Springer. 278–293. DOI: 10.1007/978-3-540-71067-7_23.

Sozeau, M. and N. Tabareau. 2014. "Universe Polymorphism in Coq". In: *Interactive Theorem Proving*. Cham: Springer International Publishing. 499–514. DOI: 10.1007/978-3-319-08970-6_32.

Spitters, B. and E. van der Weegen. 2011. "Type classes for mathematics in type theory". *Mathematical Structures in Computer Science*. 21(4): 795–825. DOI: 10.1017/S0960129511000119.

Spiwack, A. 2010. "An abstract type for constructing tactics in Coq". In: *Proof Search in Type Theory*. Edinburgh, UK. URL: https://hal.inria.fr/inria-00502500.

Spiwack, A. 2016. "Inside the design of a tactic system". URL: https://github.com/aspiwack/tacengine/releases/download/v0.1-draft/tacengine.pdf.

Stampoulis, A. and Z. Shao. 2010. "VeriML: Typed Computation of Logical Terms Inside a Language with Effects". In: *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming. ICFP '10*. Baltimore, MD, USA: ACM. 333–344. DOI: 10.1145/1863543.1863591.

Staples, M., R. Jeffery, J. Andronick, T. Murray, G. Klein, and R. Kolanski. 2014. "Productivity for Proof Engineering". In: *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. ESEM '14*. Torino, Italy: ACM. 15:1–15:4. DOI: 10.1145/2652524.2652551.

Staples, M., R. Kolanski, G. Klein, C. Lewis, J. Andronick, T. Murray, R. Jeffery, and L. Bass. 2013. "Formal Specifications Better Than Function Points for Code Sizing". In: *Proceedings of the 2013 International Conference on Software Engineering. ICSE '13*. San Francisco, CA, USA: IEEE Press. 1257–1260. DOI: 10.1109/ICSE.2013.6606692.

Sterling, J. and R. Harper. 2017. "Algebraic Foundations of Proof Refinement". *CoRR*. abs/1703.05215.

Stewart, G., L. Beringer, S. Cuellar, and A. W. Appel. 2015. "Compositional CompCert". In: *POPL*. Mumbai, India: ACM. 275–287. DOI: 10.1145/2676726.2676985.

Strachey, C. 2000. "Fundamental Concepts in Programming Languages". *Higher-Order and Symbolic Computation*. 13(1): 11–49. DOI: 10.1023/A:1010000313106.

StructTact Development Team. 2016-2019. "StructTact". URL: http://github.com/uwplse/StructTact.

Stump, A. 2017. "The calculus of dependent lambda eliminations". *Journal of Functional Programming*. 27. DOI: 10.1017/S0956796817000053.

Svendsen, K. and L. Birkedal. 2014. "Impredicative Concurrent Abstract Predicates". In: 149–168.

Swamy, N., C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoue, and S. Zanella-Béguelin. 2016. "Dependent Types and Multi-monadic Effects in F*". *SIGPLAN Not.* 51(1): 256–270. DOI: 10.1145/2914770.2837655.

Swamy, N., J. Weinberger, C. Schlesinger, J. Chen, and B. Livshits. 2013. "Verifying higher-order programs with the Dijkstra monad". In: *PLDI*. ACM. 387–398. DOI: 10.1145/2491956.2491978.

Swierstra, W. 2008. "Data Types à La Carte". *J. Funct. Program.* 18(4): 423–436. DOI: 10.1017/S0956796808006758.

Swierstra, W. 2009. "A Hoare Logic for the State Monad". In: *TPHOLs*. Vol. 5674. *LNCS*. Springer. 440–451. DOI: 10.1007/978-3-642-03359-9_30.

Syme, D. 1995. "A new interface for HOL — Ideas, issues and implementation". In: *Higher Order Logic Theorem Proving and Its Applications*. Berlin, Heidelberg: Springer. 324–339. DOI: 10.1007/3-540-60275-5_74.

Tabareau, N., É. Tanter, and M. Sozeau. 2018. "Equivalences for Free: Univalent Parametricity for Effective Transport". *Proc. ACM Program. Lang.* 2(ICFP): 92:1–92:29. DOI: 10.1145/3236787.

Tanter, É. and N. Tabareau. 2015. "Gradual Certified Programming in Coq". In: *Proceedings of the 11th Symposium on Dynamic Languages. DLS 2015*. Pittsburgh, PA, USA: ACM. 26–40. DOI: 10.1145/2816707.2816710.

Tarski, A. 1936. "Der Wahrheitsbegriff in den Formalisierten Sprachen". *Studia Philosophica*. 1: 261–405.

Trybulec, A. and H. A. Blair. 1985. "Computer Assisted Reasoning with Mizar". In: *IJCAI*. Vol. 85. Citeseer. 26–28.

Turing, A. 1949. "Checking a large routine". In: *Report of a Conference on High Speed Automatic Calculating Machines*. 67–69.

Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study. URL: https://homotopytypetheory.org/book.

Urban, C. 2008. "Nominal Techniques in Isabelle/HOL". *Journal of Automated Reasoning*. 40(4): 327–356. DOI: 10.1007/s10817-008-9097-2.

Urban, C. and C. Kaliszyk. 2011. "General Bindings and Alpha-Equivalence in Nominal Isabelle". In: *Programming Languages and Systems*. Berlin, Heidelberg: Springer. 480–500.

Valbuena, I. L. and M. Johansson. 2015. "Conditional Lemma Discovery and Recursion Induction in Hipster". *ECEASST*. 72. URL: http://journal.ub.tu-berlin.de/eceasst/article/view/1009.

Van Der Walt, P. and W. Swierstra. 2012. "Engineering proof by reflection in Agda". In: *Symposium on Implementation and Application of Functional Languages*. Springer. 157–173. DOI: 10.1007/978-3-642-41582-1_10.

"Verified cryptography for Firefox 57". URL: https://blog.mozilla.org/security/2017/09/13/verified-cryptography-firefox-57.

Voevodsky, V. 2015. "An experimental library of formalized Mathematics based on the univalent foundations". *Mathematical Structures in Computer Science*. 25(5): 1278–1294. DOI: 10.1017/S0960129514000577.

Voevodsky, V., B. Ahrens, D. Grayson, *et al.* 2011-2019. "*UniMath*: Univalent Mathematics". URL: https://github.com/UniMath.

von Neumann, J. 1993. "First Draft of a Report on the EDVAC". *IEEE Ann. Hist. Comput.* 15(4): 27–75. DOI: 10.1109/85.238389.

von Wright, J. 1994. "Program Refinement by Theorem Prover". In: *6th Refinement Workshop*. London: Springer London. 121–150. DOI: 10.1007/978-1-4471-3240-0_7.

Wadler, P. and S. Blott. 1989. "How to Make Ad-hoc Polymorphism Less Ad Hoc". In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '89*. Austin, Texas, USA: ACM. 60–76. DOI: 10.1145/75277.75283.

Wang, M., Y. Tang, J. Wang, and J. Deng. 2017. "Premise Selection for Theorem Proving by Deep Graph Embedding". In: *Advances in Neural Information Processing Systems 30*. Curran Associates, Inc. 2786–2796. URL: http://papers.nips.cc/paper/6871-premise-selection-for-theorem-proving-by-deep-graph-embedding.pdf.

Warden, S. and F. Biancuzzi. 2009. *Masterminds of Programming: Conversations with the Creators of Major Programming Languages*. O'Reilly Media.

Watt, C. 2018. "Mechanising and verifying the WebAssembly specification". In: *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM. 53–65. DOI: 10.1145/3167082.

Weirich, S., B. A. Yorgey, and T. Sheard. 2011. "Binders unbound". In: *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP 2011)*. ACM. 333–345. DOI: 10.1145/2034773.2034818.

Wenzel, M. 2006. "Structured Induction Proofs in Isabelle/Isar". In: *Mathematical Knowledge Management*. Berlin, Heidelberg: Springer. 17–30. DOI: 10.1007/11812289_3.

Wenzel, M. 2007. "Isabelle/Isar–a generic framework for human-readable proof documents". *From Insight to Proof–Festschrift in Honour of Andrzej Trybulec*. 10(23): 277–298.

Wenzel, M. 2012. "Isabelle/jEdit – A Prover IDE within the PIDE Framework". In: *Intelligent Computer Mathematics*. Berlin, Heidelberg: Springer. 468–471. DOI: 10.1007/978-3-642-31374-5_38.

Wenzel, M. 2013a. "PIDE as front-end technology for Coq". *CoRR*. abs/1304.6626.

Wenzel, M. 2013b. "Shared-Memory Multiprocessing for Interactive Theorem Proving". In: *Interactive Theorem Proving: 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*. Berlin, Heidelberg: Springer. 418–434. DOI: 10.1007/978-3-642-39634-2_30.

Wenzel, M. 2014. "Asynchronous User Interaction and Tool Integration in Isabelle/PIDE". In: *Interactive Theorem Proving: 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*. Cham: Springer International Publishing. 515–530. DOI: 10.1007/978-3-319-08970-6_33.

Wenzel, M. 2015. "Interactive Theorem Proving from the perspective of Isabelle/Isar". In: *All about Proofs, Proofs for All*. Ed. by B. Woltzenlogel Paleo and D. Delahaye. Vol. 55. *Mathematical Logic and Foundations*. London, UK: College Publications.

Wenzel, M. 2017a. "Future Prospects of Isabelle Technology". URL: http://sketis.net/wp-content/uploads/2017/11/Copenhagen2017.pdf.

Wenzel, M. 2017b. "Scaling Isabelle Proof Document Processing". URL: http://sketis.net/wp-content/uploads/2017/12/Isabelle_Scaling_Dec-2017.pdf.

Wenzel, M. 2017c. "Visual Studio Code as Prover IDE for Isabelle". URL: https://sketis.net/2017/visual-studio-code-as-prover-ide-for-isabelle.

Wenzel, M. 2018a. "Further Scaling of Isabelle Technology". URL: https://files.sketis.net/Isabelle_Workshop_2018/Isabelle_2018_paper_1.pdf.

Wenzel, M. 2018b. "Isabelle/jEdit". URL: http://isabelle.in.tum.de/dist/doc/jedit.pdf.

Wenzel, M. *et al.* 2004. "The Isabelle/Isar reference manual".

Whitehead, A. N. and B. Russell. 1997. *Principia mathematica to *56*. Cambridge University Press.

Whiteside, I. J. 2013. "Refactoring proofs". *PhD thesis*. University of Edinburgh. URL: http://hdl.handle.net/1842/7970.

Wibergh, K. 2019. "Automatic refactoring for Agda". *MA thesis*. Chalmers University of Technology and University of Gothenburg.

Wiedijk, F. 2001. "Mizar light for HOL Light". In: *International Conference on Theorem Proving in Higher Order Logics*. Springer. 378–393. DOI: 10.1007/3-540-44755-5_26.

Wiedijk, F. 2006. *The Seventeen Provers of the World: Foreword by Dana S. Scott (Lecture Notes in Computer Science / Lecture Notes in Artificial Intelligence)*. Berlin, Heidelberg: Springer-Verlag. DOI: 10.1007/11542384.

Wiedijk, F. 2009. "Statistics on digital libraries of mathematics". English. *Studies in Logic, Grammar and Rhetoric*. (18(31)): 137–151.

Wiedijk, F. 2012. "Pollack-inconsistency". *Electronic Notes in Theoretical Computer Science*. 285: 85–100. DOI: 10.1016/j.entcs.2012.06.008.

Wilcox, J. R., D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. 2015. "Verdi: A Framework for Implementing and Formally Verifying Distributed Systems". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '15*. Portland, OR, USA: ACM. 357–368. DOI: 10.1145/2737924.2737958.

Williams, T., P.-É. Dagand, and D. Rémy. 2014. "Ornaments in Practice". In: *Proceedings of the 10th ACM SIGPLAN Workshop on Generic Programming. WGP '14*. Gothenburg, Sweden: ACM. 15–24. DOI: 10.1145/2633628.2633631.

Wilson, S., J. Fleuriot, and A. Smaill. 2010. "Inductive Proof Automation for Coq". In: *Second Coq Workshop*. Edinburgh, UK. URL: https://hal.inria.fr/inria-00489496.

Wimmer, S. and P. Lammich. 2018. "Verified Model Checking of Timed Automata". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 61–78.

Wirth, N. 1971. "Program Development by Stepwise Refinement". *Commun. ACM*. 14(4): 221–227. DOI: 10.1145/362575.362577.

Woos, D., J. R. Wilcox, S. Anton, Z. Tatlock, M. D. Ernst, and T. Anderson. 2016. "Planning for Change in a Formal Verification of the Raft Consensus Protocol". In: *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs. CPP 2016*. St. Petersburg, FL, USA: ACM. 154–165. DOI: 10.1145/2854065.2854081.

Yang, K. and J. Deng. 2019. "Learning to Prove Theorems via Interacting with Proof Assistants". In: *International Conference on Machine Learning*.

Yang, X., Y. Chen, E. Eide, and J. Regehr. 2011. "Finding and Understanding Bugs in C Compilers". In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '11.* San Jose, CA, USA: ACM. 283–294. DOI: 10.1145/1993498.1993532.

Yi, K. 2006. "Educational Pearl: Proof-directed debugging revisited for a first-order version". *J. Funct. Program.* 16(6): 663–670. DOI: 10.1017/S0956796806006149.

Yu, D. and Z. Shao. 2004. "Verification of safety properties for concurrent assembly code". In: ACM. 175–188. DOI: 10.1145/1016850.1016875.

Z3 Development Team. 2008-2019. "Z3". URL: http://github.com/Z3Prover/z3/wiki.

Zalta, E. N. 2018a. "Frege's Theorem and Foundations for Arithmetic". In: *The Stanford Encyclopedia of Philosophy.* Ed. by E. N. Zalta. Fall 2018. Metaphysics Research Lab, Stanford University. URL: https://plato.stanford.edu/archives/fall2018/entries/frege-theorem/.

Zalta, E. N. 2018b. "Gottlob Frege". In: *The Stanford Encyclopedia of Philosophy.* Ed. by E. N. Zalta. Spring 2018. Metaphysics Research Lab, Stanford University. URL: https://plato.stanford.edu/archives/spr2018/entries/frege/.

Zeller, P., A. Bieniusa, and A. Poetzsch-Heffter. 2014. "Formal Specification and Verification of CRDTs". In: *Formal Techniques for Distributed Objects, Components, and Systems.* Berlin, Heidelberg: Springer. 33–48. DOI: 10.1007/978-3-662-43613-4_3.

Zhan, B. 2016. "AUTO2, a saturation-based heuristic prover for higher-order logic". *CoRR.* abs/1605.07577.

Zhang, H., G. Klein, M. Staples, J. Andronick, L. Zhu, and R. Kolanski. 2012. "Simulation Modeling of A Large Scale Formal Verification Process". In: *International Conference on Software and Systems Process.* Zurich, Switzerland: IEEE. 3–12. DOI: 10.1109/ICSSP.2012.6225979.

Zhao, J., S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. 2012. "Formalizing the LLVM intermediate representation for verified program transformations". In: *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*. 427–440. DOI: 10.1145/2103656.2103709.

Ziliani, B., D. Dreyer, N. R. Krishnaswami, A. Nanevski, and V. Vafeiadis. 2015. "Mtac: A monad for typed tactic programming in Coq". *Journal of Functional Programming*. 25. DOI: 10.1017/S0956796815000118.

Zimmermann, T. and H. Herbelin. 2015. "Automatic and Transparent Transfer of Theorems along Isomorphisms in the Coq Proof Assistant". *CoRR*. abs/1505.05028.

Zucker, J. 1994. "Formalization of classical mathematics in Automath". In: *Studies in Logic and the Foundations of Mathematics*. Vol. 133. Elsevier. 127–139. DOI: 10.1016/S0049-237X(08)70202-7.