PHILOSOPHICAL
TRANSACTIONS A

rsta.royalsocietypublishing.org

Research

Cite this article: Appel AW, Beringer L, Chlipala A, Pierce BC, Shao Z, Weirich S, Zdancewic S. 2017 Position paper: the science of deep specification. *Phil. Trans. R. Soc. A* **375**: 20160331.
http://dx.doi.org/10.1098/rsta.2016.0331

Accepted: 25 July 2017

One contribution of 8 to a discussion meeting issue 'Verified trustworthy software systems'.

Subject Areas:
software

Keywords:
formal methods, programming languages, proof assistants

Author for correspondence:
Lennart Beringer
e-mail: eberinge@cs.princeton.edu

Position paper: the science of deep specification

Andrew W. Appel[1], Lennart Beringer[1], Adam Chlipala[2], Benjamin C. Pierce[3], Zhong Shao[4], Stephanie Weirich[3] and Steve Zdancewic[3]

[1]Department of Computer Science, Princeton University, Princeton, NJ 08540, USA
[2]Computer Science and Artificial Intelligence Laboratory, MIT, 77 Mass Avenue, 32-G842, Cambridge, MA 02139, USA
[3]Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, USA
[4]Department of Computer Science, Yale University, New Haven, CT 06520, USA

LB, 0000-0002-1570-3492; AC, 0000-0001-7085-9417

We introduce our efforts within the project 'The science of deep specification' to work out the key formal underpinnings of industrial-scale formal specifications of software and hardware components, anticipating a world where large verified systems are routinely built out of smaller verified components that are also used by many other projects. We identify an important class of specification that has already been used in a few experiments that connect strong component-correctness theorems across the work of different teams. To help popularize the unique advantages of that style, we dub it *deep specification*, and we say that it encompasses specifications that are *rich*, *two-sided*, *formal* and *live* (terms that we define in the article). Our core team is developing a proof-of-concept system (based on the Coq proof assistant) whose specification and verification work is divided across largely decoupled subteams at our four institutions, encompassing hardware microarchitecture, compilers, operating systems and applications, along with cross-cutting principles and tools for effective specification. We also aim to catalyse interest in the approach, not just by basic researchers but also by users in industry.

This article is part of the themed issue 'Verified trustworthy software systems'.

THE ROYAL SOCIETY
PUBLISHING

1. The need for deep specifications

Modern hardware and software are monstrously complex. The best tool for coping with this complexity is *abstraction*—i.e. breaking up functionality into *components* or *layers*, with *interfaces* that are as narrow and clear as possible. Each interface is accompanied—implicitly or explicitly—by a *specification* expressing the contract between providers and consumers of that interface. These specifications come in a multitude of different forms: comments in code and natural-language documentation, unit tests, assertions, contracts, static types, property-based random test suites and formal specifications in various logics.

Sadly, despite widespread agreement on the importance of abstraction, specifications are often seen as an afterthought, or even a hindrance, to system development. Why? Experience has shown that it is extremely challenging to write good ones. Indeed, a maximally useful interface specification must be simultaneously *rich* (describing complex component behaviours in detail), *two-sided* (connected to both implementations and clients), *formal* (written in a mathematical notation with clear semantics to support tools such as type checkers, analysis and testing tools, automated or machine-assisted provers, and advanced IDEs (integrated development environments)) and *live* (connected via machine-checkable proofs to the implementation and client code). We call specifications with all of these properties *deep specifications*.

Most present-day interface specifications fall short in one or more of these dimensions. In many programming environments, the machine-checkable parts of an interface are just type declarations that specify the shapes of the inputs and outputs of a component. Richer aspects of the component's behaviour are either explained in comments (which are unconnected to the code and may or may not be kept up to date) or left entirely implicit. Some environments allow interfaces to be augmented with simple pre- and post-conditions or unit tests. These can help enormously, but they are limited in the properties that they can express. Other environments offer richer capabilities for writing down specifications—e.g. formal languages like Z [1], Alloy [2] or AADL [3]—but these are generally concerned with specifying properties of system *models* rather than the actual software under development. The paucity of deep specifications in mainstream software engineering imposes a mental tax on programmers that discourages innovation.

Fortunately, in the research community the situation is rapidly changing, spurred by recent progress in the language of specifications (new techniques for specifying semantics, availability of higher-order logics), the methods of connecting them to programs (new proof-search algorithms and decision procedures, expressive logical frameworks) and the adoption of programming models that are easier to reason about (on the one hand, functional programming languages with optimizing compilers and high-performance garbage collectors; on the other hand, streamlined subsets of C with clean proof theories). These advances, driven by the maturation of automatic and interactive theorem-proving technology, suggest that it is now possible to radically alter the state of the art. The result is that technologies for automatic or machine-assisted program verification—once believed to be completely impracticable [4]—are now becoming widespread.

A compelling demonstration of the viability of machine-verified development of real systems is Leroy's CompCert project [5], which specified, implemented and proved the correctness of an optimizing C compiler. CompCert employs many characteristics of deep specifications: (i) The extremal languages and also all compiler-intermediate languages are equipped with precise operational semantics, expressed as inductive relations in the Coq proof assistant. These rich internal interfaces structure the development. (ii) Each transformation is programmed in Coq's functional language *Gallina* and proved to preserve safety and functional correctness with respect to its two enclosing interfaces. Proofs are constructed interactively as proof scripts in Coq's vernacular, making use of tactics and other automation features. (iii) A running compiler is obtained by composing the Gallina functions of all transformations, extracting the resulting code into OCaml and using the standard OCaml compilation framework. (iv) The proof scripts yield formal proof objects in a variant of the calculus of inductive constructions (CiC) for which checking of proofs amounts to type checking and is *fully automatic* and independent of the original proof scripts.

CompCert is no toy: it compiles essentially the whole ISO C99 language, targets several architectures and achieves 90% of the performance of GCC's (Gnu Compiler Collection) optimization level 1. The value of CompCert's correctness proofs has surprised some observers. For example, John Regehr's Csmith project [6] used random testing to assess all popular C compilers, and reported: 'The striking thing about our CompCert results is that the middle-end bugs we found in all other compilers are absent. As of early 2011, the under-development version of CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying: we have devoted about six CPU-years to the task. *The apparent unbreakability of CompCert supports a strong argument that developing compiler optimizations within a proof framework, where safety checks are explicit and machine-checked, has tangible benefits for compiler users*'.

The principles behind CompCert apply broadly to effective verification. Similar successes have been achieved in several domains. In *hardware*, the SAFE project [7] has formalized security properties of a hardware architecture with dynamic information-flow tracking, while Peter Sewell and collaborators at the University of Cambridge have developed and tested formal models for x86, Power and ARM architectures with relaxed memory models [8,9]. In *compilers*, the Vellvm project [10] formalized the LLVM compiler's intermediate languages and proved correctness of some key optimizations. In *language semantics*, researchers have developed formal specifications of most of JavaScript [11] and ML [12]. In *operating systems*, the CertiKOS project [13–16] has built a new, fully verified and secure hypervisor kernel including formal specifications not just of the system-call APIs (application programming interfaces) but also of all of the kernel's internal abstraction layers, while the seL4 project has specified and verified both functional correctness and security properties of a microkernel in the popular L4 family [17,18]. In *networking*, there has been work to verify TCP/IP [19] and Software-Defined Networks [20]. And at the *application layer*, we have formalized and verified simple Web servers [21], and others have done work on formally specifying and verifying relational databases [22], Web browsers [23,24], file systems [25,26] and distributed systems [27].

(a) The DeepSpec hypothesis

Based on this flowering of work, we contend that the technology of deep specification is ready for wider adoption:

> Development of deeply specified computational artifacts with machine-checked proof scales from individual components to deployable systems. By integrating components at their specification interfaces, high-assurance hardware and software systems can be obtained in a flexible and reusable manner, often without unduly compromising performance, and can be integrated into existing and emerging development methods.

Validating this hypothesis requires careful re-evaluation of modelling and specification approaches employed in existing developments (including the ones mentioned above), refining existing interfaces and complementing them by specifications of novel components, validating all interfaces by verifying suitable implementations and clients.

DeepSpec (*The Science of Deep Specification* (https://deepspec.org), an 'Expedition in Computing' funded by the National Science Foundation) is carrying out this programme for interfaces and components of a typical high-assurance system, comprising an operating-system kernel, processor implementations, cryptographic libraries and applications, using the Coq proof assistant. Our specifications are primarily concerned with functional correctness, but intermediate interfaces and formal proof development routinely reveal additional invariants that amount to security or isolation guarantees. Along the way, we are identifying design principles and implicit contracts between components, and between the deployed components and the formalisms and development tools involved in their production: programming languages, compilers, testing methodologies.

Our aim in this paper is twofold. First, we want to set down in one place the argument for why 'the time is now' for widespread use of deep specifications, addressing reservations that are often articulated by computing experts who are not specialized in formal methods, with answers that are often well known within our narrower community but not outside it. Second, having made the case that this intellectual approach deserves a concerted implementation (and theoretical) effort, we outline (in §5) our team's concrete plans.

2. Motivation and objectives

To establish industrial viability of deep specifications, one needs to meet three challenges.

First, one must demonstrate that key interfaces between exemplary components of high-assurance systems can indeed be specified in such a way that producers as well as clients of an interface can be verified with moderate effort, and that thus-specified components compose to a reasonably performant system with an externally meaningful correctness guarantee.

Second, the principles applied, insights generated and development tools used during the construction of verified artefacts, like programming languages, compilers, processor-description frameworks, must themselves be equipped with mathematical precision; it is the soundness of these tools (with respect to the semantics of the formalisms they implement) that enables the progression from an individual success story (a verified stack) to an engineering discipline where novel artefacts can be predictably developed based on existing components, standardized methodologies and a well-educated workforce. To integrate deep specifications into industrial practice, it is mandatory to balance introduction of radically novel development approaches with efforts that enhance existing development and validation flows. This includes the verification of legacy code and the integration of code bases of different provenance.

The final challenge is to narrow the education gap that separates researchers from practitioners. On the one hand, this means to develop teaching materials that can be used in university lectures, industrial training courses and self-study, and to organize topical summer schools and workshops with industrial participation. On the other hand, practical knowledge must be transferred from industry to academia, for example, by visitor programmes, student placements or by inviting industrial researchers to highlight verification challenges and integration hurdles. Quite a few industrial practitioners nowadays appreciate formal verification, having already been exposed to mostly automated techniques in their daily practice and often to modern functional programming languages during their academic education.

(a) Benefits

The obvious and immediate benefit of a deeply specified and verified system is that certain aspects of its behaviour are established beyond reasonable doubt, subject to appropriateness of the externally visible specifications and modelling assumptions.

Beyond establishing the absence of implementation deficits, the true value of deep specifications lies in the ability to articulate and popularize inherently meaningful system models and reusable interface contracts, to relate such models across multiple abstraction layers in a single formal framework, and to systematically obtain or preserve guarantees concerning the various artefacts from the soundness properties of the development tools (e.g. compilers).

Modern secure system design advocates the use of hypervisors and cryptographic mechanisms to isolate mutually untrusted parties. Formal verification using deep specifications complements these techniques when applied to application code, but at the same time helps to increase our confidence in their implementations.

(b) Compositionality

The key structuring principle of deeply specified system design is compositionality, along horizontal and vertical axes.

Horizontal compositionality concerns the compatibility of multiple components that interact dynamically, despite being developed independently. Thus, a deep specification must enable the implementation of components at either side: clients *cannot* assume more than is specified, and implementations *cannot* fail to guarantee all that is specified. The closer components interact, the less clear the distinction between client and implementation becomes; interface specifications in these cases amount to semantically enhanced protocol descriptions.

Vertical compositionality concerns the ability to bridge multiple abstraction layers, as this allows reasoning about model-level properties to be treated independently from implementation aspects. Vertical compositionality enables top-down development by refinement or bottom-up development by abstraction, and is typically organized in layers or along the boundaries between different programming languages or modelling formalisms.

We illustrate the capability of proof-assistant-based specification to integrate horizontal and vertical composition using two developments from our participation in the HACMS project [28].

Our first example concerns the structuring mechanisms that allowed us to develop CertiKOS [13–16], a highly extensible certified OS kernel with support for multicore and multithreaded concurrency, fine-grained locking, kernel-level interrupts and verified device drivers, and virtualization for booting multiple Linux virtual machines on different CPU cores. The CertiKOS effort insists on using compositional programming language features, compositional semantics, compositional linking mechanisms, compositional verified compilers, compositional concurrency constructs and compositional program logics. The key idea is to decompose the specifications, the semantics and the proofs of any large, complex (software and hardware) system using the new language construct known as *certified abstraction layers* [13,16]. These layers correspond to traditional notions of components, but are certified, meaning that they come with formal deep specifications and proved implementations. They can be composed horizontally and vertically; they also support concurrency (i.e. general parallel composition), and can be compiled from one implementation language (e.g. C) into another (e.g. assembly). CertiKOS uses contextual refinement [13,29,30] as a unifying mechanism to support certified linking and prove end-to-end functional correctness properties. It also uses *fully abstract* deep specifications so security properties proved at higher abstraction levels can be propagated to the actual low-level implementation [15].

Our second example is the verification of legacy cryptographic library code, which is partitioned into code units corresponding to different primitives. Specifically, we used the Verified Software Toolchain (VST) [31] to specify the C-level API of OpenSSL's code module for the HMAC (keyed-hash message authentication code) cryptographic primitive and the subordinate module for hash function SHA-2 (secure hash algorithm), and used the Verifiable C program logic to verify their respective implementations [32,33]. In ongoing work, we validate in turn two clients of HMAC: HKDF (HMAC-based key derivation function) and HMAC-DRBG (deterministic random bit generator). These API specifications describe the functional behaviour of the C code by reference to Gallina functions codifying the appropriate NIST standard but do not express any cryptographic properties. The verification of cryptographic security (indistinguishability of HMAC from a random oracle by an attacker with polynomially bounded computational power, subject to the appropriate cryptographic assumptions on SHA256) is carried out at the model level, using the Foundational Cryptography Framework (FCF) [34], an independently developed domain-specific tool for reasoning about probabilistic programs and their transformation. The proofs of implementation correctness and of cryptographic security are provably connected, but were carried out by independent teams, neither of which needed in-depth familiarity with the other team's reasoning formalism. These teams could work independently because the interface (between implementation-level proofs and theoretical-crypto proofs) is the *deep specification* of HMAC+SHA.

DeepSpec's premise is that separation into model-level and code-level reasoning is not restricted to cryptographic code, that the principles codified in certified abstraction layers are applicable to many systems, and that horizontal and vertical compositionality is a key enabler for scaling verification.

3. Reservations and concerns

(a) Technical feasibility

Until recently, the goal to verify functional correctness of modern systems might have been considered unrealistic or simply technically out of reach. However, as demonstrated by the artefacts described above, substantial progress has been achieved over the past few decades. One contributor to this progress, of course, has been the steady advance in hardware performance, in terms of both computation and storage, that has been sustained for over half a century. Enabled by this computing power, another key factor has been the development of robust proof assistants with features such as program extraction and tactics-based programming. The expressiveness of theorem provers based on higher-order logic and dependent type theory has been put to use by new technical developments, such as a better understanding of how to define operational semantics and new reasoning principles like separation logic.

The feasibility of DeepSpec-scale verification efforts also depends on the social context of such work, which has also changed in recent years. One such social factor is the acceptance that the pervasiveness of modern IT infrastructure necessitates a concerted effort to improve system quality—given the ubiquity of security vulnerabilities and other defects in present-day systems, the need for high-assurance computing has never been greater, and, increasingly, stakeholders are recognizing its importance.

Another social factor is the perception that deep verification can be successful. CompCert and seL4 provide compelling demonstrations that proof assistants have matured to the point where it is possible to tackle such projects, which in turn has spurred community and student interest. In particular, CompCert highlighted the benefit of program extraction, and of verification-aware structuring of software using deeply specified internal interfaces (which in the case of CompCert are several intermediate languages, each equipped with a formalized operational semantics). As a consequence, increasing numbers of graduate students are interested in formal verification, and they often arrive with significant experience in systems building.

(i) Where do specifications come from?

Despite their shortcomings, RFCs (requests for comments), standards documents, user manuals, programmer comments, man pages or textbooks contain a wealth of information that an initial specification may be based on. These documents establish linguistic and conceptual frameworks that capture existing understandings by diverse stakeholders. By either consciously building upon or explicitly deviating from these artefacts, deep specifications either strengthen or modify prior understandings. Large-scale experimentation with deployed implementations to calibrate formal specifications against observed behaviour has been successfully used by the REMS project on numerous occasions [35]. Typically, these efforts suggest refinements or reformulations of the informal descriptions even if no formal verification is carried out, and are therefore valuable even to the existing stakeholders. Occasionally, specification layers or new conceptual abstractions emerge during the verification process by gradual solidification of notational shorthands or proof patterns. The ability to codify arbitrary concepts as solid abstraction boundaries and equip them with their own reasoning principles (including automation features that avoid unfolding definitions to lower-level concepts) is a distinguishing feature of the higher-order logics and type theories implemented in modern proof assistants.

(ii) What if specifications are wrong?

Specifications are just another kind of software, so they are also prone to programmer mistakes and oversights. Shortcomings that make a specification unsatisfiable are typically identified during verification, and they highlight implementation errors or (sometimes major) misunderstandings regarding the component's purpose or structure.

A specification typically describes the interface between two components, or between two abstraction layers. Suppose one proves that module A correctly implements specification S_A. This already ‘shakes out’ many problems in both A and S. Next one tries to prove that module B, a client of A, satisfies its own specification S_B, relying on services specified in S_A. But typically one finds that the properties guaranteed in S_A are too weak for client use; one revises S_A into S'_A, re-proves A against S'_A, and so on. This final specification is *two-sided*: it has been demonstrated both to be *implementable* (by A) and *useful* (to B).

Conversely, suppose one starts with a specification $S^\dagger_A$ developed during the verification of B, before the $A : S^\dagger_A$ proof is done. Typically, one finds that $S^\dagger_A$ is unsatisfiable, or at least not satisfied by A. Again, $S^\dagger_A$ is not a *two-sided* spec until the verifications of both A and B against it.

Technical choices can help to reduce specification mistakes, or to reduce their impact. For example, Gallina, the language in which substantial parts of our specifications are written, is a functional language with a clean proof theory, allowing specifications to be written declaratively while largely ignoring aspects of efficiency. Execution inside the proof assistant and property-based testing help to debug specifications early, and program extraction to OCaml, Haskell or Scala allow the early integration of functional models into surrounding code fragments.

Even slightly incorrect specifications can be valuable, as they help the multiple stakeholders to discuss their expectations using a precise vocabulary. Indeed, to avoid over-specialization, *two-sided* evaluation of interfaces should often be generalized to *multi-sided*, i.e. to validation by multiple implementations and clients. And whenever the specification is tweaked to support another implementation or client, its consistency with the existing code base is preserved, or is re-established by mechanically guided code or proof repairs. Moreover, even incorrect specifications often lead a system designer to structure code more clearly.

Finally, incorrect specifications can be tolerated as long as they are not exposed externally: if the top- and bottom-layer specifications are correct, an end-to-end correctness argument causes the dependences on the intermediate-layer specifications to drop out—their correctness is irrelevant to the larger claim. For example, the question whether *Clight*—the interface between VST and CompCert—faithfully adheres to the ANSI C standard is irrelevant for the end-to-end guarantee of the VST–CompCert toolchain. Similarly, in our correctness proof of the HMAC cryptographic primitive, the adherence of the Gallina programs for HMAC and SHA256 to the respective NIST standards drops out of the TCB (trusted code base), the relevant property being that the code calculates *some* function that satisfies the claimed cryptographic security property.

Of course, specifications may be externally inappropriate if the desired property relies on an aspect that is simply not captured by the model represented in a formalization—for example, timing side channels: the more detailed a model is, the richer specifications it permits.

(iii) Why higher-order logic? Why Coq?

In order to establish guarantees ‘beyond reasonable doubt’, and do so modularly, it is of paramount importance that the formal framework is sufficiently powerful to describe component behaviour in depth, flexible enough to support a variety of modelling approaches, founded in mathematical logic and practical enough to seamlessly tie specifications to implementations. Formal verification can take many shapes, so the framework should support a variety of modelling styles, and provide convenient ways of working with ‘standard’ mathematical abstractions, such as numbers, sets (finite), maps, relations, lists, tuples. Moreover, it should support user-defined abstractions, for example, to permit the first-class manipulation of programs and other component representations. From these ingredients, we expect to build virtually arbitrary domain-specific models and reasoning principles.

The Coq proof assistant satisfies all these requirements by equipping a foundational logic (the dependent type theory of the *calculus of inductive constructions* [36]) with a fully fledged functional programming language (Gallina) and expressive means for developing proof scripts and tactics. Coq is not the only proof system that could be used—other contenders include Isabelle/HOL and ACL2—but offers a number of distinct features that we have found useful.

In particular, Coq’s type theory with facilities for (co-)inductive-dependent data types supports abstraction and parametrization using a variety of mechanisms, including parametric polymorphism, modules and functors. These facilitate the creation of generic libraries and re-usable theories while maintaining a clean separation of object language versus metalanguage. Coq permits extraction to efficient executable code and effective automation using Ltac and type class resolution. Together, these characteristics allow proof engineers to realize the 'live' criterion of a deep specification, which requires a close link between specifications and specified artefacts.

We have also found that HOL/type theory is better suited for many specification purposes than first-order logic; partly because higher-order specifications themselves can be more succinct (and proof terms for theorems encoded in first-order logic can be exponentially larger than their higher-order counterparts), and also because the ability to quantify over higher-order objects allows for better factoring, for instance, to permit expressive concurrency specifications in the VST.

Another boon of adopting Coq is the excellent social ecosystem surrounding the tool. There has been great support by the Coq team (i.e. continued development, maintenance and improvement of the Coq tools) and a continuously increasing user base even aside from the DeepSpec project. This lively user community is important for recruiting personnel to our own efforts (there is a growing pipeline of Coq experts), for disseminating our results to a large community that can appreciate and re-use them, and for making our students employable upon PhD graduation.

(iv) Prior work on pervasive system verification

At the opposite end of the technical feasibility spectrum is the question of *novelty*. In 1989, Computational Logic, Inc. verified a hardware/software stack that included a gate-level description of a microprocessor, an assembler, linker, loader, two verified compilers, a (highly idealized) operating system and some applications, with theorems connecting all these components formulated and proven in Nqthm [37]. Obviously, system stacks in 2017 are orders of magnitude more complex than those in 1989, and the need to modernize or re-examine many of the formalizations was in fact proposed by Moore himself [38]. Although it is not entirely inconceivable that such complexity could still be handled by a modern Boyer–Moore theorem prover such as ACL2, we believe that the above-discussed features make Coq better suited to manage these complexities.

More recently, the VeriSoft project combined proofs of processor and compiler correctness to a verified stack linking a sequential subset of C to a gate-level description of the VAMP processor. The textbook [39] describes material related to this project, and an article [40] outlines subsequent work on memory invariants at the interface between a multicore hypervisor and C compilers.

The same team [41–43] also applied the VCC framework [44] to formally verify Hyper-V, a widely deployed multiprocessor hypervisor by Microsoft consisting of 100 kLOC (thousand lines of code) of concurrent C code and 5 kLOC of assembly. However, only 20% of the code is verified [44]; it is also only verified for function contracts and type invariants, not full functional correctness.

seL4 represents a modern OS microkernel, but is less modularly structured than CertiKOS and does not support multicore concurrency with fine-grained locking. The abstract and executable specifications in seL4 are 'monolithic' as they are not divided into layers to support abstraction across different kernel modules. These kernel interdependences led to more complex invariants, which may explain why the seL4 proof effort took 11 person years.

Sewell *et al.* [45] used translation validation to build a refinement proof between the semantics of the verified C source code and the corresponding binary (compiled by GCC). This proof was not done in a proof assistant (thus it has no machine-checkable proof), and the translation validator itself still has not been verified. The assembly code (600 lines) in seL4 still remains unverified.

Hawblitzel *et al.* [46] have recently developed a set of new tools based on the Dafny verifier [47] and Z_3 SMT solver [48], and applied them to build their Ironclad system, which includes a verified kernel (based on Verve [49]), verified drivers, verified system and crypto libraries, and several applications. Ironclad, however, only proves the partial correctness property (at the assembly level), which is weaker than the total correctness properties proved by seL4 and CertiKOS.

(b) Economic feasibility

(i) Costs and trade-offs

Constructing specifications and proofs for complex software and hardware increases the directly incurred development time and requires a suitable workforce. Putting a precise cost on this effort is as difficult as is the case for traditional validation methodologies, but article [18] of this volume contains some current estimates from the seL4 project.

In the hardware community, Intel's 1994 Pentium FDIV bug resulting in a write-off of $475 million [50] is generally considered as the event that brought formal verification into industrial hardware design process. ARM's recent publication of a formal specification of the ARMv8.2 processor [51] represents a natural continuation of this development and gives industrial credence to our anticipated verification of the RISC-V microarchitecture (cf. §5). It also emphasizes that the costs of verification needs be weighed against the costs of other techniques. For instance, unit testing is an expensive and labour-intensive practice that is nevertheless standard procedure for modern software engineering. Verification, by contrast, yields much stronger guarantees than typical testing approaches alone (even fuzz testing), and can be used in cases where random testing is not applicable, such as the validation of random number generators.

As verification costs drop, the first application domains where it will be economically feasible will be *safety-critical systems* and *commodity Internet-facing infrastructure*.

Safety-critical systems are systems in which failure might lead to extraordinarily expensive or catastrophic outcomes such as loss of life. Typical examples include medical devices and critical infrastructure components: power plants, energy grids, communication networks.

Of course, the judgment about whether a certain device or application is critical changes over time, and, as computing technology becomes more widely integrated, the set of critical applications changes too. As one example, recent changes in the transportation industry, whether rail, air or automotive, have made their computing infrastructure more critical [28]. As Kohno, Savage, Miller and others have demonstrated, an individual vehicle is now a network of around 40 computers connected to the Internet via cellular radio; and its software (like almost all software) is riddled with security vulnerabilities. This danger will only increase as (semi-) autonomously operated vehicles become widespread. Volkswagen's diesel fiasco demonstrates the vulnerability of entire business models to testing and specification loopholes (and social factors) [52].

Commodity Internet-facing infrastructure, such as components of TCP, TLS (transport-layer security), Web servers, Web browsers, the operating systems that run them, and the compilers that compile them, are all vulnerable to attack from sophisticated criminal (or nation-state) (or criminal nation-state) attackers anywhere in the world. But because they are heavily standardized and widely used components, design-time verification amortizes over millions of deployments. Processors, operating systems and cryptographic libraries are designed and implemented by comparatively few but highly specialized programmers, but used by millions of consumers.

As with critical infrastructure, the kinds of such components that exhibit economies of scale are changing over time. For instance, currently any Internet user benefits from verified cryptographic libraries, but in the future, the 'Internet of things' (IoT) might allow millions of devices to benefit from verified light-weight hypervisors. It should also be noted that deep specifications exhibit network effects: as more verified components and systems exist, the more valuable the next ones get, because they can rely on stronger guarantees provided by their environments and therefore provide stronger guarantees themselves.

(ii) Social context

Incidents such as Heartbleed [53] and the Snowden revelations have made security and privacy a front-page newspaper issue. Recent changes in US policy regarding browsing history triggered an analysis by the *New York Times* on the practical usability of VPNs. These and other security-related incidents have highlighted the economic costs of vulnerable infrastructure. The resulting legal or financial pressures, together with growing consumer awareness, make it increasingly likely that software companies will replace the question ‘Can I afford to verify my software?’ with ‘Can I afford **not** to verify (at least the critical parts of) my software?’.

Amazon’s recent article on the use of TLA+ describes the value proposition thus [54]:

> In industry, formal methods have a reputation for requiring a huge amount of training and effort to verify a tiny piece of relatively straightforward code, so the return on investment is justified only in safety-critical domains (such as medical systems and avionics). Our experience with TLA+ shows this perception to be wrong. At the time of this writing, Amazon engineers have used TLA+ on 10 large complex real-world systems. In each, TLA+ has added significant value, either finding subtle bugs we are sure we would not have found by other means, or giving us enough understanding and confidence to make aggressive performance optimizations without sacrificing correctness.

Our conversations with Google’s applied cryptography team and the existence of Microsoft Research’s Everest project [55] confirm: large corporations are re-evaluating the traditional cost–benefit ratio, for the components they consider critical to their business models.

Regarding the costs to find, recruit and retain programmers with the needed expertise, it is true that, even now, most computer-science undergraduate students do not gain much (if any) exposure to the tools and methodologies needed to do DeepSpec-style verification. However, Amazon’s findings suggests that traditional reservations are becoming increasingly obsolete [54]:

> Amazon now has seven teams using TLA+, with encouragement from senior management and technical leadership. Engineers from entry level to principal have been able to learn TLA+ from scratch and get useful results in two to three weeks, in some cases in their personal time on weekends and evenings, without further help or training.

The attendance numbers of our own courses, and the uptake of Pierce’s *Software Foundations* [56] and Nipkow & Klein’s *Concrete Semantics* [57] at numerous institutions, indicate the eagerness of students and educators to acquire formal-methods skills, which will further reduce costs for industrial (re)training over time. Integrating our research findings into the university curriculum will support this development by highlighting the scalability of formal techniques.

(c) Limitations and assumptions

Our methods operate literally on the AST (abstract syntax tree) representations of programs that are converted by high-level compilers to assembly code, or to RTL (register-transfer language) code in the case of processor descriptions. We thus eliminate gaps that arise from repeated pretty-printing and parsing in traditional tool flows and in methodologies that couple verification tools and code generation tools less tightly.

Our results are relative to the appropriateness and precision of the mathematical model (like operational semantics, etc.) of the bottom-most system description layer, and of policies and other guarantees at higher levels, both as represented in the Coq proof assistant. Focusing primarily on functional correctness, our models largely ignore aspects of (concrete) execution time and energy consumption, and numerous other aspects of program execution on physical devices. As is the case for validation by simulation, our processor specifications concern design-time claims and hence rest on hypotheses regarding the fabrication, packaging and procurement of hardware and the physical environment (temperature, exposure to radiation, etc.) in which a device is used. Aspects of these physical factors can be captured formally (and to some extent be experimentally validated by testing), but doing so is beyond the scope of our work.

For further discussions on the epistemic nature of formal verification using interactive proof assistants, for software, hardware and mathematics, the reader is referred to the articles by Pollack [58] and Asperti *et al.* [59], and the contributions by Harrison, Hales, Gonthier and Wiedijk to the *Notices of the American Mathematical Society’s* Special Issue on Formal Proof [60].

(i) What is the TCB? What about bugs in Coq?

The Coq proof assistant is a collection of algorithms for constructing, maintaining and checking proofs in the *calculus of inductive constructions*, and is largely implemented in OCaml. Exploiting the Curry–Howard isomorphism, Coq's TCB is small, containing a proof checker for CiC but not the components that help the user construct proofs. Although the proof checker has not been fully immune against programming mistakes, the majority of bugs have affected non-critical components or have affected efficiency rather than soundness. Of course, executing the proof checker requires correctness of the OCaml compilation environment (compiler, OS, processor, etc.).[1] To increase confidence, a sceptical user may implement a stand-alone proof checker in his programming language of choice, as was done for Twelf [61]. The metatheory of Coq was studied by Werner [62] in 1994; extending this work to more recent features would be very valuable, but is beyond the scope of our plans.

Of the specifications, only the bottom-most machine model of a refinement stack is in the TCB, plus the axioms or hypotheses referred to in a proof development and the definitions of security properties. As discussed earlier, intermediate specifications like the operational semantics of Clight or the RISC-V ISA (instruction-set architecture) do not form part of the TCB: two-sided verification turns them into auxiliary intermediate lemmas of an end-to-end proof.

(ii) Are some components left unverified? What's the bottom-most verification layer?

Our exemplary system stack omits formalization of the EDA (electronics design automation) tool chain that translates RTL to gate-level descriptions with placement and routing information. We plan to run the network stack in an unverified Linux 'guest' virtual machine (VM) of CertiKOS, isolated from verified components by CertiKOS's provably correct virtual memory protection. It can communicate with other VMs via IPCs (interprocess communications) or hypercalls. Thus, we will isolate network functionality from other clients and from the CertiKOS kernel but not formally verify its functional correctness.[2] We will apply the same approach for other software packages—and for our own components at intermediate stages of verification—but also seek to cooperate with external researchers to coordinate specification interfaces of additional components like linkers, device drivers, or other libraries.

(d) Comparison with automatic analysis techniques

Many of our deep specifications exercise features of higher-order logics: quantification at higher types, alternating quantification, impredicative quantification, recursive types, dependent types. That is why we use *interactive proof assistants* such as Coq, in which the user *builds* the proof and the system *checks* the proof (and provides programmable automation to *help* build the proof).

Other approaches to formal verification focus on *automation*, trading away expressiveness. These methods—model checkers, SAT/SMT solvers, equivalence checkers, and non-foundational[3] tools—have made significant advances in recent years and greatly help to reduce the number of bugs in software, or prove that software satisfies shallow specifications—e.g. safety properties short of functional correctness. In many cases, deep specifications include properties that are shallow enough to prove or model-check by such tools. In those cases we certainly encourage their use, especially when it leads to larger programs being proved with less human effort.

[1] And of the compilation environment used to obtain the OCaml compiler, and so on . . ..

[2] If TLS is verified, then meaningful interoperation with a verified server (via an insecure network layer) is achieved via (verified) cryptographic message authentication.

[3] A *foundational* program-verification tool is one whose implementation has been formally proved sound using a machine-checked proof. A *non-foundational* program-verification tool is a computer program lacking such assurance.

But it is important to recognize the limitations: The superior degree of automation of these techniques is often achieved by restricting the applicability to a certain domain of interest, prescribing topical modelling styles, or hard-wiring implicit assumptions into the verification algorithm. Typically, the implementations of these algorithms are not validated by soundness proofs formalized in an interactive proof assistant, although counter-example production and witness generation formats are nowadays routine. However, a witness is not the same as a proof term in λ-calculus, particularly as modern SAT/SMT engines subject formulae to numerous sophisticated manipulations before attempting to find UNSAT cores. We will be keen clients of tools resulting from recent progress on foundational witness checkers in Coq [63].

In order to be amenable to decision procedures, specification formalisms supported by (fully) automated verification tools are of limited expressiveness, lacking, for example, the higher-order reasoning capabilities that enable modularity. Much of the proof engineer's work (and scientific progress) goes into identifying sophisticated encodings, system parametrizations, or domain-specific abstractions. In contrast, dependent type theory provides a framework in which such encodings and abstractions become first-class entities that can be manipulated, composed, and—most importantly—justified in a foundational manner.

These shortcomings notwithstanding, fully automated technologies offer numerous insights that proof-assistant-based verification can seek to replicate, or to integrate using foundationally verified witness checkers such as Isabelle/HOL's Sledgehammer. A recent example that may provide concrete insight into the capabilities and limitations of both approaches is the verification of file systems, in the form of FSCQ in the world of Coq-based verification [25] and of Yggdrasil in the world of automated techniques [64].

4. Evolution or revolution?

Coq is an excellent logic and language for proofs about programs; but in what languages should the programs themselves be written? Should new proof methods be applied to old languages? Or can proofs and programs be synthesized together from high-level specifications in novel languages? We believe that both approaches have important applications.

(a) New logics for old languages

Several of the DeepSpec projects (and affiliated projects elsewhere) support program development in the C language. The benefits of using C are as follows.

- The semantic specification has been developed and studied for decades, and since 2006 has been well formalized in Coq.
- There is at least one proved-correct optimizing compiler for C. CompCert is proved correct in Coq. Our *Vellvm* project formalizes internal components of a widely used open-source C (and C++, and Haskell) compiler. As another approach, the seL4 project in Australia developed formalized (in HOL) translation validation for conventional C compilers such as GCC.
- This formalized semantics serves as an excellent abstraction boundary between the source program and the compiler. That is, the source programmer can take advantage of decades of research in compiler optimizations (flow analysis, register allocation, instruction selection, function inlining, etc.) without any of this being visible in the source-code-level proofs about the program. Synthesis methods that directly generate machine language lose this benefit.
- C programs can interact with other components using industry-standard interfaces.
- C is a general-purpose language that can support almost any application.
- Finally (and we have already noticed this in practice), there can be a division of labour: software engineers can write C programs with good performance, and verification engineers can do the specification and verification. Both engineers can view and discuss

the functional specification, but the C programmers do not have to know how to do proofs.

But what kind of program logic should be used to prove correctness of C source programs? We are trying two approaches.

Verifiable C is a Hoare logic for C programs, actually a higher-order impredicative concurrent separation logic. One writes specifications of C functions, using preconditions and postconditions; one writes loop invariants; and then one does interactive proof in Coq that the programs satisfy their specifications. The VST-Floyd proof-automation system assists in the interactive proof.

Certified Abstraction Layers treats C programs as refinements of functional programs. One writes a functional program (in Gallina) operating on an abstract state, and a C program that operates on concrete C data structures. Then, function by function or module by module, one proves refinement theorems. This approach is quite modular, allowing both 'vertical' and 'horizontal' composition.

Property-based testing is an alternative when full proofs are not cost-effective. It can substantially increase specificity by exposing the device under scrutiny to automatically generated tests that are not only syntactically admissible but also capture actual client use cases. Our *QuickChick* project extends existing smart test case generation (as implemented in QuickCheck) to Coq specifications. Where full proofs *are* desired, QuickChick can reduce the development time, as specifications can be scrutinized and debugged before formal verification is even attempted.

(b) New logics deserve new languages

Still, the languages of the 1970s can be clumsy in expressing the algorithms we want to implement. We should take advantage of 21st-century developments in functional programming, advanced type systems, process calculi and domain-specific languages. All these developments lead to languages with better proof theories, leading to simpler, easier and more automatable proofs.

The logical and programmatic features of a modern proof assistant facilitate the machine-checked realization of virtually all approaches to abstraction and refinement that have been developed over the past decades. Hence we can radically change how software or hardware is structured or developed. Examples for potentially disruptive design methods include code/proof cosynthesis as exemplified by a number of ongoing projects by the DeepSpec team.

- Hardware design: While hardware engineers traditionally write code modules and then 'throw them over the wall' to separate verification engineers, our Kami framework [65] facilitates a workflow where hardware modules are coded within Coq in the first place, to be proved against specifications as they are developed, by the very same developers.
- Dependently typed functional programming: The Gallina language embedded in Coq's logic allows streamlined correctness proofs of purely functional programs. The CompCert compiler itself is an example of such a program, proved correct in Coq. Our CertiCoq project [66] is a proved-correct compiler for Gallina, written in Gallina. But directly compiling general Gallina programs leads to certain inefficiencies: the compiled program must be purely functional (no imperative array update), and it must be garbage-collected.
- The Fiat project [67] uses Gallina as a specification language, but generates efficient software code automatically from specifications, in a correct-by-construction way that outputs proofs of correctness. That style is being applied within DeepSpec to derive efficient cryptographic-primitive code from whiteboard-level specifications.

We believe that elements of formal methods will eventually make their way as thoroughly into the standard development workflow as ideas like testing and debugging have today. To identify elements that are more likely to succeed soon, it is important to explore multiple avenues.

5. Technical programme

(a) Components and their interfaces

(i) The ISA interface: connecting microarchitectures to OS kernels

The hardware instruction-set interface is one of the most classic and effective abstraction boundaries in computer systems. It is also a crucial *formal* interface in the DeepSpec demonstration system, and we plan to exercise it thoroughly *from both sides*. 'Above' this interface come the CertiKOS operating system and CompCert and Vellvm compilers, which assume certain behaviour from processors. 'Below' this interface come verified implementations in hardware, constructed using the Kami system, which allows programming in a Coq-embedded language inspired by Bluespec [68], a high-level hardware description language based on logically atomic execution of state-change rules within encapsulated modules. For the Kami subset of Bluespec, we are building both source-level verification tools and a verified compiler to Verilog RTL.

The ISA formal interface is a good opportunity to study the pragmatics of semantics engineering, to maximize industrial adoption and applicability to many use cases. We are attempting a new style where we write the canonical version of the (sequential instruction) semantics in a carefully limited subset of Haskell; we plan to create translators from that subset to all of the common specification languages, including not just Coq but also ACL2 and Isabelle/HOL. Moreover, we hope to get human-readable documentation and a high-performance hardware simulator from the same artefact, with different translators. The semantics must include supervisor instructions and virtual-memory support, which we are in the process of adding, at the same time as we extend our verified processors to support them.

But an ISA is more than its sequential instruction semantics; the ISA of a multiprocessor must also include its *memory consistency model*. We discuss this below.

(ii) Compiler specification and verification

Whether the program is an operating system or an application, we want to reason about (and prove correctness of) the source-language program, then get guarantees about the compiled program that runs in machine language. For this we need the compiler to be proved correct.

The CompCert verified optimizing C compiler, proved correct with respect to specifications of its source and assembly languages, was a major breakthrough [5]. But it is verified only for monolithic isolated sequential programs; that do not do operating-system calls;[4] that do not use multithreading or (shared-memory) multiprocessing; that are not linked to assembly-language subroutines; with no assurance about stack overflow; with no proofs about the translation from assembly to machine language, nor about linking of separately compiled modules. These limitations make CompCert difficult to apply to real systems.

Thus, a key component of our technical programme is the *specification and verification of optimizing-compiler correctness* in the presence of system-call abstractions, multicomponent programs, multithreading with weak cache coherence; and specification/verification techniques that permit relating source-level reasoning with machine-level implementations of interactions with ISA, system calls and shared-memory operations. Concretely, we extend the CompCert compiler's correctness proof to the stronger claims needed in these contexts. This work is done at Yale, in the CertiKOS project; and at Princeton, in the Verified Software Toolchain project. Related work is done at Penn, in the Vellvm project.

An optimizing compiler performs quite complex transformations on the program. In the early years of this century, as programming on weak-cache-consistent multiprocessors became common, engineers noticed that their C compilers were making optimizations that were sound for single-threaded programs but unsound for concurrent programs. In the past two decades there have been hundreds of scientific papers addressing the specification of relaxed memory

[4] More specifically, that do not share memory buffers with operating-system calls, as for example the Unix `read` and `write` calls do.

consistency models, the specification of compiler-optimization correctness for such models, and how to verify concurrent programs written to run on weak-cache-consistent machines.

We are particularly interested (within the VST project at Princeton) in specifying compiler correctness for (weakly consistent) concurrency, so that proofs can be done *separately* of:

- source program correctness (using high-level abstractions such as concurrent separation logic);
- compiler correctness (using a specification that barely mentions concurrency at all);
- and that every execution on a weakly consistent machine behaves as if it were a sequentially consistent execution.

To accomplish these, we generalize and formalize the traditional data-race-free theorem: programs without data races will behave (on a weakly consistent machine) as if the machine were sequentially consistent. Concurrent separation logic guarantees *permission safety* at the source-language level; permission safety guarantees compiler correctness; compiler correctness guarantees permission safety at the machine-language level; permission safety guarantees data-race freedom. Composing all these theorems together allows the programmer to prove program correctness at the source program level, without any consideration of weak memory consistency. Proving each of these theorems is a substantial research project, especially because we plan to prove the theorems about real systems: the C language with Pthreads library; the CompCert compiler; real machines such as Intel IA-32, ARM, and IBM PowerPC.

Verifying the assembler and linker. CompCert performs provably correct translation from C to assembly. But assembly language is not the same as machine language; CompCert relies on standard (unverified) assemblers for this last phase of translation. Furthermore, CompCert's operational semantic definition of assembly code uses a memory-addressing model that is more abstract than the 'real' machine; it uses block-pointers and offsets rather than the flat address space of a standard Von Neumann machine. We plan to bridge these gaps with formal proofs: the relation between abstract and real memory-addressing models; correctness of the assembler. A formal treatment of the linking of .o files [69] might also be necessary.

(iii) Operating systems and compilers

What is the interface between an application program and the operating system it runs on? The application sees a virtual address space in which it can run the user-mode instruction set and perform system calls. The details of page-table mappings, interrupts, etc., are not part of the interface (or its specification); they are part of the *implementation* of that spec by the OS. The specification treats concerns such as the following.

- Memory-addressing models: addressing, read/write permissions, allocation/deallocation, runtime stack (and bounds thereof).
- Concurrency: processes, threads, memory consistency models, atomic instructions versus non-atomic loads/stores.
- System call interface: abstract predicates representing the various components of an operating system (file-system state, communication system, etc.).

Many of these issues are difficult to reason about at the level of source code; but we cannot ask the user to verify programs at the machine-language level. Somehow we *must* take advantage of the abstraction that source programming languages provide, and verify these programs as source code, with source-language-level program logics. Our compiler correctness theorem (about CompCert) must be strong enough to preserve all the correctness properties of interest.

It is not only user programs that are compiled with CompCert: so is the operating system kernel; which adds to those concerns the issue of reasoning about *interrupt handlers*, *fine-grained concurrency* and *address mappings* in virtual-memory page tables, etc. The CertiKOS team [13,16] developed an extended CompCert Clight language (called ClightX) with support for introducing new abstract states, primitives and virtual memory models at different abstraction layers. It also built a compositional thread-safe CompCertX compiler that can compile certified ClightX layers into certified assembly layers.

The CertiKOS team has also adopted the abstraction-layer-based approach for expressing interrupts, which made it possible to build certified interruptible OS kernels and device drivers [14]. It treats each device driver as if it were running on a 'logical' CPU dedicated to that device. This enables us to build a certified hierarchy of extended abstract devices over the raw hardware devices, meanwhile, systematically enforcing the isolation among different 'devices' and the rest of the kernel. The formalization of interrupts in CertiKOS consists of a realistic hardware interrupt model, and an abstract model of interrupts which is suitable for reasoning about interruptible code. These two interrupt models are proven to be contextually equivalent.

(iv) Interfaces between operating systems and compilers

From the point of view of an application program, the interface to the OS primarily consists of the system call API, specifiable in e.g. Verifiable C over abstract states; and the user-mode ISA, which the OS should provide by setting up a user-mode virtual address space.

From the point of view of the OS, an application program is a process that needs to be scheduled, makes callbacks, is equipped with virtual memory, and may either terminate by itself or need to be periodically interrupted. Additional complexity arises from concurrency at all levels (processes, threads, multi-core processors with relaxed memory models, etc.).

Traditionally one runs applications in user mode, with hardware protections (virtual memory protection, protection against executing system-mode instructions) in case the application is buggy or malicious. That works as long as the OS is actually correct; a machine-checked proof of OS correctness will be very valuable. But if the application is *also* proved correct (and non-malicious), then we can avoid the cost of traversing protection boundaries, and link the application directly into the same address space as the kernel.

Our research projects will investigate both modes. Either way, the OS/application boundary will need to be specified, and the OS and application correctness will need to be proved against this interface. (Of course, for running untrusted applications in protected user mode, one need not prove correctness of the application in order to achieve protection guarantees, that it cannot harm the OS or other applications.)

(v) Connecting to intermediate representations for C

Another component of our technical programme is exploring the specification boundaries between various intermediate representations (IRs) for LLVM, CompCert, CoreHaskell and CertiCoq.

LLVM. The LLVM (*low-level virtual machine*) is an industrial strength compiler toolchain that is widely used in practice (most notably by Apple) and in academic research [70]. Zdancewic's Vellvm project aims to build a *verified* LLVM—a framework for reasoning about programs expressed in LLVM's IR and transformations that operate on it [10,71]. Vellvm provides a mechanized formal semantics of LLVM's IR, its type system, and properties of its SSA form [72].

Vellvm will serve as a testbed for experimenting with proof-engineering techniques and as a component of a larger system involving many deep specifications. Handling deep specifications at this scale (and that evolve over time, as LLVM's must) will require significant technical advances. One challenge is the *modular definition of language semantics*. Rather than reasoning about a monolithic language definition, we plan to modularize the specification and corresponding proof development, adapting ideas from recent work on handlers for algebraic effects [73,74]. A related challenge, which is also a particularly compelling test of a modular language definition for the LLVM IR, is to factor the language semantics so that it is able to interface with different memory models. Following ideas developed in Appel's group [75], we are developing a compositional semantics with a memory-model interface suitable for multithreaded programs.

We plan to test the new design in several ways. First, we will use the framework to validate more LLVM optimizations, transformations and analyses, concentrating primarily on those related to security and reliability. Second, we plan to connect the Vellvm specification to both the Haskell and CertiCoq developments that are described next. These connections will be first tested (using QuickChick) before any full-compiler correctness proofs are attempted. Finally, the Vellvm IR should be well suited for making connections to the CompCert compiler that is used elsewhere in the DeepSpec projects.

Haskell. Programmers do not only program in C; many applications are built using components written in a combination of high- and low-level languages. Therefore, as part of this project we would like to understand how to specify and certify compilers for high-level languages. In particular, we focus on the Haskell language, and the primary compiler GHC (the Glasgow Haskell Compiler).

This part of the project is challenging for a number of reasons: Haskell is a feature-rich language, and Haskell itself is a moving target—GHC continues to be a locus of research in language design. GHC itself is an extensive, industrial-strength compiler, which has been developed without verification in mind. Rewriting GHC as a fully verified CompCert-like compiler would take more resources than we have for the project, and the result would most likely be out of date before it was finished.

Therefore, instead of specifying the entire Haskell language, we plan to develop a formal Coq specification of the GHC 'Core' intermediate language, including the syntax, type system and semantics, and connect that specification to other components of this project and to GHC. Core itself is much simpler than Haskell; it has a handful of constructs that all Haskell features elaborate to. Furthermore, it is explicitly typed, so type inference need not be part of the specification. As a result, Core is more stable than Haskell—changes to GHC's type inference algorithm and new additions that can be expressed as Core terms do not invalidate the specification.

Furthermore, we will refine our Core language specification from below via compilation. As part of its compilation pipeline, GHC performs several optimizations on Core terms (e.g. the 'simplifier'). We would like this specification to enable proofs of the correctness of those optimizations. If our Core specification is a complete AST, we will be able to incorporate verified Coq passes into GHC as a Core plug-in. That means any verified Core-to-Core analyses and passes can be live—we will be able to run them on Haskell code.

GHC also has an LLVM backend [76]: we can use this backend to connect the Core specification to Vellvm. This connection would be the foundation for a certified Haskell compiler, and would ensure that both the Core specification and Vellvm are suitable for such purposes. We plan to verify the connection to the Core implementation using tools such as property-based testing.

Finally, Core is a user-facing specification. Although we have no plans to use it as the target of compilation, we can judge the internal consistency of the specification itself. In particular, we plan to prove standard metatheoretic properties of Core, such as progress, preservation, uniqueness of types and type erasure.

(b) Specification validation and applications

(i) Cryptographic security

Cryptographic protocols represent an ideal application domain for formal methods, considering the combination of sophisticated mathematics, concise functional specifications, prevalence of bugs in popular libraries, and real-world importance of getting the details right. Our demonstration plan includes a number of cryptographic modules validated in a number of ways. The first approach links proofs of implementation correctness and cryptographic soundness using VST and FCF, as discussed earlier.

A more avant-garde option, mentioned in §4, is generation of cryptographic code automatically from specifications. We have been working on generating elliptic curve cryptography from relatively short formulae in arithmetic modulo large, carefully chosen prime numbers. Our library generalizes the essence of strategies used to implement efficient hardware arithmetic for different shapes of prime moduli, so that high-quality code for new primes can be generated quickly. The generic algorithm is proven correct, as a functional program, once and for all, and we combine that result with partial evaluation and a verified compiler to lower-level form. We are investigating how to broaden the scope of this style so that we might generate a complete TLS library largely automatically from a functional specification of reasonable complexity. However, integrating the thus generated code into a C ecosystem requires establishment of a formal bridge between Fiat and the VST.

(ii) Deep testing

Property-based random testing (PBRT)—or *automatic specification-based testing*—is a form of black-box testing where a software artefact is subjected to a large number of random checks derived from formal statements of its intended behavioural properties. For example, if the artefact under test is a function f that takes a list of numbers as its input and (hopefully) returns the same list in sorted order, then a reasonable specification of f might be $\mathit{sorted}(f(\mathit{oldlist})) \wedge \mathit{permutation}(f(\mathit{oldlist}),\ \mathit{oldlist})$, where *sorted* is a simple function that checks whether its input is ordered and *permutation* checks whether one of its argument lists is a permutation of the other. To test whether f satisfies this specification, we generate a large number of random input lists, apply f to each one, and check that *sorted* and *permutation* both yield true. PBRT was popularized in the functional programming world by the *QuickCheck* library for Haskell [77], and the past decade has seen an explosion of work in the area.

The executable specifications used by QuickCheck and related tools are *deep*: they embody rich descriptions of component behaviours (not just specific use cases, as unit tests do), they are precise and formal, and they are connected, via random testing, to concrete implementations. Indeed, we should be able to use the same deep specifications for both random testing and formal verification—random testing for low-cost bug finding early in the development process (and for components that are not cost-effective to verify), and deductive verification for high assurance after the specification and implementation have both stabilized.

Realizing this vision in practice will involve addressing some fundamental challenges. In particular:

— PBRT demands that specifications be presented in an executable form—essentially, in the form $\forall X_1, \ldots \forall X_m, \ldots P_1 \wedge \ldots \wedge P_n \Rightarrow Q$, where P_i and Q are boolean *functions* over the X_i. However, the most natural style for writing many specifications is a *relational style*. To bridge this gap, new fundamental theory and tools are needed for *extracting executable specifications from relational ones*, extending prior work on random testing in proof assistants such as Isabelle [78–80], Agda [81–83] and ACL2 [84,85].

— Current PBRT tools are limited in their ability to test *non-deterministic systems* such as operating systems. To overcome this limitation, tools are needed for allowing the testing framework to control the scheduling of the component under test.

Our main experimental platform for PBRT is a *native version of QuickCheck implemented within Coq*, dubbed *QuickChick*. This tool is being used in developing specifications and implementations:

— to debug early versions of new LLVM optimization passes with respect to the deeply specified LLVM semantics by testing that a large number of randomly generated programs behave the same before and after optimization;

— to debug the proposed Bluespec compiler with respect to the deeply specified semantics of the Bluespec and Verilog languages;

— to validate the GHC compiler by generating Haskell expressions and checking that their behaviour as specified by the Haskell Core semantics matches the behaviour of the LLVM

IR terms emitted by the LLVM backend for GHC—including the behaviour of foreign functions specified in Verifiable C and system services as described by the CertiKOS specification; and

— to allow rigorous validation of CertiKOS components (such as device drivers) for which full formal verification is impracticable.

(c) A Web server application

To evaluate top-level compositionality of deeply specified systems, we have recently begun the verification of a Web server library that is loosely based on the popular libmicrohttpd [86]. In addition to representing a typical piece of computational infrastructure, Web servers represent the Internet-enabled maintenance interface of routers and IoT devices, making their protection crucial for numerous industrial and domestic applications.

We anticipate an iterative verification process starting from a bare-bones sequential server of static pages. Concurrent to the verification of each version, advanced features are added to our implementation, improving either functionality (support for HTTPS rather than HTTP, dynamic creation of pages, reference to a background database) or performance (fine-grained multithreading using callbacks or other event-handling abstractions). Of particular interest will be the breakdown of user-level requirements (reliable concurrent servicing of HTTP requests with functionally correct results) into lower-level properties asserting isolation of multiple requests, correctness of effectful transactions, and perhaps quality-of-service aspects.

6. Conclusion

As a final point of comparison, the reader may consider *The Definition of Standard ML* almost three decades ago [87]. This specification was *rich*—it comprises static and dynamic (operational) semantics—but only *semi-formal*: the specification employs judgments in natural-deduction style and other inductive relations, but only parts of the language were (later) mechanized in a proof assistant [88–90]; it was complemented by an extensive commentary that includes pencil-and-paper proofs motivating key language aspects [91]. The specification was not *live*: the recent CakeML can be considered a machine-checked implementation [12] but no prior ML compiler adheres to the specification strictly—certainly not provably so. It was also buggy [92]: subtle errors were later eliminated in a revised version [93]. Illustrating the importance of two-sided evaluation, many mistakes were in fact only uncovered when the specification was subjected to an *unanticipated* use case, namely the development of Extended ML [94]. Nevertheless, SML'89 represented a remarkable achievement: it demonstrated to a wider audience that formal language specification that goes beyond syntactic issues is possible and provided a point of reference for numerous compilers of the greater ML family, including SML/NJ and OCaml.

As illustrated by this comparison, core objectives of deep specifications were identified several decades ago. With the advent of mature proof assistants, we can now realize the potential of these ideas to address challenges of systems in the twenty-first century, exploiting the proof assistants' unique combination of machine-checked proof, mathematical expressiveness and automation support to achieve foundational verification of complex behaviour at scale.

Data accessibility. This article has no additional data.

Competing interests. We declare we have no competing interests.

Funding. *The Science of Deep Specification* is a *Collaborative Research: Expeditions in Computing* project funded by the National Science Foundation (NSF) under the awards 1521602 (A.W.A), 1521584 (A.C.), 1521539 (S.W., S.Z., B.C.P.) and 1521523 (Z.S.).

Acknowledgements. We thank the graduate students and post-doctoral researchers of our research groups for their contributions, and our external academic and industrial collaborators for their participation. The comments we received from the editor and the anonymous referees were particularly helpful in shaping our presentation.

References

1. Spivey M. 1992 *The Z notation: a reference manual*. Englewood Cliffs, NJ: Prentice Hall.
2. Jackson D. 2012 *Software abstractions: logic, languages, and analysis*. Cambridge, MA: MIT Press.
3. Feiler PH, Gluch DP. 2012 *Model-based engineering with AADL*. New York, NY: Addison Wesley.
4. De Millo RA, Lipton RJ, Perlis AJ. 1979 Social processes and proofs of theorems and programs. *Commun. ACM* **22**, 271–280. (doi:10.1145/359104.359106)
5. Leroy X. 2009 Formal verification of a realistic compiler. *Commun. ACM* **52**, 107–115. (doi:10.1145/1538788.1538814)
6. Yang X, Chen Y, Eide E, Regehr J. 2011 Finding and understanding bugs in C compilers. In *Proc. 32nd ACM SIGPLAN Conf. on Programming Language Design and Implementation, San Jose, CA, 4–8 June*, pp. 283–294. New York, NY: ACM. (doi:10.1145/1993498.1993532)
7. Azevedo de Amorim A, Collins N, DeHon A, Demange D, Hriţcu C, Pichardie D, Pierce BC, Pollack R, Tolmach A. 2014 A verified information-flow architecture. In *Proc. 41st Annu. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, San Diego, CA, 22–24 January*, pp. 165–178. New York, NY: ACM. (doi:10.1145/2535838.2535839)
8. Sewell P, Sarkar S, Owens S, Nardelli FZ, Myreen MO. 2010 x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM* **53**, 89–97. (doi:10.1145/1785414.1785443)
9. Sarkar S, Sewell P, Alglave J, Maranget L, Williams D. 2011 Understanding power multiprocessors. In *Proc. 32nd ACM SIGPLAN Conf. on Programming Language Design and Implementation, San Jose, CA, 4–8 June*, pp. 175–186. New York, NY: ACM. (doi:10.1145/1993498.1993520)
10. Zhao J, Nagarakatte S, Martin MM, Zdancewic S. 2012 Formalizing the LLVM intermediate representation for verified program transformations. In *Proc. 39th Annu. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, Philadelphia, PA, 25–27 January*, pp. 427–440. New York, NY: ACM. (doi:10.1145/2103656.2103709)
11. Bodin M, Chargueraud A, Filaretti D, Gardner P, Maffeis S, Naudziuniene D, Schmitt A, Smith G. 2014 A trusted mechanised JavaScript specification. In *Proc. 41st ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, San Diego, CA, 22–24 January*, pp. 87–100. New York, NY: ACM. (doi:10.1145/2535838.2535876)
12. Kumar R, Myreen MO, Norrish M, Owens S. 2014 CakeML: a verified implementation of ML. In *Proc. 41st ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, San Diego, CA, 22–24 January*, pp. 179–191. New York, NY: ACM. (doi:10.1145/2578855.2535841)
13. Gu R, Koenig J, Ramananandro T, Shao Z, Wu X, Weng SC, Zhang H, Guo Y. 2015 Deep specifications and certified abstraction layers. In *Proc. 42nd Annu. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, Mumbai, India, 15–17 January*, pp. 595–608. New York, NY: ACM. (doi:10.1145/2676726.2676975)
14. Chen H, Wu X, Shao Z, Lockerman J, Gu R. 2016 Toward compositional verification of interruptible OS kernels and device drivers. In *Proc. 37th ACM SIGPLAN Conf. on Programming Language Design and Implementation, Santa Barbara, CA, 13–17 June*, pp. 431–447. New York, NY: ACM. (doi:10.1145/2908080.2908101)
15. Costanzo D, Shao Z, Gu R. 2016 End-to-end verification of information-flow security for C and assembly programs. In *Proc. 37th ACM SIGPLAN Conf. on Programming Language Design and Implementation, Santa Barbara, CA, 13–17 June*, pp. 648–664. New York, NY: ACM. (doi:10.1145/2908080.2908100)
16. Gu R, Shao Z, Chen H, Wu XN, Kim J, Sjöberg V, Costanzo D. 2016 CertiKOS: an extensible architecture for building certified concurrent OS kernels. In *Proc. 12th USENIX Symp. on Operating Systems Design and Implementation, Savannah, GA, 2–4 November*, pp. 653–669. See https://www.usenix.org/system/files/conference/osdi16/osdi16-gu.pdf.
17. Klein G *et al.* 2009 seL4: Formal verification of an OS kernel. In *Proc. ACM SIGOPS 22nd Symp. on Operating Systems Principles, Big Sky, MT, 11–14 October*, pp. 207–220. New York, NY: ACM. (doi:10.1145/1629575.1629596)
18. Klein G, Andronick J, Keller G, Matichuk D, Murray T, O'Connor L. 2017 Provably trustworthy systems. *Phil. Trans. R. Soc. A* **375**, 20150404. (doi:10.1098/rsta.2015.0404)
19. Kreitz C. 2004 Building reliable, high-performance networks with the Nuprl proof development system. *J. Funct. Program.* **14**, 21–68. (doi:10.1017/S0956796803004854)

20. Guha A, Reitblatt M, Foster N. 2013 Machine-verified network controllers. In *Proc. 34th ACM SIGPLAN Conf. on Programming Language Design and Implementation, Seattle, WA, 16–19 June*, pp. 483–494. New York, NY: ACM. (doi:10.1145/2491956.2462178)
21. Chlipala A. 2015 From network interface to multithreaded Web applications: a case study in modular program verification. In *Proc. 42nd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, Mumbai, India, 15–17 January*, pp. 609–622. New York, NY: ACM. (doi:10.1145/2676726.2677003)
22. Malecha G, Morrisett G, Shinnar A, Wisnesky R. 2010 Toward a verified relational database management system. In *Proc. 37th Annu. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, Madrid, Spain, 17–23 January*, pp. 237–248. New York, NY: ACM. (doi:10.1145/1706299.1706329)
23. Jang D, Tatlock Z, Lerner S. 2012 Establishing browser security guarantees through formal shim verification. In *Proc. 21st USENIX Security Symp., Bellevue, WA, 8–10 August 2012*, pp. 113–128. See https://www.usenix.org/system/files/conference/usenixsecurity12/sec12-final217.pdf.
24. Bohannon A, Pierce BC. 2010 Featherweight Firefox: formalizing the core of a web browser. In *Proc. 2010 Usenix Conf. on Web Application Development, Boston, MA, 23–24 June*. See http://www.usenix.org/events/webapps10/tech/full_papers/Bohannon.pdf.
25. Chen H, Ziegler D, Chajed T, Chlipala A, Kaashoek MF, Zeldovich N. 2015 Using Crash Hoare logic for certifying the FSCQ file system. In *Proc. 25th Symp. on Operating System Principles, Monterey, CA, 4–7 October*, pp. 18–37. New York, NY: ACM. (doi:10.1145/2815400.2815402)
26. Amani S *et al.* 2016 CoGENT: verifying high-assurance file system implementations. In *Proc. 21st Int. Conf. on Architectural Support for Programming Languages and Operating Systems, Atlanta, GA, 2–6 April*, pp. 175–188. New York, NY: ACM. (doi:10.1145/2872362.2872404)
27. Wilcox JR, Woos D, Panchekha P, Tatlock Z, Wang X, Ernst MD, Anderson TE. 2015 Verdi: a framework for implementing and formally verifying distributed systems. In *Proc. 36th ACM SIGPLAN Conf. on Programming Language Design and Implementation, Portland, OR, 13–17 June*, pp. 357–368. New York, NY: ACM. (doi:10.1145/2737924.2737958)
28. Fisher K, Launchbury J, Richards R. 2017 The HACMS program: using formal methods to eliminate exploitable bugs. *Phil. Trans. R. Soc. A* **375**, 20150401. (doi:10.1098/rsta.2015.0401)
29. Filipovic I, O'Hearn PW, Rinetzky N, Yang H. 2010 Abstraction for concurrent objects. *Theor. Comput. Sci.* **411**, 4379–4398. (doi:10.1016/j.tcs.2010.09.021)
30. Liang H, Hoffmann J, Feng X, Shao Z. 2013 Characterizing progress properties of concurrent objects via contextual refinements. In *CONCUR 2013 – Concurrency theory*. Lecture Notes in Computer Science, vol. 8052, pp. 227–241. Berlin, Germany: Springer. (doi:10.1007/978-3-642-40184-8_17)
31. Appel AW, Dockins R, Hobor A, Beringer L, Dodds J, Stewart G, Blazy S, Leroy X. 2014 *Program logics for certified compilers*. Cambridge, UK: Cambridge University Press.
32. Beringer L, Petcher A, Ye KQ, Appel AW 2015 Verified correctness and security of OpenSSL HMAC. In *24th USENIX Security Symposium, USENIX Security 15, Washington, DC, USA, August 12–14, 2015* (eds J Jung, T Holz), pp. 207–221. USENIX Association, https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/beringer.
33. Appel AW. 2015 Verification of a cryptographic primitive: SHA-256. *ACM Trans. Program. Lang. Syst.* **37**, 7:1–7:31. (doi:10.1145/2701415)
34. Petcher A, Morrisett G. 2015 The Foundational Cryptography Framework. In *Principles of security and trust* (eds R Focardi, AC Myers). Lecture Notes in Computer Science, vol. 9036, pp. 53–72. Berlin, Germany: Springer. (doi:10.1007/978-3-662-46666-7_4)
35. Sewell P. 2017 REMS—Rigorous Engineering of Mainstream Systems. See https://www.cl.cam.ac.uk/~pes20/rems.
36. Coquand T, Huet G. 1988 The calculus of constructions. *Info. Comput.* **76**, 95–120. (doi:10.1016/0890-5401(88)90005-3)
37. Bevier WR, Hunt WA, Moore JS, Young WD. 1989 Special issue on system verification. *J. Autom. Reasoning* **5**, 409–530. (doi:10.1007/BF00243130)
38. Moore JS. 2003 A grand challenge proposal for formal methods: a verified stack. In *Formal methods at the crossroads. From panacea to foundational support* (eds BK Aichering, TSE Maibaum).

Lecture Notes in Computer Science, vol. 2757, pp. 161–172. Berlin, Germany: Springer. (doi:10.1007/978-3-540-40007-3_11)

39. Paul W, Baumann C, Lutsyk P, Schmaltz S, Oberhauser J. 2016 *System architecture as an ordinary engineering discipline*. Berlin, Germany: Springer.
40. Cohen E, Paul WJ, Schmaltz S. 2013 Theory of multi core hypervisor verification. In *SOFSEM 2013: Theory and practice of computer science* (eds P van Emde Boas, FCA Groen, GF Italiano, JR Nawrocki, H Sack). Lecture Notes in Computer Science, vol. 7741, pp. 1–27. Berlin, Germany: Springer. (doi:10.1007/978-3-642-35843-2_1)
41. Paul W, Broy M, In der Rieden T. 2007 The Verisoft XT project. See http://www. verisoft.de.
42. Leinenbach D, Santen T. 2009 Verifying the Microsoft Hyper-V hypervisor with VCC. In *FM 2009: Formal Methods*. Lecture Notes in Computer Science, vol. 5850, pp. 806–809. Berlin, Germany: Springer. (doi:10.1007/978-3-642-05089-3_51)
43. Alkassar E, Hillebrand MA, Paul WJ, Petrova E. 2010 Automated verification of a small hypervisor. In *Verified software: theories, tools, experiments*, Lecture Notes in Computer Science, vol. 6217, pp. 40–54. Berlin, Germany: Springer. (doi:10.1007/978-3-642-15057-9_3)
44. Cohen E, Dahlweid M, Hillebrand M, Leinenbach D, Moskal M, Santen T, Schulte W, Tobies S. 2009 VCC: a practical system for verifying concurrent C. In *Theorem proving in higher order logics*, Lecture Notes in Computer Science, vol. 5674, pp. 23–42. Berlin, Germany: Springer. (doi:10.1007/978-3-642-03359-9_2)
45. Sewell T, Myreen MO, Klein G. 2013 Translation validation for a verified OS kernel. In *Proc. 34th ACM SIGPLAN Conf. on Programming Language Design and Implementation, Seattle, WA, 16–19 June*, pp. 471–482. New York, NY: ACM. (doi:10.1145/2491956.2462183)
46. Hawblitzel C, Howell J, Lorch JR, Narayan A, Parno B, Zhang D, Zill B. 2014 Ironclad apps: end-to-end security via automated full-system verification. In *Proc. 11th USENIX Symp. on Operating Systems Design and Implementation, Broomfield, CO, 6–8 October*. See https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-hawblitzel.pdf.
47. Leino KRM. 2010 Dafny: an automatic program verifier for functional correctness. In *Logic for programming, artificial intelligence and reasoning*. Lecture Notes in Computer Science, vol. 6355, pp. 348–370. Berlin, Germany: Springer. (doi:10.1007/978-3-642-17511-4_20)
48. de Moura LM, Bjørner N. 2008 Z_3: an efficient SMT solver. In *Tools and algorithms for the construction and analysis of systems*. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Berlin, Germany: Springer. (doi:10.1007/978-3-540-78800-3_24)
49. Yang J, Hawblitzel C. 2010 Safe to the last instruction: automated verification of a type-safe operating system. In *Proc. 31st ACM SIGPLAN Conf. on Programming Language Design and Implementation, Toronto, Canada, 5–10 June*, pp. 99–110. New York, NY: ACM. (doi:10.1145/1806596.1806610)
50. Nicely TR. 2011 Account of Intel's Pentium FDIV flaw. http://www.trnicely.net/pentbug/pentbug.html.
51. ARM Ltd. 2017 Exploration tools for ARMv8.2 A-profile architectures. See https://developer.arm.com/products/architecture/a-profile/exploration-tools.
52. Contag M, Li G, Pawlowski A, Domke F, Levchenko K, Holz T, Savage S. 2017 How they did it: an analysis of emission defeat devices in modern automobiles. In *Proc. 38th IEEE Symp. on Security and Privacy, San Jose, CA, 22–26 May*. New York, NY: IEEE. (doi:10.1109/SP.2017.66)
53. Durumeric Z *et al.* 2014 The matter of Heartbleed. In *Proc. 2014 Internet Measurement Conf., Vancouver, Canada, 5–7 November*, pp. 475–488. New York, NY: ACM. (doi:10.1145/2663716.2663755)
54. Newcombe C, Rath T, Zhang F, Munteanu B, Brooker M, Deardeuff M. 2015 How Amazon web services uses formal methods. *Commun. ACM* **58**, 66–73. (doi:10.1145/2699417)
55. Bhargavan K *et al.* 2017 Everest: towards a verified, drop-in replacement of HTTPS. In *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7–10, 2017, Asilomar, CA, USA*. LIPIcs (eds BS Lerner, R Bodík, S Krishnamurthi), vo1. 71 , pp. 1:1–1:12. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. (doi:10.4230/LIPIcs.SNAPL.2017.1)
56. Pierce BC, Casinghino C, Gaboardi M, Greenberg M, Hriţcu C, Sjoberg V, Yorgey B. 2013 *Software foundations*. (Electronic textbook).
57. Nipkow T, Klein G. 2014 *Concrete semantics with Isabelle/HOL*. Berlin, Germany: Springer.
58. Pollack R. 1997 How to believe a machine-checked proof. Technical Report RS-97-18, BRICS, Department of Computer Science, University of Aarhus.

59. Asperti A, Geuvers H, Natarajan R. 2009 Social processes, program verification and all that. *Math. Struct. Comput. Sci.* **19**, 877–896. (doi:10.1017/S0960129509990041)
60. Hales T, Gonthier G, Harrison J, Wiedijk F. 2008 Special issue on formal proof. *Not. Am. Math. Soc.* **55** (11), 1370–1414. See http://www.ams.org/notices/200811/tx081101370p.pdf.
61. Appel AW, Michael NG, Stump A, Virga R. 2003 A trustworthy proof checker. *J. Autom. Reason.* **31**, 231–260. (doi:10.1023/B:JARS.0000021013.61329.58)
62. Werner B. 1994 Une Théorie des Constructions Inductives. Theses, Université Paris Diderot–Paris VII.
63. Ekici B, Mebsout A, Tinelli C, Keller C, Katz G, Reynolds A, Barrett C. 2017 SMTCoq: a plug-in for integrating SMT solvers into Coq. In *Proc. 29th Int. Conf. on Computer Aided Verification*. Lecture Notes in Computer Science, vol. 10427, pp. 126–133. Cham, Switzerland: Springer. (doi:10.1007/978-3-319-63390-9_7)
64. Sigurbjarnarson H, Bornholt J, Torlak E, Wang X. 2016 Push-button verification of file systems via crash refinement. In *Proc. 12th USENIX Symp. on Operating Systems Design and Implementation, Savannah, GA, 2–4 November* (eds K Keeton, T Roscoe), pp. 1–16. See https://www.usenix.org/system/files/conference/osdi16/osdi16-sigurbjarnarson.pdf.
65. Choi J, Vijayaraghavan M, Sherman B, Chlipala A, Arvind K. 2017 Kami: a platform for high-level parametric hardware specification and its modular verification. To be presented at *22nd ACM SIGPLAN Int. Conf. on Functional Programming, Oxford, UK, 3–9 September*.
66. Anand A, Appel AW, Morrisett G, Paraskevopoulou Z, Pollack R, Savary Bélanger O, Sozeau M, Weaver M. 2017 CertiCoq: a verified compiler for Coq. In *CoqPL 2017: 3rd Int. Workshop on Coq for Programming Languages, Paris, 21 January*.
67. Chlipala A, Delaware B, Duchovni S, Gross J, Pit-Claudel C, Suriyakarn S, Wang P, Ye K. 2017 The end of history? Using a proof assistant to replace language design with library design. In *SNAPL'17: Proc. 2nd Summit on Advances in Programming Languages, Asilomar, CA, 7–10 May*.
68. Bluespec Inc., Waltham MA. 2004 Bluespec System Verilog Version 3.8 Reference Guide.
69. Kell S, Mulligan DP, Sewell P. 2016 The missing link: explaining ELF static linking, semantically. In *Proc. 2016 ACM SIGPLAN Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications, Amsterdam, The Netherlands, 2–4 November*, pp. 607–623. New York, NY: ACM. (doi:10.1145/2983990.2983996)
70. Lattner C, Adve V. 2004 LLVM: a compilation framework for lifelong program analysis and transformation. In *Int. Symp. on Code Generation and Optimization, San Jose, CA, 20–24 March*, pp. 75–86. New York, NY: IEEE. (doi:10.1109/CGO.2004.1281665)
71. Zhao J, Nagarakatte S, Martin MMK, Zdancewic S. 2013 Formal verification of SSA-based optimizations for LLVM. In *Proc. 34th ACM SIGPLAN Conf. on Programming Languages Design and Implementation, Seattle, WA, 16–19 June*, pp. 175–186. New York, NY: ACM. (doi:10.1145/2491956.2462164)
72. Zhao J, Zdancewic S. 2012 Mechanized verification of computing dominators for formalizing compilers. In *Certified Programs and Proofs*. Lecture Notes in Computer Science, vol. 7679, pp. 27–42. Berlin, Germany: Springer. (doi:10.1007/978-3-642-35308-6_6)
73. Plotkin G, Pretnar M. 2009 Handlers of algebraic effects. In *Programming Languages and Systems*. Lecture Notes in Computer Science, vol. 5502, pp. 80–94. Berlin, Germany: Springer. (doi:10.1007/978-3-642-00590-9_7)
74. Bauer A, Pretnar M. 2014 Programming with algebraic effects and handlers. *J. Logical Alg. Methods Program.* **84**, 108–123. (doi:10.1016/j.jlamp.2014.02.001)
75. Stewart G, Beringer L, Cuellar S, Appel AW. 2015 Compositional CompCert. In *Proc. 42nd Annu. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, Mumbai, India, 15–17 January*, pp. 275–287. New York, NY: ACM. (doi:10.1145/2676726.2676985)
76. Terei DA, Chakravarty MMT. 2010 An LLVM backend for GHC. In *Proc. 3rd ACM SIGPLAN Haskell Symp. 2010, Baltimore, MD, 30 September*, pp. 109–120. New York, NY: ACM. (doi:10.1145/1863523.1863538)
77. Claessen K, Hughes J. 2000 QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proc. 5th ACM SIGPLAN Int. Conf. on Functional Programming, Montreal, Canada, 18–20 September*, pp. 268–279. New York, NY: ACM. (doi:10.1145/351240.351266)
78. Bulwahn L. 2012 Smart testing of functional programs in Isabelle. In *Logic for Programming, Artificial Intelligence, and Reasoning*. Lecture Notes in Computer Science, vol. 7180, pp. 153–167. Berlin, Germany: Springer. (doi:10.1007/978-3-642-28717-6_14)

79. Bulwahn L. 2012 The new Quickcheck for Isabelle — random, exhaustive and symbolic testing under one roof. In *Certified Programs and Proofs*. Lecture Notes in Computer Science, vol. 7679, pp. 92–108. Berlin, Germany: Springer. (doi:10.1007/978-3-642-35308-6_10)
80. Brucker AD, Wolff B. 2013 On theorem prover-based testing. *Formal Aspects Comput.* **25**, 683–721. (doi:10.1007/s00165-012-0222-y)
81. Dybjer P, Haiyan Q, Takeyama M. 2003 Combining testing and proving in dependent type theory. In *Theorem Proving in Higher Order Logics*. Lecture Notes in Computer Science, vol. 2758, pp. 188–203. Berlin, Germany: Springer. (doi:10.1007/10930755_12)
82. Lindblad F, Benke M. 2006 A tool for automated theorem proving in Agda. In *Types for Proofs and Programs*. Lecture Notes in Computer Science, vol. 3839, pp. 154–169. Berlin, Germany: Springer. (doi:10.1007/11617990_10)
83. Lindblad F. 2007 Property directed generation of first-order test data. In *Proc. 8th Symp. on Trends in Functional Programming, New York, NY, 2–4 April*, pp. 105–123. Bristol, UK: Intellect.
84. Eastlund C. 2009 DoubleCheck your theorems. In *Proc. 8th Int. Workshop on the ACL2 Theorem Prover and Its Applications, Boston, MA, 11–12 May*, pp. 42–46. New York, NY: ACM. (doi:10.1145/1637837.1637844)
85. Chamarthi HR, Dillinger PC, Kaufmann M, Manolios P. 2011 Integrating testing and interactive theorem proving. In *Proc. 10th Int. Workshop on the ACL2 Theorem Prover and Its Applications, Austin, TX, 3–4 November*, EPTCS, vol. 70, pp. 4–19. (doi:10.4204/EPTCS.70.1)
86. Free Software Foundation. 2017 Gnu libmicrohttpd. See https://www.gnu.org/software/libmicrohttpd.
87. Milner R, Tofte M, Harper R. 1989 *The definition of standard ML*. Cambridge, MA: MIT Press.
88. Van Inwegen M, Gunter EL. 1993 HOL-ML. In *Higher Order Logic Theorem Proving and Its Applications* (eds JJ Joyce, CH Seger). Lecture Notes in Computer Science, vol. 780, pp. 61–74. Berlin, Germany: Springer. (doi:10.1007/3-540-57826-9_125)
89. Syme D. 1994 Reasoning with the formal definition of standard ML in HOL. In *Higher Order Logic Theorem Proving and Its Applications* (eds JJ Joyce, CH Seger). Lecture Notes in Computer Science, vol. 780, pp. 43–60. (doi:10.1007/3-540-57826-9_124)
90. Maharaj S, Gunter EL. 1994 Studying the ML module system in HOL. In *Higher Order Logic Theorem Proving and Its Applications* (eds TF Melham, J Camilleri). Lecture Notes in Computer Science, vol. 859, pp. 346–361. Berlin, Germany: Springer. (doi:10.1007/3-540-58450-1_53)
91. Milner R, Tofte M. 1991 *Commentary on standard ML*. Cambridge, MA: MIT Press.
92. Kahrs S. 1993 Mistakes and ambiguities in the definition of standard ML. Technical Report ECS-LFCS-93-257, Laboratory for Foundations of Computer Science, University of Edinburgh.
93. Milner R, Harper R, MacQueen D, Tofte M. 1997 *The definition of standard ML (revised)*, Revised edn. Cambridge, MA: MIT Press.
94. Kahrs S, Sannella D, Tarlecki A. 1997 The definition of extended ML: a gentle introduction. *Theor. Comput. Sci.* **173**, 445–484. (doi:10.1016/S0304-3975(96)00163-6)