Chapter 18
Formalization of Programming Languages

Robbert Krebbers, Alberto Momigliano, and Brigitte Pientka

Second readers: Arthur Charguéraud and Andrei Popescu

To understand the subtle and complex interactions of language concepts and avoid flaws, researchers have developed rigorous programming language definitions and proofs so that we can formally establish that languages are safe, robust, and trustworthy. We discuss results about the mechanization of programming language definitions and type systems (Section 18.2), compilers (Section 18.3), and program logics (Section 18.4).

When reasoning about programming languages in a proof assistant, an overarching question is how to represent variables, (simultaneous) substitution, contexts, as well as the restriction of named resources to certain scopes. This is summarized in Section 18.1, next.

18.1 Variable Binding

What infrastructure does a proof assistant need to mechanize programming language metatheory? The toolbox must contain

- a way to represent the syntax of the language;
- a way to encode the relevant judgments describing the semantics of the language;
- a way to state the necessary theorems and carry out the required proofs.

The proof assistant may use algebraic datatypes to encode the syntax, inductive predicates and (primitive) recursive functions to encode the semantics, and structural

Robbert Krebbers
Name, Address of Institute e-mail: name@email.address

Alberto Momigliano
DI, Università degli Studi di Milano, Italy e-mail: momigliano@di.unimi.it

Brigitte Pientka
Name, Address of Institute e-mail: name@email.address

or well-founded induction (or even coinduction) to carry out the proofs, typically by tactics or filling in proof terms. The mechanization of "high-level" programming languages (think functional ones) is characterized—some would say plagued—by the question of how to represent *bindings* and the *scope* of variables and assumptions. This muddles significantly the above picture.

A number of benchmarks and challenge problems have been used to understand, compare, and push the state of the art on variable binding (e.g., the POPLmark [20] and POPLmark Reloaded challenges [1]). We provide an overview of the different approaches and we discuss the difficulties that variable binding brings to the table, in particular the various encoding techniques that have been deployed and their consequences. To tame a large field of applications, we highlight in particular the efforts to verify the metatheory of Standard ML [166].

We will exemplify some of the issues using a simply typed lambda calculus: while the syntax and the equational/reduction semantics of the untyped lambda calculus has already been discussed in details in Section **??**, here we concentrate on a typed version (for convenience with only one base type), as the tiniest model of a typed functional programming language.[1] Accordingly, we will fix a particular reduction strategy—weak-head reduction—as typical of eager programming languages.

$$\begin{array}{lrcl} \text{Types} & A, B & ::= & \text{unit} \mid A \to B \\ \text{Terms} & M & ::= & x \mid \langle\rangle \mid \lambda x.\, M \mid M_1\, M_2 \end{array}$$

The first task is to give an encoding of the syntax in the metalanguage offered by a given proof assistant. However, choices in the representation of the syntax are inextricably linked to how we represent judgments such as typing and evaluation and how we reason about those. We therefore introduce the static and dynamic semantics of our lambda calculus before discussing the different choices for representing the syntax of terms: A lambda abstraction is the primary example of a term formed out of a *binder*, which cannot be adequately represented by a standard abstract syntax tree. An issue already mentioned in Section **??** is that terms differing only over bound names should be the "same" (α-equivalence), as witnessed by the (most of the times) silent use of "Barendregt's variable convention" [24][p. 25] in informal treatment of the lambda calculus. More crucially, functions defined on such terms should respect this discipline.

Second, as Alan Perlis famously observed in the context of programming languages, is there such a thing as a *free* variable? Consider the typing rule for abstraction: the variable x bound by the lambda in the conclusion now appears seemingly free in the context and in the body of the abstraction. Miller [163] calls this phenomenon *mobility* of binders.

Third, are we sure that the terms in the object language correspond one-to-one to a representation in our proof assistant? Such *adequacy* [109] would ensure that we

[1] Similar remarks apply to other models of programming languages such as the π-calculus, see the call to action in Carbone et al. [54].

Typing Rules

$$\frac{}{\vdash \langle\rangle : \text{unit}}\ \text{T-UNIT} \qquad \frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ \text{T-VAR}$$

$$\frac{x \notin \text{dom}(\Gamma) \quad \Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.\, M : A \to B}\ \text{T-ABS} \qquad \frac{\Gamma \vdash M_1 : A \to B \quad \Gamma \vdash M_2 : B}{\Gamma \vdash M_1 \cdot M_2 : B}\ \text{T-APP}$$

Evaluation Rules

$$\frac{\text{value } V}{V \Downarrow V}\ \text{E-VAL} \qquad \frac{M_1 \Downarrow \lambda x.\, M \quad M_2 \Downarrow V_2 \quad M\{V_2/x\} \Downarrow V}{M_1 \cdot M_2 \Downarrow V}\ \text{E-APP}$$

Values

$$\frac{}{\text{value } \lambda x.\, M}\ \text{V-ABS} \qquad \frac{}{\text{value } \langle\rangle}\ \text{V-UNIT}$$

Fig. 18.1 Statics and dynamic semantics

characterize at least all the terms present in our object language and conversely that we do not characterize more terms (known as "exotic terms" [79]).

Finally, our operational semantics relies on *capture-avoiding substitutions* (Definition **??** in Chapter **??**), a notion that is pervasive in the semantics of programming languages. However, this definition consistently poses a challenge during formalization, particularly when *proving* properties of its equational theory and when interacting with other judgments.

A dazzling number of answers have emerged since De Bruijn's seminal paper [76], as witnessed by the solutions to the POPLmark challenge.[2] Without any attempt at completeness, we list and explain the gist of the some representative approaches to binding signatures in the next subsections. These have been realized in two ways: (1) in general-purpose proof assistants, often using libraries, frontends or code generators; (2) in specialized logical frameworks. We will make some remarks about both as we go along.

18.1.1 Named Terms

This approach gives a first order account of binding signatures as a standard algebraic datatype, using a named representation of variables (both bound and free) as strings or integers:

[2] `https://www.seas.upenn.edu/~plclub/poplmark`

$$\begin{array}{lll} \texttt{datatype} & \mathsf{tm} ::= & \mathsf{nUNIT} : \mathsf{tm} \\ & \mid & \mathsf{nVAR} : \mathsf{var} \to \mathsf{tm} \\ & \mid & \mathsf{nABS} : \mathsf{var} \to \mathsf{tm} \to \mathsf{tm} \\ & \mid & \mathsf{nAPP} : \mathsf{tm} \to \mathsf{tm} \to \mathsf{tm} \end{array}$$

Using strings for variables, a term such as $\lambda f.\, \lambda x.\, f\, x$ is represented by

$$\mathsf{nABS}\ \text{“f”}\ (\mathsf{nABS}\ \text{“x”}\ (\mathsf{nAPP}\ (\mathsf{nVAR}\ \text{“f”})\ (\mathsf{nVAR}\ \text{“x”})))$$

Named terms have been used successfully in the mechanization of low-level languages, and more generally in situations where binders play a limited role (for example, to formalize function tables and record fields). For languages based on the lambda calculus, named terms are not well suited for metatheory in general, but they may be effective in restricted settings. Particularly, when defining the operational semantics of call-by-value languages, one can often restrict to closed terms (whereby substitutions do not need to be capture-avoiding) and avoid α-equivalence [149, p. 6, footnote 1]. Named terms additionally have the advantage of being human readable and mirroring some pen-and-paper proof practices. The first is particularly useful when reasoning about large concrete programs using program logics embedded in proof assistants (such as CFML or Iris, Section 18.4). The second can be a didactic asset for example when formalizing basic results in the metatheory of the lambda calculus, as in the Coq-based textbook *Programming Language Foundations* [202]. Named terms can also be chosen when using an environment-based operational semantics, so that substitutions can be avoided altogether [83].

In case one needs to reason about open terms or α-equivalence, a heavy bootstrap phase is needed for named terms to be effectively applied to programming language metatheory. Defining capture-avoiding substitution and α-equivalence is notoriously difficult. As a result, one needs to prove many seemingly trivial lemmas that have little to do with the mathematics of the problem, and need to be re-established for each case study. Accounts vary on how this issue is handled: α-equivalence can be established via explicit quotients [92] or as a derived (inductive) relation induced by renaming, possibly by means of simultaneous substitutions [71].

18.1.2 Nominal Techniques

Introduced by Gabbay and Pitts originally in the context of FM-sets [96], nominal techniques have had a significant impact in computer science and mathematics [205]. They can be viewed as a refinement of the named approach in so far as the names of bound variables are explicit, while their particular choice is irrelevant. The key insight is that binding signatures can be accounted for in terms of finite *permutations* over an abstract, but first-class datatype of *names* (or atoms). In fact, permutations (invertible renaming) have far better algebraic properties than substitutions in so far that they are intrinsically capture-avoiding; permutations can be used

to define α-equivalence and more in general *equivariant* predicates—i.e., predicates that are invariant under name swapping.

Nominal logic (or more generally nominal signatures) can be presented as an extension of first-order (typed) equational logic [203, 204]. It provides a syntactic class of names, a swapping operation that permutes two names in a term, and a freshness relation #, which relates a name to a term where it does not occur free. The α-equivalence relation is defined in terms of swapping and freshness, and so are a *name-abstraction* operation as well as a self-dual *new* quantifier that varies over fresh names and is in charge of fresh name generation. The logic includes structural principles of recursion and induction modulo α-equivalence.

Using a syntax close to Nominal Isabelle [253], we declare a nominal datatype for the lambda terms of our running example:

$$\begin{array}{lll} \texttt{nominal\_datatype} & \mathsf{tm} ::= & \mathsf{nmUNIT} : \mathsf{tm} \\ & \mid & \mathsf{nmVAR} : \mathit{name} \to \mathsf{tm} \\ & \mid & \mathsf{nmABS} : \langle \mathit{name} \rangle\, \mathsf{tm} \to \mathsf{tm} \\ & \mid & \mathsf{nmAPP} : \mathsf{tm} \to \mathsf{tm} \to \mathsf{tm} \end{array}$$

The term $\lambda f.\, \lambda x.\, f\, x$ is encoded as

$$\mathsf{nmABS}\ \langle f \rangle\ (\mathsf{nmABS}\ \langle x \rangle\ (\mathsf{nmAPP}\ (\mathsf{nmVAR}\ f)\ (\mathsf{nmVAR}\ x)))$$

for swappable names f, x, where we explicitly abstract over f and x.

Nominal datatypes, while not being standard free datatypes, may be equipped with sophisticated recursion principles that observe Barendregt's variables convention, allowing one to avoid the clash of bound and free variables during recursive calls (see [185, 204] and the recent generalization by Popescu in [209]). The corresponding phenomenon in the context of rule induction is studied in [254].

One line of applications of nominal logic was in the design of new programming languages "modulo α," either from scratch (FreshML [234]) or as a frontend to existing languages, e.g., [210].

On the proof assistant side, after an initial attempt by Gabbay of a foundational implementation of FM-HOL, the breakthrough was the release of the Nominal package in Isabelle/HOL by Urban [253]. He provided a representation of α-equated syntax based on names and equipped with induction and recursion principle that was compatible with classical higher-order logic (and the axiom of choice). Leveraging Isabelle's automation, Nominal Isabelle made possible proofs close to the pen-and-paper style, while requiring a limited amount of manual infrastructural work. Therefore the package was rather successful not only in programming language theory, but also in process calculi [33] and metamathematical applications [195]. Nominal2 [255] added, among other features, a treatment of n-ary and inductively defined binders, and while not officially distributed, it is maintained in the *Archive of Formal Proofs*.[3] Binders as first class citizens has been further generalized by Blanchette et al. [44].

[3] `https://www.isa-afp.org/entries/Nominal2.html`

The application of nominal techniques to constructive type theories [64,206,223] has not made it into practice yet, perhaps as it requires a suitable constructive notion of nominal sets or the development of new type theories with nominal features.

18.1.3 De Bruijn Indices

A De Bruijn index is a natural number representing the occurrence of a variable in a term. It refers to the number of abstractions that one needs to traverse to bind said occurrence. A variable n is bound if it is in the scope of at least n binders, otherwise it is free.

$$\begin{array}{lll} \texttt{datatype} & \mathsf{tm} ::= & \mathsf{dUNIT} : \mathsf{tm} \\ & \mid & \mathsf{dVAR} : \mathbb{N} \to \mathsf{tm} \\ & \mid & \mathsf{dABS} : \mathsf{tm} \to \mathsf{tm} \\ & \mid & \mathsf{dAPP} : \mathsf{tm} \to \mathsf{tm} \to \mathsf{tm} \end{array}$$

The term $\lambda f.\, \lambda x.\, f\, x$ is encoded as

$$\mathsf{dABS}\ (\mathsf{dABS}\ (\mathsf{dAPP}\ (\mathsf{dVAR}\ 1)\ (\mathsf{dVAR}\ 0)))$$

Terms have a unique representation, thus preempting the α-equivalence issue. On the other hand, the same variables may have different indices. Defining (capture-avoiding) substitution requires fiddling with indices, known as *shifting*; there is also some technical overhead in handling dangling pointers. That has led to refinements as "well-scoped" De Bruijn terms [43]: here we index terms with a bound that intuitively describes the size of the context in which the term is meaningful. This idea can be further refined into the *intrinsic* typing discipline [35], where the context also carries the relevant type information, so that only type-correct terms are legal. This approach, possible only in dependently typed logical frameworks and orthogonal to the choice of syntax encoding, is being widely used both for implementing *correct-by-construction* program analyzers [211] and metatheoretical investigations where certain invariants are statically preserved [1, 246].

De Bruijn terms have been intensively used in the implementation of functional languages as well as in programming language metatheory [8, 180]—see the references concerning the verification of Standard ML in Section 18.2. For relationships between De Bruijn and named syntax, including a survey of many variations, see Norrish and Vestergaard [187], Berghofer and Urban [40], and Kamareddine [124].

To try to overcome the complexity of reasoning about capture-avoiding substitutions in this approach, the *locally nameless* [59] variation has emerged, where De Bruijn indices still represent bound variables but names represent free variables when descending into a binder, doing without the need for shifting De Bruijn indices. The locally nameless representation is widely used in the *implementation* of proof assistants and programming languages, e.g. [220].

There is by now significant library support for mechanizing common tasks required by the handling of De Bruijn terms, in particular the generation and quantification of fresh names, both for programming [266] and for metatheory [57]; see tools such as Needle & Knot [129], Autosubst [238], and Allais et al.'s Generic Syntax [7]. Libraries represent a significant improvement to previous work, e.g., compared to some early Church–Rosser proofs [180, 233], where the authors had to (re)build the required infrastructure from scratch.

While De Bruijn encodings provide a unique concrete representation of the syntax of terms, the nature of the game does not fundamentally change when it comes to representing and reason about typing and evaluation rules. Some of the aforementioned approaches alleviate this task by generating the necessary definitions and some of the proofs automatically, but the effort required for large development is still notable [99, 220].

18.1.4 Higher-Order Abstract Syntax

Representations based on higher-order abstract syntax (HOAS) [164, 196] rely on the notion of binders already present in the proof assistant (the metalanguage) to model binders in our object language—namely, in our example, the binding of the variable x in the lambda-term $\lambda x.\, M$. We begin by declaring a type tm together with constants that form elements of type tm.

$$\begin{aligned} \mathsf{tm} &: \mathtt{type}. \\ \mathsf{unit} &: \mathsf{tm} \\ \mathsf{lam} &: (\mathsf{tm} \to \mathsf{tm}) \to \mathsf{tm} \\ \mathsf{app} &: \mathsf{tm} \to \mathsf{tm} \to \mathsf{tm} \end{aligned}$$

HOAS exploits the presence of function spaces in the metalanguage: hence our sample term $\lambda f.\, \lambda x.\, f\, x$ is encoded as $\mathsf{lam}\ (\diagup\!\!\diagdown f.\, \mathsf{lam}\ (\diagup\!\!\diagdown x.\, \mathsf{app}\ f\ x))$, where we write $\diagup\!\!\diagdown$ for functions from the metalanguage. Compared to previous representations, there is no variable case in HOAS. Piggybacking on the function space of the metalanguage means that we get α-renaming and object-level substitution as β-conversion for free, since we simply reuse the equational theory of the metalanguage.

HOAS encodings are not restricted to merely account for the syntax of terms. We can use its fundamental ideas to also represent formal systems such as typing derivations and evaluation. For example, the typing rule T-ABS can be read as follows:

> If for all variables x of type A we can show that M has type B, then we can establish that $\lambda x.\, M$ has type $A \to B$.

To represent such an inference rule, we can again leverage the power of the function space of a metalanguage. We encode the typing relation $M : A$ using the relation $\mathsf{hastype}\ \mathsf{M}\ \mathsf{A}$ where M is the encoding of the object-level term M and A is the en-

coding of the object-level type A. We can define a constant $\mathsf{t{-}abs}$ that encodes the typing rule T-ABS as a meta-level function as follows:[4]

$$\begin{aligned}\mathsf{t-abs} : (\Pi x{:}\mathsf{tm.hastype}\ x\ \mathsf{A} \to \mathsf{hastype}\ (\mathsf{M}\ x)\ \mathsf{B})\\ \to \mathsf{hastype}\ (\mathsf{lam}\ \mathsf{M})\ (\mathsf{arr}\ \mathsf{A}\ \mathsf{B}).\end{aligned}$$

Here we assume a constant arr which encodes the object-level function type. We use Π to quantify over all variables x and enforce that x is new. Renaming of all the inputs to lam M is then modeled by meta-level application, i.e., M x. The informal reading of the meta-level type is the same as the informal reading of the typing rule T-ABS given earlier. Using this HOAS encoding for inference rules, we now inherit weakening, contraction, and substitution lemmas also on the level of typing derivations, thereby avoiding the need to prove such lemmas individually. This often leads to very compact representations and reasoning.

What metalanguages can we use to exploit HOAS encodings? Typically, we use a pure typed lambda calculus consisting only of constants, variables, functions, and function application to encode the syntax; lam and app are then user-defined constants. Choosing a weak metalanguage that does not include recursion and case analysis ensures that HOAS representations are adequate.

Unfortunately, this also means that HOAS definitions for terms and for inference rules about them are not inductive and cannot be directly represented as an inductive type in a general-purpose proof assistant. Indeed, we are defining tm by referring to itself in the argument to lam. Similarly, in the encoding of the typing rule for T-ABS, we assume that x has type A and use it to define the typing relation of the body (i.e., $M : B$).

How can we apply induction and recursion to HOAS encodings? The crux is that if we recursively traverse a representation such as $\mathsf{lam}\ (\diagup f.\,\mathsf{lam}\ (\diagup x.\,\mathsf{app}\ f\ x))$, we will need to make a recursive call on the (open) subterm $\mathsf{lam}\ (\diagup x.\,\mathsf{app}\ f\ x)$, where f is now apparently free.

One initial solution was to have HOAS and induction coexist at the same level, either "squeezing" it in a standard proof assistant [79, 89, 114], and accepting a certain amount of compromises in the process, or by separating the HOAS intensional function space from the (primitive) recursive one via modalities, as in [225].

This separation between the level of object language specification and of reasoning has been embraced over the past two decades and have led to the implementation of domain-specific logical frameworks. In type-theoretic logical frameworks such as Twelf [197] and Beluga [198, 200], we characterize the abovementioned term $\mathsf{lam}\ (\diagup x.\,\mathsf{app}\ f\ x)$ in a context (or world) where we have $f : \mathsf{tm}$. While in Twelf the context remains ambient [224], it is explicit in Beluga. This enables the support of primitive recursion over (open) HOAS objects [199]. In [201] this is further generalized to a two-level type theory where on top of LF sits a Martin Löf type theory.

[4] We use LF syntax here where the type of the free variables M, A, and B will be reconstructed and those variables are quantified at the outside.

The reasoning logic does not need to be a type theory, but it can be formulated as a fixed-point logic [247], extended with a new quantifier (∇) to perform case analysis and induction over "mobile" variables such as f [97]. This is the foundation of the Abella proof-assistant [22].

HOAS encodings in domain-specific frameworks have been used in a variety of case studies, most prominently for the mechanization of Standard ML in Twelf [139]. On the other hand, there are areas where this approach may seem to fare less well, namely where we need to manipulate variable names concretely. In fact, the verification of program transformations such as closure conversion and hosting is in range of of HOAS [31, 262]. Other areas, such as mechanization of type inference, specifically the algorithm W, remain challenging.

These four categories of encoding techniques by no means exhaust research on binding signatures; other approaches exist, e.g., [63, 101, 177]. We also mention in passing systems that belong to the category of "language workbenches", that is environments for simplifying the creation and use of computer languages, see the overview by Erdweg et al. [87]. Among them we single out *Ott* [231] and the *K* framework [221].

18.2 Mechanization of Programming Language Definitions

We give an overview of the achievements in the mechanization of a variety of different programming language definitions. We focus on ML (Section 18.2.1), dependently typed proof assistants (Section 18.2.2), Java-like languages (Section 18.2.3), query languages (Section 18.2.4), low-level languages such as C, C++, and LLVM (Section 18.2.5), and safe low-level languages such as Rust and WebAssembly (Section 18.2.6).

18.2.1 The Verification of Standard ML

We focus here on the verification of type soundness of Standard ML, to highlight both the challenges and the achievements. For a history of the language and how the language definition came about, see MacQueen et al. [152].

Prehistory of Standard ML Verification

The definition of Standard ML [166] can be separated into two parts: the *Core* and the *Module* language, each with its own static and dynamic semantics. In the early 1990s, two ambitious project were undertaken to formalize Core ML [243, 259] and part of the Module system [153]. The proof assistants used at that time (HOL88 and HOL90) were quite primitive; hence most of the efforts were devoted to en-

hance their functionalities (support for mutual recursion) and libraries (finite maps). While the effort by VanInwegen [259] was frankly heroic, it did not yield the desired outcome, and she had to settle and prove determinism of evaluation, not only due to technological issues or some imprecisions in the definition itself, but because a naive named encoding of binders made proving properties of substitution so lengthy that time just ran out (ibidem, page 6).

Onward to Full Verification

A major milestone was the verification of type soundness of Standard ML, both Core ML and the Module system, completed in Twelf [139] using HOAS encodings to model bindings on both the term and the type level. The critical insight was to *elaborate* Standard ML into an internal language, based on the Harper–Stone interpretation [110], separating issues of scope resolution and type inference, and to state and prove soundness for the latter. The internal language includes various advanced features such as translucent modules, abstraction, polymorphism, higher kinds, references, exceptions, recursive types, and recursive functions. The success of this effort can be attributed both to the unique capabilities of HOAS and to the features of the internal language, which was designed with verification in mind. Along similar lines, Rossberg et al. [220] give an elaboration of a generalization of Standard ML's Module system into F_ω and prove its soundness in Coq with the logically nameless representation.

The verification of OCaml$_{light}$ by Owens [194] showcases the progress of mainstream proving technology. The goal and the techniques are not different from VanInwegen's: proving type soundness in HOL4 for OCaml minus modules and objects, still a nontrivial fact given the more than 300 rules describing the static and dynamic semantics. However, what made the difference was better automation of inductive definitions and of first-order theorem provers, as well as careful encoding techniques that relegated α-renaming to the typing rules (and the related proofs). Another factor is that, since there is no formal definition of OCaml, OCaml$_{light}$ is a rational reconstruction of the operational semantics of OCaml, again designed with verification in mind.

CakeML [135] introduces an approach that is different from post hoc verification, see the discussion in Section 18.3.

Type Inference, Verified

While verification of type soundness of Standard ML concentrates on the core language and the module system, a related effort aimed to verify Hindley–Milner style polymorphic type inference. Let-polymorphism is an interesting testbed for mechanization, since it involves formalizing type *schemas* and type *generalization*, which are awkward to encode without multiple binders. The handling of free variables in the W algorithm for type inference is also delicate. Early very labor-intensive

work used De Bruijn terms in Isabelle/HOL and named terms in Coq [84, 174]. The latter was redone with Nominal Isabelle [256] more elegantly. In the 2010s, more challenging problems have been tackled: for example, Garrigue [99] extended with considerable efforts soundness and completeness of type inference to structural polymorphism and recursive types, building on Aydemir et al.'s techniques [21].

18.2.2 Mechanization of Dependently Typed Proof Assistants

One long-standing open question is how to verify the metatheory of dependently typed proof assistants such as Coq, Nuprl, or Agda. The interest in this question is twofold: it represents a realistic example of formal mathematics, and it increases the trust in proof assistants themselves with the ultimate goal of achieving a formalized implementation of the core type checker for a full-scale implementation of type theory.

One of the first mechanization of pure type systems was developed by Pollack and collaborators [159, 208, 258] in the proof assistant Lego. A similar research program to ensure that Coq is consistent was led by Barras and Werner [27]. Barras also [26] formalized set-theoretic models of various type theories of the calculus of constructions family. More recently, Anand and Rahli [14] gave a formalization of Nurpl's type theory in Coq. Danielsson [73] and Chapman [56] have formalized normalizers for dependent types in Agda. Similarly, Altenkirch and Kaposi [9], Abel et al. [2], and Sozeau et al. [237] have pushed the program of formalizing type theory in type theory further, in particular with the goal of building a verified kernel of Coq.

18.2.3 Mechanization of Java-Like Languages

Since the first descriptions of Java became available growing concerns about the security of Java's code led to a long line of efforts to formalize Java's type system and operational semantics: from first posing the problem by Drossopoulou and Eisenbach [82] to a first answer by Nipkow and von Oheim [183] and to a definitive accomplishment by Klein and Nipkow [130]. This effort to further understand the type soundness of some of the features of Scala-like languages actively continues, in the form of implementations of the DOT calculus — see for example [12,102,216,217].

Bodin et al. [46] mechanized the semantics of JavaScript based on the official ECMA standard in Coq using pretty-big-step semantics [60]. They also provide a reference interpreter, which is proved correct w.r.t. the semantics, extracted to OCaml and exercised on the ECMA conformance test suite.

18.2.4 Mechanization of Query Languages

Proof assistants have been used to formalize the semantics and implementation of query languages such as SQL. The core of these languages is based on the theory of *relational algebras*, which has been formalized by Rajagopalan and Tsang in NuPrl [213], Gonzalia in Agda [104], and Oury and Swierstra in Agda [193]. Malecha et al. [154] verified an implementation of databases based on B+trees and some query optimizations in Coq. More complete query languages and more optimizations have been considered by Benzaken et al. [37] and Chu et al. [69], both in Coq.

The most complete work today is by Benzaken et al. [38], who verified an optimizing compiler, called DBCert, from a significant subset of SQL to JavaScript in Coq. Their work is based on the semantics of SQL by Benzaken et al. [36], which supports difficult aspects of SQL such as nested queries and NULL values, and on the optimizing compiler for nested relational algebras by Auerbach et al. [19].

18.2.5 Mechanization of Low-Level Languages

Low-level languages such as C, C++, and LLVM pose various challenges for mechanization. A first one is their pointer and memory representation, collectively called the *memory object model*. Low-level languages expose the bit-wise representation of data structures and pointers, but compilers perform high-level optimizations based on types and alias analysis. Another challenge is the languages' *relaxed concurrency memory model*. Low-level languages do not use a sequentially consistent model where memory accesses are simply interleaved, but allow memory accesses to appear out of order as a consequence of hardware and compiler optimizations. These challenges do not just concern mechanization in a proof assistant, but also the informal semantics in the official language standards [29, 160, 257].

The first attempt to mechanize the semantics of a subset of C in a proof assistant is by Cook and Subramanian in 1993 [70]. They defined it in terms of Lisp function in Nqthm. In 1998, Norrish mechanized a semantics of a larger subset of C in HOL as described in his PhD thesis [184]. A key aspect of Norrish's work is the semantics for expressions with side effects, which is underspecified according to the ISO C standard. In 2006, Leroy and collaborators defined a semantics of C in Coq [145, 146], which is used as the basis of the verified CompCert compiler (Section 18.3). A key ingredient of CompCert is the memory object model [148], which is shared between the C source language, the assembly target language, and all intermediate languages. In 2015, Krebbers defined a semantics for C in Coq as described in his PhD thesis [131]. He defined a small-step operational semantics, ab executable semantics (computable Coq function), a type system, and a program logic for C inspired by separation logic; further, he proved metatheoretical properties to formally relate these artifacts.

There has been significant work on the mechanization of the semantics of other low-level programming languages. For example, Norrish [186] and Ramananandro et al. [214] mechanized parts of the semantics of C++ in HOL and Coq, respectively. Zhao et al. [271], Lee et al. [140], and Zakowski et al. [270] mechanized parts of LLVM in Coq.

The aforementioned works typically focused on formalizing increasingly large subsets of the language, but have either considered a too concrete (e.g., Norrish's) or too abstract (e.g., CompCert's and Krebbers's) memory object model, prohibiting common compiler optimizations or programming patterns, respectively. To this end, there have been various mechanizations of hybrid memory models that take a middle ground [42, 126, 143]. Similarly, most of these works omitted concurrency, but there has been a plethora of research in proof assistants that focuses just on the relaxed concurrency aspects [29, 30, 125, 141, 229, 230, 257, ...].

18.2.6 Mechanization of Safe Low-Level Languages

Low-level languages such as C, C++, and LLVM have a very weak type system that provides little to no guarantees. Formally speaking, they do not enjoy a type soundness property that says that "well-typed programs cannot go wrong" [165] (i.e., cannot have undefined behavior). There has been a considerable amount of research into the design and theory of strongly typed safe low-level programming languages, see for example [170].

Rust is the first industry-supported language that uses the notion of *ownership* to control sharing of pointers and thereby guarantees memory safety and data-race freedom. The RustBelt project by Jung et al. [119] developed the first soundness proof for Rust, using the Iris separation logic framework [120, 122] in Coq. Contrary to the traditional approach of proving type soundness using the method of progress and preservation, RustBelt uses a semantic approach based on step-indexed logical relations [5]. This approach combines the soundness proof for the "safe" part of Rust with manually crafted proofs for "unsafe" libraries for locks, reference counted pointers, etc. Other languages with a notion of ownership are Mezzo [23] and Cogent [188]. Their metatheory has been mechanized in Coq and Isabelle, respectively.

WebAssembly is a typed low-level language that is implemented in all major web browsers. Watt [264] mechanized the official WebAssembly specification in Isabelle and proved type soundness using progress and preservation.

18.3 Mechanized Compiler Verification

Compiler verification is an active research topic due to its academically challenges and its practical relevance. It is challenging because (optimizing) compilers are sophisticated programs whose verification requires a combination of research con-

cerning semantics, algorithms, and proof assistants. It is practically relevant because compilers play an important role in the software pipeline. A compiler bug could turn a correct source program into an erroneous target program.

While mistakes in a compiler are less common than in source programs, research on compiler testing by Eide and Regehr [85] and Yang et al. [268] shows that all major (unverified) C compilers suffer from bugs that may silently miscompile programs. Miscompilation is especially worrisome for safety-critical programs that have been verified using formal methods (e.g., a proof assistant, model checker, or static analysis tool). Contrary to testing techniques that execute the target/compiled program, most formal methods operate at the level of the source program, so compiler bugs could invalidate all the work that went into the formal verification effort.

The idea of formally proving the correctness of a compiler goes back to at least McCarthy and Painter in 1967 [156]. They defined a semantics for a source language (arithmetic expressions) and target language (a stack machine) and proved that the result of running any source and target program is the same. While McCarthy and Painter carried out their proofs on paper, they set the goal to develop a formalism so that a machine could check the correctness of the compiler correctness result. This was first accomplished in 1972 by Milner and Weyhrauch [167] in the LCF proof assistant.

After this seminal work, compiler verification became an active area of research, typically focused on individual compilation passes, analyses, or toy languages; see the bibliography by Dave [75] for an overview. Moore [169] was arguably the first to verify a full compiler. His compiler used a high-level assembly-like source language called Piton and was verified using Nqthm. The first realistic verified compiler for a general-purpose programming language was developed by Leroy in 2006 in the CompCert project [145, 146]. CompCert is an optimizing multi-pass compiler for the C language whose correctness has been verified using Coq. The initial version of CompCert generated PowerPC assembly; current versions also support ARM, RISC-V, and x86 architectures.

One of the key insights of the CompCert breakthrough is to use Coq not just to specify and verify the compiler, but also as a purely functional programming language to implement the compiler. Coq's extraction mechanism [150] is used to turn the Coq definitions into executable OCaml code. Another key insight of CompCert, compared to traditional unverified compilers, is to use a relatively high number of intermediate languages (current versions have 20 passes and 8 intermediate languages), all with their own semantics. Many of these languages are similar, but they are designed so that they formalize the properties that compiler passes guarantee. To keep the development manageable in light of the high number of intermediate languages, CompCert factorizes out common parts, such as the memory object model [148].

Next, we take a closer look at how to formally state compiler correctness and conclude with an overview of current research on mechanized compiler verification. But first, let us show that compiler verification provides tangible benefits using a

quote from by Yang et al. [268] about the results of their Csmith tool for compiler fuzzing:

> 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.

Formalizing Compiler Correctness

For simple deterministic languages, one can say that a compiler is correct if for any source program the outcome of the resulting target program is the same as that of the source program. For more complicated languages, one has to take into account nondeterminism, nontermination, input/output, and undefined behavior, so a more refined notion of compiler correctness is needed. To see why, let us consider a simple C function:

```
int g() {
  return f1() + f2();
}
```

The evaluation order of operator and function arguments is nondeterministic according to the ISO C standard [115], so a compiler is free to decide to evaluate `f1` before `f2`, or vice versa. If `f1` and `f2` have side effects, this means that the outcome of the target function can be different depending on which evaluation order is selected by the compiler. Hence the compiler correctness statement should encompass both of these evaluation orders. Let us consider another C function:

```
int h(int *p, int x) {
  int z = *p;
  return f(x);
}
```

If the pointer `p` has been deallocated, then this function always goes wrong because accessing an invalid pointer has undefined behavior according to the ISO C standard. However, a compiler should not be expected to turn this function into a target function that also goes wrong. The variable `z` is unused, so dead code elimination may remove the variable declaration and pointer access `*p`, leading to a function that does not necessarily go wrong, but could do anything. CompCert uses the notion of a "safe backward simulation" [147, Definition 3] to account for nondeterminism and unsafe source programs—given a source program *src* and a target program produced by the compiler *tgt*, they should satisfy:

$$\text{If } \mathsf{safe}(\mathit{src}), \text{ then } \mathsf{behaviors}(\mathit{tgt}) \subseteq \mathsf{behaviors}(\mathit{src}) \tag{18.1}$$

Here, behaviors assigns a set of behaviors (say, termination with a certain return value, nontermination, undefined behavior / going wrong) to a target and source program, respectively. To allow the compiler to select a choice, as required by nondeterminism, the behaviors of the target should be a subset of those of the source, instead of both being the same. To make sure that wrong source programs do not need to go wrong in the target, the statement only holds for safe source programs (i.e., those that cannot go wrong). Compare this to formalizing the correctness statement for a higher-level language such as Java or ML, where safety is guaranteed by the type system, and one requires well-typedness instead of safety as a premise.

There are various directions in which the compiler correctness statement can be strengthened. First, the correctness statement (18.1) only accounts for the preservation of *functional* properties. Researchers have investigated verified compilers that additionally preserve nonfunctional properties such as I/O traces (as done in CompCert and CakeML), complexity [10] in the Cerco compiler, formalized in the Matita proof assistant, as well as memory consumption [42], stack bounds [263], and constant-time preservation [28], all three in CompCert. The CakeML compiler in HOL4 (discussed below) has also been proved space preserving [103]. Another nonfunctional property is secure compilation, which is discussed in Section **??**.

Second, the correctness statement (18.1) only accounts for programs that have been compiled as a *whole* by the same compiler. This is too weak of a statement because in practice programs are compiled on a per module/library basis, and finally linked together. This process is called *separate compilation*. Separate compilation improves the compilation speed, and ensures the compilation of modules/libraries written in different languages [4]. Correctness proofs for separate compilation are notoriously subtle, so many results have been mechanized in a proof assistant. Benton and Hur [34] and Neis et al. [176] developed techniques based on logical relations and simulations, respectively, for simple compilers, and mechanized their results in Coq. In the context of realistic languages, there are extensions of CompCert with support for separate compilation, notably by Ramananandro et al. [215], Stewart et al. [239] (Compositional CompCert), Kang et al. [127] (SepComp), and Song et al. [236] (CompCertM). Wang et al. [260] formalized in Coq an approach for separate compilation based on Hoare logic for a compiler from a C-like language called Cito.

Further Research on Mechanized Compiler Verification

In addition to the research on compiler verification that we discussed already, there are many more directions and projects. We discuss some of these.

- Since its inception in 2006, CompCert has been the subject of many extensions and it is available as a commercial product as of 2015 [3]. Examples of extensions are support for compilation of floating point arithmetic [47] and a verified parser [118]. Kästner et al. [136] report on the qualification of CompCert for IEC (International Electrotechnical Commission) standards for commercial use in safety-critical applications. CompCert has been extended with frontends

for different source languages. Dargane et al. developed a frontend for Mini-ML [74], the CertiCoq project [13] provides a frontend for Coq, and the Vélus compiler provides a frontend for the block diagram language Lustre [48].

- CakeML [135, 244, 245] is another major verified optimizing compiler, implemented and verified using HOL4. CakeML's source language is a significant subset of Standard ML, and its target languages are ARM, x86-54, MPIS-64, RISC-V, and Silver ISA. An interesting feature of CakeML is that it is bootstrapping and includes a verified type checker.
- An important aspect of the verification of a compiler for a high-level language (such as Java, ML, or Coq) is the formal verification of a garbage collector. Examples of such verification efforts are GCMinor for CompCert in Coq [158], a generational garbage collector for CakeML in HOL [88], and a generational garbage collector for CertiCoq in Coq [261].
- Other examples of verified compilers are the nonoptimizing bytecode compilers from Java to JVM by Strecker [240] and the already mentioned work by Nipkow and Klein [130] in Isabelle, verification of compilation passes for LLVM by Zhao et al. [272] and Kang et al. [128] in Coq, and the Cogent compiler by Amani et al. for a domain-specific language for file system implementations [11] in Isabelle. (See Section **??** for more discussion on Cogent.)
- Most of the verified compilers discussed so far target *sequential* languages. Lochbihler has verified a version of Java Threads [151] in Isabelle, and Sevcík et al. provide an extension of CompCert called CompCertTSO [230].
- One can also avoid the verification of a compiler pass or of a whole compiler by employing *translation validation* [175, 207]. Already from the beginning CompCert employed translation validation for the register allocation pass, and later extensions applied the technique to other passes [248–250]. Translation validation has been applied to the LLVM compiler [128], and has been used successfully in the seL4 project [232] with help of SMT (satisfiability modulo theories) solvers.
- There has also been work on synthesis of programs from definitions in proof assistants. Notable examples are the synthesis of ML programs from programs written in the higher-order logic of HOL by Myreen and Owens [173], and the Fiat [77] and Fiat-Crypto [86] frameworks for synthesis in Coq. (See Section **??** for more information about Fiat-Crypto.)
- A related endeavor is the verification of abstract interpreters and static analyzers. Noteworthy examples are the verification of static analyses used in CompCert [219], the verification of analyses of Java bytecode, for which Leroy provides an overview [144], and the verification of dataflow analyses in Coq by Cachera et al. [52]. A major project in this area is the Verasco static analyzer [117] built on top of the CompCert project.
- To obtain more compact proofs of verified compilers, Chlipala [65, 66] investigated the use of dependent types and proof automation in Coq. He applied his techniques to simply typed pure and impure functional languages.

18.4 Mechanized Program Logics

To prove properties of programs in a proof assistant, one could directly use the operational semantics of the programming language. While this approach works well for many metatheoretical properties, it immediately becomes unmanageable for the verification of concrete programs in languages that combine challenging concepts such as pointers, concurrency, higher-order functions, and unstructured control. It has therefore been a longtime challenge to develop *program logics* that provide reasoning principles at the right level of abstraction and tailored to the programming language of study.

As we will see in a moment, proofs assistants such as Coq, Isabelle, PVS, and HOL have been used to formalize the metatheory of program logics, including important results such as soundness of the program logic. Proof assistants have also been used to apply program logics to the verification of complex programs. When both of these parts are carried out together, this foundational approach gives a very strong result. By composing the proof of the program together with the soundness result of the program logic, the definition of the program logic is no longer part of the trusted definitions. One can go even further by connecting the program logic to a verified compiler such as CompCert, so that only the specified program property and the semantics of the target language (typically, assembly) needs to be trusted.

There exist many standalone tools that implement a program logic in one way or another, such as Boogie [25], Dafny [142], KeY [6], Viper [172], Vercors [45], Verifast [116], and Why3 [90]. Compared to program logics embedded in a proof assistant, the required level of trust is much higher for these tools. Indeed, one needs to trust the metatheory of the (often bespoke) program logic that these tools implement, as well as external SMT solvers used to discharge verification conditions. Some standalone tools have been combined with proof assistants. In particular, Why3 [90] allows verification conditions to be discharged using a combination of automatic theorem provers (such as Alt-Ergo, CVC4, Vampire, and Z3) and proof assistants (such as Coq, PVS, and Isabelle/HOL).

Hoare Logic

Program logics go back to the seminal work by Floyd [91] and Hoare [113] in the 1960s—they developed what is nowadays called Floyd–Hoare logic or just Hoare logic. The core ingredient of Hoare logic is the *Hoare triple* $\{P\}\,e\,\{Q\}$, which says that if the assertion P is true prior to the execution of program e, then the assertion Q is true after the execution of e. There are many variations on the exact formulation and semantics of Hoare triples, whether to consider total or partial correctness (i.e., with or without ensuring termination), how to deal with run-time errors, nondeterminism, undefined behavior, etc. The most important part is that the assertions P and Q are meant to specify the state of the program, and that Hoare triples can be derived using a set of structural proof rules that abstract from the programming language operational semantics.

Early mechanized results about soundness of Hoare logic go back to Sokolowski [235] in LCF and Gordon [105] in HOL. Nipkow [179] mechanized relative completeness in Isabelle/HOL. Subsequently, researchers mechanized versions of Hoare logic with features such as nondeterminism and recursive functions [181], and concurrency using the Owicki–Gries [182] and Rely-Guarantee [178] method. Variants of Hoare logic are also used in work on security, see Section **??** for discussion and examples.

Separation Logic

An open question since the inception of Hoare logic was how to reason about pointers and data structures on the heap, especially in the context of linked data structures and concurrency. In 2001, O'Hearn, Reynolds, and Yang invented separation logic [192, 218], an extension of Hoare logic to answer this question. O'Hearn and Brookes extended separation logic to support concurrency in 2004 [50, 189]. (Concurrent) separation logic has had tremendous success with many extensions and applications in theory, tools, and real-world program verification. In this section we focus on its use in proof assistants, while historical surveys can be found in [49,190].

Separation logic is based on the key idea that the *points-to assertion* $l \mapsto v$ does not just state that location l contains value v, but it also provides unique ownership of that location. This is crucial to enable local reasoning—it makes sure that an assignment via one pointer does not change the values at other locations. As a result, specifications only need to talk about the part of the heap on which the program operates, and nothing else. Unique ownership is also crucial for reasoning about concurrency—it ensures that one thread cannot touch other data (unless this data is explicitly shared via some invariant).

Separation logic is a substructural logic, concretely an instance of BI (Bunched Implication) [191]. Its key connective is the *separating conjunction* $P * Q$, which says that the heap can be subdivided into two disjoint parts, one described by P and another described by Q. The quintessential rule for local reasoning is the *frame rule*. This rule says that given a Hoare triple $\{P\}\, e\, \{Q\}$, one can derive a version $\{P * R\}\, e\, \{Q * R\}$, in which the heaps and the precondition and postcondition are extended with a heap described by R. The frame rules makes it possible to write specifications that only describe the parts P and Q of the heap that are relevant to the program e.

Early mechanizations of separation logic in a proof assistant go back to Weber in Isabelle [265], Marti et al. [155] and Appel in Coq [15], and Preoteasa in PVS [212]. In this early work, as well as in the vast majority of recent work (some of that will be discussed below), separation logic is represented using a shallow embedding [267]. Let us exemplify that with the simplest form of separation logic: classical separation logic with heaps as resources. Assertions are represented as heap predicates, entailment $P \vdash Q$ is defined as $\forall h.\ P\,h \to Q\,h$, and the separation logic connectives are defined as combinators on heap predicates. Below we give the definitions of the most important connectives:

$$\begin{aligned}
\mathsf{emp} &:= \lambda h.\ h = \emptyset \\
P * Q &:= \lambda h.\ \exists h_1, h_2.\ \mathsf{dom}(h_1) \cap \mathsf{dom}(h_2) = \emptyset \wedge h = h_1 \cup h_2 \wedge P\,h_1 \wedge Q\,h_2 \\
l \mapsto v &:= \lambda h.\ \mathsf{dom}(h) = \{l\} \wedge h(l) = v \\
\lceil \phi \rceil &:= \lambda h.\ \phi \\
\exists x : A.\ P\,x &:= \lambda h.\ \exists x : A.\ P\,x\,h
\end{aligned}$$

An important feature of a shallow embedding of separation logic is that when using a proof assistant based on higher-order logic (such as HOL or Coq) as the metalogic, one can reuse a number of features of the metalogic in the object logic. First, using the $\lceil \cdot \rceil$ connective, one can embed any proposition ϕ of the proof assistant metalogic into the object logic. Second, the type A in the quantifier $\exists x : A.\ P\,x$ is a type in the metalogic. The previous two points imply that one immediately gets support for the theory of conventional data types such as numbers, lists, sets, maps, and functions. The second point also implies that one gets higher-order impredicative quantification for free since proof assistants based on higher-order logic allow quantification over any type, including the type of separation logic propositions itself. Higher-order quantification is particularly important to embed Hoare triples in assertions, and it is useful to reason about higher-order programs. Third, lambda abstractions of the proof assistant can be used to define the binding in the quantifier $\exists x : A.\ P\,x$, which means that no (first-order) representation of binding needs to be mechanized.

Proofs in Separation Logic

An important challenge is how to use a proof assistant to construct and verify proofs of entailments and Hoare triples in separation logic. The naive approach is to unfold the definitions of the separation logic connectives and to reason about the underlying representation (i.e., heap predicates in the simple model of classical separation logic described above). As already noticed in early papers on separation logic in proof assistants, e.g., by Appel [15], this is unfeasible, because one has to reason about heap disjointness for each occurrence of the separation conjunction. The situation worsens when considering richer variants of separation logic, e.g., those with modalities for recursion [80, 81]. A more pleasant and scalable approach is to use the proof rules of separation logic which hide heap disjointness. For example:

$$P * Q \vdash Q * P \qquad \frac{P_1 \vdash Q_1 \quad P_2 \vdash Q_2}{P_1 * P_2 \vdash Q_1 * Q_2} \qquad P * (\exists x.\ Q) \vdash (\exists x.\ P * Q) \qquad \frac{\forall x.\ (P \vdash Q)}{(\exists x.\ P) \vdash Q}$$

To effectively use these rules, all existing state-of-the-art separation logic frameworks in proof assistant come with tactics to provide appropriate infrastructure for interactive or automated proofs.

Work on tactics for interactive proofs goes back to Appel [15], who developed tactics in Coq to help out with basic bookkeeping in interactive separation-logic

proofs. This line of work was later continued and extended by McCreight [157] and Bengtson et al. [32]. The latest development in this line of work is the Iris Proof Mode (IPM) by Krebbers et al. [132, 133], which extends Coq with spatial and nonspatial contexts for separation logic and provides a rich set of tactics for making separation logic proofs look and feel like ordinary Coq proofs. IPM support many features of modern concurrent separation logics such as modalities [80, 81], higher-order quantification, impredicative invariants [241], and a variety of specifications including logical refinement [252]. IPM is implemented using a combination of programming in Ltac (a scripting language for Coq proofs), logic programming with type classes, and computational reflection; methods discussed in Sections **??**, **??**, and **??**, respectively.

Research on automatic proofs for separation logic in proof assistants goes back to Tuerk [251] and Appel [16], who developed versions of the automatic Smallfoot assert checker [39] in HOL (called HOLFoot) and Coq (called Verismall), respectively. These automatic checkers provide stronger automation than their interactive counterparts, but their logical expressivity is limited. Verismall is restricted to shape properties and only supports lists and trees. While HOLFoot goes beyond shape properties, assistance through interactive proof is needed for functional correctness proofs of more complicated programs.

To address the trade-off between automation and logical expressivity, one can let the user provide hints to guide the proof search. In the context of separation-logic based verification, this approach has been used successfully in Bedrock by Chlipala [67], RefinedC by Sammler et al. [222], and Diaframe by Mulder et al. [171]. These tools aim to automatically take care of the memory-related (i.e., separation-logic) parts of proofs, but require the user to provide custom theories and prove properties about mathematical objects (such as numbers, lists, and sets) when their domain-specific automation cannot handle them automatically.

We conclude this chapter by giving some examples of state-of-the-art separation logic frameworks in proof assistants, in alphabetical order.

- Bedrock [67] is a framework for verification of low-level programs in Coq. The basis of Bedrock is an assembly language, on top of which custom DSLs are implemented using macros. Program specifications are written in a form of higher-order separation logic. The distinguishing feature of Bedrock is its strong support for automation using a combination of techniques for separation logic, Ltac tactic programming, and reification. Bedrock has been used among others to verify multithreaded web applications [68].
- CFML [58] is a framework for verification of OCaml programs in Coq. It contains a frontend that translates OCaml programs into abstract syntax in Coq. Program specifications are written in a form of higher-order separation logic. CFML uses a relatively simple model of separation logic and is therefore well-suited for education [61]. It is used to verify data structures and algorithms, e.g., a cycle detection algorithm [107], and has been extended with support for reasoning about time complexity [62]. CFML's approach has been ported to the CakeML compiler, and extended with support for exceptions and I/O [108].

- FCSL [227, 228] is a framework for the verification of concurrent programs in Coq. It focuses on fine-grained concurrency, i.e., concurrency with low-level primitives such as compare-and-swap instead of coarse-grained synchronization mechanisms such as locks. Programs are written in a monadic language embedded in Coq's dependent type theory. Specifications are given using a combination of state-transition systems and partial commutative monoids. FCSL has been used among others to verify an optimal snapshot algorithm [78].
- Iris [120, 122] is a language-independent framework for higher-order fine-grained concurrent separation logic in Coq. Iris has been used with instantiations based on ML [120], Rust [119], Go [55], C [222], Scala [102], capability machines [100], distributed systems [134], languages with session types [112], and languages with relaxed-memory concurrency [72, 123, 162]. Iris supports traditional Hoare-style specifications, as well as specifications for program refinements [94, 98], logical atomicity [121], time complexity [161], space bounds [168], crash safety [55], and security properties such as non-interference [95,106]. In addition, Iris has been used to develop semantics models of type systems such as Rust [119] and Scala [102].
- Sepref [137] is a refinement framework for the verification of imperative HOL programs [51] built on top of the Isabelle Refinement Framework [138]. Sepref uses a simple model of separation logic based on heap predicates and is aimed at the verification of imperative sequential data structures such as depth-first search and shortest path algorithms. It provides automation via tactics. Haslbeck and Lammich produce, via refinement, efficient LLVM code, e.g., for a state-of-the-art sorting algorithm [111].
- Steelcore [242] is an Iris-inspired concurrent separation logic embedded in the dependently typed language F*. Steel [93] extends Steelcore with automation based on tactics and SMT, ans has been used on among others concurrent linked data structures and a library for 2-party session types.
- VST [17,53] is a framework for the verification of C programs in Coq. The key component of VST is Verifiable C, a program logic for C based on higher-order separation logic. Verifiable C is proved sound with respect to the operational semantics of CompCert's Clight. Together with the CompCert compiler's correctness theorem, VST gives strong end-to-end correctness guarantees about the assembly code. Among others, VST has been applied to the verification of cryptographic primitives such as SHA [18], HMAC [41, 269], and X25519 [226].

References

1. Andreas Abel, Guillaume Allais, Aliya Hameer, Brigitte Pientka, Alberto Momigliano, Steven Schäfer, and Kathrin Stark. POPLMark Reloaded: Mechanizing proofs by logical relations. *J. Funct. Program.*, 29:e19, 2019.
2. Andreas Abel, Joakim Öhman, and Andrea Vezzosi. Decidability of conversion for type theory in type theory. *Proc. ACM Program. Lang.*, 2(POPL):23:1–23:29, 2018.

3. AbsInt. Formally verified compilation, 2024. Website, `https://www.absint.com/compcert/`.
4. Amal Ahmed. Verified compilers for a multi-language world. In Thomas Ball, Rastislav Bodík, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett, editors, *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA*, volume 32 of *LIPIcs*, pages 15–31. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015.
5. Amal Jamil Ahmed. *Semantics of types for mutable state*. PhD thesis, Princeton University, 2004.
6. Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Martin Giese, Elmar Habermalz, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, and Peter H. Schmitt. The key system: Integrating object-oriented design and formal methods. In Ralf-Detlef Kutsche and Herbert Weber, editors, *Fundamental Approaches to Software Engineering, 5th International Conference, FASE 2002, held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, volume 2306 of *Lecture Notes in Computer Science*, pages 327–330. Springer, 2002.
7. Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. A type and scope safe universe of syntaxes with binding: their semantics and proofs. *Proc. ACM Program. Lang.*, 2(ICFP):90:1–90:30, 2018.
8. Thorsten Altenkirch. A formalization of the strong normalization proof for system F in LEGO. In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda Calculi and Applications, International Conference on Typed Lambda Calculi and Applications, TLCA '93, Utrecht, The Netherlands, March 16-18, 1993, Proceedings*, volume 664 of *Lecture Notes in Computer Science*, pages 13–28. Springer, 1993.
9. Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016*, pages 18–29. ACM, 2016.
10. Roberto M. Amadio, Nicholas Ayache, François Bobot, Jaap Boender, Brian Campbell, Ilias Garnier, Antoine Madet, James McKinna, Dominic P. Mulligan, Mauro Piccolo, Randy Pollack, Yann Régis-Gianas, Claudio Sacerdoti Coen, Ian Stark, and Paolo Tranquilli. Certified complexity (cerco). In Ugo Dal Lago and Ricardo Peña, editors, *Foundational and Practical Aspects of Resource Analysis—Third International Workshop, FOPARA 2013, Bertinoro, Italy, August 29-31, 2013, Revised Selected Papers*, volume 8552 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2013.
11. Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby C. Murray, Gerwin Klein, and Gernot Heiser. Cogent: Verifying high-assurance file system implementations. In Tom Conte and Yuanyuan Zhou, editors, *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016*, pages 175–188. ACM, 2016.
12. Nada Amin and Tiark Rompf. Type soundness proofs with definitional interpreters. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 666–679. ACM, 2017.
13. Abhishek Anand, Andrew W. Appel, Zoe Paraskevopoulou J. Gregory Morrisett, Randy Pollack, Olivier Savary Bélanger, Matthieu Sozeau, and Matthew Weaver. Certicoq: A verified compiler for coq, 2017. In CoqPL'17: The Third International Workshop on Coq for Programming Languages.
14. Abhishek Anand and Vincent Rahli. Towards a formally verified proof assistant. In *Interactive Theorem Proving—5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8558 of *Lecture Notes in Computer Science*, pages 27–44. Springer, 2014.

15. Andrew W. Appel. Tactics for separation logic, 2006. Draft, `https://www.cs.princeton.edu/~appel/papers/septacs.pdf`.
16. Andrew W. Appel. Verismall: Verified smallfoot shape analysis. In Jean-Pierre Jouannaud and Zhong Shao, editors, *Certified Programs and Proofs—First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings*, volume 7086 of *Lecture Notes in Computer Science*, pages 231–246. Springer, 2011.
17. Andrew W. Appel. *Program Logics—for Certified Compilers*. Cambridge University Press, 2014.
18. Andrew W. Appel. Verification of a cryptographic primitive: SHA-256. *ACM Trans. Program. Lang. Syst.*, 37(2):7:1–7:31, 2015.
19. Joshua S. Auerbach, Martin Hirzel, Louis Mandel, Avraham Shinnar, and Jérôme Siméon. Q*cert: A platform for implementing and verifying query compilers. In Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1703–1706. ACM, 2017.
20. Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The poplmark challenge. In Joe Hurd and Thomas F. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, volume 3603 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2005.
21. Brian E. Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 3–15. ACM, 2008.
22. David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu, and Yuting Wang. Abella: A system for reasoning about relational specifications. *J. Formaliz. Reason.*, 7(2):1–89, 2014.
23. Thibaut Balabonski, François Pottier, and Jonathan Protzenko. The design and formalization of mezzo, a permission-based programming language. *ACM Trans. Program. Lang. Syst.*, 38(4):14:1–14:94, 2016.
24. Hendrik Pieter Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in logic and the foundations of mathematics*. North-Holland, 1985.
25. Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005.
26. Bruno Barras. Sets in Coq, Coq in sets. *J. Formaliz. Reason.*, 3(1):29–48, 2010.
27. Bruno Barras and Benjamin Werner. Coq in Coq, 1997.
28. Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. Formal verification of a constant-time preserving C compiler. *Proc. ACM Program. Lang.*, 4(POPL):7:1–7:30, 2020.
29. Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. The problem of programming language concurrency semantics. In Jan Vitek, editor, *Programming Languages and Systems—24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 283–307. Springer, 2015.
30. Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 55–66. ACM, 2011.

31. Olivier Savary Bélanger, Stefan Monnier, and Brigitte Pientka. Programming type-safe transformations using higher-order abstract syntax. *J. Formaliz. Reason.*, 8(1):49–91, 2015.
32. Jesper Bengtson, Jonas Braband Jensen, and Lars Birkedal. Charge!—A framework for higher-order separation logic in coq. In Lennart Beringer and Amy P. Felty, editors, *Interactive Theorem Proving—Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. Proceedings*, volume 7406 of *Lecture Notes in Computer Science*, pages 315–331. Springer, 2012.
33. Jesper Bengtson and Joachim Parrow. Formalising the pi-calculus using nominal logic. *Log. Methods Comput. Sci.*, 5(2), 2009.
34. Nick Benton and Chung-Kil Hur. Biorthogonality, step-indexing and compiler correctness. In Graham Hutton and Andrew P. Tolmach, editors, *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31—September 2, 2009*, pages 97–108. ACM, 2009.
35. Nick Benton, Chung-Kil Hur, Andrew Kennedy, and Conor McBride. Strongly typed term representations in Coq. *J. Autom. Reason.*, 49(2):141–159, 2012.
36. Véronique Benzaken and Evelyne Contejean. A coq mechanised formal semantics for realistic SQL queries: formally reconciling SQL and bag relational algebra. In Assia Mahboubi and Magnus O. Myreen, editors, *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, pages 249–261. ACM, 2019.
37. Véronique Benzaken, Evelyne Contejean, and Stefania Dumbrava. A coq formalization of the relational data model. In Zhong Shao, editor, *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8410 of *Lecture Notes in Computer Science*, pages 189–208. Springer, 2014.
38. Véronique Benzaken, Evelyne Contejean, Mohammed Houssem Hachmaoui, Chantal Keller, Louis Mandel, Avraham Shinnar, and Jérôme Siméon. Translating canonical SQL to imperative code in coq. *Proc. ACM Program. Lang.*, 6(OOPSLA1):1–27, 2022.
39. Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 2005.
40. Stefan Berghofer and Christian Urban. A head-to-head comparison of de bruijn indices and names. In Alberto Momigliano and Brigitte Pientka, editors, *Proceedings of the First International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, LFMTP@FLoC 2006, Seattle, WA, USA, August 16, 2006*, volume 174 of *Electronic Notes in Theoretical Computer Science*, pages 53–67. Elsevier, 2006.
41. Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. Verified correctness and security of openssl HMAC. In Jaeyeon Jung and Thorsten Holz, editors, *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, pages 207–221. USENIX Association, 2015.
42. Frédéric Besson, Sandrine Blazy, and Pierre Wilke. Compcerts: A memory-aware verified C compiler using pointer as integer semantics. In Mauricio Ayala-Rincón and César A. Muñoz, editors, *Interactive Theorem Proving—8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings*, volume 10499 of *Lecture Notes in Computer Science*, pages 81–97. Springer, 2017.
43. Richard S. Bird and Ross Paterson. De bruijn notation as a nested datatype. *J. Funct. Program.*, 9(1):77–91, 1999.
44. Jasmin Christian Blanchette, Lorenzo Gheri, Andrei Popescu, and Dmitriy Traytel. Bindings as bounded natural functors. *PACMPL*, 3(POPL):22:1–22:34, 2019.
45. Stefan Blom, Saeed Darabi, Marieke Huisman, and Wytse Oortwijn. The vercors tool set: Verification of parallel and concurrent software. In Nadia Polikarpova and Steve A.

Schneider, editors, *Integrated Formal Methods—13th International Conference, IFM 2017, Turin, Italy, September 20-22, 2017, Proceedings*, volume 10510 of *Lecture Notes in Computer Science*, pages 102–110. Springer, 2017.
46. Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. A trusted mechanised javascript specification. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 87–100. ACM, 2014.
47. Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. A formally-verified C compiler supporting floating-point arithmetic. In Alberto Nannarelli, Peter-Michael Seidel, and Ping Tak Peter Tang, editors, *21st IEEE Symposium on Computer Arithmetic, ARITH 2013, Austin, TX, USA, April 7-10, 2013*, pages 107–115. IEEE Computer Society, 2013.
48. Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. A formally verified compiler for lustre. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 586–601. ACM, 2017.
49. Stephen Brookes and Peter W. O'Hearn. Concurrent separation logic. *ACM SIGLOG News*, 3(3):47–65, 2016.
50. Stephen D. Brookes. A semantics for concurrent separation logic. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR 2004—Concurrency Theory, 15th International Conference, London, UK, August 31—September 3, 2004, Proceedings*, volume 3170 of *Lecture Notes in Computer Science*, pages 16–34. Springer, 2004.
51. Lukas Bulwahn, Alexander Krauss, Florian Haftmann, Levent Erkök, and John Matthews. Imperative functional programming with isabelle/hol. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, volume 5170 of *Lecture Notes in Computer Science*, pages 134–149. Springer, 2008.
52. David Cachera, Thomas P. Jensen, David Pichardie, and Vlad Rusu. Extracting a data flow analyser in constructive logic. In David A. Schmidt, editor, *Programming Languages and Systems, 13th European Symposium on Programming, ESOP 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 —April 2, 2004, Proceedings*, volume 2986 of *Lecture Notes in Computer Science*, pages 385–400. Springer, 2004.
53. Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. Vst-floyd: A separation logic tool to verify correctness of C programs. *J. Autom. Reason.*, 61(1-4):367–422, 2018.
54. Marco Carbone, David Castro-Perez, Francisco Ferreira, Lorenzo Gheri, Frederik Krogsdal Jacobsen, Alberto Momigliano, Luca Padovani, Alceste Scalas, Dawit Legesse Tirore, Martin Vassor, Nobuko Yoshida, and Daniel Zackon. The concurrent calculi formalisation benchmark. In Ilaria Castellani and Francesco Tiezzi, editors, *Coordination Models and Languages - 26th IFIP WG 6.1 International Conference, COORDINATION 2024, Held as Part of the 19th International Federated Conference on Distributed Computing Techniques, DisCoTec 2024, Groningen, The Netherlands, June 17-21, 2024, Proceedings*, volume 14676 of *Lecture Notes in Computer Science*, pages 149–158. Springer, 2024.
55. Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying concurrent, crash-safe systems with perennial. In Tim Brecht and Carey Williamson, editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 243–258. ACM, 2019.
56. James Chapman. Type theory should eat itself. *Electron. Notes Theor. Comput. Sci.*, 228:21–36, 2009.
57. Arthur Charguéraud. Locally nameless representation with cofinite quantification. `https://www.charygueraud.org/softs/ln`.

58. Arthur Charguéraud. Program verification through characteristic formulae. In Paul Hudak and Stephanie Weirich, editors, *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 321–332. ACM, 2010.
59. Arthur Charguéraud. The locally nameless representation. *J. Autom. Reason.*, 49(3):363–408, 2012.
60. Arthur Charguéraud. Pretty-big-step semantics. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7792 of *Lecture Notes in Computer Science*, pages 41–60. Springer, 2013.
61. Arthur Charguéraud. Separation logic for sequential programs (functional pearl). *Proc. ACM Program. Lang.*, 4(ICFP):116:1–116:34, 2020.
62. Arthur Charguéraud and François Pottier. Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits. *J. Autom. Reason.*, 62(3):331–365, 2019.
63. Xiaohong Chen and Grigore Rosu. A general approach to define binders using matching logic. *Proc. ACM Program. Lang.*, 4(ICFP):88:1–88:32, 2020.
64. James Cheney. A dependent nominal type theory. *Log. Methods Comput. Sci.*, 8(1), 2012.
65. Adam Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In Jeanne Ferrante and Kathryn S. McKinley, editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 54–65. ACM, 2007.
66. Adam Chlipala. A verified compiler for an impure functional language. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 93–106. ACM, 2010.
67. Adam Chlipala. The bedrock structured programming system: combining generative metaprogramming and hoare logic in an extensible program verifier. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA—September 25—27, 2013*, pages 391–402. ACM, 2013.
68. Adam Chlipala. From network interface to multithreaded web applications: A case study in modular program verification. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 609–622. ACM, 2015.
69. Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. Hottsql: proving query rewrites with univalent SQL semantics. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 510–524. ACM, 2017.
70. Jeffrey Cook and Sakthi Subramanian. A formal semantics for C in Nqthm. Technical Report 517D, Trusted Information Systems, 1994.
71. Ernesto Copello, Nora Szasz, and Alvaro Tasistro. Formal metatheory of the lambda calculus using stoughton's substitution. *Theor. Comput. Sci.*, 685:65–82, 2017.
72. Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. Rustbelt meets relaxed memory. *Proc. ACM Program. Lang.*, 4(POPL):34:1–34:29, 2020.
73. Nils Anders Danielsson. A formalisation of a dependently typed language as an inductive-recursive family. In *Types for Proofs and Programs, International Workshop, (TYPES'06), Revised Selected Papers*, volume 4502 of *Lecture Notes in Computer Science*, pages 93–109. Springer, 2006.
74. Zaynah Dargaye and Xavier Leroy. Mechanized verification of CPS transformations. In Nachum Dershowitz and Andrei Voronkov, editors, *Logic for Programming, Artificial*

Intelligence, and Reasoning, 14th International Conference, LPAR 2007, Yerevan, Armenia, October 15-19, 2007, Proceedings, volume 4790 of *Lecture Notes in Computer Science*, pages 211–225. Springer, 2007.
75. Maulik A. Dave. Compiler verification: a bibliography. *ACM SIGSOFT Softw. Eng. Notes*, 28(6):2, 2003.
76. N.G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church–Rosser Theorem. indagationes math, 34, 5, p. 381-392. In R.P. Nederpelt, J.H. Geuvers, and R.C. de Vrijer, editors, *Selected Papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics*, pages 375—388. Elsevier, 1994.
77. Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. Fiat: Deductive synthesis of abstract data types in a proof assistant. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 689–700. ACM, 2015.
78. Germán Andrés Delbianco, Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Concurrent data structures linked in time. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, volume 74 of *LIPIcs*, pages 8:1–8:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.
79. Joëlle Despeyroux, Amy P. Felty, and André Hirschowitz. Higher-order abstract syntax in Coq. In Mariangiola Dezani-Ciancaglini and Gordon D. Plotkin, editors, *Typed Lambda Calculi and Applications, Second International Conference on Typed Lambda Calculi and Applications, TLCA '95, Edinburgh, UK, April 10-12, 1995, Proceedings*, volume 902 of *Lecture Notes in Computer Science*, pages 124–138. Springer, 1995.
80. Robert Dockins, Andrew W. Appel, and Aquinas Hobor. Multimodal separation logic for reasoning about operational semantics. In Andrej Bauer and Michael W. Mislove, editors, *Proceedings of the 24th Conference on the Mathematical Foundations of Programming Semantics, MFPS 2008, Philadelphia, PA, USA, May 22-25, 2008*, volume 218 of *Electronic Notes in Theoretical Computer Science*, pages 5–20. Elsevier, 2008.
81. Derek Dreyer, Georg Neis, Andreas Rossberg, and Lars Birkedal. A relational modal logic for higher-order stateful adts. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 185–198. ACM, 2010.
82. Sophia Drossopoulou and Susan Eisenbach. Java is type safe—probably. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97—Object-Oriented Programming, 11th European Conference, Jyväskylä, Finland, June 9-13, 1997, Proceedings*, volume 1241 of *Lecture Notes in Computer Science*, pages 389–418. Springer, 1997.
83. Catherine Dubois. Proving ML type soundness within Coq. In Mark Aagaard and John Harrison, editors, *Theorem Proving in Higher Order Logics, 13th International Conference, TPHOLs 2000, Portland, Oregon, USA, August 14-18, 2000, Proceedings*, volume 1869 of *Lecture Notes in Computer Science*, pages 126–144. Springer, 2000.
84. Catherine Dubois and Valérie Ménissier-Morain. Certification of a type inference tool for ML: damas-milner within Coq. *J. Autom. Reasoning*, 23(3-4):319–346, 1999.
85. Eric Eide and John Regehr. Volatiles are miscompiled, and what to do about it. In Luca de Alfaro and Jens Palsberg, editors, *Proceedings of the 8th ACM & IEEE International conference on Embedded software, EMSOFT 2008, Atlanta, GA, USA, October 19-24, 2008*, pages 255–264. ACM, 2008.
86. Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic—with proofs, without compromises. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 1202–1219. IEEE, 2019.
87. Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P.

Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Comput. Lang. Syst. Struct.*, 44:24–47, 2015.

88. Adam Sandberg Ericsson, Magnus O. Myreen, and Johannes Åman Pohjola. A verified generational garbage collector for cakeml. In Mauricio Ayala-Rincón and César A. Muñoz, editors, *Interactive Theorem Proving—8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings*, volume 10499 of *Lecture Notes in Computer Science*, pages 444–461. Springer, 2017.
89. Amy P. Felty and Alberto Momigliano. Hybrid—A definitional two-level approach to reasoning with higher-order abstract syntax. *J. Autom. Reason.*, 48(1):43–105, 2012.
90. Jean-Christophe Filliâtre and Claude Marché. The why/krakatoa/caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177. Springer, 2007.
91. Robert Floyd. Assigning meaning to programs. In *Mathematical Aspects of Computer Science*, number 19 in Proceedings of Symposia in Applied Mathematics, pages 19–32, 1967.
92. Jonathan Ford and Ian A. Mason. Formal foundations of operational semantics. *High. Order Symb. Comput.*, 16(3):161–202, 2003.
93. Aymeric Fromherz, Aseem Rastogi, Nikhil Swamy, Sydney Gibson, Guido Martínez, Denis Merigoux, and Tahina Ramananandro. Steel: proof-oriented programming in a dependently typed concurrent separation logic. *Proc. ACM Program. Lang.*, 5(ICFP):1–30, 2021.
94. Dan Frumin, Robbert Krebbers, and Lars Birkedal. Reloc: A mechanised relational logic for fine-grained concurrency. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 442–451. ACM, 2018.
95. Dan Frumin, Robbert Krebbers, and Lars Birkedal. Compositional non-interference for fine-grained concurrent programs. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 1416–1433. IEEE, 2021.
96. Murdoch Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects Comput.*, 13(3-5):341–363, 2002.
97. Andrew Gacek, Dale Miller, and Gopalan Nadathur. Nominal abstraction. *Inf. Comput.*, 209(1):48–73, 2011.
98. Lennard Gäher, Michael Sammler, Simon Spies, Ralf Jung, Hoang-Hai Dang, Robbert Krebbers, Jeehoon Kang, and Derek Dreyer. Simuliris: a separation logic framework for verifying concurrent program optimizations. *Proc. ACM Program. Lang.*, 6(POPL):1–31, 2022.
99. Jacques Garrigue. A certified implementation of ML with structural polymorphism and recursive types. *Mathematical Structures in Computer Science*, 25(4):867–891, 2015.
100. Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, and Lars Birkedal. Efficient and provable local capability revocation using uninitialized capabilities. *Proc. ACM Program. Lang.*, 5(POPL):1–30, 2021.
101. Lorenzo Gheri and Andrei Popescu. A formalized general theory of syntax with bindings: Extended version. *J. Autom. Reason.*, 64(4):641–675, 2020.
102. Paolo G. Giarrusso, Léo Stefanesco, Amin Timany, Lars Birkedal, and Robbert Krebbers. Scala step-by-step: soundness for DOT with step-indexed logical relations in iris. *Proc. ACM Program. Lang.*, 4(ICFP):114:1–114:29, 2020.
103. Alejandro Gómez-Londoño, Johannes Åman Pohjola, Hira Taqdees Syeda, Magnus O. Myreen, and Yong Kiam Tan. Do you have space for dessert? a verified space cost semantics for cakeml programs. *Proc. ACM Program. Lang.*, 4(OOPSLA):204:1–204:29, 2020.

104. Carlos Gonzalia. *Relations in Dependent Type Theory*. PhD thesis, 2006.
105. Michael J. C. Gordon. Mechanizing programming logics in higher order logic. In Graham Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 387–439. Springer New York, New York, NY, 1989.
106. Simon Oddershede Gregersen, Johan Bay, Amin Timany, and Lars Birkedal. Mechanized logical relations for termination-insensitive noninterference. *Proc. ACM Program. Lang.*, 5(POPL):1–29, 2021.
107. Armaël Guéneau, Jacques-Henri Jourdan, Arthur Charguéraud, and François Pottier. Formal proof and analysis of an incremental cycle detection algorithm. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA*, volume 141 of *LIPIcs*, pages 18:1–18:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.
108. Armaël Guéneau, Magnus O. Myreen, Ramana Kumar, and Michael Norrish. Verified characteristic formulae for cakeml. In Hongseok Yang, editor, *Programming Languages and Systems—26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10201 of *Lecture Notes in Computer Science*, pages 584–610. Springer, 2017.
109. Robert Harper and Daniel R. Licata. Mechanizing metatheory in a logical framework. *J. Funct. Program.*, 17(4-5):613–673, 2007.
110. Robert Harper and Christopher A. Stone. A type-theoretic interpretation of standard ML. In Gordon D. Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pages 341–388. The MIT Press, 2000.
111. Maximilian P. L. Haslbeck and Peter Lammich. For a few dollars more: Verified fine-grained algorithm analysis down to LLVM. *ACM Trans. Program. Lang. Syst.*, 44(3):14:1–14:36, 2022.
112. Jonas Kastberg Hinrichsen, Daniël Louwrink, Robbert Krebbers, and Jesper Bengtson. Machine-checked semantic session typing. In *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*, pages 178–198. ACM, 2021.
113. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
114. Furio Honsell, Marino Miculan, and Ivan Scagnetto. An axiomatic approach to metareasoning on nominal algebras in HOAS. In Fernando Orejas, Paul G. Spirakis, and Jan van Leeuwen, editors, *Automata, Languages and Programming, 28th International Colloquium, ICALP 2001, Crete, Greece, July 8-12, 2001, Proceedings*, volume 2076 of *Lecture Notes in Computer Science*, pages 963–978. Springer, 2001.
115. ISO. *ISO/IEC 9899-2011: Programming languages – C*. ISO Working Group 14, 2012.
116. Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for C and java. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods—Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, volume 6617 of *Lecture Notes in Computer Science*, pages 41–55. Springer, 2011.
117. Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A formally-verified C static analyzer. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 247–259. ACM, 2015.
118. Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. Validating LR(1) parsers. In Helmut Seidl, editor, *Programming Languages and Systems—21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 —April 1, 2012. Proceedings*, volume 7211 of *Lecture Notes in Computer Science*, pages 397–416. Springer, 2012.

119. Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL):66:1–66:34, 2018.
120. Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.*, 28:e20, 2018.
121. Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. The future is ours: prophecy variables in separation logic. *Proc. ACM Program. Lang.*, 4(POPL):45:1–45:32, 2020.
122. Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 637–650. ACM, 2015.
123. Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. Strong logic for weak memory: Reasoning about release-acquire consistency in iris. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, volume 74 of *LIPIcs*, pages 17:1–17:29. Schloss Dagstuhl—Leibniz-Zentrum für Informatik, 2017.
124. Fairouz Kamareddine. Reviewing the classical and the de bruijn notation for [lambda]-calculus and pure type systems. *J. Log. Comput.*, 11(3):363–394, 2001.
125. Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. A promising semantics for relaxed-memory concurrency. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 175–189. ACM, 2017.
126. Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and Viktor Vafeiadis. A formal C memory model supporting integer-pointer casts. In David Grove and Stephen M. Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 326–335. ACM, 2015.
127. Jeehoon Kang, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. Lightweight verification of separate compilation. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20—22, 2016*, pages 178–190. ACM, 2016.
128. Jeehoon Kang, Yoonseung Kim, Youngju Song, Juneyoung Lee, Sanghoon Park, Mark Dongyeon Shin, Yonghyun Kim, Sungkeun Cho, Joonwon Choi, Chung-Kil Hur, and Kwangkeun Yi. Crellvm: verified credible compilation for LLVM. In Jeffrey S. Foster and Dan Grossman, editors, *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 631–645. ACM, 2018.
129. Steven Keuchel, Stephanie Weirich, and Tom Schrijvers. Needle & knot: Binder boilerplate tied up. In Peter Thiemann, editor, *Programming Languages and Systems—25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9632 of *Lecture Notes in Computer Science*, pages 419–445. Springer, 2016.
130. Gerwin Klein and Tobias Nipkow. A machine-checked model for a java-like language, virtual machine, and compiler. *ACM Trans. Program. Lang. Syst.*, 28(4):619–695, 2006.
131. Robbert Krebbers. *The C Standard Formalized in Coq*. PhD thesis, Radboud University Nijmegen, 2015.
132. Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. Mosel: a general, extensible modal framework for interactive proofs in separation logic. *Proc. ACM Program. Lang.*, 2(ICFP):77:1–77:30, 2018.

133. Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 205–217. ACM, 2017.
134. Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars Birkedal. Aneris: A mechanised logic for modular reasoning about distributed systems. In Peter Müller, editor, *Programming Languages and Systems—29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*, volume 12075 of *Lecture Notes in Computer Science*, pages 336–365. Springer, 2020.
135. Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation of ML. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 179–192. ACM, 2014.
136. Daniel Kästner, Ulrich Wünsche, Jörg Barrho, Marc Schlickling, Bernhard Schommer, Michael Schmidt, Christian Ferdinand, Xavier Leroy, and Sandrine Blazy. CompCert: Practical experience on integrating and qualifying a formally verified optimizing compiler. In *ERTS 2018: Embedded Real Time Software and Systems*. SEE, 2018.
137. Peter Lammich. Refinement to imperative HOL. *J. Autom. Reason.*, 62(4):481–503, 2019.
138. Peter Lammich and Thomas Tuerk. Applying data refinement for monadic programs to hopcroft's algorithm. In Lennart Beringer and Amy P. Felty, editors, *Interactive Theorem Proving—Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. Proceedings*, volume 7406 of *Lecture Notes in Computer Science*, pages 166–182. Springer, 2012.
139. Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of standard ML. In Martin Hofmann and Matthias Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 173–184. ACM, 2007.
140. Juneyoung Lee, Chung-Kil Hur, Ralf Jung, Zhengyang Liu, John Regehr, and Nuno P. Lopes. Reconciling high-level optimizations and low-level code in LLVM. *Proc. ACM Program. Lang.*, 2(OOPSLA):125:1–125:28, 2018.
141. Sung-Hwan Lee, Minki Cho, Anton Podkopaev, Soham Chakraborty, Chung-Kil Hur, Ori Lahav, and Viktor Vafeiadis. Promising 2.0: global optimizations in relaxed memory concurrency. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 362–376. ACM, 2020.
142. K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning—16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.
143. Rodolphe Lepigre, Michael Sammler, Kayvan Memarian, Robbert Krebbers, Derek Dreyer, and Peter Sewell. VIP: verifying real-world C idioms with integer-pointer casts. *Proc. ACM Program. Lang.*, 6(POPL):1–32, 2022.
144. Xavier Leroy. Java bytecode verification: Algorithms and formalizations. *J. Autom. Reason.*, 30(3-4):235–269, 2003.
145. Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, pages 42–54. ACM, 2006.
146. Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.

147. Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reasoning*, 43(4):363–446, 2009.
148. Xavier Leroy and Sandrine Blazy. Formal verification of a c-like memory model and its uses for verifying program transformations. *J. Autom. Reason.*, 41(1):1–31, 2008.
149. Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Inf. Comput.*, 207(2):284–304, 2009.
150. Pierre Letouzey. A new extraction for Coq. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002, Selected Papers*, volume 2646 of *Lecture Notes in Computer Science*, pages 200–219. Springer, 2002.
151. Andreas Lochbihler. Verifying a compiler for java threads. In Andrew D. Gordon, editor, *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6012 of *Lecture Notes in Computer Science*, pages 427–447. Springer, 2010.
152. David MacQueen, Robert Harper, and John H. Reppy. The history of Standard ML. *Proc. ACM Program. Lang.*, 4(HOPL):86:1–86:100, 2020.
153. Savi Maharaj and Elsa L. Gunter. Studying the ML module system in HOL. In Thomas F. Melham and Juanito Camilleri, editors, *Higher Order Logic Theorem Proving and Its Applications, 7th International Workshop, Valletta, Malta, September 19-22, 1994, Proceedings*, volume 859 of *Lecture Notes in Computer Science*, pages 346–361. Springer, 1994.
154. J. Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Toward a verified relational database management system. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 237–248. ACM, 2010.
155. Nicolas Marti, Reynald Affeldt, and Akinori Yonezawa. Formal verification of the heap manager of an operating system using separation logic. In Zhiming Liu and Jifeng He, editors, *Formal Methods and Software Engineering, 8th International Conference on Formal Engineering Methods, ICFEM 2006, Macao, China, November 1-3, 2006, Proceedings*, volume 4260 of *Lecture Notes in Computer Science*, pages 400–419. Springer, 2006.
156. John McCarthy and James Painter. Correctness of a compiler for arithmetical expressions. In *Proc. of Symposia in Applied Mathematics*, volume 19 of *Mathematical Aspects of Computer Science*, pages 33–41. American Mathematical Society, 1967.
157. Andrew McCreight. Practical tactics for separation logic. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 343–358. Springer, 2009.
158. Andrew McCreight, Tim Chevalier, and Andrew P. Tolmach. A certified framework for compiling and executing garbage-collected languages. In Paul Hudak and Stephanie Weirich, editors, *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 273–284. ACM, 2010.
159. James McKinna and Robert Pollack. Pure type systems formalized. In *Typed Lambda Calculi and Applications, International Conference on Typed Lambda Calculi and Applications, TLCA '93, Utrecht, The Netherlands, March 16-18, 1993, Proceedings*, volume 664 of *Lecture Notes in Computer Science*, pages 289–305. Springer, 1993.
160. Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. Exploring C semantics and pointer provenance. *Proc. ACM Program. Lang.*, 3(POPL):67:1–67:32, 2019.
161. Glen Mével, Jacques-Henri Jourdan, and François Pottier. Time credits and time receipts in iris. In Luís Caires, editor, *Programming Languages and Systems—28th European*

Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, volume 11423 of *Lecture Notes in Computer Science*, pages 3–29. Springer, 2019.
162. Glen Mével, Jacques-Henri Jourdan, and François Pottier. Cosmo: a concurrent separation logic for multicore ocaml. *Proc. ACM Program. Lang.*, 4(ICFP):96:1–96:29, 2020.
163. Dale Miller. Mechanized metatheory revisited. *J. Autom. Reasoning*, 63(3):625–665, 2019.
164. Dale Miller and Catuscia Palamidessi. Foundational aspects of syntax. *ACM Comput. Surv.*, 31(3es):11, 1999.
165. Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.
166. Robin Milner, Mads Tofte, and Robert Harper. *Definition of standard ML*. MIT Press, 1990.
167. Robin Milner and Richard Weyrauch. Proving compiler correctness in a mechanized logic. In Bernard Meltzer and Donald Michie, editors, *Proc. 7th Annual Machine Intelligence Workshop*, volume 7 of *Machine Intelligence*, pages 51–72. Edinburgh University Press, 1972.
168. Alexandre Moine, Arthur Charguéraud, and François Pottier. A high-level separation logic for heap space under garbage collection. *Proc. ACM Program. Lang.*, 7(POPL):718–747, 2023.
169. J. Strother Moore. A mechanically verified language implementation. *J. Autom. Reason.*, 5(4):461–492, 1989.
170. J. Gregory Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, 1999.
171. Ike Mulder, Robbert Krebbers, and Herman Geuvers. Diaframe: automated verification of fine-grained concurrent programs in iris. In Ranjit Jhala and Isil Dillig, editors, *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, pages 809–824. ACM, 2022.
172. Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation—17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*, volume 9583 of *Lecture Notes in Computer Science*, pages 41–62. Springer, 2016.
173. Magnus O. Myreen and Scott Owens. Proof-producing synthesis of ML from higher-order logic. In Peter Thiemann and Robby Bruce Findler, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, pages 115–126. ACM, 2012.
174. Wolfgang Naraschewski and Tobias Nipkow. Type inference verified: Algorithm W in Isabelle/HOL. *J. Autom. Reasoning*, 23(3-4):299–318, 1999.
175. George C. Necula. Translation validation for an optimizing compiler. In Monica S. Lam, editor, *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000*, pages 83–94. ACM, 2000.
176. Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. Pilsner: a compositionally verified compiler for a higher-order imperative language. In Kathleen Fisher and John H. Reppy, editors, *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, pages 166–178. ACM, 2015.
177. Pierre Neron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A theory of name resolution. In Jan Vitek, editor, *Programming Languages and Systems—24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 205–231. Springer, 2015.

178. Leonor Prensa Nieto. The rely-guarantee method in isabelle/hol. In Pierpaolo Degano, editor, *Programming Languages and Systems, 12th European Symposium on Programming, ESOP 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, volume 2618 of *Lecture Notes in Computer Science*, pages 348–362. Springer, 2003.
179. Tobias Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. In Vijay Chandru and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science, 16th Conference, Hyderabad, India, December 18-20, 1996, Proceedings*, volume 1180 of *Lecture Notes in Computer Science*, pages 180–192. Springer, 1996.
180. Tobias Nipkow. More Church–Rosser proofs. *J. Autom. Reason.*, 26(1):51–66, 2001.
181. Tobias Nipkow. Hoare logics for recursive procedures and unbounded nondeterminism. In Julian C. Bradfield, editor, *Computer Science Logic, 16th International Workshop, CSL 2002, 11th Annual Conference of the EACSL, Edinburgh, Scotland, UK, September 22-25, 2002, Proceedings*, volume 2471 of *Lecture Notes in Computer Science*, pages 103–119. Springer, 2002.
182. Tobias Nipkow and Leonor Prensa Nieto. Owicki/gries in isabelle/hol. In Jean-Pierre Finance, editor, *Fundamental Approaches to Software Engineering, Second Internationsl Conference, FASE'99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*, volume 1577 of *Lecture Notes in Computer Science*, pages 188–203. Springer, 1999.
183. Tobias Nipkow and David von Oheimb. Java$_{light}$ is type-safe—definitely. In David B. MacQueen and Luca Cardelli, editors, *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*, pages 161–170. ACM, 1998.
184. Michael Norrish. *C formalised in HOL.* PhD thesis, University of Cambridge, 1998.
185. Michael Norrish. Recursive function definition for types with binders. In Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan, editors, *Theorem Proving in Higher Order Logics, 17th International Conference, TPHOLs 2004, Park City, Utah, USA, September 14-17, 2004, Proceedings*, volume 3223 of *Lecture Notes in Computer Science*, pages 241–256. Springer, 2004.
186. Michael Norrish. A formal semantics for C++. Technical report, NICTA, 2008.
187. Michael Norrish and René Vestergaard. Proof pearl: De Bruijn terms really do work. In Klaus Schneider and Jens Brandt, editors, *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings*, volume 4732 of *Lecture Notes in Computer Science*, pages 207–222. Springer, 2007.
188. Liam O'Connor, Zilin Chen, Christine Rizkallah, Sidney Amani, Japheth Lim, Toby C. Murray, Yutaka Nagashima, Thomas Sewell, and Gerwin Klein. Refinement through restraint: bringing down the cost of verification. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 89–102. ACM, 2016.
189. Peter W. O'Hearn. Resources, concurrency and local reasoning. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR 2004—Concurrency Theory, 15th International Conference, London, UK, August 31—September 3, 2004, Proceedings*, volume 3170 of *Lecture Notes in Computer Science*, pages 49–67. Springer, 2004.
190. Peter W. O'Hearn. Separation logic. *Commun. ACM*, 62(2):86–95, 2019.
191. Peter W. O'Hearn and David J. Pym. The logic of bunched implications. *Bull. Symb. Log.*, 5(2):215–244, 1999.
192. Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In Laurent Fribourg, editor, *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France,*

September 10-13, 2001, Proceedings, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001.
193. Nicolas Oury and Wouter Swierstra. The power of pi. In James Hook and Peter Thiemann, editors, *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 39–50. ACM, 2008.
194. Scott Owens. A sound semantics for ocamllight. In Sophia Drossopoulou, editor, *Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4960 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2008.
195. Lawrence C. Paulson. A mechanised proof of Gödel's incompleteness theorems using Nominal Isabelle. *J. Autom. Reason.*, 55(1):1–37, 2015.
196. Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In Richard L. Wexelblat, editor, *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*, pages 199–208. ACM, 1988.
197. Frank Pfenning and Carsten Schürmann. System description: Twelf—A meta-logical framework for deductive systems. In Harald Ganzinger, editor, *Automated Deduction—CADE-16, 16th International Conference on Automated Deduction, Trento, Italy, July 7-10, 1999, Proceedings*, volume 1632 of *Lecture Notes in Computer Science*, pages 202–206. Springer, 1999.
198. Brigitte Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 371–382. ACM, 2008.
199. Brigitte Pientka and Andreas Abel. Well-founded recursion over contextual objects. In Thorsten Altenkirch, editor, *13th International Conference on Typed Lambda Calculi and Applications, TLCA 2015, July 1-3, 2015, Warsaw, Poland*, volume 38 of *LIPIcs*, pages 273–287. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015.
200. Brigitte Pientka and Jana Dunfield. Beluga: A framework for programming and reasoning with deductive systems (system description). In Jürgen Giesl and Reiner Hähnle, editors, *Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings*, volume 6173 of *Lecture Notes in Computer Science*, pages 15–21. Springer, 2010.
201. Brigitte Pientka, David Thibodeau, Andreas Abel, Francisco Ferreira, and Rébecca Zucchini. A type theory for defining logics and proofs. In *34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019*, pages 1–13. IEEE, 2019.
202. Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Andrew Tolmach, and Brent Yorgey. *Programming Language Foundations*, volume 2 of *Software Foundations*. Electronic textbook, 2022. Version 6.2.
203. Andrew M. Pitts. Nominal logic, a first order theory of names and binding. *Inf. Comput.*, 186(2):165–193, 2003.
204. Andrew M. Pitts. Alpha-structural recursion and induction. *J. ACM*, 53(3):459–506, 2006.
205. Andrew M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science*, volume 57. Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 2013.
206. Andrew M. Pitts, Justus Matthiesen, and Jasper Derikx. A dependent type theory with abstractable names. In Mauricio Ayala-Rincón and Ian Mackie, editors, *Ninth Workshop on Logical and Semantic Frameworks, with Applications, LSFA 2014, Brasília, Brazil, September 8-9, 2014*, volume 312 of *Electronic Notes in Theoretical Computer Science*, pages 19–50. Elsevier, 2014.

207. Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In Bernhard Steffen, editor, *Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28—April 4, 1998, Proceedings*, volume 1384 of *Lecture Notes in Computer Science*, pages 151–166. Springer, 1998.
208. Robert Pollack. *The theory of LEGO*. PhD thesis, University of Edinburgh, UK, 1995.
209. Andrei Popescu. Nominal recursors as epi-recursors. *Proc. ACM Program. Lang.*, 8(POPL):425–456, 2024.
210. Nicolas Pouillard and François Pottier. A fresh look at programming with names and binders. In Paul Hudak and Stephanie Weirich, editors, *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 217–228. ACM, 2010.
211. Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser. Intrinsically-typed definitional interpreters for imperative languages. *Proc. ACM Program. Lang.*, 2(POPL):16:1–16:34, 2018.
212. Viorel Preoteasa. Mechanical verification of recursive procedures manipulating pointers using separation logic. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings*, volume 4085 of *Lecture Notes in Computer Science*, pages 508–523. Springer, 2006.
213. P. Rajagopalan and Chi Ping Tsang. A generic algebra for data collections based on constructive logic. In Vangalur S. Alagar and Maurice Nivat, editors, *Algebraic Methodology and Software Technology, 4th International Conference, AMAST '95, Montreal, Canada, July 3-7, 1995, Proceedings*, volume 936 of *Lecture Notes in Computer Science*, pages 546–560. Springer, 1995.
214. Tahina Ramananandro. *Mechanized Formal Semantics and Verified Compilation for C++ Objects*. PhD thesis, École normale supérieure, 2012.
215. Tahina Ramananandro, Zhong Shao, Shu-Chun Weng, Jérémie Koenig, and Yuchen Fu. A compositional semantics for verified separate compilation and linking. In Xavier Leroy and Alwen Tiu, editors, *Proceedings of the 2015 Conference on Certified Programs and Proofs, CPP 2015, Mumbai, India, January 15-17, 2015*, pages 3–14. ACM, 2015.
216. Marianna Rapoport, Ifaz Kabir, Paul He, and Ondrej Lhoták. A simple soundness proof for dependent object types. *Proc. ACM Program. Lang.*, 1(OOPSLA):46:1–46:27, 2017.
217. Marianna Rapoport and Ondrej Lhoták. A path to DOT: formalizing fully path-dependent types. *Proc. ACM Program. Lang.*, 3(OOPSLA):145:1–145:29, 2019.
218. John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74. IEEE Computer Society, 2002.
219. Valentin Robert and Xavier Leroy. A formally-verified alias analysis. In Chris Hawblitzel and Dale Miller, editors, *Certified Programs and Proofs—Second International Conference, CPP 2012, Kyoto, Japan, December 13-15, 2012. Proceedings*, volume 7679 of *Lecture Notes in Computer Science*, pages 11–26. Springer, 2012.
220. Andreas Rossberg, Claudio V. Russo, and Derek Dreyer. F-ing modules. *J. Funct. Program.*, 24(5):529–607, 2014.
221. Grigore Rosu. $\mathbb{K}$: A semantic framework for programming languages and formal analysis tools. In Alexander Pretschner, Doron Peled, and Thomas Hutzelmann, editors, *Dependable Software Systems Engineering*, volume 50 of *NATO Science for Peace and Security Series—D: Information and Communication Security*, pages 186–206. IOS Press, 2017.
222. Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. Refinedc: automating the foundational verification of C code with refined ownership types. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 158–174. ACM, 2021.

223. Ulrich Schöpp and Ian Stark. A dependent type theory with names and binding. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *Computer Science Logic, 18th International Workshop, CSL 2004, 13th Annual Conference of the EACSL, Karpacz, Poland, September 20-24, 2004, Proceedings*, volume 3210 of *Lecture Notes in Computer Science*, pages 235–249. Springer, 2004.
224. Carsten Schürmann. *Automating the Meta Theory of Deductive Systems.* PhD thesis, Department of Computer Science, Carnegie Mellon University, 2000. CMU-CS-00-146.
225. Carsten Schürmann, Joëlle Despeyroux, and Frank Pfenning. Primitive recursion for higher-order abstract syntax. *Theor. Comput. Sci.*, 266(1-2):1–57, 2001.
226. Peter Schwabe, Benoît Viguier, Timmy Weerwag, and Freek Wiedijk. A Coq proof of the correctness of X25519 in tweetnacl. In *34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021*, pages 1–16. IEEE, 2021.
227. Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Mechanized verification of fine-grained concurrent programs. In David Grove and Stephen M. Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 77–87. ACM, 2015.
228. Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Specifying and verifying concurrent algorithms with histories and subjectivity. In Jan Vitek, editor, *Programming Languages and Systems—24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 333–358. Springer, 2015.
229. Jaroslav Sevcík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. Relaxed-memory concurrency and verified compilation. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 43–54. ACM, 2011.
230. Jaroslav Sevcík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. Compcerttso: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3):22:1–22:50, 2013.
231. Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strnisa. Ott: Effective tool support for the working semanticist. *J. Funct. Program.*, 20(1):71–122, 2010.
232. Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. Translation validation for a verified OS kernel. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 471–482. ACM, 2013.
233. Natarajan Shankar. A mechanical proof of the Church–Rosser theorem. *J. ACM*, 35(3):475–522, 1988.
234. Mark R. Shinwell, Andrew M. Pitts, and Murdoch Gabbay. Freshml: programming with binders made simple. In Colin Runciman and Olin Shivers, editors, *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, Uppsala, Sweden, August 25-29, 2003*, pages 263–274. ACM, 2003.
235. Stefan Sokolowski. Soundness of hoare's logic: An automated proof using LCF. *ACM Trans. Program. Lang. Syst.*, 9(1):100–120, 1987.
236. Youngju Song, Minki Cho, Dongjoo Kim, Yonghyun Kim, Jeehoon Kang, and Chung-Kil Hur. Compcertm: Compcert with c-assembly linking and lightweight modular verification. *Proc. ACM Program. Lang.*, 4(POPL):23:1–23:31, 2020.
237. Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. Coq Coq correct! verification of type checking and erasure for Coq, in Coq. *Proc. ACM Program. Lang.*, 4(POPL):8:1–8:28, 2020.
238. Kathrin Stark, Steven Schäfer, and Jonas Kaiser. Autosubst 2: reasoning with multi-sorted de bruijn terms and vector substitutions. In Assia Mahboubi and Magnus O. Myreen, editors, *Proceedings of the 8th ACM SIGPLAN International Conference on Certified*

Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019, pages 166–180. ACM, 2019.

239. Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. Compositional compcert. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 275–287. ACM, 2015.
240. Martin Strecker. Formal verification of a java compiler in isabelle. In Andrei Voronkov, editor, *Automated Deduction—CADE-18, 18th International Conference on Automated Deduction, Copenhagen, Denmark, July 27-30, 2002, Proceedings*, volume 2392 of *Lecture Notes in Computer Science*, pages 63–77. Springer, 2002.
241. Kasper Svendsen and Lars Birkedal. Impredicative concurrent abstract predicates. In Zhong Shao, editor, *Programming Languages and Systems—23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8410 of *Lecture Notes in Computer Science*, pages 149–168. Springer, 2014.
242. Nikhil Swamy, Aseem Rastogi, Aymeric Fromherz, Denis Merigoux, Danel Ahman, and Guido Martínez. Steelcore: an extensible concurrent separation logic for effectful dependently typed programs. *Proc. ACM Program. Lang.*, 4(ICFP):121:1–121:30, 2020.
243. Don Syme. Reasoning with the formal definition of Standard ML in HOL. In Jeffrey J. Joyce and Carl-Johan H. Seger, editors, *Higher Order Logic Theorem Proving and its Applications, 6th International Workshop, HUG '93, Vancouver, BC, Canada, August 11-13, 1993, Proceedings*, volume 780 of *Lecture Notes in Computer Science*, pages 43–60. Springer, 1993.
244. Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. A new verified compiler backend for CakeML. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 60–73. ACM, 2016.
245. Yong Kiam Tan, Scott Owens, and Ramana Kumar. A verified type system for CakeML. In Ralf Lämmel, editor, *Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages, IFL '15, Koblenz, Germany, September 14-16, 2015*, pages 7:1–7:12. ACM, 2015.
246. Peter Thiemann. Intrinsically-typed mechanized semantics for session types. In Ekaterina Komendantskaya, editor, *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*, pages 19:1–19:15. ACM, 2019.
247. Alwen Tiu and Alberto Momigliano. Cut elimination for a logic with induction and co-induction. *J. Appl. Log.*, 10(4):330–367, 2012.
248. Jean-Baptiste Tristan and Xavier Leroy. Formal verification of translation validators: a case study on instruction scheduling optimizations. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 17–27. ACM, 2008.
249. Jean-Baptiste Tristan and Xavier Leroy. Verified validation of lazy code motion. In Michael Hind and Amer Diwan, editors, *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 316–326. ACM, 2009.
250. Jean-Baptiste Tristan and Xavier Leroy. A simple, verified validator for software pipelining. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 83–92. ACM, 2010.
251. Thomas Tuerk. A formalisation of Smallfoot in HOL. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009.*

Proceedings, volume 5674 of *Lecture Notes in Computer Science*, pages 469–484. Springer, 2009.
252. Aaron Turon, Derek Dreyer, and Lars Birkedal. Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA—September 25—27, 2013*, pages 377–390. ACM, 2013.
253. Christian Urban. Nominal techniques in Isabelle/HOL. *J. Autom. Reason.*, 40(4):327–356, 2008.
254. Christian Urban, Stefan Berghofer, and Michael Norrish. Barendregt's variable convention in rule inductions. In Frank Pfenning, editor, *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings*, volume 4603 of *Lecture Notes in Computer Science*, pages 35–50. Springer, 2007.
255. Christian Urban and Cezary Kaliszyk. General bindings and alpha-equivalence in Nominal Isabelle. *Log. Methods Comput. Sci.*, 8(2), 2012.
256. Christian Urban and Tobias Nipkow. Nominal verification of algorithm W. In Y. Bertot, G. Huet, J.-J. Lévy, and G. Plotkin, editors, *From Semantics to Computer Science. Essays in Honour of Gilles Kahn*, pages 363–382. Cambridge University Press, 2009.
257. Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 209–220. ACM, 2015.
258. L. S. van Benthem Jutting, James McKinna, and Robert Pollack. Checking algorithms for pure type systems. In *Types for Proofs and Programs, International Workshop TYPES'93, Nijmegen, The Netherlands, May 24-28, 1993, Selected Papers*, volume 806 of *Lecture Notes in Computer Science*, pages 19–61. Springer, 1993.
259. Myra VanInwegen. *The Machine-Assisted Proof of Programming Language Properties*. PhD thesis, Computer And Information Science, University of Pennsylvania, 1996.
260. Peng Wang, Santiago Cuellar, and Adam Chlipala. Compiler verification meets cross-language linking via data abstraction. In Andrew P. Black and Todd D. Millstein, editors, *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 675–690. ACM, 2014.
261. Shengyi Wang, Qinxiang Cao, Anshuman Mohan, and Aquinas Hobor. Certifying graph-manipulating C programs via localizations within data structures. *Proc. ACM Program. Lang.*, 3(OOPSLA):171:1–171:30, 2019.
262. Yuting Wang and Gopalan Nadathur. A higher-order abstract syntax approach to verified transformations on functional programs. In Peter Thiemann, editor, *Programming Languages and Systems—25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9632 of *Lecture Notes in Computer Science*, pages 752–779. Springer, 2016.
263. Yuting Wang, Pierre Wilke, and Zhong Shao. An abstract stack based approach to verified compositional compilation to machine code. *Proc. ACM Program. Lang.*, 3(POPL):62:1–62:30, 2019.
264. Conrad Watt. Mechanising and verifying the webassembly specification. In June Andronick and Amy P. Felty, editors, *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, pages 53–65. ACM, 2018.
265. Tjark Weber. Towards mechanized program verification with separation logic. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *Computer Science Logic, 18th International Workshop, CSL 2004, 13th Annual Conference of the EACSL, Karpacz, Poland, September*

20-24, 2004, Proceedings, volume 3210 of *Lecture Notes in Computer Science*, pages 250–264. Springer, 2004.

266. Stephanie Weirich, Brent A. Yorgey, and Tim Sheard. Binders unbound. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 333–345. ACM, 2011.
267. Martin Wildmoser and Tobias Nipkow. Certifying machine code safety: Shallow versus deep embedding. In Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan, editors, *Theorem Proving in Higher Order Logics, 17th International Conference, TPHOLs 2004, Park City, Utah, USA, September 14-17, 2004, Proceedings*, volume 3223 of *Lecture Notes in Computer Science*, pages 305–320. Springer, 2004.
268. Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 283–294. ACM, 2011.
269. Katherine Q. Ye, Matthew Green, Naphat Sanguansin, Lennart Beringer, Adam Petcher, and Andrew W. Appel. Verified correctness and security of mbedtls HMAC-DRBG. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30—November 03, 2017*, pages 2007–2020. ACM, 2017.
270. Yannick Zakowski, Calvin Beck, Irene Yoon, Ilia Zaichuk, Vadim Zaliva, and Steve Zdancewic. Modular, compositional, and executable formal semantics for LLVM IR. *Proc. ACM Program. Lang.*, 5(ICFP):1–30, 2021.
271. Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In John Field and Michael Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 427–440. ACM, 2012.
272. Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Formal verification of ssa-based optimizations for LLVM. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 175–186. ACM, 2013.