The Curiously Empty Intersection of Proof Engineering and Computational Sciences (Extended Version)

Sampsa Kiiskinen

Abstract Proof engineering tools and techniques have not yet been applied to computational sciences. We try to explain why and investigate their potential to advance the field. More conceretely, we formalize elementary group theory in an interactive theorem prover and discuss how the same technique could be applied to formalize general computational methods, such as discrete exterior calculus. We note that such formalizations could reveal interesting insights about the mathematical structure of the methods and help us implement them with stronger correctness guarantees. We also postulate that working in this way could dramatically change the way we study and communicate computational sciences.

Key words: proof engineering, type theory, software engineering, formal verification, interactive theorem provers, abstract algebra, functional programming, Coq, OCaml

1 Introduction

Proof engineering is a subfield of software engineering, where the objective is to develop theories, techniques and tools for writing proofs of program correctness [60]. Even though proof engineering has many parallels with traditional software engineering, there are several concepts that do not translate directly, giving rise to unique challenges and approaches. Also, despite its practical roots as an engineering discipline, proof engineering is surprisingly closely associated with the foundations and philosophy of mathematics.

If you look for domains, where recent advances in proof engineering have yielded practical applications, you will find various subfields of computer science and mathematics. On the engineering side of things, the domains include model checkers [72,

Sampsa Kiiskinen
University of Jyväskylä, Faculty of Information Technology, Finland, e-mail: `tuplanolla@iki.fi`

13], interactive theorem provers [66], instruction set architectures [2], programming languages [80], network stacks [12], file systems [17], concurrent and distributed systems [63, 83], operating system kernels [45], security policies [26] and certified compilers [48]. On the more mathematical side, the domains span category theory [41], type theory [21], machine learning [61], homotopy theory [6], group theory [35], foundations of mathematics [75], geometry [37], graph theory [34], mathematical logic [56] and abstract algebra [32]. The list goes on, but computational sciences are nowhere to be found on it. Why?

While we cannot answer the question outright, we can still pose the following dichotomy: either proof engineering is unfit for computational sciences in some intricate way, which is interesting, or there is an unexploited opportunity to advance the field, which is also interesting. Let us investigate these options further!

2 Unfit for Computational Sciences?

There are a few heuristic arguments that spring into mind when you first try to explain why proof engineering might be unfit for computational sciences. While these arguments are way too simplistic to offer a satisfying explanation, they are still worth discussing, because they help us address common misconceptions surrounding the subject.

Importance Argument In computational sciences, program correctness does not matter enough to justify formal specification and proof.

Considerable parts of modern infrastructure rely on software packages developed by computational scientists. Every engineering discipline that involves modeling real-world phenomena needs tools for solving linear systems of equations, finding eigenvalues, evaluating integrals or performing Fourier transforms. This is why foundational numerical libraries, such as BLAS, LAPACK, QUADPACK and FFTW, in addition to higher-level software packages based on them, such as NumPy [78] and MATLAB [55], are almost omnipresent in the industry.

To suggest that the correctness of these software packages or programs using them does not matter is not only patently false, but also paints quite a depressing picture of the field. If practitioners care so little about the correctness of their work that they consider stronger guarantees to be unnecessary, there are way more serious issues to be addressed than the missed opportunities afforded by proof engineering. We can therefore dismiss this argument as needlessly pessimistic.

Relevance Argument In computational sciences, correctness is better established by some other method than formal specification and proof.

Testing and debugging are popular and effective ways to improve the correctness of programs, which is why they are prevalent in the industry. However, as effective as testing may be, it can only ever show the presence of bugs and never their absence

[27]. Formal proofs are the only way to demonstrate the absence of bugs[1], given that the proofs actually cover the whole specification.

Even when total absence of bugs is not important, there are still cases, where formal proofs are worth considering. As the generality of a program increases, so does the dimensionality of the space its tests need to cover. If the program keeps growing, testing will eventually become infeasible, because high-dimensional spaces simply have too much room for bugs to hide in. Throwing more computational resources at the problem can offer some relief, but it is not a sustainable solution, because there is always a limit to how much we can compute. So, while this argument may apply to some cases, it is not universally true.

Insight Argument In computational sciences, no new insights can be gained from formal specification or proof.

Even though writing formal proofs involves a decent amount of tedious busywork, it is not a clerical activity completely devoid of creative insight. Organizing existing knowledge into a form that is as beautiful and understandable as it deserves to be is both challenging and valuable. If this was not the case, category theory and reverse mathematics[2] would not exist and detailed formalizations of familiar mathematical concepts would not frequently yield unexpected insights [70].

It seems dubious that computational sciences could be so well-understood that its formalizations would be fruitless[3]. You might personally feel this way if you are only interested in the immediate results and applications of a particular theory, but otherwise this argument is needlessly narrow-minded.

Performance Argument In computational sciences, performance matters most and is at odds with formal specification and proof.

One traditional technique for improving the correctness of programs is to litter them with assertions. Assertions are logical statements of invariants that should always hold. They are checked during the execution of a program and, if they are actually found to not hold, the violation is reported and the program is terminated immediately[4].

Assertions are commonly employed in array libraries, where accessing an element at some index is preceded by a check ensuring that the index is in bounds. They are also often used to uphold preconditions of numerical routines, such as checking the condition number of a matrix before performing linear regression or checking

[1] The sole exception is a proof by exhaustion, which is a way to test things conclusively. Aside from model checking, it is rarely used in practice, because interesting things tend to be infinite or at least very large.

[2] Reverse mathematics refers to the investigation of what assumptions are needed to prove known theorems. In more poetic terms, it is the study of the necessary and the sufficient.

[3] Quantum computing seems like a particularly good candidate for formalization due to its counterintuitive nature and close ties with computer science [51].

[4] There are also static assertions that are checked during the compilation of a program, but they are usually much less flexible.

the Courant–Friedrichs–Lewy condition of a discretization scheme before solving a partial differential equation.

Another traditional technique for improving the correctness of programs is to write them in a high-level language with a dedicated runtime system. The motivation for having a runtime system is to automate some of the most tedious and error-prone programming tasks, so that you can focus your attention on more important matters.

Functional programming languages typically employ runtime systems with garbage collectors, whose purpose is to take care of allocating and releasing memory. Meanwhile, runtime systems of scripting languages almost always incorporate dynamic type systems, whose operating principle is to tag every object with a reference to its type.

Since assertions, garbage collectors and dynamic type systems all incur performance penalties, it is easy to assume that any method for improving the correctness of programs has similar drawbacks, but this is not always the case. There exist modern programming languages with so-called zero-cost abstractions that have no negative impact on performance. Typestate analysis [9] and proof erasure [66] are examples of such abstractions. Not only do they not incur performance penalties, but sometimes they even help the compiler find new performance optimizations. So, while this argument may have been true in the past, it is gradually becoming less and less convincing.

Resource Argument In computational sciences, time and money should be invested on something other than formal specification and proof.

Even though proof engineering tools are constantly becoming more accessible, writing formal specifications and proofs is still quite difficult and laborious. This is especially true in domains, where prerequisite definitions and lemmas have not yet been laid out in full. To give a concrete example, it took several months to establish enough prerequisites in topology, algebra and geometry before perfectoid spaces could be defined [15].

Luckily, formalization is not a package deal. There is nothing wrong with formalizing only the most salient parts of a theory and leaving the rest for later. If you count humans as a part of the system, this thought also relates to performance argument. The less time you and your colleagues spend programming, testing, debugging and reviewing code, the better the overall performance of the system will be. From the collective standpoint, this argument could go either way.

Feasibility Argument In computational sciences, it would be too difficult or limiting to incorporate formal specification and proof.

It is a reasonable fear that formalization could increase the barrier of entry and make adopting new ideas more difficult. However, it has been reported that proof engineering tools make for excellent teaching aids [59] and that the inflexibility towards changing definitions can be mitigated through the principle of representation independence [1].

On the flip side, you could argue that it would be too difficult or limiting to not incorporate formal specification and proof. Whether or not you subscribe to the intuitionistic philosophy of defining mathematics as a mental process, it is undisputable that humans can only hold so much information in their heads at once. As the generality of a program increases, so does the level of abstraction needed to keep it manageable. Not everything can or even should be understood as arrays of floating-point numbers.

In terms of rigor, computational sciences sit somewhere between mathematics and software engineering, so it seems unlikely that we could not overcome the same challenges as everyone else. Thus, this argument seems like a pointless worry.

3 Closer Investigation

In terms of tools, proof engineering deals with the development and use of interactive theorem provers (ITPS), automated theorem provers (ATPS), program verifiers and constraint solvers (CSS). While all of these tools could be useful in various corners of computational sciences, we want to focus our attention on ITPS, because they offer the highest level of generality. This generality is not only apparent in the problems ITPS have been used to solve, but also in the way ITPS can leverage other ATPS and CSS as their subsystems [25].

3.1 Architecture of Interactive Theorem Provers

In the early days, programming languages, such as FORTRAN[5] (from 1957) and C (from 1972), had very simple type systems. Their primary purpose was to tell the compiler what storage size, alignment and supported operations any variable was supposed to have. The compiler tracked this information and ensured it remained consistent throughout the program [16, 42].

Programming languages have evolved a lot since those days. Nowadays, there exist languages with type systems that are strong enough to express arbitrarily complicated properties. If arbitrariness does not immediately spark your imagination, you may consider such properties as “this differential equation has a unique solution”, “this numerical method never diverges” or “this program always halts”. Indeed, dependently-typed total languages can express the entirety of constructive mathematics due to a principle known as the Curry–Howard correspondence [76]. The only caveat is that the compiler cannot necessarily find a proof of a given proposition or tell if the proposition is meaningful; it can only decide whether a given proof is correct or not.

Some notable ITPS include Isabelle (from 1986), Coq (from 1989), Agda (from 2007) and Lean (from 2013). They are classified as ITPS, because they satisfy the de Bruijn criterion, which requires being able to produce elaborated proof objects that

[5] You can infer the era from the name, as lowercase letters were not invented until FORTRAN 77 was superseded by Fortran 90.

a small proof-checking kernel can verify [60]. The architecture of such ITPS roughly follows the diagram in figure 1.

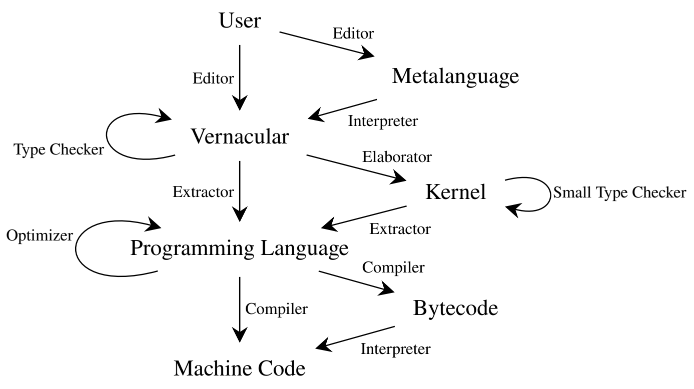

Fig. 1 Architecture of interactive theorem provers, where nodes represent the components of the system and arrows represent the data flow mechanisms between them.

The vernacular is the surface language of the system and therefore appears in all ITPS. Users mostly write their programs in the vernacular, which is why it has to be quite a rich language. The metalanguage is an additional support language that only appears in some ITPS. The purpose of the metalanguage is to help write more complicated vernacular programs and it often comes with more unprincipled features, such as nontermination, unhygienic macros and reflection. The kernel, as the name suggests, is at the core of all ITPS. Vernacular programs are compiled into kernel programs through a process called elaboration and then checked for correctness. The resulting kernel programs tend to be large, uninformative and may only ever exist ephemerally. Their sole purpose is to be type checked and possibly compiled into another language. The remaining components are common to most ordinary programming languages.

We shall use Coq as an example of an ITP, because it possibly follows this architecture most closely. In Coq, the vernacular is a dependently-typed total purely functional language called Gallina, which looks a lot like ML. Any program written in Gallina is guaranteed to terminate, because its only control flow abstractions are structurally recursive functions[6]. In Coq, the metalanguage is an untyped imperative language called Ltac[7], which bears some resemblence to Bourne shell scripts. Working with Ltac is similar to using a reversible debugger, in the sense that you change the state of your program by repeatedly doing and undoing commands. In Coq, the

[6] Even well-founded recursion is not primitive, as it is translated into structural recursion through a clever use of the accessibility predicate [14].

[7] The name is sometimes typeset as L_{tac} and stands for the obvious "language for tactics". As you can see, working in this field takes a lot of imagination.

kernel is an implementation of the calculus of inductive constructions with some small modifications [58, 23]. It is intended to be as small and simple as possible, so that the risk of having bugs in the kernel is minimized. This is very important, because all of the correctness guarantees afforded by the system hinge on the kernel being bug-free.

3.2 Theoretical Background

The kernels of all prominent ITPs to date are based on some flavors of type theory. In the more conservative camp, Coq and Lean both implement variations of the calculus of constructions, which is also occasionally called Coquand–Huet type theory [22]. In the more experimental camp, Agda implements both intuitionistic type theory, which is frequently referred to as just Martin-Löf type theory [53], and cubical type theory, which is also known as Cohen–Coquand–Huber–Mörtberg type theory [19]. Each of these type theories has enough expressive power to serve as a constructive foundation of mathematics, even though they have notable differences in the features they provide. For example, cubical type theory comes with higher-inductive types [21, 73], while the predicative calculus of inductive constructions supports a universe of proof-irrelevant propositions [33].

If you wonder why none of the ITPs implement Zermelo–Fraenkel set theory, which is widely considered to be the canonical foundation of mathematics, the reason is mostly coincidental. Nobody has found success with it and the consensus in the community seems to be that set theory only works in principle, not in practice. This attitude is also apparent in statements written by experts.

> You may consider the development of V to be outside [the] scope of mathematics and logic, but then do not complain when computer scientist[s] fashion it after their technology.
>
> I have never seen any serious proposals for a vernacular[8] based on set theory. Or[,] to put it another way, as soon as we start expanding and transforming set theory to fit the requirements for V, we end up with a theoretical framework that looks a lot like type theory.
>
> — Andrej Bauer [5]

While understanding type theory in depth is way beyond the scope of this text, it is still helpful to be familiar with its basic parlance. Type theories are conventionally presented using inference rules[9]. Each rule is written $\square/\square$ with a hole for the premises and a hole for the conclusion. Rules involve two kinds of judgements that can be placed in a context. Typing judgements are written $\square : \square$ with a hole for the term and a hole for its type. Equality judgements are written $\square \equiv \square$ with two holes for the terms that are supposed to be equal. Contexts are written $\square, \square, \ldots, \square$ with holes for judgements or other contexts. Hypothetical judgements are written $\square \vdash \square$ with a hole for the context and a hole for the judgement.

[8] The variable V refers to the formal language used for the vernacular.

[9] This presentation uses a proof calculus called natural deduction. It is popular, but far from the only option.

Even though the typing judgement $x : A$ resembles the set-theoretical membership proposition $x \in A$ and the equality judgement $x \equiv y$ resembles the set-theoretical equality proposition $x = y$, they are not semantically equivalent. The first big difference is that types are not sets. While some types, such as that of the integers $\mathbb{Z}$ and that of infinite bit strings $\mathbb{N} \to \mathbf{2}$, can be shown to form sets, many others, such as that of types $\mathcal{U}$ and that of the circle $\mathbb{S}^1$, carry more structure. The second big difference is that judgements are not propositions. While propositions are internal to the theory, judgements are not, so it is not possible to do such things as form the negation of a judgement.

Inference rules are quite an abstract concept, so let us illustrate them through a concrete example. We shall use natural numbers as an example, because they are both familiar and ubiquitous.

Natural numbers are a type with the formation rule

$$\frac{}{\Gamma \vdash \mathbb{N} : \mathcal{U}},$$

the introduction rules

$$\frac{}{\Gamma \vdash O : \mathbb{N}} \qquad \frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash S(n) : \mathbb{N}},$$

the elimination rule

$$\frac{\begin{array}{c}\Gamma, m : \mathbb{N} \vdash P(m) : \mathcal{U} \qquad \Gamma \vdash x : P(O) \\ \Gamma, m : \mathbb{N}, y : P(m) \vdash f(m, y) : P(S(m)) \qquad \Gamma \vdash n : \mathbb{N}\end{array}}{\Gamma \vdash \mathsf{ind}_P(x, f, n) : P(n)}$$

and the computation rules

$$\frac{\begin{array}{c}\Gamma, m : \mathbb{N} \vdash P(m) : \mathcal{U} \qquad \Gamma \vdash x : P(O) \\ \Gamma, m : \mathbb{N}, y : P(m) \vdash f(m, y) : P(S(m))\end{array}}{\Gamma \vdash \mathsf{ind}_P(x, f, O) \equiv x}$$

$$\frac{\begin{array}{c}\Gamma, m : \mathbb{N} \vdash P(m) : \mathcal{U} \qquad \Gamma \vdash x : P(O) \\ \Gamma, m : \mathbb{N}, y : P(m) \vdash f(m, y) : P(S(m)) \qquad \Gamma \vdash n : \mathbb{N}\end{array}}{\Gamma \vdash \mathsf{ind}_P(x, f, S(n)) \equiv f(n, \mathsf{ind}_P(x, f, n))}.$$

The introduction rules represent zero and the successor of another natural number, so any particular natural number can be constructed by applying them repeatedly. We have

$$\frac{}{\Gamma \vdash 2 : \mathbb{N}} \qquad \frac{}{\Gamma \vdash 4 : \mathbb{N}}$$

$$\frac{}{\Gamma \vdash 2 \equiv S(S(O))} \qquad \frac{}{\Gamma \vdash 4 \equiv S(S(S(S(O))))}$$

and so on.

The elimination rule represents mathematical induction, so the addition of natural numbers can be defined by applying it to either parameter. If we pick the first one, as is convention, we get

$$\frac{}{\Gamma, n : \mathbb{N}, p : \mathbb{N} \vdash n + p : \mathbb{N}}$$

$$\frac{}{\Gamma, n : \mathbb{N}, p : \mathbb{N} \vdash n + p \equiv \mathsf{ind}_{m \mapsto \mathbb{N}}(p, (m, q) \mapsto S(q), n)},$$

which reduces according to the computation rules[10].

Informal presentations are rarely as detailed as we have been so far. It usually suffices to say that natural numbers are an inductive type freely generated by

$$O : \mathbb{N} \qquad S : \mathbb{N} \to \mathbb{N},$$

because everything else can be inferred from these generators [3]. It is also sufficient to define addition by

$$\begin{aligned} \square + \square &: \mathbb{N} \times \mathbb{N} \to \mathbb{N} \\ O + p &\equiv p \\ S(n) + p &\equiv S(n + p), \end{aligned}$$

because pattern matching over structural recursion is equivalent to induction via elimination rules [20, 18]. We will follow a more informal style like this from here on out.

3.3 Role of Elaboration

Now that we know how ITPs are designed, we can discuss the role of elaboration in them. The purpose of elaboration is to convert mathematical statements expressed in the vernacular into equivalent statements that can be handled by the kernel.

As before, we shall illustrate an abstract concept through a concrete example. In the vein of choosing something that is both familiar and ubiquitous, we shall use elementary group theory as an example.

Consider the following excerpt from a typical textbook on abstract algebra.

If $h : G \to H$ is a group homomorphism, then $h(a + 2 \times b) = h(a) + 2 \times h(b)$.

If we have done our homework properly and wish to be as terse as possible[11], we can feed this excerpt into Coq almost exactly as it is stated here. This is possible due to the following mechanisms supported by its vernacular.

[10] It is a good exercise for the reader to follow the given rules to deduce $2 + 2 \equiv 4$.

[11] Being as terse as possible is generally not a good guideline for writing maintainable code, but that is beside the point.

Existential Variables The constituents of the groups are not introduced explicitly. We need to assume the existence of the carriers $A : \mathcal{U}$ and $B : \mathcal{U}$, the relations $X : A \times A \to \mathcal{U}$ and $Y : B \times B \to \mathcal{U}$, the identities $x : A$ and $y : B$, the inverses $f : A \to A$ and $g : B \to B$ and the operations $k : A \times A \to A$ and $m : B \times B \to B$ before we can even talk about the homomorphism $h : G \to H$ between the groups $G : \mathsf{IsGrp}_A(X, x, f, k)$ and $H : \mathsf{IsGrp}_B(Y, y, g, m)$.

Implicit Coercions The typing judgement $h : G \to H$ does not make sense, because G and H are groups instead of types. The function in question is actually defined between the carriers, as in $h : A \to B$, with the additional knowledge that it preserves group structure, as evidenced by $M : \mathsf{IsGrpHom}_{A,B}(X, x, f, k, Y, y, g, m, h)$. We must insert projections from G to A and H to B before we can make sense of the typing judgement.

Type Inference The types of the variables a and b are not specified. However, from the presence of $h(a)$ and $h(b)$, we can infer that $a : A$ and $b : A$.

Implicit Generalization The variables a and b are introduced without quantifiers[12]. We should implicitly generalize the equation as if it was stated under the universal quantifier $\prod_{a,b:A} \square$.

Notation Scopes The notation 2 may not refer to any particular number, because the interpretations of notations depend on their scope. We should interpret 2 as the iterated sum $1 + (1 + 0)$, where 0, 1 and $\square + \square$ refer to the constituents of an arbitrary semiring.

Type Classes The notations referring to the constituents of various algebraic structures are ambiguous. We employ operational type classes, so that we can reuse the relation $\square = \square$ for X and Y, the identity 0 for x, y and the zero integer, the inverse $-\square$ for f and g, the operation $\square + \square$ for k, m and integer addition, the identity 1 for the unit integer and the action $\square \times \square$ for an integer repetition of the operation.

Implicit Arguments The operations $\square = \square$, 0, 1, $\square + \square$ and $\square \times \square$ are not fully applied. There are implicit arguments that we can unambiguously infer to be $\square =_B \square$, $0_{\mathbb{Z}}$, $1_{\mathbb{Z}}$, $\square +_{\mathbb{Z}} \square$, $\square +_A \square$, $\square +_B \square$, $\square \times_{\mathbb{Z},A} \square$ and $\square \times_{\mathbb{Z},B} \square$.

Universe Polymorphism The universe levels of any of the sorts $\mathcal{U}$ are not given. We should be able to choose them in such a way that there will be no inconsistencies, such as Girard's paradox.

After elaboration, the excerpt will have taken roughly the following form.

> Let $A : \mathcal{U}_0$, $B : \mathcal{U}_0$, $X : A \times A \to \mathcal{U}_{-1}$, $Y : B \times B \to \mathcal{U}_{-1}$, $x : A$, $y : B$, $f : A \to A$, $g : B \to B$, $k : A \times A \to A$, $m : B \times B \to B$, $G : \mathsf{IsGrp}_A(X, x, f, k)$, $H : \mathsf{IsGrp}_B(Y, y, g, m)$, $h : A \to B$ and $M : \mathsf{IsGrpHom}_{A,B}(X, x, f, k, Y, y, g, m, h)$. We can construct the term $s : \prod_{a,b:A} Y(h(k(a, k(b, k(b, x)))), m(h(a), m(h(b), m(h(b), y))))$ or at least refute its nonexistence[13].

[12] In type theory, universal and existential quantifiers are proof-relevant, which is why they are written as sums and products.

[13] We make a distinction between being able to construct a witness and being able to derive a contradiction from the lack of a witness, because there is no inference rule that would unify these notions in the classical way.

This form is quite verbose, but consists entirely of pieces we can understand and the kernel can check.

3.4 Proof Engineering by Example

To show what proof engineering looks like in practice, we shall formalize enough elementary group theory to state and prove the previously discussed excerpt and extract useful computational code from it. Even though groups are an important concept in mathematics and appear in various corners of computational sciences, formalizing them is not particularly novel [31, 35]. Instead, the point of this adventure is to demonstrate a proof engineering technique that could be applied to other interesting concepts, such as exterior algebras and differential equations.

3.4.1 Type-Theoretical Model

We start by building a type-theoretical model of groups. The model has two levels that are open to extension and enjoy representation independence principles.

The first level is for abstract specifications, describing what it means to be a group. If there are many such specifications, they must all be equivalent. The second level is for concrete representations, exhibiting constructions of particular groups. If any particular group has several different representations, they must all be isomorphic. The model and its levels are sketched in figure 2.

The design of the model is motivated by category theory. The first level corresponds to the category of type classes [77], which is just a subcategory of the category of types. The second level corresponds to the category of groups. The levels are related by the instance resolution relation, which you can imagine to be a functor, even though it is not[14].

On the first level of the model, we have various type classes, including one for abelian groups, one for commutativity and two for groups. If we start from the type class for abelian groups[15] and fully expand all of its constituents, we get

[14] It is *not* a good exercise for the reader to ponder why the instance resolution relation is not a functor.

[15] In classical mathematics, there is the additional requirement that the carrier must be a set, but we do not include it here.

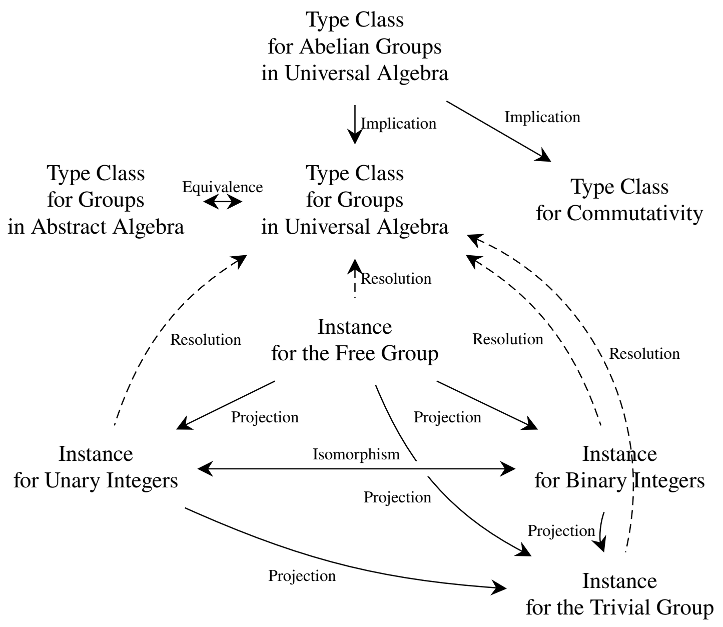

Fig. 2 A type-theoretical model of groups, where nodes represent types and arrows represent functions between them.

$$\mathsf{IsAbGrp} \;:\; \prod_{A:\mathcal{U}} (A \times A \to \mathcal{U}) \times A \times (A \to A) \times (A \times A \to A) \to \mathcal{U}$$

$$\begin{aligned}
\mathsf{IsAbGrp}_A(X, x, f, k) \equiv{}& \overbrace{\Big(\prod_{y:A} X(y, y)\Big)}^{\mathsf{IsRefl}_A(X)} \times \overbrace{\Big(\prod_{y,z:A} X(y, z) \to X(z, y)\Big)}^{\mathsf{IsSym}_A(X)} \times \\
& \Big(\prod_{y,z,w:A} X(y, z) \times X(z, w) \to X(y, w)\Big) \times \\
& \Big(\prod_{y,z,w:A} X(k(y, k(z, w)), k(k(y, z), w))\Big) \times \\
& \Big(\prod_{y,z:A} X(y, z) \to \prod_{w,v:A} X(w, v) \to X(k(y, w), k(z, v))\Big) \times \\
& \Big(\prod_{y:A} X(k(x, y), y)\Big) \times \Big(\prod_{y:A} X(k(y, x), y)\Big) \times \\
& \Big(\prod_{y:A} X(k(f(y), y), x)\Big) \times \Big(\prod_{y:A} X(k(y, f(y)), x)\Big) \times \\
& \underbrace{\Big(\prod_{y,z:A} X(y, z) \to X(f(y), f(z))\Big)}_{\mathsf{IsProper}_{1,A}(X,f)} \times \underbrace{\Big(\prod_{y,z:A} X(k(y, z), k(z, y))\Big)}_{\mathsf{IsComm}_A(X,k)}.
\end{aligned}$$

As you can see, this type class is just a product of the type class for groups and the type class for commutativity, so we can project either one out with

$$\mathsf{fst} : \prod_{A,B:\mathcal{U}} A \times B \to A \qquad \mathsf{snd} : \prod_{A,B:\mathcal{U}} A \times B \to B.$$

We have written the type class for groups as it appears in universal algebra, stating that there exists such a choice function f that any element y has the unique inverse $f(y)$. If instead we wish to write it as it appears in abstract algebra, stating that any element y has the unique inverse y^{-1}, we can transport the type class along[16]

$$\mathsf{choice} : \prod_{A,B:\mathcal{U}} \prod_{X:A\times B\to\mathcal{U}} \Big(\prod_{y:A}\sum_{z:B} X(y,z)\Big) \simeq \Big(\sum_{f:A\to B}\prod_{y:A} X(y,f(y))\Big).$$

This gives us representation independence with respect to specifications.

On the second level of the model, we have representations for various groups, including one for the free group, one for the trivial group and two for the group of integers. One of the integer representations is established over unary integers, which is an inductive type freely generated by

$$\begin{aligned} 0 &: \mathbb{Z} \\ +\square &: \mathbb{P} \to \mathbb{Z} \\ -\square &: \mathbb{P} \to \mathbb{Z} \end{aligned} \qquad \begin{aligned} 1 &: \mathbb{P} \\ 1 + \square &: \mathbb{P} \to \mathbb{P}. \end{aligned}$$

The other integer representation is established over binary integers, which is another inductive type freely generated by

$$\begin{aligned} 0 &: \mathbb{Z}^{\mathrm{b}} \\ +\square &: \mathbb{P}^{\mathrm{b}} \to \mathbb{Z}^{\mathrm{b}} \\ -\square &: \mathbb{P}^{\mathrm{b}} \to \mathbb{Z}^{\mathrm{b}} \end{aligned} \qquad \begin{aligned} 1 &: \mathbb{P}^{\mathrm{b}} \\ 2 \times \square &: \mathbb{P}^{\mathrm{b}} \to \mathbb{P}^{\mathrm{b}} \\ 1 + 2 \times \square &: \mathbb{P}^{\mathrm{b}} \to \mathbb{P}^{\mathrm{b}}. \end{aligned}$$

The representations are isomorphic, as witnessed by encoding and decoding. If we have written a group over $\mathbb{Z}$ and instead wish to have the equivalent group over $\mathbb{Z}^{\mathrm{b}}$, we can transport the structure along

$$\mathsf{code} : \mathbb{Z} \simeq \mathbb{Z}^{\mathrm{b}}.$$

This gives us representation independence with respect to carriers.

The trivial group is defined over the unit type, which is an inductive type freely generated by

$$0 : \mathbf{1}.$$

The trivial group is the terminal object in the category of groups, so any group can be projected into it with a specialization of the constant map

[16] Unlike classical choice, which is an axiom, constructive choice is just a theorem.

$$\mathsf{const}(0) : \prod_{A:\mathcal{U}} A \to \mathbf{1}.$$

The free group is defined over finite sequences with some restrictions on adjacent elements, making it a bit more elaborate to construct. We shall use lists as our finite sequences, because they are the simplest structure that works[17]. Lists are defined as an inductive type family freely generated by

$$\varnothing : \prod_{A:\mathcal{U}} A^* \qquad \square \triangleleft \square : \prod_{A:\mathcal{U}} A \times A^* \to A^*.$$

It is also customary to define the functions

$$\begin{aligned} \mathsf{fold} &: \prod_{A:\mathcal{U}} A \times (A \times A \to A) \to (A^* \to A) & \mathsf{app} &: \prod_{A:\mathcal{U}} A^* \times A^* \to A^* \\ \mathsf{map} &: \prod_{A,B:\mathcal{U}} (A \to B) \to (A^* \to B^*) & \mathsf{rev} &: \prod_{A:\mathcal{U}} A^* \to A^* \\ \mathsf{zip} &: \prod_{A,B:\mathcal{U}} A^* \times B^* \to (A \times B)^* & \mathsf{drop} &: \mathbb{N} \to \prod_{A:\mathcal{U}} A^* \to A^* \end{aligned}$$

to fold a sequence with a monoid, map a function over a sequence, zip two sequences together, append two sequences, reverse a sequence and drop a finite number of elements from a sequence[18].

Now, suppose A is a discrete type. The free group generated by A is the group defined over $F(A)$, which we decree to be

$$\begin{aligned} \varphi_A &: (\mathbf{2} \times A) \times (\mathbf{2} \times A) \to \mathbf{2} \\ \varphi((i,x),(j,y)) &\equiv \neg(i \neq j \wedge x = y) \\ F(A) &: \mathcal{U} \\ F(A) &\equiv \sum_{s:(\mathbf{2}\times A)^*} \underbrace{\mathsf{fold}(1, \square \wedge \square,}_{\mathsf{and}} \mathsf{map}(\varphi, \mathsf{zip}(s, \mathsf{drop}(1)(s)))). \end{aligned}$$

The idea is that $(\mathbf{2} \times A)^*$ is a finite sequence that is well-formed according to the adjacency relation φ. Each $\mathbf{2} \times A$ in the sequence records an element of type A with a flag of type $\mathbf{2}$ indicating whether the element is inverted. We require A to be discrete, so that we can use Hedberg's theorem[19] to prove that A is a set and that $F(A)$ is a trivial subset of $(\mathbf{2} \times A)^*$.

The free group is the initial object in the category of groups, so it can be projected into any other group with the appropriate evaluation map. Suppose $f : A \to \mathbb{Z}$

[17] Finger trees would have asymptotically better performance characteristics, but they would also be more complicated to define and reason about [39].

[18] It is a good exercise for the reader to write the definitions, because they are standard in literature on functional programming [54].

[19] The theorem states that every discrete type is a set or, in other words, that every type with decidable equality has unique identity proofs [47]. The converse is not true, as the type of infinite bit strings $\mathbb{N} \to \mathbf{2}$ forms a set, but does not admit decidable equality.

assigns an integer value to every inhabitant of some discrete type A. The free group over A can be projected into the group over $\mathbb{Z}$ with the evaluation map $e(f)$, which we deem

$$\begin{aligned} \varepsilon(f) &: \mathbf{2} \times A \to \mathbb{Z} \\ \varepsilon(f)(0, x) &\equiv f(x) \\ \varepsilon(f)(1, x) &\equiv -f(x) \end{aligned} \qquad \begin{aligned} e(f) &: F(A) \to \mathbb{Z} \\ e(f)(s) &\equiv \underbrace{\mathsf{fold}(0, \square + \square,}_{\mathsf{sum}} \mathsf{map}(\varepsilon(f), s)). \end{aligned}$$

This gives us representation independence with respect to structures.

3.4.2 Implementation in an Interactive Theorem Prover

Having built a model of groups, we can implement it in an ITP. Our implementation is written in Coq and inspired by the operative–predicative type class design [65, 67]. We mainly diverge from the original design in the way we decouple operational classes from predicative ones, export instances and bind notations.

While it would be nice to paint an honest picture of the implementation by showing every little detail, we can only fit so much on this napkin. Despite our best efforts to be terse, we have to omit some parts, as indicated by the discontinuous line numbers and admitted proofs. The curious reader can find the full implementation in the project repository [44].

As we saw on the first level of the model, groups are just monoids with inverses that are proper with respect to the underlying equivalence relation. We can formalize this notion of groups by combining `IsMon`, `IsInvLR` and `IsProper`.

```
16  Class IsGrp (A : Type) (X : A -> A -> Prop)
17    (x : A) (f : A -> A) (k : A -> A -> A) : Prop := {
18    grp_is_mon :> IsMon X x k;
19    grp_is_inv_l_r :> IsInvLR X x f k;
20    grp_is_proper :> IsProper (X ==> X) f;
21  }.
```

We could further formalize the notion of abelian groups by combining `IsGrp` and `IsComm`.

```
23  Class IsComm (A : Type) (X : A -> A -> Prop) (k : A -> A -> A) : Prop :=
24    comm (x y : A) : X (k x y) (k y x).
25
26  Class IsAbGrp (A : Type) (X : A -> A -> Prop)
27    (x : A) (f : A -> A) (k : A -> A -> A) : Prop := {
28    ab_grp_is_grp :> IsGrp X x f k;
29    ab_grp_is_comm :> IsComm X k;
30  }.
```

These type classes make up the predicative part of the design, because they map types and terms into propositions with no computational content. Other type classes that map types and terms into other types make up the operative part.

Groups have several other properties, each of which can be proven by instantiating the appropriate type class. We set up a context, where the group structure and the notations for its constituents are established through operational classes[20].

```
32  Section Context.
33
34  Context (A : Type)
35    (X : A -> A -> Prop) (x : A) (f : A -> A) (k : A -> A -> A)
36    `{!IsGrp X x f k}.
37
38  #[local] Instance has_eq_rel : HasEqRel A := X.
39  #[local] Instance has_null_op : HasNullOp A := x.
40  #[local] Instance has_un_op : HasUnOp A := f.
41  #[local] Instance has_bin_op : HasBinOp A := k.
```

In this context, we can use the tactic facilities of the metalanguage to show that $-\square$ is injective.

```
49  #[export] Instance is_inj : IsInj X f.
50  Proof.
51    note. intros y z a.
52    rewrite <- (unl_l z). rewrite <- (inv_r y).
53    rewrite a. rewrite <- (assoc y (- z) z).
54    rewrite (inv_l z). rewrite (unl_r y).
55    reflexivity. Qed.
56
57  End Context.
```

We could similarly show that 0 is a fixed point of $-\square$, $-\square$ is an involution, $\square + \square$ is cancellative on both sides and $-\square$ antidistributes over $\square + \square$.

The tactics tell the interpreter what reasoning steps it should take in order to prove the goal. After line 52, the proof is still in an incomplete state, which is presented to the user as follows.

```
1   A : Type
2   X : A -> A -> Prop
3   x : A
4   f : A -> A
5   k : A -> A -> A
6   H : IsGrp _==_ 0 -_ _+_
7   y, z : A
8   a : - y == - z
9   -------------------------------------
10  y == (y + (- y)) + z
```

Much like inference rules, the proof state is written $\square/\square$ with a hole for the hypotheses and a hole for the goal. It may look daunting, but that is just an illusion caused by its verbosity. We do not need to care about the hypotheses before *H*, because they are contextual information we can access through the operational classes. We cannot

[20] The expert reader might wonder why we go through the extra ceremony to declare the operational instances local. We do this, because it makes the parameters of predicative classes independent of operational classes and makes the distinction between projected and derived instances of predicative classes almost opaque to the user.

refer to H itself either, because we introduced it without a name on line 36. We could also infer y and z ourselves, because they are mentioned in the next hypothesis. Thus, the hypothesis a and the goal are the only things that really matter.

Moving on, group homomorphisms are functions between groups that preserve their operations and equivalences. We can formalize this notion of group homomorphisms by combining `IsGrp`, `IsBinPres` and `IsProper`.

```
Class IsGrpHom (A B : Type)
  (X : A -> A -> Prop) (x : A) (f : A -> A) (k : A -> A -> A)
  (Y : B -> B -> Prop) (y : B) (g : B -> B) (m : B -> B -> B)
  (h : A -> B) : Prop := {
  grp_hom_dom_is_grp : IsGrp X x f k;
  grp_hom_codom_is_grp : IsGrp Y y g m;
  grp_hom_hom_is_bin_pres :> IsBinPres Y k m h;
  grp_hom_hom_is_proper :> IsProper (X ==> Y) h;
}.
```

We could once again set up a context and show that the function preserves both 0 and $-\square$.

Before turning our attention to the second level of the model, we define a tactic called `ecrush`, which repeatedly tries to solve a goal using artificial intelligence[21] or to split the remaining goals into smaller subgoals that can be dispatched using simple case analyses or known arithmetic lemmas.

```
Ltac ecrush :=
  repeat (try typeclasses eauto; esplit);
  hnf in *; repeat match goal with
  | |- exists _ : unit, _ => exists tt
  | |- forall _ : unit, _ => intros ?
  | x : unit |- _ => destruct x
  end; eauto with zarith.
```

This tactic can automatically construct the group of integers from the appropriate constituents.

```
Module BinaryIntegers.

Module Additive.

#[export] Instance has_eq_rel : HasEqRel Z := eq.
#[export] Instance has_null_op : HasNullOp Z := Z.zero.
#[export] Instance has_un_op : HasUnOp Z := Z.opp.
#[export] Instance has_bin_op : HasBinOp Z := Z.add.

#[export] Instance is_grp : IsGrp eq Z.zero Z.opp Z.add.
Proof. ecrush. Qed.

End Additive.

End BinaryIntegers.
```

[21] When we say artificial intelligence, we mean logic programming, although machine learning can also be used to guide proof search.

The same applies to the trivial group.

```
Module Trivial.

Equations tt1 (x : unit) : unit :=
  tt1 _ := tt.

Equations tt2 (x y : unit) : unit :=
  tt2 _ _ := tt.

#[export] Instance has_eq_rel : HasEqRel unit := eq.
#[export] Instance has_null_op : HasNullOp unit := tt.
#[export] Instance has_un_op : HasUnOp unit := tt1.
#[export] Instance has_bin_op : HasBinOp unit := tt2.

#[export] Instance is_grp : IsGrp eq tt tt1 tt2.
Proof. ecrush. Qed.

End Trivial.
```

While formalizing the trivial group is completely straightforward, doing the same to the group of integers comes with one notable design consideration. We need to put the operational instances into their own little module, so that we can avoid the ambiguity between the additive monoid making up the group and the multiplicative monoid that could be defined otherwise. With this module structure, either monoid can be imported into a context or both can be brought together to form a semiring.

Besides forming a group, integers can act on other groups through repetition. These repetitions appear in additive structures as multiples and in multiplicative structures as powers. We can formalize this notion of repetition as `rep`.

```
Section Context.

Context (A : Type)
  {X : HasEqRel A} {x : HasNullOp A} {f : HasUnOp A} {k : HasBinOp A}
  `{!IsGrp X x f k}.

Equations rep (n : Z) (y : A) : A :=
  rep Z0 y := 0;
  rep (Zpos n) y := Pos.iter_op _+_ n y;
  rep (Zneg n) y := - Pos.iter_op _+_ n y.

End Context.
```

Even though we use additive structures as a motif for the notations, the resulting definition is actually more general, because the notations are forgotten at the end of the context. Now, while repetition is not a group action, its fibers are group homomorphisms, so we could set up a context and show that $n \times \square$ is indeed a group homomorphism for every $n : \mathbb{Z}$.

```
Section Context.

Context (A : Type)
  (X : A -> A -> Prop) (x : A) (f : A -> A) (k : A -> A -> A)
  `{!IsGrp X x f k}.

#[local] Instance has_eq_rel : HasEqRel A := X.
#[local] Instance has_null_op : HasNullOp A := x.
#[local] Instance has_un_op : HasUnOp A := f.
#[local] Instance has_bin_op : HasBinOp A := k.

#[local] Instance is_grp_hom (y : A) :
  IsGrpHom eq Z.zero Z.opp Z.add X x f k (flip rep y).
Proof. Admitted.

End Context.
```

The free group is constructed in three steps. First, we define the carrier and prove that it behaves as expected. Then, we implement the operations and verify that they preserve the expected behavior. Finally, we show that the carrier and the operations form a group. On every step, we employ an equational reasoning plugin [64] to simplify dealing with partial and recursive definitions.

We can formalize the carrier of the free group as the dependent pair `free`.

```
Module Free.

Section Context.

Context (A : Type) {e : HasEqDec A}.

Equations wfb_def (a b : bool * A) : bool :=
  wfb_def (i, x) (j, y) := decide (~ (i <> j /\ x = y)).

Equations wfb (s : list (bool * A)) : bool :=
  wfb s := decide (Forall (prod_uncurry wfb_def) (combine s (skipn 1 s))).

Equations free : Type :=
  free := {s : list (bool * A) | wfb s}.
```

These definitions relate to the model in that `Forall` is the composition of and and map, `prod_uncurry wfb_def` is φ, `combine` is zip and `skipn` is drop. It follows from `uip` that A is a set and that `free` is a trivial subset of `list (bool * A)`.

```
Lemma free_iff (x y : free) : x = y <-> proj1_sig x = proj1_sig y.
Proof.
  destruct x as [s a], y as [t b]. unfold proj1_sig. split.
  - intros c. inversion c as [d]. subst t. reflexivity.
  - intros d. subst t. f_equal. apply uip. Qed.
```

We can further formalize the operations as the functions `null`, `un` and `bin`.

```
Equations null : free :=
  null := ([]; _).
Next Obligation. ecrush. Qed.

Equations un (x : free) : free :=
  un (s; _) := (map (prod_bimap negb id) (rev s); _).
Next Obligation. Admitted.

```

```
121  Equations bin_fix (s t : list (bool * A)) :
122    list (bool * A) * list (bool * A) by struct t :=
123    bin_fix [] t := ([], t);
124    bin_fix s [] := (s, []);
125    bin_fix (a :: s) (b :: t) with wfb_def a b :=
126      | true => (a :: s, b :: t)
127      | false => bin_fix s t.
128
129  Equations bin (x y : free) : free :=
130    bin (s; _) (t; _) with bin_fix (rev s) t :=
131      | (u, v) => (app (rev u) v; _).
132  Next Obligation. Admitted.
```

These definitions relate to the model in that `[]` is ∅, `_ :: _` is □◁□ and `prod_bimap` is another variation of `map`. The only thing left to do is to show that we have a group on our hands.

```
134  #[export] Instance has_eq_rel : HasEqRel free := eq.
135  #[export] Instance has_null_op : HasNullOp free := null.
136  #[export] Instance has_un_op : HasUnOp free := un.
137  #[export] Instance has_bin_op : HasBinOp free := bin.
138
139  #[export] Instance is_grp : IsGrp eq null un bin.
140  Proof. Admitted.
141
142  End Context.
145
146  End Free.
```

Except, we cheat a little bit and admit the proofs on lines 119, 132 and 140 due to their length. Even though the admissions are ordinary proofs by induction, they cannot be automatically discovered by the compiler and, thus, require user interaction. If we wanted to show them here, we would have to expend anything from ten to a hundred lines to write each one[22].

We can now define the projections between our groups and show that they are truly structure-preserving. This is perhaps the most important part of the implementation in terms of practical applications, because it gives us the tools to embed meaningful concepts into meaningless data structures. We can formalize the evaluation map from the free group into the group of integers as `eval_Z_add`.

```
212  Module Initial.
213
214  Export Free BinaryIntegers.Additive.
215
216  Section Context.
217
```

[22] If you really wanted to prove the obligation on line 132, you would have to restate the definition using the inspect pattern, because otherwise your hypotheses would be too weak.

```
130    bin (s; _) (t; _) with (bin_fix (rev s) t; eq_refl) :=
131      | ((u, v); _) => (app (rev u) v; _).
```

```
218 Context (A : Type) {e : HasEqDec A} (f : A -> Z).
219
220 Equations eval_Z_add_def (a : bool * A) : Z :=
221   eval_Z_add_def (false, x) := f x;
222   eval_Z_add_def (true, x) := - f x.
223
224 Equations eval_Z_add (x : free A) : Z :=
225   eval_Z_add (s; _) := fold_right _+_ 0 (map eval_Z_add_def s).
```

These definitions relate to the model in that `e` witnesses that A is discrete, `eval_Z_add_def` is $\varepsilon(f)$ and `eval_Z_add` is $e(f)$. We can finally prove that the evaluation map is truly a group homomorphism.

```
227 #[local] Instance is_grp_hom :
228   IsGrpHom eq null un bin eq Z.zero Z.opp Z.add eval_Z_add.
229 Proof. Admitted.
230
231 End Context.
232
233 End Initial.
```

Except, we admit the proof on line 229 for the same reason as before. If we wanted to give the same treatment to the trivial group, we would not have to admit anything. We could simply formalize the constant map from any group into the trivial group as `const tt` and prove its properties with `ecrush`.

```
235 Module Terminal.
236
237 Export Trivial.
238
239 Section Context.
240
241 Context (A : Type)
242   (X : A -> A -> Prop) (x : A) (f : A -> A) (k : A -> A -> A)
243   `{!IsGrp X x f k}.
244
245 #[local] Instance is_grp_hom :
246   IsGrpHom X x f k eq tt tt1 tt2 (const tt).
247 Proof. ecrush. Qed.
248
249 End Context.
250
251 End Terminal.
```

This concludes our implementation of the model. Without any of its dependencies, it is 256 lines long with about as much reserved for the admitted proofs. At this point, we could easily prove the previously discussed excerpt in a just a few lines by setting up the appropriate context and invoking four well-chosen tactics[23].

[23] It is a good exercise for the reader to daydream about the way to prove the excerpt.

3.4.3 Extraction and Compilation into Machine Code

Once the implementation has passed type checking, we can carry on to extract code from it. The purpose of extraction is to erase all the proofs and translate the remaining pieces into a computationally usable form. This form is typically a module in another programming language, which can be interpreted or compiled into machine code.

We can ask Coq to extract an OCaml module from our implementation by using the rules listed in `ExtrOcamlBasic` and `ExtrOcamlZBigInt`.

```
From Coq Require Import Extraction ExtrOcamlBasic ExtrOcamlZBigInt.

Extraction "groupTheory.ml" Groups.
```

We choose OCaml as the extraction language, because Coq itself is implemented in it and, being a dialect of ML, it is familiar and has pleasantly predictable performance characteristics. The extracted module comes with the following interface.

```
module Groups : sig
  module Free : sig
    val wfb_def : 'a eqDec -> (bool * 'a) -> (bool * 'a) -> bool
    val wfb : 'a eqDec -> (bool * 'a) list -> bool
    type 'a free = (bool * 'a) list
    val null : 'a eqDec -> 'a free
    val un : 'a eqDec -> 'a free -> 'a free
    val bin_fix :
      'a eqDec -> (bool * 'a) list -> (bool * 'a) list ->
      (bool * 'a) list * (bool * 'a) list
    val bin : 'a eqDec -> 'a free -> 'a free -> 'a free
    val has_null_op : 'a eqDec -> 'a free hasNullOp
    val has_un_op : 'a eqDec -> 'a free hasUnOp
    val has_bin_op : 'a eqDec -> 'a free hasBinOp
  end
  module BinaryIntegers : sig
    module Additive : sig
      val has_null_op : Big.big_int hasNullOp
      val has_un_op : Big.big_int hasUnOp
      val has_bin_op : Big.big_int hasBinOp
    end
    val rep : 'a hasNullOp -> 'a hasUnOp -> 'a hasBinOp ->
      Big.big_int -> 'a -> 'a
  end
  module Trivial : sig
    val tt1 : unit -> unit
    val tt2 : unit -> unit -> unit
    val unit_has_null_op : unit hasNullOp
    val unit_has_un_op : unit hasUnOp
    val unit_has_bin_op : unit hasBinOp
  end
  module Initial : sig
    val eval_Z_add_def : ('a -> Big.big_int) ->
      (bool * 'a) -> Big.big_int
    val eval_Z_add : 'a eqDec -> ('a -> Big.big_int) ->
      'a Free.free -> Big.big_int
  end
  module Terminal : sig
  end
end
```

We can further compile the extracted module into less than 28 kiB of machine code and link it with the standard library to produce an executable less than 1.2 MiB in size. While the size itself is not indicative of good performance, it at least suggests that the result is not bloated.

As a case study, consider taking the abstract group expression $((x \times y) \times 1) \times (y \times y)^{-1}$, simplifying it into $x \times y^{-1}$, projecting it into the group of integers and evaluating it in a context, where $x \equiv 42$ and $y \equiv 13$. We can leverage the extracted module to do this as follows.

```
let main () =
  let open Groups in
  let e = Free.bin (=)
    (Free.bin (=) [(false, 'x'); (false, 'y')] (Free.null (=)))
    (Free.un (=) [(false, 'y'); (false, 'y')]) in
  let g = function
    | 'x' -> Big.of_int 42
    | 'y' -> Big.of_int 13
    | _ -> raise Not_found in
  Printf.printf "%s\n" (Big.to_string (Initial.eval_Z_add (=) g e))
```

This program prints 29 and exits, just as you would expect.

It should not surprise anybody that we can do simple arithmetic on small integers, but it might raise some eyebrows that we can bring symbolic manipulation and numerical computation together in such an elegant way. We can simultaneously minimize the burden of numerical computations by doing as many symbolic manipulations as possible and guarantee that the result is correct by writing proofs that are erased during compilation.

Even though the result is guaranteed to be correct, the old adage "garbage in, garbage out" still applies. Since extraction erases all the proofs and we did not explicitly add any assertions, the user is solely responsible for providing valid inputs to the extracted module. In our case study of the most roundabout way to print 29, the expressions $x \times y$ and $y \times y$ given by the user had to be well-formed or the behavior of the program would have been undefined[24]. While it would have been possible to validate inputs before using them, we deliberately avoided doing so, because it would have been unnecessary and potentially detrimental to performance.

4 Unexploited Opportunities?

We believe proof engineering tools and techniques could truly change the way computational sciences are studied and communicated. In order to explore what this

[24] Undefined behavior is bad, because it means that the program could do literally anything, including quietly producing a wrong result, crashing and setting a nearby printer on fire.

change could look like, we shall compare a traditional way to conduct a research project with a new way we consider worth trying. The comparison is, of course, exaggerated, but that is unavoidable, because nuance would only serve to make it incomprehensible.

If you want to bake a traditional research project in computational sciences, you can follow this recipe.

1. Choose an interesting problem.
2. Use pen and paper to derive an algorithm for solving your problem.
3. Write a FORTRAN or C program[25] that implements your algorithm efficiently.
4. Test your program thoroughly.
5. If defects are found, fix them and go back to step 2 or 3. Otherwise, assume that no significant defects remain.
6. Run your program and gather the results.
7. Publish your results and hopefully the source code of your program as well.
8. If your publication is not conclusive, go back to step 2. Otherwise, move on to the next problem.

However, if you want to be more radical, we suggest trying this new recipe instead.

1. Choose an interesting problem.
2. Use Coq to specify an algorithm for solving your problem.
3. Write proofs in Coq to implement and verify the correctness of your algorithm.
4. Mechanically extract OCaml code[26] from your implementation and choose one of the following options.
   a. Tune the extraction mechanism of Coq and the optimizer of the OCaml compiler until you can do all your computations using the extracted code.
   b. Link the extracted code with a C or FORTRAN program that can handle the most demanding computations, but do the rest of your computations using the extracted code.
   c. Use the extracted code as a testing oracle [68] for another C or FORTRAN program that can do all your computations.
5. Run your program and gather the results.
6. Publish your results and the source code of your program, because you cannot meaningfully separate them anymore.
7. If your publication is not conclusive, go back to step 2. Otherwise, move on to the next problem.

There are good reasons for and against the new recipe, just like there once were about the traditional one.

[25] When we say FORTRAN or C, we refer to any of their dialects, derivatives or wrappers, such as Fortran 95, C++, Julia or Python.

[26] We use Coq to produce OCaml, but you could just as well use Isabelle to produce Scala, Agda to produce Haskell or what have you.

4.1 Reasons to Get Excited

Consider formalizing a general computational method, such as discrete exterior calculus (DEC) [40], finite element method (FEM) [71] or discrete element method (DEM) [82]. At best, such a formalization could allow us to state a boundary value problem as it is written on paper, simplify the statement in a way that is guaranteed to be correct and solve the simplified form either by realizing it through extraction or by feeding it into an existing numerical solver. Any assumptions about the method would be explicit in its specification and proven to be upheld by its implementation, making accidental misuse of the method nearly impossible[27].

Besides giving us more correctness guarantees, formalizing a general method would also help us develop it further. The foundations and possible generalizations of DEC are still vague and poorly understood [43], while FEM and DEM have so many variations that it is difficult to keep track of them all [7]. With luck, a proper formalization could unify some of these methods by revealing them to be special cases of a more general theory.

Beauty and elegance are not the only reasons to pursue the unification of theories. It is a recurring problem in mathematics that seemingly novel ideas are found to be recastings of old ideas [69] or even plain wrong [74]. To make matters worse, these problems become progressively harder to notice as the level of abstraction increases or the clarity of communication deteriorates. Formalization helps us avoid these problems by forcing everyone to uphold a certain standard of rigor and openness in their communications. The emphasis on constructive foundations also pushes everyone towards being explicit about the computational content of their proofs. This philosophy is known to annoy some mathematicians [10], but fits computational scientists like a glove.

If you are still unconvinced that formalizing anything could be a good idea, consider the personal anecdote that, in our experience, it is really fun. Getting an ITP to accept your proof feels like playing a game and truly caresses your sensibilities if you enjoy logic puzzles and functional programming.

4.2 Reasons to Remain Skeptical

Since all prominent ITPS have constructive foundations and it is only possible to extract code from definitions that do not use classical axioms, learning to work with them can be difficult for people who are used to classical objects, such as real numbers or smooth manifolds. It is almost a rite of passage for newcomers to realize how useless real analysis is for computing anything. Computational methods always deal with rational approximations, while real analysis can only tell us about the limit.

[27] Unlike people working in software security, we do not need to worry about deliberate misuse, because our only adversary is reality.

Books [11] and theses [24] have been written on constructive analysis, but they are much more intricate and less popular than their classical counterparts.

Even in the realm of constructive mathematics, some things can be tricky to define inside type theory. Equality is one of them [70] and this issue also came up in our investigation of groups. We defined groups with respect to an arbitrary equivalence relation, which made the definition twice as large as it really needed to be, because we had to explicitly establish that $-\square$ and $\square + \square$ indeed preserve equivalences. The proliferation of these kinds of properties eventually leads to a problem that is colloquially known as setoid hell [4]; it is a place, where you have to manually witness the respectfulness of every operation with respect to every other relation. Univalent type theories, such as homotopy type theory and cubical type theory, alleviate the setoid problem by strengthening the concept of equality, but they are still otherwise quite immature.

Regardless of the type theory that is used, writing formal proofs in an ITP is hard work. While this is unlikely to change in the foreseeable future, more and more common tasks can be automated as time goes on. There are decision procedures for intuitionistic propositional calculus [28, 29] and quantifier-free fragments of integer arithmetics [8] as well as integrated ATPS for boolean satisfiability problems [25]. Artificial intelligence also plays a dual role: logic programming can already be used to guide heuristic reasoning [62] and the use of machine learning is being investigated for more intelligent proof search [46].

While performance is not a theoretical obstacle for formalizing computational methods, it is not a trivial concern either. It can be challenging to keep the performance of an ITP at a usable level, as the ambitions of its users and the scale of their developments grow [36]. Attempts to solve differential equations without relying on code extraction have not been able to reach satisfactory precision [52]. Extraction is not a silver bullet either, as it takes skill to manage the performance characteristics of extracted code [57, 49, 50]. However, as difficult as software engineering may be, it has been demonstrated that deeply-embedded domain-specific languages are excellent tools for abstract computational problems, such as performing automatic differentiation [30] or designing electronic circuits [81]. Even in general, these difficulties do not seem to be stopping people from bringing functional programming into computational sciences [79, 38].

5 Conclusions

Proof engineering does not seem to be unfit for computational sciences in any outstanding way. On the contrary, their intersection could actually be quite fruitful. Computational scientists would get new tools and techniques for improving the quality of their programs and proof engineers would benefit from testing their developments in a domain that aligns so well with their philosophy. Collectively, we could get improved confidence in our results, reducing anxiety and shifting some of the burden from reviewers to computers.

There are challenges to be overcome, but they all seem to be tractable with the current state of the art. Ideally, we want to see a world, where the computer is not just a worker, but a companion.

Acknowledgements

I would like to thank my friends and colleagues Paolo Giarrusso, Suvi Lahtinen and Fabi Prezja for encouragement and minor feedback on this text. I must also acknowledge Pekka Neittaanmäki for showing interest in my research and letting me crash his parties.

References

[1] Carlo Angiuli et al. "Internalizing Representation Independence with Univalence". In: *Proceedings of the ACM on Programming Languages* (Jan. 2021). Ed. by ACM. Vol. 5. POPL. ACM, 2021, pp. 12/1–30.

[2] Alasdair Armstrong et al. "ISA Semantics for ARMv8-a, RISC-v, and CHERI-MIPS". In: *Proceedings of the ACM on Programming Languages* (Jan. 2019). Ed. by ACM. Vol. 3. POPL. ACM, 2019, pp. 71/1–31.

[3] Steve Awodey, Nicola Gambino, and Kristina Sojakova. "Inductive Types in Homotopy Type Theory". In: *2012 27th Annual IEEE Symposium on Logic in Computer Science* (June 2012). Ed. by IEEE. Dubrovnik, Croatia: IEEE, 2012, pp. 95–104.

[4] Gilles Barthe, Venanzio Capretta, and Olivier Pons. "Setoids in Type Theory". In: *Journal of Functional Programming* 13.2 (2003). Special Issue on "Logical Frameworks and Metalanguages", pp. 261–293.

[5] Andrej Bauer. *What makes dependent type theory more suitable than set theory for proof assistants?* Nov. 20, 2020. eprint: `376973`. URL: `https://mathoverflow.net/q/376973`.

[6] Andrej Bauer et al. "The HoTT Library: A Formalization of Homotopy Type Theory in Coq". In: *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs* (Jan. 2017). Ed. by ACM. Paris, France: ACM, 2017, pp. 164–172.

[7] Ted Belytschko, Robert Gracie, and Giulio Ventura. "A Review of Extended/-Generalized Finite Element Methods for Material Modeling". In: *Modelling and Simulation in Materials Science and Engineering* 17.4 (2009), p. 043001.

[8] Frédéric Besson. "Fast Reflexive Arithmetic Tactics the Linear Case and Beyond". In: *Types for Proofs and Programs* (Apr. 2006). Ed. by Thorsten Altenkirch and Conor McBride. Vol. 4502. Lecture Notes in Computer Science. Nottingham, UK: Springer, 2006, pp. 48–62.

[9] Kevin Bierhoff and Jonathan Aldrich. “Modular Typestate Checking of Aliased Objects”. In: *ACM SIGPLAN Notices* 42.10 (2007), pp. 301–320.

[10] Errett Bishop. “The Crisis in Contemporary Mathematics”. In: *Historia Mathematica* 2.4 (1975), pp. 507–517.

[11] Errett Bishop and Douglas Bridges. *Constructive Analysis*. Springer, 1985.

[12] Steve Bishop et al. “Engineering with Logic: Rigorous Test-Oracle Specification and Validation for TCP/IP and the Sockets API”. In: *Journal of the ACM* 66.1 (2018), pp. 1–77.

[13] Jasmin Blanchette et al. “A Verified SAT Solver Framework with Learn, Forget, Restart, and Incrementality”. In: 61 (2018), pp. 333–365.

[14] Ana Bove and Venanzio Capretta. “Modelling General Recursion in Type Theory”. In: *Mathematical Structures in Computer Science* 15.4 (2005), pp. 671–708.

[15] Kevin Buzzard, Johan Commelin, and Patrick Massot. “Formalising Perfectoid Spaces”. In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Jan. 2020). Ed. by ACM. New Orleans, Los Angeles, USA: ACM, 2020, pp. 299–312.

[16] Lloyd Campbell et al. “Fortran 77”. In: *Communications of the ACM* 21.10 (1978). Ed. by Walt Brainerd, pp. 806–820.

[17] Haogang Chen et al. “Verifying a High-Performance Crash-Safe File System Using a Tree Specification”. In: *Proceedings of the 26th Symposium on Operating Systems Principles* (Oct. 2017). Ed. by ACM. Shanghai, China: ACM, 2017, pp. 270–286.

[18] Jesper Cockx and Andreas Abel. “Elaborating Dependent (Co)pattern Matching”. In: *Proceedings of the ACM on Programming Languages* (Sept. 2018). Ed. by ACM. Vol. 2. ICFP. ACM, 2018, pp. 75/1–30.

[19] Cyril Cohen et al. *Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom*. 2016. eprint: `1611.02108`. URL: `https://arxiv.org/abs/1611.02108`.

[20] Thierry Coquand. “Pattern Matching with Dependent Types”. In: *Proceedings of the Workshop on Types for Proofs and Programs* (June 1992). Ed. by Bengt Nordström, Kent Petersson, and Gordon Plotkin. Båstad, Sweden: Springer, 1992, pp. 71–84.

[21] Thierry Coquand, Simon Huber, and Anders Mörtberg. “On Higher Inductive Types in Cubical Type Theory”. In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science* (July 2018). Ed. by ACM. Oxford, UK: ACM, 2018, pp. 255–264.

[22] Thierry Coquand and Gérard Huet. “The Calculus of Constructions”. In: *Information and Computation* 76.2-3 (1988), pp. 95–120.

[23] Thierry Coquand and Christine Paulin. “Inductively Defined Types”. In: *COLOG-88* (Dec. 1988). Ed. by Per Martin-Löf and Grigori Mints. Vol. 417. Lecture Notes in Computer Science. Tallinn, USSR: Springer, 1990, pp. 50–66.

[24] Luís Cruz-Filipe. “Constructive Real Analysis: A Type-Theoretical Formalization and Applications”. PhD thesis. University of Nijmegen, 2004.

[25] Lukasz Czajka and Cezary Kaliszyk. “Hammer for Coq: Automation for Dependent Type Theory”. In: *Journal of Automated Reasoning* 61.1 (2018), pp. 423–453.

[26] Mads Dam et al. “Formal Verification of Information Flow Security for a Simple ARM-Based Separation Kernel”. In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (Nov. 2013). Ed. by ACM. Berlin, Germany: ACM, 2013, pp. 223–234.

[27] Edsger Wybe Dijkstra. *On the Reliability of Programs. EWD 303*. 1971.

[28] Roy Dyckhoff. “Contraction-Free Sequent Calculi for Intuitionistic Logic”. In: *The Journal of Symbolic Logic* 57.3 (1992), pp. 795–807.

[29] Roy Dyckhoff. “Contraction-Free Sequent Calculi for Intuitionistic Logic: A Correction”. In: *The Journal of Symbolic Logic* 83.4 (2018), pp. 1680–1682.

[30] Conal Elliott. “The Simple Essence of Automatic Differentiation”. In: *Proceedings of the ACM on Programming Languages* (Sept. 2018). Ed. by ACM. Vol. 2. ICFP. ACM, 2018, pp. 70/1–29.

[31] François Garillot. “Generic Proof Tools and Finite Group Theory”. PhD thesis. Ecole Polytechnique X, 2011.

[32] Herman Geuvers, Freek Wiedijk, and Jan Zwanenburg. “A Constructive Proof of the Fundamental Theorem of Algebra without Using the Rationals”. In: *Types for Proofs and Programs* (Dec. 2000). Ed. by Paul Callaghan et al. Vol. 2277. Lecture Notes in Computer Science. Durham, UK: Springer, 2002, pp. 96–111.

[33] Gaëtan Gilbert et al. “Definitional Proof-Irrelevance without K”. In: *Proceedings of the ACM on Programming Languages* (Jan. 2019). Ed. by ACM. Vol. 3. POPL. ACM, 2019, pp. 3/1–28.

[34] Georges Gonthier. “Formal Proof — The Four-Color Theorem”. In: *Notices of the ACM* 55.11 (2008), pp. 1382–1393.

[35] Georges Gonthier et al. “A Machine-Checked Proof of the Odd Order Theorem”. In: *Interactive Theorem Proving* (July 2013). Ed. by Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie. Vol. 7998. Lecture Notes in Computer Science. Rennes, France: Springer, 2013, pp. 163–179.

[36] Jason Gross. “Performance Engineering of Proof-Based Software Systems at Scale”. PhD thesis. Massachusetts Institute of Technology, 2021.

[37] Thomas Hales et al. “A Revision of the Proof of the Kepler Conjecture”. In: *The Kepler Conjecture*. Ed. by Jeffrey Lagarias. Springer, 2011, pp. 341–376.

[38] Troels Henriksen et al. “Futhark: Purely Functional GPU-Programming with Nested Parallelism and In-Place Array Updates”. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (June 2017). Ed. by ACM. Barcelona, Spain: ACM, 2017, pp. 556–571.

[39] Ralf Hinze and Ross Paterson. “Finger Trees: A Simple General-Purpose Data Structure”. In: *Journal of Functional Programming* 16.2 (2006), pp. 197–217.

[40] Anil Hirani. “Discrete Exterior Calculus”. PhD thesis. California Institute of Technology, 2003.

[41] Jason Hu and Jacques Carette. “Formalizing Category Theory in Agda”. In: *Proceedings of the 10th ACM SIGPLAN Conference on Certified Programs and Proofs* (Jan. 2021). Ed. by ACM. Denmark: ACM, 2021, pp. 327–342.

[42] Brian Kernighan and Dennis Ritchie. *The C Programming Language*. 2nd ed. Prentice Hall, 1988.

[43] Lauri Kettunen et al. “Generalized Finite Difference Schemes with Higher Order Whitney Forms”. In: *ESAIM: Mathematical Modelling and Numerical Analysis* 55.4 (2021), pp. 1439–1460.

[44] Sampsa Kiiskinen. *Discrete Exterior Zoo*. Jan. 7, 2022. URL: `https://github.com/Tuplanolla/dez`.

[45] Gerwin Klein et al. “Comprehensive Formal Verification of an OS Microkernel”. In: *ACM Transactions on Computer Systems* 32.1 (2014), pp. 1–70.

[46] Ekaterina Komendantskaya, Jónathan Heras, and Gudmund Grov. “Machine Learning in Proof General: Interfacing Interfaces”. In: *Proceedings 10th International Workshop on User Interfaces for Theorem Provers* (July 2012). Ed. by Cezary Kaliszyk and Christoph Lüth. Vol. 7941. Electronic Proceedings in Theoretical Computer Science. Bremen, Germany: OPA, 2012, pp. 15–41.

[47] Nicolai Kraus et al. “Generalizations of Hedberg’s Theorem”. In: *Typed Lambda Calculi and Applications* (June 2013). Ed. by Masahito Hasegawa. Vol. 7941. Lecture Notes in Computer Science. Eindhoven, Netherlands: Springer, 2013, pp. 173–188.

[48] Xavier Leroy. “Formal Verification of a Realistic Compiler”. In: *Communications of the ACM* 52.7 (2009), pp. 107–115.

[49] Pierre Letouzey. “A New Extraction for Coq”. In: *Types for Proofs and Programs* (Apr. 2002). Ed. by Herman Geuvers and Freek Wiedijk. Vol. 2646. Lecture Notes in Computer Science. Berg en Dal, Netherlands: Springer, 2002, pp. 200–219.

[50] Pierre Letouzey. “Extraction in Coq: An Overview”. In: *Logic and Theory of Algorithms* (June 2008). Ed. by Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe. Vol. 5028. Lecture Notes in Computer Science. Athens, Greece: Springer, 2008, pp. 359–369.

[51] Junyi Liu et al. “Formal Verification of Quantum Algorithms Using Quantum Hoare Logic”. In: *Computer Aided Verification* (July 2019). Ed. by Isil Dillig and Serdar Tasiran. Vol. 11562. Lecture Notes in Computer Science. New York City, New York, USA: Springer, 2019, pp. 187–207.

[52] Evgeny Makarov and Bas Spitters. “The Picard Algorithm for Ordinary Differential Equations in Coq”. In: *Interactive Theorem Proving* (July 2013). Ed. by Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie. Vol. 7998. Lecture Notes in Computer Science. Rennes, France: Springer, 2013, pp. 463–468.

[53] Per Martin-Löf. “An Intuitionistic Theory of Types”. In: *Twenty-Five Years of Constructive Type Theory* 36 (1998), pp. 127–172.

[54] Erik Meijer, Maarten Fokkinga, and Ross Paterson. “Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire”. In: *Conference on Functional Programming Languages and Computer Architecture* (Aug. 1991). Ed.

by John Hughes. Vol. 523. Lecture Notes in Computer Science. Cambridge, Massachusetts, USA: Springer, 1991, pp. 124–144.

[55] Cleve Moler. *MATLAB Incorporates LAPACK*. 2000.

[56] Russell O'Connor. "Essential Incompleteness of Arithmetic Verified by Coq". In: *Theorem Proving in Higher Order Logics* (Aug. 2005). Ed. by Joe Hurd and Tom Melham. Vol. 3603. Lecture Notes in Computer Science. Oxford, UK: Springer, 2005, pp. 245–260.

[57] Christine Paulin-Mohring and Benjamin Werner. "Synthesis of ML Programs in the System Coq". In: *Journal of Symbolic Computation* 15 (1993), pp. 607–640.

[58] Frank Pfenning and Christine Paulin-Mohring. "Inductively Defined Types in the Calculus of Constructions". In: *Proceedings of the 5th Conference on Mathematical Foundations of Programming Semantics* (Mar. 1989). Ed. by Michael Main et al. Vol. 442. Lecture Notes in Computer Science. New Orleans, Louisiana, USA: Springer, 1990, pp. 209–228.

[59] Benjamin Pierce. "Lambda, the Ultimate TA: Using a Proof Assistant to Teach Programming Language Foundations". In: *ACM SIGPLAN Notices* 44.9 (2009), pp. 121–122.

[60] Talia Ringer et al. "QED at Large: A Survey of Engineering of Formally Verified Software". In: *Foundations and Trends in Programming Languages* 5.2-3 (2019), pp. 102–281.

[61] Daniel Selsam, Percy Liang, and David Dill. "Developing Bug-Free Machine Learning Systems with Formal Mathematics". In: *Proceedings of the 34th International Conference on Machine Learning* (Aug. 2017). Ed. by Doina Precup and Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research. Sydney, Australia: PMLR, 2017, pp. 3047–3056.

[62] Daniel Selsam, Sebastian Ullrich, and Leonardo de Moura. *Tabled Typeclass Resolution*. 2018. eprint: `2001.04301`. URL: `https://arxiv.org/abs/2001.04301`.

[63] Ilya Sergey, James Wilcox, and Zachary Tatlock. "Programming and Proving with Distributed Protocols". In: *Proceedings of the ACM on Programming Languages* (Jan. 2018). Ed. by ACM. Vol. 2. POPL. ACM, pp. 28/1–30.

[64] Matthieu Sozeau and Cyprien Mangin. "Equations Reloaded: High-Level Dependently-Typed Functional Programming and Proving in Coq". In: *Proceedings of the ACM on Programming Languages* (Aug. 2019). Ed. by ACM. Vol. 3. ICFP. ACM, 2019, pp. 86/1–29.

[65] Matthieu Sozeau and Nicolas Oury. "First-Class Type Classes". In: *Theorem Proving in Higher Order Logics* (Aug. 2008). Ed. by Otmane Ait Mohamed, César Munoz, and Sofiène Tahar. Vol. 5170. Lecture Notes in Computer Science. Montreal, Canada: Springer, 2008, pp. 278–293.

[66] Matthieu Sozeau et al. "Coq Coq Correct! Verification of Type Checking and Erasure for Coq, in Coq". In: *Proceedings of the ACM on Programming Languages* (Jan. 2020). Ed. by ACM. Vol. 4. POPL. ACM, 2020, pp. 8/1–28.

[67] Bas Spitters and Eelis van der Weegen. "Type Classes for Mathematics in Type Theory". In: *Mathematical Structures in Computer Science* 21.4 (2011), pp. 795–825.

[68] Matt Staats, Michael Whalen, and Mats Heimdahl. "Programs, Tests, and Oracles: The Foundations of Testing Revisited". In: *2011 33rd International Conference on Software Engineering* (May 2011). Ed. by IEEE. Honolulu, Hawaii, USA: IEEE, 2011, pp. 391–400.

[69] Mary Tai. "A Mathematical Model for the Determination of Total Area Under Glucose Tolerance and Other Metabolic Curves". In: *Diabetes Care* 17.2 (1994), pp. 152–154.

[70] The Univalent Foundations Program. *Homotopy Type Theory. Univalent Foundations of Mathematics*. Institute for Advanced Study, June 19, 2013.

[71] Vidar Thomée. "From Finite Differences to Finite Elements: A Short History of Numerical Analysis of Partial Differential Equations". In: *Journal of Computational and Applied Mathematics* 128.1-2 (2001), pp. 1–54.

[72] Sophie Tourret and Jasmin Blanchette. "A Modular Isabelle Framework for Verifying Saturation Provers". In: *Proceedings of the 10th ACM SIGPLAN Conference on Certified Programs and Proofs* (Jan. 2021). Ed. by ACM. Denmark: ACM, 2021, pp. 224–237.

[73] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. "Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types". In: *Journal of Functional Programming* 31 (2021), p. 87.

[74] Vladimir Voevodsky. *The Origins and Motivations of Univalent Foundations. A Personal Mission to Develop Computer Proof Verification to Avoid Mathematical Mistakes*. Summer. 2014, pp. 8–9.

[75] Vladimir Voevodsky. "Univalent Foundations of Mathematics". In: *Logic, Language, Information and Computation* (May 2011). Ed. by Lev Beklemishev and Ruy de Queiroz. Vol. 6642. Lecture Notes in Computer Science. Philadelphia, Pennsylvania, USA: Springer, 2011, p. 4.

[76] Philip Wadler. "Propositions as Types". In: *Communications of the ACM* 58.12 (2015), pp. 75–84.

[77] Philip Wadler and Stephen Blott. "How to Make Ad-Hoc Polymorphism Less Ad Hoc". In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Jan. 1989). Ed. by ACM. Austin, Texas, USA: ACM, 1989, pp. 60–76.

[78] Stefan van der Walt, Chris Colbert, and Gael Varoquaux. "The NumPy Array: A Structure for Efficient Numerical Computation". In: *Computing in Science & Engineering* 13.2 (2011), pp. 22–30.

[79] Liang Wang and Jianxin Zhao. "OCaml Scientific Computing. Functional Programming Meets Data Science". In Progress. Jan. 1, 2022.

[80] Conrad Watt. "Mechanising and Verifying the WebAssembly Specification". In: *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Jan. 2018). Ed. by ACM. Los Angeles, California, USA: ACM, 2018, pp. 53–65.

[81] Rinse Wester. “A Transformation-Based Approach to Hardware Design Using Higher-Order Functions”. PhD thesis. Universiteit Twente, 2015.

[82] John Williams, Grant Hocking, and Graham Mustoe. “The Theoretical Basis of the Discrete Element Method”. In: *Proceedings of the NUMETA ’85 Conference* (Jan. 1985). Ed. by Gyanendra Pande and John Middleton. Swansea, UK: August Aimé Balkema, 1985, pp. 897–906.

[83] Doug Woos et al. “Planning for Change in a Formal Verification of the Raft Consensus Protocol”. In: *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs* (Nov. 2013). Ed. by ACM. Saint Petersburg, Florida, USA: ACM, 2016, pp. 154–165.