The Logic in Computer Science Column

by

Yuri Gurevich

Microsoft Research
One Microsoft Way, Redmond WA 98052, USA
gurevich@microsoft.com

Type Inference in Mathematics

Jeremy Avigad*

Abstract

In the theory of programming languages, *type inference* is the process of inferring the type of an expression automatically, often making use of information from the context in which the expression appears. Such mechanisms turn out to be extremely useful in the practice of interactive theorem proving, whereby users interact with a computational proof assistant to construct formal axiomatic derivations of mathematical theorems. This article explains some of the mechanisms for type inference used by the *Mathematical Components* project, which is working towards a verification of the Feit-Thompson theorem.

*This work has been partially supported by NSF grants DMS-0700174 and DMS-1068829. During the 2009–2010 academic year I spent a sabbatical year working on the verification of the Feit-Thompson theorem with the Mathematical Components group at INRIA-Microsoft Joint Research Centre in Orsay, and I am grateful to Georges Gonthier and the center for support. I am also grateful to Yuri Gurevich, Assia Mahboubi, Enrico Tassi, and an anonymous referee for many helpful comments, corrections, and suggestions.

1 Introduction

Consider the following mathematical assertions:

- For every x in $\mathbb{R}$, $e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!}$.
- If G and H are groups, f is a homomorphism from G to H, and a and b are in G, then $f(ab) = f(a)f(b)$.
- If F is a field of nonzero characteristic p, and a and b are in F, then

$$(a+b)^p = \sum_{i=0}^{p} \binom{p}{i} a^i b^{p-i} = a^p + b^p.$$

There is nothing unusual about these statements, but, on reflection, one notices that substantial background knowledge and assumptions are needed to parse them correctly. For example, in the first statement, we take it that the index of the summation i ranges over natural numbers, or, equivalently, nonnegative integers. Hence $i!$ is also an integer. Since x is explicitly flagged as a real number, the expression $x^i/i!$ involves division of two different types of objects, taking into account that any integer can be viewed as a real number. In the second statement, G and H are groups, which is to say, each is a set of elements equipped with a group operation and an identity element; so when we write that a and b are in G, we really mean that a and b are elements of the underlying set. The notation ab denotes multiplication in G, while the notation $f(a)f(b)$ can only be understood in terms of the multiplication in H. In the third statement, p is a nonnegative integer (in fact, a prime number, since nonzero field characteristics are prime). But unlike the summation symbol in the first statement, here the summation symbol refers to addition in F. In the third statement, $\binom{p}{i}$ is an integer, while a^i and b^{p-i} are elements of the field. How do we interpret multiplication in *that* case? One way is to notice that there is a canonical map from the integers to any ring with a 0 and a 1. Alternatively, any abelian group can be viewed as a $\mathbb{Z}$-module, which means that it supports scalar multiplication by integers, with all the expected properties; and the additive part of a ring is an abelian group.

Inferences like these are used not only to parse basic mathematical expressions, but also to reason about them correctly. For example, some "multiplications" and "additions" are commutative, and multiplication often distributes over the corresponding addition. Common manipulations with summations depend on such facts. Understanding mathematics presupposes the ability to keep track of the various domains that objects belong to and variables range over, as well as the relevant operations on those domains and their properties. Our faculties for doing

this are so ingrained that we are scarcely aware of the background knowledge we bring to the table when we read an ordinary mathematical text.

The problem is that such background knowledge has to be brought to the foreground when it comes to formalizing mathematics. Broadly speaking, *formal verification* is the practice of using formal methods to verify correctness, such as verifying that a circuit description, an algorithm, or a network or security protocol meets its specification. In this article, I will be concerned, instead, with the verification of mathematical theorems. To be sure, there is no sharp distinction between verifying mathematical statements and verifying claims about hardware and software systems, since the latter are typically expressed in mathematical terms. But ordinary mathematical theorems have a special character, and raise distinct issues and challenges.

Specifically, I will focus on *interactive theorem proving,* which involves working interactively with a proof assistant to provide enough information for the system to confirm that the theorem in question has, indeed, a formal proof. In fact, many systems actually construct a formal proof object, a complex piece of data that can be verified independently. Systems with substantial mathematical libraries include Coq [5] (including the Ssreflect extension [21]), HOL [24], HOL light [28], Isabelle [37], and Mizar [25]. In September 2004, assisted by some students at Carnegie Mellon, I verified a proof of the Hadamard/de la Vallée Poussin prime number theorem [3], using the Isabelle proof assistant. Since then number of nontrival theorems have been formalized.

, including the four-color theorem [18], the prime number theorem [3, 30], the Jordan curve theorem [26, 33], Gödel's first incompleteness theorem [42, 38], Dirichlet's theorem on primes in an arithmetic progression [29], the Cartan fixed-point theorems [9], and various theorems of measure theory [31, 35]. There are, moreover, some interesting large-scale verification projects underway. Thomas Hales is heading the *Flyspeck* project [27], which aims to verify a number of results in discrete geometry, including the Kepler conjecture. Georges Gonthier is heading the *Mathematical Components* project [17, 19], which aims to verify the Feit-Thompson theorem. Fields medalist Vladimir Voevodsky has launched a project to develop "univalent foundations" for algebraic topology, providing the basis for formal verification in a theorem prover like Coq.

Checking the details of a mathematical proof is by no means the most interesting or important part of mathematics, and formal verification is not meant to serve as a substitute for mathematical creativity and understanding. But it is generally recognized that the mathematical literature is filled with misstatements, gaps, ambiguities, overlooked cases, omitted hypotheses, and so on, and that the lack of reliability is problematic [36]. Moreover, an increasing number of proofs today rely on extensive calculation, and there are currently no standards to ensure that mathematical software is sound. Mathematicians always strive for correctness,

and formal verification is simply a technology that is designed to support that goal.

Despite the achievements to date, however, formal verification is still not "ready for prime time." There is a steep learning curve to working with an interactive theorem prover, and verifying even straightforward mathematical results can be frustrating and time consuming. We need better libraries, automated methods, and infrastructure to support verification efforts. This is an exciting time for a young and rapidly evolving field.

In this article, I will focus on one small aspect of formal verification, namely, type inference. In the mathematical setting, the challenge of type inference, roughly speaking, is to keep track of the kinds of objects that appear in a mathematical statement and put that information to good use. What is common to the previous examples is that in each case the relevant information can be inferred from context:

- In the expression "a is in G," the object of the word "in" is expected to be a set.
- In "ab," multiplication takes place in the group that a and b are assumed to be an element of.
- In "$x^i/i!$," one expects the arguments to be elements of a common structure, for which a division operation is defined.

Type inference thus involves not only inferring type information, but also inferring data and facts from type considerations. Of course, type inference is central to the theory of programming languages [39], and many of the ideas and methods that have been developed there have been transferred to the mathematical setting. But, as will become clear, mathematical type inference has a distinct flavor. Here I will focus primarily on the approach to type inference used in the Mathematical Components project, which relies on a proof language, Ssreflect, and the Coq theorem prover.

In Section 2, I will consider what is desired from a mathematical perspective. In Section 3, I will discuss some of the underlying axiomatic frameworks, and dependent type theory in particular. In Section 4, I will describe some of the mechanisms in Coq that are designed to meet the challenges posed in Section 2. In Section 5, I will describe the way some of these mechanisms are used in the Mathematical Components library, and in Section 6, I will briefly indicate some alternative approaches.

2 Mathematical type inference

One hallmark of modern mathematics is the tendency to identify mathematical objects as elements of algebraically characterized structures. Such structures, and classes of such structures, can be related in various ways:

- Structures in one class may be viewed as elements of a broader one. For example, every abelian group is, more generally, a group, and every group is, more generally, a monoid. Sometimes the inclusions are obtained by taking reducts, which is to say, ignoring parts of the structure. For example, the additive part of a ring is an abelian group, while the multiplicative part is a monoid.

- A particular structure or a structure in one class can often be embedded in a larger structure. For example, the integers can be embedded in the reals, and every integral domain can be embedded in its field of fractions.

- Uniform constructions can be used to build elements of one class of structures from elements of another. For example, the units in any ring form a group, under the associated multiplication; the set of automorphisms of a field (or those fixing some chosen subfield) form a group under composition; any metric space gives rise to a topological space determined by the metric; the field of fractions of any integral domain is a field; and the quotient of a group by a normal subgroup is again a group.

What makes this perspective useful is that it allows one to transfer insights and results gained from one domain to another, and apply background knowledge and expertise uniformly in different settings. The challenge for interactive proof assistants is to reap these benefits.

There are various ways that algebraic methods promote efficiency:

- They allow us to reuse notation. For example, one may wish to use the symbols 0 and + with respect to the integers, the reals, and arbitrary rings.

- They allow us to reuse constructions. For example, summation $\sum_{i \in I} a_i$ in the integers, reals, and an arbitrary ring can be viewed as instances of the same construction, namely an iteration of the corresponding addition. In fact, various "big" operations, including multiplication, logical operations of conjunction and disjunction, lattice operations of meet and join, and so on can be viewed as iterations of an associative operation in an arbitrary monoid.

- They allow us to reuse facts. Various identities involving big operations can be viewed as instances of general laws that can be instantiated in the different settings. For example, some identities involving summations presuppose that the addition is commutative. Other identities hold in the presence of a multiplication that distributes over addition. We implicitly recognize that such facts hold at various degrees of generality, and instantiate them as appropriate.

Any proof assistant that is designed to formalize contemporary mathematical arguments should support these types of reuse.

In the theory of programming languages, type inference allows users to omit information that can be inferred from context. For example, if we write $f(i)$ and i is known to range over the integers, we can infer that f is a function from the integers to some other domain. Various kinds of "polymorphism" allow one to reuse symbols and code across different domains. In the context of formally verified mathematics, there are really two types of information that can be inferred:

- data: for example, the appropriate multiplication in an expression $a \cdot b$, or the appropriate summation operation in an expression $\sum_{i \in A} f(i)$.

- facts: for example, the fact that $(a \cdot b) \cdot c$ is equal to $a \cdot (b \cdot c)$, when the multiplication in the relevant structure is associative.

In the next section, we will see that in certain formulations of logic, these two can be understood as instances of a common phenomena. In other words, inferring a fact can be viewed as inferring a special kind of data, namely "evidence" or "the fact" that the associated claim is true.

To summarize, in interactive theorem proving, type inference may be invoked when the system parses an expression, but also when the user applies a lemma, or searches for a lemma to apply. The goal of type inference is to allow the user to omit information systematically when such information can be inferred from context. Not only does this save time and energy and reduce tedium, but it also ensures that the expressions we type look like the mathematics we are familiar with, lending support to the claim that our formalizations adequately "capture" informal mathematical practice.

3 Dependent type theory

In order to verify mathematical proofs in a given domain, one has to first choose a formal axiomatic framework that is flexible enough to model arguments in that domain. Experience from the last century has shown that the Zermelo-Fraenkel

axioms of set theory provides a remarkably robust foundation for mathematics. Indeed, the Mizar system [25], which has perhaps the most extensive mathematical library, is based on an extension of ZF known as Tarski-Groethendieck set theory.

But, in set theory, every object is a set, meaning that the axiomatic framework does not distinguish between numbers, functions, structures, and other objects. For the purposes of type inference, it is often useful to have such distinctions built into the underlying formal system. A number of proof assistants today, including HOL [24], HOL light [28], and Isabelle [37], are based on formulations of higher-order logic like Church's simple type theory [8]. One starts with basic types, such as a type `nat` of natural numbers and a type `bool` of boolean truth values, and adds constructors for forming new types. The most important of these are function types: whenever `A` and `B` are types, so is $\texttt{A} \to \texttt{B}$, intended to denote the type of functions from `A` to `B`. One can also allow, for example, product types $\texttt{A} \times \texttt{B}$, denoting the type of ordered pairs from `A` and `B`. Most proof systems have additional mechanisms to support the definition of common mathematical data types and structures, and allow "polymorphic" variables ranging over types.

The problem with simple type theory, however, is that it is too simple, since ordinary mathematical structures often depend on parameters. For example, for each n, $\mathbb{R}^n$ is a vector space, and for every $n \geq 1$, the integers modulo n form a ring. Thus one may wish to have types

- `list A n`, denoting sequences of objects of type A, with length n; and
- `Zmod n`, denoting the ring of integers modulo n.

In *dependent type theory*, types can depend on parameters in this way. Notice that such a move tends to blur the distinction between types and terms. For example, in `list A n`, the first argument is supposed to denote a type, whereas the second argument is supposed to be a term of type `nat`. In some presentations of type theory, this is achieved by having special types, called *universes*, whose terms are also construed as types (see, for example, the presentation of Martin-Löf type theory in [47, Section 7.1]). Contemporary presentations more often take types to be inhabitants of a third level of syntactic objects, known as "sorts" or "kinds" (see [4]). The specific details need not concern us here; only the fact that terms as well as types can depend on parameters that are again terms or types.

In dependent type theory, the type $\texttt{A} \to \texttt{B}$ of functions which take an argument in `A` and return a value in `B` can be generalized to a dependent product $\prod_{\texttt{x:A}} \texttt{B(x)}$, where `B(x)` is a type that can depend on `x`. Intuitively, elements of this type are functions that map an element `a` of `A` to an element of `B(a)`. When `B` does not depend on `x`, the result is just $\texttt{A} \to \texttt{B}$. Similarly, product types $\texttt{A} \times \texttt{B}$ can be generalized to dependent sums $\sum_{\texttt{x:A}} \texttt{B(x)}$. Intuitively, elements of this type are

pairs `(a, b)`, where `a` is an element of `A` and `b` is an element of `B(a)`. When `B` does not depend on `x`, this is just $\mathtt{A} \times \mathtt{B}$.

In the next section, we will consider one particular theorem prover, Coq. Coq's underlying logic is a dependent type theory known as the *calculus of inductive constructions*, or *CIC* [12], which extends the original *calculus of constructions* due to Coquand and Huet [11]. The calculus of inductive constructions has four distinguishing features:

- It is a powerful and expressive dependent type theory.
- It incorporates the "propositions as types" correspondence.
- It is constructive, in that every expression in the system has a computational interpretation.
- The computational interpretation of terms is used in type checking.
- Type checking is decidable.

These features are not to everyone's taste, and we will see in Section 6 that other proof assistants can reject any or all of them. I will elaborate on each, in turn.

One striking feature of the Calculus of Inductive Constructions is that there are only two basic type-forming operations: dependent products and inductive types. We have already discussed dependent products. Inductive types allow one to define structures that can be characterized as the closure of a set under some basic operations, like the natural numbers, or lists and trees over a type. But, in the CIC, the construction is general enough to include dependent sums, as well as to interpret basic logical notions, like conjunction, disjunction, universal and existential quantification, and equality. In fact, the system has the logical strength of strong systems of set theory [49].

In order to interpret logical operations in terms of type-theoretic constructions, the CIC relies on what has come to be known as the Curry-Howard "propositions as types" correspondence. The point is that logical operations look a lot like operations on datatypes. For example, in propositional logic, from `A` and `B` one can conclude $\mathtt{A} \wedge \mathtt{B}$. One can read this as saying that given a proof `a` of `A` and a proof `b` of `B` of B one can "pair" them to obtain a proof `(a, b)` of $\mathtt{A} \wedge \mathtt{B}$; or given the "fact" `a` that `A` holds, and the fact `b` that `B` holds, one obtains the fact `(a, b)` that $\mathtt{A} \wedge \mathtt{B}$ holds. Moreover, from the fact that $\mathtt{A} \wedge \mathtt{B}$ holds, one can extract the fact that `A` holds, and, similarly, `B`. If you replace $\mathtt{A} \wedge \mathtt{B}$ by $\mathtt{A} \times \mathtt{B}$, this is nothing more than a characterization of the product type. In other words, if we posit a new collection `Prop` of types and take the product constructor to map elements `A : Prop` and `B : Prop` to $\mathtt{A} \times \mathtt{B}$ `: Prop`, the rules governing products for elements of `Prop` are exactly the desired logical rules for conjunction.

Under this correspondence, implications A → B are just instances of function types, and bounded universal quantifiers ∀x : A. B(x) are just instances of the dependent product construction. In other words, a proof of ∀x : A. B(x) can be viewed as a procedure which, given any object a : A returns a proof of B(a). This explains Coq's notation forall x : A, B x for dependent products. Similarly, the logical construction ∃x : A. B(x) is just an instance of the dependent sum. Using inductively defined types, given any type A one can form I_A(x,y) : Prop which, intuitively, denotes the proposition that x is equal to y as elements of A.

One can take the propositions-as-types as expressing a deep insight into the nature and meaning of logical operations [34, 48]. But one can just as well view it as a notational convenience which, moreover, allows a proof assistant to treat logical and mathematical operations uniformly. For example, one can take the transitivity of inequality on the natural numbers, leq_trans, to be a term of type

$$\forall x{:}\mathtt{nat},\ y{:}\mathtt{nat},\ z{:}\mathtt{nat},\ x \le y \to y \le z \to x \le z.$$

This last expression, in turn, it a term of type Prop. One can view leq_trans not just as the fact that less-than-or-equal is transitive, but also as a function which, given elements x, y, and z in the natural numbers as well as the facts that x ≤ y and y ≤ z, return the fact that x ≤ z. Thus, given a : nat, b : nat, and c : nat, the term leq_trans a b c denotes the implication a ≤ b → b ≤ c → a ≤ c. Moreover, we can express that H is the fact that a ≤ b by writing H : a ≤ b, in which case leq_trans a b c H denotes the implication b ≤ c → a ≤ c.

The propositions-as-types correspondence is particularly popular as a foundation for constructive mathematics, where assertions are expected to have direct computational significance. Every term in Coq can be viewed as a computational object, subject to evaluation. For example, if π_0 and π_1 denote the two projections from a product type A × B, the a term π_0(a, b) can be "reduced" or "evaluated" to a. In fact, every term in Coq can, at least in principle, be reduced to a canonical normal form. In particular, if t is a closed term of type nat, then t reduces to a numeral. Coq, moreover, makes use of this computational interpretation when checking types. For example, If C(x) is a type that depends on a value x of type A, the system can recognize that C(π_0(a, b)) is the same type as C(a).

The decidability of type checking amounts to the fact that given a term, t, and a type, T, the type-checker can, deterministically, decide whether or not t has type T. This is clearly a useful property to have, though we will see, in Section 6, that it imposes strong restrictions. Under the propositions-as-types correspondence, the decidability of type checking takes on additional significance. Suppose P is a term of type Prop, expressing, for example, Fermat's last theorem. Then a term t of type P is a proof that P is true. Proving Fermat's last theorem thus amounts to

constructing a term of type P, and the decidability of type checking implies that such a term can be recognized, algorithmically, as a valid proof.

4 Type inference in Coq

Now that we have a sense of Coq's axiomatic framework, let us explore some of the mechanisms the system offers to address the challenges raised in Section 2. Generally speaking, type inference is triggered when the system is called on to determine the type of a term, or to check that a term has an appropriate type, when some information has been left implicit. But because dependent types depend on the values of their parameters, inferring a type can entail inferring such values. Recall that in Section 2 we distinguished between two types of information that can be inferred, namely, data and facts. With the propositions-as-types correspondence in place, inferring a fact—such as the fact that multiplication is associative—is a matter of inferring a value of a type P, which is in turn of type Prop, where P expresses the expected associativity property.

We will consider three principal mechanisms. *Implicit arguments* allow users to systematically leave information out of an expression when this information can be inferred from context. *Coercions* allow users to cast, implicitly, objects of one type to objects of another. Finally, *canonical structures* let the user register certain objects as components of a larger structure, providing useful information to the type inference process.[1]

It will be helpful to illustrate these with a running example. The following definition declares a new type, group:

```
Record group : Type := Group
{
  carrier : Type;
  mulg : carrier -> carrier -> carrier;
  oneg : carrier;
  invg : carrier -> carrier;
  mulgA : forall x y z : carrier,
    mulg x (mulg y z) = mulg (mulg x y) z;
  ...
}.
```

[1]For more detail than is provided below, see Coq's online reference manual. All three mechanisms were initially introduced to Coq by Amokrane Saïbi [32, 40, 41], who credits the idea of using implicit arguments in the theorem proving context to Peter Aczel. Implicit arguments were further extended by Hugo Herbelin and Matthieu Sozeau. Canonical structures received little attention until they were revived and used aggressively by Gonthier; see, for example, [17].

According to this type declaration, `group` is a record type, consisting of a carrier, a multiplication, an identity, and an inverse. These are assumed to satisfy the requisite axioms, such as the associativity of multiplication. If `G` has type `group`, that is, `G : group`, then the components of `G` are `carrier G`, `mulg G`, `oneg G`, and so on. Conversely, given elements `my_carrier`, `my_mul`, `my_one` and so on of the right type, the term `Group my_carrier my_mul my_one ...` denotes the corresponding group.

Notice that we are relying on dependent type theory here. The type `group` is a classic example of a dependent sum, since, for example, the type of the second component, `carrier -> carrier -> carrier`, depends on the value `carrier` of the first component. The arguments of the corresponding projections bear the associated dependences. For example, the term `mulg`, which picks out the the second component, has type `forall G : group, carrier G -> carrier G -> carrier G`, a dependent product.

Notice also that the proposition-as-types correspondence is being put to good use. For example, the type of the fifth component, `mulgA`, is the proposition that `mulg` is associative. Assuming `G : group`, the term `mulgA G` has type

```
forall x y z : carrier G,
  mulg G x (mulg G y z) = mulg G (mulg G x y) z
```

which is itself a term of type `Prop`. Thus `mulgA G` denotes the fact that multiplication in `mulg G` is associative, a fact that can be applied to elements of the carrier of `G` just as in the example of `leq_trans` above. In this way, the propositions-as-types correspondence provides a natural and convenient way to think of the group structure as including not only the relevant data—the carrier of the group and group operations—but also the relevant properties.

In a context where we have `G : group`, `g : carrier G`, and `h : carrier G`, the term `mulg G g h` represents the product of `g` and `h` under the multiplication operation of `G`. The implicit arguments mechanism in Coq allows us to write `mulg _ g h`, replacing the first argument by an underscore. Doing so means that we expect the type inference algorithm to infer the value of that argument from context, by finding a solution to the constraints imposed by the fact that the resulting term should be well typed. The algorithm proceeds by instantiating the first element with a variable, `?`. The term `mulg ?` then has type `carrier ? -> carrier ? -> carrier ?`. Since this term is applied to `g : carrier G`, to get the types to work out the system has to solve a simple unification problem, namely, instantiating `?` to unify `carrier ?` with `carrier G`. Thus `?` is instantiated to `G`, and the algorithm has inferred the relevant parameter. With this in mind, one can introduce a new notation:

```
Notation "g * h" := (mulg _ g h).
```

This enables one to write `g * h` for multiplication in any group, allowing the group in question to be inferred from the type of `g`.

In this example, the implicit argument mechanism was used to infer a parameter in the application of a function, `mulg`. But the mechanism can be used just as well to infer parameters during the application of a lemma. For example, recall the transitivity lemma `leq_trans` from the last section. This takes five arguments—three natural numbers, `x`, `y`, `z`, and the facts $x \leq y$ and $y \leq z$—and returns the fact $x \leq z$. Suppose we declare the first three arguments to be implicit. Then given `H1 : a ≤ b` and `H2 : b ≤ c`, the term `leq_trans H1 H2` has type $a \leq c$. Moreover, when we are building a proof interactively in Coq, if we apply `leq_trans H1` to a subgoal $a \leq c$, type inference similarly infers the missing arguments and leaves the us with the goal $b \leq c$.

Coercions are commonly used in programming languages, for example, when adding a real and an integer triggers the coercion of the integer to a real. In the context of mathematical theorem proving, coercions have other uses as well. In our running example, one would ordinarily write `g : carrier G` to specify that `g` is an element of the carrier of `G`. Writing `g : G` instead yields an error, because the system expects something of type `Type` on the right side of the colon, and `G` has type `group`. But declaring

```
Coercion carrier : group >-> Type.
```

informs Coq that the function `carrier` can always be used to coerce a group to a type. If one then enters `g : G`, the algorithm finds itself facing a group on the right side of the colon but expecting a type, and readily inserts the coercion.

The last feature that we will discuss, canonical structures, provides an inverse to coercion, of sorts. In the example above, we used the `carrier` function to coerce a record structure to one of its projections. Canonical structures makes it possible for the type inference algorithm to pass in the other direction, and recognize a particular object as the projection of a larger structure. To illustrate, suppose we define

```
IntGroup := Group int addi zeroi negi addiA ...
```

thereby declaring the integers with addition to be an instance of a group. Somewhat perversely, this will allow us to write `mulg IntGroup i j` instead of `i + j`, when we have `i j : int`. Less perversely, this will allow us to apply general theorems about groups to this particular instance. But what happens now when we write `i * j`? This expression is shorthand for `mulg _ i j`. After instantiating the first argument to a variable, `?`, the type inference algorithm is faced with the unification problem `carrier ? = int`, and gets stuck. Declaring

```
Canonical Structure IntGroup.
```

registers the fact `carrier IntGroup = int` with the system for use in type inference. One can view this as a "hint" to the unification process [2]. Now when the type inference algorithm gets stuck as above, it can appeal to a table of such hints, and use the relevant one to recognize that the integers can be viewed as the carrier of the `IntGroup` structure. The algorithm then replaces `int` by `carrier IntGroup` and solves the unification problem.

The mechanisms just described are not exceedingly complicated, but we will see in the next section that they are remarkably robust with respect to the challenges posed in Section 2. Canonical structures can, moreover, be used in clever ways to trick the type inference algorithm into carrying out various kinds of useful automation [23].

To summarize, type checking is triggered when the user enters an expression or applies a lemma, possibly leaving some arguments and facts implicit. Coq's type inference engine has four resources at its disposal to fill in the remaining information:

1. *unification* can be used to infer implicit arguments;

2. *coercions* can be inserted to resolve a type mismatch;

3. the unification algorithm can refer to a database of *unification hints* to solve unification problems involving a projection of a *canonical structure*; and

4. when all else fails, the algorithm can simplify terms or unfold definitions according to the CIC's computational interpretation of terms, and then retry the previous steps.

Generally speaking, implicit arguments can trigger arbitrary instances of higher-order unification, which is known to be undecidable [14]. So, at best, type inference can only aim to search a reasonable fragment of the space of possible instantiations for an implicit argument. And even within decidable fragments, unpacking definitions and unfolding terms can easily lead to combinatorial explosion. Nonetheless, Coq's type inference algorithm consists, essentially, of iterating the steps above, relying on heuristics to limit the possibilities in the fourth step.

5 The mathematical components library

This section provides a brief indication of some of the ways that the mechanisms for type inference discussed in Section 4 have been used towards Gonthier's formalization of the Feit-Thompson theorem [15], which asserts that finite groups of

odd order are solvable. These examples only scratch the surface; for more detail, see [6, 17, 18, 21, 22].

Recall that Coq's logic is constructive. In contrast, many principles and methods that are commonly used in contemporary mathematics are not constructively valid. For example, constructively, one cannot assume the law of the excluded middle, or prove the existence of an x satisfying a property P by assuming there is no such x and deriving a contradiction. Extensionality fails: one cannot, in general, prove that two functions f and g from A to B are equal by proving that $f(x) = g(x)$ for every x. Choice fails as well: even if one has proved that for every x in A there is a y in B such that some property holds, one cannot assume that there is a function f that picks out such a y for every x.

On the other hand, these properties generally hold in *finite* domains. Since the Feit-Thompson theorem is an extended exploration of properties of finite groups, one would like to take advantage of these features when they are available. Thus, in the Ssreflect library, there are general structures for types with a decidable equality relation (that is, ones where the relation can be computed by a function returning a boolean value of "true" or "false," ensuring that it satisfies the law of the excluded middle); finite structures; and structures that can be equipped with choice functions. For example, one can define a structure for types with decidable equality as follows:

```
Record eqType : Type := EqType
{
  carrier : Type;
  rel : carrier -> carrier -> bool;
  ax : forall x y, (x = y) = (rel x y)
}.
```

In the last line of the record, the expression `rel x y` of type `bool` is coerced to the proposition that the value of this expression is equal to `true`. In other words, `ax` is the proposition that `rel x y` holds if and only if `x = y`. Declaring `carrier` to be a coercion allows one to write `x : T` whenever we have `T : eqType`. Implicit arguments allow one to use the notation `x == y` in place of `rel T x y` whenever `x` and `y` are elements of the carrier of such a `T`. Finally, canonical structures allow one to associate the relevant boolean equality relation with the natural numbers, so that one can write `x == y` when we have `x y : nat`, as well. (This is a slight simplification of the implementation in the Ssreflect library [17].)

Section 2 noted that "big operations" such as $\sum$, $\prod$, $\bigcap$, $\bigcup$, $\bigwedge$, $\bigvee$ can all be viewed as instances of iterations of an associative binary operation. But such operations come in many different flavors: one can sum over a list, a numeric range, or a finite set, and these summations will satisfy different properties depending on whether the underlying structure is a semigroup, an abelian semigroup, or a

ring. Ssreflect comes with an overarching "bigop" library, and once again type inference plays a key role in making it work [6].

Type inference is also used to manage algebraic class inclusions (between rings, commutative rings, fields, and son on) and algebraic constructions: for example, the set of n by n matrices over a ring forms a ring when $n > 0$, and the set of polynomials over a commutative ring again forms a commutative ring. Type inference ensures that the relevant algebraic facts are readily available, and allows a uniform use of notation [17, 20]. Definitions in the Ssreflect library have been carefully chosen so that if `G` and `H` are groups of the same type (more precisely, subgroups of some ambient group type), then the quotient notation `G / H` makes sense; but when `H` is in fact a normal subgroup of `G`, as in the usual construction of a quotient group, `G / H` is a group with all the expected properties [22]. For another example, when a group `G` happens to be abelian, it is often treated as a $\mathbb{Z}$-module and written additively. So, for example, one can write `g *+ n` for scalar multiplication of `g` by `n` whenver `g` is an element of the group and `n` is a natural number. Type inference is used to mediate between these two "views" of an abelian group.

Type inference also helps with mundane mathematical conventions. For example, Section 2 noted the conflation of groups with sets. If `G` and `H` are subgroups of an ambient finite group, and A is a subset of that group, then `G ∩ H` and `C_G(A)` (the centralizer of `A` in `G`) are both groups. But they are also just sets with the ambient group operation; an element `x` is in `G ∩ H` if and only if it is in `G` and `H`, and `x` is in `C_G(A)` if and only if `x` is in `G` and commutes with every element of `A`. Type inference mediates between these two views of a construction—that is, of yielding both a group and a set—allowing one to apply lemmas involving groups in some instances and lemmas involving sets in others. For another example, a homomorphism between groups G and H is a function between G and H equipped (using a record type) with additional properties. Coercion allows one to use ordinary function notation with morphisms, such as `f x` and `f ∘ g`. In the other direction, canonical structures automatically infer the fact that `f ∘ g` is a homomorphism when `f` and `g` are, giving `f ∘ g` a similarly dual status as function and morphism.

Canonical structures can even be used to make sense of mildly abusive mathematical notation. For example, if U and W are subspaces of a vector space V, it is common to write $U + W$ for set $\{u + w \mid u \in U, w \in W\}$. Mathematicians will often say "$U + W$ is a direct sum" when U and W have trivial intersection, ignoring the fact that this is a property of the pair (U, W) which is impossible to read off from the $U + W$ alone. Gonthier has shown, however, that canonical structures provide a convenient way of supporting this abuse of language [20].

6 Limitations and other approaches

The mechanisms supporting type inference that were described in Section 4 are not the only ones available in Coq. In particular, Coq now has a "type class" mechanism [44]. Type classes and canonical structures serve similar purposes, but whereas canonical structures are handled within the type inference loop described at the end of Section 4, the type class mechanism collects constraints that are passed to a separate inference engine at the end of the process. Spitters and van der Weegen [45] have experimented with type classes in the context of mathematical type inference, with positive results.

But one may wish to stray even further from Coq's mindset. Recall some of the key features of that proof assistant:

- An elaborate type theory is built in to the underlying axiomatic framework.
- Using the propositions-as-types correspondence, data and facts are handled in the same way, so theorems can be applied to arguments and hypotheses just as functions are applied to arguments.
- The underlying logic is constructive, and every term has computational significance.
- Type checking makes use of the computational interpretation of terms.
- Type checking is decidable.

These are very strong constraints, which interact with each other in subtle ways and place strong restrictions on the way mathematics is represented and carried out. Not every proof assistant adopts such a framework. In fact, most reject the third, allowing classical reasoning that is ubiquitous in contemporary mathematics. Similarly, the propositions-as-types correspondence is usually linked to constructive theories, though there is no reason that it cannot be adopted in classical frameworks as well.

Although the mechanisms for type inference described in this article scale reasonably well, their use in real mathematical settings can be complex and delicate. Moreover, when an expression fails to typecheck, error messages from the system are often uninformative, and it can be frustrating and difficult to diagnose the problem. There are, moreover, rigid limitations to dependent type theory that stem from the commitment to keep type checking decidable. This is so because type checking algorithms are constrained to focus on syntactic structure, without incorporating background knowledge. For example, if `list A n` denotes the type of vectors of elements of `A` of length `n`, and we have `t : list A (0 + n)`, then,

in Coq, `t` also typechecks as an element of `list A n`. In other words, Coq recognizes these two types as being the same. But entering `t : list A (n + 0)` yields a type error; Coq refuses to recognize that `list A (n + 0)` is the same as `list A n`. What is going on is that addition in Coq is defined by recursion on the first argument, so that the the term `0 + n` reduces to `n` under the computational interpretation. But the fact that `n + 0` is equal to `0` is a *mathematical* fact, and there is no general way to incorporate arbitrary mathematical information in type checking while maintaining decidability.

Still, some have explored ways of making type judgments more flexible while maintaining decidability [1, 7, 46]. An alternative is to give up the decidability of type checking, and accept the fact that some type judgments will require proof from the user. This is the path chosen by NuPrl [10] and PVS [43]. Yet another alternative is to jettison type theory altogether, and move to an axiomatic system like set theory, which offers maximum flexibility while relinquishing all the benefits of types; and then try to recapture some of those benefits by adding an extra layer of automation to register and manage domain information outside the axiomatic theory. Such "soft typing" mechanisms can be found, for example, in Mizar [25].

This article has focused on the modeling of mathematical language from the point of view of contemporary interactive theorem provers. Others [13, 16] have come at the problem from the perspective of natural language processing. In the long run, it seems likely that the various approaches will converge.

Inferring domain information is essential to modeling mathematical language and reasoning. Gonthier's work on the Feit-Thompson theorem shows that it is possible to model full-blown algebraic reasoning in an interactive proof systems. But other approaches should also be explored, and continued experimentation and innovation is needed to develop better support for verifying ordinary mathematical proofs.

References

[1] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In Aaron Stump and Hongwei Xi, editors, *Proceedings of the ACM Workshop Programming Languages meets Program Verification (PLPV) 2007*, pages 57–68. ACM, 2007.

[2] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. Hints in unification. In *Theorem Proving in Higher Order Logics (TPHOLs) 2009*, pages 84–98. Springer, Berlin, 2009.

[3] Jeremy Avigad, Kevin Donnelly, David Gray, and Paul Raff. A formally verified proof of the prime number theorem. *ACM Trans. Comput. Logic*, 9(1):2, 2007.

[4] Henk Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, 1991.

[5] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer, Berlin, 2004.

[6] Yves Bertot, Georges Gonthier, Sidi Ould Biha, and Ioana Pasca. Canonical big operators. In *Theorem Proving in Higher Order Logics (TPHOLs) 2008*, pages 86–101. Springer, Berlin, 2008.

[7] Frédéric Blanqui, Jean-Pierre Jouannaud, and Mitsuhiro Okada. The calculus of algebraic constructions. In *10th International Conference on Rewriting Techniques and Applications (RtA) 1999*, pages 301–316. Springer, Berlin, 1999.

[8] Alonzo Church. A formulation of the simple theory of types. *J. Symbolic Logic*, 5:56–68, 1940.

[9] Gianni Ciolli, Graziano Gentili, and Marco Maggesi. A Certified Proof of the Cartan Fixed Point Theorems. *J. Autom. Reasoning*, 47(3):319–336, 2011.

[10] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[11] Thierry Coquand and Gérard Huet. The calculus of constructions. *Inform. and Comput.*, 76(2-3):95–120, 1988.

[12] Thierry Coquand and Christine Paulin. Inductively defined types. In *International Conference on Computer Logic (COLOG) 1988*, pages 50–66. Springer, Berlin, 1990.

[13] Marcos Cramer, Peter Koepke, and Bernhard Schröder. Parsing and disambiguation of symbolic mathematics in the Naproche system. In Davenport, James H. (ed.) et al., eds., Intelligent computer mathematics. Springer, Berlin, 2011.

[14] Gilles Dowek. Higher-order unification and matching. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 16, pages 1009–1062. Elsevier Science, Amsterdam, 2001.

[15] Walter Feit and John G. Thompson. Solvability of groups of odd order. *Pacific Journal of Mathematics*, 13:775–1029, 1963.

[16] Mohan Ganesalingam. *The Language of Mathematics*. PhD thesis, University of Cambridge, 2009.

[17] François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. Packaging mathematical structures. In *Theorem Proving in Higher Order Logics (TPHOLs) 2009*, pages 327–342. Springer, Berlin, 2009.

[18] Georges Gonthier. Formal proof—the four-color theorem. *Notices Amer. Math. Soc.*, 55(11):1382–1393, 2008.

[19] Georges Gonthier. Advances in the formalization of the odd order theorem. In Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *Interactive Theorem Proving (ITP) 2011*, page 2. Springer, Berlin, 2011.

[20] Georges Gonthier. Point-free, set-free concrete linear algebra. In Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *Interactive Theorem Proving (ITP) 2011*, pages 103–118. Springer, Berlin, 2011.

[21] Georges Gonthier and Assia Mahboubi. An introduction to small scale reflection in Coq. *J. Formaliz. Reason.*, 3(2):95–152, 2010.

[22] Georges Gonthier, Assia Mahboubi, Laurence Rideau, Enrico Tassi, and Laurent Théry. A modular formalisation of finite group theory. In *Theorem Proving in Higher Order Logics (TPHOLs) 2009*, pages 86–101. Springer, Berlin, 2007.

[23] Georges Gonthier, Beta Ziliani, Aleksandar Nanevski, and Derek Dreyer. How to make ad hoc proof automation less ad hoc. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *International Conference on Functional Programming (ICFP) 2011*, pages 163–175. ACM, 2011.

[24] M. J. C. Gordon and T. F. Melham, editors. *Inroduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, 1993.

[25] Adam Grabowski, Artur Korniłowicz, and Adam Naumowicz. Mizar in a nutshell. *J. Formaliz. Reason.*, 3(2):153–245, 2010.

[26] Thomas C. Hales. The Jordan curve theorem, formally and informally. *Amer. Math. Monthly*, 114(10):882–894, 2007.

[27] Thomas C. Hales, John Harrison, Sean McLaughlin, Tobias Nipkow, Steven Obua, and Roland Zumkeller. A revision of the proof of the Kepler conjecture. *Discrete Comput. Geom.*, 44(1):1–34, 2010.

[28] John Harrison. HOL light: a tutorial introduction. In Mandayam Srivas and Albert Camilleri, editors, *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design*, pages 265–269, 1996.

[29] John Harrison. A formalized proof of Dirichlet's theorem on primes in arithmetic progression. *J. Formaliz. Reason.*, 2(1):63–83, 2009.

[30] John Harrison. Formalizing an analytic proof of the prime number theorem. *Journal of Automated Reasoning*, 43:243–261, 2009.

[31] Johannes Hölzl and Armin Heller. Three chapters of measure theory in Isabelle/HOL. In Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *Interactive Theorem Proving (ITP) 2011*, pages 135–151. Springer, Berlin, 2011.

[32] Gérard Huet and Amokrane Saïbi. Constructive category theory. In Gordon Plotkin, Colin P. Stirling, and Mads Tofte, editors, *Proof, language, and interaction: essays in honour of Robin Milner*, pages 235–275. MIT Press, Cambridge, MA, 2000.

[33] Artur Korniłowicz. A proof of the Jordan curve theorem via the Brouwer fixed point theorem. *Mechanized Mathematics and Its Applications*, 6(1):33–40, November 2007.

[34] Per Martin-Löf. An intuitionistic theory of types: predicative part. In H. E. Rose and J. C. Shepherdson eds., *Logic Colloqium '73*, North-Holland, Amsterdam, 1973.

[35] Tarek Mhamdi, Osman Hasan, and Sofiène Tahar. Formalization of entropy measures in hol. In Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *Interactive Theorem Proving (ITP) 2011*, pages 233–248. Springer, Berlin, 2011.

[36] Melvyn B. Nathanson. Desperately seeking mathematical proof. *Notices Amer. Math. Soc.*, 55(7):773, 2008.

[37] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*. Springer, Berlin, 2002.

[38] Russell O'Connor. Essential incompleteness of arithmetic verified by Coq. In *Theorem Proving in Higher Order Logics (TPHOLs) 2005*, pages 245–260. Springer, Berlin, 2005.

[39] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, 2002.

[40] Amokrane Saïbi. Typing algorithm in type theory with inheritance. In *Symposium on Principles of Programming Languages (POPL) '97*, pages 292–301. ACM, 1997.

[41] Amokrane Saïbi. Outils Génériques de modélisation et de démonstration pour la Formalisation des Mathématiques en théorie des Types, Application à la théorie des catégories. Ph.D. thesis, Universiy of Paris 6, 1999.

[42] Natarjan Shankar. *Metamathematics, machines, and Gödel's proof*. Cambridge University Press, Cambridge, 1994.

[43] Natarajan Shankar and Sam Owre. Principles and pragmatics of subtyping in PVS. In D. Bert, C. Choppy, and P. D. Mosses, editors, *Recent Trends in Algebraic Development Techniques (WADT) 1999*, pages 37–52. Springer, Berlin, 2000.

[44] Matthieu Sozeau and Nicolas Oury. First-class type classes. In *Theorem proving in higher order logics (TPHOLs) 2008*, pages 278–293. Springer, Berlin, 2008.

[45] Bas Spitters and Eelis van der Weegen. Type classes for mathematics in type theory. *Mathematical Structures in Computer Science*, 21(4):795–825, 2011.

[46] Pierre-Yves Strub. Coq modulo theory. In Anuj Dawar and Helmut Veith, editors, *19th EACSL Annual Conference on Computer Science Logic*, pages 549–543. Springer, Berlin, 2010.

[47] A. S. Troelstra and Dirk van Dalen. *Constructivism in Mathematics*, volume 2. North-Holland, Amsterdam, 1988.

[48] William W. Tait. Truth and proof: the Platonism of mathematics. *Synthese*, 69:341–370, 1986. Reproduced in W. D. Hart, editor, *The philosophy of mathematics*, Oxford University Press, Oxford, 1996, pages 142–167.

[49] Benjamin Werner. Sets in types, types in sets. In *Theoretical aspects of computer software*, pages 530–546. Springer, Berlin, 1997.