
Journal of Automated Reasoning **33:** 29–49, 2004.

Organizing Numerical Theories Using Axiomatic Type Classes

LAWRENCE C. PAULSON
Computer Laboratory, University of Cambridge, J.J. Thompson Avenue, Cambridge CB3 0FD, England. e-mail: lcp@cl.cam.ac.uk

(Received: 19 March 2004; accepted: 16 July 2004)

Abstract. Mathematical reasoning may involve several arithmetic types, including those of the natural, integer, rational, real, and complex numbers. These types satisfy many of the same algebraic laws. These laws need to be made available to users, uniformly and preferably without repetition, but with due account for the peculiarities of each type. Subtyping, where a type inherits properties from a supertype, can eliminate repetition only for a fixed type hierarchy set up in advance by implementors. The approach recently adopted for Isabelle uses *axiomatic type classes*, an established approach to overloading. Abstractions such as semirings, rings, fields, and their ordered counterparts are defined, and theorems are proved algebraically. Types that meet the abstractions inherit the appropriate theorems.

Key words: arithmetic, axiomatic type classes, Isabelle, overloading, polymorphism.

1. Introduction

Theorem-proving tools typically deal with many different types of numbers. The natural numbers are appropriate for foundational proofs relying on induction. The integers and the rational numbers can be appropriate for modeling computer arithmetic. The real and complex numbers are appropriate for developing a theory of mathematical analysis. In the formalization of mathematics, it is common to see several different numeric types involved in a single theorem statement.

There are many different arithmetic laws. Addition and multiplication are typically commutative and associative. Addition distributes over multiplication. Addition or multiplication by a common value can be cancelled under certain circumstances. Other laws relate to monotonicity and numerical signs. Still others concern subtraction, division, absolute value, and so forth. A further complication is that a mathematical law, when formalized in a theorem prover, often takes multiple forms. A mathematician will apply a law such as $k \times (m + n) = k \times m + k \times n$ to a term of the form $(m + n) \times k$. Corollaries of a theorem – typically special cases where certain variables are replaced by 0 or 1 – give rise to additional theorems. The dozen or so laws in a textbook multiply into hundreds of lemmas in a theorem prover.

With several numerical types and hundreds of arithmetic lemmas, we face a potential explosion: taking the obvious product will yield thousands of laws. The standard method of avoiding this problem is subtyping. An example of subtyping is to formalize the natural numbers as a subset of the reals, so that it inherits its laws from those of the reals. Consider a slightly more complicated type hierarchy with the natural numbers, the integers, the rationals, the reals, and the complex numbers:

$$\mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{Q} \subseteq \mathbb{R} \subseteq \mathbb{C}.$$

Subtyping in general is natural and useful, but using it to organize the arithmetic types in a hierarchy has serious problems.

- It is inelegant, inverting the order of mathematical construction. Starting with the natural numbers, we construct the integers, the rationals, the reals, and finally the complex numbers. The properties of the complex numbers are rightfully derived from those of the natural numbers, not the other way around.
- It is inflexible. Only implementors can define the linear hierarchy to avoid the repetition of arithmetic facts. If we developed a theory of complex polynomials, then we would have to rewrite the standard prelude to make type `complex` a subtype of type `complexpoly`. Of course, nobody would want to regard the complex numbers as a subset of the complex polynomials.
- It is confusing. Users expect the largest type to support the most operations. Type `complex` includes all the other types as subsets, but it does not possess an ordering. Properties involving orderings or signs, such as $-a < -b \iff b < a$, have to be inherited from type `real`, while other properties are inherited from type `complex`.

PVS uses subtyping to let theorems about the reals be reused for the rationals, integers, and naturals, but it does not provide the complex numbers at all [11]. PVS provides an abstract type, called `number_field`, which lies at the top of the arithmetic type hierarchy. This type satisfies the axioms of a field, but not of an ordered field. This allows development of, for example, complex numbers or nonstandard reals as subtypes.

Abstract algebra offers a better approach to formalizing arithmetic types. Roughly speaking, a *field* is a set equipped with the operations of addition, subtraction, multiplication, and division, satisfying all the familiar laws. The complex numbers are a field. An *ordered field* is a field equipped with a linear ordering that is preserved in the familiar way by addition and multiplication. The real and the rational numbers are both ordered fields. A *ring* is similar to a field but does not necessarily have division. Precise definitions of these concepts appear below. Each one requires surprisingly few axioms, and they build on one another naturally, yielding a lattice of concepts. Given such concepts, we can prove the familiar arithmetic laws starting from a minimal set of axioms. Laws that hold for rings will continue to hold for ordered rings and for fields. If we then define a type of complex polynomials and prove that it forms a ring, it immediately inherits all the properties proved for rings.

Most logical formalisms can easily express such an axiomatic development. Isabelle's *axiomatic type classes* are particularly convenient [7, 13]. They provide controlled polymorphism over a spectrum that ranges from traditional ML-style polymorphism to overloading. Concepts such as ring, ordered ring, and field can be formalized as axiomatic type classes. A type can be proved to belong to a type class, gaining access to the theorems proved for that class. At the same time, definitions of operators can be specific to each type: addition has separate definitions for the natural numbers, integers, and rationals. Although the addition operator stands for a different function for each arithmetic type, the abstract properties of addition are inherited according to the axiomatic hierarchy.

This paper can be seen on two levels: first, as a suggestion for organizing numerical theories in general, and second, as an example of the use of axiomatic type classes in Isabelle. The paper introduces Isabelle and axiomatic type classes (Section 2). It then describes the axiomatic type classes for arithmetic (Section 3). It shows how axiomatic type classes support numeric literals (Section 4). It presents a small example – the natural numbers extended with ∞ (Section 5) – and briefly comments on the usefulness of axiomatic type classes (Section 6).

2. Isabelle and Axiomatic Type Classes

Isabelle [8] is an interactive theorem prover implementing a higher-order logical framework. Isabelle supports a variety of formalisms; the most popular is higher-order logic (Isabelle/HOL). Many large case studies have been done using Isabelle concerning, for example, cryptographic protocols [10] and the Java Virtual Machine [6]. For the purposes of this paper, the most interesting aspect of Isabelle is its type system: order-sorted polymorphism [7].

Polymorphism is found in a number of functional programming languages, such as ML [9]. For example, the type of lists takes as an argument the type of the list elements. The list reversal function is *polymorphic*: it reverses any list without regard for the type of the list elements. ML assigns `rev` the following type:

```
rev : ('a list) -> ('a list)
```

The symbol `'a` is a *type variable*, which ranges over all types. Here, we see that `rev` returns a list of the same type as it is given. Moreover, it executes the same algorithm (in practice, the same code) without regard for the type of the list elements. Polymorphism therefore differs from *overloading*, where a symbol such as + represents the concept of addition for both integers and reals but with separate addition algorithms for the two types.

Polymorphism is as useful in logic as it is in programming. All versions of the theorem prover HOL [4] use it, as does Isabelle. The formally defined list reversal function is polymorphic, and so are the theorems proved about this function. For example, the theorem `rev (rev l) = l` is true regardless of the type of the list `l`. As in ML, polymorphism involves type variables: the variable `l` has type `'a`. The

rewriter can apply this theorem to simplify a term such as `rev (rev [True])`, automatically matching `'a` to the type `bool` and `l` to the term `[True]`.

In traditional polymorphism, a type variable ranges over all possible types. Isabelle requires a more refined treatment of polymorphism because certain types are part of the logical framework and are not available to users. This treatment is based on Wadler and Blott's *type classes* [12], which they introduced to support overloading in the programming language Haskell [5]. Nipkow [7] transferred their type system – a clean combination of overloading and polymorphism – into the context of theorem proving. Later, Wenzel [13] worked out the logical foundations of axiomatic type classes; he also produced an elegant implementation.

A *type class* denotes a collection of types; a *sort* is a list of type classes and denotes their intersection. Each type variable has a sort and can be instantiated by any type that belongs to all of the listed type classes. For example, Isabelle/HOL introduces the type class `zero` as follows.

axclass `zero ⊆ type`

Then, Isabelle/HOL introduces the constant `0`. (In Isabelle, the symbol `::` means "has type" as well as "has sort.")

consts `"0" :: "'a::zero"`

This declaration makes `0` polymorphic but only over the type class `zero`. The two declarations together define `zero` to be the class of all types that the constant `0` may have. Next, we can declare a particular type, such as `nat`, to be a member of class `zero`.

instance `nat :: zero ..`

This declaration informs Isabelle that we intend to use `0` as a constant of type `nat`. Strictly speaking, `0` is not a single constant but a family of constants $0::\tau$ for each type τ. These constants can be defined independently of one another.

Isabelle/HOL declares several other constants to be polymorphic over dedicated type classes. The type class `one` comprises the types that have the constant `1`; the type classes `plus` and `times` comprise the types that have the infix operators `+` and `*`, respectively. Here are the definitions of the type class `plus` and the infix operator `+`.

axclass `plus ⊆ type`
consts `"+" :: "['a::plus, 'a] => 'a"` (**infixl** `65`)

The definitions of the classes `zero` and `times` are similar to those presented above. Now, the sort `{zero,one,plus,times}` denotes the intersection of the four named type classes, comprising the types that possess all four of these constants. These will include the standard arithmetic types, but they may include other types: we have not yet specified the properties of these constants.

An *axiomatic type class* is a type class augmented with axioms constraining the constants. (Wadler and Blott [12] foresaw this possibility.) We can refer to these axioms in proofs, obtaining theorems specific to the type class. To show that a type τ is an **instance** of a particular axiomatic type class, we must prove that all the axioms hold. Typically the axioms refer to overloaded constants, which we define for type τ with the objective of satisfying the axioms. Verifying the axioms for type τ makes all the theorems proved for the type class immediately available for type τ. No further importation or instantiation is necessary.

The implementation burden for axiomatic type classes falls mainly on type inference. Type checking determines whether a polymorphic theorem may be instantiated with a specific type to draw conclusions about specific terms. Isabelle combines theorems using higher-order unification; type classes complicate the procedure by requiring a treatment of order-sorted type variables. (Before the advent of type classes, higher-order unification was monomorphic.) Nipkow [7] describes the modifications. Wenzel [13] proves the soundness of axiomatic type classes through a translation into pure higher-order logic, but his proof does not reflect the implementation. As just stated, the mechanism for ensuring that rules are applied correctly is type checking.

Isabelle's axiomatic type classes provide excellent support for overloading, but they are not a general abstraction mechanism. Any attempt to use them as such will run into a number of difficulties:

- Names of constants are required to agree. A specification of monoids that refers to `0` and `+` cannot be used to constrain `1` and `*`.
- A type can be an **instance** of a class in at most one way. This is inconvenient when there are multiple possibilities, such as whether $\leq$ on lists should denote the prefix ordering or lexicographic ordering. The same restriction rules out duality arguments such as reversing the direction of a partial ordering.

Algebraic concepts such as rings and fields are easily defined as axiomatic type classes. Unfortunately, this does not yield a useful environment for developing ring theory. The carrier of a ring could only be a type, when realistic examples frequently require the carrier to be an arbitrary set. Type instance declarations cannot appear in the middle of a proof, nor can they be rescinded. Type classes for rings and fields are useful, not for formalizing abstract algebra, but for organizing libraries of theorems about arithmetic types. The type classes for rings and fields are therefore oriented toward organizing arithmetic types and do not convey all the detail found in abstract algebra. With the definitions given below, a ring is always commutative and has a multiplicative unit (that is, 1).

Since this paper was first written, Steven Obua has greatly expanded the type class hierarchy. He has introduced standard terminology – for example, a `ring` need not be commutative – and added many new concepts. These improvements make the hierarchy more general, and Obua has used it to formalize matrices. This paper presents the original hierarchy in order to avoid complicating the presentation.

3. Axiomatic Type Classes for Arithmetic

This section presents the definitions of the axiomatic type classes used (in Isabelle 2004) to organize the arithmetical theories. They form a hierarchy as new operators are introduced, such as subtraction, division, and orderings, and as new axioms are introduced. Because classes can always be combined to form a sort, the hierarchy is a lattice, with ordered fields at the top and semirings at the bottom.

This hierarchy rests upon existing type classes for orderings. These appear below, in simplified form. Type class `ord` comprises all types for which the infix relations $<$ and $\leq$ are defined.*

```
axclass ord ⊆ type
consts  "<" :: "['a:: ord, 'a]  => bool"  (infixl 50)
consts  "≤" :: "['a:: ord, 'a]  => bool"  (infixl 50)
```

Type class `order` comprises the partial orderings, which are reflexive, transitive, and antisymmetric. This type class also specifies the relationship between $<$ and $\leq$.

```
axclass order ⊆ ord
  order_refl:    "x ≤ x"
  order_trans:   "x ≤ y ⇒ y ≤ z ⇒ x ≤ z"
  order_antisym: "x ≤ y ⇒ y ≤ x ⇒ x = y"
  order_less_le: "(x < y) = (x ≤ y ∧ x ≠ y)"
```

Type class `linorder` further constrains the ordering to be linear. Of course, the numeric types (other than `complex`) are linearly ordered.

```
axclass linorder ⊆ order
  linorder_linear: "x ≤ y ∨ y ≤ x"
```

3.1. THE BASIC TYPE CLASS HIERARCHY

A *semiring* (for our purposes) is an algebraic structure with the constants 0 and 1 and the operators + and $\times$. The binary operators are commutative and associative, and they satisfy the usual distributive law. Moreover, addition can be cancelled from the left.

```
axclass semiring ⊆ zero, one, plus, times
  add_assoc:      "(a + b) + c = a + (b + c)"
  add_commute:    "a + b = b + a"
  add_0:          "0 + a = a"
  add_left_imp_eq: "a + b = a + c ⇒ b=c"
```

* To avoid a multiplicity of trivial variants of theorems, Isabelle does not define $>$ and $\geq$.

```
  mult_assoc:     "(a * b) * c = a * (b * c)"
  mult_commute:   "a * b = b * a"
  mult_1:         "1 * a = a"

  left_distrib:   "(a + b) * c = a * c + b * c"
  zero_neq_one:   "0 ≠ 1"
```

The first line introduces the class `semiring` and places it in the sort hierarchy: it is a subclass of `zero`, `one`, `plus`, and `times` and, therefore, of their intersection. The class is further constrained by axioms.

A *ring* extends a semiring with unary minus and subtraction, which are related to addition in the obvious way.

```
axclass ring ⊆ semiring, minus
  left_minus: "- a + a = 0"
  diff_minus: "a - b = a + (-b)"
```

A *field* extends a ring with a multiplicative inverse (reciprocal) and division, which are related to multiplication in the obvious way.

```
axclass field ⊆ ring, inverse
  left_inverse:    "a ≠ 0 ⇒ inverse a * a = 1"
  divide_inverse: "a / b = a * inverse b"
```

Next come ordered versions of these algebraic structures. The basic relation symbol is $\leq$, but the strict "less than" relation ($<$) is also specified. Now, we can define an *ordered semiring* to be a semiring that is also a linear ordering; it satisfies three further axioms asserting that $0 < 1$ and that addition and multiplication preserve the ordering.

```
axclass ordered_semiring ⊆ semiring, linorder
  zero_less_one: "0 < 1"
  add_left_mono:          "a ≤ b ⇒ c + a ≤ c + b"
  mult_strict_left_mono: "a < b ⇒ 0 < c ⇒ c * a < c * b"
```

An *ordered ring* is both an ordered semiring and a ring. It must also have the absolute value function (`abs`) defined according to the axiom below. Note that `abs a` may also be written `|a|`.

```
axclass ordered_ring ⊆ ordered_semiring, ring
  abs_if: "|a| = (if a < 0 then -a else a)"
```

An *ordered field* is any field that is also an ordered ring.

```
axclass ordered_field ⊆ ordered_ring, field
```

An alternative treatment of absolute value would be to define the function `abs` once and for all, using the equation shown above. The definition could be made before types such as `nat` and `int` had been declared and before constants such as `0` and `<` possessed any definitions. Theorems such as `"|a+b| ≤ |a| + |b|"` could then be proved abstractly, from type class axioms. This approach would be fine for numeric types. Unfortunately, it presupposes a linear ordering, which is not compatible with other uses of absolute value in Isabelle. Constraining the function `abs` by a type class axiom, as done here, is more flexible.

3.2. AVOIDING REDUNDANT AXIOMS BY TYPE CLASS INCLUSIONS

We can refine the definitions given above. One of the semiring axioms becomes redundant when we move to rings: the cancellation of addition can be proved once we have subtraction. Also, the axiom $0 < 1$ of ordered semirings is no longer necessary for ordered rings: it can be proved from the axiom $0 \neq 1$ and the sign laws of multiplication. Redundant axioms are inelegant, and if they accumulate, they make **instance** declarations needlessly long. The type class system allows us to avoid introducing axioms that will become redundant. The method is best demonstrated by an example. We abandon the declarations given above and instead declare a type class `almost_semiring`, which includes all the axioms except cancellation of addition.

```
axclass almost_semiring ⊆ zero, one, plus, times
  add_assoc:      "(a + b) + c = a + (b + c)"
  add_commute:    "a + b = b + a"
  add_0:          "0 + a = a"

  mult_assoc:     "(a * b) * c = a * (b * c)"
  mult_commute:   "a * b = b * a"
  mult_1:         "1 * a = a"

  left_distrib:   "(a + b) * c = a * c + b * c"
  zero_neq_one:   "0 ≠ 1"
```

We extend this type class in two different ways. By introducing subtraction, we obtain rings, with no redundant axiom about cancellation of addition.

```
axclass ring ⊆ almost_semiring, minus
  left_minus: "- a + a = 0"
  diff_minus: "a - b = a + (-b)"
```

By assuming the axiom for cancellation of addition, we obtain semirings.

```
axclass semiring ⊆ almost_semiring
  add_left_imp_eq: "a + b = a + c ⇒ b=c"
```

To complete the development, we must prove that all rings are semirings, since it is no longer true by construction. We do so as follows.

```
instance ring ⊆ semiring
proof
  fix a b c :: 'a
  assume "a + b = a + c"
  hence  "-a + a + b = -a + a + c"
    by (simp only: add_assoc)
  thus "b = c" by simp
qed
```

Isabelle demands a proof of the axiom for cancellation of addition. After this is provided, Isabelle will regard all types that belong to class `ring` to be elements of class `semiring`. Proofs of claims are given through the keyword **by**. After this keyword comes a command, such as an invocation of the simplifier (`simp`), which here refers to the axiom `add_assoc`.

The redundant axiom $0 < 1$ is handled similarly. We declare a new type class, `almost_ordered_semiring`.

```
axclass almost_ordered_semiring ⊆ semiring, linorder
  add_left_mono:         "a ≤ b ⇒ c + a ≤ c + b"
  mult_strict_left_mono: "a < b ⇒ 0 < c ⇒ c * a < c * b"
```

Note that it extends `semiring` rather than `almost_semiring`. Its purpose is to ensure that $0 < 1$ is assumed for `ordered_semiring` but not for `ordered_ring`. Then, we prove that every ordered ring is an ordered semiring by showing that it satisfies $0 < 1$.

```
instance ordered_ring ⊆ ordered_semiring
proof
  have "(0::'a) ≤ 1*1"  by (rule zero_le_square)
  thus "(0::'a) < 1"    by (simp add: order_le_less)
qed
```

Here, the command **by** `(rule zero_le_square)` is a reference to a previously proved theorem about ordered rings. This proof of $0 < 1$ uses $0 \leq 1 \times 1$ as a lemma.

Fields and ordered fields are then defined as before. Figure 1 is a diagram of the hierarchy of type classes. The solid lines show inclusions that hold by construction, while the dashed lines show inclusions that have been proved.

3.3. REASONING WITH TYPE CLASS AXIOMS

Now that we have defined some axiomatic type classes, let us use them. Any proof may refer to the axioms of a type class. Such axioms are entirely different from

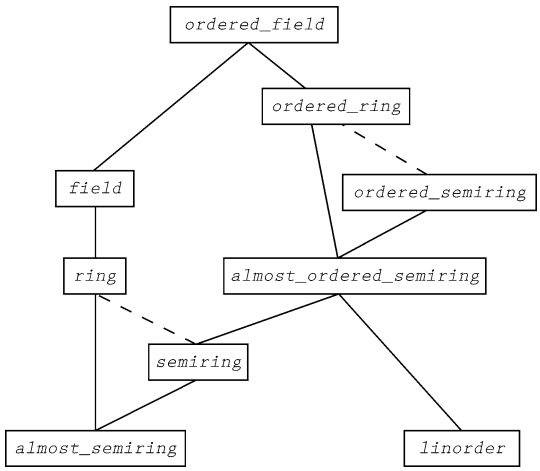

Figure 1. Type classes for organizing arithmetic theories.

axioms asserted at top level: the latter must be taken on faith, while the former will be proved later in **instance** declarations. If the axioms of a type class are inconsistent, then no type can be an **instance** of it, but no other harm is done. Isabelle keeps track of theorems whose proofs depend on type classes that are not known to contain any types. Isabelle prints a warning when such theorems are proved and does not permit their use in justifying other **instance** declarations.

Here is a trivial example of a proof. The type class `almost_semiring` has an axiom that says that 0 is the left identity of addition. By the commutative law, it is also a right identity. Here is the full proof text.

```
lemma add_0_right: "a + 0 = (a::'a::almost_semiring)"
proof -
  have "a + 0 = 0 + a"  by (simp only: add_commute)
  also have "... = a"    by simp
  finally show ?thesis .
qed
```

In the top line is a reference to a type variable, namely, `'a`, which is given an explicit class, namely, `almost_semiring`. All the type classes that the proof requires must be mentioned in this way; a type variable normally does not belong to any special type classes, preventing the use of type class axioms. The given theorem statement allows the use of the axioms of `almost_semiring`. The resulting theorem is specific to this class and to its descendants, such as `ring` and `ordered_field`. Inclusion for type classes superficially resembles subtyping. However, it is a deeper concept that reflects mathematical reasoning in its full generality rather than just the subset relation.

The full theory of rings, fields, etc., comprises approximately 250 theorems. They are general laws concerning 0, 1, $+$, $-$, $\times$, $/$, $\leq$, and $<$. To maximize general-

ity, each law refers to the weakest type classes possible. For example, any theorem involving subtraction requires at least a ring; but if it also involves the ordering, then it requires an ordered ring. Very occasionally, a proof requires more axioms than can be seen from the theorem statement alone. Here is an example, the law $a \times b = 0 \iff a = 0 \vee b = 0$.

```
lemma mult_eq_0_iff:
        "(a*b = (0::'a::ordered_ring)) = (a=0 ∨ b=0)"
```

The proof, which is omitted, appeals to the ordering. The factors are positive, negative, or zero; the nonzero cases contradict the monotonicity of multiplication. This property of multiplication fails in the ring of integers modulo 4, where $2 \times 2 = 0$; it is a theorem only in ordered rings, even though the theorem statement does not refer to an ordering. This property of multiplication also holds for fields, when the availability of division eliminates the need for an ordering. Formally, these are two different theorems.

```
lemma field_mult_eq_0_iff: "(a*b = (0::'a::field)) = (a=0 ∨ b=0)"
```

3.4. INSTANTIATING THE TYPE CLASSES TO SPECIFIC TYPES

Now, let us apply the definitions of type classes. Suppose that we have already defined the type of natural numbers (`nat`) and the relevant operators (0, 1, addition, multiplication) and have proved their basic properties. Then we may make the following declaration.

```
instance nat :: semiring
proof
  fix i j k :: nat
  show "(i+j)+k = i+(j+k)"
    by (rule nat_add_assoc)
  show "i+j = j+i"
    by (rule nat_add_commute)
  show "0+i = i"
    by simp
  show "(i*j)*k = i*(j*k)"
    by (rule nat_mult_assoc)
  show "i*j = j*i"
    by (rule nat_mult_commute)
  show "1*i = i"
    by simp
  show "(i+j) * k = i*k + j*k"
    by (simp add: add_mult_distrib)
  show "0 ≠ (1::nat)"
```

```
    by simp
  assume "k+i = k+j" thus "i=j" by simp
qed
```

Here we assert that type `nat` belongs to the type class `semiring`. Isabelle checks what it already knows about this type and then asks us to prove all the axioms given in the definition of classes `almost_semiring` and `semiring`. In this example, the axioms are verified by reference to theorems already been proved for the natural numbers, such as `nat_add_assoc`. We could instead have proved each axiom from first principles. Such details are not important. The essential points are that Isabelle keeps track of (1) which axioms must be proved and (2) which theorems now become available for type `nat`. The first of the two theorems proved above, `add_0_right`, becomes available because it holds for all semirings. The second, `mult_eq_0_iff`, does not: it requires an ordered ring. This property of multiplication can still be proved from definitions specific to type `nat`.

Now, we can prove that the natural numbers are an *ordered* semiring. This step requires proving that type `nat` is linearly ordered (for a suitable definition of $\leq$) and that addition and multiplication to respect this ordering. If we have already proved that the type belongs to classes `semiring` and `linorder`, then only three additional axioms will require proofs.

```
instance nat :: ordered_semiring
proof
  fix i j k :: nat
  show "0 < (1::nat)"
    by simp
  show "i ≤ j ⇒ k+i ≤ k+j"
    by simp
  show "i<j ⇒ 0<k ⇒ k*i < k*j"
    by (simp add: mult_less_mono2)
qed
```

The treatment of other types is similar. Type `int` is shown to be a ring as follows.

```
instance int :: ring
proof
  fix i j k :: int
  show "(i+j)+k = i+(j+k)"
    by (simp add: zadd_assoc)
  show "i+j = j+i"
    by (simp add: zadd_commute)
  show "0+i = i"
    by simp
  show "- i + i = 0"
    by simp
```

```
  show "i - j = i + (-j)"
    by (simp add: zdiff_def)
  show "(i*j)*k = i*(j*k)"
    by (rule zmult_assoc)
  show "i*j = j*i"
    by (rule zmult_commute)
  show "1*i = i"
    by simp
  show "(i+j) * k = i*k + j*k"
    by (simp add: int_distrib)
  show "0 ≠ (1::int)"
    by (simp only: Zero_int_def
                   One_int_def One_nat_def int_int_eq)
qed
```

Here Isabelle requires proofs of the axioms for the type classes `almost_semiring`, `semiring`, and `ring`. We could have split this **instance** declaration into smaller parts, but there is no reason to do so. It is good to separate the declarations for the type classes `ring` and `ordered_ring`: the theorems for `ring` can be used to help develop the theory of $\leq$ for integers.

Our concept of ordered ring requires a definition of the absolute value function; recall the discussion at the end of Section 3.1. Isabelle specifies the function `abs` to be overloaded for all types that admit unary minus. (The type class `minus` governs the constants for unary minus, subtraction, and absolute value. We could have defined separate type classes for each of these constants.) Now, as specified by the axiom given in the definition of an ordered ring, we must define the absolute value function for the integers.

```
zabs_def:  "|i::int| == if i < 0 then -i else i"
```

Given this definition, we can show type `int` to be an ordered ring.

```
instance int :: ordered_ring
proof
  fix i j k :: int
  show "i ≤ j ⇒ k+i ≤ k+j"
    by (rule zadd_left_mono)
  show "i<j ⇒ 0< ⇒ k*i < k*j"
    by (rule zmult_zless_mono2)
  show "|i| = (if i<0 then -i else i)"
    by (simp only: zabs_def)
qed
```

The proof of the absolute value axiom merely checks that it has been defined in the specified manner.

3.5. A TYPE CLASS FOR POWERS

All arithmetic types allow a number to be raised to a nonnegative power. This exponentiation operation is meaningful for any type that possesses the number 1 and multiplication and, therefore, for any semiring. We could include exponentiation in the definition of a semiring, just as `abs` is included in the definition of an ordered ring. However, it is more modular to introduce many simple type classes rather than a few complicated ones. We can always combine type classes to form sorts.

```
axclass recpower ⊆ semiring, power
  power_0:   "a ^ 0       = 1"
  power_Suc: "a ^ (Suc n) = a * (a ^ n)"
```

In this declaration, the type class `power` simply denotes the class of all types for which the power function may be used, without constraining its definition. Nonarithmetic types may use the power operator differently: to denote the n-fold composition of a relation, for example. Because relational composition does not satisfy basic algebraic properties such as those of a semiring, no single polymorphic definition of powers can suffice: this operator must be overloaded. The new class, `recpower`, extends the classes `semiring` and `power` with two axioms representing the primitive recursive definition of exponentiation. Many well-known results can be proved in this abstract setting.

```
lemma power_one:       "1^n = (1::'a::recpower)"
lemma power_one_right: "(a::'a::recpower) ^ 1 = a"
lemma power_add:    "(a::'a::recpower) ^ (m+n) = (a^m) * (a^n)"
lemma power_mult:   "(a::'a::recpower) ^ (m*n) = (a^m) ^ n"
lemma power_mult_distrib:
                  "((a::'a::recpower)*b) ^n = (a^n)*(b^n)"
```

To instantiate this type class, we need only make an equivalent definition for the type of interest. Let us define exponentiation for type `nat`:

```
primrec (power)
  "p ^ 0 = 1"
  "p ^ (Suc n) = (p::nat) * (p ^ n)"
```

Now, we can demonstrate that `nat` belongs to class `recpower`.

```
instance nat :: recpower
proof
  fix z n :: nat
  show "z^0 = 1"                by simp
  show "z^(Suc n) = z * (z^n)"  by simp
qed
```

Lemmas such as power_add now become available for type nat.

Another benefit of the type class approach is uniform simplification. Any lemmas can be declared as default simplification rules, which makes them automatically available to all simplifier invocations. Isabelle/HOL declares many general arithmetic lemmas, such as power_one_right and mult_eq_0_iff, as default simplification rules. Rewrites such as $a \times b = 0 \Leftrightarrow a = 0 \vee b = 0$ and $a^1 = a$ will be performed for all types in the appropriate type classes, ensuring that simplification behaves similarly for all the arithmetic types.

4. The Treatment of Numeric Literals

Some proofs require the use of literal constants such as 1024, which obviously must not be expanded to unary notation. In Isabelle, literal constants abbreviate terms of a data structure that corresponds to two's complement binary arithmetic. Operations such as addition, subtraction, multiplication, division, and comparisons are performed by rewriting on this data structure. Important, all this takes place within the logic: these computations are not hard-wired into Isabelle. They are reasonably efficient: the computation $1359 \times -2468 = -3354012$ takes only a tenth of a second.

Bit strings are defined to be a recursive datatype:

```
datatype
  bin = Pls
      | Min
      | Bit bin bool    (infixl "BIT" 90)
```

The constructor pls denotes the value 0, while the constructor min denotes the value -1; these both terminate the recursion. The third, infix, constructor extends a binary number with the least significant bit, coded as True $= 1$ and False $= 0$. Addition, negation, and multiplication can be defined straightforwardly in terms of this data structure. The two's complement representation avoids ugly case analyses on the signs of operands.

The type class number denotes all types for which numerals can be defined. The overloaded function number_of maps bit strings to values of some particular type of that class.

```
axclass  number ⊆ type
consts   number_of :: "bin ⇒ 'a::number"
```

These definitions let us make separate definitions of number_of for each numeric type. (If type nat did not require special treatment, then one polymorphic definition of number_of would have worked for all types.) Each definition requires a separate proof that the binary arithmetic operations – defined by primitive recursion on bit strings – agree with numerical arithmetic. Once again, type classes allow us to avoid repeating the proofs for all the arithmetic types. A new type class specifies

just one function, number_of. All of the agreement proofs refer to the definition of this valuation function.

Because the binary representation includes negative numbers, we cannot use semirings. The type class for binary arithmetic is based on rings:

```
axclass number_ring ⊆ number, ring
  number_of_Pls: "number_of bin.Pls = 0"
  number_of_Min: "number_of bin.Min = - 1"
  number_of_BIT: "number_of(w BIT x) = (if x then 1 else 0) +
                                       number_of w + number_of w"
```

Now, in order to install literal constants for any type that is at least a ring, it is necessary only to define the function number_of using exactly the recursion specified above. It remains to set up literal constants for the natural numbers, which form only a semiring. We cannot prove that type nat belongs to class number_ring – which is why it requires special treatment – but we can enter the type into the trivial class type number.

```
instance nat :: number ..
```

This declaration allows us to define the constant number_of for type nat. To evaluate a bit string as a natural number, we begin by evaluating it as an integer. Then, we apply the coercion function nat, which maps negative integers to zero and the others to the corresponding natural numbers. We arrive at the following definition:

```
nat_number_of_def:
   "(number_of :: bin ⇒ nat) v ==
    nat ((number_of :: bin ⇒ int) v)"
```

Theorems expressing agreement between the binary arithmetic operations and true natural number arithmetic are proved by case analysis on the signs of the underlying integers. Notice that number_of for type nat is defined in terms of number_of for type int. This situation is common with overloading. For instance, the complex number zero is defined in terms of the real number zero.

The final step in the implementation of new numeric type is to install Isabelle's decision procedure for linear arithmetic. This procedure works by deduction within the logic, just as constant arithmetic does, using theorems to simplify terms and to deduce contradictions involving inequalities. Using axiomatic type classes, nearly all of the necessary theorems must be proved only once. This minimizes the amount of ML code needed to set up the procedure for a new arithmetic type.

5. Example: The Natural Numbers with Infinity

Let us extend the type of natural numbers with a new element, infinity. The natural numbers are extended with ∞ by obvious equations such as $\infty + n = \infty \times n = \infty$.

This type is used in the HOL system for reasoning about the lengths of finite and infinite lists. Michael Gordon suggested this example, adding, "It is an overloading nightmare in HOL, with lots of theorems covering the various cases when arguments are finite and infinite."

Let us define this type in Isabelle and attempt to civilize its theory using type classes. We declare type `natinf` to consist of natural numbers of the form `Finite` *n* and the constant `Infinity`.

```
datatype natinf = Finite nat | Infinity
```

We declare the type to belong to the trivial classes `zero`, `one`, `plus` and `times`.

```
instance natinf :: "{zero,one,plus,times}" ..
```

The definitions of zero and one refer to the corresponding natural numbers.

```
zero_def: "0 == Finite 0"
 one_def: "1 == Finite 1"
```

The definitions of addition and multiplication are by case analysis on whether the operands are finite or not.

```
add_def:
  "M + N == (case M of Finite m ⇒
                       (case N of Finite n ⇒ Finite (m+n)
                                | Infinity ⇒ Infinity)
                     | Infinity ⇒ Infinity)"
 mult_def:
  "M * N == (case M of Finite m ⇒
                       (case N of Finite n ⇒ Finite (m*n)
                                | Infinity ⇒ Infinity)
                     | Infinity ⇒ Infinity)"
```

Type `natinf` is an "almost semiring." The required properties have trivial proofs by simplification with automatic case splitting.

```
instance natinf :: almost_semiring
proof
  fix M N K :: natinf
  show "(M + N) + K = M + (N + K)"
    by (simp add: add_def split: natinf.split)
  show "M + N = N + M"
    by (simp add: add_def split: natinf.split)
  show "0 + M = M"
    by (simp add: zero_def add_def split: natinf.split)
  show "(M * N) * K = M * (N * K)"
```

```
    by (simp add: mult_def split: natinf.split)
  show "M * N = N * M"
    by (simp add: mult_def split: natinf.split)
  show "1 * M = M"
    by (simp add: one_def mult_def split: natinf.split)
  show "(M + N) * K = M * K + N * K"
    by (simp add: add_def mult_def left_distrib
          split: natinf.split)
  show "0 ≠ (1::natinf)"
    by (simp add: zero_def one_def)
qed
```

Now, let us examine the treatment of orderings. We define $\leq$ to extend the analogous relation on the natural numbers, with ∞ obviously the greatest element. We define $<$ as dictated by the type classes for orderings.

```
  le_def:
    "M ≤ N == (case M of Finite m ⇒
                         (case N of Finite n ⇒ Finite (m≤n)
                                  | Infinity ⇒ True)
                       | Infinity ⇒ (N=Infinity))"

  less_def: "M < N == M ≤ N ∧ M ≠ (N::natinf)"
```

Type `natinf` is linearly ordered. The five `linorder` axioms are stated by using **show** or **assume** ... **thus**. The proofs are again trivial.

```
instance natinf :: linorder
proof
  fix M N K :: natinf
  show "M ≤ M"  by (simp add: le_def split: natinf.split)
  {
    assume "M ≤ N" and "N ≤ M" thus "M = N"
      by (simp add: le_def split: natinf.split_asm)
   next
    assume "M ≤ N" and "N ≤ K" thus "M ≤ K"
      by (simp add: le_def split: natinf.split_asm)
   next
    show "(M < N) = (M ≤ N ∧ M ≠ N)"
      by (simp add: less_def)
    show "M ≤ N ∨ N ≤ M"
      by (simp add: le_def linorder_linear split: natinf.split)
  }
qed
```

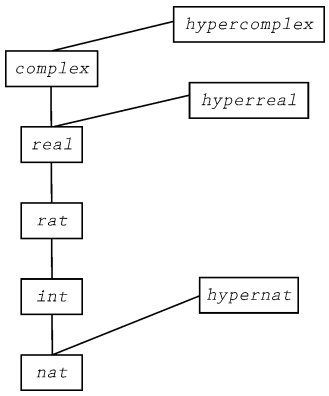

Figure 2. The arithmetic types of Isabelle/HOL.

At this point, we have accomplished much by comparison with the equivalent formalization in HOL. Type classes (not provided in HOL) allow existing theories to be reused. The Isabelle proof script is considerably shorter than the HOL one, but it provides access to many more theorems. Showing that `natinf` belongs to `linorder` makes Isabelle's library of facts about linear orderings available. Showing that `natinf` belongs to `almost_semiring` makes a few numeric lemmas available, as well as operators for sums and products over finite sets.

Unfortunately, type `natinf` lacks many key properties. Addition cannot be cancelled, and multiplication is not strictly monotonic: we have $\infty+M = \infty = \infty+N$ and $\infty \times M = \infty = \infty \times N$ for all M and N. The type cannot be shown to be a semiring, and most arithmetic laws require at least this. Without cancellation of addition, we cannot even prove $0 \times a = 0$, and indeed $0 \times \infty = \infty$ for type `natinf`. We now see that simply adding infinity to the natural numbers does not yield a theory that satisfies the usual arithmetic laws. Axiomatic type classes not only allow proofs to be reused but also provide a framework for analyzing proposed arithmetic types.

6. Conclusions

Axiomatic type classes work well in practice. Isabelle/HOL is distributed with eight arithmetic types (Figure 2). Five of these are traditional: the natural, integer, rational, real, and complex numbers. Three more arise from nonstandard analysis: the hypernatural, hyperreal, and hypercomplex numbers. The hyperreal numbers include infinitesimal and infinite values in addition to the usual real numbers; they allow a rigorous formalization of intuitive arguments about two values being infinitesimally close together, for example. The hypercomplex numbers are related to the hyperreal numbers in the obvious way. The hypernatural numbers, which are less well known, extend the natural numbers with infinite values; they form an ordered semiring and are therefore much better behaved than type `natinf`. All

three "hyper" types were defined from the standard types (using ultrafilters) by Fleuriot [2]. Most users will not require nonstandard analysis, but it is significant that type classes can cope with so many different types.

Subtyping is not the way to formalize such a complicated type hierarchy. Its inelegance would become glaringly obvious if we had to derive properties of the natural numbers from those of the hypercomplex numbers. Users would have to know about obscure types such as the hypercomplexes and hyperreals simply because of their position near the top of the hierarchy.

This paper can be seen as advocacy for axiomatic type classes. However, the key point is simply axiomatic development of abstract mathematics followed by application to concrete instances. A theory interpretation construct such as that of IMPS [1] may also work as a basis for organizing theories of arithmetic types. In order to provide a uniform notation, however, the approach also depends upon overloading. Theory interpretation of itself does not necessarily allow the symbol + to take on several different meanings in a single expression.

Also relevant is the constructive algebraic hierarchy formalized in Coq by Geuvers et al. [3]. They define concepts such as groups and rings. There appears to be no overloading of the familiar arithmetic operators. Instead, the work is a basis for developing constructive algebra.

Acknowledgments

Tobias Nipkow and Markus Wenzel equipped Isabelle with axiomatic type classes. The type classes for ordered rings and fields are based on earlier work by Gertrud Bauer and Wenzel. Mike Gordon suggested the example of extended natural numbers and provided information about the HOL formalization. Nipkow and Phil Wadler commented on this paper. Thanks are also due to the anonymous referees.

References

1. Farmer, W. M., Guttman, J. D. and Thayer, F. J.: IMPS: An interactive mathematical proof system, *J. Automated Reasoning* **11**(2) (1993), 213–248.
2. Fleuriot, J. D. and Paulson, L. C.: Mechanizing nonstandard real analysis, *LMS J. Comput. Math.* **3** (2000), 140–190, `http://www.lms.ac.uk/jcm/3/lms1999-027/`.
3. Geuvers, H., Pollack, R., Wiedijk, F. and Zwanenburg, J.: A constructive algebraic hierarchy in Coq, *J. Symbolic Comput.* **34**(4) (2002), 271–286.
4. Gordon, M. J. C. and Melham, T. F.: *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*, Cambridge University Press, 1993.
5. Hudak, P.: *The Haskell School of Expression*, Cambridge University Press, 2000.
6. Klein, G. and Nipkow, T.: Verified bytecode verifiers, *Theoret. Comput. Sci.* **298** (2003), 583–626.
7. Nipkow, T.: Order-sorted polymorphism in Isabelle, in G. Huet and G. Plotkin (eds.), *Logical Environments*, Cambridge University Press, 1993, pp. 164–188.
8. Nipkow, T., Paulson, L. C. and Wenzel, M.: *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, LNCS 2283, Springer, 2002.
9. Paulson, L. C.: *ML for the Working Programmer*, 2nd edn, Cambridge University Press, 1996.

10. Paulson, L. C.: Inductive analysis of the Internet protocol TLS, *ACM Transactions on Information and System Security* **2**(3) (1999), 332–351.
11. The PVS standard prelude, `http://pvs.csl.sri.com/doc/prelude.html`, 2003.
12. Wadler, P. and Blott, S.: How to make ad-hoc polymorphism less ad hoc, in *16th Annual Symposium on Principles of Programming Languages*, ACM Press, 1989, pp. 60–76.
13. Wenzel, M.: Type classes and overloading in higher-order logic, in E. L. Gunter and A. Felty (eds.), *Theorem Proving in Higher Order Logics: TPHOLs '97*, LNCS 1275, Springer, 1997, pp. 307–322.