Elements of Programming

Alexander Stepanov
Paul McJones

Elements of Programming

$$
\begin{array}{l}
\mathit{Ring}(\mathsf{T}) \triangleq \\
\quad \mathit{AdditiveGroup}(\mathsf{T}) \\
\quad \wedge\ \mathit{Semiring}(\mathsf{T})
\end{array}
$$

Matrices with integer coefficients constitute a ring.

$$
\begin{array}{l}
\mathit{CommutativeRing}(\mathsf{T}) \triangleq \\
\quad \mathit{AdditiveGroup}(\mathsf{T}) \\
\quad \wedge\ \mathit{CommutativeSemiring}(\mathsf{T})
\end{array}
$$

Integers constitute a commutative ring; polynomials with integer coefficients constitute a commutative ring.

A *relational concept* is a concept defined on two types. *Semimodule* is a relational concept that connects an additive monoid and a commutative semiring:

$$
\begin{array}{l}
\mathit{Semimodule}(\mathsf{T}, \mathsf{S}) \triangleq \\
\quad \mathit{AdditiveMonoid}(\mathsf{T}) \\
\quad \wedge\ \mathit{CommutativeSemiring}(\mathsf{S}) \\
\quad \wedge\ \cdot : \mathsf{S} \times \mathsf{T} \to \mathsf{T} \\
\quad \wedge\ (\forall \alpha, \beta \in \mathsf{S})(\forall \mathfrak{a}, \mathfrak{b} \in \mathsf{T}) \\
\qquad\qquad \begin{array}{rcl}
\alpha \cdot (\beta \cdot \mathfrak{a}) & = & (\alpha \cdot \beta) \cdot \mathfrak{a} \\
(\alpha + \beta) \cdot \mathfrak{a} & = & \alpha \cdot \mathfrak{a} + \beta \cdot \mathfrak{a} \\
\alpha \cdot (\mathfrak{a} + \mathfrak{b}) & = & \alpha \cdot \mathfrak{a} + \alpha \cdot \mathfrak{b} \\
1 \cdot \mathfrak{a} & = & \mathfrak{a}
\end{array}
\end{array}
$$

If $\mathit{Semimodule}(\mathsf{T}, \mathsf{S})$, we say that T is a semimodule *over* S. We borrow terminology from vector spaces and call elements of T *vectors* and elements of S *scalars*. For example, polynomials with non-negative integer coefficients constitute a semimodule over non-negative integers.

Theorem 5.1 $\mathit{AdditiveMonoid}(\mathsf{T}) \Rightarrow \mathit{Semimodule}(\mathsf{T}, \mathbb{N})$, where scalar multiplication is defined as $\mathfrak{n} \cdot \mathfrak{x} = \underbrace{\mathfrak{x} + \cdots + \mathfrak{x}}_{\mathfrak{n} \text{ times}}$.

Proof: It follows trivially from the definition of scalar multiplication together with associativity and commutativity of the monoid operation. For example,

$$
\begin{aligned}
\mathfrak{n} \cdot \mathfrak{a} + \mathfrak{n} \cdot \mathfrak{b} &= (\mathfrak{a} + \cdots + \mathfrak{a}) + (\mathfrak{b} + \cdots + \mathfrak{b}) \\
&= (\mathfrak{a} + \mathfrak{b}) + \cdots + (\mathfrak{a} + \mathfrak{b}) \\
&= \mathfrak{n} \cdot (\mathfrak{a} + \mathfrak{b})
\end{aligned}
$$

Using power from Chapter 3 allows us to implement multiplication by an integer in $\log_2 n$ steps.

Strengthening the requirements by replacing the additive monoid with an additive group and replacing the semiring with a ring transforms a semimodule into a module:

$$
\begin{array}{l}
\mathit{Module}(\mathsf{T}, \mathsf{S}) \triangleq \\
\quad \mathit{Semimodule}(\mathsf{T}, \mathsf{S}) \\
\quad \wedge\ \mathit{AdditiveGroup}(\mathsf{T}) \\
\quad \wedge\ \mathit{Ring}(\mathsf{S})
\end{array}
$$

Lemma 5.4 Every additive group is a module over integers with an appropriately defined scalar multiplication.

Computer types are often partial models of concepts. A model is called *partial* when the operations satisfy the axioms where they are defined but are not everywhere defined. For example, the result of concatenation of strings may not be representable, because of memory limitations, but concatenation is associative whenever it is defined.

5.2 Ordered Algebraic Structures

When a total ordering is defined on the elements of a structure in such a way that the ordering is consistent with the structure's algebraic properties, it is the *natural total ordering* for the structure:

$$
\begin{array}{l}
\mathit{OrderedAdditiveSemigroup}(\mathsf{T}) \triangleq \\
\quad \mathit{AdditiveSemigroup}(\mathsf{T}) \\
\quad \wedge\ \mathit{TotallyOrdered}(\mathsf{T}) \\
\quad \wedge\ (\forall a, b, c \in \mathsf{T})\ a < b \Rightarrow a + c < b + c
\end{array}
$$

$$
\begin{array}{l}
\mathit{OrderedAdditiveMonoid}(\mathsf{T}) \triangleq \\
\quad \mathit{OrderedAdditiveSemigroup}(\mathsf{T}) \\
\quad \wedge\ \mathit{AdditiveMonoid}(\mathsf{T})
\end{array}
$$

$$
\begin{array}{l}
\mathit{OrderedAdditiveGroup}(\mathsf{T}) \triangleq \\
\quad \mathit{OrderedAdditiveMonoid}(\mathsf{T}) \\
\quad \wedge\ \mathit{AdditiveGroup}(\mathsf{T})
\end{array}
$$

Lemma 5.5 In an ordered additive semigroup, $a < b \wedge c < d \Rightarrow a + c < b + d$.

Lemma 5.6 In an ordered additive monoid viewed as a semimodule over natural numbers, $a > 0 \wedge n > 0 \Rightarrow na > 0$.

Lemma 5.7 In an ordered additive group, $a < b \Rightarrow -b < -a$.

Total ordering and negation allow us to define absolute value:

```
template<typename T>
    requires(OrderedAdditiveGroup(T))
T abs(const T& a)
{
    if (a < T(0)) return -a;
    else          return  a;
}
```

The following lemma captures an important property of `abs`.

Lemma 5.8 In an ordered additive group, $a < 0 \Rightarrow 0 < -a$.

We use the notation $|a|$ for the absolute value of a. Absolute value satisfies the following properties.

Lemma 5.9

$$|a - b| = |b - a|$$
$$|a + b| \leq |a| + |b|$$
$$|a - b| \geq |a| - |b|$$
$$|a| = 0 \Rightarrow a = 0$$
$$a \neq 0 \Rightarrow |a| > 0$$

5.3 Remainder

We saw that repeated addition in an additive monoid induces multiplication by a non-negative integer. In an additive group, this algorithm can be inverted, obtaining division by repeated subtraction on elements of the form $a = nb$, where b divides a. To extend this to division with remainder for an arbitrary pair of elements, we need ordering. The ordering allows the algorithm to terminate when it is no longer

possible to subtract. As we shall see, it also enables an algorithm to take a logarithmic number of steps. The subtraction operation does not need to be defined everywhere; it is sufficient to have a partial subtraction called *cancellation*, where $a - b$ is only defined when b does not exceed a:

CancellableMonoid(T) $\triangleq$
  OrderedAdditiveMonoid(T)
  $\wedge$ $- : T \times T \to T$
  $\wedge$ $(\forall a, b \in T)\, b \leq a \Rightarrow a - b$ is defined $\wedge\ (a - b) + b = a$

We write the axiom as $(a - b) + b = a$ instead of $(a + b) - b = a$ to avoid overflow in partial models of *CancellableMonoid:*

```
template<typename T>
    requires(CancellableMonoid(T))
T slow_remainder(T a, T b)
{
    // Precondition: a ≥ 0 ∧ b > 0
    while (b <= a) a = a - b;
    return a;
}
```

The concept *CancellableMonoid* is not strong enough to prove termination of slow_remainder. For example, slow_remainder does not always terminate for polynomials with integer coefficients, ordered lexicographically.

Exercise 5.1 Give an example of two polynomials with integer coefficients for which the algorithm does not terminate.

To ensure that the algorithm terminates, we need another property, called the *Axiom of Archimedes:*[1]

ArchimedeanMonoid(T) $\triangleq$
  CancellableMonoid(T)
  $\wedge$ $(\forall a, b \in T)\, (a \geq 0 \wedge b > 0) \Rightarrow$ slow_remainder(a, b) terminates
  $\wedge$ QuotientType : *ArchimedeanMonoid* $\to$ *Integer*

1. " . . . the excess by which the greater of (two) unequal areas exceeds the less can, by being added to itself, be made to exceed any given finite area." See Heath [1912, page 234].

Observe that termination of an algorithm is a legitimate axiom; in this case it is equivalent to

$$(\exists n \in \mathsf{QuotientType}(T))\ a - n \cdot b < b$$

While the Axiom of Archimedes is usually given as "there exists an integer n such that $a < n \cdot b$," our version works with partial Archimedean monoids where $n \cdot b$ might overflow. The type function QuotientType returns a type large enough to represent the number of iterations performed by slow_remainder.

Lemma 5.10 The following are Archimedean monoids: integers, rational numbers, binary fractions $\{\frac{n}{2^k}\}$, ternary fractions $\{\frac{n}{3^k}\}$, and real numbers.

We can trivially adapt the code of slow_remainder to return the quotient:

```
template<typename T>
    requires(ArchimedeanMonoid(T))
QuotientType(T) slow_quotient(T a, T b)
{
    // Precondition: a ≥ 0 ∧ b > 0
    QuotientType(T) n(0);
    while (b <= a) {
        a = a - b;
        n = successor(n);
    }
    return n;
}
```

Repeated doubling leads to the logarithmic-complexity power algorithm. A related algorithm is possible for remainder.[2] Let us derive an expression for the remainder u from dividing a by b in terms of the remainder v from dividing a by $2b$:

$$a = n(2b) + v$$

Since the remainder v must be less than the divisor $2b$, it follows that

$$u = \begin{cases} v & \text{if } v < b \\ v - b & \text{if } v \geq b \end{cases}$$

2. The Egyptians used this algorithm to do division with remainder, as they used the power algorithm to do multiplication. See Robins and Shute [1987, page 18].

That leads to the following recursive procedure:

```
template<typename T>
    requires(ArchimedeanMonoid(T))
T remainder_recursive(T a, T b)
{
    // Precondition: a ≥ b > 0
    if (a - b >= b) {
        a = remainder_recursive(a, b + b);
        if (a < b) return a;
    }
    return a - b;
}
```

Testing $a - b \geq b$ rather than $a \geq b + b$ avoids overflow of $b + b$.

```
template<typename T>
    requires(ArchimedeanMonoid(T))
T remainder_nonnegative(T a, T b)
{
    // Precondition: a ≥ 0 ∧ b > 0
    if (a < b) return a;
    return remainder_recursive(a, b);
}
```

Exercise 5.2 Analyze the complexity of remainder_nonnegative.

Floyd and Knuth [1990] give a constant-space algorithm for remainder on Archimedean monoids that performs about 31% more operations than remainder_nonnegative, but when we can divide by 2 an algorithm exists that does not increase the operation count.[3] This is likely to be possible in many situations. For example, while the general k-section of an angle by ruler and compass cannot be done, the bisection is trivial.

HalvableMonoid(T) $\triangleq$

ArchimedeanMonoid(T)

$\wedge$ half : T $\to$ T

$\wedge$ $(\forall a, b \in T)\,(b > 0 \wedge a = b + b) \Rightarrow \text{half}(a) = b$

Observe that half needs to be defined only for "even" elements.

3. Dijkstra [1972, page 13] attributes this algorithm to N. G. de Bruijn.