Elements of Programming

Alexander Stepanov (A9.com) and Paul McJones

Stanford EE380
November 3, 2010

Abstract

This talk is an introduction to the book *Elements of Programming* published by Addison Wesley in 2009. The book presents practical programming as a mathematical discipline, where every programming construct has its place.

History

- C++ STL (1994) followed research on generic programming by Musser & Stepanov (1979–1992).
- At SGI (1996–1999) and Adobe (2004–2006), Stepanov taught courses on this approach.
- McJones collaborated with Stepanov (2007–2009) in weaving the material into its current form inspired by classical mathematical texts.

Acknowledgments

- Sean Parent
- Bjarne Stroustrup
- Jon Brandt
- John Wilkinson
- and many others

Audience

- The book addresses programmers who aspire to a deeper understanding of their discipline.
- In that, it is similar to Dijkstra's *A Discipline of Programming*.

Programming Language

- Requirements:
  - Powerful abstraction facilities
  - Faithful representation of the underlying machine
- Our solution:
  - A small subset of C++
  - Type requirements written as structured comments
  - An appendix specifying the subset (written by Parent and Stroustrup)

Premise

- The book applies the deductive method to programming by affiliating programs with the mathematical theories that enable them to work.
- This allows decomposition of complex systems into components with mathematically specified behavior.
- It leads to efficient, reliable, secure, and economical software.

Algorithmic Decomposition

- Complex algorithms are decomposable into simpler components with carefully defined interfaces.
- The components so discovered are then used to implement other algorithms.
- The iterative process going from complex to simple and back is central to the discovery of systematic catalogs of efficient components (such as STL).

The Fabric of the Book

- Three interwoven strands:
  - Specifications of relevant mathematical theories
  - Algorithms written in terms of these theories
  - Theorems describing their properties
- The book is intended to be read from beginning to end: *everything is connected*.

A Chain of Algorithms

- Memory-adaptive stable sort
- Memory-adaptive merge
- Rotate
- GCD
- Remainder

An example

- Simple, elegant, leads to interesting theory

```
T remainder(T a, T b)
{
    // Precondition: a ≥ b > 0
    if (a - b >= b) {
        a = remainder(a, b + b);
        if (a < b) return a;
    }
    return a - b;
}

// First appears in Rhind Papyrus
```

Intuition

- Reduce the problem of finding the remainder after division by *b* to the problem of remainder after division by 2*b*.

Correctness

- 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

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

Termination Condition

- If *b* is repeatedly doubled, it will eventually get sufficiently close to *a*.

What is the type *T*?

- Natural numbers
- Line segments
- Nonnegative real numbers

Syntactic Requirements

- + and –
- < and >=

Semantic Requirements

- + is associative, commutative
- – obeys cancellation law
- < is consistent with +
- >= is complement of <

Termination Requirements

- An integral quotient type exists
- $(\exists n \in \mathrm{QuotientType}(T))\ a - n \cdot b < b$

Archimedean Monoid

- A type satisfying these syntactic and semantic requirements is called an *Archimedean monoid*.

Concept

- A collection of syntactic and semantic requirements
- A set of types satisfying these requirements
- A set of algorithms enabled by these requirements

Concept: Affiliated Types

- Quotient type in Archimedean monoid
- Field of coefficients in a vector space

Concept: Operations

- Signatures using type and affiliated types
  - $+ : T \times T \to T$
  - $< : T \times T \to \text{Boolean}$
  - scalar product : $T \times T \to \text{CoefficientType}(T)$

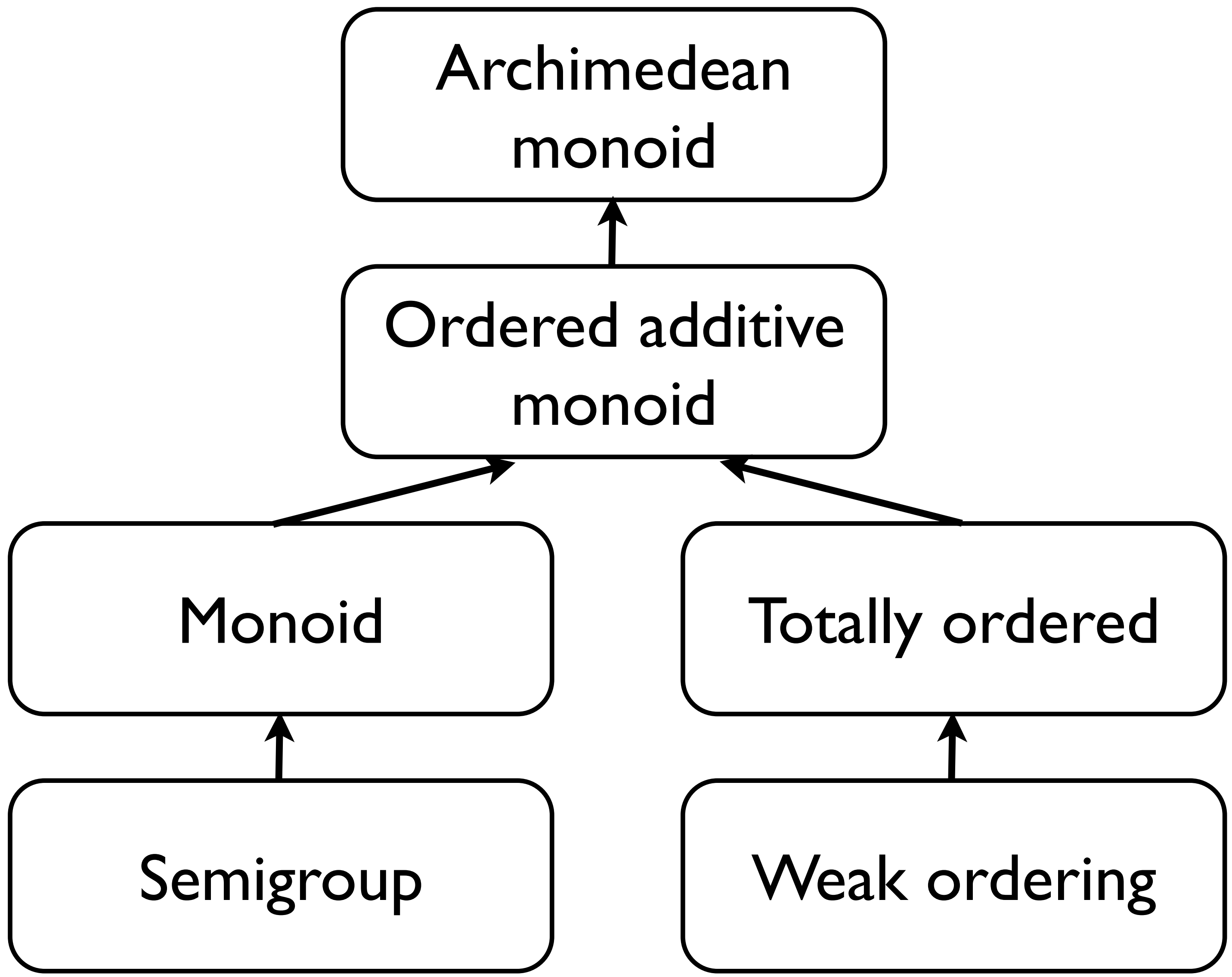
Concept: Axioms

- $+$ is associative and commutative
- $<$ is a strict total ordering
- $(\exists n \in \mathrm{QuotientType}(T))\ a - n \cdot b < b$

Values, Types, Concepts

- Value is a sequence of 0's and 1's together with its interpretation.
- Type is a set of values with the same interpretation function and operations on these values.
- Concept is a collection of similar types.
- Examples are 00000011_2, `uint8_t`, ring.

A Forest of Concepts



Respecting the Domain

```
T remainder(T a, T b)
{
    // Precondition: a ≥ b > 0
    if (a - b >= b) {
        a = remainder(a, b + b);
        if (a < b) return a;
    }
    return a - b;
}
```

Not:
`a >= b + b`

Partial Models

- Operations are partial.
- Axioms hold only when operations are defined.
- `int32_t` is a partial model of integers.

Plan of the Book

- Chapter 1 describes values, objects, types, procedures, and concepts.
- Chapters 2–5 describe algorithms on algebraic structures, such as semigroups and totally ordered sets.
- Chapters 6–11 describe algorithms on abstractions of memory.
- Chapter 12 describes objects containing other objects.
- The afterword presents our reflections on the approach presented by the book.

Memory

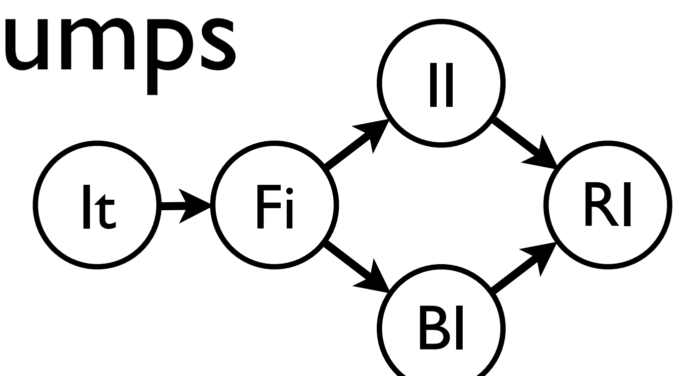
- Mathematics defines many concepts dealing with values: monoids, fields, compact spaces, ...
- Computers place values in memory.
- We define concepts for programming with memory.

Iterator

- Location in linear memory
- Affiliated types: ValueType, DistanceType
- Operations: successor, source
- Axiom:
  - If successor(*i*) is defined, source(*i*) is defined

Iterator Concepts

- Iterator: unidirectional, single-pass
- ForwardIterator: unidirectional, multipass
- BidirectionalIterator: bidirectional
- IndexedIterator: forward jumps
- RandomAccessIterator: random jumps



Coordinate Structures

- Iterator has unique successor.
- BifurcateCoordinate has left successor and right successor.
- LinkedBifurcateCoordinate has mutable successors.
- BidirectionalBifurcateCoordinate has predecessor.

Conclusions

- Programming is an iterative process:
  - Studying useful problems
  - Finding efficient algorithms for them
  - Distilling the concepts underlying the algorithms
  - Organizing the concepts and algorithms into a coherent mathematical theory.
- Each new discovery adds to the permanent body of knowledge, but each has its limitations.
- Theory is good for practice, and vice versa.