Elements of Programming

Alexander Stepanov
Paul McJones

Elements of Programming

This page intentionally left blank

Elements of Programming

Alexander Stepanov
Paul McJones

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States please contact:
International Sales
international@pearson.com

Visit us on the Web: www.informit.com/aw

Library of Congress Cataloging-in-Publication Data

Stepanov, Alexander A.
Elements of programming/Alexander Stepanov, Paul McJones.
p. cm.
Includes bibliographical references and index.
ISBN 0-321-63537-X (hardcover : alk. paper)
1. Computer programming. 2. Computer algorithms. I. McJones, Paul. II. Title.
QA76.6.S726 2009
005.1–dc22 2009007604

ISBN-13: 978-0-321-63537-2
ISBN-10: 0-321-63537-X

Text printed in the United States on recycled paper at Edwards Brothers in Ann Arbor, Michigan.
Third printing, June 2010

Contents

Preface

This book applies the deductive method to programming by affiliating programs with the abstract mathematical theories that enable them to work. Specification of these theories, algorithms written in terms of these theories, and theorems and lemmas describing their properties are presented together. The implementation of the algorithms in a real programming language is central to the book. While the specifications, which are addressed to human beings, should, and even must, combine rigor with appropriate informality, the code, which is addressed to the computer, must be absolutely precise even while being general.

As with other areas of science and engineering, the appropriate foundation of programming is the deductive method. It facilitates the decomposition of complex systems into components with mathematically specified behavior. That, in turn, is a necessary precondition for designing efficient, reliable, secure, and economical software.

The book is addressed to those who want a deeper understanding of programming, whether they are full-time software developers, or scientists and engineers for whom programming is an important part of their professional activity.

The book is intended to be read from beginning to end. Only by reading the code, proving the lemmas, and doing the exercises can readers gain understanding of the material. In addition, we suggest several projects, some open-ended. While the book is terse, a careful reader will eventually see the connections between its parts and the reasons for our choice of material. Discovering the architectural principles of the book should be the reader's goal.

We assume an ability to do elementary algebraic manipulations.[1] We also assume familiarity with the basic vocabulary of logic and set theory at the level of undergraduate courses on discrete mathematics; Appendix A summarizes the notation that we use. We provide definitions of a few concepts of abstract algebra when they are

1. For a refresher on elementary algebra, we recommend Chrystal [1904].

needed to specify algorithms. We assume programming maturity and understanding of computer architecture[2] and fundamental algorithms and data structures.[3]

We chose C++ because it combines powerful abstraction facilities with faithful representation of the underlying machine.[4] We use a small subset of the language and write requirements as structured comments. We hope that readers not already familiar with C++ are able to follow the book. Appendix B specifies the subset of the language used in the book.[5] Wherever there is a difference between mathematical notation and C++, the typesetting and the context determine whether the mathematical or C++ meaning applies. While many concepts and programs in the book have parallels in STL (the C++ Standard Template Library), the book departs from some of the STL design decisions. The book also ignores issues that a real library, such as STL, has to address: namespaces, visibility, inline directives, and so on.

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.

Acknowledgments

We are grateful to Adobe Systems and its management for supporting the Foundations of Programming course and this book, which grew out of it. In particular, Greg Gilley initiated the course and suggested writing the book; Dave Story and then Bill Hensler provided unwavering support. Finally, the book would not have been possible without Sean Parent's enlightened management and continuous scrutiny of the code and the text. The ideas in the book stem from our close collaboration, spanning almost three decades, with Dave Musser. Bjarne Stroustrup deliberately evolved C++ to support these ideas. Both Dave and Bjarne were kind enough to come to San Jose and carefully review the preliminary draft. Sean Parent and Bjarne Stroustrup wrote the appendix defining the C++ subset used in the book. Jon Brandt reviewed multiple drafts of the book. John Wilkinson carefully read the final manuscript, providing innumerable valuable suggestions.

2. We recommend Patterson and Hennessy [2007].

3. For a selective but incisive introduction to algorithms and data structures, we recommend Tarjan [1983].

4. The standard reference is Stroustrup [2000].

5. The code in the book compiles and runs under Microsoft Visual C++ 9 and g++ 4. This code, together with a few trivial macros that enable it to compile, as well as unit tests, can be downloaded from www.elementsofprogramming.com.

The book has benefited significantly from the contributions of our editor, Peter Gordon, our project editor, Elizabeth Ryan, our copy editor, Evelyn Pyle, and the editorial reviewers: Matt Austern, Andrew Koenig, David Musser, Arch Robison, Jerry Schwarz, Jeremy Siek, and John Wilkinson.

We thank all the students who took the course at Adobe and an earlier course at SGI for their suggestions. We hope we succeeded in weaving the material from these courses into a coherent whole. We are grateful for comments from Dave Abrahams, Andrei Alexandrescu, Konstantine Arkoudas, John Banning, Hans Boehm, Angelo Borsotti, Jim Dehnert, John DeTreville, Boris Fomitchev, Kevlin Henney, Jussi Ketonen, Karl Malbrain, Mat Marcus, Larry Masinter, Dave Parent, Dmitry Polukhin, Jon Reid, Mark Ruzon, Geoff Scott, David Simons, Anna Stepanov, Tony Van Eerd, Walter Vannini, Tim Winkler, and Oleg Zabluda. We thank John Banning, Bob English, Steven Gratton, Max Hailperin, Eugene Kirpichov, Alexei Nekrassov, Mark Ruzon, and Hao Song for finding errors in the first printing. We thank Foster Brereton, Gabriel Dos Reis, Ryan Ernst, Abraham Sebastian, Mike Spertus, Henning Thielemann, and Carla Villoria Burgazzi for finding errors in the second printing.[6]

Finally, we are grateful to all the people who taught us through their writings or in person, and to the institutions that allowed us to deepen our understanding of programming.

6. See www.elementsofprogramming.com for the up-to-date errata.

This page intentionally left blank

About the Authors

Alexander Stepanov studied mathematics at Moscow State University from 1967 to 1972. He has been programming since 1972: first in the Soviet Union and, after emigrating in 1977, in the United States. He has programmed operating systems, programming tools, compilers, and libraries. His work on foundations of programming has been supported by GE, Brooklyn Polytechnic, AT&T, HP, SGI, and Adobe. In 1995 he received the *Dr. Dobb's Journal* Excellence in Programming Award for the design of the C++ Standard Template Library.

Paul McJones studied engineering mathematics at the University of California, Berkeley, from 1967 to 1971. He has been programming since 1967 in the areas of operating systems, programming environments, transaction processing systems, and enterprise and consumer applications. He has been employed by the University of California, IBM, Xerox, Tandem, DEC, and Adobe. In 1982 he and his coauthors received the ACM Programming Systems and Languages Paper Award for their paper "The Recovery Manager of the System R Database Manager."

This page intentionally left blank

Chapter 2
Transformations and Their Orbits

This chapter defines a transformation as a unary regular function from a type to itself. Successive applications of a transformation starting from an initial value determine an orbit of this value. Depending only on the regularity of the transformation and the finiteness of the orbit, we implement an algorithm for determining orbit structures that can be used in different domains. For example, it could be used to detect a cycle in a linked list or to analyze a pseudorandom number generator. We derive an interface to the algorithm as a set of related procedures and definitions for their arguments and results. This analysis of an orbit-structure algorithm allows us to introduce our approach to programming in the simplest possible setting.

2.1 Transformations

While there are functions from any sequence of types to any type, particular classes of signatures commonly occur. In this book we frequently use two such classes: *homogeneous predicates* and *operations*. Homogeneous predicates are of the form $\mathsf{T} \times \cdots \times \mathsf{T} \to \mathsf{bool}$; operations are functions of the form $\mathsf{T} \times \cdots \times \mathsf{T} \to \mathsf{T}$. While there are n-ary predicates and n-ary operations, we encounter mostly unary and binary homogeneous predicates and unary and binary operations.

A *predicate* is a functional procedure returning a truth value:

$$
\begin{aligned}
&\mathit{Predicate}(\mathsf{P}) \triangleq \\
&\qquad \mathit{FunctionalProcedure}(\mathsf{P}) \\
&\quad \wedge\ \mathsf{Codomain}(\mathsf{P}) = \mathsf{bool}
\end{aligned}
$$

A homogeneous predicate is one that is also a homogeneous function:

$$\begin{aligned} &\mathit{HomogeneousPredicate}(\mathsf{P}) \triangleq \\ &\quad\quad \mathit{Predicate}(\mathsf{P}) \\ &\quad \wedge\ \mathit{HomogeneousFunction}(\mathsf{P}) \end{aligned}$$

A *unary predicate* is a predicate taking one parameter:

$$\begin{aligned} &\mathit{UnaryPredicate}(\mathsf{P}) \triangleq \\ &\quad\quad \mathit{Predicate}(\mathsf{P}) \\ &\quad \wedge\ \mathit{UnaryFunction}(\mathsf{P}) \end{aligned}$$

An *operation* is a homogeneous function whose codomain is equal to its domain:

$$\begin{aligned} &\mathit{Operation}(\mathsf{Op}) \triangleq \\ &\quad\quad \mathit{HomogeneousFunction}(\mathsf{Op}) \\ &\quad \wedge\ \mathsf{Codomain}(\mathsf{Op}) = \mathsf{Domain}(\mathsf{Op}) \end{aligned}$$

Examples of operations:

```
int abs(int x) {
    if (x < 0) return -x; else return x;
} // unary operation

double euclidean_norm(double x, double y) {
    return sqrt(x * x + y * y);
} // binary operation

double euclidean_norm(double x, double y, double z) {
    return sqrt(x * x + y * y + z * z);
} // ternary operation
```

Lemma 2.1 $\mathsf{euclidean\_norm}(x, y, z) = \mathsf{euclidean\_norm}(\mathsf{euclidean\_norm}(x, y), z)$

This lemma shows that the ternary version can be obtained from the binary version. For reasons of efficiency, expressiveness, and, possibly, accuracy, the ternary version is part of the computational basis for programs dealing with three-dimensional space.

A procedure is *partial* if its definition space is a subset of the direct product of the types of its inputs; it is *total* if its definition space is equal to the direct product. We follow standard mathematical usage, where partial function includes total function. We call partial procedures that are not total *nontotal*. Implementations of some total functions are nontotal on the computer because of the finiteness of the representation. For example, addition on signed 32-bit integers is nontotal.

A nontotal procedure is accompanied by a precondition specifying its definition space. To verify the correctness of a call of that procedure, we must determine that the arguments satisfy the precondition. Sometimes, a partial procedure is passed as a parameter to an algorithm that needs to determine at runtime the definition space of the procedural parameter. To deal with such cases, we define a *definition-space predicate* with the same inputs as the procedure; the predicate returns true if and only if the inputs are within the definition space of the procedure. Before a nontotal procedure is called, either its precondition must be satisfied, or the call must be guarded by a call of its definition-space predicate.

Exercise 2.1 Implement a definition-space predicate for addition on 32-bit signed integers.

This chapter deals with unary operations, which we call *transformations:*

$$\begin{aligned}
&\mathit{Transformation}(\mathsf{F}) \triangleq \\
&\quad \mathit{Operation}(\mathsf{F}) \\
&\wedge\ \mathit{UnaryFunction}(\mathsf{F}) \\
&\wedge\ \mathsf{DistanceType} : \mathit{Transformation} \to \mathit{Integer}
\end{aligned}$$

We discuss DistanceType in the next section.

Transformations are self-composable: $f(x)$, $f(f(x))$, $f(f(f(x)))$, and so on. The definition space of $f(f(x))$ is the intersection of the definition space and result space of f. This ability to self-compose, together with the ability to test for equality, allows us to define interesting algorithms.

When f is a transformation, we define its powers as follows:

$$f^n(x) = \begin{cases} x & \text{if } n = 0, \\ f^{n-1}(f(x)) & \text{if } n > 0 \end{cases}$$

To implement an algorithm to compute $f^n(x)$, we need to specify the requirement for an integer type. We study various concepts describing integers in Chapter 5. For now we rely on the intuitive understanding of integers. Their models include signed and unsigned integral types, as well as arbitrary-precision integers, with these operations and literals:

	Specifications	C++
Sum	$+$	`+`
Difference	$-$	`-`
Product	$\cdot$	`*`
Quotient	$/$	`/`
Remainder	mod	`%`
Zero	0	`I(0)`
One	1	`I(1)`
Two	2	`I(2)`

where I is an integer type.

That leads to the following algorithm:

```
template<typename F, typename N>
    requires(Transformation(F) && Integer(N))
Domain(F) power_unary(Domain(F) x, N n, F f)
{
    // Precondition: n ≥ 0 ∧ (∀i ∈ N) 0 < i ≤ n ⇒ f^i(x) is defined
    while (n != N(0)) {
        n = n - N(1);
        x = f(x);
    }
    return x;
}
```

2.2 Orbits

To understand the global behavior of a transformation, we examine the structure of its *orbits*: elements reachable from a starting element by repeated applications of the transformation. y is *reachable* from x under a transformation f if for some $n \geq 0$, $y = f^n(x)$. x is *cyclic* under f if for some $n \geq 1$, $x = f^n(x)$. x is *terminal* under f if and only if x is not in the definition space of f. The *orbit* of x under a transformation f is the set of all elements reachable from x under f.

Lemma 2.2 An orbit does not contain both a cyclic and a terminal element.

Lemma 2.3 An orbit contains at most one terminal element.

If y is reachable from x under f, the *distance* from x to y is the least number of transformation steps from x to y. Obviously, distance is not always defined.

Given a transformation type F, DistanceType(F) is an integer type large enough to encode the maximum number of steps by any transformation $f \in F$ from one element of T = Domain(F) to another. If type T occupies k bits, there can be as many as 2^k values but only $2^k - 1$ steps between distinct values. Thus if T is a fixed-size type, an integral type of the same size is a valid distance type for any transformation on T. (Instead of using the distance type, we allow the use of any integer type in power_unary, since the extra generality does not appear to hurt there.) It is often the case that all transformation types over a domain have the same distance type. In this case the type function DistanceType is defined for the domain type and defines the corresponding type function for the transformation types.

The existence of DistanceType leads to the following procedure:

```
template<typename F>
    requires(Transformation(F))
DistanceType(F) distance(Domain(F) x, Domain(F) y, F f)
{
    // Precondition: y is reachable from x under f
    typedef DistanceType(F) N;
    N n(0);
    while (x != y) {
        x = f(x);
        n = n + N(1);
    }
    return n;
}
```

Orbits have different shapes. An orbit of x under a transformation is

infinite	if it has no cyclic or terminal elements
terminating	if it has a terminal element
circular	if x is cyclic
ρ-shaped	if x is not cyclic, but its orbit contains a cyclic element

An orbit of x is *finite* if it is not infinite. Figure 2.1 illustrates the various cases.

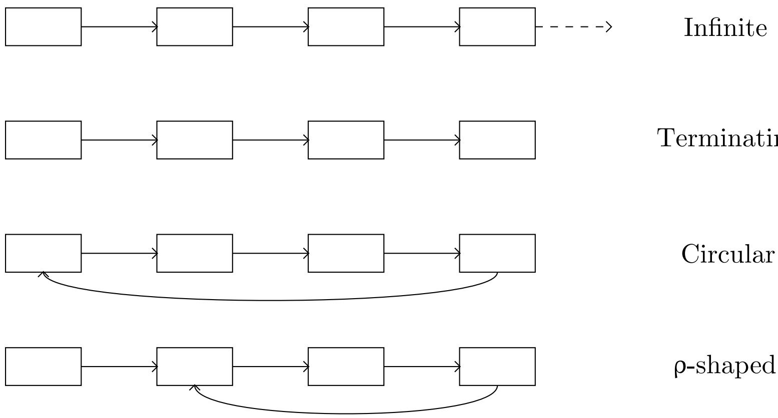

Figure 2.1 Orbit Shapes.

The *orbit cycle* is the set of cyclic elements in the orbit and is empty for infinite and terminating orbits. The *orbit handle*, the complement of the orbit cycle with respect to the orbit, is empty for a circular orbit. The *connection point* is the first cyclic element, and is the first element of a circular orbit and the first element after the handle for a ρ-shaped orbit. The *orbit size* o of an orbit is the number of distinct elements in it. The *handle size* h of an orbit is the number of elements in the orbit handle. The *cycle size* c of an orbit is the number of elements in the orbit cycle.

Lemma 2.4 $o = h + c$

Lemma 2.5 The distance from any point in an orbit to a point in a cycle of that orbit is always defined.

Lemma 2.6 If x and y are distinct points in a cycle of size c,

$$c = \mathsf{distance}(x, y, f) + \mathsf{distance}(y, x, f)$$

Lemma 2.7 If x and y are points in a cycle of size c, the distance from x to y satisfies

$$0 \leq \mathsf{distance}(x, y, f) < c$$

2.3 Collision Point

If we observe the behavior of a transformation, without access to its definition, we cannot determine whether a particular orbit is infinite: It might terminate or cycle back at any point. If we know that an orbit is finite, we can use an algorithm to determine the shape of the orbit. Therefore there is an implicit precondition of orbit finiteness for all the algorithms in this chapter.

There is, of course, a naive algorithm that stores every element visited and checks at every step whether the new element has been previously encountered. Even if we could use hashing to speed up the search, such an algorithm still would require linear storage and would not be practical in many applications. However, there is an algorithm that requires only a constant amount of storage.

The following analogy helps to understand the algorithm. If a fast car and a slow one start along a path, the fast one will catch up with the slow one if and only if there is a cycle. If there is no cycle, the fast one will reach the end of the path before the slow one. If there is a cycle, by the time the slow one enters the cycle, the fast one will already be there and will catch up eventually. Carrying our intuition from the continuous domain to the discrete domain requires care to avoid the fast one skipping past the slow one.[1]

The discrete version of the algorithm is based on looking for a point where fast meets slow. The *collision point* of a transformation f and a starting point x is the unique y such that

$$y = f^n(x) = f^{2n+1}(x)$$

and $n \geq 0$ is the smallest integer satisfying this condition. This definition leads to an algorithm for determining the orbit structure that needs one comparison of fast and slow per iteration. To handle partial transformations, we pass a definition-space predicate to the algorithm:

```
template<typename F, typename P>
    requires(Transformation(F) && UnaryPredicate(P) &&
        Domain(F) == Domain(P))
Domain(F) collision_point(const Domain(F)& x, F f, P p)
{
    // Precondition: p(x) ⇔ f(x) is defined
    if (!p(x)) return x;
```

1. Knuth [1997, page 7] attributes this algorithm to Robert W. Floyd.

```
    Domain(F) slow = x;         // slow = f^0(x)
    Domain(F) fast = f(x);      // fast = f^1(x)
                                // n ← 0 (completed iterations)
    while (fast != slow) {      // slow = f^n(x) ∧ fast = f^(2n+1)(x)
        slow = f(slow);         // slow = f^(n+1)(x) ∧ fast = f^(2n+1)(x)
        if (!p(fast)) return fast;
        fast = f(fast);         // slow = f^(n+1)(x) ∧ fast = f^(2n+2)(x)
        if (!p(fast)) return fast;
        fast = f(fast);         // slow = f^(n+1)(x) ∧ fast = f^(2n+3)(x)
                                // n ← n + 1
    }
    return fast;                // slow = f^n(x) ∧ fast = f^(2n+1)(x)
    // Postcondition: return value is terminal point or collision point
}
```

We establish the correctness of collision_point in three stages: (1) verifying that it never applies f to an argument outside the definition space; (2) verifying that if it terminates, the postcondition is satisfied; and (3) verifying that it always terminates.

While f is a partial function, its use by the procedure is well defined, since the movement of fast is guarded by a call of p. The movement of slow is unguarded, because by the regularity of f, slow traverses the same orbit as fast, so f is always defined when applied to slow.

The annotations show that if, after $n \geq 0$ iterations, fast becomes equal to slow, then fast $= f^{2n+1}(x)$ and slow $= f^n(x)$. Moreover, n is the smallest such integer, since we checked the condition for every $i < n$.

If there is no cycle, p will eventually return false because of finiteness. If there is a cycle, slow will eventually reach the connection point (the first element in the cycle). Consider the distance d from fast to slow at the top of the loop when slow first enters the cycle: $0 \leq d < c$. If $d = 0$, the procedure terminates. Otherwise the distance from fast to slow decreases by 1 on each iteration. Therefore the procedure always terminates; when it terminates, slow has moved a total of $h + d$ steps.

The following procedure determines whether an orbit is terminating:

```
template<typename F, typename P>
    requires(Transformation(F) && UnaryPredicate(P) &&
        Domain(F) == Domain(P))
bool terminating(const Domain(F)& x, F f, P p)
```

```
{
    // Precondition: p(x) ⇔ f(x) is defined
    return !p(collision_point(x, f, p));
}
```

Sometimes we know either that the transformation is total or that the orbit is nonterminating for a particular starting element. For these situations it is useful to have a specialized version of collision_point:

```
template<typename F>
    requires(Transformation(F))
Domain(F)
collision_point_nonterminating_orbit(const Domain(F)& x, F f)
{
    Domain(F) slow = x;          // slow = f^0(x)
    Domain(F) fast = f(x);       // fast = f^1(x)
                                 // n ← 0 (completed iterations)
    while (fast != slow) {       // slow = f^n(x) ∧ fast = f^{2n+1}(x)
        slow = f(slow);          // slow = f^{n+1}(x) ∧ fast = f^{2n+1}(x)
        fast = f(fast);          // slow = f^{n+1}(x) ∧ fast = f^{2n+2}(x)
        fast = f(fast);          // slow = f^{n+1}(x) ∧ fast = f^{2n+3}(x)
                                 // n ← n + 1
    }
    return fast;                 // slow = f^n(x) ∧ fast = f^{2n+1}(x)
    // Postcondition: return value is collision point
}
```

In order to determine the cycle structure—handle size, connection point, and cycle size—we need to analyze the position of the collision point.

When the procedure returns the collision point

$$f^n(x) = f^{2n+1}(x)$$

n is the number of steps taken by slow, and $2n + 1$ is the number of steps taken by fast.

$$n = h + d$$

where h is the handle size and $0 \leq d < c$ is the number of steps taken by `slow` inside the cycle. The number of steps taken by `fast` is

$$2n + 1 = h + d + qc$$

where $q > 0$ is the number of full cycles completed by `fast` when it collides with `slow`. Since $n = h + d$,

$$2(h + d) + 1 = h + d + qc$$

Simplifying gives

$$qc = h + d + 1$$

Let us represent h modulo c:

$$h = mc + r$$

with $0 \leq r < c$. Substitution gives

$$qc = mc + r + d + 1$$

or

$$d = (q - m)c - r - 1$$

$0 \leq d < c$ implies

$$q - m = 1$$

so

$$d = c - r - 1$$

and $r + 1$ steps are needed to complete the cycle.

Therefore the distance from the collision point to the connection point is

$$e = r + 1$$

In the case of a circular orbit $h = 0$, $r = 0$, and the distance from the collision point to the beginning of the orbit is

$$e = 1$$

Circularity, therefore, can be checked with the following procedures:

```
template<typename F>
    requires(Transformation(F))
bool circular_nonterminating_orbit(const Domain(F)& x, F f)
{
    return x == f(collision_point_nonterminating_orbit(x, f));
}

template<typename F, typename P>
    requires(Transformation(F) && UnaryPredicate(P) &&
        Domain(F) == Domain(P))
bool circular(const Domain(F)& x, F f, P p)
{
    // Precondition: p(x) ⇔ f(x) is defined
    Domain(F) y = collision_point(x, f, p);
    return p(y) && x == f(y);
}
```

We still don't know the handle size h and the cycle size c. Determining the latter is simple once the collision point is known: Traverse the cycle and count the steps.

To see how to determine h, let us look at the position of the collision point:

$$f^{h+d}(x) = f^{h+c-r-1}(x) = f^{mc+r+c-r-1}(x) = f^{(m+1)c-1}(x)$$

Taking $h + 1$ steps from the collision point gets us to the point $f^{(m+1)c+h}(x)$, which equals $f^h(x)$, since $(m + 1)c$ corresponds to going around the cycle $m + 1$ times. If we simultaneously take h steps from x and $h + 1$ steps from the collision point, we meet at the connection point. In other words, the orbits of x and 1 step past the collision point converge in exactly h steps, which leads to the following sequence of algorithms:

```
template<typename F>
    requires(Transformation(F))
Domain(F) convergent_point(Domain(F) x0, Domain(F) x1, F f)
{
    // Precondition: (∃n ∈ DistanceType(F)) n ≥ 0 ∧ f^n(x0) = f^n(x1)
    while (x0 != x1) {
```

```
        x0 = f(x0);
        x1 = f(x1);
    }
    return x0;
}

template<typename F>
    requires(Transformation(F))
Domain(F)
connection_point_nonterminating_orbit(const Domain(F)& x, F f)
{
    return convergent_point(
        x,
        f(collision_point_nonterminating_orbit(x, f)),
        f);
}

template<typename F, typename P>
    requires(Transformation(F) && UnaryPredicate(P) &&
        Domain(F) == Domain(P))
Domain(F) connection_point(const Domain(F)& x, F f, P p)
{
    // Precondition: p(x) ⇔ f(x) is defined
    Domain(F) y = collision_point(x, f, p);
    if (!p(y)) return y;
    return convergent_point(x, f(y), f);
}
```

Lemma 2.8 If the orbits of two elements intersect, they have the same cyclic elements.

Exercise 2.2 Design an algorithm that determines, given a transformation and its definition-space predicate, whether the orbits of two elements intersect.

Exercise 2.3 The precondition of convergent_point ensures termination. Implement an algorithm convergent_point_guarded for use when that precondition is not known to hold, but there is an element in common to the orbits of both x0 and x1.

2.4 Measuring Orbit Sizes

The natural type to use for the sizes o, h, and c of an orbit on type T would be an integer count type large enough to count all the distinct values of type T. If a type T occupies k bits, there can be as many as 2^k values, so a count type occupying k bits could not represent all the counts from 0 to 2^k. There is a way to represent these sizes by using distance type.

An orbit could potentially contain all values of a type, in which case o might not fit in the distance type. Depending on the shape of such an orbit, h and c would not fit either. However, for a ρ-shaped orbit, both h and c fit. In all cases each of these fits: $o - 1$ (the maximum distance in the orbit), $h - 1$ (the maximum distance in the handle), and $c - 1$ (the maximum distance in the cycle). That allows us to implement procedures returning a triple representing the complete structure of an orbit, where the members of the triple are as follows:

Case	m0	m1	m2
Terminating	$h - 1$	0	terminal element
Circular	0	$c - 1$	x
ρ-shaped	h	$c - 1$	connection point

```
template<typename F>
    requires(Transformation(F))
triple<DistanceType(F), DistanceType(F), Domain(F)>
orbit_structure_nonterminating_orbit(const Domain(F)& x, F f)
{
    typedef DistanceType(F) N;
    Domain(F) y = connection_point_nonterminating_orbit(x, f);
    return triple<N, N, Domain(F)>(distance(x, y, f),
                                   distance(f(y), y, f),
                                   y);
}

template<typename F, typename P>
    requires(Transformation(F) &&
        UnaryPredicate(P) && Domain(F) == Domain(P))
triple<DistanceType(F), DistanceType(F), Domain(F)>
orbit_structure(const Domain(F)& x, F f, P p)
{
    // Precondition: p(x) ⇔ f(x) is defined
```

```
    typedef DistanceType(F) N;
    Domain(F) y = connection_point(x, f, p);
    N m = distance(x, y, f);
    N n(0);
    if (p(y)) n = distance(f(y), y, f);
    // Terminating: m = h − 1 ∧ n = 0
    // Otherwise: m = h ∧ n = c − 1
    return triple<N, N, Domain(F)>(m, n, y);
}
```

Exercise 2.4 Derive formulas for the count of different operations (f, p, equality) for the algorithms in this chapter.

Exercise 2.5 Use orbit_structure_nonterminating_orbit to determine the average handle size and cycle size of the pseudorandom number generators on your platform for various seeds.

2.5 Actions

Algorithms often use a transformation f in a statement like

```
x = f(x);
```

Changing the state of an object by applying a transformation to it defines an *action* on the object. There is a duality between transformations and the corresponding actions: An action is definable in terms of a transformation, and vice versa:

```
void a(T& x) { x = f(x); }     // action from transformation
```

and

```
T f(T x) { a(x); return x; } // transformation from action
```

Despite this duality, independent implementations are sometimes more efficient, in which case both action and transformation need to be provided. For example, if a transformation is defined on a large object and modifies only part of its overall state, the action could be considerably faster.

Exercise 2.6 Rewrite all the algorithms in this chapter in terms of actions.

Project 2.1 Another way to detect a cycle is to repeatedly test a single advancing element for equality with a stored element while replacing the stored element at ever-increasing intervals. This and other ideas are described in Sedgewick, et al. [1979], Brent [1980], and Levy [1982]. Implement other algorithms for orbit analysis, compare their performance for different applications, and develop a set of recommendations for selecting the appropriate algorithm.

2.6 Conclusions

Abstraction allowed us to define abstract procedures that can be used in different domains. Regularity of types and functions is essential to make the algorithms work: `fast` and `slow` follow the same orbit because of regularity. Developing nomenclature is essential (e.g., orbit kinds and sizes). Affiliated types, such as distance type, need to be precisely defined.

Index

S

This page intentionally left blank