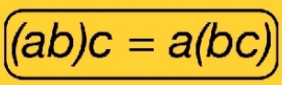
ELEMENTS OF PROGRAMMING

Alexander Stepanov
Paul McJones



Semigroup Press
Palo Alto • Mountain View

Elements of Programming

Alexander Stepanov

Paul McJones

$(ab)c = a(bc)$

Semigroup Press

Palo Alto • Mountain View

ISBN-13: 978-0-578-22214-1

First printing, June 2019

Contents

Preface to Authors' Edition

After ten years in print, our publisher decided against further printings and has reverted the rights to us. We have decided to publish *Elements of Programming* in two forms: a free PDF and a paperback; see `elementsofprogramming.com` for details.

The book is now typeset by us using LaTeX, and the text includes corrections for all errata reported to us from previous printings (see the Acknowledgments). We will attempt to apply corrections promptly.

We have made no changes other than these corrections, and do not expect to do so in the future. Readers may be interested in this additional book on the same subject:

From Mathematics to Generic Programming
by Alexander A. Stepanov and Daniel E. Rose
Addison-Wesley Professional, 2014

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

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

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 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

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.

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.

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. We thank Shinji Dosaka, Ryan Ernst, Steven Gratton, and Abraham Sebastian for finding errors in the third printing. We thank Matt Austern, Robert Jan Harteveld, Daniel Krügler, Volker Lukas, Veljko Miljanic, Doug Morgan, Jeremy Murphy, Qiu Zongyan, Mark Ruzon, Yoshiki Shibata, Sean Silva, Andrej Sprogar, Mitsutaka Takeda, Stefan Vargyas, and Guilliam Xavier for finding errors in the (third and) fourth printing. We thank Jeremy Murphy, Robert Southee, and Yutaka Tsutano for finding errors in the sixth printing, and Fernando Pelliccioni for proofreading the Authors' Edition.[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.

Chapter 1

Foundations

Starting with a brief taxonomy of ideas, we introduce notions of value, object, type, procedure, *and* concept *that represent different categories of ideas in the computer. A central notion of the book,* regularity, *is introduced and elaborated. When applied to procedures, regularity means that procedures return equal results for equal arguments. When applied to types, regularity means that types possess the equality operator and equality-preserving copy construction and assignment. Regularity enables us to apply* equational reasoning *(substituting equals for equals) to transform and optimize programs.*

1.1 Categories of Ideas: Entity, Species, Genus

In order to explain what objects, types, and other foundational computer notions are, it is useful to give an overview of some categories of ideas that correspond to these notions.

An *abstract entity* is an individual thing that is eternal and unchangeable, while a *concrete entity* is an individual thing that comes into and out of existence in space and time. An *attribute*—a correspondence between a concrete entity and an abstract entity—describes some property, measurement, or quality of the concrete entity. *Identity*, a primitive notion of our perception of reality, determines the sameness of a thing changing over time. Attributes of a concrete entity can change without affecting its identity. A *snapshot* of a concrete entity is a complete collection of its attributes at a particular point in time. Concrete entities are not only physical entities

but also legal, financial, or political entities. Blue and 13 are examples of abstract entities. Socrates and the United States of America are examples of concrete entities. The color of Socrates' eyes and the number of U.S. states are examples of attributes.

An *abstract species* describes common properties of essentially equivalent abstract entities. Examples of abstract species are natural number and color. A *concrete species* describes the set of attributes of essentially equivalent concrete entities. Examples of concrete species are man and U.S. state.

A *function* is a rule that associates one or more abstract entities, called *arguments*, from corresponding species with an abstract entity, called the *result*, from another species. Examples of functions are the successor function, which associates each natural number with the one that immediately follows it, and the function that associates with two colors the result of blending them.

An *abstract genus* describes different abstract species that are similar in some respect. Examples of abstract genera are number and binary operator. A *concrete genus* describes different concrete species similar in some respect. Examples of concrete genera are mammal and biped.

An entity belongs to a single species, which provides the rules for its construction or existence. An entity can belong to several genera, each of which describes certain properties.

We show later in the chapter that objects and values represent entities, types represent species, and concepts represent genera.

1.2 Values

Unless we know the interpretation, the only things we see in a computer are 0s and 1s. A *datum* is a finite sequence of 0s and 1s.

A *value type* is a correspondence between a species (abstract or concrete) and a set of datums. A datum corresponding to a particular entity is called a *representation* of the entity; the entity is called the *interpretation* of the datum. We refer to a datum together with its interpretation as a *value*. Examples of values are integers represented in 32-bit two's complement big-endian format and rational numbers represented as a concatenation of two 32-bit sequences, interpreted as integer numerator and denominator, represented as two's complement big-endian values.

A datum is *well formed* with respect to a value type if and only if that datum represents an abstract entity. For example, every sequence of 32 bits is well formed when interpreted as a two's-complement integer; an IEEE 754

floating-point NaN (Not a Number) is not well formed when interpreted as a real number.

A value type is *properly partial* if its values represent a proper subset of the abstract entities in the corresponding species; otherwise it is *total*. For example, the type `int` is properly partial, while the type `bool` is total.

A value type is *uniquely represented* if and only if at most one value corresponds to each abstract entity. For example, a type representing a truth value as a byte that interprets zero as false and nonzero as true is not uniquely represented. A type representing an integer as a sign bit and an unsigned magnitude does not provide a unique representation of zero. A type representing an integer in two's complement is uniquely represented.

A value type is *ambiguous* if and only if a value of the type has more than one interpretation. The negation of ambiguous is *unambiguous*. For example, a type representing a calendar year over a period longer than a single century as two decimal digits is ambiguous.

Two values of a value type are *equal* if and only if they represent the same abstract entity. They are *representationally equal* if and only if their datums are identical sequences of 0s and 1s.

Lemma 1.1 If a value type is uniquely represented, equality implies representational equality.

Lemma 1.2 If a value type is not ambiguous, representational equality implies equality.

If a value type is uniquely represented, we implement equality by testing that both sequences of 0s and 1s are the same. Otherwise we must implement equality in such a way that preserves its consistency with the interpretations of its arguments. Nonunique representations are chosen when testing equality is done less frequently than operations generating new values and when it is possible to make generating new values faster at the cost of making equality slower. For example, two rational numbers represented as pairs of integers are equal if they reduce to the same lowest terms. Two finite sets represented as unsorted sequences are equal if, after sorting and eliminating duplicates, their corresponding elements are equal.

Sometimes, implementing true *behavioral* equality is too expensive or even impossible, as in the case for a type of encodings of computable functions. In these cases we must settle for the weaker *representational* equality: that two values are the same sequence of 0s and 1s.

Computers *implement* functions on abstract entities as functions on values. While values reside in memory, a properly implemented function on

values does not depend on particular memory addresses: It implements a mapping from values to values.

A function defined on a value type is *regular* if and only if it respects equality: Substituting an equal value for an argument gives an equal result. Most numeric functions are regular. An example of a numeric function that is not regular is the function that returns the numerator of a rational number represented as a pair of integers, since $\frac{1}{2} = \frac{2}{4}$, but numerator($\frac{1}{2}$) $\neq$ numerator($\frac{2}{4}$). Regular functions allow *equational reasoning:* substituting equals for equals.

A nonregular function depends on the representation, not just the interpretation, of its argument. When designing the representation for a value type, two tasks go hand in hand: implementing equality and deciding which functions will be regular.

1.3 Objects

A *memory* is a set of words, each with an *address* and a *content*. The addresses are values of a fixed size, called the *address length*. The contents are values of another fixed size, called the *word length*. The content of an address is obtained by a *load* operation. The association of a content with an address is changed by a *store* operation. Examples of memories are bytes in main memory and blocks on a disk drive.

An *object* is a representation of a concrete entity as a value in memory. An object has a *state* that is a value of some value type. The state of an object is changeable. Given an object corresponding to a concrete entity, its state corresponds to a snapshot of that entity. An object owns a set of *resources*, such as memory words or records in a file, to hold its state.

While the value of an object is a contiguous sequence of 0s and 1s, the resources in which these 0s and 1s are stored are not necessarily contiguous. It is the interpretation that gives unity to an object. For example, two `double`s may be interpreted as a single complex number even if they are not adjacent. The resources of an object might even be in different memories. This book, however, deals only with objects residing in a single memory with one address space. Every object has a unique *starting address*, from which all its resources can be reached.

An *object type* is a pattern for storing and modifying values in memory. Corresponding to every object type is a value type describing states of objects of that type. Every object belongs to an object type. An example of an object type is integers represented in 32-bit two's complement little-endian

format aligned to a 4-byte address boundary.

Values and objects play complementary roles. Values are unchanging and are independent of any particular implementation in the computer. Objects are changeable and have computer-specific implementations. The state of an object at any point in time can be described by a value; this value could in principle be written down on paper (making a snapshot) or *serialized* and sent over a communication link. Describing the states of objects in terms of values allows us to abstract from the particular implementations of the objects when discussing equality. Functional programming deals with values; imperative programming deals with objects.

We use values to represent entities. Since values are unchanging, they can represent abstract entities. Sequences of values can also represent sequences of snapshots of concrete entities. Objects hold values representing entities. Since objects are changeable, they can represent concrete entities by taking on a new value to represent a change in the entity. Objects can also represent abstract entities: staying constant or taking on different approximations to the abstract.

We use objects in the computer for the following three reasons.

1. Objects model changeable concrete entities, such as employee records in a payroll application.

2. Objects provide a powerful way to implement functions on values, such as a procedure implementing the square root of a floating-point number using an iterative algorithm.

3. Computers with memory constitute the only available realization of a universal computational device.

Some properties of value types carry through to object types. An object is *well formed* if and only if its state is well formed. An object type is *properly partial* if and only if its value type is properly partial; otherwise it is *total*. An object type is *uniquely represented* if and only if its value type is uniquely represented.

Since concrete entities have identities, objects representing them need a corresponding notion of identity. An *identity token* is a unique value expressing the identity of an object and is computed from the value of the object and the address of its resources. Examples of identity tokens are the address of the object, an index into an array where the object is stored, and an employee number in a personnel record. Testing equality of identity tokens corresponds to testing identity. During the lifetime of an application,

a particular object could use different identity tokens as it moves either within a data structure or from one data structure to another.

Two objects of the same type are *equal* if and only if their states are equal. If two objects are equal, we say that one is a *copy* of the other. Making a change to an object does not affect any copy of it.

This book uses a programming language that has no way to describe values and value types as separate from objects and object types. So from this point on, when we refer to types without qualification, we mean object types.

1.4 Procedures

A *procedure* is a sequence of instructions that modifies the state of some objects; it may also construct or destroy objects.

The objects with which a procedure interacts can be divided into four kinds, corresponding to the intentions of the programmer.

1. *Input/output* consists of objects passed to/from a procedure directly or indirectly through its arguments or returned result.

2. *Local state* consists of objects created, destroyed, and usually modified during a single invocation of the procedure.

3. *Global state* consists of objects accessible to this and other procedures across multiple invocations.

4. *Own state* consists of objects accessible only to this procedure (and its affiliated procedures) but shared across multiple invocations.

An object is passed *directly* if it is passed as an argument or returned as the result and is passed *indirectly* if it is passed via a pointer or pointerlike object. An object is an *input* to a procedure if it is read, but not modified, by the procedure. An object is an *output* from a procedure if it is written, created, or destroyed by the procedure, but its initial state is not read by the procedure. An object is an *input/output* of a procedure if it is modified as well as read by the procedure.

A *computational basis* for a type is a finite set of procedures that enable the construction of any other procedure on the type. A basis is *efficient* if and only if any procedure implemented using it is as efficient as an equivalent procedure written in terms of an alternative basis. For example, a basis for unsigned k-bit integers providing only zero, equality, and the successor

function is not efficient, since the complexity of addition in terms of successor is exponential in k.

A basis is *expressive* if and only if it allows compact and convenient definitions of procedures on the type. In particular, all the common mathematical operations need to be provided when they are appropriate. For example, subtraction could be implemented using negation and addition but should be included in an expressive basis. Similarly, negation could be implemented using subtraction and zero but should be included in an expressive basis.

1.5 Regular Types

There is a set of procedures whose inclusion in the computational basis of a type lets us place objects in data structures and use algorithms to copy objects from one data structure to another. We call types having such a basis *regular*, since their use guarantees regularity of behavior and, therefore, interoperability.[1] We derive the semantics of regular types from built-in types, such as `bool`, `int`, and, when restricted to well-formed values, `double`. A type is *regular* if and only if its basis includes equality, assignment, destructor, default constructor, copy constructor, total ordering,[2] and underlying type.[3]

Equality is a procedure that takes two objects of the same type and returns true if and only if the object states are equal. Inequality is always defined and returns the negation of equality. We use the following notation:

	Specifications	C++
Equality	$a = b$	`a == b`
Inequality	$a \neq b$	`a != b`

Assignment is a procedure that takes two objects of the same type and makes the first object equal to the second without modifying the second. The meaning of assignment does not depend on the initial value of the first object. We use the following notation:

	Specifications	C++
Assignment	$a \leftarrow b$	`a = b`

1. While regular types underlie the design of STL, they were first formally introduced in Dehnert and Stepanov [2000].
2. Strictly speaking, as becomes clear in Chapter 4, it could be either total ordering or default total ordering.
3. Underlying type is defined in Chapter 12.

A *destructor* is a procedure causing the cessation of an object's existence. After a destructor has been called on an object, no procedure can be applied to it, and its former memory locations and resources may be reused for other purposes. The destructor is normally invoked implicitly. Global objects are destroyed when the application terminates, local objects are destroyed when the block in which they are declared is exited, and elements of a data structure are destroyed when the data structure is destroyed.

A *constructor* is a procedure transforming memory locations into an object. The possible behaviors range from doing nothing to establishing a complex object state.

An object is in a *partially formed* state if it can be assigned to or destroyed. For an object that is partially formed but not well formed, the effect of any procedure other than assignment (only on the left side) and destruction is not defined.

Lemma 1.3 A well-formed object is partially formed.

A *default constructor* takes no arguments and leaves the object in a partially formed state. We use the following notation:

	C++
Local object of type `T`	`T a;`
Anonymous object of type `T`	`T()`

A *copy constructor* takes an additional argument of the same type and constructs a new object equal to it. We use the following notation:

	C++
Local copy of object `b`	`T a = b;`

1.6 Regular Procedures

A procedure is *regular* if and only if replacing its inputs with equal objects results in equal output objects. As with value types, when defining an object type we must make consistent choices in how to implement equality and which procedures on the type will be regular.

Exercise 1.1 Extend the notion of regularity to input/output objects of a procedure, that is, to objects that are modified as well as read.

While regularity is the default, there are reasons for nonregular behavior of procedures.

1. A procedure returns the address of an object; for example, the built-in function `addressof`.
2. A procedure returns a value determined by the state of the real world, such as the value of a clock or other device.
3. A procedure returns a value depending on own state; for example, a pseudorandom number generator.
4. A procedure returns a representation-dependent attribute of an object, such as the amount of reserved memory for a data structure.

A *functional procedure* is a regular procedure defined on regular types, with one or more direct inputs and a single output that is returned as the result of the procedure. The regularity of functional procedures allows two techniques for passing inputs. When the size of the parameter is small or if the procedure needs a copy it can mutate, we pass it *by value*, making a local copy. Otherwise we pass it *by constant reference*. A functional procedure can be implemented as a C++ function, function pointer, or function object.[4]

This is a functional procedure:

```
int plus_0(int a, int b)
{
    return a + b;
}
```

This is a semantically equivalent functional procedure:

```
int plus_1(const int& a, const int& b)
{
    return a + b;
}
```

This is semantically equivalent but is not a functional procedure, because its inputs and outputs are passed indirectly:

```
void plus_2(int* a, int* b, int* c)
{
    *c = *a + *b;
}
```

In plus_2, `a` and `b` are input objects, while `c` is an output object. The notion of a functional procedure is a syntactic rather than semantic property: In our terminology, plus_2 is regular but not functional.

4. C++ functions are not objects and cannot be passed as arguments; C++ function pointers and function objects are objects and can be passed as arguments.

The *definition space* for a functional procedure is that subset of values for its inputs to which it is intended to be applied. A functional procedure always terminates on input in its definition space; while it may terminate for input outside its definition space, it may not return a meaningful value.

A *homogeneous* functional procedure is one whose input objects are all the same type. The *domain* of a homogeneous functional procedure is the type of its inputs. Rather than defining the domain of a nonhomogeneous functional procedure as the direct product of its input types, we refer individually to the input types of a procedure.

The *codomain* for a functional procedure is the type of its output. The *result space* for a functional procedure is the set of all values from its codomain returned by the procedure for inputs from its definition space.

Consider the functional procedure

```
int square(int n) { return n * n; }
```

While its domain and codomain are `int`, its definition space is the set of integers whose square is representable in the type, and its result space is the set of square integers representable in the type.

Exercise 1.2 Assuming that `int` is a 32-bit two's complement type, determine the exact definition and result space.

1.7 Concepts

A procedure using a type depends on syntactic, semantic, and complexity properties of the computational basis of the type. Syntactically it depends on the presence of certain literals and procedures with particular names and signatures. Its semantics depend on properties of these procedures. Its complexity depends on the time and space complexity of these procedures. A program remains correct if a type is replaced by a different type with the same properties. The utility of a software component, such as a library procedure or data structure, is increased by designing it not in terms of concrete types but in terms of requirements on types expressed as syntactic and semantic properties. We call a collection of requirements a *concept*. Types represent species; concepts represent genera.

In order to describe concepts, we need several mechanisms dealing with types: type attributes, type functions, and type constructors. A *type attribute* is a mapping from a type to a value describing some characteristic of the type. Examples of type attributes are the built-in type attribute `sizeof(T)` in C++, the alignment of an object of a type, and the number

of members in a `struct`. If F is a functional procedure type, Arity(F) returns its number of inputs. A *type function* is a mapping from a type to an affiliated type. An example of a type function is: given "pointer to T," the type T. In some cases it is useful to define an *indexed* type function with an additional constant integer parameter. For example, a type function returning the type of the i*th* member of a structure type (counting from 0). If F is a functional procedure type, the type function Codomain(F) returns the type of the result. If F is a functional procedure type and $\mathtt{i} < \mathsf{Arity}(\mathsf{F})$, the indexed type function InputType(F, i) returns the type of the i*th* parameter (counting from 0).[5] A *type constructor* is a mechanism for creating a new type from one or more existing types. For example, `pointer(T)` is the built-in type constructor that takes a type `T` and returns the type "pointer to `T`"; `struct` is a built-in $\mathfrak{n}$-ary type constructor; a structure template is a user-defined $\mathfrak{n}$-ary type constructor.

If $\mathcal{T}$ is an $\mathfrak{n}$-ary type constructor, we usually denote its application to types $\mathsf{T}_0, \ldots, \mathsf{T}_{\mathfrak{n}-1}$ as $\mathcal{T}_{\mathsf{T}_0,\ldots,\mathsf{T}_{\mathfrak{n}-1}}$. An important example is pair, which, when applied to regular types T_0 and T_1, returns a `struct` type $\mathsf{pair}_{\mathsf{T}_0,\mathsf{T}_1}$ with a member $\mathfrak{m}0$ of type T_0 and a member $\mathfrak{m}1$ of type T_1. To ensure that the type $\mathsf{pair}_{\mathsf{T}_0,\mathsf{T}_1}$ is itself regular, equality, assignment, destructor, and constructors are defined through memberwise extensions of the corresponding operations on the types T_0 and T_1. The same technique is used for any tuple type, such as triple. In Chapter 12 we show the implementation of $\mathsf{pair}_{\mathsf{T}_0,\mathsf{T}_1}$ and describe how regularity is preserved by more complicated type constructors.

Somewhat more formally, a *concept* is a description of requirements on one or more types stated in terms of the existence and properties of procedures, type attributes, and type functions defined on the types. We say that a concept is *modeled by* specific types, or that the types *model* the concept, if the requirements are satisfied for these types. To assert that a concept $\mathcal{C}$ is modeled by types $\mathsf{T}_0, \ldots, \mathsf{T}_{\mathfrak{n}-1}$, we write $\mathcal{C}(\mathsf{T}_0, \ldots, \mathsf{T}_{\mathfrak{n}-1})$. Concept $\mathcal{C}'$ *refines* concept $\mathcal{C}$ if whenever $\mathcal{C}'$ is satisfied for a set of types, $\mathcal{C}$ is also satisfied for those types. We say that $\mathcal{C}$ *weakens* $\mathcal{C}'$ if $\mathcal{C}'$ refines $\mathcal{C}$.

A *type concept* is a concept defined on one type. For example, C++ defines the type concept *integral type*, which is refined by *unsigned integral type* and by *signed integral type*, while STL defines the type concept *sequence*. We use the primitive type concepts *Regular* and *FunctionalProcedure*, corresponding to the informal definitions we gave earlier.

We define concepts formally by using standard mathematical notation. To define a concept $\mathcal{C}$, we write

5. Appendix B shows how to define type attributes and type functions in C++.

$$
\begin{aligned}
&\mathcal{C}(\mathsf{T}_0, \ldots, \mathsf{T}_{n-1}) \triangleq \\
&\quad \mathcal{E}_0 \\
&\wedge\ \mathcal{E}_1 \\
&\wedge\ \ldots \\
&\wedge\ \mathcal{E}_{k-1}
\end{aligned}
$$

where $\triangleq$ is read as "is equal to by definition," the T_i are formal type parameters, and the $\mathcal{E}_j$ are concept clauses, which take one of three forms:

1. Application of a previously defined concept, indicating a subset of the type parameters modeling it.

2. Signature of a type attribute, type function, or procedure that must exist for any types modeling the concept. A procedure signature takes the form $\mathsf{f} : \mathsf{T} \to \mathsf{T}'$, where T is the domain and T' is the codomain. A type function signature takes the form $\mathsf{F} : \mathcal{C} \to \mathcal{C}'$, where the domain and codomain are concepts.

3. Axiom expressed in terms of these type attributes, type functions, and procedures.

We sometimes include the definition of a type attribute, type function, or procedure following its signature in the second kind of concept clause. It takes the form $\mathsf{x} \mapsto \mathcal{F}(\mathsf{x})$ for some expression $\mathcal{F}$. In a particular model, such a definition could be overridden with a different but consistent implementation.

For example, this concept describes a unary functional procedure:

$$
\begin{aligned}
&\mathit{UnaryFunction}(\mathsf{F}) \triangleq \\
&\quad \mathit{FunctionalProcedure}(\mathsf{F}) \\
&\wedge\ \mathsf{Arity}(\mathsf{F}) = 1 \\
&\wedge\ \mathsf{Domain} : \mathit{UnaryFunction} \to \mathit{Regular} \\
&\qquad \mathsf{F} \mapsto \mathsf{InputType}(\mathsf{F}, 0)
\end{aligned}
$$

This concept describes a homogeneous functional procedure:

$$
\begin{aligned}
&\mathit{HomogeneousFunction}(\mathsf{F}) \triangleq \\
&\quad \mathit{FunctionalProcedure}(\mathsf{F}) \\
&\wedge\ \mathsf{Arity}(\mathsf{F}) > 0 \\
&\wedge\ (\forall \mathsf{i}, \mathsf{j} \in \mathbb{N})(\mathsf{i}, \mathsf{j} < \mathsf{Arity}(\mathsf{F})) \Rightarrow (\mathsf{InputType}(\mathsf{F}, \mathsf{i}) = \mathsf{InputType}(\mathsf{F}, \mathsf{j})) \\
&\wedge\ \mathsf{Domain} : \mathit{HomogeneousFunction} \to \mathit{Regular} \\
&\qquad \mathsf{F} \mapsto \mathsf{InputType}(\mathsf{F}, 0)
\end{aligned}
$$

Observe that

$$(\forall \mathsf{F} \in \mathit{FunctionalProcedure})\ \mathit{UnaryFunction}(\mathsf{F}) \Rightarrow \mathit{HomogeneousFunction}(\mathsf{F})$$

An *abstract* procedure is parameterized by types and constant values, with requirements on these parameters.[6] We use function templates and function object templates. The parameters follow the `template` keyword and are introduced by `typename` for types and `int` or another integral type for constant values. Requirements are specified via the `requires` clause, whose argument is an expression built up from constant values, concrete types, formal parameters, applications of type attributes and type functions, equality on values and types, concepts, and logical connectives.[7]

Here is an example of an abstract procedure:

```
template<typename Op>
    requires(BinaryOperation(Op))
Domain(Op) square(const Domain(Op)& x, Op op)
{
    return op(x, x);
}
```

The domain values could be large, so we pass them by constant reference. Operations tend to be small (e.g., a function pointer or small function object), so we pass them by value.

Concepts describe properties satisfied by all objects of a type, whereas *preconditions* describe properties of particular objects. For example, a procedure might require a parameter to be a prime number. The requirement for an integer type is specified by a concept, while primality is specified by a precondition. The type of a function pointer expresses only its signature, not its semantic properties. For example, a procedure might require a parameter to be a pointer to a function implementing an associative binary operation on integers. The requirement for a binary operation on integers is specified by a concept; associativity of a particular function is specified by a precondition.

To define a precondition for a family of types, we need to use mathematical notation, such as universal and existential quantifiers, implication, and so on. For example, to specify the primality of an integer, we define

6. Abstract procedures appeared, in substantially the form we use them, in 1930 in van der Waerden [1930], which was based on the lectures of Emmy Noether and Emil Artin. George Collins and David Musser used them in the context of computer algebra in the late 1960s and early 1970s. See, for example, Musser [1975].
7. See Appendix B for the full syntax of the `requires` clause.

property(N : *Integer*)
prime : N
$\quad n \mapsto (|n| \neq 1) \wedge (\forall u, v \in \mathsf{N})\, uv = n \Rightarrow (|u| = 1 \vee |v| = 1)$

where the first line introduces formal type parameters and the concepts they model, the second line names the property and gives its signature, and the third line gives the predicate establishing whether the property holds for a given argument.

To define regularity of a unary functional procedure, we write

property(F : *UnaryFunction*)
regular_unary_function : F
$\quad f \mapsto (\forall f' \in \mathsf{F})(\forall x, x' \in \mathsf{Domain}(\mathsf{F}))$
$\qquad (f = f' \wedge x = x') \Rightarrow (f(x) = f'(x'))$

The definition easily extends to n-ary functions: Application of equal functions to equal arguments gives equal results. By extension, we call an abstract function regular if all its instantiations are regular. In this book every procedural argument is a regular function unless otherwise stated; we omit the precondition stating this explicitly.

Project 1.1 Extend the notions of equality, assignment, and copy construction to objects of distinct types. Think about the interpretations of the two types and axioms that connect cross-type procedures.

1.8 Conclusions

The commonsense view of reality humans share has a representation in the computer. By grounding the meanings of values and objects in their interpretations, we obtain a simple, coherent view. Design decisions, such as how to define equality, become straightforward when the correspondence to entities is taken into account.

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 $\mathsf{T} \times \ldots \times \mathsf{T} \to \mathsf{bool}$; operations are functions of the form $\mathsf{T} \times \ldots \times \mathsf{T} \to \mathsf{T}$. While there are $\mathfrak{n}$-ary predicates and $\mathfrak{n}$-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:

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

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

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

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

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

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

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

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:*

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

We discuss DistanceType in the next section.

Transformations are self-composable: $f(x), f(f(x)), f(f(f(x)))$, and so on. 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 $n \geqslant 0$, $y = f^n(x)$. x is *cyclic* under f if for some $n \geqslant 1$, $x = f^n(x)$. 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 $T = \mathsf{Domain}(F)$ 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 unsigned 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

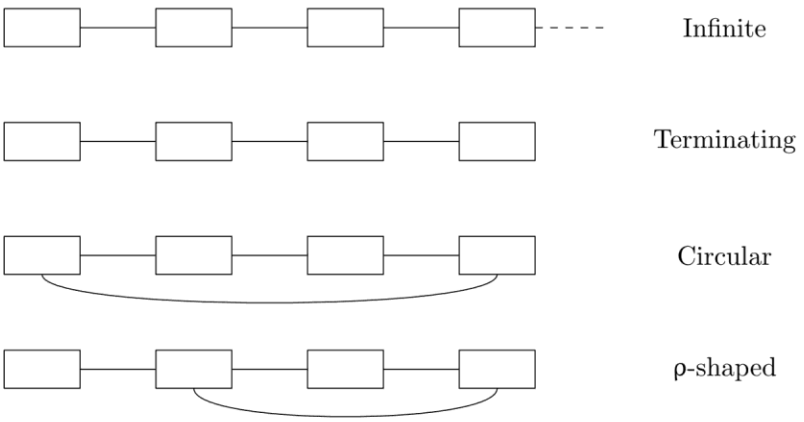


Figure 2.1: Orbit Shapes

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.

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

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

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 $n \geqslant 0$ 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
```

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

```
    if (!p(x)) return x;
    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 $\mathtt{fast}$ is guarded by a call of p. The movement of $\mathtt{slow}$ is unguarded, because by the regularity of f, $\mathtt{slow}$ traverses the same orbit as $\mathtt{fast}$, so f is always defined when applied to $\mathtt{slow}$.

The annotations show that if, after $n \geqslant 0$ iterations, $\mathtt{fast}$ becomes equal to $\mathtt{slow}$, then $\mathtt{fast} = f^{2n+1}(x)$ and $\mathtt{slow} = f^n(x)$. 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, $\mathtt{slow}$ will eventually reach the connection point (the first element in the cycle). Consider the distance d from $\mathtt{fast}$ to $\mathtt{slow}$ at the top of the loop when $\mathtt{slow}$ first enters the cycle: $0 \leqslant d < c$. If $d = 0$, the procedure terminates. Otherwise the distance from $\mathtt{fast}$ to $\mathtt{slow}$ decreases by 1 on each iteration. Therefore the procedure always terminates; when it terminates, $\mathtt{slow}$ 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 $0 \leqslant d < c$ 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 $0 \leqslant r < c$. Substitution gives

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

or

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

$0 \leqslant d < c$ 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. [1982], 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.

Chapter 3

Associative Operations

This chapter discusses associative binary operations. Associativity allows regrouping the adjacent operations. This ability to regroup leads to an efficient algorithm for computing powers of the binary operation. Regularity enables a variety of program transformations to optimize the algorithm. We then use the algorithm to compute linear recurrences, such as Fibonacci numbers, in logarithmic time.

3.1 Associativity

A binary operation is an operation with two arguments:

$$\begin{array}{l} \mathit{BinaryOperation}(\mathsf{Op}) \triangleq \\ \quad \mathit{Operation}(\mathsf{Op}) \\ \quad \land\ \mathsf{Arity}(\mathsf{Op}) = 2 \end{array}$$

The binary operations of addition and multiplication are central to mathematics. Many more are used, such as min, max, conjunction, disjunction, set union, set intersection, and so on. All these operations are *associative:*

$$\begin{array}{l} \textbf{property}(\mathsf{Op} : \mathit{BinaryOperation}) \\ \quad \mathsf{associative} : \mathsf{Op} \\ \qquad \mathsf{op} \mapsto (\forall a, b, c \in \mathsf{Domain}(\mathsf{Op}))\ \mathsf{op}(\mathsf{op}(a, b), c) = \mathsf{op}(a, \mathsf{op}(b, c)) \end{array}$$

There are, of course, nonassociative binary operations, such as subtraction and division.

When a particular associative binary operation op is clear from the context, we often use implied multiplicative notation by writing ab instead of $\mathsf{op}(a, b)$. Because of associativity, we do not need to parenthesize an expression involving two or more applications of op, because all the groupings are equivalent: $(\cdots(a_0a_1)\cdots)a_{n-1} = \cdots = a_0(\cdots(a_{n-2}a_{n-1})\cdots) = a_0a_1\cdots a_{n-1}$. When $a_0 = a_1 = \cdots = a_{n-1} = a$, we write a^n: the nth *power* of a.

Lemma 3.1 $a^n a^m = a^m a^n = a^{n+m}$ (powers of the same element commute)

Lemma 3.2 $(a^n)^m = a^{nm}$

It is not, however, always true that $(ab)^n = a^n b^n$. This condition holds only when the operation is commutative.

If f and g are transformations on the same domain, their *composition*, $g \circ f$, is a transformation mapping x to $g(f(x))$.

Lemma 3.3 The binary operation of composition is associative.

If we choose some element a of the domain of an associative operation op and consider the expression $\mathsf{op}(a, x)$ as a unary operation with formal parameter x, we can think of a as the transformation "multiplication by a." This justifies the use of the same notation for powers of a transformation, f^n, and powers of an element under an associative binary operation, a^n. This duality allows us to use an algorithm from the previous chapter to prove an interesting theorem about powers of an associative operation. An element x has *finite order* under an associative operation if there exist integers $0 < n < m$ such that $x^n = x^m$. An element x is an *idempotent element* under an associative operation if $x = x^2$.

Theorem 3.1 An element of finite order has an idempotent power (Frobenius [1895]).

Proof. Assume that x is an element of finite order under an associative operation op. Let $g(z) = \mathsf{op}(x, z)$. Since x is an element of finite order, its orbit under g has a cycle. By its postcondition,

$$\mathsf{collision\_point}(x, g) = g^n(x) = g^{2n+1}(x)$$

for some $n \geqslant 0$. Thus

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

and x^{n+1} is the idempotent power of x. □

Lemma 3.4 collision_point_nonterminating_orbit can be used in the proof.

3.2 Computing Powers

An algorithm to compute a^n for an associative operation op will take a, n, and op as parameters. The type of a is the domain of op; n must be of an integer type. Without the assumption of associativity, two algorithms compute power from left to right and right to left, respectively:

```
template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power_left_associated(Domain(Op) a, I n, Op op)
{
    // Precondition: n > 0
    if (n == I(1)) return a;
    return op(power_left_associated(a, n - I(1), op), a);
}


template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power_right_associated(Domain(Op) a, I n, Op op)
{
    // Precondition: n > 0
    if (n == I(1)) return a;
    return op(a, power_right_associated(a, n - I(1), op));
}
```

The algorithms perform $n - 1$ operations. They return different results for a nonassociative operation. Consider, for example, raising 1 to the 3rd power with the operation of subtraction.

When both a and n are integers, and if the operation is multiplication, both algorithms give us exponentiation; if the operation is addition, both give us multiplication. The ancient Egyptians discovered a faster multiplication algorithm that can be generalized to computing powers of any associative operation.[1]

1. The original is in Robins and Shute [1987, pages 16–17]; the papyrus is from around 1650 BC but its scribe noted that it was a copy of another papyrus from around 1850 BC.

Since associativity allows us to freely regroup operations, we have

$$a^n = \begin{cases} a & \text{if } n = 1 \\ (a^2)^{n/2} & \text{if } n \text{ is even} \\ (a^2)^{\lfloor n/2 \rfloor} a & \text{if } n \text{ is odd} \end{cases}$$

which corresponds to

```
template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power_0(Domain(Op) a, I n, Op op)
{
    // Precondition: associative(op) ∧ n > 0
    if (n == I(1)) return a;
    if (n % I(2) == I(0))
        return power_0(op(a, a), n / I(2), op);
    return op(power_0(op(a, a), n / I(2), op), a);
}
```

Let us count the number of operations performed by power_0 for an exponent of n. The number of recursive calls is $\lfloor \log_2 n \rfloor$. Let ν be the number of 1s in the binary representation of n. Each recursive call performs an operation to square a. Also, $\nu - 1$ of the calls perform an extra operation. So the number of operations is

$$\lfloor \log_2 n \rfloor + (\nu - 1) \leqslant 2 \lfloor \log_2 n \rfloor$$

For $n = 15$, $\lfloor \log_2 n \rfloor = 3$ and the number of 1s is four, so the formula gives six operations. A different grouping gives $a^{15} = (a^3)^5$, where a^3 takes two operations and a^5 takes three operations, for a total of five. There are also faster groupings for other exponents, such as 23, 27, 39, and 43.[2]

Since power_left_associated does $n - 1$ operations and power_0 does at most $2\lfloor \log_2 n \rfloor$ operations, it might appear that for very large n, power_0 will always be much faster. This is not always the case. For example, if the operation is multiplication of univariate polynomials with arbitrary-precision integer coefficients, power_left_associated is faster.[3] Even for this simple algorithm, we do not know how to precisely specify the complexity requirements that determine which of the two is better.

2. For a comprehensive discussion of minimal-operation exponentiation, see Knuth [1997, pages 465–481].
3. See McCarthy [1986].

The ability of power_0 to handle very large exponents, say 10^{300}, makes it crucial for cryptography.[4]

3.3 Program Transformations

power_0 is a satisfactory implementation of the algorithm and is appropriate when the cost of performing the operation is considerably larger than the overhead of the function calls caused by recursion. In this section we derive the iterative algorithm that performs the same number of operations as power_0, using a sequence of program transformations that can be used in many contexts.[5] For the rest of the book, we only show final or almost-final versions.

power_0 contains two identical recursive calls. While only one is executed in a given invocation, it is possible to reduce the code size via *common-subexpression elimination:*

```
template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power_1(Domain(Op) a, I n, Op op)
{
    // Precondition: associative(op) ∧ n > 0
    if (n == I(1)) return a;
    Domain(Op) r = power_1(op(a, a), n / I(2), op);
    if (n % I(2) != I(0)) r = op(r, a);
    return r;
}
```

Our goal is to eliminate the recursive call. A first step is to transform the procedure to *tail-recursive form*, where the procedure's execution ends with the recursive call. One of the techniques that allows this transformation is *accumulation-variable introduction*, where the accumulation variable carries the accumulated result between recursive calls:

```
template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power_accumulate_0(Domain(Op) r, Domain(Op) a, I n,
                              Op op)
```

4. See the work on RSA by Rivest et al. [1978].

5. Compilers perform similar transformations only for built-in types when the semantics and complexity of the operations are known. The concept of regularity is an assertion by the creator of a type that programmers and compilers can safely perform such transformations.

```
{
    // Precondition: associative(op) ∧ n ⩾ 0
    if (n == I(0)) return r;
    if (n % I(2) != I(0)) r = op(r, a);
    return power_accumulate_0(r, op(a, a), n / I(2), op);
}
```

If r_0, a_0, and n_0 are the original values of r, a, and n, this invariant holds at every recursive call: $ra^n = r_0 a_0^{n_0}$. As an additional benefit, this version computes not just power but also power multiplied by a coefficient. It also handles zero as the value of the exponent. However, power_accumulate_0 does an unnecessary squaring when going from 1 to 0. That can be eliminated by adding an extra case:

```
template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power_accumulate_1(Domain(Op) r, Domain(Op) a, I n,
                              Op op)
{
    // Precondition: associative(op) ∧ n ⩾ 0
    if (n == I(0)) return r;
    if (n == I(1)) return op(r, a);
    if (n % I(2) != I(0)) r = op(r, a);
    return power_accumulate_1(r, op(a, a), n / I(2), op);
}
```

Adding the extra case results in a duplicated subexpression and in three tests that are not independent. Analyzing the dependencies between the tests and ordering the tests based on expected frequency gives

```
template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power_accumulate_2(Domain(Op) r, Domain(Op) a, I n,
                              Op op)
{
    // Precondition: associative(op) ∧ n ⩾ 0
    if (n % I(2) != I(0)) {
        r = op(r, a);
        if (n == I(1)) return r;
    } else if (n == I(0)) return r;
    return power_accumulate_2(r, op(a, a), n / I(2), op);
}
```

A *strict tail-recursive* procedure is one in which all the tail-recursive calls are done with the formal parameters of the procedure being the corresponding arguments:

```
template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power_accumulate_3(Domain(Op) r, Domain(Op) a, I n,
                              Op op)
{
    // Precondition: associative(op) ∧ n ⩾ 0
    if (n % I(2) != I(0)) {
        r = op(r, a);
        if (n == I(1)) return r;
    } else if (n == I(0)) return r;
    a = op(a, a);
    n = n / I(2);
    return power_accumulate_3(r, a, n, op);
}
```

A strict tail-recursive procedure can be transformed to an iterative procedure by replacing each recursive call with a `goto` to the beginning of the procedure or by using an equivalent iterative construct:

```
template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power_accumulate_4(Domain(Op) r, Domain(Op) a, I n,
                              Op op)
{
    // Precondition: associative(op) ∧ n ⩾ 0
    while (true) {
        if (n % I(2) != I(0)) {
            r = op(r, a);
            if (n == I(1)) return r;
        } else if (n == I(0)) return r;
        a = op(a, a);
        n = n / I(2);
    }
}
```

The recursion invariant becomes the *loop invariant*.

If $n > 0$ initially, it would pass through 1 before becoming 0. We take advantage of this by eliminating the test for 0 and strengthening the pre-

condition:

```
template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power_accumulate_positive_0(Domain(Op) r,
                                       Domain(Op) a, I n,
                                       Op op)
{
    // Precondition: associative(op) ∧ n > 0
    while (true) {
        if (n % I(2) != I(0)) {
            r = op(r, a);
            if (n == I(1)) return r;
        }
        a = op(a, a);
        n = n / I(2);
    }
}
```

This is useful when it is known that $n > 0$. While developing a component, we often discover new interfaces.

Now we relax the precondition again:

```
template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power_accumulate_5(Domain(Op) r, Domain(Op) a, I n,
                              Op op)
{
    // Precondition: associative(op) ∧ n ⩾ 0
    if (n == I(0)) return r;
    return power_accumulate_positive_0(r, a, n, op);
}
```

We can implement power from power_accumulate by using a simple identity:

$$a^n = aa^{n-1}$$

The transformation is *accumulation-variable elimination:*

```
template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power_2(Domain(Op) a, I n, Op op)
{
```

```
    // Precondition: associative(op) ∧ n > 0
    return power_accumulate_5(a, a, n - I(1), op);
}
```

This algorithm performs more operations than necessary. For example, when n is 16, it performs seven operations where only four are needed. When n is odd, this algorithm is fine. Therefore we can avoid the problem by repeated squaring of a and halving the exponent until it becomes odd:

```
template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power_3(Domain(Op) a, I n, Op op)
{
    // Precondition: associative(op) ∧ n > 0
    while (n % I(2) == I(0)) {
        a = op(a, a);
        n = n / I(2);
    }
    n = n / I(2);
    if (n == I(0)) return a;
    return power_accumulate_positive_0(a, op(a, a), n, op);
}
```

Exercise 3.1 Convince yourself that the last three lines of code are correct.

3.4 Special-Case Procedures

In the final versions we used these operations:

```
n / I(2)
n % I(2) == I(0)
n % I(2) != I(0)
n == I(0)
n == I(1)
```

Both `/` and `%` are expensive. We can use shifts and masks on non-negative values of both signed and unsigned integers.

It is frequently useful to identify commonly occuring expressions involving procedures and constants of a type by defining *special-case* procedures. Often these special cases can be implemented more efficiently than the general case and, therefore, belong to the computational basis of the type. For

built-in types, there may exist machine instructions for the special cases. For user-defined types, there are often even more significant opportunities for optimizing special cases. For example, division of two arbitrary polynomials is more difficult than division of a polynomial by $\mathtt{x}$. Similarly, division of two Gaussian integers (numbers of the form $\mathtt{a} + \mathtt{bi}$ where $\mathtt{a}$ and $\mathtt{b}$ are integers and $\mathtt{i} = \sqrt{-1}$) is more difficult than division of a Gaussian integer by $1 + \mathtt{i}$.

Any integer type must provide the following special-case procedures:

$$
\begin{aligned}
&\mathit{Integer}(\mathrm{I}) \triangleq \\
&\quad \mathsf{successor} : \mathrm{I} \to \mathrm{I} \\
&\qquad n \mapsto n + 1 \\
&\wedge\ \mathsf{predecessor} : \mathrm{I} \to \mathrm{I} \\
&\qquad n \mapsto n - 1 \\
&\wedge\ \mathsf{twice} : \mathrm{I} \to \mathrm{I} \\
&\qquad n \mapsto n + n \\
&\wedge\ \mathsf{half\_nonnegative} : \mathrm{I} \to \mathrm{I} \\
&\qquad n \mapsto \lfloor n/2 \rfloor, \text{ where } n \geqslant 0 \\
&\wedge\ \mathsf{binary\_scale\_down\_nonnegative} : \mathrm{I} \times \mathrm{I} \to \mathrm{I} \\
&\qquad (n, k) \mapsto \lfloor n/2^k \rfloor, \text{ where } n, k \geqslant 0 \\
&\wedge\ \mathsf{binary\_scale\_up\_nonnegative} : \mathrm{I} \times \mathrm{I} \to \mathrm{I} \\
&\qquad (n, k) \mapsto 2^k n, \text{ where } n, k \geqslant 0 \\
&\wedge\ \mathsf{positive} : \mathrm{I} \to \mathsf{bool} \\
&\qquad n \mapsto n > 0 \\
&\wedge\ \mathsf{negative} : \mathrm{I} \to \mathsf{bool} \\
&\qquad n \mapsto n < 0 \\
&\wedge\ \mathsf{zero} : \mathrm{I} \to \mathsf{bool} \\
&\qquad n \mapsto n = 0 \\
&\wedge\ \mathsf{one} : \mathrm{I} \to \mathsf{bool} \\
&\qquad n \mapsto n = 1 \\
&\wedge\ \mathsf{even} : \mathrm{I} \to \mathsf{bool} \\
&\qquad n \mapsto (n \bmod 2) = 0 \\
&\wedge\ \mathsf{odd} : \mathrm{I} \to \mathsf{bool} \\
&\qquad n \mapsto (n \bmod 2) \neq 0
\end{aligned}
$$

Exercise 3.2 Implement these procedures for C++ integral types.

Now we can give the final implementations of the power procedures by using the special-case procedures:

```
template<typename I, typename Op>
```

```
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power_accumulate_positive(Domain(Op) r,
                                     Domain(Op) a, I n,
                                     Op op)
{
    // Precondition: associative(op) ∧ positive(n)
    while (true) {
      if (odd(n)) {
          r = op(r, a);
          if (one(n)) return r;
      }
      a = op(a, a);
      n = half_nonnegative(n);
    }
}


template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power_accumulate(Domain(Op) r, Domain(Op) a, I n,
                            Op op)
{
    // Precondition: associative(op) ∧ ¬negative(n)
    if (zero(n)) return r;
    return power_accumulate_positive(r, a, n, op);
}


template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power(Domain(Op) a, I n, Op op)
{
    // Precondition: associative(op) ∧ positive(n)
    while (even(n)) {
        a = op(a, a);
        n = half_nonnegative(n);
    }
    n = half_nonnegative(n);
    if (zero(n)) return a;
    return power_accumulate_positive(a, op(a, a), n, op);
```

```
}
```

Since we know that $a^{n+m} = a^n a^m$, a^0 must evaluate to the identity element for the operation op. We can extend power to zero exponents by passing the identity element as another parameter:[6]

```
template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power(Domain(Op) a, I n, Op op, Domain(Op) id)
{
    // Precondition: associative(op) ∧ ¬negative(n)
    if (zero(n)) return id;
    return power(a, n, op);
}
```

Project 3.1 Floating-point multiplication and addition are not associative, so may give different results when they are used as the operation for power and power_left_associated; establish whether power or power_left_associated gives a more accurate result for raising a floating-point number to an integral power.

3.5 Parameterizing Algorithms

In power we use two different techniques for providing operations for the abstract algorithm.

1. The associative operation is passed as a parameter. This allows power to be used with different operations on the same type, such as multiplication modulo n.

2. The operations on the exponent are provided as part of the computational basis for the exponent type. We do not choose, for example, to pass half_nonnegative as a parameter to power, because we do not know of a case in which there are competing implementations of half_nonnegative on the same type.

In general, we pass an operation as a parameter when an algorithm could be used with different operations on the same type. When a procedure is defined with an operation as a parameter, a suitable default should be specified whenever possible. For example, the natural default for the operation passed to power is multiplication.

6. Another technique involves defining a function identity_element such that identity_element(op) returns the identity element for op.

Using an operator symbol or a procedure name with the same semantics on different types is called *overloading*, and we say that the operator symbol or procedure name is *overloaded* on the type. For example, + is used on natural numbers, integers, rationals, polynomials, and matrices. In mathematics + is always used for an associative and commutative operation, so using + for string concatenation would be inconsistent. Similarly, when both + and × are present, × must distribute over +. In power, half_nonnegative is overloaded on the exponent type.

When we instantiate an abstract procedure, such as collision_point or power, we create overloaded procedures. When actual type parameters satisfy the requirements, the instances of the abstract procedure have the same semantics.

3.6 Linear Recurrences

A *linear recurrence function of order* k is a function f such that

$$f(y_0, \ldots, y_{k-1}) = \sum_{i=0}^{k-1} a_i y_i$$

where coefficients $a_0, a_{k-1} \neq 0$. A sequence $\{x_0, x_1, \cdots\}$ is a *linear recurrence sequence of order* k if there is a linear recurrence function of order k—say, f—and

$$(\forall n \geqslant k)\, x_n = f(x_{n-1}, \ldots, x_{n-k})$$

Note that indices of x decrease. Given k *initial values* $x_0, \ldots, x_{k-1}$ and a linear recurrence function of order k, we can generate a linear recurrence sequence via a straightforward iterative algorithm. This algorithm requires $n-k+1$ applications of the function to compute x_n, for $n \geqslant k$. As we will see, we can compute x_n in $O(\log_2 n)$ steps, using power.[7] If $f(y_0, \ldots, y_{k-1}) = \sum_{i=0}^{k-1} a_i y_i$ is a linear recurrence function of order k, we can view f as performing vector inner product:[8]

$$\begin{bmatrix} a_0 & \cdots & a_{k-1} \end{bmatrix} \begin{bmatrix} y_0 \\ \vdots \\ y_{k-1} \end{bmatrix}$$

If we extend the vector of coefficients to the *companion matrix* with 1s on its subdiagonal, we can simultaneously compute the new value x_n and

7. The first $O(\log n)$ algorithm for linear recurrences is due to Miller and Brown [1966].
8. For a review of linear algebra, see Kwak and Hong [2004]. They discuss linear recurrences starting on page 214.

shift the old values $x_{n-1}, \ldots, x_{n-k+1}$ to the correct positions for the next iteration:

$$\begin{bmatrix} a_0 & a_1 & a_2 & \cdots & a_{k-2} & a_{k-1} \\ 1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 1 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & 0 \end{bmatrix} \begin{bmatrix} x_{n-1} \\ x_{n-2} \\ x_{n-3} \\ \vdots \\ x_{n-k} \end{bmatrix} = \begin{bmatrix} x_n \\ x_{n-1} \\ x_{n-2} \\ \vdots \\ x_{n-k+1} \end{bmatrix}$$

By the associativity of matrix multiplication, it follows that we can obtain x_n by multiplying the vector of the k initial values by the companion matrix raised to the power $n-k+1$:

$$\begin{bmatrix} x_n \\ x_{n-1} \\ x_{n-2} \\ \vdots \\ x_{n-k+1} \end{bmatrix} = \begin{bmatrix} a_0 & a_1 & a_2 & \cdots & a_{k-2} & a_{k-1} \\ 1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 1 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & 0 \end{bmatrix}^{n-k+1} \begin{bmatrix} x_{k-1} \\ x_{k-2} \\ x_{k-3} \\ \vdots \\ x_0 \end{bmatrix}$$

Using power allows us to find x_n with at most $2\log_2(n-k+1)$ matrix multiplication operations. A straightforward matrix multiplication algorithm requires k^3 multiplications and $k^3 - k^2$ additions of coefficients. Therefore the computation of x_n requires no more than $2k^3 \log_2(n-k+1)$ multiplications and $2(k^3-k^2)\log_2(n-k+1)$ additions of the coefficients. Recall that k is the order of the linear recurrence and is a constant.[9]

We never defined the domain of the elements of a linear recurrence sequence. It could be integers, rationals, reals, or complex numbers: The only requirements are the existence of associative and commutative addition, associative multiplication, and distributivity of multiplication over addition.[10]

The sequence f_i generated by the linear recurrence function

$$\mathsf{fib}(y_0, y_1) = y_0 + y_1$$

of order 2 with initial values $f_0 = 0$ and $f_1 = 1$ is called the Fibonacci sequence.[11] It is straightforward to compute the nth Fibonacci number f_n by using power with 2×2 matrix multiplication. We use the Fibonacci sequence to illustrate how the k^3 multiplications can be reduced for this particular case. Let

9. Fiduccia [1985] shows how the constant factor can be reduced via modular polynomial multiplication.

10. It could be any type that models semiring, which we define in Chapter 5.

11. Leonardo Pisano, *Liber Abaci*, first edition, 1202. For an English translation, see Sigler [2002]. The sequence appears on page 404.

$$F = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

be the companion matrix for the linear recurrence generating the Fibonacci sequence. We can show by induction that

$$F^n = \begin{bmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{bmatrix}$$

Indeed:

$$\begin{aligned} F^1 &= \begin{bmatrix} f_2 & f_1 \\ f_1 & f_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \\ F^{n+1} &= FF^n \\ &= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{bmatrix} \\ &= \begin{bmatrix} f_{n+1} + f_n & f_n + f_{n-1} \\ f_{n+1} & f_n \end{bmatrix} = \begin{bmatrix} f_{n+2} & f_{n+1} \\ f_{n+1} & f_n \end{bmatrix} \end{aligned}$$

This allows us to express the matrix product of F^m and F^n as

$$\begin{aligned} F^m F^n &= \begin{bmatrix} f_{m+1} & f_m \\ f_m & f_{m-1} \end{bmatrix} \begin{bmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{bmatrix} \\ &= \begin{bmatrix} f_{m+1}f_{n+1} + f_m f_n & f_{m+1}f_n + f_m f_{n-1} \\ f_m f_{n+1} + f_{m-1} f_n & f_m f_n + f_{m-1} f_{n-1} \end{bmatrix} \end{aligned}$$

We can represent the matrix F^n with a pair corresponding to its bottom row, (f_n, f_{n-1}), since the top row could be computed as $(f_{n-1} + f_n, f_n)$, which leads to the following code:

```
template<typename I>
    requires(Integer(I))
pair<I, I> fibonacci_matrix_multiply(const pair<I, I>& x,
                                     const pair<I, I>& y)
{
    return pair<I, I>(
        x.m0 * (y.m1 + y.m0) + x.m1 * y.m0,
        x.m0 * y.m0 + x.m1 * y.m1);
}
```

This procedure performs only four multiplications instead of the eight required for general 2×2 matrix multiplication. Since the first element of the bottom row of F^n is f_n, the following procedure computes f_n:

```
template<typename I>
    requires(Integer(I))
I fibonacci(I n)
{
    // Precondition: n ⩾ 0
    if (n == I(0)) return I(0);
    return power(pair<I, I>(I(1), I(0)),
                 n,
                 fibonacci_matrix_multiply<I>).m0;
}
```

3.7 Accumulation Procedures

The previous chapter defined an action as a dual to a transformation. There is a dual procedure for a binary operation when it is used in a statement like

```
x = op(x, y);
```

Changing the state of an object by combining it with another object via a binary operation defines an *accumulation procedure* on the object. An accumulation procedure is definable in terms of a binary operation, and vice versa:

```
void op_accumulate(T& x, const T& y) { x = op(x, y); }
  // accumulation procedure from binary operation
```

and

```
T op(T x, const T& y) { op_accumulate(x, y); return x; }
  // binary operation from accumulation procedure
```

As with actions, sometimes independent implementations are more efficient, in which case both operation and accumulation procedures need to be provided.

Exercise 3.3 Rewrite all the algorithms in this chapter in terms of accumulation procedures.

Project 3.2 Create a library for the generation of linear recurrence sequences based on the results of Miller and Brown [1966] and Fiduccia [1985].

3.8 Conclusions

Algorithms are *abstract* when they can be used with different models satisfying the same requirements, such as associativity. Code optimization depends on equational reasoning; unless types are known to be regular, few optimizations can be performed. Special-case procedures can make code more efficient and even more abstract. The combination of mathematics and abstract algorithms leads to surprising algorithms, such as logarithmic time generation of the nth element of a linear recurrence.

Chapter 4

Linear Orderings

This chapter describes properties of binary relations, such as transitivity and symmetry. In particular, we introduce total and weak linear orderings. We introduce the concept of stability of functions based on linear ordering: preserving order present in the arguments for equivalent elements. We generalize min *and* max *to order-selection functions, such as the median of three elements, and introduce a technique for managing their implementation complexity through reduction to constrained subproblems.*

4.1 Classification of Relations

A *relation* is a predicate taking two parameters of the same type:

$$\begin{aligned}&\mathit{Relation}(\mathsf{R}) \triangleq \\ &\quad \mathit{HomogeneousPredicate}(\mathsf{R}) \\ &\wedge\ \mathsf{Arity}(\mathsf{R}) = 2\end{aligned}$$

A relation is *transitive* if, whenever it holds between a and b, and between b and c, it holds between a and c:

$$\begin{aligned}&\textbf{property}(\mathsf{R} : \mathit{Relation}) \\ &\mathsf{transitive} : \mathsf{R} \\ &\quad \mathsf{r} \mapsto (\forall \mathsf{a}, \mathsf{b}, \mathsf{c} \in \mathsf{Domain}(\mathsf{R}))\,(\mathsf{r}(\mathsf{a}, \mathsf{b}) \wedge \mathsf{r}(\mathsf{b}, \mathsf{c}) \Rightarrow \mathsf{r}(\mathsf{a}, \mathsf{c}))\end{aligned}$$

Examples of transitive relations are equality, equality of the first member of a pair, reachability in an orbit, and divisibility.

A relation is *strict* if it never holds between an element and itself; a relation is *reflexive* if it always holds between an element and itself:

property(R : *Relation*)
strict : R
$\quad r \mapsto (\forall a \in \mathsf{Domain}(R))\, \neg r(a, a)$

property(R : *Relation*)
reflexive : R
$\quad r \mapsto (\forall a \in \mathsf{Domain}(R))\, r(a, a)$

All the previous examples of transitive relations are reflexive; proper factor is strict.

Exercise 4.1 Give an example of a relation that is neither strict nor reflexive.

A relation is *symmetric* if, whenever it holds in one direction, it holds in the other; a relation is *asymmetric* if it never holds in both directions:

property(R : *Relation*)
symmetric : R
$\quad r \mapsto (\forall a, b \in \mathsf{Domain}(R))\, (r(a, b) \Rightarrow r(b, a))$

property(R : *Relation*)
asymmetric : R
$\quad r \mapsto (\forall a, b \in \mathsf{Domain}(R))\, (r(a, b) \Rightarrow \neg r(b, a))$

An example of a symmetric transitive relation is "sibling"; an example of an asymmetric transitive relation is "ancestor."

Exercise 4.2 Give an example of a symmetric relation that is not transitive.

Exercise 4.3 Give an example of a symmetric relation that is not reflexive.

Given a relation $r(a, b)$, there are *derived relations* with the same domain:

$$\mathsf{complement}_r(a, b) \Leftrightarrow \neg r(a, b)$$
$$\mathsf{converse}_r(a, b) \Leftrightarrow r(b, a)$$
$$\mathsf{complement\_of\_converse}_r(a, b) \Leftrightarrow \neg r(b, a)$$

Given a symmetric relation, the only interesting derivable relation is the complement, because the converse is equivalent to the original relation.

A relation is an *equivalence* if it is transitive, reflexive, and symmetric: