Structured Concurrent Programming

Jayadev Misra
The University of Texas at Austin
Austin, Texas 78712, USA
email: misra@cs.utexas.edu

December 4, 2014

To the memory of Edsger Wybe Dijkstra (1930–2002)

Colleague, Mentor, Friend

Preface

One way to avoid a headache is to have no head, so goes a saying in Sanskrit. Concurrency was not a headache as long as we wrote only sequential programs. It started to become a minor headache with the advent of operating systems that managed a card-reader, a printer and a disk simultaneously. Now that we have many million mobile, and potentially interacting, devices, we do have a giant headache.

Concurrent programs are hard to design and understand. Unlike sequential programs in which a programmer assumes the role of an executing machine and reasons about his program step by step, concurrency creates, through forks, many executing threads that can not be followed linearly. Concurrency has traditionally excited a lot of passion, for the intricate programming and verification issues that it throws up.

This book presents our attempt to understand concurrency and develop concurrent programs in a more structured style, using a theory called *Orc*. The work is inspired by earlier works with similar goals, CCS [36], CSP [24] and π-Calculus [38]. Unlike earlier works, the emphasis in Orc is on the combinators that allow us to create larger programs from some given components, a process that can be applied hierarchically. The basic components themselves are not part of the calculus. Orc programming language, based on Orc calculus, has been used to solve a large number typical problems in concurrency.

Acknowledgement First and foremost, most of the credit for the shape of the Orc programming language, including the elegant integration of the Orc calculus with functional programming constructs, goes to David Kitchin. He and Adrian Quark implemented the first version, using Java, that included most of the current standard library. The current version of the language, implemented in Scala, is the work of David Kitchin and John Thywissen; it simplifies the previous implementation and extends it in many directions including the notion of "class". The attractive web interface at `http://orc.csres.utexas.edu` is due to Adrian Quark. I also appreciate the contributions of Bryan McCord in helping develop the documentation and Sidney Rosario for help in implementation.

Tony Hoare was an enthusiastic earlier supporter and mentor for this work, collaborating on the first denotational semantics of Orc. My gratitude to Tony

extends far beyond his support for this work. I have learned a great deal about programming and programming languages since I started this project, thanks to the patient and clear explanations by William Cook. William developed the first operational semantics of Orc and has been a collaborator in this project. Albert Benveniste, of IRISA/INRIA, and Claude Jard, of Ecole Normale Supérieure de Cachan, France, have been our collaborators for a long time. They and their students have extended the concepts of the language, applied the concepts in a variety of practical domains and helped us to reexamine our core beliefs (and prejudices). Jose Meseguer, and his student Musab AlTurki, have implemented the real-time rewriting semantics of Orc [2] in their system `Maude`; interactions with them have been a pleasure.

I am grateful for financial support over the years from an endowment funded by the Schlumberger Foundation. The National Science Foundation has supported this research through awards CCR-0204323 and CCF-0811536.

Austin, Texas J. Misra
December 2012

Contents

Chapter 1

Orchestration

1.1 On Building Large Software Systems

This book is about a theory of programming, called *Orc*, developed by me and my collaborators. The philosophy underlying Orc is that: (1) large programs should be composed out of components, which are possibly heterogeneous (i.e., written in a variety of languages and implemented on a variety of platforms), (2) the system merely *orchestrates* the executions of its components in some fashion but does not analyze or exploit their internal structures, and (3) the theory of orchestration constitutes the essential ingredient in a study of programming.

I am sorry if I have already disappointed the reader. None of the points made above is startling. Building large systems out of components is as old as computer science; it was most forcefully promulgated by Dijkstra in his classic paper on Structured Programming [11] nearly a half century ago. It is the cornerstone of what is known as object-oriented programming [19, 35]. In fact, it is safe to assert that every programming language includes some abstraction mechanism that allows design and composition of components.

It is also well-understood that the internal structure of the components is of no concern to its user. Dijkstra [11] puts it succinctly: "we do not wish to know them, it is not our business to know them, it is our business not to know them!". Lack of this knowledge is essential in order that a component may be replaced by another at a later date, perhaps a more efficient one, without affecting the rest of the program.

Component-based design makes hierarchical program construction possible. Each component itself may be regarded as a program in its own right, and designed to orchestrate its subcomponents, unless the component is small enough to be implemented directly using the available primitive operations of a programming language. Hierarchical designs have been the accepted norm for a very long time.

Where Orc differs from the earlier works is in insisting that programming be a study of composition mechanisms, and *just that*. In this view, system building

consists of assembling components, available elsewhere, using a limited set of combinators. The resulting system could itself be used as a component at a higher level of assembly.

There are few restrictions on components. A component need not be coded in a specific programming language; in fact, a component could be a cyber-physical device or a human being that can receive and respond to the commands sent by the orchestration mechanism. Components may span the spectrum in size from a few lines of code, such as to add two numbers, to giant ones that may do an internet search or manage a database. Time scales for their executions may be very short (microseconds) to very long (years). The components may be real-time dependant. A further key aspect of Orc is that the orchestrations of components may be performed concurrently rather than sequentially.

We advocate an *open* design in which only the composition mechanisms are fixed and specified, but the components are not specified. Consequently, even primitive data types and operations on them are not part of the Orc calculus. Any such operation has to be programmed elsewhere to be used as a component. By contrast, most traditional designs restrict the smallest components to the primitives of a fixed language, which we call a *closed* design. Closed designs have several advantages, the most important being that a program's code is in a fixed language (or combinations of languages) and can be analyzed at any level of detail. The semantics of the program is completely defined by the semantics of the underlying programming language. It can be run on any platform that supports the necessary compiler. Perhaps the most important advantage is that the entire development process could be within the control of a team of individuals or an organization; then there are fewer surprises. In spite of these advantages for a closed system design, we do not believe that this is the appropriate model for large-scale programming in the future; we do not believe that a single programming language or a set of conventions will encompass the entirety of a major application; we do not believe that a single organization will have the expertise or resources to build very large systems from scratch, or that a large program will run on a single platform.

The second major aspect of Orc is on its insistence on concurrency in orchestration. Dijkstra [11] found it adequate to program with three simple sequential constructs, sequential composition, a conditional and a looping construct [1] . However, most modern programming systems, starting from simple desktop applications to mobile computing, are explicitly or implicitly concurrent. It is difficult to imagine any substantive system of the future in purely sequential terms.

We advocate concurrency not as a means to improving the performance of execution by using multiple computers, but for ease in expressing interactions among components. Concurrent interactions merely specify a large number of alternatives in executing a program; the actual implementation may indeed be sequential. Expressing the interactions in sequential terms often limits the

[1] Dijkstra did not explicitly include function or procedure definition. This was not essential for his illustrative examples. In his later work, he proposed non-deterministic selection using guarded commands [12, 14] as a construct, though concurrency was not an explicit concern.

options for execution as well as making a program description cumbersome. Components may also be specified for real time execution, say in controlling cyber-physical devices.

Almost all programming is sequential. Concurrency is essential but rarely a substantial part of programming. There will be a very small part of a large program that manages concurrency, such as arbitrating contentions for shared resource access or controlling the proliferation (and interruption) of concurrent threads. Yet, concurrency contributes mightily to complexity in programming. Sprinkling a program with concurrency constructs has proven unmanageable; the scope of concurrency is often poorly delineated, thus resulting in disaster in one part of a program when a different part is modified. *Concurrent program testing can sometimes show the presence of bugs and sometimes their absence.* It is essential to use concurrency in a disciplined manner. Our prescription is to use sequential components at the lowest-level, and orchestrate them, possibly, concurrently.

In the rest of this chapter, we argue the case for the orchestration model of programming, and enumerate a specific set of *combinators* for orchestration. These combinators constitute the Orc calculus, which we introduce informally in Chapter 2 and more formally in Chapter 3. Orc calculus, analogous to the λ-calculus, is not a suitable programming language. Chapter 4 includes a very small programming language built upon the calculus; we ought to call it a "notation" rather than a "language" because it is designed to explore the nature of component integration, rather than build industrial-scale applications. The rest of the book describes our experience in using this notation to code a variety of common programming idioms and some small applications.

1.2 Structure of Orc

1.2.1 Components, also known as *Sites*

Henceforth, we use the term *site* for a component [2].

The notion of a (mathematical) function is fundamental to computing. Functional programming, as in ML [37] or Haskell [15], is not only concise and elegant from a scientist's perspective, but also economical in terms of programming cost. Imperative programming languages often use the term "function" with a broader meaning; a function may have side-effects. A site is an even more general notion. It includes any program component that can be embedded in a larger program, as described below.

The starting point for any programming language is a set of primitive built-in operations or services. Primitive operations in typical programming languages are arithmetic and boolean operations, such as "add", "logical or" and "greater than". These primitive operations are the givens; new operations are built from the primitive ones using the constructs of the language. A typical language has

[2] This terminology is a relic of our earlier work in which web services were the only components. We use "site" more generally today for any component.

a fixed set of primitive operations. By contrast, Orc calculus has no built-in primitive operation. Any program whose execution can be initiated, and that responds with some number of results, may be regarded as a primitive operation, i.e. a site, in Orc.

The definition of site is broad. Sites could be primitive operations of common programming languages, such as the arithmetic and boolean operations. A site may be an elaborate function, say, to compress a jpeg file for transmission over a network, or to search the web. It may return many results one by one, as in a video-streaming service or a stock quote service that delivers the latest quotes on selected stocks every day. It may manage a mutable store, such as a database, and provide methods to read from or write into the database. A site may interact with its caller during its execution, such as an internet auction service. A site's execution may proceed concurrently with its caller's execution. A site's behavior may depend on the passage of real time.

We regard humans as sites for a program that can send requests and receive responses from them. For example, a program that coordinates the rescue efforts after an earthquake will have to accept inputs from the medical staff, firemen and the police, and direct them by sending commands and information to their hand-held devices. Cyber-physical devices, such as sensors, actuators and robots, are also sites.

Sites may be higher-order in that they accept sites as parameters of calls and produce sites as their results. We make use of many factory sites that create and return sites, such as communication channels. Orc includes mechanisms for defining new sites by making use of already-defined sites.

1.2.2 Combinators

The most elementary Orc *expression* is simply a site call. A combinator combines two expressions to form an expression. The results published by expressions may be bound to immutable variables. There are no mutable variables in Orc; any form of mutable storage has to be programmed as a site.

Orc calculus has four combinators: "parallel" combinator, as in `f|g`, executes expressions `f` and `g` concurrently and publishes whatever either expression publishes; "sequential" combinator, as in `f >x> g`, first executes `f`, binds each of its publications to a different instance of variable `x` and then executes a separate instance of `g` for each such binding; "pruning" combinator, as in `f <x< g`, executes `f` and `g` concurrently, binds the first value published by `g` to variable `x` and then terminates `g`, here `x` may appear in `f`; and "otherwise" combinator, as in `f;g`, introduces a form of priority-based execution by first executing `f`, and then `g` only if `f` halts without publishing any result.

There is one aspect worth noting even in this very informal description. An expression may publish multiple values just as a site does. For example, if each of `f` and `g` publishes a single value, then `f|g` publishes both of those values; and `(f|g)>x> h` executes multiple instances of expression `h`, an instance for each publication of `f|g`. Informal meanings of the combinators are given in Chapter 2 and the formal meanings in Chapter 3.

1.2.3 Consequences of Pure composition

The combinators for composition are agnostic about the components they combine. So, we may combine very small components, such as for basic arithmetic and boolean operations drawn from a library, to simulate the essential data structures for programming. This, in turn, allows creations of yet larger components, say for sorting and searching. Operations to implement mutable data structures, such as for reading or writing to a memory location, can also be included in a program library. A timer that operates in real time can provide the basics for real time programming. Effectively, a general purpose concurrent programming language can be built starting with a small number of essential primitive components in a library. This is the approach taken in the Orc language design in Chapter 4.

Even though it is possible to design any kind of component starting with a small library of components, we do not advocate doing so in all cases. The point of orchestration is to reuse components wherever possible rather than building them from scratch, and components built using Orc may not have the required efficiency for specific applications.

1.3 Concluding Remarks

There is a popular saying that the internet is the computer. That is no less or no more true than saying that a program library is a computer. This computer remains inactive in the absence of a driving program. Orc provides the rudiments of a driving program. It is simultaneously the most powerful language that can exploit available programs as sites, and the least powerful programming language in the absence of sites.

A case against a grand unification theory of programming It is the dream of every scientific discipline to have a grand unification theory that explains all observations and predicts all experimental outcomes with accuracy. The dream in an engineering discipline is to have a single method of constructing its artifacts, cheaply and reliably. For designs of large software systems, we dream of a single, preferably small, programming language with an attendant theory and methodology that suffices for the constructions of concise, efficient and verifiable programs. As educators we would love to teach such a theory.

Even though we have not realized this dream for all domains of programming, there are several *effective theories* for limited domains. Early examples include boolean algebra for designs of combinational circuits and BNF notation for syntax specification of programming languages. Powerful optimization techniques have been developed for relational database queries. Our goal is to exploit the powers of many limited-domain theories by combining them to solve larger problems. A lowest-level component should be designed very carefully for efficiency, employing the theory most appropriate for that domain, and using the most suitable language for its construction. Our philosophy in Orc is to

recognize and admit these differences, and combine efficient low-level components to solve a larger problem. Orc is solely concerned with how to combine the components, not how a primitive component should be constructed.

Bulk vs. Complexity It is common to count the number of lines of code in a system as a measure of its complexity. Even though this is a crude measure, we expect a system with ten times as many lines of code to be an order of magnitude more complex. Here we are confusing *bulk* with *complexity*; that bulkier a program, the more complex it is. There are very short concurrent programs, say with about 20 lines, that are far more complex than a thousand line sequential program. Concurrency adds an extra dimension to complexity. In a vague sense, the complexity in a sequential program is additive, whereas in a concurrent program it is multiplicative.

The philosophy of Orc is to delegate the bulkier, but less complex parts to components and reserve the complexity for the Orc combinators. Though solvers of partial differential equations can be coded entirely in Orc using the arithmetic and boolean operations as sites, this is not the recommended option. It should be coded in a more suitable language, but concurrent executions of multiple instances of the solvers, with different parameters, for instance, should be delegated to Orc.

Some sweeping remarks about programming Consider the following scenario. A patient receives an electronic prescription for a drug from a doctor. The patient compares prices at several near-by pharmacies, and chooses the cheapest one to fill the prescription. He pays the pharmacy and receives an electronic invoice which he sends to the insurance company for reimbursement with instructions to deposit the amount in his bank account. Eventually, he receives a confirmation from his bank. The entire computation is mediated at each step by the patient who acquires data from one source, does some minor computations and sends data to other sources.

This computing scenario is repeated millions of times a day in diverse areas such as business computing, e-commerce, health care and logistics. In spite of the extraordinary advances in mobile computing, human participation is currently required in every major step in most applications. This is not because security is the over-riding concern, but that the infrastructure for efficient mediation is largely absent, thus contributing to cost and delay in these applications. We believe that humans can largely be eliminated, or assigned a supporting role, in many applications. Doing so is not only beneficial in terms of efficiency, but also essential if we are to realize the full potential of the interconnectivity among machines, using the services and data available in the internet, for instance. We would prefer that humans advise and report to the machines, rather than that humans direct the machines in each step.

The initial impetus for Orc came from attempting to solve such problems by orchestrating the available web services. Ultimately, languages outgrow the initial motivations of their design and become applicable in a broader domain. Orc

is currently designed for component integration and concurrency management in general.

The programming community has had astonishing success in building large software systems in the last 30 years. We routinely solve problems today that were unimaginable even a decade ago. Our undergraduates are expected to code systems that would have been fit for a whole team of professional programmers twenty years ago. What would programs look like in the future? We can try to interpolate. The kinds of problems the programmers will be called upon to solve in the next two decades will include: health care systems automating most of their routine tasks and sharing information across hospitals and doctors (for example, about adverse reaction to drugs); communities and organizations sharing and analyzing data and responding appropriately, all without human intervention; disaster recovery efforts, including responding to anticipated disasters (such as, shutting down nuclear reactors well before there is a need to) being guided by a computer; the list goes on. These projects will be several orders of magnitude larger than what we build today. We anticipate that most large systems will be built around orchestrations of components. For example, a system to run the essential services of a city will not be built from scratch for every city, but will combine the pre-existing components such as for traffic control, sanitation and medical services. Software to manage an Olympic game will contain layer upon layers of interoperating components.

A Critique of Pure Composition A theory such as Orc, based as it is on a single precept, may be entirely wrong. It may be too general or too specific, it may prove to be too cumbersome to orchestrate components, say in a mobile application, or it may be suitable only for building rapid prototypes but may be too inefficient for implementations of actual systems. These are serious concerns that can not be argued away. We will try to address these issues in this book in two ways: (1) prove results about the calculus, independent of the components, that will establish certain desirable theoretical properties, and (2) supply enough empirical evidence that justifies claims about system building. While (1) is largely achievable (see Chapter 3), (2) is a never-ending task. We supply empirical evidence in this book by programming a large number of commonly occurring programming patterns. We hope that the evidence presented in this book will spur others to investigate Orc, and add to the body of evidence for or against it.

Chapter 2

Orc Calculus

2.1 Introduction

This chapter introduces the *Orc calculus*, the foundation of our theory of structured concurrent programming. The essential elements of the calculus are: (1) *site*, (2) *combinator* and (3) *definition*. A site is a service of any kind that a program can call and from which it may receive responses, combinators provide the mechanisms for orchestrating the site calls, and the definitions provide the essential abstraction mechanism for defining sites in Orc. The calculus does not include data types, or typical concurrent programming notions such as thread, process and synchronization; such concepts can be programmed from the constructs of the calculus.

We describe the calculus informally in this chapter and more formally in Chapter 3.

2.2 Site

Summary A site is a service that is called like a traditional procedure and it responds with some number of values, called *publications*. It may, optionally, send an indication when its computation on behalf of this call has ended. Thus, there are three kinds of responses: (1) a *non-terminal* response that has an associated value, (2) a *terminal response with a value* indicating that the computation has ended, so there will be no further response, and (3) a *terminal response without a value* indicating just the end of computation, but no value. A response is *positive* if it carries a value; it could be terminal or non-terminal. A *negative* response carries no value, it is always terminal. A *helpful* site is one that sends a terminal response whenever its computation on behalf of a call ends. □

We next elaborate on the nature of site call and response.

A site is any entity that can accept input from a program and respond to it. A site may be a program component, a cyber-physical device or a human being

who can communicate with a computer.

A site is *called* in order to trigger its execution and the site *responds* by returning values. First, we describe the specifics of call and response and then show that various kinds of more general interactions may be simulated through the call/response protocol.

2.2.1 Site Call

A site call looks like a procedure or function call in other programming languages. A site call has zero or more parameters[1] , where each parameter is a variable, and the list of parameters are comma-separated within parentheses. If the parameter list is empty the parentheses are still used, as in `M()`. A site may have a variable number of parameters; in fact, the library site that constructs a tuple out of its arguments may have any number of arguments exceeding 1.

A site call is executed even when some of its parameters are not bound to values. We call this form of execution *lenient*. As each parameter gets bound to a value, the value is transmitted to the site. A site may respond even before receiving values of all its parameters.

We call a site *strict* if it needs the values of all its parameters before it can start execution. Though all sites are called leniently, a site may specify that its behavior is strict. For example, a strict site for "logical-or" needs values of both its boolean arguments before it sends its response. A lenient site for the same function may respond with `true` if one of the argument values is `true` even though the other argument may not yet be bound to a value.

A site may be called concurrently, possibly from many parts of the same program. The site may execute the calls it receives in some sequential order, or it may execute multiple instances of its code concurrently. Concurrent execution introduces the possibility that the different execution threads may interfere in reading/writing to shared mutable store. How a site handles concurrent calls is a part of its specification.

More Elaborate Call Protocols We sketch how more elaborate site call protocols may be implemented using the Orc protocol.

Typically, large components do not have a single interface. A site may provide a variety of *methods* that may be called directly. This is supported directly in the Orc language. It can be simulated in the Orc calculus by the user (i.e., the orchestrating program) first calling a site with a method name as a parameter; the site responds with the name of another site that performs the actions of that method, and then the user may call that site with the appropriate parameters of that method.

Many sites engage in a dialog with a user before they accept the proper input from the user. For example, to interact with a bank, the user first inputs the

[1] Henceforth, we use the terms "argument" and "parameter" interchangeably. A "formal parameter" is the name of the variable being used for a parameter and "actual parameter" is the value of the formal parameter in a call.

"user name", the user receives a pass-phrase by which he authenticates the site, the user then inputs the "password", and the site responds by allowing the user to access the account. We can think of this protocol as being equivalent to a series of smaller steps: the user calls the bank with "user name" as a parameter, receives a response that includes a pass-phrase and another site name A, user authenticates the pass-phrase by calling another site B, then the user calls A with the password and expects to receive a site name C for access to the account.

A site may engage in continual interactions with its caller during its execution. These are known as reactive programs, first introduced by Manna and Pnueli [33]. We may imagine that the site interacts with its caller through input and output channels; the initial call to the site establishes the identities of these channels. Sites may also interact through websites whose identities can be similarly established.

2.2.2 Site Response

A site responds to a call by sending some number (zero, non-zero finite or infinite) of values; optionally, it may also send an indication when its computation on behalf of this call has ended. We say that the site *publishes* each value, and the values are the *publications* of the site. The termination indication from a site, if any, allows the caller to stop waiting for further publications. Note that a site may send a termination indication without sending any publication when the computation ends without any output by the site. Further, many sites, particularly the external ones, never send a termination indication when their computation on behalf of a call terminates.

Sites may follow a variety of protocols to send their responses to callers. We abstract from these protocols and adopt the convention that a site sends three kinds of responses: (1) a *non-terminal* response that has an associated value, (2) a *terminal response with a value* indicating that the computation has ended, so there will be no further response, and (3) a *terminal response without a value* indicating just the end of computation, but no value. Call a response *positive* if it carries a value; it could be terminal or non-terminal. A *negative* response does not carry a value, and it is always terminal.

We require of an implementation that a terminal response be received last by the caller. An implementation may ensure this requirement by having each terminal response carry the count of the number of non-terminal responses preceding it so that the caller processes all non-terminal responses before processing the terminal response.

The non-terminal responses from a site are received by the caller in arbitrary order; so they form a set, not a sequence. If the site intends to send a stream of values, as for a video file, it should assign sequence numbers to the individual responses and have the caller reassemble the responses into a stream. The end of the stream is a terminal response.

Helpful Site Not all sites send the termination indication. A site is *helpful for a call* if its computation on behalf of the call is infinite or it sends a terminal

response (at the end of computation). A site is *helpful* if it is helpful for every call. Typical web services that are called as sites are not expected to be helpful.

Parameter and Response Values There are no designated data types or special values in Orc, except the value **`signal`** that carries no data. A signal may be a parameter value in a site call or published by a site. A site may publish any value, such as integers, lists and trees, XML records and Java objects, that are merely passed on to other sites by the orchestrating program. A site is also treated as a value; so, the actual parameters of a site call may themselves be sites and the publications of a site may also be sites.

Site Implementation It is irrelevant for our theory how a site is implemented, or the servers on which the computations of a site take place. It is possible that a site is down-loadable —as is the case with most Java applets— which causes a site call to result in a download and the execution of the application on the client's machine. More elaborate schemes for migration and execution may be implemented for certain sites. These details are of no concern in the theory of Orc; they are relevant only for efficient implementation.

More on Site Response Sites are not just mathematical functions, nor are they quite comparable to procedures in traditional languages that publish at most one value. We give a few examples of site responses.

A site may send only a single terminal response with a value that is a mathematical function of its arguments; logical-or is such a site. A site's response could be non-deterministic, as in a random number generator; standard library site `Random(n)` sends a single terminal response with a value that is a random natural number smaller than `n`. A site may send only a negative response; library site **`stop`** has this behavior (see below). A site may publish for some argument values, but not for others; see **`Ift`** below. A site call may be blocked waiting for another site call to change the site state; reading from an empty buffer is blocked until another site call writes some data into the buffer. A site may send a very large number of positive responses, as in a video-streaming service. A site's response may depend on the passage of time: a site that publishes an "alarm" signal at a specific time every day has an unbounded number of publications each of which is time-dependent.

2.2.3 Common Sites

Sites are treated as uninterpreted symbols in the Orc calculus. We introduce a few common sites below for use in our examples. These sites are easy to code in any common programming language. All of them are available in the Orc standard library. Each site publishes a single terminal response, so it is helpful. Each site serializes executions of concurrent calls in some arbitrary order.

`Ift(b)`, **`Iff(b)`**: Site **`Ift(b)`** publishes a signal immediately if `b` is (bound to) `true`, and sends a negative response if `b` is `false` or non-boolean. Site

Iff has the opposite behavior; it publishes a signal if `b` is `false`, and sends a negative response otherwise.

stop and **signal**: **stop** is an abbreviation for **Iff**`(true)`, and **signal** is same as **Ift**`(true)`. Thus, **stop** does not publish and **signal** immediately publishes a signal. Technically, **stop** is not a site but a primitive of the calculus. The distinction is important in semantic description.

Constants: It is convenient to imagine that each constant value is actually a site, a site of zero arity. Thus, there is a site corresponding to each integer and real constant, such as `3`, `-1`, `2.5`, and `3.0E-2`, that we wish to represent in a computer. Call to any such site responds immediately with a non-terminal response with the corresponding value. Similarly, boolean values `true` and `false` are published by the sites `true` and `false` respectively, and Unicode strings, such as `"37wxy.pq"`, are published by the corresponding sites. We adopt the convention that calls to constant sites are written without parentheses so that they resemble the familiar literals; for example, we write just `3` instead of `3()` to call the corresponding site.

Any implementation would clearly use a different technique for representing constants, rather than employong sites. Representation by sites unifies their treatment in the Orc calculus.

`Rwait(t)`: A call to `Rwait(t)`, where `t` is bound to a non-negative integer value, publishes a signal, a terminal response, exactly[2] `t` time units later. Henceforth, we take a time unit to be 1 millisecond. Observe that `Rwait(t)`, where `t` is bound to 0, behaves as **signal**.

2.3 Orc program Structure

An Orc program is an expression. An expression is either (1) a site call, (2) two constituent expressions combined using a *combinator*, or (3) a site definition followed by an expression. This structure supports repeated and nested site definitions which we illustrate in full generality in Chapters 3 and 4. We restrict ourselves to definitions without nesting in this chapter, which is sufficient to explain the main ideas of Orc.

From the syntax of expressions, any expression is preceded by a, possibly empty, sequence of definitions. The expression following all the definitions in a program is called its *goal expression.* A program execution starts by executing its goal expression.

An expression engages in two possible activities during its execution: (1) site calls and (2) publications. Its execution may (1) may halt eventually having published a finite number of values (possibly zero), (2) continue forever publishing a finite or infinite number of values, or (3) blocked indefinitely waiting for responses from some sites. An expression *halts* if it can engage in no

[2] An implementation can only approximate this guarantee.

further computation, that is: (1) it has received all responses from every site it has called (guaranteed by receiving a terminal response from each called site), and (2) it will not call any site nor publish any further value. An expression that never publishes is *silent*, though it may still call sites. We define these notions formally in Section 3.7.1 (page 65).

The most elementary expression is a site call. Any value associated with a positive response of the site becomes a publication of the expression.

Orc has four combinators: parallel (`|`), sequential (`>x>`), pruning (`<x<`), and otherwise (`;`). Each combinator captures a different aspect of concurrency. Syntactically, the combinators are written infix. Their order of precedence from stronger to weaker binding power is as follows: sequential, parallel, pruning, otherwise. Thus, `f >x> M(x,y)<y< h|Rwait(5); 3` is same as `((f >x> M(x,y))<y< (h|Rwait(5))); 3`.

Orc programs have no mutable variables. A mutable store has to be managed by a site written in some other language. The Orc program can manipulate such a store only by calling the site, but never directly.

Execution Engine We assume that an Orc program is executed on a computer that can run an arbitrary number of concurrent computations simultaneously, and that provides a real time-service so that the same real time is visible to all computations. Clearly, this is an idealized model; an implementation can only approximate this ideal. In a later chapter, we show how the computation of an Orc program can be distributed over a set of machines while providing the illusion of the single time-service. The idealized computing model simplifies the description of Orc semantics.

2.3.1 Parallel Combinator

Given that `f` and `g` are Orc expressions, `f|g` is executed by starting the executions of `f` and `g` immediately and concurrently. Executions of `f` and `g` are interleaved arbitrarily in the execution of `f|g`. Any site call (or publication) by a constituent expression, `f` or `g`, is also a site call (or publication) by `f|g`. There is no direct communication between `f` and `g` during the execution.

Example

Expression `2|3` calls the sites `2` and `3` simultaneously. Since each of these sites publishes the corresponding value immediately, expression `2|3` publishes 2 and 3 immediately, in either order.

Expression `Google(x)|Yahoo(x)` calls the sites `Google` and `Yahoo` simultaneously. □

The parallel combinator is commutative and associative, that is, `f|g` is equivalent to `g|f` and `(f|g)|h` to `f|(g|h)`. We prove these results in Chapter 3 using the operational semantics of Orc.

2.3.2 Sequential Combinator

In `f >x> g`, first the execution of `f` is started. Each value published by `f` initiates a separate execution of `g` wherein `x` is bound to that published value. Execution of `f` continues in parallel with the executions of all instances of `g`. The publications of the instances of `g` are the publications of `f >x> g`. The values published by `f` are internally consumed by being bound to the various instances of `x`. If `f` publishes no value, no execution of `g` occurs, and `f >x> g` does not publish.

Example

Expression `2 >x> Double(x)` is executed as follows. First, site `2` is called and it publishes `2`. Then the published value `2` is bound to `x` and site `Double` is called with parameter `x`. Suppose site `Double` publishes double the value of its parameter; then, the entire expression publishes `4`.

In expression `(2|3)>x> Double(x)`, since `2|3` publishes 2 and 3 in either order, site `Double` is called twice with parameter `x` bound to `2` and `3`. Therefore, the entire expression publishes `4` and `6` in either order.

In `( CNN()|BBC())>x> Email(a, x)`, sites `CNN` and `BBC` may each publish a news page or send no response. Assume that site call `Email(a, x)` sends the value of `x` by email to address `a`, where `a` is already bound to a value. Since either or both of `CNN` and `BBC` may or may not publish, `Email` may be called 0, 1 or 2 times depending on the number of responses received from `CNN` and `BBC`.

`stop` `>x> 2` behaves as **`stop`**; it publishes nothing and halts. We show in Section 3.7.3 (page 67) that **`stop`** `>x> g`, for any expression `g`, is equivalent to **`stop`**. Similarly, **`signal`** `>x> g` is equivalent to `g`, though, technically, this is called a weak equivalence. □

Notational Convention Exploiting the sequential combinator, we write constants instead of just variables for parameters in a site call. Thus, writing `M(2,y)`, where `M` is a site, is a shorthand for `2 >x> M(x,y)` and `M(2,3)` is `2 >x> (3 >y> M(x,y))`.

We adopt the convention that the sequential combinator is right associative, i.e. `f >x> g >y> h` is interpreted as `f >x> (g >y> h)`. When `x` is not used in `g`, we use the short-hand `f >> g` for `f >x> g`. □

The sequential combinator generalizes the traditional sequential composition for a concurrent world: if `f` just publishes a signal when it has completed its execution, then `f >> g` behaves like a program in which `g` follows `f`.

2.3.3 Pruning Combinator

The two combinators described so far can only spawn new computations, but never terminate an executing computation. The next combinator allows us to

do just that. In `f <x< g`, both `f` and `g` are started concurrently. Recall that site calls are lenient; so, the site calls in `f` that have `x` as a parameter can proceed even though `x` is not bound to a value. If `g` publishes a value, then `x` is bound to that value and `g`'s execution is terminated. Terminating the execution of expression `g` has the effect that no further site calls are made from `g` and a response received later from a site that was called from `g` is ignored.

In contrast to `( CNN()|BBC())>x> Email(a, x)`, the following expression calls `Email` at most once. Below, `a` is already bound to a value.

```
Email(a, x) <x< ( CNN()|BBC() )
```

Here, executions of `Email(a, x)` and `CNN()|BBC()` are started simultaneously. The site call `Email(a, x)` is made even though `x` is not bound to a value, though the execution of `Email` would be suspended until `x` is bound. The execution of `CNN()|BBC()` calls `CNN` and `BBC` simultaneously. As soon as a value is received from either site call, the value is bound to `x`, execution of `CNN()|BBC()` is terminated, and `Email`'s execution resumes. If one of `CNN()` or `BBC()` subsequently publishes a value, it is ignored. If `CNN()|BBC()` never publishes, `Email`'s execution is permanently suspended.

Notational Convention By convention, the pruning combinator is left associative, i.e., `f <x< g <y< h` is interpreted as `(f <x< g)<y< h`. When `x` is not used in `f`, we use the short-hand `f << g` for `f <x< g`.

Parallelism in the Pruning combinator Execution of `f <x< g` starts concurrent executions of `f` and `g`. Therefore, `(f <x< g)<y< h` starts concurrent executions of `f <x< g` and `h`, i.e., concurrent executions of all the component expressions, `f`, `g` and `h`. Orc programming language exploits this feature of the pruning combinator to introduce concurrency in a program without explicit programmer intervention.

2.3.4 Otherwise Combinator

Otherwise combinator exploits halting of expressions. In `f;g`, execution of `f` is first started. If `f` halts and it has not published during its execution, then `g` is started. If `f` ever publishes a value, then `g` is ignored.

This combinator is particularly useful in programming with helpful sites, introduced in Section 2.2.2 (page 21). Recall that a helpful site responds with a terminal response when its computation ends. Halting of a helpful expression, i.e., one that calls only helpful sites, can be detected from the terminal responses received from those sites. A general site either responds or stays unresponsive forever; in the latter case, the caller waits forever and the halting can not be detected. So, this combinator is not particularly useful if such a site is called from `f`.

All "Common Sites" introduced in Section 2.2 (page 19) are helpful (though they may be blocked indefinitely waiting for some condition to hold, as in reading from an empty channel). For example, **`Ift(b)`** either publishes a terminal

response with an associated signal value if `b` is `true`, or sends a terminal response without any value, a negative response, if `b` has any other value.

Example

Expression `1;2` publishes value 1. The second expression, `2`, is ignored after `1` publishes. And, `stop;2` publishes 2.

In `( Ift(b); N())>x> Email(a, x)`, an email is sent to `a` if `b` is `true` or if `N` responds. We do not assume that `N` is helpful.

Let `f` be an expression that halts (without publishing a value); it merely calls some sites. Then `f ; g` results in the sequential execution of `f` followed by `g`.

Notational Convention The otherwise combinator will be shown to be associative in Section 3.7.3. So, we abbreviate both `(f ; g); h` and `f ; (g ; h)` by `f ; g ; h`.

2.4 Site Definition

A site definition in Orc is written as `def E(flist)= f`; here `E` is the name of the site being defined, `flist` its list of formal parameters and `f` its *body*, which is an expression. Site definitions may be recursive.

Given a definition `def E(flist)= f`, a call `E(alist)` executes the body `f`, called the *goal* expression of the definition, with the actual parameters `alist` substituted for the formal parameters `flist`. A site call publishes every value published by the execution of its body. If multiple concurrent calls are made to site `E`, all instances of `f` execute concurrently. Concurrent execution causes no difficulty unless the instances read/write mutable store; if they do, the executions of different instances may interfere with possibly disastrous consequences.

An expression that includes a definition `D` followed by expression `g` is written as `D # g` where `#` is a separator symbol. The separator `#` is optional, i.e., it can be replaced with a white space if the next non-white space is an alphanumeric symbol. We omit it wherever possible, though we sometimes use it for readability even though it is optional.

Orc implementation guarantees that a site defined in Orc that calls only helpful sites is helpful.

Example

```
--The logical ''not'' function.
def not(x) = Iff(x) >> true | Ift(x) >> false

-- The logical ''or'' function.
def or(x,y) =
     Ift(x) >> true
```

```
    | Ift(y) >> true
    | Iff(x) >> Iff(y) >> false
```

Observe that `or(x,y)` may start publishing even before both of its arguments are bound to values; for example, if `x` is bound to `true`, the execution of `Ift(x)>> true` will publish a value.

Site `or(x,y)` publishes two values when both `x` and `y` are `true`; in all other cases it publishes just one value. We can force publication of just one value by using the pruning combinator.

```
def or(x,y) =
   b <b<
     (  Ift(x) >> true
      | Ift(y) >> true
      | Iff(x) >> Iff(y) >> false
     )
```

or, more simply when both x and y are bound:

```
def or(x,y) =
     Ift(x) >> true ; Ift(y) >> true ; false
```

2.4.1 Closure

Defining a site creates a special value called a *closure*; the name of the site is a variable and its bound value is the closure. A closure is treated like any other value; it may be published or passed as an argument in a call.

As an example, consider the following definition of site `apply` where `M` is a closure. Site `M` accepts two parameters. Calling `apply(M,x,y)` has the same effect as calling `M(x,y)`.

```
def apply(M,x,y) = M(x,y)
```

Then, `apply(Times,2,3)` publishes 6 and `apply(or,true,false)` publishes `true`.

We treat a more substantial example now. The site call `Rwait(t)` publishes a signal after `t` milliseconds. Suppose we want a program to publish a signal after a very long period, perhaps a year. The program may compute the waiting time in milliseconds and wait for that length of time. The computed time may be a very large number, possibly overflowing a computer word. Therefore, we adopt a different strategy. Define 4 auxiliary sites: `Rwaits`, `Rwaitm`, `Rwaith` and `Rwaitd`. These sites publish a signal after a second, minute, hour and day, respectively. Then, we define site `longwait` that has as argument the number of days, hours, minutes and seconds, and it publishes a signal after this amount of time has elapsed.

We simplify the program by first we defining the site `repeat` that takes a closure `f` and a non-negative integer `n` as arguments. Closure `f` has no arguments and it publishes a signal after completion of its execution. Site `repeat` executes `f n` times in succession and then publishes a signal. Site `sub(x,y)` returns `x-y`, and `equals` returns `true` if its two arguments are equal and `false` otherwise.

```
def repeat(f,n) =
  (Ift(b) >> signal ; f() >> repeat(f,m) )
    <b< equals(n,0)
    <m< sub(n,1)
```

Next, we define the auxiliary sites, and `longwait` as a sequence of calls to `repeat`.

```
def Rwaits() =  Rwait(1000)
def Rwaitm() =  repeat(Rwaits,60)
def Rwaith() =  repeat(Rwaitm,60)
def Rwaitd() =  repeat(Rwaith,24)

def longwait(d,h,m,s) =
    repeat(Rwaitd,d)
 >> repeat(Rwaith,h)
 >> repeat(Rwaitm,m)
 >> repeat(Rwaits,s)
```

The execution of `longwait` calls `repeat` every second, thus incurring an overhead that will skew the computation of accumulated time over a long period. It may be preferable to define `def Rwaitm()= Rwait(60000)`, so that `repeat` is called every minute.

Why Site Calls are lenient We expect that a call to a site can be replaced by the body of the definition with appropriate substitutions of actual parameters for formal parameters. This replacement should be valid in all contexts. In particular, given the definition `def E(x)= f`, we should expect that `E(x)<x< g` and `f <x< g` will be the same program. In `f <x< g`, execution of `f` is started even if `x` is not bound. Therefore, execution of `E(x)` should also start under the same conditions, that is, the call to `E` must be lenient.

2.5 Examples

We have already seen a few small examples in connection with each combinator. The goal of this section is to solve several standard examples from concurrent, as well as sequential, programming. Despite the austerity of Orc's combinators, we are able to encode a variety of idioms concisely. The examples are necessarily small, and the solutions often awkward, in the absence of a proper programming language. We will redo some of these examples in Chapter 4 after introducing the Orc programming language.

We introduce appropriate sites to solve each example. Some of the sites are of general utility; they can be coded in any standard programming language, and they are also available in the Orc standard library (sites in the standard library are coded in Java or Scala). We also introduce domain-dependent sites to solve specific examples.

2.5.1 Common Idioms

2.5.1.1 Sequencing

The most common programming idiom is to execute a sequence of steps. Imperative programming languages use a constructor, typically written with a semicolon, `f ; g`, to sequence the given computations. In typical imperative programming, computation of `f` has the side effect of modifying certain data structures and `f` only publishes a signal on termination of its computation. In functional programming, sequencing is function composition, where the inner function `f` publishes a value that is supplied to the outer function `g`.

Orc includes the more general combinator `>x>` to sequence steps. In the expression `f >x> g`, `f` may cause side effects, as in an imperative program, by calling sites and it may publish an arbitrary number of values, not just signal. If `f` always publishes a single signal and then halts, then `f >x> g` implements the sequencing of `f` and `g`.

2.5.1.2 Conditional And

A common programming pattern is to first evaluate a boolean b and only if it is `true` evaluate boolean b'. The result of the evaluation is `true` if both b and b' evaluate to `true`, `false` if b evaluates to `false` or if b evaluates to `true` and b' to `false`. For example, in searching over a boolean array `p` whose highest index is n, we may first check that index `i` is within bounds using `i <= n` and then evaluate `p(i)`. It is inadmissible to call a site that performs the conjunction of `i <= n` and `p(i)` because `p(i)` may be evaluated first when `i > n`.

The following expression achieves the goal.

```
Ift(b) >> b' ; false
```

2.5.1.3 Guarded Command

A guarded command `b -> c` has a *guard* `b` that is a boolean expression and a command `c` that may be likened to an Orc expression. If `b` is simply a variable, this guarded command may be written in Orc calculus as: `Ift(b)>> c`. Orc language permits boolean expressions to appear as arguments of site calls; so, this expression is a valid Orc language expression even when `b` is not just a variable.

Given a set of guarded commands, `b -> c`, `b' -> c'` and `b'' -> c''`, say, the following expression concurrently executes all commands whose guards are `true`.

```
Ift(b) >> c | Ift(b') >> c' | Ift(b'') >> c''
```

If none of the guards is `true`, no command is executed, and the entire expression is equivalent to **`stop`**. If multiple guards are `true`, *all* commands corresponding to `true` guards are executed concurrently. If the guards are mutually disjoint, the command corresponding to a single `true` guard, if any, is executed. Therefore, a conditional expression of the form **`if`** `b` **`then`** `f` **`else`** `g` is encoded by

```
Ift(b) >> f | Iff(b) >> g
```

It is sometimes required to execute just one command whose associated guard is true even when the guards are not mutually disjoint. The following solution examines the guards in order and chooses the first one that is true. Assume that each command publishes at least one value when it is executed.

```
Ift(b) >> c ; Ift(b') >> c' ; Ift(b'') >> c''
```

To solve the problem more generally when a command need not publish a value when executed, examine all the guards concurrently and pick one, arbitrarily, that is true; then execute the associated command.

```
k >> ... execute command number k
  <k< (Ift(b) >> 0 | Ift(b') >> 1 | Ift(b'') >> 2)
```

The program fragment "`execute command number k`" can be written as

```
  equals(k,0) >p> Ift(p) >> c
| equals(k,1) >p> Ift(p) >> c'
| equals(k,2) >p> Ift(p) >> c''
```

It can be more easily expressed using pattern matching of Orc language, a topic that we cover in Section 4.3.5. For completeness, we show this program fragment here though this is not part of the Orc calculus.

```
k >0> c | k >1> c' | k >2> c''
```

2.5.1.4 Non-determinism

All of Orc's combinators, except the pruning combinator, are entirely deterministic in the sense that the execution proceeds in exactly the same fashion in every run of the program. The pruning combinator chooses the first published value from its right side expression and then terminates execution of its right side. The choice is non-deterministic if there are several publications that may happen first. This feature can be exploited to implement non-deterministic choice in executing either `f` or `g`.

```
(Ift(b) >> f | Iff(b) >> g)
   <b< (true | false)
```

The expression `(true | false)` publishes both `true` and `false` in some order. We have no a-priori knowledge of which of these values is published first; that depends on how the Orc scheduler is implemented. Therefore, `b` is bound to either `true` or `false` non-deterministically, and either `f` or `g` is chosen for execution.

2.5.1.5 Filtering

An elementary form of filtering the publications of a computation is captured in this example: execute expression `f` until it publishes `true`, then terminate the computation of `f` and publish a signal.

```
b  <b< ( f >x> Ift(x) )
```

Here, if `f` ever publishes `true`, then `f >x> Ift(x)` publishes a signal; then, the entire expression publishes a signal and halts.

2.5.1.6 Branching

The sequential combinator `>x>` permits a branching style of computation. Consider publishing the Cartesian product of two sets, say, the sets $\{2, 3, 5\}$ and $\{1, 6, 7\}$. We can enumerate the sets using the parallel combinator, and for each combination of enumerated values publish a tuple. The sequential combinator permits combining the separate enumerations. Site call `Tuple(x,y)` forms a tuple of the argument values. Assume that `Tuple` is strict.

```
(2|3|5)    >x>
(1|6|7)    >y>
Tuple(x,y)
```

Note that the publications appear in arbitrary order.

A branching computation may be used for enumerations of various structures. In particular, many backtrack style algorithms (see Section 5.3 in page 128) and recursive descent parsers (see Section 5.3.7 in page 132) can be programmed in this style.

2.5.1.7 Fork-Join

One of the most common concurrent programming idioms is *fork-join*: execute two expressions `f` and `g` concurrently and wait for a result from both before proceeding. Assume that `f` and `g` publish at most one value each.

```
Tuple(x,y) <x< f <y< g
```

Since `Tuple` is strict, both `x` and `y` have to be available to call `Tuple`. That is, the expression publishes only when both `f` and `g` have published.

2.5.1.8 Phase Synchronization

Given are expressions `M()>x> f` and `N()>y> g`. It is required to execute them independently except that `f` and `g` should be started only after *both* `M` and `N` have responded; therefore, merely executing `M()>x> f | N()>y> g` does not achieve this goal. We regard `M` and `N` as the first phase of the corresponding expressions and `f` and `g` the second phase. Phase synchronization requires that the start of each phase be synchronized. We use fork-join to implement phase synchronization.

```
   (Tuple(x,y) <x< M() <y< N())
>z>
   (  fst(z) >x> f
    | snd(z) >y> g
   )
```

Here, sites `fst` and `snd` extract the first and second components of a tuple. (Orc language provides pattern matching facilities to allow implicit tuple formation and component extraction.)

The given phase synchronization scheme is easily generalized to multiple component expressions and multiple phases within each.

2.5.2 Time, Timeout and Interruption

Orc is designed to communicate with the external world, and one of the most important characteristics of the external world is the passage of time. The only mechanism in Orc that deals with real time is the site `Rwait`. We use `Rwait` together with the pruning combinator to enforce a timeout. Timeout is one form of interruption. We also discuss general interruption mechanism in this section.

2.5.2.1 Repeated Concurrent Execution

Consider executing three different instances of expression `f` where the start of different instances are separated by 1 sec intervals. The simplest solution is

```
   f
| Rwait(1000) >> f
| Rwait(2000) >> f
```

We prefer a different way of writing this solution that has the virtue of naming `f` only once.

```
(0 | 1000 | 2000) >j> Rwait(j) >> f
```

or, more simply

```
(0 | 1 | 2) >i> Times(i,1000) >j> Rwait(j) >> f
```

where `Times(x,y)` publishes `x` times `y`.

If we were to extend this solution to execute `f` twenty times, say, we need a site that publishes `(0 | 1 | ... 19)`. We define site `upto` where `upto(n)`, for a non-negative integer `n`, publishes all non-negative integers smaller than `n`. Note that `upto(0)` is same as **`stop`**.

```
def upto(n) =
    (Ift(b) >>
       ( m | upto(m) ) <m< Sub(n,1) )
     <b< Positive(n)
```

Here, `Positive(n)` publishes `true` if and only if `n` is positive, and `Sub(x,y)` publishes `x-y`.

Site `upto` is quite commonly used; it belongs in the standard library. To execute f twenty times at 1 sec intervals, write

```
upto(20) >i> Times(i,1000) >j> Rwait(j) >> f
```

2.5.2.2 Metronome

A metronome publishes a signal at regular intervals. The following expression implements a bounded metronome: it publishes `n` signals `t` time units apart starting immediately. The solution uses site `upto` defined above.

```
bmetronome(n,t) = upto(n) >i> Times(i,t) >m> Rwait(m)
```

A small application of `bmetronome` is to execute an expression periodically. Below, expression `f` is executed `n` times at intervals of `t` time units.

```
bmetronome(n,t) >> f
```

The unbounded version of `bmetronome` publishes a signal every `t` time units forever. Observe that its structure is entirely different from `bmetronome`.

```
def metronome(t) = signal | Rwait(t) >> metronome(t)
```

As with `bmetronome`, we may use `metronome` to execute `f` at intervals of `t` time units forever: `metronome(t)>> f`. A more interesting example is the following site that publishes an alternating sequence of 0's and 1's, starting with a 0. Publications occur at 5 msec intervals.

```
def zeroone() =
    metronome(10) >> 0 | Rwait(5) >> metronome(10) >> 1
```

Note that `metronome(1)>> stop` computes forever, neither publishing a value nor halting.

2.5.2.3 Priority

It is required to call a site `N` immediately but publish any value received no earlier than `t` time units from now. That is, if the publication from `N` arrives before `t` it is delayed until `t`, and if it is received at or after `t` it is immediately published. We are interested in publishing at most one response of `N`. The following solution combines an immediate call to `N` with a wait.

```
(Rwait(t) >> x) <x< N()
```

We use this program to implement an elementary form of priority. It is required to call sites `M` and `N` immediately and publish one of their publications, but we grant priority to `M` in the following sense. For a window of `t` time units `M`'s response is published immediately. During this window, any response from `N` is held until the time interval has elapsed and then published, unless `M` has published earlier. If neither `M` nor `N` publishes within this window, then the first response received from either is published.

```
z <z<
  M() | ( (Rwait(t) >> x) <x< N() )
```

2.5.2.4 Polling

In concurrent programs it is often required to test if a boolean condition has become true by repeated polling. Below, we define site `await(t)` that calls site `M` every `t` milliseconds, and publishes a signal when `M` publishes `true`, and then terminates the computation. Assume that `M` always publishes a boolean.

```
def await(t) =
  M() >b>
   (  Ift(b)
    | Iff(b) >> Rwait(t) >> await(t)
   )
```

or, more simply

```
def await(t) =
  M() >b>
   (  Ift(b) ; Rwait(t) >> await(t) )
```

An even simpler solution uses the site `metronome` defined in Section 2.4 (page 27).

```
x <x< ( metronome(t) >> M() >b> Ift(b) )
```

Here, `metronome(t)` publishes a signal every `t` time unit that initiates the computation corresponding to `M()>b> Ift(b)`. Its effect is to call `M`, test its publication `b`, and publish a signal only if `b` is true. Further, the entire computation is terminated when the signal is published. Note that

```
metronome(t) >> M() >b> Ift(b)
```

has almost the desired behavior, but it does not terminate the computation after a publication. Consequently, it will continue calling `M` forever and may even publish multiple signals.

Polling is an important paradigm in concurrent programming. A telephone may be programmed to re-dial a number every 30 seconds until a non-busy tone is received, a channel may be polled periodically for arrival of a message and a customer may poll the web-site of a store every day to determine if a particular item has become available.

2.5.2.5 Timeout

Timeout can be implemented using the pruning combinator with `Rwait`. Here is a simple example; bind `true` to `x` if expression `f` publishes within `k` time units and `false` otherwise, and publish `x`.

```
x
  <x< (f >> true | Rwait(k) >> false)
```

We apply timeout with the polling example of Section 2.5.2.4. Use site `await` to publish `true` if it succeeds within `k` time units and `false` otherwise.

```
x <x< ( await(t) >> true | Rwait(k) >> false )
```

This idiom can be used in a variety of ways to manage concurrent time-based interactions with external processes. We show one such example next.

Flow rate computation A problem that is common in internet computing is to determine the download speed from a web site, so that the fastest web site can be contacted for download.

Suppose site `M` publishes a stream of values. A strategy is to receive the values for some length of time, say 3 seconds, and count the number of responses received. We need three sites to keep track of the count of responses (in the Orc standard library there is a single site to manage counts): (1) `init` to initialize the count to 0, (2) `incr` to increment the current count, and (3) `read` to return the current count value. Below, the expression runs three sequential computations: (1) initialize the count, (2) receive responses from `M` for 3 seconds and increment the count for each response received, and (3) read and publish the count after 3 seconds. Observe that each value received from `M` increments the count, but that part of the computation never publishes; `b` is assigned a value only after 3 seconds elapse.

```
   init()
>> (b <b< (M() >> incr() >> stop | Rwait(3000) ) )
>> read()
```

2.5.2.6 Interruption

Timeout is one form of interruption of an ongoing computation. The pruning combinator can be used for any interruption. Suppose we wish to execute `f` and wait for its publication, but we would like to interrupt and terminate `f` if some specific condition arises that is external to the computation of `f`. The condition may be set by a different part of the same Orc program or by another agent, say a customer cancelling an order so that the order processing computation has to be interrupted. Let the site `InterruptGet` respond only when the given condition arises. The desired program is then

```
z <z< ( f | InterruptGet() )
```

We can replace `InterruptGet()` by `Rwait(t)` to implement timeout. We can include multiple interruption conditions as alternatives, each of which is encoded by a different site.

It is often required to identify the alternative that publishes a value within a pruning combinator. In the example above, we may want to know whether `f` succeeded or if there is a timeout. In case of multiple interrupts, we would like to know which interrupt occurred so that the proper interrupt processing routines can be invoked. The standard strategy is to publish a tuple `(i,v)` where `i` identifies the alternative that published the output and `v` is the value published. In the example above, there are only two alternatives and we use boolean values to identify them. Using the Tuple site introduced in Section 2.5.1.7 (page 32) we modify the program given above.

```
z <z<
      (  f              >x> Tuple(true,x)
       | InterruptGet() >x> Tuple(false,x)
      )
```

For further computation with `z`, a site is needed to extract its components.

2.5.3 Logical Connectives

2.5.3.1 2-valued Logic

Suppose the site call `M()` may or may not publish a signal; similarly for `N()`. It is required to publish a signal when (1) both site calls `M()` and `N()` do ("logical and"), and (2) at least one of them does ("logical or").

```
M() >> N()            -- logical and
b <b< ( M() | N() )   -- logical or
```

Suppose further that site `M` is finite (i.e., engages in only finite computation) and helpful (see Section 2.2.2) so that it sends a terminal response to indicate the end of its computation. Then there is a simpler version of logical or: `M(); N()`. In this program, `N()` will not be called if `M()` publishes, unlike the previous solution in which both `M()` and `N()` are called.

Logical negation can be implemented only for a finite helpful site `M`. The following program publishes a signal if and only if the site call `M()` does not publish.

```
( M() >> true ; false ) >b> Iff(b)
```

Here, the expression `( M()>> true ; false )` publishes `true` if `M()` publishes and `false` if `M()` does not publish. Hence, boolean variable `b` is true if and only if `M()` publishes. The entire expression publishes a signal if and only if `b` is `false`, i.e., `M()` does not publish.

We show the program for a more complex logical connective, exclusive-or. First, we define site `xor`, the exclusive-or of two boolean values, that publishes a signal if exactly one value is `true`, and it never publishes otherwise.

```
def xor(p,q) =
     Ift(p) >> Iff(q)
   | Iff(p) >> Ift(q)
```

Observe the usefulness of a guarded command structure in this case.

Next, we define exclusive-or based on signals; publish a signal if exactly one of `M` and `N` publishes given that both `M` and `N` are finite and helpful. Below, `p` is `true` if and only if `M` publishes and, similarly, `q` is true if and only if `N` publishes.

```
xor(p,q)
 <p< (M() >> true ; false)
 <q< (N() >> true ; false)
```

2.5.3.2 3-valued Logic: Parallel-and, Parallel-or

Let sites `M` and `N` publish boolean values, `true` or `false`, or do not publish at all. The logical connectives are then operations in 3-valued logic: the result of a connective is a boolean such that any value received from a site in the future can not invalidate the published result. Thus, (1) logical-and publishes `true` only if both site calls publish `true`, publishes `false` if either site publishes `false`, and does not publish otherwise (i.e., if neither site has published, or only one site has published and the publication is `true`), (2) logical-or publishes `false` only if both site calls publish `false`, publishes `true` if either site publishes `true`, and does not publish otherwise. These connectives are often called "Parallel-and" and "Parallel-or", respectively, to differentiate them from their strict counterparts which require both sites to publish before they can compute the result. We show the program for parallel-or; the program for parallel-and is a dual of this program. Site `or`, defined in Section 2.4 (page 27), already implements this specification. It remains to tie the call to `or` with the values received from the sites.

```
or(p,q)
  <p< M()
  <q< N()
```

Here, if either `p` or `q` becomes bound to `true`, then the expression publishes `true` immediately, regardless of whether the other variable is bound or not. Observe that if both `p` and `q` are `true` then the expression publishes 3 values, all `true`. We can force termination of the computation after the first publication by using the pruning combinator:

```
b <b< (or(p,q)
         <p< M()
         <q< N() )
```

Parallel-or on helpful sites If both sites `M` and `N` are finite and helpful, we get a simpler program for parallel-or:

```
( M() | N() ) >p> Ift(p) >> true ; false
```

If either site publishes `true`, then the expression publishes `true`. If neither site publishes `true`, then `( M()| N())>p> Ift(p)` halts; therefore, the expression publishes `false`. As before, to avoid multiple publications of `true`, we can employ the pruning combinator.

```
true << (( M() | N() ) >p> Ift(p) ) ; false
```

Observe that if `M` publishes `false` and `N` never publishes but sends only a negative response, then the expression publishes `false`; this is unlike the previous solution in which no value is published in this case.

2.5.4 Probabilistic computation

Site call `Random(n)`, where `n` is a non-negative integer, publishes a random non-negative integer smaller than `n` with uniform probability. This site is in the Orc standard library and it is available in one form or another in nearly every programming language. Note that `Random(0)` is same as **`stop`**.

First, we use `Random` to define a few auxiliary sites. The site defined below publishes a random boolean value, both values being published with equal probability. We use site `equals` where `equals(n,0)` publishes `true` if `n` is zero and `false` otherwise.

```
def randomBool() = Random(2) >n> equals(n,0)
```

As an example of the use of this site, execute one of `f` and `g` non-deterministically, both choices being equally likely.

```
RandomBool() >b> (Ift(b) >> f | Iff(b) >> g)
```

Next, we define a biased boolean; `biasedBool(p)` publishes `true` with probability p and `false` with probability $1 - p$, where p is a real number between 0 and 1. We implement this site by choosing a random number uniformly in a large interval, say 1000, and publishing `true` if and only if the number is strictly less than $p \times 1000$. Site call `Lt(n,m)` returns `true` if and only if `n` is strictly less than `m` and `false` otherwise, and `Times(x,y)` publishes `x` times `y`.

```
def biasedBool(p) =
    Random(1000) >n> Times(p,1000) >m> Lt(n,m)
```

The following site publishes a signal immediately with probability p and never responds with probability $1 - p$.

```
def randomResponse(p) = biasedBool(p) >b> Ift(b)
```

Now consider a site that publishes a boolean value or never responds. Suppose it publishes with probability `r`, and in case it publishes it publishes `true` with probability `p`. We need to simulate such a site in testing the parallel-or program of Section 2.5.3.2 (page 38), for instance.

```
def randomBoolResponse(r,p) =
    randomResponse(r) >> biasedBool(p)
```

Next, consider a more sophisticated simulation application in which a site responds with a boolean with probability as above, but its response is delivered at a time that is uniformly distributed up to `t`.

```
def randomBoolTimeResponse(r,p,t) =
    randomBoolResponse(r,p) >b>
    Random(t)               >w>
    Rwait(w)                >>   b
```

2.5.5 Reactive Programming

Reactive programming has acquired many different connotations since it was first introduced by Manna and Pnueli [33]. Roughly speaking, a reactive program interacts with its environment during its execution. The interaction could take many forms: reading input from a sensor, sending a file to a printer and responding to possible printer failure, altering the settings on Mars Rover ... A reactive program may not have all its inputs when it starts executing; it acquires the required inputs during its execution. It may run forever if inputs are continually supplied to it. A mobile phone is an example of a perpetually running reactive program that responds to the inputs from its user.

In this section, we develop a very small reactive program, the control of a book-reader device. The operation of the device is as follows. Once the device is powered on, its first page (numbered 0) is rendered and displayed. The user can navigate to the next or previous page by pressing `forward` or `backward` button; the next (previous) page is the current page if there is no higher (lower) numbered page. If the user presses no button for `T` time units, the device is switched off.

There are several domain-dependent sites in this case. Site `render` is called with a page number as input; it retrieves the page, renders and displays it, and then publishes a signal. Any call to `render` invalidates all ongoing renderings. Sites `forward` and `backward` publish a signal when the user presses the corresponding button. Site `switchoff` switches off the device.

Below, `bookReader(n)` renders and displays the page numbered `n` and waits for user input simultaneously. If there is a user input, it terminates all ongoing computations and displays the next or previous page. If there is no user input for `T` time units, it switches off the device. Below, `Addmod(n)` publishes `n+1` if `n` is not the highest page number, otherwise it publishes `n`. Similarly, `Submod(n)` publishes `n-1` if `n` is positive and 0 otherwise. The program assumes that the device is switched on. It starts by rendering page 0.

```
def bookReader(n) =
     (  Ift(b) >> (bookReader(m) <m< Addmod(n) )
      | Iff(b) >> (bookReader(m) <m< Submod(n) )
     )
 <b< (  render(n)   >> stop
      | forward()   >> true
      | backward()) >> false
      | Rwait(T)    >> switchoff() >> stop
     )

bookReader(0)
```

Observe that a user input overrides any rendering operation. If the user presses, say, the forward button quickly several times, then the intermediate pages are not rendered or displayed. In fact, every press of the forward button immediately terminates all ongoing computations.

This example, though simple and simplified, is typical of event-driven sys-

tems. Such a system waits for external stimuli, such as button presses and sensor readings, and internal stimuli, typically the passage of time. Each stimulus is handled by a separate piece of code. We have shown orchestration of the pieces for a very simple example.

2.6 Concluding Remarks

The goal of this chapter was to introduce the Orc calculus and demonstrate that the calculus can express solutions to a variety problems that arise in programming, concurrent programming in particular. The calculus is unusual in that it is not self contained, but has to make use of sites some of which may be externally defined. As our examples show, a library of standard sites is often sufficient to solve intricate problems by suitably orchestrating the calls to them.

Orc expressions may publish an arbitrary number of values. This contrasts with most programming languages where an expression returns a single value. Orc's choice is dictated by the nature of concurrent computations: an expression may consist of two independent expressions each of which may return a value. Since Orc sites may be recursive, an expression that calls a site may publish an unbounded number of values. Since **`stop`** is an expression, it is possible for an expression to publish no value. Admitting an arbitrary number of publications has simplified our theory considerably. In particular, any combinator can combine any two Orc expressions; there are no restrictions at all.

A calculus does not make a programming language. It is frustrating to program basic arithmetic operations by making site calls; moreover, site definitions can not even be nested. The goal of the calculus is to provide the minimal machinery so that its semantics can be described easily and a programming language can be built around it. We give the semantics in Chapter 3 and propose a programming language in Chapter 4 that adds a few elementary features and a standard library to the calculus.

Chapter 3

Orc Semantics

3.1 Introduction

We introduced the Orc calculus in Chapter 2 without defining a formal syntax or semantics. The goal of that chapter was to acquaint the reader with the Orc combinators, their (approximate) meanings and the style of concurrent programming espoused by Orc. In this chapter, we define the syntax and semantics of Orc rigorously. The semantic description is given in two parts, first without the notion of real time and next with real time.

Describing formal semantics is a large and complex undertaking for any large and complex system. For such systems, typically, formalism is abandoned, to be replaced with a description whose precision varies widely. Quite often, meanings are described for the common cases, in prose and with examples, but the uncommon or boundary cases are simply ignored. A colleague of mine once enquired about the meaning of a particularly troublesome program written in a certain hardware design language; the language designer responded with "who would ever write such a program?". This may sound blasphemous to a theorist, but it is perfectly reasonable for a language whose domain of applications is well-understood. The domain of concurrent applications is not yet clearly understood; applications range from circuit design to running Olympic games. We are still in the early days of concurrent programming to rely on approximate meanings of our language constructs.

There are three important ways in which we utilize the formal semantics of Orc. First, it provides a guarantee to an Orc programmer about the meaning of a program. Second, it provides the specification for an implementation. Third, it permits derivations of algebraic identities among Orc expressions that are useful both for programmers, in structuring their programs, and implementers, in automatically reducing programs to normal forms for efficient execution.

A complex language like `C++`, for instance, has a formal semantics given by its compiler. It meets the needs of the programmers to the extent that they can run experiments to test their hypotheses about semantics. It meets the

needs of the implementers to the extent that they have to conform to an earlier implementation. But such a semantics is not concise. We advocate a semantics that is not only formal, but concise, that permits succinct derivation of facts and settles arguments quickly. To that end, we propose a very simple set of rules that describe the meaning of each Orc combinator in isolation. Since Orc calculus imposes no constraints on the compositions of expressions, the rules can be applied hierarchically to derive meanings of arbitrary programs.

Roadmap for this chapter We describe the syntax of Orc by a grammar in Section 3.1 (Table 3.1). Next, we define an *asynchronous* semantics in which time plays no role; see Section 3.4 (page 46); the rules are summarized in Figure 3.1 (page 81). Certain expressions can be shown to be equivalent using the semantics. Two different notions of equivalence are introduced: *strong equivalence* in Section 3.6 (page 56) and *weak equivalence* in Section 3.8.3 (page 73). A number of useful identities for strong equivalence are developed in Section 3.7 (page 65) and for weak equivalence in Section 3.8.4 (page 74) and Section 3.8.5 (page 75), that are used in the rest of the book.

Section 3.9 (page 75) contains a *synchronous* semantics in which passage of time is taken into account. The synchronous semantics is an enhancement of the asynchronous semantics; each transition carries a time of occurrence with it. We define equivalence in the synchronous theory in a similar manner and show that the corresponding identities still hold. A summary of synchronous semantics rules appears in Figure 3.3 (page 83).

The complete theory needs an astounding number of proofs that will be a book on its own. We prove a few select identities that capture most of the essential ideas.

Notation Orc expressions are written in mathematical font in this chapter.

3.2 Syntax

The syntax of Orc is given in Table 3.1. An expression is either (1) *stop*, (2) a site call, written as $e(\bar{x})$, where e is either a site name or a variable that would be bound to a site name, and $\bar{x}$ is a list of variables (actual parameters), (3) an expression built out of two constituent expressions using a combinator, or (4) a site definition followed by an expression. The combinators in decreasing order of precedence are: sequential, parallel, pruning and otherwise. A declaration is a site definition that has a name followed by a list of variables, $\bar{x}$, that constitute its formal parameters, and a body that is an expression. A program is an expression.

Internal and External Site A site defined within an Orc program is an *internal* site. A site defined externally (not within an Orc program) is an *external* site. Their semantic treatment differ, because an external site's execution is outside the control of Orc. Operational semantics for external sites has to rely

f,g	$\in$	*Expression*	::=
		stop	Basic Expression
	|	$e(\bar{x})$	Site Call
	|	$f \mid g$	Parallel Combinator
	|	$f >x> g$	Sequential Combinator
	|	$f <x< g$	Pruning Combinator
	|	$f \,;\, g$	Otherwise Combinator
	|	$D \# f$	prefixing declaration
D	$\in$	*Declaration*	::=
		def $E(\bar{x}) = f$	Site Definition
		Program	::= f

Table 3.1: Syntax of Orc Calculus

on the specifications of such sites but not the transformation of their code, as we do for internal sites.

3.3 Asynchronous Semantics

Substitution, Binding Expression $(x := c).f$ is obtained from f by substituting every free occurrence of x by c, where c is a value, a variable or an expression. We generalize substitution to a set of pairs, $\{(x_0 := c_0)\ , (x_1 := c_1), \cdots\}$, whereby distinct free variables x_0, x_1, $\cdots$, are replaced by c_0, c_1, $\cdots$. Let B denote such a set of pairs; $B.f$ denotes the expression obtained from f by making the substitutions B in f.

Formally, $B.f$ is defined in Table 3.2. There $B.\bar{x}$ stands for the list obtained by substituting every variable name in $\bar{x}$ by the corresponding value (or variable) from B and leaving all other variable names unchanged. Substitution $(B\backslash x)$ where x is a variable, is same as B except that the substitution for variable x, if any, is removed. Observe that a site or expression name may also be a variable (see Section 2.4.1 on "closure"); hence, the substitution also applies to such variables. Note that if x is not a free variable in f, then $B.f = (B\backslash x).f$ for any substitution B. Further, $(B\backslash x)\backslash y = (B\backslash y)\backslash x$, for all x and y even when they are identical variables. Empty substitution, denoted by ϕ, has no effect on an expression.

Application of substitution has higher priority than any Orc combinator. We use the notation $(B.B').f$ and $B.B'.f$ to denote $B.(B'.f)$.

In $(x := c)$, if c is a value and not a variable name or expression, we call the substitution a *binding*. Sequential and pruning combinators have component expressions whose publications result in bindings. Substitution of a variable by another variable or expression is required when parameters of a site are replaced.

$\{\}.f$	$=$	f
$B.stop$	$=$	$stop$
$B.(e(\bar{x}))$	$=$	$(B.e).(B.\bar{x})$
$B.(f \mid g)$	$=$	$B.f \mid B.g$
$B.(f >x> g)$	$=$	$B.f >x> (B\backslash x).g$
$B.(f <x< g)$	$=$	$(B\backslash x).f <x< B.g$
$B.(f \texttt{;} g)$	$=$	$B.f \texttt{;} B.g$
$B.(\texttt{def}\ E(\bar{x}) = f\ )$	$=$	$\texttt{def}\ E(\bar{x}) = (B\backslash \bar{x}).f$
$B.(D \# f)$	$=$	$B.D \# B.f$

Table 3.2: Definition of substitution application

Operational Semantics Orc has a small-step operational semantics given by a labeled transition system. The states correspond to Orc expressions and labels are the events. A *transition* $f \xrightarrow{a} f'$ denotes that execution of expression f may cause *event* a and a subsequent *move* to expression f'. A rule of the form

$$\frac{f \xrightarrow{a} f'}{f \mid g \xrightarrow{a} f' \mid g}$$

denotes that if f moves with event a to expression f' (this is called the *precondition* of the transition), then expression $f \mid g$ may move with event a to expression $f' \mid g$. The rule merely shows a possible transition; the transition is not guaranteed to happen in any particular execution even if the precondition holds.

Events There are three kinds of events: (1) $\surd M(\bar{v})$ denotes the event of calling site M with a list of actual parameters $\bar{v}$, (2) $!c$ denotes publication of value c and ! publication of a signal, and (3) τ is an event corresponding to an internal transition that is not intended to be observable by the environment of the expression.

3.4 Transition Rules

The transition rules are explained below for each syntactic form and summarized in Figure 3.1.

3.4.1 Expression *stop*

There is no transition starting from *stop*.

3.4.2 External Site Call and Response

First, we show that lenient calls to external sites can be replaced by strict calls. Consider the site call $M(x_1, \cdots, x_n)$. Any such site M can be regarded as $n+1$ sites $M_0, M_1, \cdots, M_n$ that collaborate to simulate M. The initial site call is made to M_0 and M_i is called only when variable x_i gets bound to a value. Thus, the call $M(x_1, \cdots, x_n)$ can be replaced by $M_0() \mid M_1(x_1) \mid \cdots \mid M_n(x_n)$ where each call is strict. The response of original M may be received from any number of M_is. Henceforth, we restrict ourselves to defining the semantics of strict site calls for external sites. Even though this translation shows that a strict call need not have more than one parameter, we define the semantics for an arbitrary number of parameters because the same notation is used for calls to internal sites.

It is common in small-step operational semantics to extend the underlying language to represent intermediate states. First, we allow $e(\bar{p})$ as an expression where e is a site (a value or a variable) and $\bar{p}$ a list that may include both values or variables. Expression $e(\bar{p})$ denotes the state where some of the actual parameters have been bound to values. Next, we allow $?M(\bar{p})$ as an expression, denoting that a call has been made to site M and its response is pending.

A strict site call is made only if the site name and all its parameters are bound to values. The rule for such a call is given by:

$$M(\bar{v}) \xrightarrow{\surd M(\bar{v})} ?M(\bar{v}) \qquad \text{(SiteCall)}$$

where all items in $\bar{v}$ are values and M is bound to a site name. Note that $?M(\bar{v})$ should actually be a handle uniquely identifying the call in an execution, because the same site may be called with identical parameter values in an execution. We have chosen a non-unique representation here only to simplify the presentation.

We repeat the material from Section 2.2.2 (page 21) regarding the kinds of responses. A site sends three kinds of responses: (1) *non-terminal* response that has an associated value indicating that the computation at the site may not have ended, (2) *terminal response with a value* indicating that the computation has ended, so there will be no further response, and (3) *terminal response without a value* indicating the end of computation. Call a response, terminal or non-terminal, a *positive* response if it carries a value. A *negative* response does not carry a value, and it is always terminal. A *helpful* site is one that sends a terminal response whenever its computation on behalf of a call ends.

There is no order among the non-terminal responses, i.e., the caller may receive these responses in any order unrelated to the the order of sending by the site. The terminal response, if any, is always received after all non-terminal responses.

Below, publication of value v is written as $!v$ and of a **`signal`** as $!$.

$$\frac{?M(\bar{v}) \text{ receives a non-terminal response } !c}{?M(\bar{v}) \xrightarrow{!c} ?M(\bar{v})} \qquad \text{(NT-Res)}$$

$$\frac{?M(\bar{v}) \text{ receives a terminal response } !c}{?M(\bar{v}) \xrightarrow{!c} \mathit{stop}} \quad \text{(T-Res)}$$

$$\frac{?M(\bar{v}) \text{ receives a negative response}}{?M(\bar{v}) \xrightarrow{\tau} \mathit{stop}} \quad \text{(NegRes)}$$

The responses of some of the fundamental sites are as follows. *Ift*(b) is equivalent to *signal* if b is true and to *stop* otherwise, *Iff*(b) is equivalent to *signal* if b is false and to *stop* otherwise, and a constant site, such as 3, sends only a terminal response with value 3.

Convention We use τ, the internal event, for calls to common sites and constant sites. Thus, we use $3 \xrightarrow{\tau} ?3$, rather than $3 \xrightarrow{\sqrt{3}} ?3$. This convention treats such calls as internal events rather than events observable by the environment of the expression. It allows us to derive a number of useful identities involving the common sites.

3.4.3 Internal Site Call and Response

The semantics of internal sites, the sites defined within an Orc program, are defined differently from those for external sites because we can exploit the available code of an internal site. Each call to an internal site is replaced by the body of the definition with actual parameter values substituted for the formal parameters.

Consider a site definition $\mathit{def}\ E(\bar{x}) = g$, which we abbreviate to D in the following discussion. This definition precedes an expression, as in $D \# f$. Any transition of f is unaffected by the presence of such a definition.

$$\frac{f \xrightarrow{a} f'}{D \# f \xrightarrow{a} D \# f'} \quad \text{(DefPass)}$$

The next rule allows us to get rid of a definition. We would expect to have a rule of the form $D \# f \xrightarrow{\tau} (g/E).f$, that is, substitute the site name by its body in f. This is an acceptable rule if the definition is not recursive, i.e., g has no occurrence of the site name E. In case the site is recursively defined, g has an occurrence of E and $(g/E).f$ would have an occurrence of E; having lost the site definition, we would be unable to have any further transition with E. To overcome this problem, we encode the site definition by its closure, a triple $\langle E, \bar{x}, g\rangle$, and substitute all occurrences of E in f by this triple. The following transition is allowed provided g has no free variable other than the ones in $\bar{x}$, or E; free variables of an expression are defined formally in Table 3.4 (page 67). In particular, a closure is a value and it has no free variable.

$$\frac{\begin{array}{c} D \text{ is } \mathit{def}\ E(\bar{x}) = g \\ \mathit{free}(g) \subseteq \{E\} \cup \{\bar{x}\} \end{array}}{D \# f \xrightarrow{\tau} (E := \langle E, \bar{x}, g\rangle).f} \quad \text{(DefScope)}$$

A subexpression $E(\bar{p})$ of f is replaced by $(E := \langle E, \bar{x}, g\rangle)(\bar{p})$ by application of the (DEFSCOPE) rule. Expression $(E := \langle E, \bar{x}, g\rangle)(\bar{p})$ is not an Orc expression, but an intermediate expression for which we need a transition rule. First, the actual parameters $\bar{p}$ are substituted for the formal parameters $\bar{x}$ in g; note that $\bar{p}$ may include both values and variables that are not yet bound to values, because, unlike for an external site call, the execution of an internal site commences as soon as it is called even though all parameters may not be bound. Next, $\langle E, \bar{x}, g\rangle$ is substituted for E to enable further recursive expansion.

$$\langle E, \bar{x}, g\rangle(\bar{p}) \xrightarrow{\tau} (E := \langle E, \bar{x}, g\rangle).(\bar{x} := \bar{p}).g \qquad \text{(DefCall)}$$

Note that an internal site call is executed leniently so that the execution of the body may begin even before all the formal parameters are bound.

Internal Site Response Just like a external site, an internal site responds by sending positive and negative responses. Each publication corresponding to an internal site call is a non-terminal response. It can be shown that any internal site that calls only helpful sites is helpful. The halting of such a site can be detected just like halting of any Orc expression, see Section 3.7.5 (page 70). The implementation sends a negative response on detection of halting.

3.4.4 Parallel Combinator

Expression $f \mid g$ is executed by executing each of its component expressions independently. Therefore, any transition in f or in g is a transition in $f \mid g$.

$$\frac{f \xrightarrow{a} f'}{f \mid g \xrightarrow{a} f' \mid g} \qquad \text{(ParL)}$$

$$\frac{g \xrightarrow{a} g'}{f \mid g \xrightarrow{a} f \mid g'} \qquad \text{(ParR)}$$

3.4.5 Sequential Combinator

Expression $f >x> g$ has two kinds of transitions corresponding to (1) non-publication and (2) publication transitions of f. A non-publication transition of f (written as "non-pub" in the rules) is also a transition of $f >x> g$. A publication transition of f creates an instance of g with the published value substituted for all occurrences of x in g, and the event itself is recorded as an internal event. Expression g is a template that is instantiated for each publication of f; transitions within the instances of g are *not* part of the transitions of $f >x> g$. They will be recorded elsewhere (in a parallel combinator).

$$\frac{f \xrightarrow{a} f' \qquad \text{non-pub } a}{\begin{array}{c} f >x> g \xrightarrow{a} f' >x> g \\ f \gg g \xrightarrow{a} f' \gg g \end{array}} \qquad \text{(SeqN)}$$

$$\frac{f \xrightarrow{!c} f'}{\begin{array}{c} f >x> g \xrightarrow{\tau} f' >x> g \mid (x := c).g \\ f \gg g \xrightarrow{\tau} f' \gg g \mid g \end{array}} \quad \text{(SeqP)}$$

3.4.6 Pruning Combinator

This and the following section (Section 3.4.7) logically belong after the definition of strong equivalence among expressions, Section 3.6.2 (page 57), because the notion of strong equivalence is required to define a transition rule in each of these sections. None of the transition rules discussed here (or, for that matter, for the other combinators) is used in any of the sections upto Section 3.7. The sequence of presentation, though unusual, first shows the transition rules which motivates the need for defining bisimulation and strong equivalence.

We explain strong equivalence in informal terms here. Roughly, strongly equivalent expressions f and g, written as $f \cong g$, are indistinguishable in their behaviors, i.e., f and g can be substituted for each other within any expression and the resulting expressions are strongly equivalent. In particular, an expression that is equivalent to *stop* can not make any transition, and it may be replaced by *stop* within any other expression.

Expression $f <x< g$ has four kinds of transitions corresponding to: (1) transitions of f, that are recorded as transitions of $f <x< g$, (2) non-publication transitions of g, that are recorded as transitions of $f <x< g$, (3) a publication transition of g, that causes termination of the remaining computation of g, substitution of the published value for x in f and an internal event of $f <x< g$, and (4) the case where $g \cong stop$, which we explain next.

Consider the expression $f <x< stop$. It is clear that x will never be bound to a value. To denote that variable x will never be bound, we bind x to a fictitious value, $\not{!}$, with the meaning that if x is a free variable in expression $e(\bar{p})$, then $(x := \not{!}).e(\bar{p})$ is equivalent to *stop*, i.e., it has no transition. So, if $e = x$ or one of the free variables in $\bar{p}$ is x, then $e(\bar{p})$ may be replaced by *stop*. Free variables of an expression f are defined in Table 3.4 (page 67). The substitution rules from Table 3.2 (page 46) apply for $\not{!}$ as well.

$$\frac{f \xrightarrow{a} f'}{\begin{array}{c} f <x< g \xrightarrow{a} f' <x< g \\ f \ll g \xrightarrow{a} f' \ll g \end{array}} \quad \text{(PruL)}$$

$$\frac{g \xrightarrow{a} g' \qquad \text{non-pub } a}{\begin{array}{c} f <x< g \xrightarrow{a} f <x< g' \\ f \ll g \xrightarrow{a} f \ll g' \end{array}} \quad \text{(PruN)}$$

$$\frac{g \xrightarrow{!c} g'}{\begin{array}{c} f <x< g \xrightarrow{\tau} (x := c).f \\ f \ll g \xrightarrow{\tau} f \end{array}} \quad \text{(PruP)}$$

$$\frac{g \cong stop}{\begin{array}{c} f <\!x\!< g \xrightarrow{\tau} (x := \not{!}).f \\ f \ll g \xrightarrow{\tau} f \end{array}} \quad \text{(PruH)}$$

Note: Rule (PRUP) shows that after g publishes a value its execution does not play a role in further execution of $f <\!x\!< g$; that is, it is terminated. Therefore, any internal site for which g may have a pending call is also terminated.

3.4.7 Otherwise Combinator

Expression $f \;;\; g$ has three kinds of transitions corresponding to: (1) non-publication transitions of f, that are recorded as transitions of $f \;;\; g$, (2) a publication transition of f, that is recorded as a transition of $f \;;\; g$ and results in discarding g, (3) f is equivalent to *stop*, which causes the execution of g to begin.

$$\frac{f \xrightarrow{a} f'}{f \;;\; g \xrightarrow{a} f' \;;\; g} \quad \text{(OthN)}$$

$$\frac{f \xrightarrow{!c} f'}{f \;;\; g \xrightarrow{!c} f'} \quad \text{(OthP)}$$

$$\frac{f \cong stop}{f \;;\; g \xrightarrow{\tau} g} \quad \text{(OthH)}$$

3.4.8 Trace

A *trace* of an expression is the sequence of events at any point during its execution. Let relation $\rightarrow\!*$ be the transitive closure of the transition relation $\rightarrow$, together with the empty move ϵ. Below, we denote concatenation by juxtaposition so that as is a followed by s. Formally, $\rightarrow\!*$ is defined by

$$f \xrightarrow{\epsilon}\!* f \qquad \frac{f \xrightarrow{a} f'',\ f'' \xrightarrow{s}\!* f'}{f \xrightarrow{as}\!* f'}$$

If $f \xrightarrow{s}\!* f'$, we say that s is a *trace* of f. We have included $f \xrightarrow{\epsilon}\!* f$ to guarantee that the set of traces of an expression is prefix-closed. Every trace is finite in length. Henceforth, we abbreviate $f \xrightarrow{a} f'',\ f'' \xrightarrow{s}\!* f'$ to $f \xrightarrow{a} f'' \xrightarrow{s}\!* f'$.

Example

For the expression $((M(x) \mid x) >\!y\!> R(y)) <\!x\!< (N() \mid S())$ we show the derivation of a trace in detail. Here M, N, R and S are sites. In this execution $N()$

publishes value 5. The derivation shows the event that happens for each transition of the main expression. The event of the main expression is typically caused by a transition of a subexpression; we show the derivation for the corresponding subexpression within braces following the event.

$$((M(x) \mid x) >y> R(y)) <x< (N() \mid S())$$

$\overset{\surd S()}{\rightarrow}$ {

$S() \overset{\surd S()}{\rightarrow} ?S()$ from (SITECALL)

$N() \mid S() \overset{\surd S()}{\rightarrow} N() \mid ?S()$ from (PARR)

apply (PRUN)

}

$$((M(x) \mid x) >y> R(y)) <x< (N() \mid ?S())$$

$\overset{\surd N()}{\rightarrow}$ { similar to the above }

$$((M(x) \mid x) >y> R(y)) <x< (?N() \mid ?S())$$

$\overset{\tau}{\rightarrow}$ {

$?N() \overset{!5}{\rightarrow} ?N()$ from (POSRES)

$?N() \mid ?S() \overset{!5}{\rightarrow} ?N() \mid stop$ from (PARL)

apply (PRUP)

}

$$(M(5) \mid 5) >y> R(y)$$

$\overset{\surd M(5)}{\rightarrow}$ {

$M(5) \overset{\surd M(5)}{\rightarrow} ?M(5)$ from (SITECALL)

$M(5) \mid 5 \overset{\surd M(5)}{\rightarrow} ?M(5) \mid 5$ from (PARL)

apply (SEQN)

}

$$(?M(5) \mid 5) >y> R(y)$$

$\overset{\tau}{\rightarrow}$ {

$?5 \overset{!5}{\rightarrow} stop$ from (POSRES)

$?M(5) \mid 5 \overset{!5}{\rightarrow} ?M(5) \mid stop$ from (PARR)

apply (SEQP)

}

$$(?M(5) \mid stop) >y> R(y) \mid R(5)$$

The trace corresponding to this execution is:

$$\surd S() \quad \surd N() \quad \tau \quad \surd M(5) \quad \tau$$ □

3.5 Strong Bisimulation

We develop a theory that allows us to prove that two expressions are equivalent. Strong equivalence of f and g implies that their trace sets are identical. But, more generally, it implies that f and g may be substituted for each other within any expression and the resulting expressions are equivalent. Thus, f and g are indistinguishable to any observer.

Equivalence plays a central role in Orc. Orc interpreter uses a number of heuristics to improve performance that are based on replacing an expression by an equivalent one. Orc programmer needs to know a few identities among the expressions for effective programming. We state and prove a number of identities in Section 3.7.3 (page 67). Some of these identities are intuitively obvious from the informal semantic description given in Chapter 2; their proofs validate the intuition.

We develop two notions of equivalence, *strong* and *weak*. Weak equivalence ignores the τ events; strong equivalence does not ignore any event. We define these notions below using the notion of *bisimulation*. There is a vast literature on bisimulation; see Milner [36, 38] who has developed the theory based on the earlier work of Park [43]. We give a very brief overview of the theory that is sufficient for our purposes.

Alpha Equivalence There is a very simple form of equivalence, Alpha Equivalence, obtained by renaming the bound variables. Rename the bound variables of an expression with names that do not occur in that expression; the resulting expression is equivalent to the original. See Table 3.4 (page 67) for a definition of free variables of an expression; the remaining variables are bound.

3.5.1 Definition of Strong Bisimulation

Bisimulation is reminiscent of state minimization in finite state machines. A typical strategy in state minimization is to first postulate an equivalence relation $\mathcal{S}$ over the states, and then show that each transition in the machine respects $\mathcal{S}$ in the following sense: for any pair of states (f, g) in $\mathcal{S}$ and any transition from f, there is a transition with identical label from g and both transitions lead to equivalent states (under $\mathcal{S}$). It follows, by induction on the number of transitions, that the sequence of labels along all finite paths, i.e., the languages, are identical for equivalent states.

Bisimulation is a generalization of this idea. The machines need not have finite number of states, nor does $\mathcal{S}$ have to be an equivalence relation. Different transitions from a state may be labeled with the same symbol. In our study, each state corresponds to an expression, and the transitions and their labels are given by the semantic rules. Henceforth, we use "state" and "expression" interchangeably. The sequence of labels along a path from an expression is a trace of that expression. Our interest is in proving that certain expressions have the same set of traces.

Definition: Strong Bisimulation

A binary relation $\mathcal{S}$ over expressions is a *strong bisimulation* if for every (f,g) in $\mathcal{S}$ (also written as $f \mathcal{S} g$)

for every $f \xrightarrow{a} f'$, there is some $g \xrightarrow{a} g'$ such that $f' \mathcal{S} g'$, and
for every $g \xrightarrow{b} g''$, there is some $f \xrightarrow{b} f''$ such that $f'' \mathcal{S} g''$. □

Example

We show that $f \mid g$ and $g \mid f$ are strongly bisimilar. Define relation $\mathcal{S}$ by $f \mid g \ \mathcal{S} \ g \mid f$, for all f and g. In order to prove that $\mathcal{S}$ is a strong bisimulation we need to show that (1) for every transition of $f \mid g$ there is a matching transition of $g \mid f$ and the resulting pair of expressions are again in $\mathcal{S}$, and conversely, (2) for each transition of $g \mid f$. We will only show (1) since the proof of (2) is symmetric.

Any transition of $f \mid g$ is one of two types: a transition of f that results in a transition of $f \mid g$ by applying (PARL), or a transition of g that results in a transition of $f \mid g$ by applying (PARR). We consider each of these transitions in turn.

• Transition $f \xrightarrow{a} f'$: Applying (PARL), we have $f \mid g \xrightarrow{a} f' \mid g$. We display the matching transition $g \mid f \xrightarrow{a} g \mid f'$, which is justified by applying (PARR) with $f \xrightarrow{a} f'$. Now, we have $f' \mid g \ \mathcal{S} \ g \mid f'$ from the definition of $\mathcal{S}$, using f' and g in place of f and g.

• Transition $g \xrightarrow{b} g'$: The proof is similar to above. □

This proof, in particular, establishes that the trace sets of $f \mid g$ and $g \mid f$ are identical. We show a number of short-cuts later that simplify bisimulation proofs.

3.5.2 Properties of Strong Bisimulation

Pairs (f,g) in a strong bisimulation are said to be *strongly bisimilar*. The following theorem shows that for strongly bisimilar expressions, the single transition in the definition of bisimulation can be generalized to any finite sequence of transitions. Therefore, strongly bisimilar expressions have identical trace sets.

Theorem 3.1. Let $f \mathcal{S} g$ where $\mathcal{S}$ is a strong bisimulation. Then, for every $f \xrightarrow{u}* f'$, there is some $g \xrightarrow{u}* g'$ such that $f' \mathcal{S} g'$. Conversely, for every $g \xrightarrow{v}* g''$ there is some $f \xrightarrow{v}* f''$ such that $f'' \mathcal{S} g''$.

Proof: We prove the first part, that for every pair of expressions f and g, where $f \mathcal{S} g$, for every $f \xrightarrow{u}* f'$ there is some $g \xrightarrow{u}* g'$ such that $f' \mathcal{S} g'$. The proof in the other direction is similar. Proof is by induction on n, the length of u.

For $n = 0$, both f and g include the trace of length 0, the empty trace, by definition of trace, and the resulting expressions, f and g, are strongly bisimilar.

Next, suppose the assertion holds for all strongly bisimilar expressions for traces of length n, for some n, $n \geq 0$. Let $f \stackrel{as}{\rightarrow}* f'$ where s is a trace of length n; we show that there is some $g \stackrel{as}{\rightarrow}* g'$ and $f' \mathcal{S} g'$.

$f \stackrel{a}{\rightarrow} f'' \stackrel{s}{\rightarrow}* f'$, for some f''	, from $f \stackrel{as}{\rightarrow}* f'$
$g \stackrel{a}{\rightarrow} g''$, where $f'' \mathcal{S} g''$	, $f \mathcal{S} g$, $\mathcal{S}$ is a strong bisimulation
$g'' \stackrel{s}{\rightarrow}* g'$ and $f' \mathcal{S} g'$	, $f'' \mathcal{S} g''$ and induction on s
$g \stackrel{as}{\rightarrow}* g'$ and $f' \mathcal{S} g'$	, from above two facts □

Corollary 3.1. Strongly bisimilar expressions have the same trace sets.

This theorem provides a convenient tool to show that two expressions have the same trace sets: define a relation $\mathcal{S}$ that includes the given expressions, and prove that $\mathcal{S}$ is a strong bisimulation. Bisimulation is an inductive (actually, co-inductive) way of establishing equality. Not all equalities can be proved using bisimulation. That is, Theorem 3.1 provides only a sufficient condition for the equality of trace sets; two expressions may have the same trace sets but they may not be strongly bisimilar. The example in Figure 3.2 (page 82) is from Milner [36]. The trace sets of X and P are identical, but they are not bisimilar, because neither Y nor Z is bisimilar to Q.

The following proposition is from Milner [36], Chapter 4. See this reference for the proofs (item (5), below, is not mentioned in that reference, but it is easy to prove from the others).

Proposition 3.1.

1. The identity relation is a strong bisimulation.
2. The inverse of a strong bisimulation is a strong bisimulation.
3. The relational product of strong bisimulations is a strong bisimulation.
4. The union of strong bisimulations is a strong bisimulation.
5. The reflexive, symmetric and transitive closure of a strong bisimulation is a strong bisimulation.

3.5.3 The Largest Strong Bisimulation

There are many strong bisimulations in general. Since our interest is in identifying as many bisimilar pairs of expressions as possible, we explore the possibility of defining a "largest" bisimulation. Indeed there is one such relation that we write as $\sim$. It is the union of all strong bisimulations. From the previous proposition, since the union of strong bisimulations is a strong bisimulation, $\sim$ is a strong bisimulation. The following proposition, from Milner [36], gives additional important properties of this relation.

Proposition 3.2.

1. $\sim$ is the largest strong bisimulation, i.e., any strong bisimulation is a subset of $\sim$.
2. $\sim$ is an equivalence relation.
3. (converse of the bisimilarity property for $\sim$) For any f and g, suppose

   for every $f \xrightarrow{a} f'$, there is some $g \xrightarrow{a} g'$ such that $f' \sim g'$, and
   for every $g \xrightarrow{b} g''$, there is some $f \xrightarrow{b} f''$ such that $f'' \sim g''$.

   Then, $f \sim g$.

3.6 Strong Equivalence

We expect two expressions to be equivalent if they may engage in the same sequence of events. But that is not enough. Equivalent expressions should be interchangeable; that is, they can be substituted for each other within any expression without altering the latter's behavior. To see the implication of this, consider expression x with only a free variable x and expression *stop*. Neither expression has a transition; so, $x \sim stop$. Yet, the two expressions may behave differently in the same context: $0 >x> x$ is different from $0 >x> stop$.

We overcome this difficulty by considering the various ways to bind the free variables of an expression. Binding encodes the context in which an expression f may be embedded, such as within $f <x< g$ where a publication of g binds the free variable x of f. We will consider a restricted class of bisimulations, the ones that are closed under bindings, for equivalence.

3.6.1 Binding-Closure

A relation $\mathcal{S}$ is *closed under binding*, or simply *closed*, if for every f, g and binding B,

$f \mathcal{S} g$ implies $B.f \mathcal{S} B.g$. □

So, $\sim$ is not closed because $x \sim stop$ but $(x := 0).x \not\sim (x := 0).stop$.

The following proposition is analogous to Proposition 3.1. The proofs are straightforward.

Proposition 3.3.

1. The identity relation is closed.
2. The inverse of a closed relation is closed.
3. The relational product of closed relations is closed.
4. The union of closed relations is closed.
5. The reflexive, symmetric and transitive closure of a closed relation is closed.

3.6.2 Definition of Strong Equivalence

The strong equivalence relation, written as $\cong$, is the union of all closed bisimulations. □

Example

We prove that $f \mid g \cong g \mid f$ for all f and g. We proved in Section 3.5.1 that $f \mid g \; \mathcal{S} \; g \mid f$ where $\mathcal{S}$ is a strong bisimulation. It remains to show that $\mathcal{S}$ is closed, i.e., given $f \mid g \; \mathcal{S} \; g \mid f$ for all f and g we show that $B.(f \mid g) \; \mathcal{S} \; B.(g \mid f)$ for all bindings B. From Table 3.2 (page 46), $B.(f \mid g) = B.f \mid B.g$ and $B.(g \mid f) = B.g \mid B.f$. Since $f \mid g \; \mathcal{S} \; g \mid f$ for all f and g, replacing f by $B.f$ and g by $B.g$, the desired result follows.

The following proposition is analogous to Proposition 3.2.

Proposition 3.4.

1. $\cong$ is the largest strong bisimulation that is closed under binding. That is, $\cong$ is a closed strong bisimulation and any closed strong bisimulation is a subset of $\cong$.
2. $\cong$ is a subset of $\sim$.
3. $\cong$ is an equivalence relation.
4. Given that $f \cong g$, for every binding B, $B.f \cong B.g$.
5. For expressions f, g and binding B, suppose that

   for every $B.f \xrightarrow{a} f'$, there is some $B.g \xrightarrow{a} g'$ such that $f' \cong g'$, and
   for every $B.g \xrightarrow{b} g''$, there is some $B.f \xrightarrow{b} f''$ such that $f'' \cong g''$.

   Then, $f \cong g$.

Proof: Proof of (1) is straightforward. To see (2), observe that $\cong$ is a bisimulation and $\sim$ is the largest bisimulation. For (3), let $\cong^*$ be the reflexive, symmetric and transitive closure of $\cong$. We show that $\cong^* = \cong$ which proves that $\cong$ is an equivalence relation.

Since $\cong$ is a strong bisimulation so is $\cong^*$, from Proposition 3.1. Further, since $\cong$ is closed so is $\cong^*$, from Proposition 3.3. Therefore, $\cong^*$ is a closed strong bisimulation. Since $\cong$ is the largest closed strong bisimulation, $\cong^* \subseteq \cong$. Now, $\cong \subseteq \cong^*$. Therefore, $\cong^* = \cong$.

Proof of (4) follows from the fact that $\cong$ is closed under binding.

To prove (5), for the given f and g define relation $\mathcal{S}$ as follows: $\mathcal{S} = \{(B.f, B.g) | \; B \text{ is any binding}\}$. We show that $(\cong \cup \mathcal{S})$ is a closed strong bisimulation.

First, we show that $\mathcal{S}$ is closed. Let $(\bar{f}, \bar{g}) \in \mathcal{S}$. Then $(\bar{f}, \bar{g}) = (B.f, B.g)$, for some binding B. For any binding B', we show that $(B'.\bar{f}, B'.\bar{g}) \in \mathcal{S}$. $(B'.\bar{f}, B'.\bar{g}) = (B'.(B.f), B'.(B.g)) = ((B'.B).f, (B'.B).g) \in \mathcal{S}$, by definition

of $\mathcal{S}$ because $(B'.B)$ is a binding. Since both $\cong$ and $\mathcal{S}$ are closed, $(\cong \cup \mathcal{S})$ is closed.

Next, $(\cong \cup \mathcal{S})$ is a strong bisimulation because the conditions for bisimilarity apply for every pair (f, g) in $\cong$ and in $\mathcal{S}$. Hence, $(\cong \cup \mathcal{S})$ is a closed strong bisimulation, and, therefore, a subset of $\cong$. Using empty binding B, $(B.f, B.g) = (f, g) \in \mathcal{S} \subseteq \cong$. Therefore, $f \cong g$. □

3.6.3 Strategies for Equivalence Proofs

In a proof of $P \cong Q$ expressions P and Q typically contain variables that denote subexpressions. For example, in proving $f \mid g \cong g \mid f$, the variables are f and g. These variables are not Orc variables, but *meta-variables* that denote that they may be replaced by any expression. We write $P(f, g, h)$ to denote that expression P contains meta-variables f, g and h, and perhaps others that are not of significance in the ensuing discussion. We construct two kinds of proofs in this chapter: (1) congruence proofs of the form $P(f) \cong P(g)$ given that $f \cong g$, and (2) identities of the form $P(f, g, h) \cong Q(f, g, h)$ given certain conditions on the meta-variables f, g and h.

Proof of an identity $P(f, g, h) \cong Q(f, g, h)$ involves the following steps: (1) postulate binary relation $\mathcal{S}$ by $P(f, g, h)\ \mathcal{S}\ Q(f, g, h)$, (2) show that $\mathcal{S}$ is a strong bisimulation, and (3) $\mathcal{S}$ is closed. Then, $\mathcal{S} \subseteq \cong$ and $P(f, g, h) \cong Q(f, g, h)$.

Though the definition of bisimulation can be applied directly, the proof is often long and tedious. Many proof steps can be omitted if P and Q have similar structures. Often the properties of Orc transitions can be exploited to omit certain proof steps. We develop a few such heuristics next.

3.6.3.1 Structure Preserving Transitions

We show that any transition of $P(f)$ arising out of a non-publication transition of f can be ignored in bisimulation proofs. Thus, the only transitions of f that need to be considered are publications and a transition due to rule (OTHH) of the otherwise combinator.

Observation 3.1. Let $P(f)$ be an expression in which f occurs at most once. Consider a non-publication transition $f \xrightarrow{a} f'$ that causes a transition of $P(f)$. Then the transition of $P(f)$ is $P(f) \xrightarrow{a} P(f')$. □

This observation claims that a non-publication transition causes f to be replaced by f' in $P(f)$; thus, the structure of P is retained. For example, given that $f \xrightarrow{a} f'$, $f \mid g \xrightarrow{a} f' \mid g$. This applies even when f is not an immediate subexpression of the larger expression, as in $(f \mid g) \gg h \xrightarrow{a} (f' \mid g) \gg h$.

The requirement that there be at most one occurrence of f eliminates invalid transitions such as $f \mid f \xrightarrow{a} f' \mid f'$. Further, a transition of f may not cause a transition of $P(f)$, as in $h \gg f$.

The proof of the observation is by induction on the structure of P. Consider the transitions in the semantic rules (DEFPASS), (PARL), (PARR), (SEQN),

(PRUL), (PRUN) and (OTHN). (Some of these rules also cover publication transitions though they must be considered in a bisimulation proof.) Each of them retain structure.

Lemma 3.1. Let $P(f)$ and $Q(f)$ have exactly one occurrence of f in each. Any non-publication transition of $f \xrightarrow{a} f'$ that causes a transition of $P(f)$ and $Q(f)$ may be omitted in bisimulation proof of $P(f) \sim Q(f)$.

Proof: A bisimulation proof of $P(f) \sim Q(f)$ starts by postulating relation $\mathcal{S}$ where $P(f)\ \mathcal{S}\ Q(f)$, for all f. We have to show that for any transition a of $P(f)$ there exists a corresponding transition of $Q(f)$ such that the resulting expressions P' and Q', respectively, satisfy $P'\ \mathcal{S}\ Q'$, and conversely for the transitions of $Q(f)$. Now, if a is a non-publication transition $f \xrightarrow{a} f'$ then:

$P(f) \xrightarrow{a} P(f')$	, from Observation 3.1
$Q(f) \xrightarrow{a} Q(f')$	, from Observation 3.1
$P(f')\ \mathcal{S}\ Q(f')$	, definition of $\mathcal{S}$

Therefore, the bisimulation conditions are always satisfied for such transitions. □

This lemma is particularly useful in dealing with sequential and pruning combinators.

3.6.3.2 Binding-Closure Proofs

Any equivalence proof, say of $P \cong Q$, requires both a proof of bisimulation and a proof of binding-closure: postulate a relation $\mathcal{S}$ by $P\ \mathcal{S}\ Q$, show that $\mathcal{S}$ is a strong bisimulation and that $\mathcal{S}$ is binding-closed, i.e., $B.P\ \mathcal{S}\ B.Q$ for every binding B. We can often eliminate the binding-closure proof. The key observation is that P and $B.P$ always have the same structure, only the meta-variables of P are replaced by their bounded versions to obtain $B.P$. For example, $f \mid g$ has the same structure as $B.(f \mid g)$ because, from Table 3.2 (page 46), $B.(f \mid g) = B.f \mid B.g$. Replacing f by $B.f$ and g by $B.g$ gives $B.(f \mid g)$. The more interesting cases are for the sequential and pruning combinators. Then, $B.(f >x> g) = B.f >x> (B\backslash x).g$ and $B.(f <x< g) = (B\backslash x).f <x< B.g$. So, the replacements are $(f := B.f, g := (B\backslash x).g)$ for the former and $(f := (B\backslash x)f, g := B.g)$ for the latter.

Henceforth, assume that every expression has at most one occurrence of any specific meta-variable. We show the meta-variables f and g, as in $P(f, g)$, only when we wish to emphasize their occurrence in P, other meta-variables of P are not shown.

The discussion above can be expressed as follows. For any B and $P(f, g)$, $B.P(f, g) = P(f', g')$ where f' is the transformation of f under binding B, and similarly g'. For ease of formal manipulation we write $f_{B,P(f,g)}$ for f'. Thus, $f_{B,f\ >x>\ g} = B.f$ and $g_{B,f\ >x>\ g} = (B\backslash x).g$. Then, for an expression P with meta-variable f, we have:

$$B.P(f) = P(f_{B,P(f)}) \qquad (*)$$

If P has more meta-variables, say f and g, $B.P(f,g) = P(f_{B,P(f,g)}, g_{B,P(f,g)})$.

Define P and Q to be *similar* if for every binding B and every meta-variable f, $f_{B,P} = f_{B,Q}$. Similarity means that every binding transforms P and Q to similar expressions. As an example, $f \mid g$ and $f \mathbf{;} g$ are similar because $f_{B,f \mid g} = B.f = f_{B,f} \mathbf{;}_g$ and $g_{B,f \mid g} = B.g = g_{B,f} \mathbf{;}_g$. However, $f >x> g$ and $g >x> f$ are not similar because $f_{B,f >x>_g} = B.f$ whereas $f_{B,g >x>_f} = (B\backslash x).f$. Note that similar expressions have the same meta-variables.

Lemma 3.2. Given $P \mathcal{S} Q$ and that P and Q are similar, $\mathcal{S}$ is closed.

Proof: For any P and Q such that $P \mathcal{S} Q$ we show that $B.P \mathcal{S} B.Q$. We prove the result when P and Q have a single meta-variable f; extensions to more variables is straightforward. We write $P(f)$ and $Q(f)$ for P and Q below.

$$
\begin{array}{ll}
 & B.P(f) \\
= & \{\text{Using } (*)\} \\
 & P(f_{B,P(f)}) \\
= & \{\text{From definition of similar, } f_{B,P(f)} = f_{B,Q(f)}\} \\
 & P(f_{B,Q(f)}) \\
\mathcal{S} & \{\text{Given: } P(f) \mathcal{S} Q(f) \text{ for all } f. \text{ Use } f_{B,Q(f)} \text{ for } f\} \\
 & Q(f_{B,Q(f)}) \\
= & \{\text{Using } (*)\} \\
 & B.Q(f)
\end{array}
$$

□

It is easy to establish if P and Q are similar by inspection. In fact, as we show next, similarity is assured in congruence proofs so that all binding-closure proofs can be eliminated.

Binding Closure in Congruence Proofs A congruence proof establishes $P(f) \cong P(g)$ given that $f \cong g$. The following lemma shows that the binding-closure part of the proof can be omitted. First, note that

$$f_{B,P(f)} \cong g_{B,P(g)} \qquad (**)$$

because the same binding B' is applied to f and g in $P(f)$ and $P(g)$, respectively, so that $f_{B,P(f)} = B'.f$ and $g_{B,P(g)} = B'.g$, for some B'. And, given $f \cong g$, $B'.f \cong B'.g$, from Proposition 3.4, part(4) (page 57).

Lemma 3.3. Let $P(f) \mathcal{S} P(g)$ for all f and g such that $f \cong g$. Then $\mathcal{S}$ is closed.

Proof: P and Q may have meta-variables other than f. Let h be one such variable. We show the proof when h is the only such variable; extensions to more variables is straightforward.

Given is relation $\mathcal{S}$ where $P(f,h) \mathcal{S} P(g,h)$ for all f and g such that $f \cong g$ and all h. We show that $B.P(f,h) \mathcal{S} B.P(g,h)$ for all f and g such that $f \cong g$.

$$
\begin{array}{ll}
 & B.P(f,h) \\
= & \{\text{Using } (*)\} \\
 & P(f_{B,P(f,h)}, h_{B,P(f,h)}) \\
= & \{h_{B,P(f,h)}) = h_{B,P(g,h)}, \text{ the terms depend only on the structure of } P\} \\
 & P(f_{B,P(f,h)}, h_{B,P(g,h)}) \\
\mathcal{S} & \{\text{From } (**),\ f_{B,P(f)} \cong g_{B,P(g)}.\ \text{Using the definition of } \mathcal{S}\ \} \\
 & P(g_{B,P(g)}, h_{B,P(g,h)}) \\
= & \{\text{Using } (*)\} \\
 & B.P(g,h) \qquad \square
\end{array}
$$

3.6.3.3 Fundamental Identities with the Parallel Combinator

The next section, Section 3.6.3.4, depends on a number of identities involving the parallel combinator which we list below.

Parallel-combinator Identities

1. (zero) $f \mid stop \cong f$ and $stop \mid f \cong f$
2. (commutativity) $f \mid g \cong g \mid f$
3. (associativity) $f \mid (g \mid h) \cong (f \mid g) \mid h$
4. (congruence) Given that $f \cong g$, we have $f \mid h \cong g \mid h$ and $h \mid f \cong h \mid g$, for any h.

We have already seen the proof of commutativity in the examples in Sections 3.5.1 and Section 3.6. The remaining identities are proved similarly. These identities also hold if $\cong$ is replaced by $\sim$.

3.6.3.4 Bifurcation

In contrast to non-publication transitions that retain the structure of the surrounding expression (see Section 3.6.3.1), publications always change the structure. The most problematic involves the sequential combinator. For instance, in proving $f >x> h \cong g >x> h$ given that $f \cong g$, we postulate relation $\mathcal{S}$ by $f >x> h \ \mathcal{S}\ g >x> h$ for all f, g and h where $f \cong g$. In proving that $\mathcal{S}$ is a bisimulation, we will have to consider the publication transition $f \xrightarrow{!c} f'$. There is a matching transition of g, $g \xrightarrow{!c} g'$ since $f \cong g$. These transitions, respectively, cause:

$f >x> h \xrightarrow{\tau} f' >x> h \mid (x := c).h$, and
$g >x> h \xrightarrow{\tau} g' >x> h \mid (x := c).h$,

Now, we need to show $f' >x> h \mid (x := c).h \ \mathcal{S}\ g' >x> h \mid (x := c).h$. But this proof can not be completed because the structures of the expressions are now different so that relation $\mathcal{S}$ does not hold over them. A similar problem arises in proofs with the pruning combinator when the right side expression

publishes. One possible solution for this problem is to do the bisimulation proof with a more general relation $\mathcal{S}'$ where $f >x> h \mid p\ \mathcal{S}'\ g >x> h \mid q$ for all f, g, h, p and q where $f \cong g$ and $p \cong q$. This complicates the proof considerably. We prove a theorem that ameliorates this problem. The theorem is a generalization of "upto bisimulation" result of Milner (see Chapter 4, Lemma 5 in [36]), but it applies only to Orc expressions.

Definition: Bifurcation For relations S and T their *bifurcation* is a relation. It is written as $S \mid T$ and is given by: $f\ (S \mid T)\ g$ if there exist f_0, f_1, g_0 and g_1 such that

$$f \cong f_0 \mid f_1,\ g \cong g_0 \mid g_1,\ f_0\ S\ g_0, \text{ and } f_1\ T\ g_1$$

Bifurcation over more than two relations is similarly defined. We often write

$$f \cong f_0 \mid f_1\ (S \mid T)\ g_0 \mid g_1 \cong g$$

to express the relationships between the expressions.

The following proposition lists a number of useful properties of bifurcation that are used in the sequel. Here, R, S and T are binary relations and id is the identity relation.

Proposition 3.5. (Properties of Bifurcation)

1. Bifurcation is commutative, i.e, $(S \mid T) = (T \mid S)$.
2. Bifurcation is associative, i.e, $R \mid (S \mid T) = (R \mid S) \mid T$.
3. Bifurcation is monotonic in both arguments, i.e., given that $S \subseteq S'$ and $T \subseteq T'$, $(S \mid T) \subseteq (S' \mid T)$ and $(S \mid T) \subseteq (S \mid T')$.
4. $S \subseteq (id \mid S)$ and $S \subseteq (S \mid \cong)$.
5. $\cong\ =\ (id \mid \cong)\ =\ (\cong \mid \cong)$
6. If both S and T are binding-closed, then so is $(S \mid T)$.

Proof: Proofs of parts(1), (2) and (3) are straightforward. For part(4), we show that given $f\ S\ g$, we have $f\ (id \mid S)\ g$. This is because

$$\begin{array}{ll} & f\ S\ g \\ \Rightarrow & \{id \text{ is reflexive, so } stop\ id\ stop\text{; definition of bifurcation}\} \\ & stop \mid f\ (id \mid S)\ stop \mid g \\ \Rightarrow & \{f \cong stop \mid f,\ g \cong stop \mid g \text{ from Section 3.6.3.3}\} \\ & f\ (id \mid S)\ g \end{array}$$

The other part of part(4) follows:

$$
\begin{array}{ll}
 & S \\
\subseteq & \{\text{first part of (4)}\} \\
 & id \mid S \\
\subseteq & \{id \subseteq \cong \text{ , monotonicity of bifurcation (part 3)}\} \\
 & \cong \mid S \\
= & \{\text{commutativity of bifurcation (part 1)}\} \\
 & S \mid \cong
\end{array}
$$

For part(5), first we show that $\cong \subseteq (id \mid \cong) \subseteq (\cong \mid \cong)$.

$$
\begin{array}{ll}
S \subseteq (id \mid S) & \text{, first part of part(4)} \\
\cong \subseteq (id \mid \cong) & \text{, using } \cong \text{ in place of } S \text{ above} \\
(id \mid \cong) \subseteq (\cong \mid \cong) & \text{, monotonicity (part 3) with } id \subseteq \cong \\
\cong \subseteq (id \mid \cong) \subseteq (\cong \mid \cong) & \text{, from above two}
\end{array}
$$

Now, we show that $(\cong \mid \cong) \subseteq \cong$. Combined with $\cong \subseteq (id \mid \cong) \subseteq (\cong \mid \cong)$, proven above, this completes the proof. We show that for any f and g where $f\ (\cong \mid \cong)\ g$, we have $f \cong g$.

$$
\begin{array}{ll}
f\ (\cong \mid \cong)\ g & \text{, given} \\
f \cong f_0 \mid f_1\ (\cong \mid \cong)\ g_0 \mid g_1 \cong g & \\
 & \text{, definition of bifurcation} \\
f_0 \cong g_0 \text{ and } f_1 \cong g_1 & \text{, from above} \\
f_0 \mid f_1 \cong g_0 \mid g_1 & \text{, congruence result from Section 3.6.3.3} \\
f \cong g & \text{, } f \cong f_0 \mid f_1 \text{ and } g \cong g_0 \mid g_1
\end{array}
$$

Proof of part 6 is straightforward, using the fact from Table 3.2 that $B.(f_0 \mid f_1) = B.f_0 \mid B.f_1$.

In preparation for the main theorem, we prove the following lemma.

Lemma 3.4. Let $\mathcal{S}$ be a binary relation such that given $f\ \mathcal{S}\ g$
for every $f \xrightarrow{a} f'$ there is some $g \xrightarrow{a} g'$ such that $f'\ (\mathcal{S} \mid \cong)\ g'$, and
for every $g \xrightarrow{a} g''$ there is some $f \xrightarrow{a} f''$ such that $f''\ (\mathcal{S} \mid \cong)\ g''$.
Then, $(\mathcal{S} \mid \cong)$ is a strong bisimulation.

Proof: We show that the conditions in the definition of strong bisimulation are met by $(\mathcal{S} \mid \cong)$. Let $f\ (\mathcal{S} \mid \cong)\ g$. For any $f \xrightarrow{a} f'$ we show $g \xrightarrow{a} g'$ such that $f'\ (\mathcal{S} \mid \cong)\ g'$, and conversely. This proves the result.

Given $f\ (\mathcal{S} \mid \cong)\ g$, we have $f \cong f_0 \mid f_1\ (\mathcal{S} \mid \cong)\ g_0 \mid g_1 \cong g$. From $f \cong f_0 \mid f_1$ and $f \xrightarrow{a} f'$, since $\cong$ is a bisimulation, there is some $f_0 \mid f_1 \xrightarrow{a} f_2$ such that $f' \cong f_2$. Any transition of $f_0 \mid f_1$ is due to either a transition of f_0 or f_1. We consider the two cases separately.

- $f_0 \xrightarrow{a} f_0'$:

$$
\begin{array}{ll}
f_0 \mid f_1 \xrightarrow{a} f_0' \mid f_1 = f_2 & \text{, } f_0 \xrightarrow{a} f_0' \text{ and rule (PARL)} \\
f' \cong f_0' \mid f_1 & \text{, } f_2 = f_0' \mid f_1 \\
\text{there is some } g_0 \xrightarrow{a} g_0' \text{ where } f_0'\ (\mathcal{S} \mid \cong)\ g_0' &
\end{array}
$$

	, $f_0 \ \mathcal{S} \ g_0$ and lemma conditions
$g_0 \mid g_1 \xrightarrow{a} g_0' \mid g_1$	, $g_0 \xrightarrow{a} g_0'$ and rule (PARL)
$f_0' \mid f_1 \ ((\mathcal{S} \mid \cong) \mid \cong) \ g_0' \mid g_1$	, $f_0' \ (\mathcal{S} \mid \cong) \ g_0'$, $f_1 \cong g_1$
$f_0' \mid f_1 \ (\mathcal{S} \mid (\cong \mid \cong)) \ g_0' \mid g_1$	, associativity of bifurcation
$f_0' \mid f_1 \ (\mathcal{S} \mid \cong) \ g_0' \mid g_1$	, $(\cong \mid \cong) = \cong$ from Proposition 3.5, Part(5)
$f' \ (\mathcal{S} \mid \cong) \ g'$	, $f' \cong f_0' \mid f_1$, where $g' \cong g_0' \mid g_1$

• $f_1 \xrightarrow{a} f_1'$:

there is some $g_1 \xrightarrow{a} g_1'$ where $f_1' \cong g_1'$	
	, from $f_1 \cong g_1$
$g_0 \mid g_1 \xrightarrow{a} g_0 \mid g_1'$	, $g_1 \xrightarrow{a} g_1'$ and rule (PARR)
$f_0 \mid f_1 \xrightarrow{a} f_0 \mid f_1'$	, $f_1 \xrightarrow{a} f_1'$ and rule (PARR)
$f_0 \mid f_1' \ (\mathcal{S} \mid \cong) \ g_0 \mid g_1'$	, $f_0 \ \mathcal{S} \ g_0$, $f_1' \cong g_1'$
$f' \ (\mathcal{S} \mid \cong) \ g'$	, $f' \cong f_0 \mid f_1'$, let $g' \cong g_0 \mid g_1'$

Theorem 3.2. (Bifurcation Theorem) Let $\mathcal{S}$ be a binary relation such that given $f \ \mathcal{S} \ g$

for every $f \xrightarrow{a} f'$ there is some $g \xrightarrow{a} g'$ such that
$f' \ \mathcal{S} \ g'$, $f' \cong g'$, or $f' \ (\mathcal{S} \mid \cong) \ g'$, and (C1)

for every $g \xrightarrow{a} g''$ there is some $f \xrightarrow{a} f''$ such that
$f'' \ \mathcal{S} \ g''$, $f'' \cong g''$, or $f'' \ (\mathcal{S} \mid \cong) \ g''$. (C2)

Then, $f \sim g$. Further, if $\mathcal{S}$ is closed then $f \cong g$.

Proof: First, note that the conditions (C1, C2) hold trivially for relation id. Next, that if the conditions (C1, C2) hold for any two relations then they hold for the union of the relations. Therefore, the conditions hold for $\mathcal{S}' = \mathcal{S} \cup id$.

Now we show that both relations $\mathcal{S}'$ and $\cong$ are subsets of $(\mathcal{S}' \mid \cong)$.

$\mathcal{S}'$
$\subseteq$ {from Proposition 3.5, part 4}
$\mathcal{S}' \mid \cong$

$\cong$
$\subseteq$ {Proposition 3.5, part 5 }
$id \mid \cong$
$\subseteq$ {$id \subseteq \mathcal{S}'$, Proposition 3.5, part 3 }
$\mathcal{S}' \mid \cong$

Therefore, condition (C1) can be simply written as:

given $f \ \mathcal{S}' \ g$ for every $f \xrightarrow{a} f'$ there is some $g \xrightarrow{a} g'$ such that $f' \ (\mathcal{S}' \mid \cong) \ g'$, and similarly for (C2).

Apply Lemma 3.4 to conclude that $(\mathcal{S}' \mid \cong)$ is a bisimulation. Now $\mathcal{S} \subseteq \mathcal{S}'$, and from Proposition 3.5, part(4), $\mathcal{S}' \subseteq (\mathcal{S}' \mid \cong)$. So, $\mathcal{S} \subseteq (\mathcal{S}' \mid \cong)$. Hence $\mathcal{S}$ is a subset of a bisimulation; so, for every $f \ \mathcal{S} \ g$, $f \sim g$. Further, if $\mathcal{S}$ is closed then so is $\mathcal{S}'$ because $\mathcal{S}'$ is the union of two relations that are closed. Then so is $(\mathcal{S}' \mid \cong)$, from Proposition 3.5, part 6. So, if $\mathcal{S}$ is closed $(\mathcal{S}' \mid \cong)$ is a closed bisimulation, i.e., $(\mathcal{S}' \mid \cong) \subseteq \cong$. Thus, $\mathcal{S} \subseteq \cong$; so, for every $f \ \mathcal{S} \ g$, $f \cong g$. □

Note: The theorem does *not* establish that S is a strong bisimulation nor that it is closed, merely that $\mathcal{S} \subseteq \cong$.

Caveats in Equivalence Proofs A proof of $f \cong g$ has to display a relation $\mathcal{S}$ such that $\mathcal{S} \subseteq \cong$ and $f \ \mathcal{S}\ g$. For the first part, show that $\mathcal{S}$ is a closed strong bisimulation, or, more generally, that it is a subset of a closed strong bisimulation. It is *not* sufficient to display $\mathcal{S}$ that is: (1) a closed subset of a strong bisimulation, or (2) a strong bisimulation and a subset of a closed set. Given $f \ \mathcal{S}\ g$ we can not conclude that $f \cong g$ in either case. The first scenario merely implies that $\mathcal{S}$ is closed but not necessarily a bisimulation. For a counterexample with the second scenario, use $\sim$ for $\mathcal{S}$; $\sim$ is a strong bisimulation and it is a subset of the universal relation that includes all pairs of expressions and, hence, closed.

3.7 Asynchronous Semantics: Identities

An equivalence relation is a *congruence* if equivalent expressions can be substituted for each other in any context. To show that $\cong$ is a congruence we have to show that $P(f) \cong P(g)$ whenever $f \cong g$. It is sufficient to show that every Orc combinator preserves $\cong$, that is, given $f \cong g$, $f * h \cong g * h$ and $h * f \cong h * g$, for every expression h and every Orc combinator $*$; see Table 3.3. We give a sample proof in Section 3.7.2.

The fact that $\cong$ is a congruence is fundamental to Orc, both in theory and practice. It permits us to substitute strongly equivalent expressions for each other in any context. A number of optimizations in the Orc interpreter depend on this result.

We state and prove a number of other identities in Sections 3.7.3 and 3.7.4. First, we define the notions of *silent* and *halting* formally.

3.7.1 Silent, Halting

In Section 2.3, we called an expression *silent* if it never publishes. We can now define the term formally. Expression n is *silent* if $n \cong stop \ll n$. It can be shown, using bisimulation and binding-closure, that n is silent if and only if $n \cong n \gg stop$.

Expression f *halts* (or f is halting) means $f \cong stop$.

A halting expression is clearly silent. However, an expression may be silent but not halting. Consider $metronome(1) \gg stop$, where $metronome(t)$, as defined in Section 2.5.2.2 (page 34), publishes a signal every t time units. Execution of $metronome(1) \gg stop$ continues indefinitely without publishing a value; so, this expression is silent though not halted. It is quite common in Orc programs for expressions to repeatedly read data from a channel, say, and engage in computations that write to other channels, but never publish.

We can describe may other characterstics of expressions using the equivalence relation. For example, that f never publishes and never eventually halts is

expressed by $f \gg stop \;\cong\; f; g$, for all g. That f publishes if it eventually halts is given by $f; g \;\cong\; f$, for all g. A more elaborate property is f publishes once and then halts or never publishes is expressed by $f >x> x \;\cong\; x <x< f$.

3.7.2 Strong Equivalence is a Congruence

We show that all Orc combinators preserve strong equivalence, i.e., given that $f \;\cong\; g$, the identities in Table 3.3 hold for all h. We can obtain similar identities by replacing $>x>$ by $\gg$ and $<x<$ by $\ll$. Further, prefixing a definition D to equivalent expressions results in equivalent expressions. And, a definition D in which the bodies are equivalent expressions and the site name and parameter lists are identical are equivalent. Below, D is any definition.

$f \mid h$	$\cong$	$g \mid h$		$h \mid f$	$\cong$ $h \mid g$
$f >x> h$	$\cong$	$g >x> h$		$h >x> f$	$\cong$ $h >x> g$
$f <x< h$	$\cong$	$g <x< h$		$h <x< f$	$\cong$ $h <x< g$
$f \,;\, h$	$\cong$	$g \,;\, h$		$h \,;\, f$	$\cong$ $h \,;\, g$
$D \,\#\, f$	$\cong$	$D \,\#\, g$		$(def\ E(\bar{x}) = f) \,\#\, h$	$\cong$ $(def\ E(\bar{x}) = g) \,\#\, h$

Table 3.3: Strong Equivalence Preservation

The congruences involving the parallel combinator have been discussed in Section 3.6.3.3 (page 61).

Simplifications in Congruence Proofs Congruence proofs admit several simplifications. In proving $P(f) \;\cong\; P(g)$ given $f \;\cong\; g$, it is typical to define relation $\mathcal{S}$ by $P(f)\ \mathcal{S}\ P(g)$ for all f and g where $f \;\cong\; g$, and then prove that $\mathcal{S}$ is a closed strong bisimulation. The bisimulation proof need only show that for every transition of f, $f \xrightarrow{a} f'$, there is a transition $g \xrightarrow{a} g'$ such that resulting states of $P(f)$ and $P(g)$ are equivalent; the corresponding proof for every transition of g is unnecessary because by reversing the roles of f and g the same proof suffices. Next, slightly extending the result of Lemma 3.1 (page 59), there is no need to consider the non-publication transitions of f. Finally, we can skip the proof that $\mathcal{S}$ is closed, using Lemma 3.3 (page 60).

Proof of a congruence, $f >x> h \;\cong\; g >x> h$, given $f \;\cong\; g$: We prove one of the hardest congruences, $f >x> h \;\cong\; g >x> h$. The proof itself is very short because of the simplifications discussed above. The proof is further simplified by applying the Bifurcation Theorem (Theorem 3.2, page 64).

Define $f >x> h \;\; \mathcal{S} \;\; g >x> h$, where $f \;\cong\; g$ and h is arbitrary. We show that $\mathcal{S} \;\subseteq\; \cong$, thus proving this result.

In the bisimulation part of the proof, we need consider only the transitions of f and g only, because only such transitions cause transitions of $f >x> h$ and $g >x> h$. We consider only transitions of f and omit those of g, by symmetry, as discussed above under simplifications. Further, non-publication transitions

$$
\begin{array}{lcl}
\mathit{free}(\mathit{stop}) & = & \{\} \\
\mathit{free}(E(\bar{p})) & = & \mathit{free}(\bar{p}) \\
\mathit{free}(e(\bar{p})) & = & \mathit{free}(\bar{p}) \cup \{e\} \\
\mathit{free}(f \mid g) & = & \mathit{free}(f) \cup \mathit{free}(g) \\
\mathit{free}(f >x> g) & = & \mathit{free}(f) \cup (\mathit{free}(g) - \{x\}) \\
\mathit{free}(f <x< g) & = & (\mathit{free}(f) - \{x\}) \cup \mathit{free}(g) \\
\mathit{free}(f \,;\, g) & = & \mathit{free}(f) \cup \mathit{free}(g)
\end{array}
$$

Table 3.4: Free variables; e is a variable and E a bound name.

of f can be ignored, as discussed in Lemma 3.1 (page 59). Also, binding closure proof can be omitted for a congruence. The only remaining part is to consider the publication transition of f, $f \stackrel{!c}{\rightarrow} f'$, and show that the conditions of the bifurcation theorem are met.

$$
\begin{array}{ll}
f \stackrel{!c}{\rightarrow} f' & \text{, assume} \\
f >x> h \stackrel{\tau}{\rightarrow} f' >x> h \mid (x := c).h & \text{, semantic rule (SEQP)} \\
\text{there is some } g \stackrel{!c}{\rightarrow} g' \text{ where } f' \cong g' & \text{, from } f \cong g \\
g >x> h \stackrel{\tau}{\rightarrow} g' >x> h \mid (x := c).h & \text{, semantic rule (SEQP)} \\
f' >x> h \;\; \mathcal{S} \;\; g' >x> h & \text{, } f' \cong g' \text{ and definition of } \mathcal{S} \\
f' >x> h \mid (x := c).h \; (\mathcal{S} \mid \cong) \; g' >x> h \mid (x := c).h & \\
 & \text{, definition of bifurcation} \\
\mathcal{S} \subseteq \cong & \text{, Bifurcation theorem (page 64)}
\end{array}
$$

3.7.3 Identities

The identities, given below, are grouped into five categories, one for each of the four combinators and a category about the distributivity of one combinator over another. The identities for $>x>$ and $<x<$ can be specialized by replacing the combinators by $\gg$ and $\ll$, respectively, and dropping the constraints, if any, on x.

In the identities, we write $\mathit{free}(f)$ for the set of free variables of f, formally defined in Table 3.4. In that table, for a list of parameters $\bar{p}$, $\mathit{free}(\bar{p})$ is the set of variables in $\bar{p}$. Note that $x \notin \mathit{free}(f)$ implies that $B.f = (B \backslash x).f$, for any binding B. And, $\mathit{free}(B.f) \subseteq \mathit{free}(f)$, for any binding B because a binding does not introduce new variables.

Parallel Combinator We repeat these identities here from Section 3.6.3.3 (page 61) for completeness.

1. (zero of $\mid$) $f \mid \mathit{stop} \cong f$ and $\mathit{stop} \mid f \cong f$
2. (commutativity of $\mid$) $f \mid g \cong g \mid f$
3. (associativity of $\mid$) $f \mid (g \mid h) \cong (f \mid g) \mid h$

Sequential Combinator

1. (left zero of $>x>$) $\mathit{stop} >x> f \cong \mathit{stop}$
2. (associativity of $>x>$) $f >x> (g >y> h) \cong (f >x> g) >y> h$, if $x \notin \mathit{free}(h)$

The first identity shows that *stop* is a left zero. Note that *stop* is not a right zero, $f >x> \mathit{stop} \ncong \mathit{stop}$. Associativity shows that under a side condition, $x \notin \mathit{free}(h)$, this combinator is associative. (By convention, $f >x> g >y> h$ is interpreted as $(f >x> g) >y> h$.) The side condition is required: consider $(0 >x> x) >y> \mathit{add}(x, y)$ and $0 >x> (x >y> \mathit{add}(x, y))$, where *add* publishes the sum of its parameters. In $(0 >x> x) >y> \mathit{add}(x, y)$, the parameter of *add* is a free variable of the entire expression, whereas in $0 >x> (x >y> \mathit{add}(x, y))$ it is bound. This difference is exploited in the binding $(x := 1)$. Then, $(x := 1).((0 >x> x) >y> \mathit{add}(x, y)) = (0 >x> x) >y> \mathit{add}(1, y)$, which publishes $1 + 0 = 1$. And, $(x := 1).(0 >x> (x >y> \mathit{add}(x, y)) = 0 >x> (x >y> \mathit{add}(x, y))$, which publishes $0 + 0 = 0$.

Pruning Combinator The following identities hold for all f, g, h and silent n; see Section 3.7.1 (page 65) for definition of silent.

1. (right zero of $\ll$) $f \ll \mathit{stop} \cong f$
2. (generalization of right zero) $f \ll g \cong f \ll (\mathit{stop} \ll g) \cong f \mid (\mathit{stop} \ll g)$
3. (relation between $\ll$ and $<x<$) $f \ll g \cong f <x< g$, if $x \notin \mathit{free}(f)$.
4. (commutativity) $(f <x< g) <y< h \cong (f <y< h) <x< g$ if $x \notin \mathit{free}(h)$, $y \notin \mathit{free}(g)$, and x, y are distinct.

From (1) *stop* behaves as a right zero of $\ll$. A generalization of (1) appears in (2); the term $\mathit{stop} \ll g$ behaves as g but never publishes. The relationship between $\ll$ and $<x<$ is shown in (3). Identity (4) shows that the combinator is commutative under mild restrictions. In that identity, distinctness of x and y is important: $(x <x< 2) <x< 3 \ncong (x <x< 3) <x< 2$.

The associativity identity, $(f <x< g) <y< h \cong f <x< (g <y< h)$, does not hold. Here, $g \xrightarrow{!c} g'$ transforms the left expression to $(x := c).f <y< h$ and the right one to $(x := c).f$, and these two expressions are not equivalent.

Observe that $f <x< \mathit{stop} \xrightarrow{\tau} (x := \not{\mathit{1}}).f$, from (PRUH). So, the two expressions are not strongly equivalent; they can be shown to be weakly equivalent, using the definition of weak equivalence from Section 3.8 (page 72).

We can derive the following identities from the ones given above.

- *stop* is silent, from (right zero) using *stop* for f and the definition of silent.
- $\mathit{stop} \ll g$ is silent, from part (2) using *stop* for f. In fact, $\mathit{stop} <x< g$ is also silent.

- (pruning with silent expression) $f \ll n \;\cong\; f \mid n$, from part (2) and definition of silent.
- $(f \ll g) \ll h \;\cong\; (f \ll h) \ll g$, from commutativity.
- (Generalization of part 3) $f \ll g \;\cong\; f <x< g$, if $(x := \not!).f \;\cong\; f$:

$$\begin{array}{ll} & f <x< g \\ \cong & \{(x := \not!).f \;\cong\; f\} \\ & (x := \not!).f <x< g \\ \cong & \{x \notin \mathit{free}((x := \not!).f).\ \text{From part (3) above}\} \\ & (x := \not!).f \ll g \\ \cong & \{(x := \not!).f \;\cong\; f\} \\ & f \ll g \end{array}$$

Otherwise Combinator

1. (associativity of ;) $(f \,;\, g) \,;\, h \;\cong\; f \,;\, (g \,;\, h)$

This identity is a generalization of the similar identity for sequential composition in traditional programming languages. Observe that $\mathit{stop} \,;\, f \ncong f$ because the left expression produces a τ event before it starts behaving like the right expression f. Similarly, $f \,;\, \mathit{stop} \ncong f$ because in case $f = \mathit{stop}$, the left expression generates a τ event whereas the right expression has no transition at all. If we ignore the τ events, then these become identities. The expressions are then said to be *weakly equivalent*, a notion we define in Section 3.8 (page 72).

Distributivity The following identities show how one combinator distributes over another.

1. ($\mid$ over $>x>$; left distributivity)
   $(f \mid g) >x> h \;\cong\; f >x> h \mid g >x> h$
2. ($\mid$ over $<x<$) $(f \mid g) <x< h \;\cong\; (f <x< h) \mid g$, if $x \notin \mathit{free}(g)$
3. ($>y>$ over $<x<$) $(f >y> g) <x< h \;\cong\; (f <x< h) >y> g$
   if $x \notin \mathit{free}(g)$, and x and y are distinct.
4. ($<x<$ over otherwise) $(f <x< g) \,;\, h \;\cong\; (f \,;\, h) <x< g$, if $x \notin \mathit{free}(h)$.

Distributivity identities are the most difficult ones to justify intuitively; therefore, formal proofs are essential. There seems to be no distributivity identity of ($\mid$ over ;) or ($>x>$ over ;).

There is no "right distributivity" identity of $\mid$ over $>x>$ similar to the left distributivity: $f >x> (g \mid h) \;\cong\; f >x> g \mid f >x> h$. This identity holds only with additional restrictions on f, that it be "functional". That is, f publishes at most one value and the same value each time and it calls only functional sites or common sites. Thus, an expression that returns a random number is not functional, nor is one that creates a site and publishes its identity.

3.7.4 Proofs of the Identities

The proofs of the identities follow the pattern we have shown in Section 3.7.2 (page 66). The proofs often appeal to the bifurcation theorem, Theorem 3.2 (page 64). Further, some of the proofs make use of the other identities. In particular, associativity of the sequential combinator, proven below, assumes the left distributivity of $\mid$ over $>x>$.

Proving Associativity of the Sequential Combinator: We show

$$f >x> (g >y> h) \cong (f >x> g) >y> h, \text{ if } x \notin \mathit{free}(h).$$

Define $f >x> (g >y> h)\ \mathcal{S}\ (f >x> g) >y> h$, for every f, g and h where $x \notin \mathit{free}(h)$. We show that $\mathcal{S} \subseteq \cong$, thus proving this result. The main tool is the bifurcation theorem, Theorem 3.2. By inspection of the two expressions we can establish that they are similar (see Section 3.6.3.2, page 59), therefore, according to Lemma 3.2, the binding-closure condition of the theorem is met. To establish the other conditions of the theorem, we need consider only the publication transitions, according to Lemma 3.1 (page 59), and only those of f because only the transitions of f cause transitions in the expressions on both sides.

$f \xrightarrow{!c} f'$, assume
$f >x> (g >y> h) \xrightarrow{\tau} f' >x> (g >y> h) \mid (x := c).(g >y> h)$
, semantic rule (SEQP)
$f >x> (g >y> h) \xrightarrow{\tau} f' >x> (g >y> h) \mid ((x := c).g) >y> h$
, from Table 3.2 (page 46) and $x \notin \mathit{free}(h)$

We show a matching transition of the right expression:

$f >x> g \xrightarrow{\tau} f' >x> g \mid (x := c).g$
, from $f \xrightarrow{!c} f'$ and rule (SEQP)
$(f >x> g) >y> h \xrightarrow{\tau} (f' >x> g \mid (x := c).g) >y> h$
, from above and rule (SEQN)
$(f >x> g) >y> h \xrightarrow{\tau} (f' >x> g) >y> h \mid ((x := c).g) >y> h$
, left distributivity of $\mid$ over $>x>$
$f' >x> (g >y> h) \mid ((x := c).g) >y> h \quad (\mathcal{S} \mid \cong)$
$(f' >x> g) >y> h \mid ((x := c).g) >y> h$, definition of bifurcation
$\mathcal{S} \subseteq \cong$, Bifurcation Theorem (page 64)

3.7.5 Halting Revisited

We show the conditions under which an expression halts, that is it is equivalent to *stop*. These conditions can be used to implement the semantic rule (PRUH) for the pruning combinator, (OTHH) for the otherwise combinator and to send a negative response to an internal site call. Halting of any Orc expression can be detected using these rules provided the expression calls only helpful sites.

1. If f is a (strict) site call in which x is a parameter then $(x := \not\prime).f$ halts, from the definition of strict.
2. If $f \cong stop$ and $g \cong stop$, then $f \mid g \cong stop$, from the identities of the parallel combinator.
3. If $f \cong stop$, then $f >x> g \cong stop$, from the identities of the sequential combinator.

Expression $f <x< g$ never halts because (1) if g does not halt it makes some transition, and so does $f <x< g$, and (2) if g halts, then $f <x< g$ makes a τ transition according to rule (PRUH).

Expression $f \,;\, g$ never halts: if f never halts, then either rule (OTHN) or (OTHP) can be applied, and if f halts, then (OTHH) can be applied.

3.7.6 A Summary of the Identities

- (Identities of $\mid$)
  1. (zero of $\mid$) $f \mid stop \cong f$ and $stop \mid f \cong f$
  2. (commutativity of $\mid$) $f \mid g \cong g \mid f$
  3. (associativity of $\mid$) $f \mid (g \mid h) \cong (f \mid g) \mid h$
- (Identities of $>x>$)
  1. (left zero of $>x>$) $stop >x> f \cong stop$
  2. (associativity of $>x>$) $f >x> (g >y> h) \cong (f >x> g) >y> h$, if $x \notin free(h)$.
- (Identities of $<x<$)
  1. (right zero of $\ll$) $f \ll stop \cong f$
  2. (generalization of right zero) $f \ll g \cong f \ll (stop \ll g) \cong f \mid (stop \ll g)$
  3. (relation between $\ll$ and $<x<$) $f \ll g \cong f <x< g$, if $x \notin free(f)$.
  4. (commutativity) $(f <x< g) <y< h \cong (f <y< h) <x< g$ if $x \notin free(h)$, $y \notin free(g)$, and x, y are distinct.
- (Identities of $\,;\,$)
  1. (associativity of $\,;\,$) $(f \,;\, g) \,;\, h \cong f \,;\, (g \,;\, h)$
- (Distributivity Identities)
  1. ($\mid$ over $>x>$; left distributivity) $(f \mid g) >x> h \cong f >x> h \mid g >x> h$
  2. ($\mid$ over $<x<$) $(f \mid g) <x< h \cong (f <x< h) \mid g$, if $x \notin free(g)$.
  3. ($>y>$ over $<x<$) $(f >y> g) <x< h \cong (f <x< h) >y> g$ if $x \notin free(g)$, and x and y are distinct.
  4. ($<x<$ over otherwise) $(f <x< g) \,;\, h \cong (f \,;\, h) <x< g$, if $x \notin free(h)$.

3.8 Weak Bisimulation

We remarked earlier that *stop* **;** f is not strongly equivalent to f because the former produces a τ event and only then behaves like f; the two expressions would be identical if we ignore this τ event. To this end, we define weak bisimulation and weak equivalence by ignoring the τ events. Weak bisimulation and weak equivalence have many of the properties of their strong counterparts. In particular, weakly equivalent expressions may be substituted for each other within any expression and the resulting expressions are weakly equivalent.

3.8.1 Definition of Weak Bisimulation

We define weak bisimulation in a manner analogous to strong bisimulation. For sequence of events s and t, let $s \stackrel{\setminus\tau}{=} t$ denote that s and t are identical after removing τ events from both sequences. Note that $\stackrel{\setminus\tau}{=}$ is an equivalence relation. The definition of weak bisimulation makes use of $\rightarrow*$, the transitive closure of $\rightarrow$, defined in Section 3.4.8 (page 51).

Definition: Weak Bisimulation A binary relation $\mathcal{S}$ over expressions is a *weak bisimulation* if for every (f,g) in $\mathcal{S}$ (also written as $f \mathcal{S} g$)

for every $f \stackrel{a}{\rightarrow} f'$, there is some $g \stackrel{a'}{\rightarrow}* g'$ such that $f' \mathcal{S} g'$ and $a \stackrel{\setminus\tau}{=} a'$, and
for every $g \stackrel{b}{\rightarrow} g''$, there is some $f \stackrel{b'}{\rightarrow}* f''$ such that $f'' \mathcal{S} g''$ and $b \stackrel{\setminus\tau}{=} b'$. □

Pairs (f,g) in a weak bisimulation are said to be *weakly bisimilar*. Then a transition of f may be matched by a *sequence* of transitions of g, and conversely. Analogous to Theorem 3.1 (page 55), we have

Theorem 3.3. Let $f \mathcal{S} g$ where $\mathcal{S}$ is a weak bisimulation. Then, for every $f \stackrel{u}{\rightarrow}* f'$, there is some $g \stackrel{u'}{\rightarrow}* g'$ such that $f' \mathcal{S} g'$ and $u \stackrel{\setminus\tau}{=} u'$. Conversely, for every $g \stackrel{v}{\rightarrow}* g''$ there is some $f \stackrel{v'}{\rightarrow}* f''$ such that $f'' \mathcal{S} g''$ and $v \stackrel{\setminus\tau}{=} v'$.

Proof: We show that for every trace s of f that has length n, where $n \geq 0$, g has a trace t where $s \stackrel{\setminus\tau}{=} t$. Proof is by induction on n, the lengths of traces, and is similar to the proof of Theorem 3.1 (page 55). □

Analogous to Proposition 3.1 (page 55), we have the following proposition.

Proposition 3.6.

1. The identity relation is a weak bisimulation.
2. The inverse of a weak bisimulation is a weak bisimulation.
3. The relational product of weak bisimulations is a weak bisimulation.
4. The union of weak bisimulations is a weak bisimulation.

5. The reflexive, symmetric and transitive closure of a weak bisimulation is a weak bisimulation. □

3.8.2 The Largest Weak Bisimulation

Analogous to the largest strong bisimulation, $\sim$, we define $\approx$ as the union of all weak bisimulations. Analogous to Proposition 3.2 (page 55), we have:

Proposition 3.7.

1. $\approx$ is the largest weak bisimulation, i.e., any weak bisimulation is a subset of $\approx$.
2. $\approx$ is an equivalence relation.
3. (converse of the bisimilarity property for $\approx$) For any f and g, suppose

   for every $f \xrightarrow{a} f'$, there is some $g \xrightarrow{a'}* g'$ such that
   $f' \mathcal{S} g'$ and $a \overset{\setminus\tau}{=} a'$, and
   for every $g \xrightarrow{b} g''$, there is some $f \xrightarrow{b'}* f''$ such that
   $f'' \mathcal{S} g''$ and $b \overset{\setminus\tau}{=} b'$.

   Then, $f \approx g$. □

From the definition of weak bisimilarity, it follows that

Proposition 3.8. Expressions that are strongly bisimilar are also weakly bisimilar, i.e., $\sim \subseteq \approx$. □

Proposition 3.9. If the only transition of f is $f \xrightarrow{\tau} f'$, then $f \approx f'$.

Proof: We use property (3) from Proposition 3.7: the τ transition of f is matched by the empty trace of f' and any transition a of f' is matched by the trace τa of f. □

3.8.3 Weak Equivalence

Analogous to the strong equivalence relation, define weak equivalence relation, written as $\overset{\approx}{=}$, as the union of all closed weak bisimulations. The following proposition is analogous to Proposition 3.4 (page 57).

Proposition 3.10.

1. $\overset{\approx}{=}$ is the largest weak bisimulation that is closed under binding. That is, $\overset{\approx}{=}$ is a closed weak bisimulation and any closed weak bisimulation is a subset of $\overset{\approx}{=}$.
2. $\overset{\approx}{=}$ is a subset of $\approx$.

3. $\overset{\approx}{=}$ is an equivalence relation.

4. Given that $f \overset{\approx}{=} g$, for every binding B, $B.f \overset{\approx}{=} B.g$.

5. For expressions f, g and binding B, suppose that

   for every $B.f \xrightarrow{a} f'$, there is some $B.g \xrightarrow{a'}* g'$ such that $f' \overset{\approx}{=} g'$, and $a \overset{\setminus\tau}{=} a'$, and

   for every $B.g \xrightarrow{b} g''$, there is some $B.f \xrightarrow{b'}* f''$ such that $f'' \overset{\approx}{=} g''$ and $b \overset{\setminus\tau}{=} b'$.

   Then, $f \overset{\approx}{=} g$. □

Analogous to the bifurcation theorem, Theorem 3.2 (page 64), we have:

Theorem 3.4. (Weak-Bifurcation Theorem) Let $\mathcal{S}$ be a binary relation such that given $f \mathcal{S} g$

for every $f \xrightarrow{a} f'$ there is some $g \xrightarrow{a'}* g'$ such that
$f' \mathcal{S} g'$, $f' \overset{\approx}{=} g'$, or $f' (\mathcal{S} \mid \overset{\approx}{=}) g'$, and $a \overset{\setminus\tau}{=} a'$, and (C1)

for every $g \xrightarrow{a} g''$ there is some $f \xrightarrow{a'}* f''$ such that
$f'' \mathcal{S} g''$, $f'' \overset{\approx}{=} g''$, or $f'' (\mathcal{S} \mid \overset{\approx}{=}) g''$ and $a \overset{\setminus\tau}{=} a'$. (C2)

Then, $f \approx g$. Further, if $\mathcal{S}$ is closed then $f \overset{\approx}{=} g$. □

3.8.4 Congruences under Weak Equivalence

All Orc combinators preserve weak equivalence, i.e., given that $f \overset{\approx}{=} g$, the identities in Table 3.5 hold for all h. We can obtain similar identities by replacing $>x>$ by $\gg$ and $<x<$ by $\ll$. Also, as in strong equivalence, prefixing by a definition preserves weak equivalence, and replacing the body of a definition by a weakly equivalent expression results in an weakly equivalent definition. In Table 3.5, $f \overset{\approx}{=} g$, h is arbitrary and D is any definition. We prove one of these results below.

$f \mid h$	$\overset{\approx}{=}\ g \mid h$	$h \mid f$	$\overset{\approx}{=}\ h \mid g$
$f >x> h$	$\overset{\approx}{=}\ g >x> h$	$h >x> f$	$\overset{\approx}{=}\ h >x> g$
$f <x< h$	$\overset{\approx}{=}\ g <x< h$	$h <x< f$	$\overset{\approx}{=}\ h <x< g$
$f \,;\, h$	$\overset{\approx}{=}\ g \,;\, h$	$h \,;\, f$	$\overset{\approx}{=}\ h \,;\, g$
$D \# f$	$\overset{\approx}{=}\ D \# g$	$(def\ E(\bar{x}) = f) \# h$	$\overset{\approx}{=}\ (def\ E(\bar{x}) = g) \# h$

Table 3.5: Weak Equivalence Preservation

Proof of $h <x< f \overset{\approx}{=} h <x< g$, given $f \overset{\approx}{=} g$:
Postulate relation $\mathcal{S}$ by,

$h <\!x\!< f \;\; \mathcal{S} \;\; h <\!x\!< g$, for all f, g and h, where $f \stackrel{\approx}{=} g$.

Because of the symmetry in both sides, we will only consider the transitions of $h <\!x\!< f$. Further, non-publication transitions of f and g can be ignored, as discussed in Lemma 3.1 (page 59). Similarly, all transitions of h can also be ignored because they do not change the structure of the resulting expressions. Also, binding closure proof can be omitted for a congruence. The only remaining part is to consider the publication transition of f, $f \stackrel{!c}{\rightarrow} f'$, and show that the conditions of the weak bifurcation theorem, Theorem 3.4, are met.

• Publication transitions of f: Let $f \stackrel{!c}{\rightarrow} f'$. Given $f \approx g$, there is some $g \stackrel{\tau^*!c\tau^*}{\rightarrow} * g'$ such that $f' \approx g'$. Then, $g \stackrel{\tau^*}{\rightarrow} * g'''$ and $g''' \stackrel{!c}{\rightarrow} g''$ for some g'' and g'''.

$$
\begin{array}{ll}
h <\!x\!< g \stackrel{\tau^*}{\rightarrow} * \; h <\!x\!< g''' & \text{, given } g \stackrel{\tau^*}{\rightarrow} * \; g''' \text{ repeatedly apply (PRUN)} \\
h <\!x\!< g''' \stackrel{!c}{\rightarrow} (x := c).h & \text{, given } g''' \stackrel{!c}{\rightarrow} g'' \text{ apply (PRUP)} \\
h <\!x\!< g \stackrel{\tau^*!c}{\rightarrow} * \; (x := c).h & \text{, combining the above two} \\
h <\!x\!< f \stackrel{!c}{\rightarrow} (x := c).h & \text{, given } f \stackrel{!c}{\rightarrow} f' \text{ apply (PRUP)}
\end{array}
$$

The resulting expressions in the last two lines of the proof are both $(x := c).h$; so, they are weakly bisimilar. Apply the weak bifurcation theorem to conclude the proof. □

3.8.5 Identities under Weak Equivalence

All the identities for strong equivalence hold for weak equivalence as well. Additionally, we have

(left unit of $\gg$)	$signal \gg f$	$\stackrel{\approx}{=}$	f		
(right unit of $>\!x\!>$)	$f >\!x\!> x$	$\stackrel{\approx}{=}$	f		
(left zero of ;)	$stop \; ; f$	$\stackrel{\approx}{=}$	f		
(right zero of ;)	$f \; ; stop$	$\stackrel{\approx}{=}$	f		
(*Ift* properties)	$Ift(true)$	$\stackrel{\approx}{=}$	$signal$	$Ift(false)$	$\stackrel{\approx}{=}$ $stop$
(*Iff* properties)	$Iff(true)$	$\stackrel{\approx}{=}$	$stop$	$Iff(false)$	$\stackrel{\approx}{=}$ $signal$

3.9 Synchronous Semantics

The asynchronous semantics is appropriate for an execution in which time plays no particular role. The sites are called sometime after the call becomes due, values are published anytime after they are available, and decisions about the order of execution of independent events are left to the scheduler. Elimination

of explicit real time is based on Dijkstra's famous dictum [13] that independent processes execute at unknown but finite speeds[1] . For example, from the asynchronous semantics, $x <x< (1 \mid 1000)$ publishes either "1" or "1000" depending on the behavior of the scheduler; for the Orc programmer the choice is non-deterministic.

This picture changes when we introduce the common site *Rwait* in an expression. Expression $x <x< (Rwait(1) \gg 1 \mid Rwait(1000) \gg 1000)$ definitely publishes "1"; it is no longer a choice of the scheduler. A time order has been imposed on the events.

The timed semantics of Orc is a simple extension of the asynchronous semantics, by associating a non-negative real number as a time component with each event. Transition $f \stackrel{t,a}{\rightarrow} f'$ denotes that f *may* move to f' with a as its *first* event exactly t time units after the execution of f starts. Thus, the time associated with a transition is relative to the start of evaluation of the expression.

The rules for common sites are in the form, for example,

$$3 \stackrel{0,\tau}{\rightarrow} ?3, \quad ?3 \stackrel{0,!3}{\rightarrow} stop$$

(Here, we have applied the convention of using τ as the call event for common sites.) The rule for *Rwait* is similar, except that the response is received at a later time.

$$Rwait(t) \stackrel{0,\tau}{\rightarrow} ?Rwait(t), \quad ?Rwait(t) \stackrel{t,!}{\rightarrow} stop$$

3.9.1 Time-shifted Expressions

Suppose we have $f \stackrel{t,a}{\rightarrow} f'$. We may expect to conclude that $f \mid g \stackrel{t,a}{\rightarrow} f' \mid g$, using (PARL). However, this is incorrect. For example, from $?Rwait(1) \stackrel{1,!}{\rightarrow} stop$ we can not conclude $?Rwait(1) \mid ?Rwait(3) \stackrel{1,!}{\rightarrow} stop \mid ?Rwait(3)$. The passage of 1 unit of time has altered the expression $?Rwait(3)$ so that it behaves as $?Rwait(2)$; it will publish in 2 time units, not 3. That is, we should deduce $?Rwait(1) \mid ?Rwait(3) \stackrel{1,!}{\rightarrow} stop \mid ?Rwait(2)$.

A *time-shifted expression* f^t, where $t \geq 0$, is the expression resulting from f after its execution of t time units *without occurrence of an event*. Thus, $?Rwait(3)^1 = ?Rwait(2)$. If it is not possible for t time units to elapse without f engaging in an event, f^t is defined to be $\bot$. For instance, $?Rwait(1)^2 = \bot$ because $?Rwait(1)$ definitely publishes before 2 time units have elapsed. Expression $\bot$ has no transition. Any expression containing $\bot$ as a subexpression is $\bot$; this is where $\bot$ is different from *stop*. Transition $f \stackrel{t,a}{\rightarrow} \bot$ denotes that f does not engage in the given transition.

[1] "(otherwise) it will make the proper working (of a program) a rather unstable equilibrium, sensitive to any change in the different speeds, as may easily arise by replacement of a component by another say, replacement of a line printer by a faster model ..."

$stop^t$	=	$stop$
$?Rwait(s)^t$	=	$?Rwait(s-t)$ if $s \geq t$, $\perp$ otherwise
$\dfrac{?M(\bar{v}) \stackrel{s,a}{\rightarrow} M'}{?M(\bar{v})^t \stackrel{s-t,a}{\rightarrow} M',\ \text{if } s \geq t}$		
$?M(\bar{v})^t = \perp,\ \text{if } s < t$		
$e(\bar{x})^t$	=	$e(\bar{x})$ if e is not bound
$(f \mid g)^t$	=	$f^t \mid g^t$
$(f >x> g)^t$	=	$f^t >x> g$
$(f <x< g)^t$	=	$f^t <x< g^t$
$(f \,;\, g)^t$	=	$f^t \,;\, g$
$(D \# f)^t$	=	$D \# f^t$

Table 3.6: Definition of Time-shifted Expressions

We define f^t formally in Table 3.6 (page 77). To understand the definitions first look at the ones corresponding to the combinators, the last four in that table, as they are easier to justify. The definition for the parallel combinator is justified because the two subexpressions act independently, and any transition in one, say f at time t, affects g by only "aging" it to g^t. For $(f >x> g)^t$, only f is aged since its execution starts as soon as the execution of $(f >x> g)$ starts, but since g's execution does not start until it is spawned by a publication of f, there is no change in that subexpression. For $(f <x< g)^t$, executions of both f and g start as soon as $f <x< g$ starts; so, both start aging right-away. The argument for $(f \,;\, g)^t$ is similar to that for $(f >x> g)^t$.

Next, we need to define $e(\bar{x})^t$ where e denotes a site; the name e may or may not be bound, and none, some or all of its parameters may be bound. A site call can start only if its name is bound; so $e(\bar{x})^t = e(\bar{x})$ where e is not bound. We consider the remaining cases next. Observe that for any expression f, f^0 should be just f. So, we consider the case for $t > 0$.

If the name of a site, say M, is bound, then its execution must have started already and it must have been transformed to $?M(\bar{p})$. So, we need not define $M(\bar{x})^t$ for $t > 0$; we need only define $?M(\bar{x})^t$. If $?M(\bar{x}) \stackrel{s,a}{\rightarrow} M'$ then $?M(\bar{v})^t \stackrel{s-t,a}{\rightarrow} M'$, provided $s \geq t$; if $s < t$ then $?M(\bar{v})^t = \perp$.

Applying this rule to $?Rwait(s)$, since $?Rwait(s) \stackrel{s,!}{\rightarrow} stop$, we have

$$?Rwait(s)^t \stackrel{s-t,!}{\rightarrow} stop \text{ for } s \geq t, \text{ and } ?Rwait(s)^t = \perp \text{ for } s < t.$$

Therefore, for $s \geq t$, $?Rwait(s)^t = ?Rwait(s-t)$. For every other common site M, and $t > 0$, $?M^t = \perp$. Thus, $?Signal^1 = \perp$, and so is $?Ift(true)^2$.

Next, consider $E(\bar{p})$ where E is an internal site name. Similar arguments show that its execution must have started and transformed it to $(\bar{p}/\bar{x}).f$ where $[E(\bar{x}) = f]$ is the site definition. Therefore, this case needs no further consideration. This completes the explanation of the definitions in Table 3.6.

We leave it to the reader to show that if $f^t \overset{s,a}{\rightarrow} f'$ is a transition then so is $f \overset{s+t,a}{\rightarrow} f'$. And, $(f^s)^t = f^{s+t}$.

3.9.2 Synchronous Semantic Transition Rules

The synchronous semantic rules, given in Figure 3.3 (page 83), are identical to their asynchronous counterparts of Figure 3.1 (page 81). The only enhancement is the inclusion of a time component in each transition and the replacement of expressions by their time-shifted counterparts.

3.9.3 Trace, Bisimulation, Equivalence

3.9.3.1 Trace in Timed Semantics

As in Section 3.4.8 (page 51), we define the *trace* relation, $\rightarrow*$, as the transitive closure of the transition relation $\rightarrow$, together with the empty transition ϵ. That is,

$$f \overset{\epsilon}{\rightarrow}* f \qquad \frac{f \overset{t,a}{\rightarrow} f'',\ f'' \overset{s}{\rightarrow}* f'}{f \overset{(t,a)s}{\longrightarrow}* f'}$$

Consider trace $(t_0, a_0) \cdots (t_i, a_i) \cdots (t_n, a_n)$ of expression f. Here a_0 starts t_0 time units after the start of execution of f, and a_i starts t_i units after a_{i-1}, $i > 0$. Define *absolute time* T_i of a_i to be the time at which a_i occurs after the execution of f starts. So,

$$T_i = \begin{cases} t_0 & \text{if} \quad i = 0 \\ T_{i-1} + t_i & \text{if} \quad i > 0 \end{cases}$$

The *end time* of a trace is the absolute time of its last event, or 0 if the trace is empty. For traces u and v, $u \overset{\backslash\!\backslash \tau}{=} v$ holds if and only if:

1. u and v have the same end times, and
2. u and v have the same sequence of non-τ events with the same absolute times.

It is easy to see that $\overset{\backslash\!\backslash \tau}{=}$ is an equivalence relation. And, for any sequence s of $(0, \tau)$ events only, $s \overset{\backslash\!\backslash \tau}{=} \epsilon$. Further,

Observation 3.2. Given $u \overset{\backslash\!\backslash \tau}{=} v$ and $u' \overset{\backslash\!\backslash \tau}{=} v'$, $uu' \overset{\backslash\!\backslash \tau}{=} vv'$. □

This result relies on the fact that u and v have the same end times so that u' and v' start at the same time.

3.9.3.2 Strong and Weak Bisimulation

Definition of strong bisimulation is exactly the same as before for the asynchronous case. Weak bisimulation is also the same except that $\overset{\backslash\!\backslash \tau}{=}$ replaces $\overset{\backslash \tau}{=}$ in the definition, as follows.

Definition: Weak Synchronous Bisimulation A binary relation $\mathcal{S}$ over expressions is a *weak synchronous bisimulation* if for every (f,g) in $\mathcal{S}$ (also written as $f\ \mathcal{S}\ g$)

for every $f \xrightarrow{a} f'$, there is some $g \xrightarrow{a'}* g'$ such that $f'\ \mathcal{S}\ g'$ and $a \overset{\backslash\backslash\tau}{=} a'$, and for every $g \xrightarrow{b} g''$, there is some $f \xrightarrow{b'}* f''$ such that $f''\ \mathcal{S}\ g''$ and $b \overset{\backslash\backslash\tau}{=} b'$. □

We have exactly the same theorem, Theorem 3.1 (page 55), for both synchronous and asynchronous strong bisimulations. We repeat the theorem here.

Theorem 3.5. Let $f\ \mathcal{S}\ g$ where $\mathcal{S}$ is a strong bisimulation. Then, for every $f \xrightarrow{u}* f'$, there is some $g \xrightarrow{u}* g'$ such that $f'\ \mathcal{S}\ g'$. Conversely, for every $g \xrightarrow{v}* g''$ there is some $f \xrightarrow{v}* f''$ such that $f''\ \mathcal{S}\ g''$. □

Corollary: Strongly bisimilar expressions have identical trace sets in timed semantics. □

The theorem corresponding to weak bisimulation, Theorem 3.3 (page 72), is the same except that $\overset{\backslash\backslash\tau}{=}$ replaces $\overset{\backslash\tau}{=}$. Its proof does require an additional fact, Observation 3.2 given above.

Theorem 3.6. Let $f\ \mathcal{S}\ g$ where $\mathcal{S}$ is a weak bisimulation. Then, for every $f \xrightarrow{u}* f'$, there is some $g \xrightarrow{v}* g'$ such that $f'\ \mathcal{S}\ g'$ and $u \overset{\backslash\backslash\tau}{=} v$. Conversely, for every $g \xrightarrow{v}* g''$ there is some $f \xrightarrow{u}* f''$ such that $f''\ \mathcal{S}\ g''$ and $u \overset{\backslash\backslash\tau}{=} v$.

Proof: Proof is as before, except that Observation 3.2 is essential in the inductive case. Given $f \xrightarrow{a} f'' \xrightarrow{u}* f'$ and that $\mathcal{S}$ is a weak bisimulation, there is some $g \xrightarrow{a'}* g''$ such that $f''\ \mathcal{S}\ g''$ and $a \overset{\backslash\backslash\tau}{=} a'$. Applying induction on $f''\ \mathcal{S}\ g''$ and $f'' \xrightarrow{u}* f'$, there is some $g'' \xrightarrow{u'}* g'$ such that $f'\ \mathcal{S}\ g'$ and $u \overset{\backslash\backslash\tau}{=} u'$. Thus, we have shown $g \xrightarrow{a'u'}* g'$ where $f'\ \mathcal{S}\ g'$ and $au \overset{\backslash\backslash\tau}{=} a'u'$; the latter follows from the observation above. Proof in the other direction is similar. □

Corollary: Given weakly bisimilar expressions f and g, for any trace s of f there is a trace t of g where $s \overset{\backslash\backslash\tau}{=} t$, and conversely. □

Equivalence The propositions for asynchronous strong bisimulation, Propositions 3.1 (page 55) and 3.2 (page 55), and for asynchronous weak bisimulation, Propositions 3.6 (page 72) and 3.7 (page 73), apply in the synchronous case as well. We define equivalence, both strong and weak, in exactly the same way. Consequently, all the identities developed for the asynchronous case apply to the synchronous case as well. We can prove additional identities involving time, such as, $Rwait(0) \sim signal$ and $Rwait(s) \gg Rwait(t) \approx Rwait(s+t)$.

3.10 Concluding Remarks

A more elegant treatment of semantics uses structural equivalence [38]. We start with the following equivalences.

1. $(x := \not!).e(x) \ \equiv \ stop$,
2. $f \mid g \ \equiv \ g \mid f$
3. $(f \mid g) \mid h \ \equiv \ f \mid (g \mid h)$
4. $f \mid stop \ \equiv \ f$
5. $stop \ >x> \ f \ \equiv \ stop$
6. $f \ <x< \ stop \ \equiv \ (x := \not!).f$

Then, one of the rules, (PARL) or (PARR), can be dropped because parallel combinator is symmetric, from (2).

Our intent was to introduce the minimum required number of concepts. Therefore, we have chosen not to introduce structural equivalence.

A real-time rewriting semantics of Orc appears in Meseguer and Al Turki [2].

Acknowledgement Manfred Broy pointed out an oversight in one of the semantic rules in an earlier version.

$$M(\bar{v}) \overset{\surd M(\bar{v})}{\rightarrow} ?M(\bar{v}) \qquad \text{(SiteCall)}$$

$$\frac{?M(\bar{v}) \text{ receives a non-terminal response } !c}{?M(\bar{v}) \overset{!c}{\rightarrow} ?M(\bar{v})} \qquad \text{(NT-Res)}$$

$$\frac{?M(\bar{v}) \text{ receives a terminal response } !c}{?M(\bar{v}) \overset{!c}{\rightarrow} \mathit{stop}} \qquad \text{(T-Res)}$$

$$\frac{?M(\bar{v}) \text{ receives a negative response}}{?M(\bar{v}) \overset{\tau}{\rightarrow} \mathit{stop}} \qquad \text{(NegRes)}$$

$$\frac{f \overset{t,a}{\rightarrow} f'}{D \# f \overset{t,a}{\rightarrow} D \# f'} \qquad \text{(DefPass)}$$

$$\frac{D \text{ is } \mathit{def}\, E(\bar{x}) = g \quad \mathit{free}(g) \subseteq \{E\} \cup \{\bar{x}\}}{D \# f \overset{\tau}{\rightarrow} (\langle E, \bar{x}, g\rangle / E).f} \qquad \text{(DefScope)}$$

$$\langle E, \bar{x}, g\rangle(\bar{p}) \overset{\tau}{\rightarrow} (\langle E, \bar{x}, g\rangle / E).(\bar{p}/\bar{x}).g \qquad \text{(DefCall)}$$

$$\frac{f \overset{a}{\rightarrow} f'}{f \mid g \overset{a}{\rightarrow} f' \mid g} \qquad \text{(ParL)}$$

$$\frac{g \overset{a}{\rightarrow} g'}{f \mid g \overset{a}{\rightarrow} f \mid g'} \qquad \text{(ParR)}$$

$$\frac{f \overset{a}{\rightarrow} f' \qquad \text{non-pub } a}{\begin{array}{c} f >x> g \overset{a}{\rightarrow} f' >x> g \\ f \gg g \overset{a}{\rightarrow} f' \gg g \end{array}} \qquad \text{(SeqN)}$$

$$\frac{f \overset{!c}{\rightarrow} f'}{f >x> g \overset{\tau}{\rightarrow} f' >x> g \mid (x := c).g \quad f \gg g \overset{\tau}{\rightarrow} (f' \gg g) \mid g} \qquad \text{(SeqP)}$$

$$\frac{f \overset{a}{\rightarrow} f'}{\begin{array}{c} f <x< g \overset{a}{\rightarrow} f' <x< g \\ f \ll g \overset{a}{\rightarrow} f' \ll g \end{array}} \qquad \text{(PruL)}$$

$$\frac{g \overset{a}{\rightarrow} g' \qquad \text{non-pub } a}{\begin{array}{c} f <x< g \overset{a}{\rightarrow} f <x< g' \\ f \ll g \overset{a}{\rightarrow} f \ll g' \end{array}} \qquad \text{(PruN)}$$

$$\frac{g \overset{!c}{\rightarrow} g'}{\begin{array}{c} f <x< g \overset{\tau}{\rightarrow} (x := c).(f) \\ f \ll g \overset{\tau}{\rightarrow} f \end{array}} \qquad \text{(PruP)}$$

$$\frac{g \cong \mathit{stop}}{\begin{array}{c} f <x< g \overset{\tau}{\rightarrow} (x := \not{\ell}).f \\ f \ll g \overset{\tau}{\rightarrow} f \end{array}} \qquad \text{(PruH)}$$

$$\frac{f \overset{a}{\rightarrow} f'}{f \,;\, g \overset{a}{\rightarrow} f' \,;\, g} \qquad \text{(OthN)}$$

$$\frac{f \overset{!c}{\rightarrow} f'}{f \,;\, g \overset{!c}{\rightarrow} f'} \qquad \text{(OthP)}$$

$$\frac{f \cong \mathit{stop}}{f \,;\, g \overset{\tau}{\rightarrow} g} \qquad \text{(OthH)}$$

Figure 3.1: Asynchronous Semantics of Orc

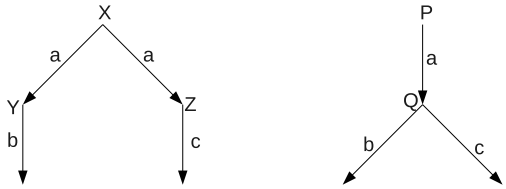

Figure 3.2: P and X have identical trace sets, but they are not bisimilar

$$M(\bar{v}) \overset{0,\surd M(\bar{v})}{\rightarrow} ?M(\bar{v}) \quad \text{(SiteCall)}$$

$$\frac{?M(\bar{v}) \text{ receives a non-terminal response } !c,\ t \text{ units after the call}}{?M(\bar{v}) \overset{t,!c}{\rightarrow} ?M(\bar{v})} \quad \text{(NT-Res)}$$

$$\frac{?M(\bar{v}) \text{ receives a terminal response } !c,\ t \text{ units after the call}}{?M(\bar{v}) \overset{t,!c}{\rightarrow} \mathit{stop}} \quad \text{(T-Res)}$$

$$\frac{?M(\bar{v}) \text{ receives a negative response } t \text{ units after the call}}{?M(\bar{v}) \overset{t,\tau}{\rightarrow} \mathit{stop}} \quad \text{(NegRes)}$$

$$\frac{f \overset{t,a}{\rightarrow} f'}{D \,\#\, f \overset{t,a}{\rightarrow} D \,\#\, f'} \quad \text{(DefPass)}$$

$$\frac{f \overset{t,a}{\rightarrow} f'}{f \mid g \overset{t,a}{\rightarrow} f' \mid g^t} \quad \text{(ParL)}$$

$$\frac{g \overset{t,a}{\rightarrow} g'}{f \mid g \overset{t,a}{\rightarrow} f^t \mid g'} \quad \text{(ParR)}$$

$$\frac{f \overset{t,a}{\rightarrow} f' \qquad \text{non-pub } a}{\begin{array}{c} f >x> g \overset{t,a}{\rightarrow} f' >x> g \\ f \gg g \overset{t,a}{\rightarrow} f' \gg g \end{array}} \quad \text{(SeqN)}$$

$$\frac{f \overset{t,!c}{\rightarrow} f'}{\begin{array}{c} f >x> g \overset{t,\tau}{\rightarrow} f' >x> g \mid (x := c).g \\ f \gg g \overset{t,\tau}{\rightarrow} (f' \gg g) \mid g \end{array}} \quad \text{(SeqP)}$$

$$\frac{D \text{ is } \mathit{def}\, E(\bar{x}) = g \qquad \mathit{free}(g) \subseteq \{E\} \cup \{\bar{x}\}}{D \,\#\, f \overset{0,\tau}{\rightarrow} (\langle E, \bar{x}, g\rangle / E).f} \quad \text{(DefScope)}$$

$$\langle E, \bar{x}, g\rangle(\bar{p}) \overset{0,\tau}{\rightarrow} (\langle E, \bar{x}, g\rangle / E).(\bar{p}/\bar{x}).g \quad \text{(DefCall)}$$

$$\frac{f \overset{t,a}{\rightarrow} f' \qquad \text{non-pub } a}{\begin{array}{c} f <x< g \overset{t,a}{\rightarrow} f' <x< g^t \\ f \ll g \overset{t,a}{\rightarrow} f' \ll g^t \end{array}} \quad \text{(PruL)}$$

$$\frac{g \overset{t,a}{\rightarrow} g' \qquad \text{non-pub } a}{\begin{array}{c} f <x< g \overset{t,a}{\rightarrow} f^t <x< g' \\ f \ll g \overset{t,a}{\rightarrow} f^t \ll g' \end{array}} \quad \text{(PruN)}$$

$$\frac{g \overset{t,!c}{\rightarrow} g'}{\begin{array}{c} f <x< g \overset{t,\tau}{\rightarrow} (x := c).(f^t) \\ f \ll g \overset{t,\tau}{\rightarrow} f^t \end{array}} \quad \text{(PruP)}$$

$$\frac{g \cong \mathit{stop}}{\begin{array}{c} f <x< g \overset{0,\tau}{\rightarrow} (x := \not{l}).f \\ f \ll g \overset{0,\tau}{\rightarrow} f \end{array}} \quad \text{(PruH)}$$

$$\frac{f \overset{t,a}{\rightarrow} f'}{f \,;\, g \overset{t,a}{\rightarrow} f' \,;\, g} \quad \text{(OthN)}$$

$$\frac{f \overset{t,!c}{\rightarrow} f'}{f \,;\, g \overset{t,!c}{\rightarrow} f'} \quad \text{(OthP)}$$

$$\frac{f \cong \mathit{stop}}{f \,;\, g \overset{0,\tau}{\rightarrow} g} \quad \text{(OthH)}$$

Figure 3.3: Synchronous Semantics of Orc

Chapter 4

Orc Programming Language

4.1 Introduction

Orc calculus, described in Chapter 2, provides only minimal facilities for actual programming. The purpose of the calculus is to exhibit the smallest number of concepts adequate for our purpose; it constitutes a "model of limitation" in that the inadequacies of the theory, the negative results, can be explored within the calculus. A programming language constitutes a "model of computation" in that the solutions of actual problems, the positive results, can be demonstrated easily. In this sense, Orc calculus is analogous to the λ-calculus and the Orc language to a functional programming language. The gap between the calculus and the language is, however, narrower in Orc. The language features of Orc are mere syntactic sugar that are easily translated to a program in Orc calculus, possibly employing sites from the Orc standard library. The Orc translator converts an Orc language program to the calculus notation and an interpreter executes the calculus directly. We show the translations of all the language features in this chapter.

The Orc combinators can be superposed on any host language that supports site calls and that can accommodate concurrent computations; Launchbury and Trevor [31] have ported the Orc combinators to Haskell [15] by judicious use of monads. We could have used a traditional language like Java as the host. We decided to design Orc language so that it departs very little from the calculus, include concurrency as the default mode for control flow, and allow complete freedom in designing and using sites.

Like a functional programming language Orc avoids mutable variables. Sites coded in other programming languages, which includes many library sites[1] , may maintain and update mutable stores. Unlike a functional programming language, however, Orc's expressions may publish multiple values, exploit real time and engage in non-deterministic computations.

The description of Orc language in this chapter mentions only the features

[1] All sites defined in the standard library are coded in Java, Scala or Orc as of this writing.

that are essential to understanding the programs in this book. See the website at [16] for a complete description of the language and the standard library.

A Small Example We give a small example that illustrates some, but not all, of the features of the Orc language. We show its translation below.

The following program runs a series of trials. Each trial consists of rolling a pair of dice. A trial succeeds if the total shown by the two dice is 7. Site call `exp(n)` publishes the number of successful trials in `n` trials.

```
def roll() = Random(6) + 1

def exp(0) = 0
def exp(n) = exp(n-1) + (if roll() + roll() = 7 then 1 else 0)
```

The program includes two sites, `roll` and `exp`. Site `roll()` publishes a random value between 1 and 6; it makes use of the standard library site, `Random`, where `Random(n)`, for positive integer `n`, publishes a random integer i, $0 \leq i < n$. Site `exp` is "clausally defined"; the first definition applies if the site argument is zero and the second definition applies if it is non-zero. In the first case the number of successful trials is 0 since the number of trials is 0. In the second case, the number of successful trials in `n` trials is the sum of the number in `n-1` trials and in a single trial; the latter is 1 provided the two dice rolls add up to 7 and 0 otherwise.

We show a translation of the definition of `exp` into Orc calculus[2].

```
def exp(n) =
   ( Ift(b) >> 0
   | Iff(b) >>
     ( add(x,y)
         <x< ( exp(m,c) <m< sub(n,1) ) )
         <y< ( ( Ift(bb) >> 1 | Iff(bb) >> 0 )
                   <bb< equals(p,7)
                     <p< add(q,r)
                        <q< roll()
                        <r< roll() )

   ) <b< equals(n,0)
```

The symbol "+" in the original program is replaced by a call to the standard site `add`. The arguments of `add` are variables, not expressions, as is demanded in the Orc calculus. So, in the translated version, the arguments are replaced by variables that are bound to the values of these expressions. Site `sub(x,y)` returns `x-y`, and `equals` returns `true` if its two arguments are equal and `false` otherwise.

[2]The translations given in this chapter are not necessarily the ones in the Orc implementation; the latter includes many optimizations. The site names in the standard library are possibly different from the ones shown here.

The original program does not use any of the concurrency combinators of Orc explicitly. In fact, the program looks entirely functional and deterministic, with the sole exception of the site call `Random(6)`. Yet, each nested expression translates into a use of the pruning combinator, making this program implicitly concurrent without any programmer intervention. We will show that all $2n$ calls to `roll` may be executed concurrently.

Road map for this chapter Orc language includes two novel features besides the combinators: (1) *deflation*, described in Section 4.3.3 (page 90), that allows an Orc expression to appear in full generality wherever a variable can be used, such as for actual parameters in site calls, and (2) a very general notion of *object classes*, described in Section 4.6 (page 113), that allows not only data encapsulation, but concurrent instantiations and invocations of object instances, possibly depending on the passage of real time; in particular, an instance may engage in autonomous computation without an explicit method call. The remaining features of Orc language are either syntactic sweeteners or well-known features from functional programming. These features include various standard data types and data structures (Section 4.3, page 88), use of patterns in accessing data and binding them to the component variables (Section 4.3.5, page 94), and clausal definitions of sites (Section 4.4.2.3, page 101). Those familiar with functional programming may still need to understand how concurrency combinators and site calls are integrated with traditional functional programming features. We show the translation of each feature into Orc calculus.

4.2 Preliminaries

We discuss some preliminary material in this section.

Syntax for Comments An inline comment starts with two dashes, `--`, and runs to the end of the line. A multi-line comment starts with `{-` and ends with `-}`. Multi-line comments can be nested, so `{-` starts a new multi-line comment, and `-}` ends the current multi-line comment.

Characterstics of Sites Orc calculus makes few demands on a site. A site may be called in a procedural style and it may respond with zero or more values. While a theory can be built on these minimal assumptions, programmers often have, or need, additional information about sites that they can exploit. We describe some useful site characterstics that may, possibly, appear in a site specification; some of these have been described in the earlier chapters.

A site that always responds with a terminal response when its computation on behalf of a call ends is *helpful*, see Section 2.2.2 (page 21). It can be shown that any site whose definition includes calls only to helpful sites is helpful, see Section 3.7.5 (page 70). Henceforth, any expression that calls only helpful sites is termed helpful.

A *pure* site implements a mathematical function. The site always publishes just a terminal response; therefore, it is helpful. The response is deterministic. It has no side effect discernible to the caller of the site. The standard library includes a number of pure sites including the traditional ones for operations on numeric and boolean data. Site **`Ift`** is pure even though it does not respond with a value if its argument is `false`, but sends only a negative response. Site `Rwait` is also a pure site though its response is delayed. It behaves like any other pure site whose computation consumes a noticable amount of real time.

A *quasi-pure* site always publishes just a terminal response, like a pure site. Further, its side effects, if any, are unobservable to the caller. However, the response may not be deterministic, as in `Random`. Note that a call to `Random` has the side effect of changing the seed that will be used to compute the next random number, though the side effect is internal to the site and can not be directly observed by a caller. All factory sites in the standard library, see Section 4.5 (page 106), are *quasi-pure*. For example, a call to the factory site `Channel` publishes a site that behaves like a FIFO channel; different calls return different instances of channel and the side effects are entirely internal to `Channel`. By contrast, site `Println` always publishes just a signal as a terminal response, but has the side effect of displaying its argument string that is observable; so it is not pure or quasi-pure.

Halted and Silent expression We informally described the notions of halted and silent expressions in Section 2.3 (page 23) and formally in Section 3.7.1 (page 65). An expression *halts* or is *halted* if it can not engage in any activity. Expressions **`stop`** and **`stop`** `>> 3` are halted. An expression is *silent* if it never publishes. A halted expression, therefore, is silent. A silent expression may still call sites and continue executing forever, as in `metronome(1)>>` **`stop`**; see Section 2.5.2.2 (page 34) for a definition of `metronome`. Halting of an expression can be detected only if it calls helpful sites.

4.3 Basic Data and Control Structures

The standard data types of the Orc language are: *number*, *boolean* and *string*, and the unitary data *signal*; we cover the standard data types in Section 4.3.1. The data structuring mechanisms are: *tuple*, *list* and *record*, which we cover in Section 4.3.4 (page 91). Orc also includes *pattern* construction mechanisms as described in Section 4.3.5 (page 94). We describe conditional expression, a syntactic convenience, in Section 4.3.2 and the deflation mechanism in Section 4.3.3 (page 90).

Most of material on basic data types and structures is standard. There is one important aspect in which Orc's treatment of this material is different: *exceptions* arising in function evaluation. In imperative or functional programs every expression is required to return a value. So, precise rules have to be formulated for exception handling, say for division by zero. Orc expressions need not publish a value. Thus, Orc elides the question "what is the value of

3/0?" by letting 3/0 halt without publishing. A computation may still proceed in other concurrent threads, or the entire computation may terminate without any publication. The implementation (since it is coded in Java) issues an error message, but it has no bearing on the computation proper.

4.3.1 Primitive Data Types

Orc calculus has no built in data type or data structure. Orc language includes numeric, boolean and string constants along with the standard operators on them. The operators are written in the traditional infix or prefix style, as appropriate. The constants are translated to calls on constant sites. Expressions such as `x+y` and `~b` are translated to `add(x,y)` and `not(b)`, respectively, which are sites in the standard library[3] . The available arithmetic operators are the standard ones, such as add, multiply and remainder (written as `%`) with their traditional priorities, and minimum and maximum over two arguments, written as `min` and `max`, respectively. The boolean operators are `~` (logical negation), `&&` (logical and) and `||` (logical or). For strings `s` and `t`, `s+t` denotes their concatenation. Other operations on strings are taken from the Java library and described in the Orc language manual [16]. The sites corresponding to these operations are pure.

Notation for Arithmetic Relations The standard arithmetic relations are: `=` (equals), `/=` (not equals), `:>` (greater than), `<:` (less than), `=>` (greater than or equal to) and `<=` (less than or equal to). The symbols `:>` and `<:` have been chosen to avoid confusion with the sequential and the pruning combinators that use `>` and `<`.

Example

Numbers: `5`, `-2.71e-5`, `1+2`, `min(-1,2.71828)`, `3/0`
Booleans: `false`, `true && (false || true)`, `0.4 = 2.0/5`
String: `"Orc"`, `"ceci nest pas une |"`, `"Try"+ "Orc"`

4.3.2 Conditional Expression

A conditional expression of the form **`if`** `e` **`then`** `f` **`else`** `g`, where `e`, `f` and `g` are expressions, is similar to such constructs in traditional programming languages. First, `e` is executed and any one value published by it is chosen arbitrarily, as described in Section 4.3.3. If the value is `true/false`, then the execution continues with `f/g`. If `e` does not publish or the published value that is chosen is non-boolean, then the expression halts.

Orc combinators have higher precedence than the key words, **`if`**, **`then`** and **`else`**, so **`if`** `e * e'` **`then`** `f * f'` **`else`** `g * g'`, where each `*` denotes any Orc combinator, is **`if`** `(e * e')` **`then`** `(f * f')` **`else`** `(g * g')`.

[3]The names of sites in the standard library may be different from the ones used in the book. The site names used here are for illustrative purposes only.

Example

`if true then "blue"else "green",`	publishes `"blue"`
`if "fish"then "yes"else "no",`	halts
`if false then 4+5 else 4+true,`	halts
`if true then 0/5 else 5/0,`	publishes `0`

Conditional Expression Translation Expression `if e then f else g` is translated to `(if b then f else g)<b< e` which is further translated to `(Ift(b)>> f | Iff(b)>> g)<b< e`. The use of the pruning combinator to choose a single value is the norm, as explained next.

4.3.3 Deflation

Each argument of a site call in Orc calculus is required to be a single variable. Orc language allows any expression to appear as an argument. Since Orc expressions may publish multiple values, a single value for the argument has to be extracted from an expression. *Deflation* is a translation mechanism that extracts a single value from an expression that may publish many values. Expression `C(..., e, ..)`, where expression `e` occupies a position where only a single value is expected, is translated to `C(..., x, ..)<x< e` where `x` is a fresh variable. The context `C` may contain multiple nested expressions to be deflated, so this translation may be applied multiple times. For example, the expression `M(e,f,g)`, where `M` is a site and `e`, `f` and `g` are expressions, is translated to `M(x,y,z)<x< e <y< f <z< g`; parenthesization is unnecessary since the variables are independent. And, deflation is also applied across multiple levels of nesting: `N(M(e,f,g))`, where `N` is a site, is translated to `N(t)<t< (M(x,y,z)<x< e <y< f <z< g)`.

From the semantics of the pruning combinator, the two parts in a deflated expression are evaluated concurrently. *Therefore, all arguments of a site call are always evaluated concurrently.* In fact, concurrency is the norm, and it is implemented without programmer intervention. It can be exploited if multiple processors are available. But there is no obligation that the actual execution be concurrent in an asynchronous program (where real time plays no role); in a single processor implementation, the calls are made in some order chosen by the scheduler. In a synchronous program, where calls to real-time dependant sites appear (See Chapter 10), the arguments of a site call have to be evaluated concurrently.

Comingling Orc combinators in expressions Deflation allows comingling of Orc combinators arbitrarily with other operators within expressions. We consider some examples.

- The expression `(1|2)* (10|100)` is translated to:

```
Times(x,y) <x< (1|2) <y< (10|100)
```

Therefore, the first argument of `Times` is either `1` or `2` and the second is either `10` or `100`; the expression publishes a single value from the set `{10,100,20,200}`.

- (Non-Deterministic Choice) Choose `f` or `g` non-deterministically and execute it: **`if`** `(true|false)`**`then`** `f` **`else`** `g`. As an extension of this example, execute `f` if call to site `M` receives a response within `100` time units and execute `g` otherwise:
  `if` `(M()>> true | Rwait(100)>> false)`**`then`** `f` **`else`** `g`.

- (Translating part of the Dice roll Program) Given below is a translation of `exp(n-1)+ (`**`if`** `roll()+ roll()= 7` **`then`** `1` **`else`** `0)` that appears as part of the dice roll program of Section 4.1. First, replace `+`, `-` and `=` by calls to `add`, `sub` and `equals` to get

```
add(exp(n-1),
    (if equals(add(roll(),roll()), 7) then 1 else 0)
   )
```

  Next, apply deflation starting with the arguments of the inner `add` and then `equals`. Then, translate the conditional expression and finally apply deflation on the outer `add`.

```
add(x,y)
  <x< ( exp(m,c) <m< sub(n,1) )
  <y< ( ( Ift(bb) >> 1 | Iff(bb) >> 0 )
           <bb< equals(p,7)
             <p< add(q,r)
                <q< roll()
                <r< roll() )
```

  Observe that the two calls to `roll()` are made concurrently. More subtle is the fact that the expressions corresponding to variables `x` and `y` are also concurrently executed. Therefore, all calls to `roll()` in the execution, $2 \times n$ such calls, are made concurrently.

4.3.4 Data Structures

Orc language includes the data structures *tuple*, *list* and *record*. The syntax of each is described below. Each data structure is translated as a call on a constructor site with the elements of the data structure as arguments; the site returns the corresponding structured value. The elements may be expressions which are deflated concurrently before the call. Every constructor site corresponding to a data structure is pure, and it is strict in the sense that values of all the arguments are required in order for the site to publish a value. Therefore, if any of the expressions corresponding to an argument is halted, the call itself is halted. If no expression is halted but some expression is silent, the call is silent.

There are additional sites specific to each data structure that are described below. The component of structured values are typically accessed through pattern matching (see Section 4.3.5, page 94).

4.3.4.1 Tuple

A *tuple* is a sequence of at least two values. Orc does not have 0-tuples or 1-tuples. A tuple expression is written as `(`e_0`,...,`e_i`,...)`, where each e_i is an expression. This expression publishes `(`v_0`,...,`v_i`,...)` where all e_is are executed concurrently and each e_i deflates to value v_i. The tuple elements may be of different types. Pure sites `fst` and `snd` publish the first and second components of their argument tuple, respectively, if the argument tuple has exactly two components, otherwise they halt.

Example

`(0,true,''last'')` publishes `(0,true,''last'')`
`((0,0*0),(1,1*1),(2,2*2))` publishes `((0,0),(1,1),(2,4))`
`fst(((0,0*0),(1,1*1),(2,2*2)))` publishes `(0,0)`

Example

- This example implements timeout. If expression `f` publishes value `v` within `k` time units then the following expression publishes the tuple `(true,v)` and halts; otherwise, in case of timeout, it publishes `(false,signal)`. The publication can be analyzed by other parts of the program to determine if there was a timeout, and retrieve `v` in case there was not one.

```
x <x< ( f >y> (true,y) | Rwait(k) >y> (false,y) )
```

- It is required to execute two expressions `f` and `g` concurrently and wait for a result from each before proceeding. We used the following program for *fork-join* in Section 2.5.1.7 (page 32).

```
Tuple(x,y) <x< f <y< g
```

Site call `Tuple(x,y)` can be replaced by just `(x,y)`. We can do even better; simply write `(f,g)` for the whole expression, because, using deflation, this is `(x,y)<x< f <y< g`.

- We introduced phase synchronization in Section 2.5.1.8 (page 32). Given expressions `M()>x> f` and `N()>y> g`, it is required to execute them independently except that `f` and `g` be started only after *both* `M` and `N` have responded. Combining them merely through a parallel combinator does not have the desired effect. The solution in Section 2.5.1.8 can be slightly simplified by using the tuple constructor directly.

```
   (M(),N())
>z>
```

```
(  fst(z) >x> f
 | snd(z) >y> g
)
```

We will shortly see how to eliminate `fst` and `snd` using pattern matching.

4.3.4.2 List

Tuples represent sequences of fixed length and, possibly, varying element types. Lists are intended for sequences with varying length and a fixed element type. An empty list is written as `[]`. A list expression is $[e_0, \ldots, e_i, \ldots]$ where each e_i is an expression. This expression publishes $[v_0, \ldots, v_i, \ldots]$ where each e_i concurrently deflates to value v_i. The list elements could be of different types, though it is strongly recommended that they be of the same type. The list constructor is written as a colon (`:`) infix style; it is a pure site with two arguments, an item and a list.

Site `head` publishes the first element of the argument list, and `tail` the remaining list after removal of the head item. Both sites are pure and `head([])` and `tail([])` are silent. Library site `each` takes a list as argument and publishes all its elements in arbitrary order.

Example

```
[true && true] = [true]
[5, 5 + true, 5] is silent
3:[5, 7] = [3, 5, 7]
3:[ ] = [3]
3:[ [ ] ] = [3,[]]
each([3, 5, 7])  -- publishes 3, 5 and 7 in arbitrary order
```

4.3.4.3 Record

A record is a set of *keys* with associated *bindings*, where a key is a variable name and a binding is a value. Expression $\{. k_0= e_0, \ldots, k_i= e_i, \ldots, k_n= e_n .\}$ has distinct keys k_i and the corresponding bindings e_i. Given that each e_i deflates to v_i, the expression publishes $\{. k_0= v_0, \ldots, k_i= v_i, \ldots, k_n= v_n .\}$. Expression `{. .}` denotes an empty record.

Elements of a record `r` can be accessed through its keys: `r.k` publishes the binding of key `k`; it is silent if it does not have a key `k` or `r` is silent. Given records `r` and `s`, `r+s` is a record that includes all keys and their bindings from both records; if a key appears in both records, the binding from the last record is retained. Therefore, `+` is associative, but not commutative. And, the record $\{. k_0= e_0, \ldots, k_i= e_i, \ldots, k_n= e_n .\}$ is same as
$\{. k_0= e_0 .\}$ + ... + $\{. k_i= e_i .\}$ + ... + $\{. k_n= e_n .\}$.

Example

This example combines list, record and tuple. Given below is a list of records where each record has three keys —`name`, `college`, `accepted`— and the value

associated with `name` is a tuple of two strings, with `college` a string and with `accepted` a boolean.

```
[
  {. name = ("Harry Q.","Error"), college = "CMU",
     accepted = false .},
  {. name = ("Sally", "Hacker"), college = "M.I.T.",
     accepted = true .},
  {. name = ("D. F.", "Virus"), college = "Podonk U.",
     accepted = true .}
]
```

4.3.5 Pattern

Tuple, list and record expressions publish structured values. We are often interested in deconstructing the structure and binding its component values to variables. For example, given that `t` is a tuple of two components, we can bind `f` and `s` to the first and second components by `fst(t)>x> snd(t)>y> ...`. A much simpler way is to use patterns whereby the site calls for necessary deconstruction are generated by the compiler. For this example, we will write `t >(x,y)> ...`.

4.3.5.1 Pattern Structure

A pattern is either a (1) literal, (2) variable, (3) wild-card, or (4) a structure (tuple, list or record) each component of which is a pattern. A literal is a constant, as described in Section 4.3.1 (page 89), a variable is simply a variable name and a wild-card is written as _. The only restriction on a pattern is that the variables in it be distinct; so, (x,x,y) is an illegal pattern. Here are some examples of patterns.

Example

```
4                          --  a pattern that is just a literal
x                          --  a pattern that is just a variable
_                          --  a pattern that is just a wild-card
(3,4)                      --  Tuple pattern with two literals
(_,4)                      --  Tuple pattern that includes a wild-card
[x,y]                      --  List pattern that includes variables
(x,_,(y,[(_,_),(z,4)]))    --  nested pattern
```

4.3.5.2 Pattern Match

Pattern p *matches* value v, primitive or structured, if it is possible to replace every variable in p by a value to obtain v. Here, we regard every wild-card in p as a distinct variable. We explain how a pattern match and the corresponding bindings are determined in a top-down style; there is also a slight exception in the case of record structures, which we describe below.

If p is a literal, it matches v if and only if v is exactly the same literal. If p is a variable or wild-card, it always matches v with v as the value of the variable. If p is a tuple or list, then it matches v if and only if v has the same structure and each of their corresponding components match (hierarchically). If p is a record then it matches v if and only if v is a record, the key names in p is a *subset* of the key names in v, and the corresponding key bindings match; subset matching is the only exception to the general rule.

Example

Pattern	Structured value	Matches?	Variable Binding
`(x,y)`	`((3,4),4)`	Yes	`x = (3,4), y = 4`
`(_,_)`	`(3,4)`	Yes	
`h:t`	`[1, 2, 3]`	Yes	`h = 1, t = [2, 3]`
`{. a = x .}`	`{. a = "no", b=[2] .}`	Yes	`x = "no"`
`(x,y)`	`[3,4]`	No	
`(x,(y,z))`	`((3,4),4)`	No	

4.3.5.3 Pattern Usage

Patterns can be used whenever an expression publishes a structured value that has to be deconstructed for use in other expressions. For example, `(f,g)` implements a fork-join of the expressions `f` and `g`, and publishes a tuple; we may access the individual components of the tuple by `(f,g)>(x,y)> ...`. Therefore, patterns may be used in place of just variables in sequential and pruning combinators, and in definitions of sites. We show some examples below.

Example

- Dismantle a tuple

```
(3,6,9) >(x,y,z)> ( x | y | z )
```

publishes the elements of a triple in arbitrary order.

- Convert a set of tuples to a set of lists

```
( (3,4) | (2,6) | (1,5) ) >(x,y)> [x,y]
```

publishes `[1, 5], [2, 6], [3, 4]` in arbitrary order.

- Deconstruct an argument

```
def sum(y) = if (y = []) then 0 else (y >x:xs> x + sum(xs))
```

Site `sum` publishes the sum of the elements of its argument list.

- Using wild-card

```
(0,(2,2),[5,5,5])  >(_,(_,x),_)>
[[1,3],[2,4]]      >[[_,y],[_,z]]>
[x, y, z]
```

publishes `[2, 3, 4]`.

• Filtering

```
((false, true) | (true, false) | (false, false)) >(true, x)>
    x
```

publishes the second component of each tuple whose first component is `true`; in this case just one `false` is published.

• Pattern matching with records

```
   {. name = ("Harry Q.","Error"), college = "CMU",
      accepted = false .}
 >{. name = (first,last), accepted = true .}>
first+" " + last + "'s application was accepted"
```

does not publish, whereas

```
{. name = ("Sally", "Hacker"), college = "M.I.T.",
   accepted = true .}
 >{. name = (first,last), accepted = true .}>
first+" " + last + "'s application was accepted"
```

publishes `"Sally Hacker's application was accepted"`.

• Pattern in site definition

```
def implies(false, _) = true
```

This definition defines a site only for the first argument `false`; it publishes `true` no matter what the second argument is. If it is called with the first argument `true`, the site execution halts, and so does the site call itself. Such patterns appear in clausal definitions of sites; see Section 4.4.2.3 (page 101).

Pattern Usage in Combinators Given `f >p> g` where `p` is a pattern, any publication of `f` that fails to match `p` is simply ignored. In particular, if `f` publishes exactly one value that does not match `p`, then `f >p> g` halts at that point. This is particularly useful for matching a value `v` against a sequence of patterns in order: `v >p_1> 1 ; v >p_2> 2 ; v >p_3> 3 ...` publishes `i` where `p_i` is the first pattern in the sequence that matches `v`. Similarly, in `f <p< g` any publication of `g` that fails to match `p` is ignored, and *the execution of* `g` *continues*.

The wild-card pattern matches every value. So, `f >_> g` is same as `f >> g` and `f <_< g` is `f << g`.

4.4 Declaration

The only form of declaration in Orc calculus is site definition. We describe enhancements of site definition, including patterns in formal parameters, in Section 4.4.2.2 (page 100). Orc language also includes other forms of declaration: **`val`** (see Section 4.4.1) and **`def class`** (see Section 4.6, page 113). Orc language syntax from Table 3.1 (page 45) have been extended by these constructs in Table 4.1, below.

f,g	$\in$	*Expression*	::=
		stop	Basic Expression
	|	$e(\bar{x})$	Site Call
	|	$f \mid g$	Parallel Combinator
	|	$f >x> g$	Sequential Combinator
	|	$f <x< g$	Pruning Combinator
	|	$f \texttt{;} g$	Otherwise Combinator
	|	$D \# f$	prefixing declaration
D	$\in$	*Declaration*	::=
		`def` $E(\bar{x}) = f$	Site Definition
	|	**`val`** $x = f$	Val declaration
	|	**`def class`** $E(\bar{x}) = f$	Class declaration
		Program	::= f

Table 4.1: Orc Language Syntax

As before, a declaration is terminated by a #. The # is optional, i.e., it can be replaced with a white space if the next non-white space is an alphanumeric symbol. We omit it wherever possible, yet we sometimes use it for readability even though it may be optional. As before, the expression part in a declaration is called its *goal expression*.

4.4.1 val

Declaration **`val`** `x = g` binds `x` to the first publication of `g` and terminates the execution of `g`. This can be used to rewrite an expression with a pruning combinator, `f <x< g`, such that the relative positions of `f` and `g` are switched and variable `x` appears first:

```
val x = g
  f
```

This syntactic sweetener is so useful that direct use of the pruning combinator is mostly unnecessary.

Multiple val declarations may appear consecutively; they are merely nestings of the pruning combinator.

```
val radius = 1.0
val pi     = 3.1416 #
(2*pi*radius, pi*radius**2)
```

This stands for

```
((2*pi*radius, pi*radius**2)<pi< 3.1416)<radius< 1.0
```

It should be clear from this example why we prefer to use **val** instead of the pruning combinator in such cases.

Let us define a pair of sites, `perim` and `area`, to compute the circumference and the surface area of a circle.

```
val pi       = 3.1416
def perim(r) = 2*pi*r
def area(r)  = pi*r**2
```

Suppose we want to access `pi` only in the two sites shown above. We can create a rudimentary package facility using nesting of declarations, embedding sites and other **val**s within a **val**.

```
val circle =
  val pi       = 3.1416
  def perim(r) = 2*pi*r
  def area(r)  = pi*r**2 #
(perim,area)
```

Now, the names `pi`, `perim` and `area` are inaccessible from outside the declaration of `circle`. And, `circle` is a tuple of two site closures. These sites may be accessed as follows to compute the circumference and surface area for a specific `radius`: `circle >(p,a)> (p(radius),a(radius))`.

Suppose we wish to have access to the two sites `perim` and `area` all over the program, but call them `circumference` and `surface_area`. We can add to the given program:

```
val (circumference,surface_area) = circle
```

A **val** declaration is used to assign a name to a value that we plan to use multiple times in the program. Here is a small relational database.

```
val applicants =
[
  {. name = ("Harry Q.","Error"), college = "CMU",
     accepted = false .},
  {. name = ("Sally", "Hacker"), college = "M.I.T.",
     accepted = true .},
  {. name = ("D. F.", "Virus"), college = "Podonk U.",
     accepted = true .}
]
```

We may write programs using `applicants` to stand for the corresponding list:

```
each(applicants) >a>
(
    a >{. name = (first,last), accepted = true .}>
```

```
        first+" " + last + "'s application was accepted"
    | a >{. name = (first,last), accepted = false .}>
        first+" " + last + "'s application was rejected"
)
```

publishes in arbitrary order:

```
"Harry Q. Error's application was rejected"
"Sally Hacker's application was accepted"
"D. F. Virus's application was accepted"
```

A **val** declaration can contain any Orc expression; therefore, the value assigned to the variable may be non-deterministic or based on real time. Here is a reworking of timeout from Section 2.5.2.5 (page 35) that binds `true` to variable `x` if `f` publishes within `k` time units and `false` otherwise.

```
val x = f >> true | Rwait(k) >> false
```

The expression in **val** may never publish a value though it may continue to execute. This is often a convenient way to do a background computation, such as monitoring a program execution or collecting statistics.

Translation of val Expression (**val** `x = g # f`) is translated to `f <x< g`. So, (**val** `_ = g # f`) is `f << g`.

4.4.2 Site Definition

We have described site definition and closure in Section 2.4 (page 27). Sites may be nested to any depth within other declarations. Orc language includes additional syntactic constructs relating to sites, mutual recursion, patterns in parameters and clausal definitions, that we describe in the following sections.

4.4.2.1 lambda construct

It is sometimes useful to introduce an anonymous site whose definition is directly used in place of a call to the site. Such a site has a parameter list and a body but no name. The site closure, without the name, is then available to be used wherever the site name could have been used. An expression of the form

```
lambda(X) = f
```

where `X` is a list of parameters and `f` is any expression, is equivalent to

```
def temp(X) = f # temp
```

where `temp` is a fresh name. This program merely publishes the closure corresponding to `temp`, exactly what the **lambda** construct does.

As a small example, consider a site with three arguments; the first one is a site that is to be applied to the other two arguments.

```
def apply(fun,x,y) = fun(x,y)
```

We wish to replace `fun` by `addsq` which is defined as:

```
def addsq(x,y) = x**2 + y**2
```

The call `apply(addsq,3,4)` is equivalent to:

```
apply(lambda(x,y) = x**2 + y**2,3,4)
```

The advantage of this form is that it eliminates the name `addsq`. We see more substantive examples of **`lambda`** usage later in this chapter.

Site Composition Let `f` be a site without any argument that publishes at most one value and `g` a site that takes just one argument and publishes an arbitrary number of values. Then the composition of `f` and `g` is a site that first executes `f` and supplies its publication to `g` as argument, as defined below.

```
def comp(f,g) = ( lambda() = g(f()) )
```

For example, `comp(lambda()= 3,lambda(x)= 2*x)()` publishes 6.

For the more general case where `f` has, say, 2 arguments, but at most one publication, define composition by

```
def comp2(f,g) = ( lambda(x,y) = g(f(x,y)) )
```

As an application of `comp2`, define arithmetic and geometric mean of two real numbers by the following closures. Each of these closures take two real numbers as arguments.

```
val amean = comp2( lambda(x,y) = x+y, lambda(v) = v/2+0.0 )
val gmean = comp2( lambda(x,y) = x*y, lambda(v) = sqrt(v) )
```

We observe that site composition is not necessarily associative, unlike composition in functional programming.

4.4.2.2 Patterns in Formal Parameters

The parameter in a site definition may be a pattern. When such a site is called, the actual and formal parameters have to match. The argument values are matched against the pattern as described in Section 4.3.5.2 (page 94). The matching and execution of the site body are performed along with the evaluations of the arguments, as described below.

The arguments of the call are evaluated concurrently. Any time a pattern match fails, because a literal value in the site definition does not match the corresponding argument value, some argument evaluation halts where the expected value is supposed to match a literal, or the structures are different, the call halts. When it is determined that the pattern match will succeed, even though some of the arguments may not have been evaluated, the execution of the site's body starts. Subsequently, some of the arguments will be bound to values or their evaluations may halt without publishing a value. In the first case, the corresponding formal parameter is bound to the same value. In the latter case, the corresponding formal parameter is bound to the fictitious value, $\not!$, introduced in Section 3.4.6 (page 50). Recall that any strict site call in which $\not!$ is an argument is equivalent to **`stop`**.

Consider

```
def test(false,x) = ...
```

For a call `test(p,q)`, where `p` and `q` are expressions, both arguments are evaluated concurrently. If the evaluation of `p` halts or if it is a value other than `false`, then the call halts. If `p` is `false`, then the evaluation for `q` continues (unless it is already complete) concurrently with the execution of the body of `test`. If the evaluation of `q` halts without publishing a value, `x` is assigned the fictitious value ↯. If the evaluation of `q` publishes value `v`, then `x` is bound to `v`. If the evaluation of `q` never publishes nor halts, then `x` is never bound to a value.

4.4.2.3 Clausal Definition

A site may be defined by a sequence of *clauses*: repeated site definitions with the same identifier but different parameters. A clause defines the behavior of the site for a subset of its possible argument values. A clausal definition is useful when different subsets of argument values have substantially different behaviors.

Each clause in a site definition must have the same number of parameters. The clauses are tested in sequence from top to bottom to identify one whose parameters match the actual parameters of the call. If no clause matches, then the site call halts. Clauses may include patterns for their parameters, and the patterns are matched exactly as described in Section 4.4.2.2, above.

We saw an example of clausal definition at the beginning of this chapter:

```
def exp(0) = 0
def exp(n) = ...
```

The first clause, `def exp(0)= 0`, describes the site behavior if the call argument is 0, and the next clause, `def exp(n)= ...` for all other argument values. When `exp` is called with some parameter value `v`, the clauses are tested in sequence from top to bottom to identify the first one whose parameters match `v`; body of the first such clause is executed.

Example

• Fibonacci: Here is a definition of Fibonacci in clausal form.

```
def fib(0) = 0
def fib(1) = 1
def fib(n) = fib(n-1) + fib(n-2)
```

Here is a more efficient version of Fibonacci.

```
def fib(n) =
  def H(0) = (0,1)
  def H(n) = H(n-1) >(x,y)> (y,x+y)
H(n) >(x,_)> x
```

• List of random bits: The following site publishes a list of random bits where the list length is specified as the parameter.

```
def randlist(0) = []
def randlist(n) = Random(2): randlist(n-1)
```

• Exhaustive clausal definition: The following definition of a site enumerates all its parameter values.

```
def implies(false,false) = true
def implies(false,true)  = true
def implies(true,false)  = false
def implies(true,true)   = true
```

We may write this definition more simply:

```
def implies(true,false)  = false
def implies(_,_)         = true
```

• Areas of Geometric Figures: The following site computes the areas of certain geometric figures.

```
def area(("rectangle",x,y)) = x*y
def area(("square",s))      = area(("rectangle",s,s))
def area(("circle",r))      = 3.1416 * area(("square",r))
```

Observe that the pattern in the formal parameter is a tuple in each case. Therefore, each clause has just one parameter, though the tuples are of different length.

• Clausal Definition with real-time computation: Given below is a site that has a list of sites as its argument. It calls all argument sites concurrently to request a bid for an auction. A site may return a bid, an integer. If it does not respond within 8 seconds, its bid is taken to be 0. The site publishes the highest bid.

```
def auction([])   =  0

def auction(b:bs) =
    max(b() | Rwait(8000) >> 0, auction(bs))
```

Site `max` publishes the maximum value of its two arguments. The first argument of `max` implements a simple time out: if a response from `b` is received within 8 seconds, the response value is used, otherwise, `Rwait(8000)` completes and the value of the first argument is 0. The second argument of `max` is simply a recursive call to obtain the highest bid from the sites in the tail of the argument list. The two arguments of `max` are evaluated concurrently; therefore all bidders will be called concurrently.

Translating Clausal definition A site with an empty parameter list does not have a clausal definition, because all except the first clause are redundant. For a site with a single parameter a clausal definition is of the following form where `a`, `b` and `c` are constants, variables or wild-card (all clauses after a variable or wild-card for a parameter pattern are redundant):

```
def fun(a) = f
def fun(b) = g
  ...
def fun(c) = h
```

Transform this code fragment to

```
def fun(x) =
  x >a> f; x >b> g; ... ; x >c> h
```

Unfortunately, this translation is incorrect. If the argument matches the pattern `a` and `f` is **`stop`**, then the next pattern `b` will be matched against the argument. To avoid this problem, first match the patterns and publish an index corresponding to the pattern that matched, and then choose the proper clause body to execute. Below `n` represents the number of clauses.

```
def fun(x) =
  (x >a> 0; x >b> 1; ... ; x >c> n) >v>
  (  v >0> f
   | v >1> g
   | ...
   | v >n> h
  )
```

If none of the patterns match, the otherwise construct halts and so does the site call.

For a site with multiple parameters, we give the translation for a special case: all patterns are strict. So, values of all parameters have to be known before the matching clause can be identified. For the general case of lenient patterns, the translation is more elaborate and we omit it here. The translation for the special case resembles the one given above, by treating the actual parameter list as a tuple value and the formal parameter list of each clause as a tuple pattern. For example,

```
def fun(a,a') = f
def fun(b,b') = g
  ...
def fun(c,c') = h
```

is translated to

```
def fun(x,y) =
  ((x,y) >(a,a')> 0; (x,y) >(b,b')> 1; ... ; (x,y) >(c,c')> n)
 >v>
  (  v >0> f
   | v >1> g
   | ...
   | v >n> h
  )
```

4.4.2.4 Mutual Recursion

Sites may be mutually recursive. There is no special syntax for mutual recursion; mutually recursive definitions appear contiguously in the program text. We consider a small example, removing redundant white space from a string.

Given a list of symbols, it is required to publish the same list by (1) removing all white spaces in its beginning, and (2) reducing all other blocks of consecutive white spaces to a single white space. Thus, given the input string (where `+` denotes a white space) `+++Mary++++had+a++little+++lamb++`, the program should publish `Mary+had+a+little+lamb+` and then halt. The problem is easily solved using a finite state transducer, as shown in Figure 4.1; here x refers to a non-white space. The initial state, `first`, ignores all white space, and on reading a non-white space reproduces it and transits to state `next`. State `next` reproduces every non-white space, and on reading a white space reproduces it and transits to state `first`.

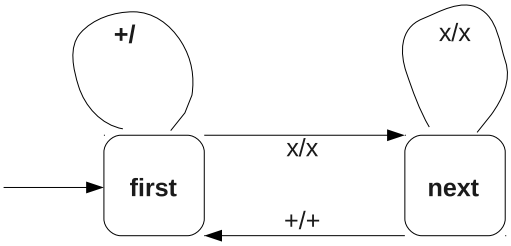

Figure 4.1: White Space Compression

We can represent this finite state machine by a program where each state corresponds to a site. The input is given as a list of symbols. We use pattern matching to extract the head and tail of a non-empty list.

```
def compressWhite(str) =
  def first ([])   =  []
  def first(" ":xs)=  first(xs)
  def first(x:xs)  =  x:next(xs)

  def next([])     =  []
  def next(" ":xs) =  " ":first(xs)
  def next(x:xs)   =  x:next(xs)

first(str)
```

Translating Mutual Recursion We encode a set of mutually recursive sites by a single site, `F`. Parameters of `F` mimic the parameters of the original sites, but there is an additional parameter, `b`, that identifies the specific site that is being called. Site `F` is clausally defined corresponding to each constituent site. A call to an original site is replaced by call to `F` with the appropriate value of

b. We illustrate the procedure with the pair of sites first and next defined above.

Introduce a new site firstnext that has a boolean parameter b to identify the site being called, true for first and false for next.

```
def firstnext(true, [])      =  []
def firstnext(true, " ":xs)  =  firstnext(true,xs)
def firstnext(true, x:xs)    =  x:firstnext(false,xs)

def firstnext(false, [])     =  []
def firstnext(false, " ":xs) =  " ":firstnext(true,xs)
def firstnext(false, x:xs)   =  x:firstnext(false,xs)
```

Next, we redefine the original sites so that they can be called from outside the body of F. For the example above:

```
def first(p) = firstnext(true,p)
def next(p)  = firstnext(false,p)
```

Any use of first as an argument to another site, i.e., as a closure, within the definition of firstnext is replaced by

```
lambda(x) = firstnext(true,x)
```

4.4.2.5 Effect of Concurrent Calls

A site may be called concurrently from several parts of a program. The site behavior may be different for the same set of sequential and concurrent invocations. The difference is not merely because of interference due to concurrent executions, but that the site computation may be time-based. To illustrate the difference, we consider the well-known site *map* over lists: map(f,xs), where f is a site and xs a list of possible arguments of f, publishes a list of the same length as xs. The element in the published list corresponding to element x of xs is f(x). In a language like Haskell [15] this may be coded by

```
map(_, [])   = []
map(f, x:xs) = y : map(f,xs)
                 where y = f(x)
```

This piece of code may apply f to each element of the list in arbitrary order. Since f is side-effect free, execution is sequential, and each application of f publishes the same value each time it is called with the same argument, the resulting mapped list is identical no matter the order in which f is applied.

The Orc program below explicitly specifies that f should be applied over the elements of the list sequentially from left to right. Observe that there is at most one instance of f executing at any time.

```
def seqmap(_, [])   = []
def seqmap(f, x:xs) = f(x) >y> (y : seqmap(f,xs) )
```

And, the following program applies f sequentially from right to left.

```
def seqmap'(_, [])   = []
def seqmap'(f, x:xs) = seqmap'(f,xs) >ys> (f(x) : ys)
```

Site `parmap`, which is defined below, applies `f` concurrently over the elements of `xs`. Deflation forces all calls to `parmap` to unfold concurrently. Assume for the moment that concurrent executions of `f` do not interfere.

```
def parmap(_, [])   = []
def parmap(f, x:xs) = f(x) : parmap(f,xs)
```

The three versions may publish different results if, for example, `f` waits for a day and then queries a news site. Queries will be staggered by a day each in both `seqmap` and `seqmap'` whereas they will be made simultaneously in `parmap`.

4.5 Factory Sites

A *higher-order site* is one that accepts a site as argument of a call or publishes a site as a result. In this section we introduce a number of standard *factory* sites that are higher-order sites, similar to a class in object-oriented programming [19, 35]. A factory site creates and publishes an instance, called an *object*, that has its own store, and has methods by which the store may be accessed and/or updated. The methods of an object are sites. An example of a factory site is `channel` that creates an unbounded channel. A factory site may be called repeatedly to create multiple instances of similar objects.

Methods as Sites In object-oriented programming languages, `obj.m()` denotes a call on method `m` of the object `obj`. Instances of factory sites typically include methods, and we adopt a similar notation in calling them. For example, channel `c` might include methods `get` and `put` to get values from and put values on that channel, respectively. Such calls would be written as `c.get()`, or to put a particular value, `c.put(6)`. Actually, the channel object is an Orc record in which the keys are the methods and the bindings are closures (denoting sites). Therefore, `c.put(6)` retrieves the closure corresponding to `c.put` and calls it with argument `6`. We show translations of factory sites with methods in Section 4.6.6.

It is possible for an object to have just one method. In that case, typically, the object is itself a site that is called directly.

Dots are left associative, so `a.b.c(x)` is `((a.b).c)(x)`.

Atomicity An object with methods is *atomic* if concurrent executions of multiple methods have the same effect as executing the methods sequentially in some order. An implementation can not serialize the execution simply by the sequence in which the calls are made. For example, a call waiting to receive an item from a channel that is empty at the time of the call must permit execution of a method that puts an item in the channel.

A site specification should describe if it creates atomic objects. All factory sites in the standard library create atomic objects. Objects created by classes, see Section 4.6, page 113, are not necessarily atomic.

4.5.1 Factory site `Ref`

A call to `Ref()` publishes a site that behaves as a rewritable storage location, also called a *reference*. A reference is a site with two methods, `read` and `write`. The `read` method publishes the current value stored at the location; it blocks if no value has been stored yet. The `write` method overwrites the stored value at the location by the argument of the call and publishes a signal; it never blocks. There is no restriction on the kind of value that can be stored at a location.

```
Ref() >r> r.write(3) >>  r.read()
```

publishes 3.

A call to `Ref` may optionally include the initial value to be stored at the location. Thus, `Ref(3)>r> r.read()` is another way of writing the last example.

If the site published by `Ref` is to be used over a wide scope, it is preferable to write:

```
val r = Ref()
```

which permits `r` to be used over the scope of the declaration of this **`val`**.

Notation We introduce the following abbreviations for any site that has methods named `read` and `write`:

`r?` is translated to `r.read()`, and
`r := x` to `r.write(x)`

Using this notation, `Ref()>r> r.write(3)>> r.read()` may be written as `Ref()>r> r := 3 >> r?`.

A common source of error in connection with `Ref` is to use `r` where `r?` is intended, the former publishes the location whereas the latter publishes the value stored at the location. Imperative programming languages do not usually support publishing a location, just the value at a location. So, they use `r` for the value stored at location `r`, different from our usage.

4.5.2 Factory site `Cell`

A call to `Cell()` creates a write-once storage location. It has the same two methods as `Ref`, `read` and `write`, with similar meanings and notational abbreviations, except that a call to `write` halts if the location already has a value.

4.5.3 Factory site Semaphore

A call to `Semaphore(k)` publishes a `semaphore` with initial value `k`, where `k` is a non-negative integer. A semaphore has two associated methods: `acquire` and `release`, commonly known as `P` and `V`, respectively, in the literature. The `acquire` method decrements the value of the semaphore provided it is positive and then publishes a signal; if the semaphore value is zero, the call is *blocked*, and the caller remains waiting for the value to become positive. The `release` method increments the value of the semaphore, and then publishes a signal. Execution of a `release` is never blocked, and may, possibly, unblock a blocked caller to `acquire`. It follows that the semaphore value is always non-negative since it is decremented only if it is positive.

A semaphore guarantees weak fairness for its callers to `acquire`: (1) a `release` unblocks an arbitrary blocked caller to `acquire` if there is more than one such caller, and (2)for any particular semaphore, an execution has either a finite number of `release`s or every blocked caller to `acquire` is eventually unblocked.

A semaphore is used to serialize concurrent computations, as in the following example.

Example

Define a site that increments the value at the reference taht is supplied as the site argument; assume that the value stored at the reference is integer. Here is a possible program.

```
def inc(r) = r := r? + 1
```

The body of `inc` will be expanded to include a call to `r.read` followed by a call to `add` and then a call to `r.write`. Site `inc` may be called concurrently, as in `inc(r)| inc(r)`. Concurrent executions of `inc` may interfere; as a result `inc(r)| inc(r)` may not increase the value stored at `r` by 2. We can remedy this problem using a semaphore.

```
val s = Semaphore(1)
def inc(r) = s.acquire() >> r := r? + 1 >> s.release()
```

Note that s is declared outside the site body so that it retains its value across different calls to `inc`.

This solution has one major drawback. Given `inc(r)| inc(t)`, where `r` and `t` are two different references, their incrementations will be serialized because a single semaphore controls access to all references. This is particularly troublesome if reference `r`, say, has not been initialized whereas `t` has been; then, incrementation of `t` will have to wait until `r` gets initialized; so, it may wait forever. We can resolve this problem by associating a different semaphore with each reference, and calling `inc` with both the semaphore and the reference as arguments.

```
def inc(r,s) = s.acquire() >> r := r? + 1 >> s.release()
```

This requires that whenever a reference is created, a semaphore is created as well and associated with it. We can define a factory site `Ref'` using the class construct (see Section 4.6) that publishes a tuple `(r,s)`, where `r` is a reference and `s` the associated semaphore.

4.5.4 Factory site `Channel`

A call to factory site `Channel()` creates and publishes an asynchronous unbounded FIFO channel (also called a *buffer*). There are two methods on a channel, `put` and `get`. The number of items in a channel is the number of completed puts minus the completed gets; the channel is *empty* if the number of items in it is zero and *non-empty* otherwise. Upon creation, a channel is empty. The call `c.put(v)` for a channel `c` appends `v` as the last item of `c` and then publishes a signal; this operation is never blocked and the resulting channel is non-empty. The call `c.get()` removes the first value from `c` and publishes it, or blocks waiting for a value if the channel is empty.

The following program publishes the value 10 after a 1000 msec delay.

```
val c = Channel()
  Rwait(1000) >> c.put(10) >> stop
| c.get()
```

Site `Channel()` is among the most important of the standard factory sites. It can be used to simulate a `Semaphore`, `Ref` or `Cell`. It is the basis for creating process networks in which processes (also called *actors*) compute autonomously and interact with other processes using channels; see Section 8.2 (page 208). Below, we show few small examples of the use of channels.

Producer-Consumer The producer-consumer paradigm, see Dijkstra [13], is a simple process network in which a channel is used as a buffer to smooth the variation in the speeds of independent computations. Site `producer()` generates an item each time it is called. A call to `consumer(v)` consumes item `v` and publishes a value. It is required to call both producer and consumer repeatedly to produce and consume values. A possible solution is shown below in which site `repP` calls `producer()` repeatedly and publishes all the values that it generates.

```
def repP() = producer() >x> (x | repP() )

repP() >v> consumer(v)
```

A drawback of this solution is that the values produced may be consumed out-of-order. A more serious objection is that concurrent executions of different instances of `consumer` may interfere. We remedy both problems in the solution below, employing a channel `in` on which the producer puts all its publications; the `consumer` consumes one item at a time from this channel and puts its publications on channel `out`. Below, `repeatP` calls `producer()` repeatedly and publishes all the values that it generates; similarly, `repeatC` removes each item from channel `in`, consumes it and puts the result on channel `out`.

```
   val in  = Channel()
   val out = Channel()
   def repeatP() = producer() >x> in.put(x) >>  repeatP()
   def repeatC() =
       in.get() >x> consumer(x) >v> out.put(v) >> repeatC()

repeatP() | repeatC()
```

Sequencing Computations We show how to run two computations sequentially using a channel. It is required to first execute expression `f` and then `g` after `f` halts; if `f` never halts `g` is never executed. This is a considerable generalization of sequencing in imperative or functional programming because `f` may publish multiple values or no value and may or may not halt. If `f` publishes exactly one value and then immediately halts, `f >x> g` suffices. If `f` publishes no value and then halts, `f ; g` suffices. For the general case, we convert `f` to `f'` where `f'` behaves as `f` but does not publish; publications of `f` are stored in channel `ch`. Then `f' ; g` has nearly the behavior we desire, except for the publications of `f`. We add a parallel computation, `rep(ch)`, to repeatedly read from `ch` and publish the values.

```
  def rep(c) = c.get() >x> (x | rep(c) )
  val ch     = Channel()

    f >x> ch.put(x) >> stop ; g
  | rep(ch)
```

Multiple publications with Pruning Let expression `g` publish tuples; the first component, called a *tag*, is a boolean, and the second component is the *value*. It is required to publish all the values that `g` publishes until it publishes a tuple with `false` tag, and then terminate `g`.

Since `g` may have to be terminated, `g` has to appear in the right side of a pruning combinator. A normal application of the pruning combinator, as in `f <P< g`, where `P` is any pattern, can transfer at most one publication of `g` to `f`, because a publication that does not match `P` is discarded and a publication that matches `P` terminates `g`. Using a channel, however, we can communicate all the desired publications of `g` to `f`. In the solution below, every publication of `g` with a `true` tag is put on channel `c`. The right side of the pruning combinator publishes a **`signal`** only when `g` publishes a `false` tag. We use site `rep` defined in the previous example.

```
   val c = Channel()

   rep(c)  <<
           (g >x> v >>
              (  v >(true,v')> c.put(v') >> stop
               | v >(false,_)> signal
              )
```

```
)
```

4.5.5 Factory site `Array`

The call to factory site `Array(n)` creates and publishes a site that behaves like an array of references. In the following discussion, call this publication `ar`. The call `ar(i)`, for $0 \leq i < n$, publishes the corresponding reference. Just as any other reference, `ar(i).write(v)` (or equivalently `ar(i):= v`) assigns value `v` to element `ar(i)`, and `ar(i).read()` (or `ar(i)?`) publishes its value. All array elements are initialized to a special value *null*. The length of `ar` is published by `ar.length?`. Note that an array of zero length is permissible. This is useful if arrays of different lengths have to be created depending on the value of a non-negative parameter.

```
Array(3) >a> a(2) := 2 >> a(2)?
```

publishes 2. Here is a more elaborate example.

```
val ymc =
    Array(3) >d>
    d(0) := "yellow"  >>
    d(1) := "magenta" >>
    d(2) := "cyan"    >>
    d
```

Variable `ymc` is an array with three elements; elements are references with values `"yellow"`, `"magenta"`, `"cyan"`. Observe the computation within the **`val`** to assign values to array elements, ending with the publication of the entire array.

We show how to create arrays with specified indices (not just starting at 0) and multidimensional arrays in Section 5.4.7.

We often want to create an array of specific objects, not just references. The program fragment below creates an array of 3 channels.

```
val chs =
  Array(3) >ar>
     ar(0) := Channel() >>
     ar(1) := Channel() >>
     ar(2) := Channel() >>
     ar
```

Now `chs` is an array of references each of which has a channel as a value. For example, `chs(2)` is a reference to a channel, not a channel itself; `chs(2)?` is the channel and we can call `chs(2)?.put(5)`, for instance. To simplify access to the channels, define site `chacc` that allows direct access to channel `i` by calling `chacc(i)`.

```
def chacc(i) = chs(i)?
```

This solution, to create an array of channels and call them directly bypassing the references, is quite elaborate. The need to define an array of objects, and refer to its elements directly without going through a layer of reference value, is so common that we introduce another factory site, `Table`, to do just that.

4.5.6 Factory Site `Table`

`Table` is a site derived from `Array`. A call to `Table(n,fun)`, where `n` is a natural number and `fun` a site over natural numbers, publishes a site `fun'` that mimics `fun` over the first `n` natural numbers, as follows. When `Table(n,fun)` is called `fun(i)`, for all natural numbers `i`, $0 \leq i < n$, are called concurrently and their first publications are stored in an array as the corresponding values of `fun'`. Then `fun'` always returns the stored value. Even though `fun` may be non-deterministic or time-dependent, `fun'` always returns the same value. The call to `Table` does not return a value until all calls to `fun` have completed. Therefore, if `fun(i)` halts for some `i`, $0 \leq i < n$, so does `Table(n,fun)`. If no `fun(i)` halts, but the computation is silent for some, then so is `Table(n,fun)`. The only operation on `Table(n,fun)` is to access one of its elements by executing `fun'(i)`. In particular, the elements of a `Table` can not be overwritten with new values.

We show how to *cache* an array of immutable values, such as an array of fixed integers, as well as create arrays of objects, such as channels and semaphores, using this feature. Consider:

```
val p = Table(5,id)
val q = Table(5,Semaphore)
val r = Table(5,Ref)
```

Given that `id` is the identity site, `p(i)` publishes `i` and it can be replaced anywhere by `i`. We may regard `p` as an array where `p(i)` is `i`. The declaration for `q` is more interesting; `q(i)` is the value published by `Semaphore(i)`, that is, a semaphore with initial value `i`. Different calls to `q(i)` publish the same semaphore. We can update the semaphore value by calling `q(i).acquire()` and `q(i).release()`, but `q(i)` never publishes a different semaphore nor any other object. And, `r` is an array of references where `r(i)` is initialized to `i`.

Quite often, we want to call `Table(n,fun)` where `fun` has no parameter, such as `Channel()` to construct an array of channels. We can define an auxiliary site `Channel'`, say, that has one parameter, which the site ignores and publishes a channel; then use `Table(n, Channel')`. We use the **`lambda`** construct described in Section 4.4.2.1 (page 99) to define and use `Channel'` anonymously.

```
val sq = Table(5, lambda(i) = i*i)
val ch = Table(5, lambda(_) = Channel())
val s0 = Table(5, lambda(_) = Semaphore(0))
```

Here, `sq` is an immutable array of the squares of the first 5 natural numbers. And, `ch` is an array of channels. Unlike `q` of the previous example, `s0` is an array of semaphores each of which is initialized to 0.

If `Table` is often called with a second argument that is a site without parameters, it may be useful to use `Table0`, defined below, instead.

```
def Table0(n,f) = Table(n,lambda(_) = f())
```

Example

This example illustrates the use of an array of channels. A multiplexor process reads inputs from `n` input channels in a round robin fashion where n is defined elsewhere and writes each value to an output channel. First, we declare `in` as an array of `n` channels, and then define the site, `mux`, corresponding to the multiplexor.

```
val in  = Table0(n,Channel)
val out = Channel()

def mux(i) =
    in(i).get() >x> out.put(x) >>
    mux((i+1) % n) -- "% n" is the value modulo n

mux(0)
```

Implementation of `Table` A `Table` is defined as a site, similar to the one used for creating an array of channels in Section 4.5.5. Below, `ar` is the array where the table elements are stored, `fill(i)` populates the first i elements of `ar` concurrently. Site `araccess` publishes the *value* of `ar(i)`, and `Table` publishes the site `araccess`.

```
def Table(n, fun) =
  val ar = Array(n)

    def fill(0) = signal
    def fill(i) = ((ar(i) := fun(i)), fill(i-1)) >> signal

    def araccess(i) = ar(i)?

fill(n-1) >> araccess
```

4.6 Class

The class construct allows coding of factory sites in Orc. It is sometimes more convenient to build a site in Orc, rather than in Java, Scala, or other object-oriented languages, because the site's internal structure is best expressed by an orchestration.

Like a typical class in object-oriented programming, a class in Orc provides an abstraction mechanism that encapsulates the implementation details of data structures and exposes only the method names by which they can be accessed or updated. Unlike a typical object, however, the methods of a class instance may be invoked concurrently, as is the case with site invocations in Orc. Further, an object may engage in autonomous computations without explicit method calls, for example sending monthly statements to bank customers (a periodic computation), alerts to listeners based on weather data (an event-driven computation),

or rearranging the internal data structure for more efficient subsequent access, such as doing a garbage collection (a background computation). Specifically, a composite site can be used to: (1) create objects with methods that may be concurrently invoked, (2) extend behaviors of existing sites, and (3) create active sites that are executed autonomously, and not necessarily through explicit method calls.

4.6.1 Class Syntax

A composite site is defined like a prime site except that **`def`** for a prime site is replaced by **`def class`**. A class definition is a *declaration* that may appear anywhere that a **`def`** or **`val`** may appear, see Orc syntax in Table 4.1 (page 97). Therefore, a class may be nested within another declaration and it may include other declarations, **`def`**, **`val`** or **`def class`**.

A class may be defined by a set of clauses, exactly as a prime site definition. In that case all clauses must include the same set of names for declarations: **`def`**, **`val`** and **`class`**. Classes may be mutually recursive, just like prime sites.

4.6.2 Class Semantics

A call to a class creates an object that is simply a record. Every declaration within the class becomes a part of the object: a **`val`** declaration binds a name to a value, a prime site definition is a method of the object, and a class definition is a class associated with the object that may be further instantiated. While the names of prime and composite sites defined within a class are available as keys of the record, the names associated with **`val`**s are not exported.

A class instance, an object, includes mutable variables defined using **`val`**. Different instances of that class will create different instances of the mutable variables. **`val`** declarations are analogous to private fields in object-oriented programming. A class declaration typically includes a **`val`**. Otherwise, class instances have no mutable store associated with them and the methods can be coded directly as prime sites.

Each **`def`** within the body of the class creates a method of the instantiated object; a method is itself a site. The methods of an object are called using dot access, which simply accesses the fields of the corresponding record. When a method is called, the corresponding site is executed, perhaps accessing the mutable objects created by **`val`**s. If a class includes no **`def`**, its instances have no methods; they can only engage in autonomous computations defined by the goal expression.

Each **`def class`** within the body of a class is simply a class which is associated with each instance. Like any other class, such a class may be instantiated and its methods called.

The methods of an object may be executed concurrently through concurrent invocations. Multiple calls to the same method may even be executed concurrently with each other. The methods are not necessarily atomic; so if a method

accesses mutable store, care must be taken to ensure that multiple method invocations do not interfere with each other in unexpected ways. A method may be made atomic through locking, for instance; see the example in Section 4.6.3.1.

Terminology We have two kinds of site declarations in Orc: (1) a site defined using the **`def`** construct is called a *prime* site, (2) a site defined using the **`def class`** construct is called a *composite* site. A composite site is also called just a *class* and its instances *objects*.

Note: In Chapter 3, a site defined in an Orc program was called *internal* in order to contrast it with *external* sites whose codes were not available. The only form of declaration in the Orc calculus uses **`def`**, so the internal sites considered in that chapter are prime sites. In the Orc language, an additional form of internal site, defined using **`def class`**, is available, and we use the new terminology to distinguish between them.

Semantical difference between Prime and Composite Site Both prime and composite sites, being sites, obey all the semantic rules for sites. Additionally, a composite site satisfies all the semantic rules for prime sites except for the following two differences. These differences are practical design decisions, motivated by our experience in solving a wide variety of problems.

1. (Publication) A prime site publishes the values resulting from the execution of its goal expression. A composite site executes its goal expression just like a prime site, but *its publications are ignored*. Unlike a prime site a composite site publishes exactly one value, the object.

   Since a class is a site, a call to it is lenient as for a prime site; so, the goal expression's execution may begin even before the parameters are bound. However, a class needs *all* its formal parameter values before it can publish the object.

2. (Resilience) In executing an expression of the form `f <x< g`, execution of `g` is terminated when `g` publishes. The executions of sites called from `g` may or may not be terminated since such sites are not necessarily under the control of Orc implementation. A site called from `g` whose execution continues under these circumstances is called *resilient*. A class and the methods of its instances are resilient, like most external sites.

4.6.3 Example of Class

4.6.3.1 Simple Counter

The following example implements a factory site for a simple counter. There are three methods: `inc`, which increments the value of the counter and publishes a signal; `dec`, which decrements the value of the counter and publishes a signal; and `mag`, which publishes the current value (magnitude) of the counter. The

initial magnitude of the counter is supplied as an argument at the time of class instantiation. The implementation stores the magnitude in a reference.

```
def class ctr(n) =
  val r      = Ref(n)  -- n is the initial counter value
  def inc()  = r := r? + 1
  def dec()  = r := r? - 1
  def mag()  = r?

  {- A ctr instance has no ongoing computation -}
stop
```

Below, we instantiate and use the simple counter.

```
val c = ctr(2)

c.dec() >> c.mag()
```

publishes 1.

Next, we instantiate two counters and access them concurrently. The concurrent accesses do not interfere because they refer to different mutable stores.

```
{- Test concurrent accesses -}

  val c1 = ctr(2)
  val c2 = ctr(0)

     c1.dec() >> c1.mag()
   | c2.dec() >> c2.inc() >> c2.mag()
```

Value 1 is published by the first alternative in the goal expression, as before, and 0 by the second alternative.

Concurrent Method Invocations Concurrent invocations of methods may interfere. Class construct does not automatically create atomic methods. In the program below, concurrent calls access and update the same counter concurrently, with disastrous results.

```
{- Test with multiple instances of ctr -}

  val c = ctr(1)

    c.dec() >> c.mag()
  | c.dec() >> c.mag()
```

If each method is atomic, i.e., a method call executes to completion before the next call to the method is started, we would expect one component expression to publish 0 and the other to halt. Since the methods are not atomic, it is possible to get two publications with value 0 (which is what happens in our tests with the Orc interpreter).

We can force a method to be atomic by using lock-based access. Below, executions of any pair of methods are mutually exclusive. This example also illustrates how to build an enhanced class without altering the underlying class.

```
def class ConcCtr(n) =
  val c   = ctr(n)
  val sem = Semaphore(1)

  def inc() =
    sem.acquire() >> c.inc() >> sem.release()

  def dec() =
    sem.acquire() >> c.dec() >> sem.release()

  def mag() =
    sem.acquire() >> c.mag() >x> sem.release() >> x


{- There is no ongoing computation -}
stop
```

Observe that each method must eventually release the semaphore so that competing method calls may acquire the semaphore and execute.

Inheritance We implement a primitive form of inheritance whereby we modify the behavior of one of the methods. Here, a call to `dec` when the counter value is 0 is blocked until it becomes positive and then the counter is decremented. We introduce semaphore `valsem` that has the same value as the counter.

```
def class BlockedConcCtr(n) =
val c      = ConcCtr(n)
val valsem = Semaphore(n)

def inc() = c.inc() >> valsem.release()
def dec() = valsem.acquire() >> c.dec()
def mag() = c.mag()

{- There is no ongoing computation -}
stop
```

Since we are using `ConcCtr` in this implementation at most one method of a class instance `c` may execute at any point. Hence, there is no race condition or other forms of interference in accessing the common store. There is no possibility of decrementing the counter to a negative value, because a call to `dec` is blocked until the counter value becomes positive.

4.6.3.2 Broadcast

All the goal expressions for classes shown so far have been merely **`stop`**. We show an example in which the goal expression of the class is engaged in a never-ending computation without publishing a value. Class `Broadcast` is instantiated with a channel `source` as its argument. The goal expression of a class instance repeatedly reads any available value from `source` and sends it to a set of listeners

along their individual input channels. Below, `listeners` is a reference whose value is a list of channels, the input channels of the listeners. Listeners may be added with the `addListener` method which is called with the input channel of the listener as an argument. As before, `rep(c)` repeatedly reads from channel `c` and publishes the value it reads, and `each(cs)`, where `cs` is a list, publishes the elements of `cs` in arbitrary order.

```
def class Broadcast(source) =
  val listeners = Ref([])

  def addListener(ch) =
    listeners? >fs>
    listeners := ch:fs

  {- The ongoing computation of a broadcast -}
  rep(source) >item> each(listeners?) >sink> sink.put(item)
```

This class does not allow removing a listener from the broadcast list; that can be added as an additional method to the class.

4.6.3.3 Symmetric Cell

We introduced factory site `Cell()`, a write-once store, in Section 4.5.1. A write operation on a cell halts if it has already been assigned a value and a read operation is blocked, waiting for a value to be assigned, if it is unassigned. For some applications, see Section 7.5.2 (page 200), it is useful to have a symmetric version of `Cell()` in which a read operation halts, instead of blocking, if the cell is unassigned.

Below, `symCell` implements such a store. Its data structure includes a reference `r` whose content is a list, and `c` a `Cell`. If the instance of `symCell` is unassigned, which is the case initially, `r` contains an empty list and `c` is unassigned. If the instance has been assigned value `x`, `r` contains `[x]` and `c` a signal. The `read` method, shown below, halts if `r` contains an empty list because the pattern match in `r? >[x]> x` fails. If the instance has a value, the pattern match succeeds and `read` publishes the value. The `write` method first attempts to store a signal in `c`, and if successful, stores the value in `r`.

```
def class symCell() =
  val (r,c) = (Ref([]), Cell())

  def read()   =  r? >[x]> x
  def write(x) =  c := signal >> r := [x]

stop
```

Linearizability The implementation of `symCell` does not use any lock for its operations beyond the atomicity provided in the operations on `r` and `c`. So, executions of concurrent reads and writes may potentially interfere. We

show that each operation may be regarded as atomic even though the steps of the operations may be interleaved. We draw upon the theory of linearizability, first promulgated in Misra [39] and later refined and generalized in Herlihy and Wing [20]. The essential result is that to prove atomicity it suffices to show that at any point in the computation on an instance of `symCell` there exists a linear order of executions of the completed operations on it that yields exactly the same result for each operation.

At any point in a computation, `r = []` if and only if no write has completed, because the first completed write stores its argument value as its last (atomic) step in `r`. If there is no completed write, then all the completed reads have halted, and they may be ordered arbitrarily. If there is a completed write, let `w` be the first write to have completed; `w` is well-defined since the order of the last step of a write determines the first completed write. The argument value of `w` is stored in `r` permanently because any writing into `r` is preceded by writing to cell `c`. Order the completed operations as follows: all completed reads that have halted precede `w`; all completed reads that return a value succeed `w` and all other completed writes succeed `w`. Note that any extension of the computation preserves this order because `w` remains fixed.

4.6.4 Export control of methods

The current implementation of class exports all the prime and composite sites defined within it so that they can be accessed from outside the class definition. It is often useful to specify a subset of the sites for export. We show a strategy of export control that can be easily automated. We illustrate the technique using the example of class `ctr`, defined in Section 4.6.3.1 (page 115).

Class `ctr` includes three site definitions, `inc`, `dec` and `mag`. Suppose we wish to export only `inc` and `mag`. Call this class `ctrSmall`, and define it as follows. Note that `ctr'` has exactly the same definition as `ctr` except for its formal parameters which appear as formal parameters of `ctrSmall`.

```
def ctrSmall(n) =
    def class ctr'() =
      val r      = Ref(n) -- n is the initial counter value

      def inc()  = r := r? + 1
      def dec()  = r := r? - 1
      def mag()  = r?

    stop # -- end of ctr' definition

  val c = ctr'()

  {.
   inc = c.inc
   mag = c.mag
  .}
```

Now, `ctrSmall` publishes an object exactly as `ctr` does except that the former has a restricted set of available methods.

4.6.5 Halting the Execution of a Class Instance

Creating an instance of a class also starts execution of the goal expression of that instance. This execution is autonomous, and it can not be interrupted or halted. We show how a class can be designed so that its caller can halt the execution of the instance in the future.

The essential idea is to introduce a method in the class, call it `terminate`, that can be called to halt the execution of an instance. The goal expression of the class is modified so that it halts if the `terminate` method has been called, in the style of interruption discussed in Section 2.5.2.6 (page 36). Introduce a semaphore that is initially 0 and set it to 1 when `terminate` is called. The goal expression of the class is modified such that it continually checks the semaphore value and halts its execution if the semaphore value is 1. Below, `ge` denotes the orginal goal expression.

```
  val s = Semaphore(0)

  def terminate() = s.release()

--  Goal expression ge is modified to

    (x >> stop) <x< (ge | s.acquire())
```

4.6.6 Translation of Class

We show the translation in two parts, the first part implements the publication requirement and the second, additionally, implements resilience.

4.6.6.1 Implementing Publication Requirement

A class is translated to a prime site. The body of the site is identical to that of the original class except that the goal expression of the site is different. The prime site's goal expression has two subexpressions that run concurrently: (1) `ge >> stop`, where `ge` is the goal expression of the class, to ensure that `ge` is executed but its publications are discarded, and (2) a subexpression that publishes a record with all the methods of the class and their bindings.

We illustrate the translation using the class `ctr`, defined previously and reproduced below for convenience. We write the goal expression as `ge` here to better explain how it will be translated; the goal expression of the original class `ctr` is `stop`.

```
  def class ctr(n) =
    val r      = Ref(n)

    def inc()  = r := r? + 1
```

```
    def dec()  = r := r? - 1
    def mag()  = r?

    {- Below, the goal expression, ge, is stop -}
  ge
```

The translation of `ctr` is as follows.

```
def ctr(n) =
  val r      = Ref(n)

  def inc()  = r := r? + 1
  def dec()  = r := r? - 1
  def mag()  = r?

{- Goal expression of ctr(n) -}

  ge >> stop -- ge is stop in this example

| {.
    inc = inc,
    dec = dec,
    mag = mag
  .}
```

An instance of `ctr`, say `c` in `val c = ctr(5)`, is just a record with key values `inc`, `dec` and `mag` that are bound to the closures of the similarly named sites. Thus, `c.inc` refers to the site for the specific instance `c`. Creating this closure needs the value of `r`, a free variable in that site. Computation of the value of a free variable, in general, may be time-consuming, or even be suspended waiting for other variables to be bound. The other component of the goal expression is `ge >> stop`, which starts executing immediately at the time of the site call, even before the class publishes its value.

4.6.6.2 Implementing Resilience Requirement

The given translation does not implement resilience. If the goal expression or a method call is expected to continue computation even when its caller is terminated, we have to enhance this translation. For example, if a program includes `val c = ctr(5)`, then the execution of `ctr` terminates as soon as it publishes its first value, the record containing the method names. Therefore, the execution of the original goal expression of the class is also terminated. This does not present a problem for the specific case of `ctr` whose goal expression is `stop`. But it will have disastrous consequences for `Broadcast`, say.

We make use of a site from the standard library, `Resilient`, that takes the closure of an internal site as its argument and returns an equivalent site whose execution continues even after its caller is terminated. We modify the previous translation as follows. Introduce a new definition `ctr'` that mimics `ctr(n)` of the previous translation. Next, apply `Resilient` to the methods of `ctr'` and its

goal expression so that they are not terminated when the caller is terminated. Then, embed `ctr'` in `ctr(n)`. Note that the goal expression of `ctr(n)` first executes `Resilient(ctr')` which publishes a closure whose execution publishes the desired instance of `ctr(n)`.

```
def ctr(n) =
  def ctr'() =
    val r      = Ref(n)

    def inc()  = r := r? + 1
    def dec()  = r := r? - 1
    def mag()  = r?

  {- Goal expression of ctr(n) -}

    ge >> stop -- ge is stop in this example

  | {.
      inc = Resilient(inc),
      dec = Resilient(dec),
      mag = Resilient(mag)
    .}

# -- End of the definition of ctr'

Resilient(ctr') >newctr> newctr()
```

Custom Classes The translation shown here can be used as a template to create classes that obey different rules, say about resilience or export control. For example, the user may define a prime site according to this template that effectively defines a class in which only some of the methods are resilient, or the class itself is non-resilient. Also, only a limited set of methods may be exported, or some variables defined by **`val`** could be exported by modifying the contents of the exported record.

4.7 Concluding Remarks

We have described the Orc language in some detail, with many examples, to highlight the integration of functional features with imperative, non-deterministic and time-dependent aspects. Though many of the features, such as data types, patterns and clausal definitions of functions are well-known, their integration in Orc requires special attention, particularly for concurrent execution. For every feature of Orc language, we have shown a translation to Orc calculus. The implementation of the language closely follows the translations given in this chapter, though there are several key optimizations.

Chapter 5

Programming Idioms

5.1 Introduction

We have seen a number of examples of the use of Orc combinators in the previous chapters. Since these combinators are quite different from any in the literature, we illustrate their use on a number of simple programming problems in this chapter. The programming patterns illustrated here are applicable in many areas of programming.

One of the novel features of Orc is the use of light-weight concurrency for program structuring. Typically, concurrency is introduced in a program to enable its execution on multiple processors. Or, it is used to model the entities in the problem domain, such as multiple servers or user programs, that are concurrently active. Concurrency is not typically used as a tool for program structuring where only a single processor is available and the problem admits of a sequential solution.

Orc's concurrency combinators can be used in the traditional ways. Additionally, we argue that concurrency is a structuring principle in its own right, even for problems that are typically solved by sequential programs. We illustrate such structuring on a variety of problems in this chapter.

5.2 Enumeration

> "Programs to solve combinatorial search problems may often be simply written by using multiple-valued functions. Such programs, although impossible to execute directly on conventional computers, may be converted in a mechanical way into conventional backtracking programs." – R.W. Floyd[18]

An enumeration problem usually asks for all (or some) values satisfying a predicate. Enumerations could be quite easy —enumerate all integers in a certain range— to intractable or even impossible —enumerate all programs

that halt. We consider a few enumeration problems in this section that admit relatively easy solutions.

A functional programming language like Haskell [15] is excellent at solving enumeration problems. Solutions can often be described recursively. Haskell also includes a "list comprehension" feature to specify a list of values that satisfy a list of constraints.

Orc is not specifically designed for solving combinatorial problems like enumeration. For complex enumerations, as for every other problem, it is best to use sites that excel at solving such problems. But a small amount of enumeration is essential in practical programming, and Orc can succinctly solve many simple enumeration problems. Even though Orc does not include list comprehension, its sequential combinator is effective in encoding a series of constraints.

The enumeration programs that we show in this section publish the values in arbitrary order. It is equally easy to publish a single list that includes all the enumerated values, provided the enumeration problem is finite. We show the solution to an infinite enumeration problem in this section, and ordered infinite enumeration in Section 8.2.2. Lazy execution and infinite list enumeration are discussed in Section 6.3 (page 165).

5.2.1 Partially Ordered Enumeration: Dovetail

The abstract version of *dovetail computation* is illustrated by the following problem: given a boolean matrix that is infinite in both dimensions, find a matrix element that is `true`, if one exits. Clearly, searching the matrix in order by rows or columns is ineffective, because such a search may never terminate along a row, say, while a matrix element in a different row may be `true`.

Before attacking this problem, we solve a very simple enumeration problem: enumerate all 2-tuples of positive integers that add up to a given value, n, and the first element of the tuple does not exceed the second element. The strategy is to enumerate all integers j, $1 \leq j \leq n/2$, as the first element and $n - j$ as the second element. Below, `enumtuple(n)` publishes all such tuples for integer n; for $n < 2$ the program halts.

```
def enumtuple(2) = (1,1)

def enumtuple(n) =
    upto(n/2) >i> i+1 >j> (j,n-j)
```

Note that `upto(n/2)` publishes all natural numbers below `n/2`; so, the expression `upto(n/2)>i> i+1` publishes all positive integers up to `n/2`, and including it if `n` is even.

Applying the enumeration for infinite matrix search We can apply the following strategy to solve the problem of searching an infinite matrix. Assume that indices in both dimensions are positive integers. For any integer n, $2 \leq n$, examine all matrix elements whose indices add up to n. Do this search in increasing order for all n. It is clear that the search will look at every matrix

element; element with index (p, q) will be examined when $n = p + q$. This is the essence of dovetail computation. Note that while there is a strict order over n there is no order over the tuples that add up to n, and we enumerate them in arbitrary order. Thus, the enumeration of indices is partially ordered.

We now have all the ingredients for a solution. Below, `ibm` represents the infinite boolean matrix; typically, `ibm` would be a site that has two positive integers as arguments and that publishes the value of the corresponding matrix entry. Site `dovetail(n)` publishes `true` if $ibm(p, q)$ is true for some (p, q), where $p+q \geq n$. It first examines all entries (p, q), where $p+q = n$, in arbitrary order. If some entry is `true`, the index of the entry is published and the execution terminates. If none of the entries is `true`, then `dovetail(n+1)` is called.

```
def dovetail(n) =
    enumtuple(n) >(p,q)> ibm(p,q) >b> Ift(b) >> (p,q)
  ; dovetail(n+1)
```

The computation never terminates if all matrix entries are `false`.

This strategy can be easily generalized to do a dovetail computation over a d dimensional infinite matrix, by enumerating d-tuples that sum to a specific value.

Computing 2-Taxicab numbers An integer is a n-taxicab number if it can be expressed as a sum two cubes of positive integers in at least n different ways. Often, the n-taxicab number is defined to be the smallest such number. Here, we sketch a strategy to compute all 2-taxicab numbers, i.e., numbers that can be written as $x^3 + y^3$ and $u^3 + v^3$ where $(x, y) \neq (u, v)$. The smallest such number[1] is 1729 which is $9^3 + 10^3$ and $1^3 + 12^3$.

The straightforward computation involves enumerating all positive integers and then determining for each if it is a 2-taxicab number. This is a very expensive computation procedure. Instead, we enumerate all numbers $p^3 + q^3$ by first enumerating (p, q) that sum to a specific value. That is: (1) starting at 2, enumerate integers n in increasing order, (2) for each n, enumerate (p, q) such that $p + q = n$ (3) for each (p, q), let $s = p^3 + q^3$, (4) store s in a database D if $s \notin D$, and publish s if $s \in D$ and s has not been published earlier.

5.2.2 Partitioning a number

A partition of a positive integer `n` is a multiset of positive integers whose sum is `n`. We represent a multiset by a list, and we take the sum of the items of an empty list to be 0. It is required to publish all partitions of any given `n`. For `n = 3`, the partitions are `[3], [2,1]` and `[1,1,1]`.

[1]This number has a famous history. G. H. Hardy relates a conversation with Srinivasa Ramanujan: "I remember once going to see him when he was lying ill at Putney. I had ridden in taxi-cab No. 1729, and remarked that the number seemed to be rather a dull one, and that I hoped it was not an unfavourable omen. "No", he replied, "it is a very interesting number; it is the smallest number expressible as the sum of two [positive] cubes in two different ways."

It is quite easy to write a site definition that first enumerates all integers `x` from 1 through `n`, for each such `x` enumerates all partitions of `n-x` recursively, and for each such partition `xs` reports `x:xs` as a partition of `n`. Unfortunately, this procedure publishes lists that are permutations of each other but that correspond to the same partition, so that both `[1,2]` and `[2,1]` are reported as partitions of 3.

To overcome this problem, we represent a multiset uniquely by a list whose items are non-increasing in value from left to right. The strategy is still the same as described above, but we need a more general site, `parts(n,p)`, that publishes all non-increasing lists corresponding to the partitions of `n` in which all elements are at most p, where $p \geq 0$. Then, `partition(n)` is simply `parts(n,n)`.

The definition of `parts` is given below. In the general case for positive `n`, `upto(p)>x> x+1` publishes all values in the range 1 to `p` and each such value is called `q` in the ensuing computation; for every `q`, `parts(n-q,q)` publishes every partition of `n-q` in which the parts do not exceed `q` and each such list is called `qs` in the ensuing computation; and for all such `q` and `qs`, `q:qs` is a publication of `parts(n,p)`. Observe the manner in which the sequential combinator is used to create a branching structure of computation. The definition of `parts` includes the possibility of n being zero or negative that may arise as a result of the recursive calls.

```
def parts(n,p) =
    Ift(n <: 0) >> stop
  | Ift(n  = 0) >> []
  | Ift(n :> 0) >>
     upto(p) >x> x+1 >q> parts(n-q,q) >qs> q:qs

def partition(n) = parts(n,n)
```

5.2.3 Permutations

The use of the sequential combinator in enumeration is highlighted, yet again, in the following example that enumerates all permutations of the first n positive integers, where $n \geq 0$. For $n = 0$, the only publication is an empty list. For positive n, all permutations of numbers 1 through n are published (in arbitrary order), each as a list. In all cases, $n!$ lists are published.

The enumeration strategy is as follows for positive n. Enumerate all permutations of the first $n - 1$ positive numbers. For each such permutation zs, insert n in all possible ways in zs, and publish the resulting n lists. Site call `insert(zs)` inserts n in all possible ways in zs and publishes the resulting lists in arbitrary order. All these lists are computed concurrently. The code for insert operation consists of two separate cases: (1) insert n as the head of zs, and (2) insert n as a non-head item, using the sequential combinator for insertion.

```
def perm(0) = []
def perm(n) =
```

```
    def insert([]) = [n]
    def insert(x:xs) = n:x:xs | insert(xs) >ys> x:ys

  perm(n-1) >zs> insert(zs)
```

5.2.4 Infinite Set Enumeration

Many infinite enumeration problems can be solved using the branching structure provided by the sequential combinator along with recursion. The following program publishes all binary strings including the string of length 0. A binary string is represented by a list of 0s and 1s. The published strings are in no particular order. It is not even true that the strings are published in order of their lengths.

Note that the definition of `bin` uses "unguarded recursion" in the second alternative where `bin` is called immediately after its invocation without any other intervening action. Concurrent computation in the first alternative ensures that its publication can be used to advance the computation in the second alternative.

```
def bin() =
    []
  | bin() >xs> (0:xs | 1:xs)

bin()
```

5.2.5 Divide and Conquer

Many enumeration problems employ "Divide and Conquer" where a divide procedure partitions the problem into several subproblems so that the solutions of the subproblems can be combined to obtain a solution of the original problem. Typically, the subproblems are similarly divided unless they can be solved directly and their solutions combined. The solution of the original problem specifies the division and combination procedures, and how to solve the smallest problem instances. Each of the subproblems can often be solved recursively and concurrently.

The following site enumerates all binary trees of `n` nodes. A tree is represented as follows: the empty tree (with 0 nodes) is given by the empty list, and a non-empty tree is given by a list of two items corresponding to the left subtree and the right subtree.

The enumeration strategy is an elementary application of divide and conquer: enumeration of a tree of n nodes, for positive n, amounts to enumerating all binary trees of size i, $0 \leq i < n$, as left subtree and of size $(n - i - 1)$ as right subtree (the remaining node is the root of the tree). Each sub-enumeration problem is solved recursively. The smallest problem corresponding to $n = 0$ is solved directly.

```
def enumTree(0) = []
def enumTree(n) =
  upto(n)         >i>
  enumTree(i)     >x>
  enumTree(n-i-1) >y>
  [x,y]
```

For example, `enumTree(3)` publishes:

```
[[], [[], [[], []]]]
[[[], []], [[], []]]
[[], [[[], []], []]]
[[[], [[], []]], []]
[[[[], []], []], []]
```

A major drawback of this style of enumeration is that the same subtree may be computed several times. For n exceeding 10, say, `enumTree(2)` will be executed many times for *both* left and right subtrees. The problem becomes acute for higher values of n. A better strategy is to enumerate the trees in order of increasing size using the trees already enumerated. This is a form of memoization, which we discuss in Section 7.4.

5.3 Controlling Execution Order

We consider various ways of ordering the executions of different expressions in a program. We cast the problem as a search problem, though the strategies for ordering are applicable more generally.

Let `f` and `g` be two computations that search different parts of a search space. Assume that each of these computations either publishes a single value, the result of a successful search, or halts (silently) on failure. It is required to publish any successful search result, or halt (silently) if neither search succeeds. The searches are non-interfering in that they do not modify any store that is accessed by both `f` and `g`. In many cases, `f` and `g` may themselves be recursively defined to further divide the search space.

We can identify at least 3 search strategies by executing `f` and `g` in different orders. In fact, these strategies are about the scheduling of computations `f` and `g`; they can be applied in any context, not just for searches.

1. Execute `f` and `g` concurrently: This is simply `f | g`. If only one search result is needed, we write the program as:

   ```
   val x = f | g
   x
   ```

2. Execute `f` and `g` sequentially: Here, `f ; g` solves the problem, because if `f` fails then it eventually halts and `g` is executed. Again, if only a single search result is needed we use:

   ```
   val x = f ; g
   x
   ```

3. Execute `f` and `g` concurrently, but with priority for `f`: Start both `f` and `g` concurrently, but publish a result from `f` if it succeeds, otherwise from `g`. Observe that `f ; g` does not solve the problem because `g` is not started until `f` fails.

```
(x ; y)
  <x< f
  <y< g
```

   This solution has one drawback. The computation of `g` is not terminated as soon as `f` publishes a value. We modify the program slightly so that if `f` publishes then `g` also publishes the same value, and hence its computation is then terminated.

```
(x ; y)
  <y< x | g
  <x< f
```

We rewrite this program more transparently using variable `z` that is bound to a value as soon as `f` publishes, or `f` halts and `g` publishes.

```
val z =
  val x = f
  val y = g
x ; y

z
```

Search strategies may make use of real-time. If `f` publishes a result within the first `t` time units, then publish this result discarding any publication of `g`; if `f` does not publish within `t`, then publish result from `f` or `g` whichever is computed first. We have seen this form of priority in Section 2.5.2.3; in the current setting, this amounts to:

```
val x = f | (Rwait(t),g) >(_,v)> v
x
```

Round-robin execution of `f` and `g` based on time-out is yet another scheduling strategy. There are many elaborate search strategies that depend on the specifics of the problem. We show a few below for the example of *subset sum*.

Subset Sum We illustrate execution ordering on a particularly simple problem. Given an integer `n` and a multiset of integers as a list `xs` it is required to determine if there is a sublist of `xs` that adds up to `n`; further, publish one or all such sublists.

This problem is typically solved using backtracking, which is appropriate when we are required to express the solution using a single thread of computation. We present a number of solutions, based on the strategies we have just discussed, which are either concurrent, sequential or a mixture of the two. In this example, concurrency does not involve access to shared data, but merely managing a number of computation threads.

5.3.1 Subset Sum in Parallel

We define `parsum(n,xs)`, where `n` is an integer and `xs` a list of integers, that publishes all sublists of `xs` that sum to `n`; there may be no such sublist and then `parsum(n,xs)` is silent.

It is easy to understand the first two clauses of the program, below. If $n = 0$ and the given list is empty, then the unique solution is the empty list. If $n \neq 0$ and the given list is empty then there is no solution. The crux of the design is in the last clause. Consider two mutually exclusive cases for a solution sublist: (1) the first element of the list, `x`, is included in the solution, and (2) `x` is not included in the solution. Since these two cases are exclusive, enumeration for the two cases may proceed concurrently without interference.

```
def parsum(0,[])    = []
def parsum(n,[])    = stop
def parsum(n, x:xs) =
  parsum(n-x,xs) >ys> x:ys | parsum(n,xs)
```

A call to `parsum(-5,[-2,4,1,-3,8,-7])` publishes

```
[-2, 4, -7]
[4, 1, -3, -7]
[-2, -3]
```

whereas `parsum(6,[2,5,1])` is silent.

This program is easily modified, using the otherwise combinator, to publish a value in all cases indicating the presence or absence of solutions.

5.3.2 Any solution in Subset Sum

We modify the program given above so that it publishes just one solution if one exists and is otherwise silent. The solution uses the pruning combinator.

```
def anysum(0,[])    = []
def anysum(n,[])    = stop
def anysum(n, x:xs) = zs
    <zs< (anysum(n-x,xs) >ys> x:ys | anysum(n,xs) )
```

5.3.3 Subset Sum using Backtracking

We show a backtracking solution that publishes the *first* solution. We take "first" to mean the smallest in a total order of the sublists, which we define next.

Elements of `xs` have natural numbers as indices, starting at 0 for the head element. The *index* of a sublist is the list of indices of its elements. Define a sublist to be smaller than another if its index is lexicographically smaller.

Site `seqsum(n,xs)`, where `n` is an integer and `xs` a list of integers, publishes the smallest sublist of `xs` that sums to `n`. As before, there may be no such sublist and then `seqsum(n,xs)` is silent. The code is identical to that for

`parsum` except that the parallel combinator in the last clause is replaced by the otherwise combinator. Thus, all sublists in which the first element `x` of the input list is included are searched before any in which `x` is excluded, and this strategy is applied recursively for all elements of the input list.

```
def seqsum(0,[])    = []
def seqsum(n,[])    = stop
def seqsum(n, x:xs) =
  x:seqsum(n-x,xs) ; seqsum(n,xs)
```

A call to `seqsum(-5,[-2,4,1,-3,8,-7])` publishes `[-2, 4, -7]`.

5.3.4 Subset Sum using Concurrency and Backtracking

Next, we implement strategy (3), discussed earlier, in which the searches are run concurrently but the lexicographically smallest sublist that meets the search criterion is published. The program is a slight modification of `seqsum`.

```
def parseqsum(0,[])        = []
    def parseqsum(n,[])    = stop
    def parseqsum(n, x:xs) =
      val z =
        val p = x:parseqsum(n-x,xs)
        val q = parseqsum(n,xs)
      (p ; q)
    z
```

A call to `parseqsum(-5,[-2,4,1,-3,8,-7])` publishes `[-2, 4, -7]`, as expected. Observe that both searches in the last clause may fail so that neither `p` nor `q` is assigned a value. Then `p ; q` halts.

5.3.5 Subset Sum using Umbrella Search

The next program combines the strategies of `parsum` and `seqsum` so that the search tasks are forked out concurrently for a few specified levels and then each search proceeds sequentially, a strategy we call *umbrella search*[2] . Umbrella search may be based on real-time rather on the number of levels, as we show in Section 5.3.6 (page 132). Below, `umbrellasum` has 3 arguments; its first argument is the number of levels for which concurrent search tasks are forked, and the other two arguments, as before, are `n` and `xs`. The search publishes at most one result.

```
def seqsum(0,[])          = []
def seqsum(n,[])          = stop
def seqsum(n, x:xs)       =
  x:seqsum(n-x,xs) ; seqsum(n,xs)

def umbrellasum(_,0,[])   = []
def umbrellasum(_,n,[])   = stop
```

[2]The name "Umbrella Search" is due to David Kitchin.

```
  def umbrellasum(0,n, xs)  = seqsum(n, xs)
  def umbrellasum(nl,n, xs) =
    val z =
      x:umbrellasum(nl-1, n-x,xs) | umbrellasum(nl-1, n,xs)
    z
```

A call to `umbrellasum(4,-5,[-2,4,1,-3,8,-7])` publishes `[-2, 4, -7]`, as expected.

5.3.6 Subset Sum using Timed Umbrella Search

The next program employs umbrella search in which the transition between parallel and sequential searches is triggered by the passage of time rather than a pre-determined number of levels. Here, parallel searches run for the first `t` time units, and then sequential searches. Below, `timedsum(t,n, xs)` has `t` as the time parameter, and `n` and `xs` have their usual meanings. It includes definition of site `search'` that works exactly like the `umbrellasum` of Section 5.3.5 except that the test for levels in `umbrellasum` is replaced by a test that determines if `t` units have elapsed since the site invocation. A mutable variable `b` is initially set `true` and set `false` after the passage of `t` units. The goal expression of `timedsum` calls `search'` and concurrently calls `Rwait(t)` to set `b` appropriately.

```
  def timedsum(t,n, zs)  =
    val b = Ref(true)

    def search'(0,[])    = []
    def search'(n,[])    = stop
    def search'(n, x:xs) =

      if b? then
            (val z  = search'(n-x,xs) >ys> x:ys | search'(n,xs)
             z)
            else (x:search'(n-x,xs) ; search'(n,xs) )

     search'(n, zs) | Rwait(t) >> b := false >> stop

--Test input; run in parallel for 1 msec.

    timedsum(1,-5,[-2,4,1,-3,8,-7,6,-1,3,-8])

-- output
[3, -8]
```

5.3.7 Recursive Descent Parsing, 2-function calculator

Recursive descent parsing of a string according to a given grammar illustrates searching over a parse tree. We illustrate the procedure for computing the value of an arithmetic expression with just plus and times as the only operators. The grammar for such expressions is given below.

A grammar (in BNF) consists of a set of productions where each production defines a non-terminal. A production consists of one or more alternatives. Each alternative is a string of terminals and non-terminals. We liken the alternatives to the parallel combinator and concatenation of symbols to the sequential combinator. Accordingly, we translate a grammar to an Orc program in an almost-mechanical fashion.

```
expr    ::= term   | term + expr
term    ::= factor | factor * term
factor  ::= number | (expr)
number  ::= digit  | digit number
digit   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Here, an `expr` is a properly formed arithmetic expression, written as a string of symbols consisting of unsigned integers as the operands and `+` and `*` as the operators. Thus, `((31*5))+(03+3)` is a proper `expr`, whereas `3**5` is not. Observe that the grammar is not left-recursive, i.e., in a production defining some non-terminal, the non-terminal does not appear as the leftmost string in any alternative of that production.

Normally, the user would provide the input as a string. The standard site `characters` converts its argument string to a list of symbols (a symbol is a string of length 1). So, `characters("3+4")` publishes `["3","+","4"]`. Henceforth, we assume that input is available as a list of symbols. Further, we assume that the list contains no white space. All white spaces in the beginning of the list and around the operators `+` and `*` may be removed using a lexical scanner site.

To parse an expression we introduce a site for each non-terminal. Each site takes a list of symbols `xs` as argument. For every prefix of `xs` that is an instance of the given non-terminal, the site publishes a pair `(n,ys)` where `n` is the value of the non-terminal derived from the prefix and `ys` is the remaining suffix of `xs`. Thus, given `["3","+","4"]` as input to site `expr`, the two prefixes that are `expr` are `["3"]` and `["3","+","4"]`, and the corresponding publications are `(3,["+","4"])` and `(7,[])`. If no prefix is an instance of the given non-terminal, then there is no publication and the call halts silently.

To see how a production in the grammar is turned into a site definition, consider the production

```
expr ::= term | term + expr
```

The following definition of site `expr` consists of two parallel branches, one for each alternative. The first one corresponds to the alternative `term`; it publishes whatever `term(xs)` publishes. The second one corresponding to `term + expr` first applies `term(xs)` to obtain pairs `(n,ys)`, and for each such pair determines if `ys` is of the form `"+":zs`, for some `zs`. If that is the case, it calls `expr(zs)` which publishes pairs `(m,ws)`. Then, `xs` has a prefix that is an `expr` whose value is `n+m` and the remaining suffix is `ws`; therefore, this alternative publishes `(n+m,ws)`. The computation is carried out for all possible prefixes matching this pattern because of the branching structure of the computation induced by the sequential combinator. Also, note the use of pattern matching to dissect the input string.

```
def expr(xs)   =   term(xs)
                 | term(xs) >(n,ys)>
                   ys >"+":zs> expr(zs) >(m,ws)> (n+m,ws)
```

We make one small optimization for the final program. There is no need to compute `term(xs)` in both alternatives. This computation may be done just once and reused. This amounts to applying the right distributivity of `|` over `>x>` (see Section 3.7.3). Another way to view this transformation is as a rewrite of the production:

```
expr ::= term (ϵ | + expr)
```

The resulting program is

```
def expr(xs) =
     term(xs) >(n,ys)>
     ( (n,ys) | ys >"+":zs> expr(zs) >(m,ws)> (n+m,ws) )
```

We apply this transformation also to the production for `term`. The production for `factor` is translated directly. The production for `number` needs a different treatment. It is not possible to compute the value of a number from its first digit and the value of the remaining suffix; 10 and 100 will compute to the same value from the first digit 1 and the suffix whose value is 0. So, we introduce an auxiliary site `number'` that has two arguments, a number `i` and a list `xs`. It computes the value of the number whose prefix is number `i` and whose suffix is the longest possible prefix of `xs`; it publishes the value of the number and the remaining suffix of `xs`. Thus, `number'(35,["0","1","+"])` publishes `(3501,["+"])` and `number'(35,["+"])` publishes `(35,["+"])`. Site `number` calls `number'` only if the next symbol in its input is a digit. The definition of `digit` follows the grammar directly.

The program is shown in Figure 5.1. A given list `xs` is a valid expression only if `expr(xs)` publishes `(n,[])`, for some `n`. The value of the expression is `n` in that case.

Recursive descent is a simple parsing technique. It is not particularly efficient. The purpose of the example is to demonstrate that divide and conquer applied to the parsing problem leads to a simple solution in Orc that mirrors the structure of the grammar.

An Aside, 4-function Calculator We complete the design of the calculator to include the other two arithmetic operators, subtraction and division. The extensions do not show any new Orc programming technique; they merely complete a job (literally) half done. There is, however, an interesting question about subtraction that we solve here.

First, the extension for division is straightforward. The grammar is extended by adding a suitable alternative.

```
term ::= factor (ϵ | * term | / term)
```

So, the definition of site `term` is similarly extended by adding an alternative. Note that the added alternative uses `((n+0.0)/m,ws)` instead of `(n/m,ws)`, because the latter truncates the result to an integer value if the operands are integer, whereas the former carries out the division in floating point mode.

```
def term(xs)   =
     factor(xs) >(n,ys)>
     ( (n,ys)
      | ys >"*":zs> term(zs) >(m,ws)> (n*m,ws)
      | ys >"/":zs> term(zs) >(m,ws)> ((n+0.0)/m,ws)
     )
```

Next, consider an extended grammar that includes subtraction. The obvious production is:

```
expr ::= term (ϵ | + expr | - expr)
```

This grammar is perfectly adequate for syntax checking, but inadequate for computing the value of an expression from the values of its components. Given an expression of the form $t_0 - t_1 + t_2$, where each t_i is a term, the expression will be parsed as an initial term t_0 and a remaining expression $t_1 + t_2$. The value of the original expression can not be computed from those of t_0 and $t_1 + t_2$. That is, the syntax does not reflect the semantics. This problem can be overcome by defining the grammar for `expr` differently.

```
expr ::= expr (ϵ | + term | - term)
```

However, the grammar is now left-recursive, and the direct translation we have been employing so far is inapplicable.

We overcome the problem by introducing a new non-terminal, `expr'`, which is an expression whose initial term is known to be negated. Thus, the parse of $t_0 - t_1 + t_2$ will identify t_0 as a term and $t_1 + t_2$ as `expr'`.

```
expr  ::= term (ϵ | + expr | - expr')
expr' ::= term (ϵ | + expr | - expr')
```

Observe that `expr` and `expr'` generate the same set of strings (a fact that can be proved by induction on the number of steps in a derivation). Yet, the latter reflects the semantics of the expression. The value of `expr'` is computed by subtracting the value of its first term from the value of the remaining expression. The complete program is shown in Figure 5.2.

Using Mutable storage in Parsing The parsing strategies defined so far represent the string to be parsed by a list of symbols. Each site calls a site by sending it a list. Since the lists are read-only, we can avoid the overhead of parameter passing by storing the input list in an array, and simply passing two indices to designate a sublist in the array. See discussion of mutable store in Chapter 7.

5.3.8 Angelic and Demonic Non-determinism

Computer science has two prevalent ways of dealing with the "fork in the road" dilemma: take both roads or take either one arbitrarily. The first strategy is usually called *angelic* non-determinism, and the second *demonic*.

Angelic non-determinism is often used in the theory of non-deterministic automata where an input string is accepted if *any* possible execution of the automaton can lead to an accepting state. It is suitable for solving search problems where all states in a search space are to be explored, and the search space is described by successive transitions from an initial state to other states. An implementation on a uniprocessor typically uses backtracking whereas a multi-processor may explore a (bounded) number of paths concurrently. Angelic non-determinism is typically introduced by a programmer to simplify the description and structure of a program. Therefore, it is sometimes known as internal non-determinism.

Demonic non-determinism is typically externally imposed, though it may also be introduced by a programmer to simplify the description of a program or speed-up its execution. A program that is waiting to receive data from two external sources contends with demonic non-determinism; it can not predict the source from which it will next receive its data, so the program has to account for either possibility in its subsequent execution. Demonic non-determinism is also employed internally (i.e., deliberately) in randomized algorithms to either simplify the program structure or improve its performance, see Lehmann and Rabin [32] for an example (that is also described in Section 9.1.4, page 243).

Language features for non-determinism typically support demonic, leaving the programmer to explicitly implement angelic non-determinism where needed. To the best of our knowledge, no existing programming language (other than Orc) permits combination of both forms of non-determinism in a single program.

Orc includes combinators for both angelic and demonic non-determinism. The execution of `f | g` starts concurrent executions of both `f` and `g`, and, more generally, `f >x> g` explores execution of `g` for all possible publications of `f`. These combinators can be used to implement angelic non-determinism. Demonic non-determinism is implemented by the pruning combinator where in `f <x< g`, execution of `f` must contend with any publication of `g`. Perhaps the simplest Orc program to illustrate demonic behavior is one that chooses to execute either `f` or `g` arbitrarily:

```
if (false | true) then f else g
```

which is translated to

```
(Ift(b) >> f | Iff(b) >> g)
  <b< (false | true)
```

The demonic choice in the first line is illusory, because only one of the alternatives will be executed determined by the value of `b` The demonic choice in the second line is real.

We have already seen a number of examples of the use of both combinators. We show a small example next that highlights the intermingling of both forms

of non-determinism using the Orc combinators. Here, angelic non-determinism allows us to explore all possible paths in a search space, and demonic permits a more efficient evaluation strategy when only some of the paths need to be explored.

Exploring a Game Tree We design a program to explore a finite game tree to compute whether a player has a winning position. We consider a two person game with players A and B. The players move alternately. For a position p in the game the player to move (the *mover* at p) loses if he has no available move. Assume that the moves can not go on forever. Therefore, the moves and the positions can be depicted by a finite game tree, and each position is a winning position for one of the players.

Suppose A is the mover at position p. A is a winner, i.e., has a winning position, if there is *some* move at p to a position q such that q is a losing position for B. And, A has a losing position at p if there is no such move, that is, for *all* moves at p player B wins at all subsequent positions. The proposition defining a winning position can be written as: there exists a move at p to some position q such that for all moves from q to positions r there is a move from each r such that ... The proposition is an alternation of existential and universal quantifiers over moves. It combines alternate demonic and angelic non-determinism.

Below, site `wins` computes if the mover at a given position is a winner. A call to `wins(p)`, where `p` is a position, publishes `true` if the mover at `p` is a winner and `false` otherwise. The site call `moves(p)` publishes all positions reachable by making a single move at position `p`; if there is no available move at `p` then `moves(p)` is silent. Site `moves` encodes the rules for legal moves; it is defined elsewhere.

```
def wins(p) =
  b <b< (
          (moves(p) >q> Iff(wins(q)) >> true)
          ; false
        )
```

All moves from `p` are explored using the sequential combinator. If any move leads to a losing position for the opponent, further explorations are abandoned and `true` is published by a demonic choice. And, `false` is published only if no move shows a losing position for the opponent. The recursive computation ensures alternate applications of angelic and demonic non-determinism. The program is easily modified to compute a winning strategy from a given position, i.e., a sequence of moves for the winning player corresponding to any set of moves by the opponent. The winning strategy is chosen non-deterministically when there are several possible such strategies.

5.4 Programming with Closure

We introduced *closure* in Section 2.4 (page 27). It is a value that encodes a site with a subset (possibly none) of its parameters. We show a number of examples

of the use of closure in this section.

In some cases, a class definition can be replaced by just a site definition. In case the class has just one method the class can be coded as a site that publishes a closure corresponding to the method. This is often a major simplification. (A class that has no method can also be written as a simple **def**. Such classes are used for doing background computations without publishing, such as monitoring of the computation, collecting statistics and garbage collection.)

5.4.1 Information Hiding

Consider a small example in which several concurrent threads increase a shared variable, `total`. Each thread calls `score(x)` to increase `total` by `x` and receives the current value of `total` on completion of the site call. We employ a semaphore, `sem`, to allow at most one caller to have access to `total` at any time.

```
val total = Ref(0)
val sem = Semaphore(1)

def score(x) =
  sem.acquire()        >>
  total := total? + x  >>
  total?               >v>
  sem.release()        >>
  v
```

There are two major objections to this solution. First, variables `total` and `sem` are exposed to computations outside `score` where they may be inadvertently modified. Second, the same program can not be used to run several independent instances of `score` with different associated `total`s; we have to rewrite this program for each instance with different names for the mutable variables. This problem becomes acute if the number of instances is not known a-priori but instances are created dynamically.

We can solve both problems using closure. Enclose the given program within another site definition that publishes site `score` when it is called.

```
def cumulative() =
  val total = Ref(0)
  val sem = Semaphore(1)

  def score(x) =
    sem.acquire()        >>
    total := total? + x  >>
    total?               >v>
    sem.release()        >>
    v

score
```

Now, variables `total` and `sem` are local to `cumulative()`. Each call to `cumulative()` returns a closure that can be used by the concurrent threads exactly as they used site `score` earlier. Further, different calls to `cumulative()` returns different instances of `score`. A typical usage may be

```
val score1 = cumulative()
val score2 = cumulative()

score1(2) | score1(3) | score2(5) | score2(7)
```

5.4.2 Performance Profiling

Consider a site that has a single argument, and that publishes a single value for any argument and then halts. We create a site that accepts any such site `f` as argument and publishes a site `f'` that mimics `f`, by publishing exactly the same value that `f` publishes for any argument, and additionally, the amount of time consumed for `f`'s computation on any argument.

The implementation makes use of the factory site `Stopwatch` that is described in Section 10.2. A `Stopwatch` instance has methods `start()` and `pause()` where the former starts the stopwatch and the latter pauses it and publishes the elapsed time at that moment.

```
def profile(f) =
  val sw = Stopwatch()

  def f'(x) = sw.start() >> f(x) >v> (sw.pause(),v)

f'
```

To compute the running time of `f` on `x`, call `profile(f)(x)`. Calling `profile(Rwait)(100)` publishes `(100,signal)`, as we would expect.

A slightly different program for computing the running time is shown in Section 10.1.4 (page 251).

5.4.3 Access rights management

The problem of access rights management is best explained by a small example. The `copy` site, shown below, continually reads an item from an input channel channel `in` and writes it to channel `out`.

```
def copy(in,out) = in.get() >x> out.put(x) >> copy(in,out)
```

Now, `copy` needs to apply only the `get` method to `in` and `put` method to `out`, yet it has access to all available methods of both channels. This violates good programming practice. Henceforth, whenever it is convenient, we restrict the access rights of a site: a site is given access to only those methods of its argument sites that are essential for its execution. Using this principle, we rewrite `copy` as

```
def copy(read,write) =
    read() >x> write(x) >> copy(read,write)
```

where a typical call to `copy` is of the form `copy(in.get,out.put)`. Thus, a site is passed closures, such as `read` and `write`, that include only a subset of the methods of the associated object.

More generally, let `b` be an object with methods `p`, `q` and `r`. To restrict access on `b` to methods `p` and `q` only, define

```
val bpq = {. p = b.p, q = b.q .}
```

Now, `bpq.p` and `bpq.q` act as `b.p` and `b.q` whereas `bpq.r` is unavailable. A site call may pass `bpq` as an argument, instead of `b`, in order to restrict access of the called site to only these two methods of `b`.

5.4.4 Session id Management

Many client-server interactions are structured as *sessions*, where a session includes several calls by the client and corresponding responses by the server. The server typically identifies the client and the session using a key, called a *session id*. The client supplies the session id with each call. In high security applications the session id may change with each call during a single session. The client starts a session with a key of its choice and the server publishes for each call the result of its computation and the key to be used for the next call.

We use the generic term *ticket* for such a key, whether it changes or not, that the client supplies with each call. We show how to interpose a site between the client and the server so that the ticket is managed transparently without the client's involvement.

Below, site `ticket_wrapper` is called with the identity of the server `S`, a closure, as its argument. It publishes a closure `S'` that the client calls instead of `S`. The closure stores, and updates, the ticket value in a mutable variable. In the following implementation a random 30 bit number is used as the initial ticket.

```
def ticket_wrapper(server)

  val id = Ref(Random(2**30))

  def newserver(x) = server(x,id?) >(v,t)> id := t >> v

newserver

-- Usage with a specific server S

val S' = ticket_wrapper(S)

S'(arg)  -- Successive calls to S'
```

Multiple callers may call `ticket_wrapper` with the same or different server as argument. The calls may even be concurrent. Different callers will carry

different streams of tickets and engage in separate dialogs. These calls do not interfere because `ticket_wrapper` does not read or write any external variable.

A ticket is a useful programming paradigm even in non-security applications. A server can identify a client by its ticket and tailor its response accordingly. For example, consider a server that manages a long sequence of data, say a genetic sequence or a video file. Each client receives the entire sequence, one item with each call. The server may maintain a database of all the clients and the portion of the sequence they have seen so far so that it can respond appropriately to their next calls. Or, more simply, it hands out a ticket when it responds to a call, in addition to the item from the sequence; the ticket is the index to the item in the sequence the client should receive next.

5.4.5 Task Scheduling

A particularly simple view of task scheduling is that there is a fixed set of tasks each of which is a site. Execution of a task either ends with the publication of a signal or the task is never started, so it halts. There is a given policy according to which the tasks are to be executed. The effect of the execution of a task is irrelevant to the scheduler.

Below, the task scheduler is called with a list of tasks, each a closure without argument, and `policy`, a closure without argument. A call to `policy` publishes an index to a task in the list. The corresponding task is executed by calling the associated closure. The steps are repeated after the task execution terminates. The implementation below first transfers the tasks to an array so that random access to a task using its index is efficient.

```
def schedule(fs,policy) =
  val tasknum   = length(fs)
  val taskarray = Array(tasknum)

  {- function populate() transfers from fs to taskarray -}

  def populate(_,[]) = signal
  def populate(i,g:gs) = taskarray(i) := g >> populate(i+1,gs
      )

  def exec() = policy() >j> taskarray(j)?() >> stop ; exec()

  {- Goal of Scheduler -}

populate(0,fs) >> exec()
```

This form of task scheduling constitutes the essence of the UNITY [6] programming theory where tasks are called *actions*. Given below is a short program written in the UNITY notation. The program includes two variables, `x` and `y`, and two actions, written as guarded commands separated by `||` that modify these variables.

```
x,y = 0,0
```

```
   x < y --> x:= x + 1
||           y:= y + 1
```

Below, the program is translated to Orc where the variables are `Refs` and each action is a site. The policy is to pick one of the actions randomly for execution. If the guard of the chosen action is not true, then the action execution has no effect. The Orc program prints a line identifying the action and the resulting values of the variables.

```
val (x,y) = (Ref(0),Ref(0))

def f1() =
  Ift(x? <: y?) >>  x := x? + 1 >> Println((1,x?,y?))

def f2() =
  y := y? + 1 >> Println((2,x?,y?))

def policy() = Random(2)

schedule([f1,f2], policy)
```

The more general task scheduling problems in which the tasks are created and/or deleted during an execution can also be cast in this form.

5.4.6 Currying

Currying is a term used in functional programming for partial evaluation of a function of multiple arguments. The function is evaluated at some (or all) of its arguments and a closure published (in case the function is evaluated at all its arguments the resulting value may be regarded as a closure, a function of 0 arity). A curried site in Orc publishes a closure, as shown in the following example.

A call to the following uncurried site, `div(d,i)` where `d` and `i` are integers, publishes `true` if `d` divides `i` and `false` otherwise:

```
def div(d,i)= (i%d = 0).
```

A curried version of `div` is shown below. It has just one argument `d`, and it publishes a site (a closure) that behaves as `div(d,i)` given argument `i`.

```
def divides(d)= lambda(i)= (i%d = 0).
```

The call protocols for these examples are: `div(3,15)` and `divides(3)(15)`.

We may curry any number of arguments and nest the curried definition as in the following examples.

```
def add() = lambda(x,y,z) = x+y+z
```

```
def add'() = lambda(x) =
               lambda(y) =
                 lambda(z) = x+y+z

-- Usage
add()(2,3,5)
add'()(2)(3)(5)
```

Currying is effectively applied within a definition where a site is called multiple times with different arguments where the values of some of the arguments are fixed. Consider the following small example where site `divsome` has the specification: `divsome(d)(i)`, where `d` and `i` are integers, publishes `true` if `i` is divisible by some integer k, $2 \leq k \leq d$, and `false` otherwise. We show two different curried definitions of `divsome`. The first one is explicitly concurrent and the second one achieves concurrency through a recursive definition.

```
def divsome(d)  =
  lambda(i) =
    b <b<
    (upto(d-1) >k> Ift(divides(k+2)(i)) >> true ; false)
```

and,

```
def divsome(1) = (lambda(i) = false)
def divsome(d) =
  lambda(i) = divides(d)(i) || divsome(d-1)(i)
```

Site `divsome` may be used in a very inefficient to test to determine if an integer greater than 1 is composite:

```
def composite(j) = divsome(j-1)(j)
```

5.4.7 Multidimensional Structures

Factory site `Array` creates a 1-dimensional array whose lower index is 0. We show how to create arrays with specified indices (not just starting at 0) and multidimensional arrays. Closures play a crucial role in the construction.

It is easy to create an array with arbitrary upper and lower indices. Below, `Array'(m,n)` creates an array with lower bound `m` and upper bound `n` for the index; so, the array has $n - m + 1$ elements.

```
def Array'(lo,hi) =
  val ar = Array(hi-lo+1)

  def access(i) = ar(i-lo)

access
```

The site publishes a closure that allows access to a specific item. A typical usage is

```
val a = Array'(-3,2)

a(-3) := 0
```

Site `matrix`, below, defines a matrix with specified lower and upper bounds for each dimension. The elements of the matrix are stored in an array and site call `access(i,j)` retrieves the appropriate element from the array. The published value of `matrix` is a closure, `access`, that takes two indices and retrieves the corresponding matrix element.

```
def matrix2((lo1,hi1),(lo2,hi2)) =
  val mat =  Array((hi1-lo1+1)*(hi2-lo2+1))

  def access(i,j) =  mat((i-lo1)*(hi2-lo2+1)+j)

access

-- Typical usage

matrix2((-2,0),(-1,3)) >a>
a(-1,2) := 5 >> a(-1,2)?
```

Beyond 2-dimensional Matrix We extend the template for 2-dimensional matrix to construct higher dimensional matrices in a uniform manner, as shown in Figure 5.3 (page 152). We define a single site, `matrix`, that accommodates any number of dimensions. We explain its construction below.

The input parameter for `matrix` is a list of bounds where a bound is a pair of integers, `lo` and `hi` with $lo \leq hi$, that describes the possible values of the index for that dimension. An instance of a matrix, say

```
val m3 = matrix([(-2,0),(-1,3),(1,3)])
```

may access its individual items using the accepted matrix notation, as in `m3(-1,2,1)`. This last requirement will turn out to be the most difficult one to satisfy.

First, given a list of bounds `bounds` compute the size of the matrix and allocate storage for it in an array of the same size. This is the right place to check if the bounds are appropriate, that the lower bound for a dimension does not exceed its upper bound. In case it does, the call halts. Additionally, the site could print an error message, which we do not implement here.

```
{- Argument of size is a list of bounds.
   size Publishes the number of matrix elements.
-}
def size([]) = 1
def size((lo,hi):bs) = Ift(lo <= hi) >> (hi-lo+1) * size(bs)

val ar = Array(size(bounds))
```

Next, given a *list* of indices compute the location of the matrix element. Site call `item(is)`, where `is` is a list of indices, publishes the corresponding matrix element. An auxiliary site `index` computes the linear index of the element from the list index `is`. Site `index` checks to ensure that each index falls within the bounds for that dimension and halts if it does not.

```
def item(is) =

{- index(acc,xs,is) has
    acc: accumulator for index computation
    xs : list of bounds
    is : list of indices
   Compute acc+j where j is the linear index
   of the element at is.
-}

  def index(acc,[],[]) = acc

  def index(acc, (lo,hi):xs, i:is) =
      Ift(lo <= i && i <= hi) >>
      index(acc*(hi-lo+1)+(i-lo), xs, is)

ar(index(0,bounds,is))
```

Site `item` can be used to access the matrix items. However, `item` takes a list of indices as argument whereas we would like to access the elements using their normal matrix notation; that is, we would like to write `m3(-1,2,1)` rather than `m3([-1,2,1])`. One possible approach would be define another site `access` and have `matrix` publish the closure `access`, as shown below. There is a **`def`** for each dimension that the user may possibly need.

```
def access()      = item([])
def access(i)     = item([i])
def access(i,j)   = item([i,j])
  .
  .
```

Current implementation of Orc supports only fixed arity for sites, so this definition of `access` is illegal. So, we modify the solution as follows. Let `access` take a list as input and publish a closure corresponding to the length of the list, as in

```
def access([_,_]) = lambda(i,j) = item([i,j])
```

The call `matrix(bounds)` publishes `access(bounds)`, a closure that accepts as many indices as the length of the list as argument and publishes the corresponding matrix element. The complete program appears in Figure 5.3 (page 152). The program allows for up to 3-dimensional matrices; extension for any specific dimension involves defining `access` for that dimension using the given template.

Shared data structure In many applications with two dimensional and higher dimensional matrices it is required to partition a matrix into submatrices. We extend the definition of matrix to allow such partitioning. In fact, we allow any submatrix of a given matrix to be named and treated like a matrix on its own. Different submatrices may overlap so that modification of the value of an element in one submatrix may be reflected in another, as in a shared data structure.

This extension requires a change in the interface and the manipulations of closures in the implementation. First, a matrix is defined as a class, simply called `matrix'`, that includes a single method `submatrix`. For instance, execution of

```
val mat = matrix'([(-3,2),(-5,-3),(2,10)])
```

results in `mat` being bound to a matrix object with the given bounds. It is no longer possible to access the elements of `mat` simply by writing `mat(0,-1,2)`, say. Instead,

```
val lu  = mat.submatrix([(-1,1),(-1,-1),(2,3)])
```

for example, binds `lu` to a submatrix of `mat` as specified by the bounds. Now the elements of `lu` can be accessed, as in `lu(0,-1,2)`. The entire matrix can be accessed by creating a submatrix with the original dimensions.

```
val whole_mat = matrix'([(-3,2),(-5,-3),(2,10)])
```

The implementation of `matrix'` is nearly the same as of `matrix`. In addition to modifying the interfaces, as described above, the goal expression of site `item` is modified to use the dimensions of the specified submatrix, not that of the original matrix (as was the case in the definition of `matrix`). The entire program is shown in Figure 5.4 (page 153).

A worthwhile extension is to allow each submatrix to specify the coordinate system to address its elements. This coordinate system may be different from that of the original matrix. For example, `lu` may designate that its "leftmost" corner element, i.e., the one with index `(-1,-1,2)` in the original matrix, be accessed using index `(1,0,0)`, say. It is sufficient to specify the coordinate transform for just one element, as above, and then the coordinates of all other elements can be determined. We have not shown this extension here; it is easy to modify `matrix'` to accomodate ths extension.

Multidimensional Table We extend `Table`, which is 1-dimensional, to multiple dimensions. The definition `multiTable(ld,fun)`, shown in Figure 5.5, has the dimensions as a list in `ld` as for a multidimensional matrix; additionally, argument `fun` is a site of the same arity as the length of list `ld`. The program instantiates a matrix of the appropriate size, computes `fun` for each matrix location and stores it there. It publishes a closure that allows access to the contents of each matrix location as in a `Table`.

The goal expression enumerates all indices by calling `enum` and populates the matrix by calling `pop` which invokes `fun`. The program, shown in Figure 5.5,

allows for up to 3-dimensions; extension for any specific dimension involves defining `pop` and `acc` for that dimension following the given template.

A mesh of processes can be nicely described using a 2-dimensional table. A mesh is a 2-dimensional grid where a process is located at each grid point and is connected by channels to its four immediate neighbors in the horizontal and vertical directions. The processes at the mesh boundary have fewer neighbors. A mesh structure is often used in solving heat transfer problems using finite difference approximation of Laplace equations. We develop the skeleton of such a solution, below.

Arguments of site `laplace`, below, are the dimensions of the grid and other parameters, not shown, that are specific to the problem. Instantiate the incoming channels to the processes as `in` and the outgoing channels as `out` using multidimensional tables. Processes are set up as sites. The computation of the process at a typical grid point `(i,j)` is shown below as `point(i,j)`. A process reads data along incoming channels from its four neighbors, computes and then writes to its outgoing channels, as a step. (We do not show the processes at the boundary of the grid.) The steps are repeated until some convergence criterion is met. We show a template for an unending computation.

```
def laplace(m,n, ...) =

  val in =
    multiTable([(0,m-1),(0,n-1)], lambda(i,j) = Channel())

  val out =
    multiTable([(0,m-1),(0,n-1)], lambda(i,j) = Channel())

  def read(i,j)    = in(i,j).get()
  def write(i,j,x) = out(i,j).put(x)

  def point(i,j) =
   (read(i-1,j),read(i+1,j),read(i,j-1),read(i,j+1))
   >(u,v,x,y)>

    -- using input (u,v,x,y) solve one step for point (i,j).
    -- Put result in variable res

    ...

   >> (write(i-1,j,res),write(i+1,j,res),
       write(i,j-1,res),write(i,j+1,res))

   >> point(i,j)

-- Goal of laplace: start all step(i,j) concurrently

upto(m) >i> upto(n) >j> point(i,j)
```

5.4.8 Warshall's Algorithm for Graph Transitive Closure

We illustrate the application of closure in coding an algorithm due to Warshall [46] that computes the transitive closure of a finite, directed graph. Given is a boolean matrix c where $c(i,j) = true$ if and only if there is an edge from node i to node j; assume that the graph has n nodes numbered 0 through $n-1$. The transitive closure matrix t has $t(i,j) = true$ if and only if there is a path from node i to node j. If for every i, $c(i,i) = true$ then t computes paths of length 0 or more; if $c(i,i) = false$ then t computes paths of length 1 or more, which is often useful in determining if a node belongs to a loop.

Warshall's algorithm computes a succession of matrices c^k, for $0 \leq k \leq n$, where $c^k(i,j) = true$ if and only if there is a path from node i to node j in which all intermediate node numbers are less than k. It follows that $c^0 = c$ and $c^n = t$, and

$$c^{k+1}(i,j) = c^k(i,j) \vee (c^k(i,k) \wedge c^k(k,j)) \qquad \text{(W)}$$

Thus, c^{k+1} may be computed from c^k in $O(n^2)$ elementary steps resulting in a $O(n^3)$ algorithm for transitive closure.

We encode equation (W) in two different ways to illustrate different aspects of closure.

Connection Matrix given as a closure Let matrix c be given by a site c, i.e., a closure, where edge (i,j) exists if and only if $c(i,j)$ publishes `true`. We code Warshall's algorithm below in which each intermediate matrix is also represented by a closure. Here, `step(c,k)` assumes that argument `c` is c^k and it publishes c^{k+1}, using equation (W). The auxiliary site `loop` applies `step` in increasing order of `k` starting at `k = 0`. The entire algorithm is started by calling `warshall(w,n)` where `w` is a closure representing the initial matrix, and `n` describes the dimension of the matrix.

```
def warshall(w,n) =

  def step(c,k) =
    lambda(i,j) = c(i,j) || ( c(i,k) && c(k,j) )

  def loop(c,k) =
    if k = n then c else (step(c,k) >d> loop(d,k+1) )

loop(w,0)
```

The program shown above does not actually publish the transitive closure matrix. What it publishes is a site that can be called to retrieve the value of any specific element of the transitive closure matrix. Its computation time is largely independent of the size of the graph. The actual computation takes place when the site `t` published by `warshall(w,n)` is called, as in,

```
warshall(w,n) >t> t(i,j)
```

for some specific `i` and `j`. Then closure `t` unfolds to publish the transitive closure value at `(i,j)`. The execution time would be $\emptyset(n^3)$ for any such computation.

This style of computation is extremely inefficient if even a few elements of `t` are to be retrieved. If it is required to print the entire transitive closure matrix, there will be considerable duplication of computation steps. The solution given below is a better alternative in such cases.

Connection Matrix given as a Table Assume that the input matrix is given as an immutable multidimensional table, `multiTable`, as defined in Section 5.4.7. The modified transitive closure program is nearly identical to the one above except that `step(c,k)` takes a `multiTable c` which is actually c^k and publishes a `multiTable` corresponding to c^{k+1}.

```
def warshall'(w,n) =

  def step(c,k) =
    multiTable(
               [(0,n-1),(0,n-1)],
               lambda(i,j) = c(i,j) || ( c(i,k) && c(k,j) )
              )

  def loop(c,k) =
    if k = n then c else (step(c,k) >d> loop(d,k+1) )

loop(w,0)
```

The publication of `warshall'(w,n)` is a `multiTable t` representing the transitive closure matrix. Computation of `t(i,j)` takes constant time.

5.5 Concluding Remarks

This chapter has illustrated a variety of combinations of Orc expressions to solve a number of small problems. Clearly, the same types of combinations can be applied to larger expressions, or program components by treating them as sites. We shall see other kinds of programming idioms in the following chapters.

```
def calculate2(xs) =

  def expr(xs)   =
       term(xs) >(n,ys)>
       ( (n,ys) | ys >"+":zs> expr(zs) >(m,ws)> (n+m,ws) )

  def term(xs)   =
       factor(xs) >(n,ys)>
       ( (n,ys) | ys >"*":zs> term(zs) >(m,ws)> (n*m,ws) )

  def factor(xs) =
       number(xs) | xs >"(":ys> expr(ys) >(n,")":zs)> (n,zs)

  def number'(i,[])   = (i,[])
  def number'(i,x:xs) =
       digit(x) >v> number'(10*i+v,xs); (i,x:xs)

  def number([])   = stop
  def number(x:xs) = digit(x) >v> number'(v,xs)

  def digit(x) =
      x >"0"> 0
    | x >"1"> 1
    | x >"2"> 2
    | x >"3"> 3
    | x >"4"> 4
    | x >"5"> 5
    | x >"6"> 6
    | x >"7"> 7
    | x >"8"> 8
    | x >"9"> 9

expr(xs) >(n,[])>  "expression value = " + n;
                   "expression is illegal"

-- Usage

characters("3*(4+6*(2+3)+8)+19*2") >xs> calculate2(xs)

-- publishes: "expression value = 164"
```

Figure 5.1: 2-function Calculator

```
def calculate4(xs) =

  def expr(xs)   =
       term(xs) >(n,ys)>
       ( (n,ys)
        | ys >"+":zs> expr(zs)  >(m,ws)> (n+m,ws)
        | ys >"-":zs> expr'(zs) >(m,ws)> (n+m,ws)
       )

  def expr'(xs)   =
       term(xs) >(n,ys)>
       ( (-n,ys)
        | ys >"+":zs> expr(zs)  >(m,ws)> (-n+m,ws)
        | ys >"-":zs> expr'(zs) >(m,ws)> (-n+m,ws)
       )

  def term(xs)   =
       factor(xs) >(n,ys)>
       ( (n,ys)
        | ys >"*":zs> term(zs) >(m,ws)> (n*m,ws)
        | ys >"/":zs> term(zs) >(m,ws)> ((n+0.0)/m,ws)
       )

  def factor(xs) =
       number(xs) | xs >"(":ys> expr(ys) >(n,")":zs)> (n,zs)

  def number'(i,[])   = (i,[])
  def number'(i,x:xs) =
       digit(x) >v> number'(10*i+v,xs); (i,x:xs)

  def number([]) = stop
  def number(x:xs) = digit(x) >v> number'(v,xs)

  def digit(x) =
      x >"0"> 0
   |  x >"1"> 1
   |  x >"2"> 2
   |  x >"3"> 3
   |  x >"4"> 4
   |  x >"5"> 5
   |  x >"6"> 6
   |  x >"7"> 7
   |  x >"8"> 8
   |  x >"9"> 9

expr(xs) >(n,[])>  "expression value = " + n;
                   "expression is illegal"
```

Figure 5.2: 4-function Calculator

```
def matrix(bounds) =

  {- Argument of size is a list of bounds -}
  def size([]) = 1
  def size((lo,hi):bs) =
      Ift(lo <= hi) >> (hi-lo+1) * size(bs)

  val ar = Array(size(bounds))

  def item(is) =

  {- index(acc,xs,is) has
     acc: accumulator for index computation
     xs : list of bounds
     is : list of indices
    Compute acc+j where j is the linear index
    of the element at is.
  -}

    def index(acc,[],[]) = acc

    def index(acc, (lo,hi):xs, i:is) =
        Ift(lo <= i && i <= hi) >>
        index(acc*(hi-lo+1)+(i-lo), xs, is)

  ar(index(0,bounds,is)) -- Goal of item(is)

  -- Define up to 3 dimensional matrices.

  def access([])      = lambda()      = item([])
  def access([_])     = lambda(i)     = item([i])
  def access([_,_])   = lambda(i,j)   = item([i,j])
  def access([_,_,_]) = lambda(i,j,k) = item([i,j,k])

access(bounds) -- Goal of matrix(bounds)

-- Usage

val mat = matrix([(-3,2),(-5,-3),(2,10)])
mat(0,-4,5) := 0+(-4)+5 >> mat(0,-4,5)?

-- publishes 1
```

Figure 5.3: Multidimensional matrix

```
-- Define a matrix with a method to name submatrix
  def class matrix'(bounds) =

    {- Argument of size is a list of bounds -}
    def size([]) = 1
    def size((lo,hi):bs) =
        Ift(lo <= hi) >> (hi-lo+1) * size(bs)

    val ar = Array(size(bounds))

    def submatrix(dim) =
      def item(is) =

      {- index(acc,xs,is) has
         acc: accumulator for index computation
         xs : list of bounds
         is : list of indices
        Compute acc+j where j is the linear index
        of the element at is.
      -}

        def index(acc,[],[]) = acc

        def index(acc, (lo,hi):xs, i:is) =
            Ift(lo <= i && i <= hi) >>
            index(acc*(hi-lo+1)+(i-lo), xs, is)

      -- Goal of item(is)
      ar(index(0,dim,is))

      -- Define up to 3 dimensional matrices.

      def access([])      = lambda()      = item([])
      def access([_])     = lambda(i)     = item([i])
      def access([_,_])   = lambda(i,j)   = item([i,j])
      def access([_,_,_]) = lambda(i,j,k) = item([i,j,k])

  {- Goal of submatrix -}
  access(dim)

  {- Goal of matrix -}
  stop

  -- Usage

  val mat = matrix'([(-3,2),(-5,-3),(2,10)])

  val lu  = mat.submatrix([(-1,1),(-1,-1),(2,3)])

  lu(0,-1,2) := 1 >> lu(0,-1,2)?

  -- publishes 1
```

Figure 5.4: Submatrix

```
def multiTable(ld,fun) =
  val mat = matrix(ld)

-- Site enum enumerates all index lists with the bounds
-- given in its argument.

  def enum([]) = []
  def enum((lo,hi):xs) =
    upto(hi-lo+1) >i> enum(xs) >ys> (i+lo):ys

 -- Populate the table.
 -- Define up to 3 dimensional tables.

  def pop([])      = mat()      := fun()
  def pop([i])     = mat(i)     := fun(i)
  def pop([i,j])   = mat(i,j)   := fun(i,j)
  def pop([i,j,k]) = mat(i,j,k) := fun(i,j,k)

  def acc([])      = lambda()      = mat()?
  def acc([_])     = lambda(i)     = mat(i)?
  def acc([_,_])   = lambda(i,j)   = mat(i,j)?
  def acc([_,_,_]) = lambda(i,j,k) = mat(i,j,k)?

-- Goal of multiTable(ld,fun)
enum(ld) >is> pop(is) >> stop ; -- populate the table
acc(ld)
```

Figure 5.5: Multidimensional Table

Chapter 6

Programming with Lists

An Orc program may communicate with a site using a variety of data structures, each implemented as a site. A list is one such data structure. Other common data structures are channel, which is available in the Orc standard library, html pages and XML records. Lists are the dominant data structure in functional programming. Even though they are not as important in Orc, they are still very useful, and we often use lists for data manipulation internally within Orc programs.

Section 6.1 describes how the well-known list operations are coded in Orc, particularly for concurrent execution. In Section 6.2 (page 161) we introduce a data structure, *powerlist* [41, 42, 1, 26, 29, 5], that is especially suited for parallel algorithms. Orc is not designed for lazy execution; yet, we can define and manipulate lists, both finite and infinite, in a lazy style in Orc, see Section 6.3 (page 165).

6.1 Parallel List Operations

A list, as defined in typical functional languages and Orc, is an inherently sequential data structure. The list elements can only be accessed one at a time from head onwards. Any parallelism in operating on a list comes from applying some operation on one or more elements near the head of the list and simultaneously operating on the remaining part of the list. We show algorithms for several classical list operations.

6.1.1 Map

The map over a list applies a given function, `f`, to each list element and publishes the resulting list. The definition in Haskell is:

```
map(_,[])    = []
map(f, x:xs) = f(x) : map(f,xs)
```

This is exactly the parallel list map in Orc, which we reproduce here from Section 4.4.2.5. Site `parmap` applies a site `f`, instead of a function, in parallel over the elements of `xs`; deflation forces all calls to `parmap` to unfold concurrently.

```
def parmap(_, [])   = []
def parmap(f, x:xs) = f(x) : parmap(f,xs)
```

As an example of the use of `parmap`, consider broadcasting a message to a list of `listeners`. A message is sent to an individual listener `r` by executing `send(r)`. Then the broadcast is simply

```
parmap(send, listeners)
```

For completeness, we reproduce from Section 4.4.2.5 the map program that applies `f` sequentially from left to right.

```
def seqmap(_, [])   = []
def seqmap(f, x:xs) = f(x) >y> (y : seqmap(f,xs) )
```

6.1.2 Filter

Given is a list and a site `test` that publishes a boolean value when called with any element of the list as argument. It is required to publish the sublist on which `test` is true. The following Orc program applies `test` on the head element of the list and filters the remaining list simultaneously. This has merit only if multiple invocations of `test` can be executed simultaneously. The algorithm runs in time linear in list length because the result sublist can be formed only sequentially.

```
def filter([])   = []
def filter(x:xs) =
    (test(x) , filter(xs)) >(b,ys)> (Ift(b) >> x:ys ; ys)
```

A variation of `filter` is to publish a single boolean value, `true` if the filtered list is non-empty and `false` if it is empty. This is same as computing logical *or* over the test values. We can use the parallel-or idiom from Section 2.5.3.2 (page 38).

```
def parallel_or([])   = false
def parallel_or(x:xs) =

  val b = Ift(test(x)) >> true | parallel_or(xs)

b ; false
```

6.1.3 Fold

Given a non-empty list `[x0, x1, ..., xn]` and a site `f` of two arguments, we consider a simple fold operation defined as

```
fold([x0]) = x0
fold([x0, x1, ..., xn]) = f(x0, fold([x1, ..., xn])
```

A sequential implementation in Orc mimics the definition:

```
def fold(_,[x])  = x
def fold(f,x:xs) = f(x,fold(xs))
```

If `f` is strict, then an execution of `f` requires values of both its arguments, `x` and `fold(xs)`. Therefore, inductively, the computation is sequential. If `f` has no special property, then we can do no better. If `f` is an associative operator then we can apply `f` to several elements of the list concurrently, assuming that the list elements can be accessed in parallel. If `f` is, additionally, commutative, then we can squeeze even more concurrency out of the solution.

6.1.3.1 Associative Fold

We show a parallel implementation of associative fold that builds a sequence of lists. The initial list is the given one; assume that it is non-empty. Each subsequent list is about half the size of the previous list, obtained by folding disjoint pairs of adjacent items. Site `pairfold` builds such a reduced list from an argument list. The associative fold site, `afold`, calls `pairfold` repeatedly until the reduced list has just one item, which is the required result of fold.

The steps within one instance of `pairfold` may be performed in parallel, though different instances of `pairfold` have to be executed in sequence.

```
def afold(f, [x])   = x
def afold(f, xs)    =

 def pairfold([])     = []
 def pairfold([x])    = [x]
 def pairfold(x:y:xs) = f(x,y) : pairfold(xs)

afold(f, pairfold(xs))
```

The number of calls to `pairfold` is logarithmic in the initial list size. If the list elements can be accessed simultaneously, a reduced list can be formed in constant time, proportional to the time required to compute one instance of `f` (assuming that list formation from its elements takes constant time).

Note: Even though the original list is non-empty, `pairfold([])` may still be called from the body of `pairfold`.

Map and Associative Fold A commonly occurring situation is to first apply a map `m` and then fold `f` to a non-empty list of elements. Clearly, we can apply the operations in sequential order. A better solution, when `f` is associative, is to apply the map to a list element when it is retrieved for the first time and use `afold`. This requires a slight change in the `afold` program. The first pass over the list applies both map and fold over the list, and subsequent passes just fold the list using `afold`.

```
def map_afold(m,f, [x])   = m(x)
def map_afold(m,f, xs)    =

  def map_pairfold([])     = []
  def map_pairfold([x])    = [m(x)]
  def map_pairfold(x:y:xs) = f(m(x),m(y)) : map_pairfold(xs)

afold(f, map_pairfold(xs))
```

6.1.3.2 Associative Commutative Fold

The associative fold algorithm of the previous section sequentializes the phases of the computation. As a result, elements that have already been computed during a phase and are ready to be folded must wait until all elements of a phase have been computed. The next program avoids this problem if the fold operator `f` is both associative and commutative.

The strategy is to continually read two items from the list, fold them and append the result to the list until there is just one item in the list. The final item is the desired result. Multiple threads can run concurrently. Unfortunately, the list data structure is not appropriate for such a strategy. It is much better to use a channel that permits both adding and removing items at the same cost. First, we describe how the elements of a channel are folded, and then, very briefly, sketch how to transfer the items from a list to a channel.

Folding items from a channel Suppose that the first `n` elements of a channel `c` are to be folded. If `n` is 1, the first element is read from the channel and published. If `n` exceeds 1, then `f(fold(n/2), fold(n/2))` performs fold simultaneously for two halves of the items in the channel (assuming that `n` is a power of 2), and applies `f` to the results of these two folds. For general values of n, the two occurrences of $n/2$ are replaced by the ceiling and floor of $n/2$; since integer division rounds down the fractional value to the next lower integer, floor of $n/2$ for positive n is simply $n/2$ and the ceiling is $n - n/2$.

```
def chFold(c,1) =  c.get()
def chFold(c,n) =  f(chFold(c,n/2) , chFold(c,n-n/2))
```

The solution is parallel. Yet, it suffers from a major shortcoming. Each recursive call partitions the channel elements into two halves, and the halves have to be computed independently and then folded. Consequently, intermediate results computed in different halves can not be folded immediately. We overcome this problem in the next solution by writing intermediate results back to the channel where they can be immediately used for folding.

A "thread" is a snippet of computation that reads two values from the channel, folds them and writes the result to the channel. It does not publish a value. The code for a thread is:

```
c.put(f(c.get(), c.get())) >> stop
```

Observe that both values from the channel are retrieved in parallel. Below, `threads(k)` starts `k` threads concurrently.

```
def threads(0) = stop
def threads(k) =
      threads(k-1)
    | c.put(f(c.get(), c.get())) >> stop
```

We note two facts about `threads(k)` given that the initial number of items in the channel is n: (1) upon completion of the execution of `threads(k)` the channel contains $n-k$ items whose fold yields the same result as the fold on the initial channel content, and (2) the computation is guaranteed to be completed if $n > k$. Henceforth, F denotes the result of fold applied on the initial channel content. We sketch proofs of these two assertions.

At any point in the execution of `threads(k)`, let chs be the multiset of items in the channel and ths be the multiset of items that have been read by the threads that are still executing. First, we show that fold applied to the elements of $(chs \cup ths)$ yields F. Initially, ths is empty, so the claim is true vacuously. The operations that affect chs and ths are: (1) a thread reading an item from the channel, and (2) a thread writing a value to the channel. In (1), $(chs \cup ths)$ does not change, so the result holds. In (2), a thread writes $f(x, y)$ corresponding to two items x and y. Before the write, both x and y belonged to ths, hence to $(chs \cup ths)$. After a write the thread is no longer executing; so, x and y have been removed from ths and $f(x, y)$ added to chs. Effectively, x and y have been replaced by $f(x, y)$ in $(chs \cup ths)$, thus keeping the fold of $(chs \cup ths)$ invariant. On completion of `threads(k)`, there is no executing thread, so ths is empty. Hence, the fold of chs at that point is F. Each thread consumes two items and writes one item to the chnnel. So, `threads(k)` consumes k items, leaving $n - k$ items in the channel on its completion.

The termination argument requires $n > k$. Without this assumption, there could be a deadlock among the executing threads: given $n \leq k$, each thread may read one item from the channel, thus consuming all the items, and no thread can then proceed, each waiting to read one more item from the channel. Given $n > k$ some thread among the k threads will be able to read two items from the channel, using the pigeon-hole principle. So, some thread will fold two items, write the result to the channel and complete its execution, thus converting the problem to one with $k-1$ threads and k or more items in the channel, a situation that is handled by the inductive hypothesis.

The correctness argument suggests that we may execute `threads(n-1)`, which is both guaranteed to complete and leave a single value in the channel that is the desired result. This observation leads to the following program.

```
def cfold(c,n)   =
  def threads(0) = stop
  def threads(k) =
        threads(k-1)
      | c.put(f(c.get(), c.get())) >> stop
```

```
threads(n-1) ; c.get()
```

Transferring items from a list to a channel If it is required to apply associative and commutative fold over a list, it is best to transfer the list elements to a channel and apply fold. We show two different algorithms for this computation depending on whether the list length is available initially (typically, `length(xs)`, for a list `xs`, can be computed in constant time, because most implementations of list retain its length).

Unknown List length If the list length is not available initially, then one strategy is to first transfer the list elements to a channel and count the number of elements during the transfer. Then `cfold(c,n)` may be executed where `c` is the name of the channel. We show an alternate algorithm that runs the transfer and fold concurrently.

Below, the primary site is `combine`. When `combine(xs, n)` is called, it operates on list `xs` and channel `c` whose length is `n`. The site concurrently transfers items from `xs` to `c` and applies fold to the items in both. Initially, in the goal expression, `c` is empty and `n` is 0. Note that we use the pruning combinator wherever possible to initiate concurrent computation.

```
def list_cfold(f, xs) =
  val c = Channel()

  def combine([], 0) = stop
  def combine([], 1) = c.get()
  def combine([], n) = (c.get(), c.get()) >(x,y)>
                       (combine([], n-1) <<  c.put(f(x,y)) )

  def combine([x], n)      = combine([], n+1) << c.put(x)
  def combine(x:y:rest, n) =
      combine(rest, n+1) << c.put(f(x,y))


  combine(xs, 0)
```

Known List length Below, `list_cfold_n(f,xs,n)` is called with parameters `f` and `xs` as in `list_cfold` and `n` which is the length of `xs`.

The execution strategy is a follows. Execution of `xfer(xs)` transfers all items from list `xs` to channel `c`, and `combine(m)` publishes fold of `m` items from channel `c`. The goal expression `xfer(xs)| combine(n)` starts executing even when `c` does not have `n` items; execution of `combine(n)` waits for `xfer(xs)` to transfer an item to the channel if it can not proceed.

```
def list_cfold_n(f,xs,n) =
  val c = Channel()

  -- Transfer all items of the argument list to channel c
```

```
    def xfer([])    = stop
    def xfer(x:xs)  = xfer(xs) << c.put(x)

    -- combine(m) computes fold of m items from channel c

    def combine(1) =  c.get()
    def combine(m) =
        c.get() >x> c.get() >y>
        ( c.put(f(x,y)) >> stop | combine(m-1))

{- Goal of list_cfold_n -}
xfer(xs) | combine(n)
```

6.2 Powerlist

Parallel operations on an Orc list are difficult because a list is an inherently sequential data structure. In this section, we consider a list structure, called Powerlist [41, 42, 1, 26, 29, 5], that permits parallel access to its elements. It is expected that the powerlist operators can be implemented efficiently on some computing platform, such as a hypercubic architecture, in constant or logarithmic time. We can express a number of data parallel algorithms succinctly using this structure. A powerlist is implemented by a site. We give a very short introduction to powerlist, and show a small number of examples of its usage. We refer the reader to the original sources for more details.

A powerlist is a finite list consisting of 2^n elements, for some $n \geq 0$. The list is enclosed within angular brackets to distinguish it from sequential lists. Thus, $\langle 2 \rangle$ and $\langle 2, 3 \rangle$ are powerlists, a smallest list being one with a single element. The type of a powerlist is given by the type of its elements — all of which have the same type — and its length. A powerlist may include powerlists as elements, to represent matrices and other higher-dimensional structures.

A powerlist may be constructed out of two powerlists of the same size and type using either of the following operators. The *tie* operator, written as $\parallel$, concatenates the elements of the component powerlists[1] . The *zip* operator, written as $\bowtie$, interleaves the elements of the two lists starting with the first list. Both operators are written infix. Thus, $\langle 1, 2 \rangle \parallel \langle 3, 4 \rangle$ is $\langle 1, 2, 3, 4 \rangle$ whereas $\langle 1, 2 \rangle \bowtie \langle 3, 4 \rangle$ is $\langle 1, 3, 2, 4 \rangle$. These operators are analogous to *cons* on sequential lists. We apply the usual pattern matching on these operators. We show a few small examples next.

Note: The examples in this section are written using mathematical font because the tie and zip operations on powerlists can not be displayed using the typewriter font.

[1]This operator was written as a single vertical bar in earlier papers. We adopt a new notation here to distinguish it from parallel composition combinator of Orc.

The following site reverses a powerlist.

$$def\ rev(\langle x\rangle) = \langle x\rangle$$
$$def\ rev(p \parallel q) = rev(q) \parallel rev(p)$$

We could have written the last clause in the definition using $\bowtie$ instead of $\parallel$:

$$def\ rev(p \bowtie q) = rev(q) \bowtie rev(p)$$

Sites rr and rl rotate a powerlist by one place to the right and left, respectively.

$$def\ rr(\langle x\rangle) = \langle x\rangle \qquad def\ rl(\langle x\rangle) = \langle x\rangle$$
$$def\ rr(u \bowtie v) = rr(v) \bowtie u \qquad def\ rl(u \bowtie v) = v \bowtie rl(u)$$

A more involved permutation arises in Fast Fourier Transform [9]. Assign an n-bit index in standard order to each element of a powerlist P. It is required to permute P to P' such that the element with index b in P has index b' in P' where b' is the reversal of the bit string b. Site inv, defined below, implements the desired permutation. It can be understood as follows. The effect of the permutation is that the highest bit of an index becomes its lowest bit; thus, the left half of the list (those whose highest bits are 0) are the elements with even indices in the resulting list, i.e., the first sublist that is obtained by unzipping P'. Since all bits are reversed, the same scheme has to be applied recursively.

$$def\ inv(\langle x\rangle) = \langle x\rangle$$
$$def\ inv(p \parallel q) = inv(p) \bowtie inv(q)$$

The definition of inv may be likened to a De Morgan law in boolean algebra, where tie and zip are logical "and" and "or", respectively, and inv is the negation. The analogy suggests that the following identity holds; its proof can be established easily using induction.

$$def\ inv(p \bowtie q) = inv(p) \parallel inv(q)$$

Notational Convention For the examples with powerlists we coerce an operator that is applicable to the elements of powerlists to the powerlists themselves. Let p and q be powerlists of the same length and $+$ a binary operator applicable to an element of p and the corresponding element of q; then $p + q$ is the powerlist obtained by applying $+$ to the elements of p and q pointwise. Also, $x + p$, where x is an element and p a powerlist, is a powerlist obtained by applying $x+$ to each element of p. Programs employing these conventions are easily translated to programs in Orc; the conventions are not generally useful outside the powerlist theory, so they are not part of the Orc language.

Associative Fold, Prefix Sum It is easy to program associative fold: construct the fold of the two halves independently and then combine the two values using the given function.

$$\begin{array}{l} def\ fold(\_, \langle x \rangle) = x \\ def\ fold(f, p \parallel q) = f(fold(f, p), fold(f, q)) \end{array}$$

There seems to be no faster way to compute the result if f is also commutative.

Next, we show computation of the *prefix sum*, which applies fold to every non-empty prefix of the argument powerlist. Henceforth, for notational convenience, we write the fold function as a binary infix operator $\oplus$. As before, the function is associative, and we further assume that it has a left zero element, *Zero*, such that $Zero \oplus x = x$, for every x. The prefix sum of list $\langle x_0, x_1, .., x_i, .., x_N \rangle$ is $\langle x_0, x_0 \oplus x_1, .., x_0 \oplus x_1 \oplus .. \ x_i, .., x_0 \oplus x_1 \oplus .. \oplus x_N \rangle$.

The following algorithm is due to Ladner and Fischer [30], see Misra [41] for a derivation. The algorithm first applies $\oplus$ to adjacent elements x_{2i}, x_{2i+1} to compute the list $\langle x_0 \oplus x_1, .. \ x_{2i} \oplus x_{2i+1}, .. \rangle$. This list has half as many elements as the original list; its prefix sum is then computed recursively. The resulting list is $\langle x_0 \oplus x_1, .., x_0 \oplus x_1 \oplus .. \oplus x_{2i} \oplus x_{2i+1}, \ldots \rangle$. This list contains half of the elements of the final list; the missing elements are $x_0, x_0 \oplus x_1 \oplus x_2, .., x_0 \oplus x_1 \oplus .. \oplus x_{2i}, ...$ These elements can be computed by "adding" $x_2, x_4, ..,$ appropriately to the elements of the already computed list.

First, we define a site to insert *Zero* as the leftmost element of a powerlist and discard its rightmost element.

$$\begin{array}{l} def\ rs(\langle x \rangle) = \langle Zero \rangle \\ def\ rs(p \bowtie q) = rs(q) \bowtie p \end{array}$$

The Ladner-Fischer scheme is defined by site lf.

$$\begin{array}{l} def\ lf\langle x \rangle = \langle x \rangle \\ def\ lf(p \bowtie q) = \\ \quad val\ t = lf(p \oplus q) \\ (rs(t) \oplus p) \bowtie t \end{array}$$

Polynomial Evaluation A polynomial with coefficients p_j, $0 \leq j < 2^n$, where $n \geq 0$, may be represented by a powerlist p whose j^{th} element is p_j. The polynomial value at some point x is $\sum_{0 \leq j < 2^n} p_j \times x^j$. For $n > 0$, this quantity is

$$\begin{array}{lll} & \sum_{0 \leq j < 2^{n-1}} p_{2j} \times x^{2j} & + \sum_{0 \leq j < 2^{n-1}} p_{2j+1} \times x^{2j+1} \\ = & \sum_{0 \leq j < 2^{n-1}} p_{2j} \times (x^2)^j & + \ x \times \sum_{0 \leq j < 2^{n-1}} p_{2j+1} \times (x^2)^j. \end{array}$$

The coefficients in the two summands are merely the ones with even and odd indices, which can be extracted by applying zip to "deconstruct" the list. The following site, *ep*, evaluates a polynomial using this strategy.

$$def\ \ ep(\langle c\rangle, x) = c$$
$$def\ \ ep(p \bowtie q, x) = ep(p, x^2) + x \times ep(q, x^2)$$

Fast Fourier Transform For a polynomial p of degree n, its Fourier transform is the polynomial evaluated at a list of points $\langle \omega^0, \omega^1, \ .. \ , \omega^{n-1}\rangle$, where ω is the n^{th} principal root of 1. An efficient algorithm [9] can compute this transform in $O(n \ \log n)$ sequential steps or $O(\log n)$ parallel steps. We merely present the algorithm below, see [41] for a derivation. Below, assume that $powers(m)$ publishes $\langle \omega^0, \omega^1, \ .. \ , \omega^{m-1}\rangle$ where ω is the $(2 \times m)^{th}$ principal root of 1; this computation is similar to that of polynomial evaluation.

$$def\ \ FFT(\langle x\rangle) = \langle x\rangle$$
$$def\ \ FFT(p \bowtie q) =$$
$$\quad val\ \ (p', q') = (FFT(p), FFT(q))$$
$$\quad val\ \ u = powers(length(p))$$
$$(p' + u \times q') \parallel (p' - u \times q')$$

Observe that both operators for powerlist construction, tie and zip, are used in the algorithm, much like in the definition of inv.

Sorting Networks Given a powerlist, our goal is to sort its elements in ascending order from left to right. We present two remarkable algorithms for parallel sorting due to Batcher[4]. We merely present the algorithms here; see [41] for the correctness of these powerlist algorithms.

A general method of sorting is given by

$$def\ \ sort(\langle x\rangle) = \langle x\rangle$$
$$def\ \ sort(p \bowtie q) = sort(p)\ \ merge\ \ sort(q)$$

where $merge$ (written as a binary infix operator) creates a single sorted powerlist out of the elements of its two argument powerlists each of which is sorted. In this section, we show two different methods for implementing $merge$.

A binary comparison operator, $\updownarrow$, is used in these algorithms. The operator (written infix) is applied to a pair of equal length powerlists, p and q; it creates a single powerlist out of the elements of p and q by

$$p \updownarrow q = (p\ \ \min\ \ q) \bowtie (p\ \ \max\ \ q)$$

That is, the $2i^{th}$ and $(2i+1)^{th}$ items of $p \updownarrow q$ are $(p_i\ \ \min\ \ q_i)$ and $(p_i\ \ \max\ \ q_i)$, respectively.

Bitonic Sort A sequence of numbers $x_0, x_1, .., x_i, .., x_N$ is *bitonic* if there is an index i, $0 \leq i \leq N$, such that $x_0, x_1, .., x_i$ is monotonic (ascending or descending) and $x_i, .., x_N$ is monotonic. The function bi, given below, applied to a bitonic powerlist returns a sorted powerlist of the original items.

$def\ bi(\langle x \rangle) = \langle x \rangle$
$def\ bi(p \bowtie q) = bi(p) \updownarrow bi(q)$

For sorted powerlists u and v, the powerlist $(u \parallel rev(v))$ is bitonic; thus u and v can be merged by applying bi to $(u \parallel rev(v))$. The form of the recursive definition suggests that bi can be implemented in $O(\log\ N)$ parallel steps, where N is the length of the argument powerlist.

Batcher Merge The following site merges two sorted powerlists of the same length, using $O(\log\ N)$ parallel steps, where N is the length of each argument powerlist.

$def\ bm(\langle x \rangle, \langle y \rangle) = \langle x \rangle \updownarrow \langle y \rangle$
$def\ bm(r \bowtie s, u \bowtie v) = bm(r, v) \updownarrow bm(s, u)$

6.3 Lazy lists

6.3.1 Lazy, Eager, Strict, Lenient Executions

In executing a function or procedure call `f(e)` in any language, we can distinguish four styles of execution[2] , corresponding to two styles for the evaluation of `e` and two for the execution of `f`. The evaluation of `e` is *eager* if it begins immediately; it is *lazy* if it begins only when the value of `e` is needed. The execution of `f` is *lenient* if it begins even before the value of `e` is available; it is *strict* if it begins only after `e`'s value is available.

The evaluation in ML [37] is eager and strict, whereas in Haskell [15] it is lenient and lazy. Orc is lenient and eager. The remaining possibility, strict and lazy, makes no sense because it requires `f` to start only after `e` has been evaluated and `e`'s evaluation to start only when `f`'s execution needs it. Table 6.1 shows the execution styles of these languages and Orc.

	Lazy	Eager
Strict		ML
Lenient	Haskell	Orc

Table 6.1: Execution Styles

Eager execution in Orc implies that in `f | g` executions of `f` and `g` *must* start immediately. Otherwise, real-time based computations, in particular timeout, can not be implemented. Thus,

```
Rwait(1) >> 1 | Rwait(2) >> 2
```

[2]These observations are due to David Kitchin.

will definitely publish 1, because `Rwait(1)` and `Rwait(2)` will be executed eagerly, and so also expression `1` following the response of `Rwait(1)`. In computations where real time plays no role, executions of both `f` and `g` in `f | g` may start asynchronously at arbitrary times. This execution style can not be distinguished from the lenient and eager style of Orc because it satisfies the asynchronous semantics of Orc, as described in Section 3.3 (page 45).

It is possible to write Orc programs that effectively mimic strict or lazy execution styles. A site definition `f(x,y)` in Orc can be made strict very easily: replace the body of `f(x,y)` by `x >> y >> f(x,y)` so that `f`'s execution starts only after both `x` and `y` are bound to values. A similar transformation can be made at the caller's end in case the definition of `f` is not accessible.

Lazy execution, on the other hand, requires some more work. Though the semantics of the calculus is eager, some of the language features are translated in such a manner that the execution becomes lazy. Most prominently, conditional expression **`if`** `e` **`then`** `f` **`else`** `g` is translated to

```
(Ift(b) >> f | Iff(b) >> g) <b< e
```

This implies that `g` is not executed if `e` is true; in fact neither `f` nor `g` is executed if `e` isilent or non-boolean. We discuss lazy execution in detail next.

6.3.2 Lazy Execution

Lazy evaluation[3] is a powerful mechanism for function evaluation. In a programming language like Haskell [15], a function may have an argument whose evaluation would never complete because it might be an infinite list. In typical lazy evaluation the function evaluation and argument evaluation are run in tandem, the argument is evaluated only when it is needed and only as much of it as needed.

We show how Orc can simulate lazy evaluation. Consider the very simplest case, a site definition `f(x)` where `x` is to be bound to some primitive data value. To evaluate `x` only when needed, change the site definition to `f(x')` where `x'` is a closure that publishes the value of `x` when it is called. Every occurrence of `x` in the body of `f` is replaced by `x'()`. Thus, `x` is evaluated only when needed. Now, the caller of `f` has to create the appropriate closure corresponding to `x'` and supply that as a parameter of call to `f`. Further, the same value of `x` must be supplied by different calls to `x'` since `x` can be bound to only one value. So, `x'` *must* memoize the computed value, see Section 7.4 (page 193) for memoization strategies.

This simple strategy does not work if `x` is structured data. In that case, lazy execution demands that computation of `x` need to proceed only up to the point to enable `f` to commence its execution; not all of `x` need be computed. For example, `x` may be a list and only one item of the list may be needed for

[3] "lazy Evaluation" is the preferred term in functional programming because the goal of execution is to evaluate an expression to publish its value. We use "lazy execution" for Orc because an execution does not merely evaluate but may also produce side effects and, possibly, publish multiple values.

the execution of `f` to proceed. In fact, `x` may be an infinite list and computing all of it would be impossible. Our strategy for lazy execution in this case is the standard one, as in ML. We illustrate the techique only for lists. Similar techniques can be used to compute arbitrary structured data in a lazy manner.

6.3.2.1 Definition of Lazy List

A *lazy list* is either empty, denoted by `[]`, or a list of two items `[v,th]`, where `v` is the head element and `th` a closure of 0 arguments that publishes a lazy list when it is called. An infinite lazy list is a lazy list that is never empty. As an example,

```
def ones() = [1,ones]
```

represents an infinite lazy list of 1's. Henceforth, we call the closure associated with a lazy list a *thunk* to distinguish it from more general forms of closures.

Note: Observe that a thunk, like `ones`, may appear unguarded in the definition of the corresponding site. Unguarded recursion of a site, that is, making a call to a site within its own definition, though useful (see Section 5.2.4, page 127, for an example), requires a great deal of care. □

An infinite list is always lazy. A finite lazy list can be easily created from a finite list. First, define site `nil` that publishes an empty list. The thunk `nil` is used in many of the examples.

```
def nil() = []
```

Then the lazy lists corresponding to finite lists `["0"]` and `["1"]` are

```
val zero = ["0",nil]
val one  = ["1",nil]
```

Unlike the lazy lists of functional programming languages, Orc lazy lists need not be deterministic. Below, `random_stream()` publishes a lazy list that represents an infinite sequence of random bits.

```
def random_stream() = [Random(2), random_stream]
```

Note: Our definition of lazy list subverts a principle of good programming: the two elements of the list are of different types. While Orc supports mixing types in the list elements, it is strongly recommended that this practice be avoided. We can overcome the problem by defining a lazy list as a tuple, `(v,th)`. Or, we can define lazy list by a class. We have chosen the current definition to simplify notation in the descriptions of the algorithms.

6.3.2.2 Common Operations on Lazy Lists

Corresponding to sequential `map` operation on finite lists, we have:

```
def lazymap(f,[]) = []
def lazymap(f,[x,th]) = [f(x),lambda() = lazymap(f,th())]
```

The operation corresponding to `filter` on finite lists is given below, where `p` is a predicate on the elements of the list.

```
def lazyfilter([],_) = []
def lazyfilter([v,th],p) =
  def rest() = lazyfilter(th(),p)

if p(v) then [v,rest] else rest()
```

The following site prints the elements of a lazy list successively on the display device. It uses the standard site `Println` that prints its actual parameter on a single line.

```
def lazyprintln([]) = stop
def lazyprintln([v,th]) = Println(v) >> lazyprintln(th())
```

The elements of a lazy list may be time-separated in their publications. Below, `movie(stream,t)` publishes the elements of the infinite lazy list `stream` separated by time `t` each. This can be used to separate video frames by about $1/24^{th}$ of a second to play a movie (but all movies are finite, and a separate clause must be added to the definition to account for this).

```
def movie([v,th],t) =  v | Rwait(t) >> movie(th(),t)
```

We combine some these operations to define the list of all natural numbers, all even natural numbers and all odd natural numbers.

```
def nats()  =
  [0, lambda() = lazymap( lambda(z) = 1+z, nat()   )]
def evens() =
  [0, lambda() = lazymap( lambda(z) = 2+z, evens() )]
def odds()  =
  [1, lambda() = lazymap( lambda(z) = 2+z, odds()  )]
```

Alternatively, we can define `evens()` by doubling each element of `nats()` using `lazymap`, and, similarly for `odds()`.

Fair Merge A *fair merge* of two infinite lists `list1` and `list2` is a list that includes both `list1` and `list2` as sublists, retaining the duplicates, if any, that appear in both lists. There is, however, no a-priori knowledge of the relative order among the elements of the different sublists in the published list. Below, we use a random bit generator to select the argument list from which the next item is chosen for publication.

```
def fairmerge([v,th],[v',th']) =
  if Random(2) = 0 then
     [v , lambda() = fairmerge(th(),[v',th']) ]
  else
     [v', lambda() = fairmerge([v,th],th'())  ]
```

Assume that the random bit generator is at least random to the extent that it does not publish the same bit forever from any point onward. Then it can be

asserted that the next element of every argument list will eventually be selected for inclusion in the published list. Thus, the list `fairmerge(evens(),odds())` is a list of all natural numbers in which the even numbers appear in order and so do the odd numbers, but their relative order is arbitrary.

It is instructive to solve the fair merge problem in a slightly different manner. Define a site `fairmerge'` that takes three arguments, the two lists and an infinite list of random bits according to which the two lists are to be merged.

```
def fairmerge'([v,th],[v',th'],[b,bs]) =
    b >0> [v , lambda() = fairmerge'(th(),[v',th'],bs()) ]

  | b >1> [v', lambda() = fairmerge'([v,th],th'(),bs())  ]
```

Site `fairmerge` is then easily defined using `random_stream` defined previously.

```
def fairmerge(list1,list2) =
  fairmerge'(list1,list2,random_stream())
```

Fibonacci Sequence A nice exercise in lazy execution is to publish the sequence of Fibonacci numbers. It also illustrates the use of mutual recursion in defining lazy lists. For the informal description, let fib be the sequence of Fibonacci numbers. Then $fib = 0 : 1 : (fib + tail(fib))$ where $fib + tail(fib)$ denotes the item by item sum of the two sequences. It is easy to verify that this is a proper definition of the Fibonacci sequence: writing fib_i as the i^{th} item of fib, for $i \geq 0$, we deduce from the given equation that $fib_0 = 0$, $fib_1 = 1$ and $fib_{i+2} = fib_i + (tail(fib))_i = fib_i + fib_{i+1}$.

Site `lazysum` publishes the item by item sum of two infinite lazy lists.

```
def lazysum([x,th],[y,th']) =
    [x+y, lambda() = lazysum(th(),th'())]
```

Below, site `fib` represents the Fibonacci sequence. Its head element is 0 and tail is the site `tfib`. And, `tfib`'s definition is derived from

$fib = 0 : 1 : (fib + tail(fib))$, which gives
$tail(fib) = 1 : (fib + tail(fib))$

```
def fib()  = [0,tfib]
def tfib() = [1, lambda() = lazysum(fib(),tfib())]
```

Replacing `fib` by its body in the definition of `tfib` we get

```
   [1, lambda() = lazysum(fib(),tfib())]
= [1, lambda() = lazysum([0,tfib],tfib())]
```

Thus, we eliminate mutual recursion to get:

```
def tfib() = [1, lambda() = lazysum([0,tfib],tfib())]
def fib()  = [0,tfib]
```

The same algorithm, using channels, is implemented in Section 8.2.4.

Performance Issues Orc does not support lazy execution by default, so the implementation is not optimized for this form of execution. It would be much faster to compute certain infinite lists by retaining part of the state information in their arguments, as we show below, for natural number sequence.

```
def cheapnat(i) = [i, lambda() = cheapnat(i+1)]
val cheapnats   = cheapnat(0)
```

Or, for Fibonacci,

```
def cheapfib(a, b) = [a, lambda() = cheapfib(b, a+b)]
val cheapfibs      = cheapfib(0,1)
```

We next show a number of standard examples of lazy execution.

6.3.2.3 Prime numbers using Sieving

First, define a site `sieve` that takes an infinite lazy list of positive integers as argument and publishes another infinite lazy list in which no element is divisible by any previous element. Next, define the sequence of primes to be the result of sieving the list of natural numbers beginning at 2.

To define `sieve`, first define site `notdiv` where `notdiv(m,n)` publishes `true` if `n` is not divisible by `m`, and `false` otherwise. Then the procedure for sieving a lazy list `[x,th]` is to: (1) keep `x` as the head element of the published list since, vacuously, it is not divisible by any prior element, (2) apply `lazyfilter` to `th()`, with `notdiv(x)` as the predicate, to remove all multiples of `x`, and (3) sieve the resulting list. Note that the list of natural numbers starting at 2 is obtained by removing the first two elements of `nats()`, defined earlier.

```
def notdiv(m,n) = (n%m /= 0)

def sieve([x,th]) =
    [x,
     lambda() =
       sieve(lazyfilter(th(),lambda(n) = notdiv(x,n)))
    ]

def primes() = nats() >[_,u]> u() >[_,v]> sieve(v())
```

6.3.2.4 Hamming Sequence

We solve a problem, attributed to Hamming in Dijkstra [14], using lazy execution. It is required to publish integers of the form $2^i \times 3^j \times 5^k$, for all non-negative integers i, j and k, in increasing order. Denoting the desired infinite list by h, we have $h = 1 : merge(2 \times h, 3 \times h, 5 \times h)$, where $k \times h$ is an abbreviation for the list obtained by multiplying each item in h by k, and *merge* is a function whose arguments are increasing lists and whose value is the list obtained by merging the argument lists into an increasing list (thus, dropping the duplicates).

First, we develop a site to merge two infinite increasing lists. Definition of `merge` is standard.

```
def merge([x,cl],[y,cl']) =
      Ift(x <: y) >> [x,lambda() = merge(cl(),[y,cl']) ]
    | Ift(x  = y) >> [x,lambda() = merge(cl(),cl'())   ]
    | Ift(x :> y) >> [y,lambda() = merge([x,cl],cl'()) ]
```

Below, `hamming` involves computation of $merge(2\times h, 3\times h, 5\times h)$. Pointwise multiplication of a value with all elements of a lazy list is implemented by site `mult`.

```
def mult(i,xs) = lazymap(lambda(z) = i*z,xs)

def hamming() =
  val h = hamming

  [1, lambda() =
    (merge(mult(2,h()),
       merge(mult(3,h()),mult(5,h())) )
    )
  ]
```

The same algorithm is implemented using channels in Section 8.2.4.

6.3.2.5 Enumerating the strings of a Regular Expression

We show a program for enumerating all the strings denoted by a regular expression. The number of such strings is quite often infinite, thus requiring lazy execution. The program is inspired by the solution in McIlroy [34].

A regular expression is defined over a specified alphabet. A regular expression is either: (1) a single symbol of the alphabet, or for regular expressions e and f (2) (alternation) $e \cup f$, (3) (concatenation) $e.f$, or (4) (Kleene closure) e^*. An example of a regular expression over the alphabet of binary digits is $(0 \cup (1.1))^*$. Alternation, $e \cup f$, is typically written as $e \mid f$; we have chosen a non-standard notation in order to avoid confusion with the usage of | in Orc.

Each regular expression denotes a non-empty set of strings[4] . The denotations appear in Table 6.2 where $\overline{e}$ is the denotation of expression e. Here, symbol a is a generic symbol from the alphabet. We have overloaded the use of $\cup$; it is used both for alternation of regular expressions as well as for set union. The symbol $\otimes$ stands for a form of Cartesian product of two sets where the tuple of strings from the two sets is replaced by their concatenation. The ϵ in the definition of $\overline{e^*}$ is the empty string, written as `""` in the program. The definition of $\overline{e^*}$ is given by a recursive equation where we take the denotation to be the smallest set (the least fixed-point) satisfying the equation. It can be shown that a unique smallest set exists.

Suppose that there is a given total order over the symbols of the alphabet. Define a total order on the strings over the alphabet as follows: (1) a shorter string is smaller than a longer one, (2) for two strings of equal length, the

[4]Standard definition of regular expression includes empty set of strings as a possible denotaton. We have eliminated this possibility in favor of a simpler presentation.

$$\begin{array}{rcl} \overline{a} & = & \{a\} \\ \overline{e \cup f} & = & \overline{e} \cup \overline{f} \\ \overline{e.f} & = & \overline{e} \otimes \overline{f} \\ \overline{e^*} & = & \{\epsilon\} \cup (\overline{e} \otimes \overline{e^*}) \end{array}$$

Table 6.2: Regular Expression Denotations

string that is lexicographically smaller is smaller. Thus, ϵ is the smallest string, "11" < "000" and "01" < "10".

We show a program to publish the denotation of a given regular expression as a list of strings in strict order. The essential programming problem is to implement $\cup$, $\otimes$ and Kleene closure over ordered lists so that the published lists are ordered.

Implementing $\cup$: An implementation of $\cup$, given by site `union` below, takes two ordered lazy lists of strings as arguments and publishes an ordered lazy list that represents their union. The program for `union` is very similar to that of `merge` of Section 6.3.2.4 except that the order relations are different. Strings are compared lexicographically in Orc, so "11" > "000" in Orc whereas "11" < "000" according to the order relation we have just defined. So, strings are first compared by length and only if they are of equal length, they are compared lexicographically.

We define site `compare` where `compare(s,t)` publishes `"lt"`, `"eq"` or `"gt"` depending on whether string `s` is less than, equal to or greater than `t`. We use a helper site `comp` to compare strings of equal lengths.

```
def compare(s,t) =
  def comp(x,y) =
     Ift(x <: y) >> "lt"
   | Ift(x =  y) >> "eq"
   | Ift(x :> y) >> "gt"

-- Goal Expression of compare

    Ift(s.length() <: t.length()) >> "lt"
  | Ift(s.length() =  t.length()) >>  comp(s,t)
  | Ift(s.length() :> t.length()) >> "gt"
```

Site `union` is the counterpart of `merge` except that it includes additional clauses in its definition to take the union of finite lazy lists.

```
def union([],[]) = []
def union([],v) =  v
def union(u,[]) =  u
def union([s,th],[t,th']) =
  compare(s,t) >v>
  (
```

```
    v >"lt"> [s,lambda() = union(th(),[t,th']) ]
  | v >"eq"> [s,lambda() = union(th(),th'()) ]
  | v >"gt"> [t,lambda() = union([s,th],th'()) ]
  )
```

Implementing $\otimes$: The Cartesian product over lazy lists is implemented by site `prod`; note that Orc uses `+` for string concatenation. If either argument list is empty, the resulting list is empty. Otherwise, the argument lists are $(x : xs)$ and $(y : ys)$, for lazy lists xs and ys. The Cartesian product of $(x : xs)$ and $(y : ys)$ has 3 component lists: $[x + y]$, $[x] \otimes ys$ and $xs \otimes (y : ys)$. Each of the component lists is ordered, by assumption. Further, $x + y$ is the smallest string in the product. So, the head element of the result list is $x + y$, and its tail is obtained by doing a union over the remaining two component lists.

```
def prod([],_) = []
def prod(_,[]) = []
def prod([x,th],[y,th']) =
  [x+y,
   lambda() = union(
                    lazymap(lambda(z) = x+z,th'()),
                    prod(th(),[y,th'])
                   )]
```

Implementing Kleene Closure: The denotation of e^*, given in Table 6.2, is $\{\epsilon\} \cup (\overline{e} \otimes \overline{e^*})$. This equation seemingly provides a direct procedure for the computation of the denotation making use of the sites already defined.

```
def Kleene_closure(xs) =
    union(["",nil], prod(xs, Kleene_closure(xs)) )
```

However, this computation is unending because the head element of the resulting list can not be computed. Therefore, we rewrite the definition by considering two separate cases. (1) `xs` contains the empty string: then, `xs` is of the form `["",th]` and it is sufficient to compute `Kleene_closure(th())`, (2) `xs` does not contain the empty string: from the defining equation, the result list contains the empty string and `prod(xs, _)` does not contain the empty string since `xs` does not contain it; so, head element of the result list is `""` and its tail is the thunk that publishes `prod(xs, Kleene_closure(xs))`.

```
def Kleene_closure(["",th]) = Kleene_closure(th())
def Kleene_closure(xs) =
    [ "",
      lambda() = prod(xs, Kleene_closure(xs))
    ]
```

We show the complete program in Figure 6.1 (page 175).

A small test of the program for the regular expression $(0 \cup (1.1))^*$ is shown below.

```
val zero = ["0",nil]
val one  = ["1",nil]

lazyprintln(
  Kleene_closure(
    union(
      zero, prod(one,one) ) ) )
```

The following strings are printed in order ("" is actually a line with no content):

```
"" 0 00 11 000 011 110 0000 0011 0110 1100 1111 00000 ...
```

6.4 Concluding Remarks

Lists are essential in most functional programming languages, less so in Orc. Channels provide a more flexible data structure that allow adding and removing items at equal cost. Further, channels can represent lazy lists more directly. We rework many of the examples of this chapter in Chapter 8 using channels. However, a channel is a mutable data structure, so, a list is a preferred alternative wherever it can be used efficiently.

```
def compare(s,t) =
  def comp(x,y) =
     Ift(x <: y) >> "lt"
   | Ift(x =  y) >> "eq"
   | Ift(x :> y) >> "gt"

   Ift(s.length() <: t.length()) >> "lt"
 | Ift(s.length() =  t.length()) >>  comp(s,t)
 | Ift(s.length() :> t.length()) >> "gt"

def union([],[]) = []
def union([],v) =  v
def union(u,[]) =  u
def union([s,th],[t,th']) =
  compare(s,t) >v>
  (
    v >"lt"> [s,lambda() = union(th(),[t,th']) ]
  | v >"eq"> [s,lambda() = union(th(),th'()) ]
  | v >"gt"> [t,lambda() = union([s,th],th'()) ]
  )

def prod([],_) = []
def prod(_,[]) = []
def prod([x,th],[y,th']) =
  [x+y,
   lambda() = union(
                    lazymap(lambda(z) = x+z,th'()),
                    prod(th(),[y,th'])
                  )]

def Kleene_closure(["",th]) = Kleene_closure(th())
def Kleene_closure(xs) =
    [ "",
      lambda() = prod(xs, Kleene_closure(xs))
    ]
```

Figure 6.1: Enumerating Strings of a Regular Expression

Chapter 7

Programming with Mutable Store

7.1 Introduction

Manipulation of mutable store is a rich source of errors. So is the execution of concurrent threads, even with immutable variables. Their mixture is highly explosive unless handled with extreme discipline. The danger of overwriting the mutable store in one thread while concurrently reading the store in another thread is a classic pattern of error. The care required to distinguish between mutable and immutable variables is neatly captured in the following example[1].

For an immutable boolean variable `b`, `if b then f else g` is equivalent to `b >true> f | b >false> g`. For a mutable boolean variable `c`, `if c? then f else g` is *not* equivalent to `c? >true> f | c? >false> g`. To see the difference, note that `if c? then f else g`, which is translated to `(Ift(x)>> f | Iff(x)>> g)<x< c?`, accesses the value of `c?` just once, and depending on this value sets the value of the immutable variable `x`. Consequently, it chooses one of the branches and discards the other depending on the value of `x`. But the expression `c? >true> f | c? >false> g` accesses the value of `c` twice, one for each branch independently and possibly at different times. Each access to `c` may yield a different result, thus possibly satisfying both pattern matches and resulting in the executions of both `f` and `g`, or satisfying neither and resulting in no further execution.

As a general principle mutable store should be avoided. Unfortunately, it is not possible to do so in all situations, mainly because of performance reasons. Consider solving the following problem in functional programming and also using mutable store: does a given list contain each one of the first million natural numbers exactly once? When it is absolutely essential to use mutable store, there should only be a few such data structures, and, preferably, they should be accessed through Orc factory sites, such as `Ref`, `Cell`, `Array`, `Semaphore` and

[1] Example due to John Thywissen.

`Channel`. Other types of mutable store can be created from these factory sites as a class. Semaphore and channel are probably the easiest sites for disciplined use. A number of examples that use semaphore, in connection with synchronization protocols, appear in Chapter 9. Chapter 8 is devoted to a study of channels.

The most problematic mutable variables in concurrent programming are the `Ref` variables. Their usage should be limited to a small scope, and concurrent access to such variables should be designed with extreme care. A `Cell` is a more disciplined version of `Ref`. Since a cell is a write-once variable, every reader of a cell receives the same value when the read operation succeeds; in particular concurrent reads and writes do not cause data race. A concurrent program in which mutable variables are always written with the same value should use cells for the mutable variables, and completely *avoid locking or other synchronization mechanisms*. Section 7.3 contains several examples that illustrate the use of cells.

7.2 In-situ Array Manipulation

We show a few small examples using arrays in this section. In all these examples, there is never any contention in access to the array elements. So, no locking or other special protocol is required.

7.2.1 Random Permutation

A method due to Fisher and Yates [17], also appearing in Knuth [28], randomly permutes the elements of an array a in place. The crux of the algorithm is in site `randomize` where `randomize(i)` randomizes the initial segment of length i of a. For a segment of length greater than 1 `randomize`: (1) swaps the last item of the segment with a random element of the segment, and then (2) randomizes the prefix of this segment excluding the last item in a similar fashion. Observe that the items chosen for swap may be the same elements, in which case the swap has no effect. The algorithm is entirely sequential. It can be shown that all permutations are equally likely provided the random number generator is unbiased.

Below, `permute(a)` performs the randomization of array `a` by making a call to `randomize(a.length())`. The helper site `swap` exchanges the values of the argument `Ref` variables and then publishes a signal.

```
def permute(a) =
  def swap(i,j) = (i?, j?) >(x,y)> (i := y, j := x) >> signal

  def randomize(1) = signal
  def randomize(i) =
    Random(i)         >r>
    swap(a(i-1),a(r)) >>
    randomize(i-1)

randomize(a.length?)
```

Observe that the logic and control flow of the algorithm are entirely independent of the values of the mutable variables. Since the algorithm operates sequentially, data race is not a problem.

7.2.2 Odd-Even Transposition Sort

Odd-even transposition sort is a simple in-situ sorting algorithm. It is relatively inefficient as a sequential sorting algorithm, but it may be efficient for parallel architectures in which the interconnection structure allows efficient communication among neighbors. We pick this example to demonstrate a mixture of sequential and concurrent control; the execution consists of sequential *phases* where each phase involves concurrent executions of threads.

We are given an array a of n items, $a(0)$ through $a(n-1)$, that has to be sorted in place so that $a(i) \leq a(i+1)$, for all i, $0 \leq i < n-1$. The sorting procedure consists of two phases, *even* and *odd*, that are executed alternately until the array is sorted. In the even phase, the adjacent items $a(2 \times i)$ and $a(2 \times i+1)$, for all i, are put in order, i.e., if $a(2 \times i) > a(2 \times i+1)$, the items are exchanged. In the odd phase, all pairs $a(2 \times i-1)$ and $a(2 \times i)$ are put in order. The algorithm alternately executes the phases until neither phase exchanges any pair of items. It can be shown that the array is sorted at that time. Assume that all items of the array are integers.

Each phase can be executed concurrently for each pair. However, the phases themselves are executed in a strict sequential order. We show a concise representation in Orc that highlights both concurrent and sequential aspects of this algorithm.

First, we describe the basic site that sorts two elements $a(i)$ and $a(j)$ in place, where $i < j$, in non-decreasing order. It publishes `true` if an exchange occurs, `false` otherwise.

```
def sort2(i,j) = -- i < j
  (a(i)?,a(j)?) >(u,v)>
  (if u :> v then (a(i) := v, a(j) := u) >> true
             else false)
```

Site *phase* is used to carry out the computation of a single phase. We define this site recursively: $phase(i)$ denotes a computation carried over the first i elements of a in which the rightmost element, $a(i-1)$, participates in a comparison (if $i \geq 2$). The site publishes a boolean, `true` if an exchange occurs during this computation, `false` otherwise.

```
def phase(0) = false
def phase(1) = false
def phase(i) = sort2(i-2,i-1) || phase(i-2)
```

Observe that the standard logical *or* operation (denoted by `||`) is strict; therefore, $phase(i)$ publishes its result only on completion of all the subcomputations.

The computation of $phase(i)$ is concurrent for all pairs because of deflation. A full phase for the entire array is either $phase(n)$ or $phase(n-1)$, one of which is an odd phase and the other even; it is immaterial which one is started first

as long as they are executed alternately. The entire computation is a repetition of $phase(n)$ followed by $phase(n-1)$ until both phases publish `false`.

```
def cyclePhases() =
  phase(n) >b> phase(n-1) >c>
  (if b || c then cyclePhases() else signal)
```

We put the pieces together below. Site `oddevensort` takes an array as argument, sorts it in place, and publishes a signal on completion.

```
def oddevensort(a) =
  val n = a.length?   -- n >= 1

  def sort2(i,j) = -- i < j
    (a(i)?,a(j)?) >(u,v)>
    (if u :> v then (a(i) := v, a(j) := u) >> true
               else false)

  def phase(0) = false
  def phase(1) = false
  def phase(i) = sort2(i-2,i-1) || phase(i-2)

  def cyclePhases() =
    phase(n) >b> phase(n-1) >c>
    (if b || c then cyclePhases() else signal)

cyclePhases()
```

7.2.3 Quicksort

An outstanding example of divide and conquer is quicksort [22]. This is one of the most studied algorithms in computer science. Its performance has been studied extensively, by Knuth [27] and Sedgewick [44] in particular. A variety of implementations exist on different architectures, and many variants of quicksort have been developed that improve its performance for specific platforms.

The structure of the algorithm has also been studied extensively, mainly in teaching program development using recursion. Yet, a concise description of the algorithm that does justice to its various aspects —mutable store, recursion and concurrency— does not appear in the literature. Functional programs typically do not admit in-situ permutation of data elements, imperative programs are typically sequential and do not highlight concurrency, and typical concurrency constructs do not combine well with recursion. Concurrency turns out to be an essential ingredient in the simplification of the algorithm description. Further, it is easy transform the concurrent algorithm for sequential implementation.

We treat this example for its pedagogic value. Presenting quicksort as a concurrent program shows its structure more clearly, and allows a number of options for implementation on a multiprocessor. If an application requires a

high-performance sorting algorithm, the algorithm should be explicitly coded to exploit the characteristics of the computing platform.

7.2.3.1 The Program Structure

We define a site, `quicksort`, that takes an array `a` of numbers as argument, sorts that array in place in ascending order, and publishes a signal upon completion. An auxiliary site, `segmentsort`, defined within `quicksort`, sorts a segment of the argument array in place. The goal expression of `quicksort` is a call on `segmentsort` with the entire array as the segment.

Sorting a segment A segment is given by a pair (u, v) of indices into the array; the items in the segment are $a(i)$, $u \leq i < v$. Thus, the length of the segment is $v - u$ if $u \leq v$, and 0 otherwise. For any segment (u, v), execution of `segmentsort(u,v)` sorts the segment in place in ascending order, and publishes a signal upon completion. The goal expression of `quicksort` is simply `segmentsort(0, a.length?)`.

Henceforth, we use "left" of an item or index to denote the item or index with a smaller index; similarly, "right" . Site `segmentsort` uses a helper site `part` that has three arguments (p, s, t), where p is a number greater than or equal to $a(s)?$, and s and t specify a segment. Execution of `part(p,s,t)` permutes the items of the segment such that all items in the left of the segment are at most p and in the right are greater than p; the dividing index between left and right sub-segments is the value published by `part`. More formally, execution of `part(p,s,t)` publishes m where $a(i) \leq p$ for all i, $s \leq i \leq m$, and $a(i) > p$, for all i, $m < i < t$. Observe that the left sub-segment is always non-empty because $a(s)? \leq p$, though the right sub-segment may be empty.

The declaration and goal expression of `segmentsort` are shown below. Site `swap`, defined in Section 7.2.1, takes two references as arguments, exchanges their values, and returns a signal. `segmentsort` immediately publishes a signal if the segment length is less than 2, since such a segment is trivially sorted. Otherwise, `part(a(u)?, u, v)` publishes m with the given meaning; then `a(u)` and `a(m)` are swapped. The two resulting subsegments are sorted independently and concurrently by `(segmentsort(u, m), segmentsort(m+1,v))`, and a signal published on their completion.

```
def swap(i,j) = (i?, j?) >(x,y)> (i := y, j := x) >> signal

  def segmentsort(u, v) =

  if v - u >= 2 then
    part(a(u)?, u, v) >m>
    swap(a(u),a(m))   >>
    (segmentsort(u, m), segmentsort(m+1,v)) >>
    signal

  else signal
```

Partitioning a segment We define `part(p,s,t)` using two auxiliary sites, `lr` and `rl`, whose codes are shown below. These sites scan the segment from the left and right respectively, looking for the first out-of-place element. Site `lr` publishes the index of the leftmost item in the segment that exceeds `p`, or simply `t` if there is none. Site `rl` publishes the index of the rightmost item that is less than or equal to `p`; since the value at `a(s)` is less than or equal to `p` such an index always exists.

```
def lr(i) = Ift(i <: t) >> Ift(a(i)? <= p) >> lr(i+1); i
def rl(i) = Ift(a(i)? :> p) >> rl(i-1) ; i
```

Observe that the code of `lr` tests `a(i)?` only if $i < t$.

If two out-of-place elements are found by `lr` and `rl`, they are swapped using site `swap`. Next the unscanned portion of the segment is partitioned further until the entire segment has been scanned. Sites `lr` and `rl` may safely be executed concurrently since they do not modify the array elements. The goal expression of `part(p,s,t)` is:

```
(lr(s+1), rl(t-1)) >(s', t')>
(if(s' <: t') then swap(a(s'), a(t')) >> part(p, s', t')
 else t'
)
```

The `quicksort` program in its entirety is given in Figure 7.1 (page 183).

7.2.3.2 Correctness

The correctness of the program as presented here is fairly standard. The only involved part is the goal expression of `part(p,s,t)` shown above. We give an informal argument of its correctness.

Whenever `part(p,s,t)` is called either from `segmentsort` or recursively from `part`, $s < t$ holds. The execution of `part(p,s,t)` starts by executing both `lr(s+1)` and `rl(t-1)`; the executions are non-interfering and they do not modify the segment.

Execution of `lr(s+1)` publishes s' where:

$$a(i)? \leq p, \text{ for all } i,\ s+1 \leq i < s', \text{ and}$$
$$\text{either (1) } s' = t \text{ or (2) } s' < t \text{ and } a(s')? > p \qquad (*1)$$

Similarly, execution of `rl(t-1)` publishes t' where:

$$a(i)? > p, \text{ for all } i,\ t' < i < t, \text{ and } a(t')? \leq p \qquad (*2)$$

Our goal is to show that the execution of `part(p,s,t)` permutes the items in the segment and publishes m where:

$$a(i) \leq p \text{ for all } i,\ s \leq i \leq m, \text{ and}$$
$$a(i) > p, \text{ for all } i, \text{ where } m < i < t \qquad (*3)$$

```
def quicksort(a) =

  def swap(i,j) = (i?, j?) >(x,y)> (i := y, j := x) >> signal

  def segmentsort(u, v) =

    def part(p,s,t) =

      def lr(i) = Ift(i <: t) >> Ift(a(i)? <= p) >> lr(i+1); i
      def rl(i) = Ift(a(i)? :> p) >> rl(i-1) ; i             #

    -- Goal expression of part

    (lr(s+1), rl(t-1)) >(s',t')>
    (if(s' <: t') then swap(a(s'), a(t')) >> part(p,s',t')
     else t'
    )                                                       #

  -- Goal expression of segmentsort

  if v - u :> 1 then
    part(a(u)?, u, v) >m>
    swap(a(u),a(m))   >>
    (segmentsort(u, m), segmentsort(m+1,v)) >>
    signal

  else signal

-- Goal expression of quicksort

segmentsort(0, a.length?)
```

Figure 7.1: Quicksort

Following the execution of `(lr(s+1), rl(t-1))>(s', t')>` we have the following 3 cases.

- $s' < t'$: Then, $s' < t$ because $t' < t$. From (*1) and (*2),

  $a(i)? \leq p$, for all i, $s+1 \leq i < s'$, and $a(s')? > p$,
  $a(i)? > p$, for all i, $t' < i < t$, and $a(t')? \leq p$

Executing `swap(a(s'), a(t'))` has the effect that

  $a(i)? \leq p$, for all i, $s+1 \leq i \leq s'$, and
  $a(i)? > p$, for all i, $t' \leq i < t$

Executing `part(p, s', t')` then establishes (*3), using induction on segment length.

- $s' \geq t'$ and $s' = t$: From (*1), the entire segment has items less than or equal to p (recall that $a(s)? \leq p$). In particular, $a(t-1)? \leq p$. From (*2), $t' = t-1$, which satisfies (*3) with $m = t'$.

- $s' \geq t'$ and $s' < t$: From (*1) and (*2)

  $a(i)? \leq p$, for all i, $s+1 \leq i < s'$, and $a(s')? > p$,
  $a(i)? > p$, for all i, $t' < i < t$, and $a(t')? \leq p$

Therefore, $t' = s' - 1$ and (*3) is satisfied with $m = t'$.

7.2.3.3 Remarks on the `quicksort` program

The quicksort program is highly concurrent, in sorting multiple segments simultaneously as well as scanning a segment from both ends simultaneously. Curiously, all the Orc combinators are used in this program *except* the parallel combinator. The program can be made entirely sequential by replacing concurrent execution within each tuple by a sequential combinator. Thus,

```
(lr(s+1), rl(t-1)) >(s',t')>, and
(segmentsort(u, m), segmentsort(m+1,v))
```

can be replaced, respectively, by

```
lr(s+1) >s'> rl(t-1) >t'>, and
segmentsort(u, m) >> segmentsort(m+1,v)
```

The `swap` site can also be written sequentially.

```
def swap(a,b) = a? >x> b? >y> a := y >> b := x
```

Even though the given program is highly concurrent and it manipulates a shared mutable store, it uses no locking or explicit synchronization. This is because the algorithm partitions the array into segments each of which is under the control of a single thread, with one exception. The exception is that a segment may be scanned simultaneously by both `lr` and `rl`, but neither of them modifies the segment.

7.3 Graph Traversal

We consider a few graph traversal algorithms that visit the nodes of a graph in some specified order. In particular, we consider algorithms for marking reachable nodes from a specific node, *breadth-first* and *depth-first* traversals. In each case, a mutable store is used to record the nodes that have already been visited. These algorithms are standard and we refer the reader to a text book for longer descriptions and proofs of these algorithms. Here, we give a brief sketch of each algorithm as it pertains to its coding in Orc.

The structure of a directed graph can be given by a site definition by enumeration of the neighbors of each node. For example, the graph shown in Figure 7.2 is represented by

```
def succ(0) = [1,2]
def succ(1) = []
def succ(2) = [1,3]
def succ(3) = [0]
```

so that `succ(0)` is `[1,2]` and `succ(0)` is `[]`. This is an appropriate representation when the graph structure is immutable. An undirected graph is similarly represented where the successors are simply the neighbors (i.e., the nodes connected directly to a given node); therefore, each edge (i, j) is represented by having j in i's neighbor list and i in j's.

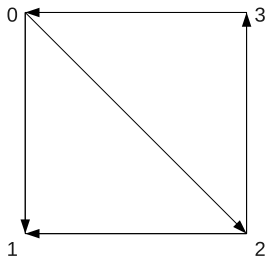

Figure 7.2: An example of a directed graph

In order to enumerate all successors of a specific node i we use the standard library site `each` that publishes all elements of a list. Its definition is:

```
each([])   = stop
each(x:xs) = x | each(xs)
```

Thus, `each(succ(i))` publishes all successors of node i.

7.3.1 Reachability

First, we start with an elementary algorithm to identify the nodes reachable from a specific node. Consider a directed graph with a specified `root` node. It is required to mark every node that is reachable from `root` by a path of length 0

or more. We propose a concurrent algorithm and a sequential algorithm. Each algoritm publishes a table of cells, one cell per node in the graph, where a cell is assigned value `true` if the corresponding node is reachable, and is assigned `false` otherwise. Assume that the nodes are numbered consecutively starting at 0. Henceforth, *marking a node* means that the corrsponding cell is assigned `true`.

7.3.1.1 A concurrent algorithm for reachability

This algorithm implements a flooding technique by first marking `root` and then starting the algorithm concurrently from all its successors as if each one is `root`. After conclusion of the markings the cells corresponding to the unmarked nodes are assigned `false`.

The only point of interest is to eliminate duplicate explorations from a node. We do so by starting new searches only from freshly marked nodes. Further, concurrent computations may attempt to mark a node multiple times. Since marking a node is equivalent to writing to a cell, multiple markings are prevented; at most one value can be assigned to a cell and subsequent attempts for assignment halt silently.

In Figure 7.3 (page 187), site `reachConc` marks the reachable nodes and publishes a table that holds the marking information. Its arguments, `(n,root,succ)`, are as follows: there are `n` nodes in the graph, numbered 0 through n − 1; reachable nodes from `root` node are to be marked; and `succ` represents the graph structure, as described previously. So, `reach(10,0,succ)>t> t(3)?` publishes `true` if and only if in a given graph of 10 nodes whose structure is given by `succ` node 3 is reachable from node 0.

The program uses an auxiliary site `sprout`. Execution of `sprout(i)` marks every unmarked successor of `i`, applies `sprout` to the nodes so marked and halts eventually without publishing a value.

The goal expression applies `sprout` to the `root`, and after its termination assigns `false` to the cells corresponding to the unmarked nodes and publishes the table of cells.

7.3.1.2 A sequential algorithm for reachability

A sequential reachability algorithm, `reachSeq`, defined below, has the same interface and an analogous goal expression as site `reachConc` of the concurrent version. Analogous to the auxiliary site `sprout` we have a site `sproutSeq` that has the same functionality but that executes sequentially. Site `sproutSeq` is called with a list of nodes that are to be explored. Its execution marks every node reachable from any node in the list and then halts. The execution of `sproutSeq([])` merely halts since there is no node to be marked. The execution of `sproutSeq(x:xs)` first attempts to mark `x`. If the attempt succeeds then it explores all successors of `x` and then executes `sproutSeq(xs)`. If the attempt fails then `x` is already marked and its successors have been explored already; so, no further action is required for `x` and only `sproutSeq(xs)` is executed.

```
def reachConc(n,root,succ) =
  val mark = Table(n, lambda(_) = Cell() )

-- Below, mark(i) := true halts if node i is already marked
-- otherwise, it marks node i.
  def sprout(i) =
    mark(i) := true >> each(succ(i)) >j>  sprout(j)

-- Goal Expression of reachConc
sprout(root);
upto(n) >j> mark(j) := false >> stop;
mark

-- usage; first set up the graph
val N       = 5
def succ(0) = [1,2,4]
def succ(1) = [3]
def succ(2) = [3,4]
def succ(3) = []
def succ(4) = [1]

-- Goal expression of the program:
-- publish the reachability list from node 4

reachConc(N,4,succ) >r> [r(0)?, r(1)?, r(2)?, r(3)?, r(4)?]

-- publishes [false, true, false, true, true]
```

Figure 7.3: Concurrent algorithm for reachability

```
def reachSeq(n,root,succ) =
  val mark = Table(n, lambda(_) = Cell() )

  def sproutSeq([])   = stop
  def sproutSeq(x:xs) =
      mark(x) := true >> sproutSeq(succ(x));
      sproutSeq(xs)

sproutSeq([root]);
upto(n) >j> mark(j) := false >> stop;
mark
```

Exploring Nodes in Breadth-First Order The order in which the nodes are marked by `reachSeq`, given a list of marked nodes `(x:xs)`, is to mark all reachable nodes from `x` before the nodes in `xs` are explored, i.e., a depth-first style of marking. Here, we show a breadth-first style algorithm for sequential marking, i.e., one in which only the *immediate* unmarked successors of `x` are marked and then the nodes in `xs` are explored.

We consider breadth-first computation in the next section, so we don't dwell on this algorithm in any detail. The only goal here is to show how to order computations a, b and c where a is first attempted, if it succeeds then b is executed else c is executed. The definition of `sproutSeq(x:xs)`, above, is modified so that if the marking of `x` succeeds then the nodes in `xs` are explored followed by the nodes in `succ(x)`; if it fails (because `x` is already marked) only the nodes in `xs` are explored. The trick is to encode the success or failure of the marking of `x` in a boolean variable `b`.

```
def reachSeq'(n,root,succ) =
  val mark = Table(n, lambda(_) = Cell() )

  def sproutSeq'([])   = stop
  def sproutSeq'(x:xs) =
      (mark(x) := true >> true; false) >b>
      (if b then (sproutSeq'(xs); sproutSeq'(succ(x)) )
            else  sproutSeq'(xs)
      )

sproutSeq'([root]);
upto(n) >j> mark(j) := false >> stop;
mark
```

7.3.2 Breadth-first Traversal

The goal of breadth-first traversal of a directed graph is to construct a *breadth-first tree* over the reachable nodes from node `root`. Define the *distance* of any reachable node as the smallest number of edges on any path from `root` to that node. A breadth-first tree has the specified `root` node as its root. And, a

reachable node at distance `d+1` has a node of distance `d` as its parent in the tree.

All breadth-first tree construction algorithms are based on the following scheme. A node is *marked* means that its parent has been determined. The algorithm maintains a set of nodes S and a non-negative integer d with the invariant: all nodes in S are at distance d and they are all marked. Initially, $d = 0$ and S contains only the root, which we mark in a vacuous style. A round of computation replaces every x, $x \in S$, by its unmarked successors, setting x as their parent. Thus, a round maintains the invariant with d replaced by $d+1$. We discuss a sequential and a concurrent implementation of this scheme.

7.3.2.1 Sequential breadth-first traversal

The algorithm is similar to the reachability algorithm: whenever a successor y of x is marked, x is designated to be the parent of y in the breadth-first tree. We maintain a table of cells, as in the reachability algorithm, but a cell for a node is assigned the name of its parent.

The algorithm implements set S as a list of nodes. All nodes in S have been marked but their successors may not yet have been marked. Initially, the `root` is marked and placed in the list. At each step, the first node x of the list is removed and all its successors are explored as follows. Any marked successor of x is discarded; any unmarked successor is marked by setting its parent to x and placed at the *end of the list*. It is essential that the node be placed at the end of the list, which ensures that the distances of the nodes in the list are monotonic. The execution of the algorithm continues until the list becomes empty. At that point the set of parents encodes the breadth-first tree.

We start with a helper site `expand`, where `expand(x)` is called with a marked node `x`. It marks all unmarked successors of `x` and publishes the list of nodes so marked. It uses an auxiliary site `expand'` where `expand'(x, xs)` is called with a marked node `x` and a list `xs`, a subset of the successors of `x` that includes all unmarked successors of `x`. And `expand'(x, xs)` marks all the unmarked nodes in `xs` (with parent `x`) and publishes the list of nodes so marked.

```
def expand(x) =

  def expand'(_,[]) = []
  def expand'(x, z:zs) =
    (parent(z) := x >> z: expand'(x,zs)) ; expand'(x,zs)

expand'(x,succ(x))
```

Below, site `bfs` publishes the `parent` table representing the breadth-first tree. Its arguments, `(n,root,succ)`, are exactly as in the reachability algorithm of Section 7.3.1. The auxiliary site `bfs'` takes a list of nodes as its argument. For a non-empty argument list `bfs'` takes the first node of the argument list, marks its as yet unmarked successors (using site `expand`), appends the list of the nodes so marked to the end of the list and repeats these steps until the argument list becomes empty. Then it publishes a signal. The goal expression

```
def bfs(n,root,succ) =
  val parent = Table(n, lambda(_) = Cell() )

  def expand(x) =

    def expand'(x,[])    = []
    def expand'(x, z:zs) =
      parent(z) := x >> z:expand'(x,zs) ; expand'(x,zs)

  -- Goal of expand
  expand'(x,succ(x))

  def bfs'([])   = signal
  def bfs'(x:xs) = bfs'(append(xs,expand(x)))

-- Goal of bfs
parent(root) := N >> bfs'([root]) >> parent
```

Figure 7.4: Sequential Breadth-First Traversal

of `bfs` marks the `root`, calls `bfs'([root])` and then publishes `parent`. We take the parent of `root` to be `n`, a non-existent node. Site `append` on lists is in the standard library.

```
def bfs(n,root,succ) =
  val parent = Table(n, lambda(_) = Cell() )

  def bfs'([])   = signal
  def bfs'(x:xs) = bfs'(append(xs,expand(x)))

parent(root) := n >> bfs'([root]) >> parent
```

The complete program is given in Figure 7.4.

The program can be made more compact by defining a single site that takes three arguments, a node, a list of its successors that are yet to be examined (as in `expand'`, above) and a list as in `bfs'`. We show the program in Figure 7.5 without further explanation.

7.3.2.2 Concurrent breadth-first traversal

We develop a concurrent version of breadth-first traversal from the program in Figure 7.4. The definition of `bfs'` offers a rich source of concurrency. The execution of `bfs'(zs)` applies `expand` to each node in `zs`, appends all the resulting lists in order and applies `bfs'` to the final list. This amounts to first applying `map` with site `expand` to the elements of `zs`; this step can be performed concurrently, see Section 6.1.1 (page 155). Then, `append`, being an associative operation, can be applied concurrently using the concurrent associative fold site, `afold` of Section 6.1.3.1 (page 157). We can gain even more concurrency

```
def bfs(n,root,neighbors) =
  val parent = Table(n, lambda(_) = Cell() )

  def bfs''(_,[],[])    = signal
  def bfs''(_,[],z:zs)  = bfs''(z,succ(z), zs)
  def bfs''(x,y:ys, zs) =
      parent(y) := x >> bfs''(x,ys,append(zs,[y])) ;
      bfs''(x,ys,zs)

parent(root):= n >> bfs''(root,succ(root),[]) >> parent
```

Figure 7.5: A compact program for breadth-first traversal

```
def conc_bfs(n,root,succ) =
  val parent = Table(n, lambda(_) = Cell() )

  def expand(x) =
    if succ(x) = [] then []
    else map_afold
         (
          lambda(y) = parent(y) := x >> [y] ; [] ,
          append,
          succ(x)
         )

  def bfs'([]) = signal
  def bfs'(xs) = bfs'(map_afold(expand, append, xs) )

parent(root) := n >> bfs'([root]) >> parent
```

Figure 7.6: Concurrent Breadth-first Traversal

by combining map and fold into one site which interleaves their executions; see `map_afold` of Section 6.1.3.1.

The definition of `bfs'` becomes:

```
def bfs'([]) = signal
def bfs'(xs) = bfs'(map_afold(expand, append, xs) )
```

Next, we apply a similar transformation to `expand` with a map and associative fold operation. We may view `expand(x)` as taking a list, `succ(x)`, as input. The map operation scans an element `y` of `succ(x)`, marks it if it is unmarked and publishes `[y]`, and publishes `[]` if `y` is already marked. The fold operation is again `append`; it simply appends the lists, of length 0 or 1, published for each element of `succ(x)` by map. Since `map_afold` is defined only over a non-empty list, it is called only if `succ(x)` is non-empty. The complete program appears in Figure 7.6.

Mapping with `expand` The map operation in functional programming applies a true mathematical function to the elements of a list. Since the applications of functions are non-interfering, a parallel map can apply the mapping function on all elements of a list simultaneously. In applying map with `expand` in Figure 7.6, executions of two `expand`s may interfere in that both may attempt to mark a specific node. Yet, their executions are non-interfering in the following sense: concurrent execution of multiple `expand`s marks exactly the union of all nodes each `expand` attempts to mark, though the marked nodes may acquire different parents. For instance, if both nodes `p` and `q` have node `r` as a successor, and `expand(p)` and `expand(q)` both attempt to mark `r` simultaneously, one of them will succeed and `r` will acquire a parent, either `p` or `q`. Even though the result is non-deterministic, it suffices for the construction of the breadth-first search tree. This property allows us to apply parallel map with `expand` as the mapping site.

Non-deterministic outout of concurrent breadth-first traversal The breadth-first search tree published by `conc_bfs` is non-deterministic. This is because the successors of many nodes are explored concurrently, and such explorations may mark the reachable nodes in undetermined order. As described above, if both nodes `p` and `q` have node `r` as a successor, and `expand(p)` and `expand(q)` both attempt to mark `r` simultaneously, one of them will succeed and `r` will acquire a parent, either `p` or `q`. Both `p` and `q` have the same distance from the root; so, either one is an acceptable choice for parent of `r`.

7.3.3 Depth-first Traversal

We develop a sequential program for depth-first traversals of undirected graphs. It is possible to traverse a directed graph in this style, though we will not develop that program. The traversal program, `dfs`, publishes a *depth-first tree* as described below. Site `dfs` has the same interface as `bfs` —the traversal starts from a specified node called `root`— except that a list of neighbors is used in `dfs` in place of the successors `succ` of a directed graph as in `bfs`.

Site `dfs` uses an auxiliary site `dfs'` that takes a node and a subset of its neighbors as arguments. It builds a depth-first tree from the given node using only the supplied set of neighbors as children and halts on completion. In contrast to the breadth-first traversal, `dfs'` marks just one neighbor, if possible, and starts a depth-first traversal from that neighbor. Only after marking all descendants of this neighbor, `dfs'` backtracks to commence the traversal from the next node in its neighbor list. The goal expression of `dfs` first marks the `root`, then calls `dfs'(root,neighbors(root))`, and finally publishes the table `parent` that encodes the depth-first tree.

In Figure 7.7, the expression `parent(x):= i >> dfs'(x,neighbors(x))` in `dfs'` halts if either `x` is already marked or `dfs'(x,neighbors(x))` completes execution. Either condition starts execution of `dfs'(i,xs)`.

We show that execution of `dfs'(ys)`, for any `ys`, halts eventually. At the time of the call of `dfs'` let p be the number of *orphans*, the nodes that are as

```
def dfs(n,root,neighbors) =
  val parent = Table(n, lambda(_) = Cell() )

  def dfs'(_,[])   = stop

  def dfs'(i,x:xs) =
    parent(x) := i >> dfs'(x,neighbors(x)) ;
    dfs'(i,xs)

parent(root) := n >> dfs'(root,neighbors(root)) ; parent
```

Figure 7.7: Depth-First Traversal

yet unmarked, so have no parents. The proof is by induction on the measure $(p, |ys|)$. We show that each step of execution of `dfs'(i,ys)` either halts or decreases this tuple lexicographically. Clearly, the execution halts if `ys` is `[]`. Consider the point where `dfs'(i,x:xs)` is called. If `x` is already marked, then `dfs'(i,xs)` is executed; inductively, `dfs'(i,xs)` halts eventually because $(p, |xs|)$ is lexicographically smaller than $(p, |x : xs|)$. If `x` is unmarked, then executing `parent(x):= i` decreases the number of orphans, thus decreasing p, and hence the measure.

7.4 Memoization

Memoization is another name for caching the results of a computation that may be invoked multiple times. On the first invocation the result is computed and stored, and subsequent invocations receive the stored result avoiding recomputation. The poster-child for memoization is the Fibonacci function defined by

```
def fib(0) = 0
def fib(1) = 1
def fib(i) = fib(i-1) + fib(i-2)
```

If the computation follows the recursive definition directly then `fib(6)`, say, will call `fib(5)` and `fib(4)`, and `fib(5)` calls `fib(4)` again. In fact, during the computation of `fib(n)` the number of calls to `fib(i)`, for $0 \leq i \leq n$, is $fib(n+1-i)$. Thus, memoization saves an exponential amount of computation in this case.

Memoization is not appropriate for computations that involve sites that publish time-sensitive results, nor sites whose publications depend on the state of the computation. We consider only sites that publish at most one value. This restriction can be removed by storing all published values when a site is called for the first time (provided the end of the computation can be detected), and returning those values when a subsequent call is attempted.

First, consider memoization of a site `f` that has no argument, is not recursive and that publishes just one value. Site `memof`, below, stores the computed result in cell `res`. To indicate if the result has been computed, we use yet another cell, `done`.

```
val done = Cell()
val res  = Cell()

def memof() =
  res? << (done := signal >> res := f())
```

In the very first call to `memof`, a signal is stored in `done`, `f` called, the publication of `f`, if any, is stored in `res` and this value is published. Note that reading from an unassigned cell, `res` in this case, is blocked until the cell is is assigned a value. In subsequent calls to `memof`, the right side of the pruning combinator halts at the point where the assignment to `done` is attempted without executing `f()`, while the left side of the combinator publishes the value stored in `res`. Even if concurrent calls are made to `memof`, which is likely in a computation such as Fibonacci, `f` will be executed just once because the assignment to `done` acts as a guard for `f`'s execution. Defining `res` as a cell ensures that the value is computed and stored just once.

Typically, we would hide the variables `done` and `res` within a class, as shown in Figure 7.8. The class takes a site `f` as its argument and publishes a site from which we can extract the memoized equivalent of `f`.

```
def class simplememo(f) =
  val done = Cell()
  val res  = Cell()

  def main() =
    res? << (done := signal >> res := f())

stop

-- typical usage
val memof = simplememo(f).main

-- Now, call memof() in place of f()
```

Figure 7.8: A class for Memoization

7.4.1 Lazy Table

Orc language provides a very simple memoization scheme in the factory site `Table`; a call to `Table(n,fun)` calls `fun(i)` for each each i, $0 \leq i < n$, and stores the results in an array from which they are retrived on subsequent calls.

In this case memoization is used not necessarily for performance gain, but to retrieve the same data value in each call. For example,

```
val ch = Table(5, lambda(_) = Channel())
```

defines an array of channels, and guarantees that every access to `ch(3)`, say, refers to the same channel.

The standard implementation of `Table` is eager in the sense that all table entries are computed at the time of instantiation. We define a lazy table below in which a table entry is computed at the time of the first access of that entry. This is useful in the situation where computing a table entry is expensive in terms of resources, it is immaterial when the entry is computed, and it is likely that not all entries will be accessed. The price paid in creating a lazy table is the extra storage for memoization.

Below, site `lazyTable` has two arguments, `n` and `fun`, as in `Table.` The implementation of `lazyTable` uses two standard tables, corresponding to variables `res` and `done`, of Figure 7.8 (page 194). An user uses `lazyTable(n, fun)` in place of `Table(n, fun)`, and automatically gets the memoized equivalent.

```
def lazyTable(n, fun) =
  val res  = Table(n, lambda(_) = Cell() )
  val done = Table(n, lambda(_) = Cell() )

  def access(i) =
    res(i)? << (done(i) := signal >> res(i) := fun(i) )

access
```

7.4.2 Memoizing Sites that have arguments

Consider a site, such as `fib`, that has arguments. Additionally, it includes clausal definition and recursion. We show its memoization below in which each recursive call is replaced by a call to the memoized counterpart. The only real issue is to fix the size of the storage where the results have to be cached. Therefore, we need to know the largest argument, `N`, with which `fib` may be called. We use a table of size `N+1` of cells to hold the value of `fib(i)`, $0 \leq i \leq N$, and a similar table corresponding to `done` of Figure 7.8 (page 194).

```
  val N    = 100
  val done = Table(N+1, lambda(_) = Cell())
  val res  = Table(N+1, lambda(_) = Cell())

  def mfib(0) =
    res(0)? <<
     (done(0) := signal >> res(0) := 0)

  def mfib(1) =
    res(1)? <<
     (done(1) := signal >> res(1) := 1)
```

```
def mfib(i) =
  res(i)? <<
   (done(i) := signal >> res(i) := mfib(i-1) + mfib(i-2))
```

This template may be applied to memoize any site that has arguments. As in Figure 7.8 (page 194), the entire definition may be enclosed within a class to prevent external access to `N`, `done` and `res`. Note that there is no real need to memoize the computations of `mfib(0)` and `mfib(1)`; we could have defined their values to be 0 and 1, respectively.

Memoization works particularly well for `fib` because its argument is a small natural number and the computation of `fib(n)` requires values of all `fib(i)`, for `i` smaller than `n`, multiple times. In the general case, the site may have multiple arguments and the arguments may be of arbitrary types that may not map efficiently to a set of consecutive natural numbers. We consider the case of multiple arguments in the next section.

7.4.3 Automatic Memoization of Recursive Functions

This section may be skipped on first reading.

The memoization technique shown in Section 7.4.2 is quite general. However, a *manual* translation of the site definition is required to create the memoized version. In this section, we show a technique for (almost) automatic memoization. We will define a site that accepts a site `f` as an argument and publishes the memoized site corresponding to `f`. The size parameter, similar to `N` for `mfib`, also has to be supplied as an argument. We illustrate the technique where `f` has just one argument, though it is easily generalized to sites with multiple arguments. This material is inspired by Cook [8]; it mirrors the definition of inheritance in object oriented programming.

The site to be memoized, `f`, may have been defined recursively. The first step is to convert the recursive definition to a non-recursive one; this translation is manual, though thoroughly routine. A recursive call is replaced by a call to a site that appears as an additional argument. For example, definition of `fib` may be converted to `gfib` where the recursive call is replaced by a call to an argument site `self`.

```
def gfib(0, _) = 0
def gfib(1, _) = 1
def gfib(n, self) = self(n-1) + self(n-2)
```

We can retrieve `fib` from `gfib` using a fixed point computation as follows. Below, site `fix` takes a site `f` of two arguments and publishes a site of one argument by treating the second argument as a recursive call.

```
def fix(f) =
  def q(x) = f(x,q)
q
```

It is easy to show that `fib = fix(gfib)`; that is, `fib(i)= fix(gfib)(i)` for all natural numbers `i`, by induction on `i`. Note that `fib(i)` is not the same as `gfib(i, gfib)` because the second argument of `gfib` is a site that has just 1 argument, not two.

Next, we define site `memoize` that takes as arguments a non-recursive site `g` and a size parameter `n`. Site `g` has two arguments where the second argument is for the self-referential call as in `gfib`; the second argument is ignored in case there is no self-referential call. The implementation is the standard one, similar to the one shown in Section 7.4.2 (page 195), using the two standard tables, `res` and `done`.

```
def memoize(g,n) =
  val res  = Table(n+1, lambda(_) = Cell())
  val done = Table(n+1, lambda(_) = Cell())

  def main(i,self) =
    res(i)? <<
    (done(i) := signal >> res(i) := g(i,self) )
```

Now, `main` is the memoized equivalent of `g`. It has two arguments exactly as in `g`. The next step is to use `fix` to eliminate the second argument. So, `fix(main)` is the memoized equivalent of `g`. The user's task is to create a non-recursive site `g` from `f`, like `gfib` from `fib`, and call `memoize(g,n)` to obtain a memoized site with size parameter `n`. We show the complete program and its usage in creating memoized Fibonacci in Figure 7.9.

Sites with Multiple arguments We can memoize sites with multiple arguments in a fashion similar to the one argument case. If each argument can be mapped easily to consecutive natural numbers, then `done` and `res` of Figure 7.9 (page 198) are implemented as multidimensional structures of appropriate dimensions (see Section 5.4.7, page 143, for defining multidimensional structures). If the arguments do not map conveniently to consecutive natural numbers, use hashing to map them to a limited domain, the size of the has table. During program execution, the arguments of a call are first hashed and the index to the hash table is used as the index to `res` and `done`.

There is no single `memoize` site that can accept sites of different arity. A different `memoize` site has to be coded for sites having 0 arguments, 1 argument, ... To overcome this problem, convert a site with multiple arguments to one with a single argument. The single argument is a list that includes all the arguments. Thus, given site `f` of three arguments, say, define site `f'`

```
def f'([x,y,z]) = body of f, with f replaced by f'
def f(x,y,z) = f'([x,y,z])
```

Here, external callers may continue calling `f` with its original interface. But each call to `f` in the body of `f'` is replaced by a call to `f'` with suitable change in the call interface. Orc allows list members to be of different types; so, this transformation can be applied to arbitrary sites. If `f` has no argument, `f'` has empty list as its argument.

```
def memoize(g,n) =
  val res  = Table(n+1, lambda(_) = Cell())
  val done = Table(n+1, lambda(_) = Cell())

  def main(i,self) =
    res(i)? <<
    (done(i) := signal >> res(i) := g(i,self) )

  def fix(f) =
    def q(x) = f(x,q)
  q

  fix(main) -- Goal of memoize

 -- Example: memoize fib

def gfib(0, _) = 0
def gfib(1, _) = 1
def gfib(n, self) = self(n-1) + self(n-2)

val mfib = memoize(gfib,100)

-- Compute fib(50) by calling mfib(50)

mfib(50)
```

Figure 7.9: Memoization Site

In order to memoize `f`, first memoize `f'` to obtain `memof'`. Then create the memoized version of `f`, `memof`, by

```
def memof(x,y,z) = memof'([x,y,z])
```

7.5 Pointer-based Data Structures

Pointer-based data structures, or linked data structures, require manipulation of mutable store. A pointer in Orc is implemented by a `Ref` variable whose value is the location of a `Ref` variable. That is, given that variables `r` and `s` are both `Refs`, `s := r` stores a pointer to `r` in `s`. While `s?` retrieves the current value of `s`, which is `r`, `s??` retrieves the value stored in `r`. We illustrate the manipulation of pointers in building data structures that expand (and contract) over time. We use the term *node* for a single item in the data structure.

We first show that implementation of a stack that includes a single pointer in each node. Next, we show a more involved example, the implementation of a binary search tree in which every node includes two pointers. For both cases, we also show implementations that permit concurrent access to the data structures.

7.5.1 Stack implemented as a singly-linked list

We implement a stack as a singly-linked list. The stack permits two operations: `push(x)` adds element `x` to the top of the stack and publishes a signal, and `pop()` removes the top item of the stack, provided the stack is non-empty, and publishes that item; `pop()` halts if the stack is empty (it is easy to modify the implementation so that `pop()` waits, rather than halts, if the stack is empty until a `push` puts an element on the stack.).

A node in the stack has a tuple as value. The tuple consists of an item value and a pointer to the next node below this node in the stack; the pointer is *null* if there is no next node. A pointer is implemented as a list of at most one item; the list is empty if the pointer is null, otherwise the list holds the location of the next node. Variable `top` holds a pointer to the top node of the stack or it is null if there is no top node.

The implementation is straightforward. Execution of `push(x)` creates a `Ref`, stores `x` and the pointer to the current top node of the stack in it, and stores a pointer to this node in `top`. Execution of `pop()` retrieves the tuple from the top node of the stack, stores the pointer to the next node in top, and publishes the value of the stored element. If top stores a null pointer, i.e., `top?` is `[]`, then the pattern match `top? >[r]>` fails and the execution halts. Initially, `top` holds a null pointer.

```
def class stack() =
  val top = Ref([])

  def push(x) =  Ref((x,top?)) >r> top := [r]
  def pop()  =  top? >[r]> r? >(x,s)>  top := s >> x
```

```
  stop
```

The only safe mode of operation with this implementation is sequential access to the stack: only one method is called at a time. If concurrent calls are made to any two of the methods, even the same methods, it is very likely that the method executions will interfere. To allow concurrent calls, but not concurrent method executions, use a semaphore (a lock) to permit only one method to execute at a time, as shown below.

```
def class stack() =
  val sem = Semaphore(1)
  val top = Ref([])

  def push(x) =
    sem.acquire() >>
    Ref((x,top?)) >r> top := [r] >>
    sem.release()

  def pop()  =
    sem.acquire() >>
    top? >[r]> r? >(x,s)>  top := s >>
    sem.release() >> x

stop
```

7.5.2 Sequential Binary Search Tree

We next show an algorithm with more sophisticated pointer manipulation. In particular, we permit concurrent access to the data structure, which is implemented without any explicit lock.

We implement a set with two operations: (1) `insert(x)` inserts item `x` in the set provided `x` is not already in the set; it publishes `true` if and only if the item was not already there and the insertion was successful, (2) `search(x)` publishes `true` if and only if `x` is in the set. The semantics, as described, is for an implementation with sequential access, and our first implementation is for sequential access only.

The set is maintained as a binary search tree. We implement each tree node as a `symCell` from Section 4.6.3.3. A `symCell`, like a `Cell()`, may be assigned at most once. But unlike a `Cell()` a read operation on an unassigned `symCell` halts instead of blocking. A node is *empty* if it is unassigned. A node is *full* if it contains a tuple with three entries `(lp,v,rp)` where `lp` and `rp` are pointers, i.e., `symCell`s to `p`'s left and right children, respectively, and `v` is an element of the set. Either pointer may be null, that is, it references an empty node. Henceforth, we take the elements of the set to be integers so that their magnitudes can be compared.

First, we describe the routine that searches for a given key, `key`, starting at a specified node `p`. The search publishes a tuple `(q,b)` where `q` is the node where the search ends, successfully or unsuccessfully, and `b` is a boolean denoting the result of search (`true` for success and `false` for failure). The search proceeds as follows: (1) if `p` is unassigned then the search ends by publishing `(p,false)`, (2) if `p` contains `(lp,v,rp)` and: (2.1) `key` is less than `v` then the search is started from `lp`, (2.2) `key` is equal to `v` then the search is successful and it publishes `(p,true)`, and (2.3) `key` is greater than `v` then the search is started from `rp`.

```
def searchloop(p,key) = -- p is a symCell

   p? >(lp,v,rp)>
    (  Ift(key <: v) >> searchloop(lp,key)
     | Ift(key =  v) >> (p,true)
     | Ift(key :> v) >> searchloop(rp,key)
    )
  ; (p,false)
```

Observe that if `p` is unassigned, the read operation `p?` halts. This is detected using the otherwise combinator and then `(p,false)` is published.

The tree has a root node, `root`, that is initially empty.

```
val root  = symCell()
```

The search routine that searches for a given `key` simply applies `searchloop` starting at `root`.

```
def search(key) = searchloop(root,key) >(_,b)> b
```

To insert a key, first search for it. Let the search publish a tuple `(q,b)`. If `b` is `true` then the element is already in the tree and `insert` publishes `false`; if `b` is `false` then the element is inserted by storing at `q` a tuple with item value `key` and null pointers for both children, i.e., `(symCell(), key, symCell())`, and then publishing `true`.

```
def insert(key) =
    searchloop(root,key) >(q,b)>
    (Ift(b) >> false;
     q := (symCell(), key, symCell()) >> true
    )
```

We implement an additional operation on the set, `sort`, that publishes all elements of the set in increasing order in a list. For an empty tree the published list is empty. For a nonempty tree the program constructs the lists for the left and right subtrees, and appends them. The helper site `traverse(p)` creates the corresponding list for the elements in the subtree rooted at `p`.

```
def sort() =
  def traverse(p) =
    p? >(lp,v,rp)> append(traverse(lp), v: traverse(rp))
  ; []

traverse(root)
```

The entire program is enclosed within a class definition; it appears in Figure 7.10.

```
def class bst() =
  val root = symCell()

  def searchloop(p,key) = -- p is a symCell

     p? >(lp,v,rp)>
      (  Ift(key <: v) >> searchloop(lp,key)
       | Ift(key =  v) >> (p,true)
       | Ift(key :> v) >> searchloop(rp,key)
      )
    ; (p,false)

  def search(key) = searchloop(root,key) >(_,b)> b

  def insert(key) =
       searchloop(root,key) >(q,b)>
      (Ift(b) >> false;
       q := (symCell(),key, symCell()) >> true
      )

  def sort() =
    def traverse(p) =
      p? >(lp,v,rp)> append(traverse(lp), v: traverse(rp))
    ; []

  traverse(root)

stop
```

Figure 7.10: Sequential binary search tree implementation

7.5.3 Concurrent binary search tree

The binary search tree program of Figure 7.10 (page 202) will almost surely break if its methods are called concurrently. We next modify the program to withstand concurrent access. We can employ the simple solution of using a lock to ensure that at most one method executes at a time. But we show a more sophisticated solution that allows concurrent method executions as long as they are non-interfering, without using explicit locks. A `symCell` includes an implict lock for concurrent accesses, and we exploit this feature in the implementation.

A `search` is a read operation and an `insert` a write operation; so, we can use the Readers-Writers solution from Section 9.1.3 (page 239) whereby multiple `search` operations are permitted simultaneously whereas just one `insert` may execute at a time. But we can do even better, by allowing concurrent

insert operations as long as they do not interfere. Since an `insert` is a `search` followed by a write step, we can let insertions proceed concurrently with other insertions and searches until there is an attempt to write. If the writes are made to different locations they can be executed in arbitrary order. Thus, all method executions may proceed completely concurrently, including executions of multiple insertions, as long as the insertions are in different parts of the tree. Next, we describe the steps needed when writes are attempted at the same location.

If multiple writes are attempted at the same location, an arbitration is required. The arbitration is provided automatically by `symCell`. In Figure 7.10, the only update of a tree node takes place within site `insert` in the assignment `q := (symCell(),key, symCell())`. This assignment may not execute for a specific writer because another writer may have already written into `symCell` `q`; in that case, the execution of this writer halts and the insertion must be retried. The retry, as described next, can start from `q` itself because the nodes in the path to `q` have not been modified.

We empoly an auxiliary site `insert'` that is called with an argument node `p`. It attempts to insert `key` in the subtree rooted at `p`. Site `insert'` behaves almost exactly like the previous `insert` site; the only difference is that if the assignment `q := (symCell(),key, symCell())` fails then `insert'(q)` is called. Thus, assignment to `q` is replaced by the following code fragment.

```
q := (symCell(),key, symCell())  >> true;
insert'(q)
```

Now, `insert(key)` is simply a call to `insert'(root)`.

The entire program is shown in Figure 7.11 (page 205). The program uses no explicit lock. Yet, each operation, `insert`, `search` and `sort`, can be considered atomic, and at any point in the execution all completed operations can be linearly ordered so that every `search` and `sort` publishes a value consistent with the operations preceding it, as explained under Linearizability in Section 4.6.3.3.

Progress properties of the program The concurrent binary search tree does not guarantee that any specific writer, w, will eventually succeed. This is because there may be an unending stream of writers each of which writes into a location just before w's attempt to write into that location. However, this scenario also establishes that *some* writer makes progress, not necessarily w. Therefore, there is system-wide progress though, possibly, individual starvation for writers.

7.6 Concluding Remarks

Mutable stores are essential in most applications. Orc provides a number of factory sites to create mutable stores with built-in concurrency access control mechanisms. A programmer may create even more involved sites using the given ones as the base.

We have sketched a few simple applications using mutable store in order to demonstrate how they are integrated into Orc. We have also shown how concurrent accesses to mutable stores may be handled. For `Ref` variables that are concurrently accessed, the only reasonable solution is to introduce explicit locks for access. However, other kinds of mutable store, `Cell` and `symCell` for instance, allow disciplined access. A complicated algorithm for concurrent binary search tree has been coded in Section 7.11 without using explicit locks.

```
def class conc_bst() =
  val root = symCell()

  def searchloop(p,key) = -- p is a symCell

     p? >(lp,v,rp)>
      (  Ift(key <: v) >> searchloop(lp,key)
       | Ift(key =  v) >> (p,true)
       | Ift(key :> v) >> searchloop(rp,key)
      )
    ; (p,false)

  def search(key) = searchloop(root,key) >(_,b)> b

  def insert(key) =

    def insert'(p) =
      searchloop(p,key) >(q,b)>
      (Ift(b) >> false;
       q := (symCell(),key, symCell())  >> true;
       insert'(q)
      )

  insert'(root)

  def sort() =
    def traverse(p) =
      p? >(lp,v,rp)> append(traverse(lp), v: traverse(rp))
    ; []

  traverse(root)

stop
```

Figure 7.11: Concurrent binary search tree implementation of set

Chapter 8

Programming with Channels

8.1 Introduction

Channels are as important for concurrent programming as lists are for sequential programming. There are a number of reasons why channels are important: (1) there is a causal order between the send and receive of a message along a channel; therefore, a channel can be used to implement causal order among different events of a concurrent program, (2) a channel is a mutable store with disciplined access facilities, and (3) a channel can smooth speed variations between a sender and a receiver, and a bounded channel can be used as a synchronization mechanism between them. Additionally, Orc uses channels in a variety of ways to supplant the powers of the combinators. For example, we may wish a computation to store all its publications in a channel rather than publish them directly, see Section 8.2.1. We illustrate a few other channel-based programming paradigms in Section 8.2. We discuss basic message communicating processes in Section 8.3 and networks of such processes in Section 8.4. Networks that have regular connection structures among component processes and those that evolve over time depending on the computation are discussed in Section 8.5.

One of the great virtues of a FIFO channel is that the sender and the receiver on a channel may run independently as long as there are items in the channel. This is because the `put` and `get` operations on a channel *commute* in that the order of operations is immaterial on a non-empty channel; the resulting states of the sender, receiver and the channel are identical no matter the order in which the `put` and the `get` occur[1] . Commutativity is an important property in a concurrent system, because the processes whose individual operations commute can be executed independently without the fear of data-race [25, 40]. Further, the output of a process is monotone as a function of its inputs for processes

[1] Hoare uses the term *semi-commute* since `put` and `get` do not commute for an empty channel.

with single input channels, in the sense that supplying more data to the process can only lengthen its outputs; no prior output becomes incorrect as a result of receiving more input.

8.2 Programming Idioms with Channels

8.2.1 Execution Scheduling

Many complex thread scheduling problems become tractable by using channels; we show a few examples.

It is required to execute expression `f` until it publishes a value, then start the execution of `g` and continue the execution of `f`. Note that at most one instance of `g` is executed, and all publications of both `f` and `g` are to be published. The problem statement requires the first publication of `f` to be treated differently from its other publications. We do so by storing all publications of `f` in a channel, and in a separate computation reading and processing the items from that channel, as follows. The first item read from the channel triggers (1) publication of that item, (2) initiation of `g`, and (3) publications of the remaining items in the channel. We use a helper site `pub` that continuously reads from its argument channel and publishes the values.

```
def pub(ch) = ch.get() >x> (x | pub(ch))

  val c = Channel()

  f >x> c.put(x) >> stop
| c.get() >x> (x | g | pub(c))
```

As a second example, it is required to execute expressions `f` and `g` until `g` publishes; the publication of `g` is ignored but both `f` and `g` are terminated at that time. Thus, `g` acts as an interrupter of `f`. This problem can not be solved using only the Orc combinators. This is because (1) the termination requirement on `f` dictates that it should occur in the right of a pruning combinator, whereas (2) the use of a pruning combinator dictates that `f` will be terminated as soon as it publishes, thus preventing `f` from publishing more than one value. We overcome this problem by having `f` never actually publish, but write its publications on a channel. We repeat the definition of `pub` below for completeness.

```
def pub(ch) = ch.get() >x> (x | pub(ch))

  val c = Channel()

pub(c) << ( f >x> c.put(x) >> stop | g )
```

As a variation of this problem, consider terminating `f` when `g` publishes, but letting `g` continue its execution. This problem combines elements of both of the earlier problems. We show a solution below where publications of `f` and `g` are stored in channels `c` and `d` respectively, and published by `pub(c)` and `pub(d)`.

```
def pub(ch) = ch.get() >x> (x | pub(ch))

val c = Channel()
val d = Channel()

  g >x> d.put(x) >> stop
| ( (pub(c) | y | y >> pub(d))
     <y< ( f >x> c.put(x) >> stop | d.get() )
  )
```

Observe that `pub(d)` starts execution only after `y` is bound to a value.

8.2.2 Ordered Output

The publications of an expression have no order. For example, in the following program `nat(i)` publishes all natural numbers starting at `i`, and the goal expression prints all natural numbers on a display using the standard site `Println`. There is no guarantee that the numbers appear in order on the display because `nat(0)` publishes in arbitrary order.

```
def nat(i) = i | nat(i+1)

nat(0) >x> Println(x)
```

Sometimes it is necessary to establish an order among the publications of an expression. The expression then should write to a channel instead of publishing, as shown below for `nat(0)`. Below, `Print(ch)` publishes the items from its argument channel in order.

```
def Print(ch) = ch.get() >x> Println(x) >> Print(ch)

val c = Channel()
def nat(i) = c.put(i) >> nat(i+1)

nat(0) | Print(c)
```

Observe that the call `c.put(i)` is completed before `nat(i+1)` is called. Therefore, the numbers are stored in order in channel `c`. By similar argument, the numbers are printed in order on the display by site `Println`.

Publishing all binary strings in order Consider the problem of publishing all binary strings. A simple solution is given in Section 5.2.4. That solution publishes the strings in arbitrary order. Here, we propose a solution that writes the strings to a channel such that the strings are in order.

Below, the program outputs the publications on channel `ch`. However, it needs to read its prior outputs in order to compute the next string; so, every output in `ch` is replicated in another channel `ch'` that is used internally for computation. Site `put2` writes a given string to both `ch` and `ch'`. A binary string is represented by a list of 0s and 1s, which should be interpreted in the reverse order of the list.

```
  val ch  = Channel()
  val ch' = Channel()

  def put2(xs) = (ch.put(xs) , ch'.put(xs))

  def bin'() =
    ch'.get() >xs> ( put2(0:xs) , put2(1:xs) ) >> bin'()

  put2([]) >> bin'()
```

8.2.3 Multi-Reader Channel

A standard channel loses an item once it is read; so two readers on the same channel will read different items from it. In this section, we develop a more general version of channel that allows all its readers to read the same sequence of items. This is useful in situations where the channel contents are needed to compute the subsequent items to be put in the channel, as in the example in Section 8.2.2 to write all binary strings to a channel.

We define a multi-reader channel as a class whose argument n is the maximum number of readers it supports. The readers are numbered 0 through $n-1$. The `put` method on such a channel has the same interface as for a standard channel. The `get` method has an argument i, a natural number below n, that identifies the reader so that this reader is supplied the appropriate next item from the channel. The implementation shown below is quite simple; it represents the multi-reader channel by n separate standard channels, $ch(0)$ through $ch(n-1)$. A `put` operation writes into every channel and a `get` operation by reader i reads from channel $ch(i)$.

```
def class MultiReaderChannel(n) =
  val ch = Table(n, lambda(_)   = Channel() )
  val s  = Semaphore(1)         -- Disallow concurrent puts

  def put(x) =
    s.acquire() >>
    (upto(n) >i> ch(i).put(x) >> stop)
    ; s.release()

  def get(i) = ch(i).get()
stop

-- Usage: Both get operations return 5.

  val mrch = MultiReaderChannel(2)

    mrch.put(5)
  | mrch.get(0)
  | mrch.get(1)
```

There are more elaborate ways of storing items in a multi-reader channel to avoid their duplications. One possibility is to store the items in an array if the channel length is known to be bounded; then, each reader's state is given by an index to this array. Another possibility is to keep the items that are unread by *all* readers in one channel and the remaining items in separate channels for each reader.

Next, we show the use of multi-reader channels in enumerating recursively-defined infinite structures.

8.2.4 Lazy Execution with Channels

We have described lazy execution in some detail in Section 6.3.2 (page 166). Most of the examples in that section showed manipulations of lazy lists. Here, we show how to do some of those examples using channels. In executing a site call `f(x)`, where x represents a data structure with many items, possibly infinite, we liken the computation of `x` to that of a producer and the execution of `f` to a consumer. The producer and the consumer may be connected by a bounded channel that will prevent the producer from over-producing beyond the needs of the consumer by a bounded amount. If the bounded channel length is 1, say, then the producer may produce at most 2 more items beyond the need of the consumer (there could be one item in the channel and the producer may be waiting to add another item). We can also create bounded channels of length 0 using rendezvous-based communication so that producer may produce just one more item beyond the consumer's need; see Section 9.1.1.

Computing Fibonacci sequence lazily A lazy execution that implements a recursively defined structure both produces and consumes items of the structure. A simple example, computation of the Fibonacci sequence, illustrates the situation. We showed in Section 6.3.2 that the Fibonacci sequence may be defined by $fib = 0 : 1 : (fib + tail(fib))$ where $fib + tail(fib)$ denotes the item by item sum of the two sequences. We showed a computation procedure by lazy execution in that section. Here, we redo that example using a multi-reader channel.

Recursion requires that the output sequence must be read at each step to continue the computation. So, we use a multi-reader channel with 3 readers. Reader 0 is external to this program to allow for reading out the Fibonacci numbers, and readers 1 and 2 are internal readers. During the execution of `fibseq` reader 1 reads the entire sequence, fib, and reader 2 reads $tail(fib)$. The goal expression initially stores values 0 and 1, the first two Fibonacci numbers, in the channel, and lets reader 2 read and discard the first item so that subsequently it reads only the items of $tail(fib)$ in `fibseq`.

```
val ch = MultiReaderChannel(3)

def fibseq() =
  (ch.get(1) + ch.get(2)) >x> ch.put(x) >> fibseq()
```

```
ch.put(0) >> ch.put(1) >> ch.get(2) >> fibseq()
```

To make the evaluation lazy, note that no more than 2 items need to be generated beyond what has been consumed at any stage. So, we can use a bounded version of multi-reader channel in place of the unbounded version used above. It is easy to design a bounded multi-reader channel in Orc that has the channel bound and number of readers as parameters.

In many cases, the channel bounds may not be known a-priori or are very hard to compute. Then, define and use a lazy multi-reader channel in which an item is `put` in the channel only if some reader has read all the items from the channel, otherwise the `put` is blocked. We leave these generalizations to the reader.

Hamming Sequence The Hamming sequence is the increasing sequence of integers of the form $2^i \times 3^j \times 5^k$, for all non-negative integers i, j and k. We showed a program for enumeration of this sequence in Section 6.3.2.4 (page 170) using lazy execution. Here, we show how to solve this problem using a multi-reader channel.

Denoting the desired sequence by an infinite list h, we have $h = 1 : merge(2\times h, 3\times h, 5\times h)$. Here, $k\times h$ is an abbreviation for the list obtained by multiplying each item in h by k. The *merge* function has three arguments that are increasing lists of numbers and the function value is the list obtained by merging the argument lists into an increasing list (dropping the duplicate values). Our goal is to compute the items of h in a channel which we also call `h`.

First, rewrite the equation for h as $h = merge(1 : 2 \times h, 1 : 3 \times h, 1 : 5 \times h)$. We imagine that there are three logical channels that carry the items of $1 : 2\times h$, $1 : 3 \times h$ and $1 : 5 \times h$. The goal is to create a single channel, `h`, that carries the merged values of these three channels.

The program is considerably simplified by having `h` as a multi-reader channel that serves as the three logical channels, as well as the channel on which the outputs are written. So, `h` has four readers, the one reader external to the program that is designated as reader 0, and the three readers for the three logical channels. The first logical channel is simulated by reader 1 reading from `h` and multiplying the item read by 2, similarly, readers 2 and 3 multiply the item read from `h` by 3 and 5, respectively.

We define site `merge` below. The arguments `x`, `y` and `z` of `merge` are the smallest items from each of the three logical channels that are yet to be output. In each step, `merge` picks the smallest of these items, `m`, writes it to `h`, and then reads the next item from every logical channel that supplied `m`. Observe that a value, for instance 6, is supplied by $1 : 2 \times h$ as well as $1 : 3 \times h$; so, 6 has to be removed and the next values should be read from both channels. These steps are repeated forever.

Initially, the first item of each logical channel is 1, that is, $2^0 \times 3^0 \times 5^0$, so the goal expression is `merge(1,1,1)`.

```
val h = MultiReaderChannel(4) -- h has 4 readers
```

```
def merge(x,y,z) =
  min(min(x,y),z) >m> h.put(m) >>
  ( if m=x then 2 * h.get(1) else x,
    if m=y then 3 * h.get(2) else y,
    if m=z then 5 * h.get(3) else z
  )
  >(x,y,z)>
  merge(x,y,z)

merge(1,1,1)
```

8.2.5 Exception Handling

An exception is an event that arises rarely in a computation. Exceptions are either unexpected events, e.g., hardware failure, division by zero, or crash of one component of a system, or expected though infrequent events, such as reaching the end of a file during file processing. The unexpected events are "black swans", however their occurrences must be anticipated and handled. By contrast, the end of file event is guaranteed to happen. We may treat all rare events as exceptions.

The rarity of an exception does not simplify the programming task. Exception handling requires a thread to report, or *raise* an exception, and another thread to detect its occurrence, or *catch*, and handle it. In Orc terms, a site call returns a value that carries a special return code, say an indication of division by zero or the end of file. Though the caller can handle the exception by checking each return code, doing so makes for a very messy program structure, especially since site calls may be nested or recursive. We show how channels may be used to streamline exception handling.

To motivate the problem, consider a situation that may arise during a dialog between a bank and its customer. While the customer is completing a transaction, the bank may randomly request additional authentication information. The customer, as a human, can cope with such requests easily. For software, it requires both detection of the request as an exception, transition to an exception handler and resumption of the original computation on completion of exception handling; exception handling breaks up the smooth flow of the main computation. This is exactly the situation a Web programmer faces when he programs a mashup.

The program structure can be simplified if the customer supplies a channel name on which the exceptions are reported by the bank. The customer concurrently monitors the channel and runs its regular computation. The skeleton of such a program is shown below. Here `request(x)` sends a request from a client to a server. The server may respond to the request by sending a result or it may request authentication information. The client supplies a channel name, `exc`, to the server for sending exception request, `r`. The client handles the exception by sending the authentication information corresponding to `r`, `auth(r)`, along `exc` using a concurrent thread.

```
def request(x) =
  val exc = Channel()

  server(x,exc)
| exc.get() >r> exc.put(auth(r)) >> stop
```

The program shown here handles a single exception; further, exception handling is within the code of `request`. These restrictions can be easily removed. A channel can hold a sequence of exceptions and a different channel may be used to communicate the resolutions of the exceptions. Different kinds of exceptions may be reported on different channels. For example, in a matrix based computation, underflow, overflow, division by zero and singularity of a matrix may be reported on different channels. The channels may be monitored in different parts of the program, thus allowing exception handling to be separated from the code proper.

The exception handlers can be coded in a less obtrusive style as follows. Each exception handler is a class without any method. Its goal expression waits to receive the exception report along a specified channel and then handles it. Instantiating the class makes the handler active. Below, we rewrite the code for `request`.

```
def class exception(ch) =

  ch.get() >r> ch.put(auth(r)) >> stop #

val exc = Channel()

val _ = exception(exc)

def request(x) = server(x,exc)
```

Exception handlers are typically quite elaborate. We have not adequately addressed all the intricacies in designing elaborate exception handlers, merely how to use channels to separate the flow of control for the main computation from exception handling.

8.3 Message Communicating Process

A message communicating process is connected to a set of channels, input channels from which it may receive data and output channels on which it may send data; it may send and receive on the same channel. A process computes autonomously, possibly, using some of the data it receives from its input channels, and, possibly, sending the results of its computation along the output channels. As before, we assume that the channels are unbounded, first-in-first-out and error-free, the kind of channel created by the factory site `Channel()`. In Section 8.4, we discuss networks constructed from processes and channels. A network may also be treated as a process within a larger network. In this

section we consider only primitive processes that do not contain processes as components.

8.3.1 Simple Processes

We introduce a few simple sites for message handling in this section. The simplest such site is one that copies items from an input channel to an output channel forever; it is shown schematically in Figure 8.1.

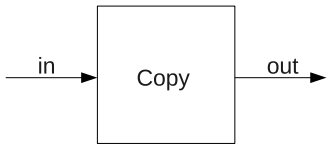

Figure 8.1: A copy Process

The `copy` site has been defined in Section 5.4.3 (page 139) and reproduced here.

```
def copy(read,write) =
    read() >x> write(x) >> copy(read,write)
```

where a typical call to `copy` is of the form `copy(in.get,out.put)`, for channels `in` and `out`. This interface restricts the access rights of `copy` to only read from its input channels and only write to its output channels. (In fact, `read` and `write` could be arbitrary sites, not necessarily methods on channels, and `copy` merely receives data from one site and sends it to the other site perpetually.)

A closely related site is a `multiplexor` that copies from many input channels to a single output channel. We have seen one such site in Section 4.5.6 that copies values from several input channels in a round-robin fashion to a single output channel. The round-robin computation may block inputs from being read if one channel is permanently empty. We design a fair multiplexor below that reads every input from every channel.

Below, site `mux` reads inputs from a list of channels, given by its first argument, and writes to the channel specified in the second argument. Accesses to all channels are restricted according to the access rights convention.

```
def mux([],write) = stop
def mux(read:reads, write) =
    copy(read,write) | mux(reads, write)
```

In the second clause, there is a copy computation for each input channel. If an input channel has a data item, it will be eventually copied over to the output channel. In this sense, `mux` implements a "fair merge" , or interleaving, of the contents of the input channels in arbitrary order.

In many practical applications, such as in sending requests from a multitude of clients to a server, a multiplexor may need to append the identity of the client

to the request. Typically, the request from a client already includes the client identity; otherwise, the multiplexor writes a tuple of the channel name and its content to the output channel.

Analogous to a multiplexor is a demultiplexor, `demux`, that copies values from one input channel to many output channels. Below, the first argument of `demux` specifies the input channel and the second argument the list of output channels to which every data item must be copied. The auxiliary site `fanout` writes its first argument's value on all the channels in the list given by its second argument, and then publishes a signal.

```
def demux(read,writes) =
  def fanout(x , [])           = signal
  def fanout(x , write:writes) =
    (write(x), fanout(x,writes)) >> signal

read() >x> fanout(x,writes) >> demux(read, writes)
```

A demultiplexor may, more generally, write the values on a specific set of channels depending on each input data value. The definition of `demux` can be easily modified to accommodate this generality. For example, the following site directs input `x` to one of the two output channels, given below by `write` and `write'`, depending on the boolean result returned by a `test` site on `x`.

```
demux'(read,test,write,write') =
      read() >x> test(x) >b>
      (if b then write(x) else write'(x) ) >>
      demux'(read,test,write,write')
```

Next, we show a site that merges inputs from two channels into a single output channel according to some order relation. Assume that the data values in each input channel appear in some total order. The output channel carries the merged sequence according to the given order. Below, we assume that the data values are integers. The first two arguments of `merge` refer to input channels and the third argument to the output channel. The auxiliary site `merge'` takes the last two data values read from each of the channels, outputs the smaller one, reads the next value from the appropriate channel and repeats these steps forever.

```
def merge(read,read',write) =
  def merge'(x,y) =
      if x <= y then
             write(x)  >> read()  >x> merge'(x,y)
      else ( write(y)  >> read'() >y> merge'(x,y) )

(read(), read'()) >(x,y)> merge'(x,y)
```

The `merge` site defined in Section 8.2.4 under "Hamming Sequence" differs from the one shown here in that the former discards duplicate values from different channels.

8.3.2 Transducer

A transducer site, in its simplest form, reads a data value from its input channel, applies some transformation to it that produces a single result, writes the result on its output channel, and repeats these steps forever. Such a site is given below, where its first and last arguments refer to the input and output channels and the middle argument is a site that transforms the data.

```
def xduce(read,f,write) =
    read() >x> write(f(x)) >> xduce(read,f,write)
```

The `copy` site of the previous section is a special case where `f` is the identity function.

Site `xduce` is memoryless; each input is treated independent of all other inputs. We consider a few variations in which the transducer remembers (part of) the history of its inputs. First, we consider a site `prefixSum` that writes the sum of all the inputs it has read so far as its output, starting after it reads its first input. The auxiliary site `sum` outputs the sum of all the items read so far plus its argument value with which it was initially called.

```
def prefixSum(read,write) =
  def sum(s) = read() >x> write(s+x) >> sum(s+x)

sum(0)
```

A transducer may not produce an output for each input. Below, site `mean` repeatedly produces the mean value of a block of n consecutive numbers, $n > 0$, from the input, so, a single value is output for n input values. The auxiliary site `avg` reads all the items for a single block and outputs its mean. It has arguments `(s,k)` where `s` is the sum of the values read so far in the current block and `k` the number of items remaining to be read in the block.

```
def mean(read,write,n) =
  def avg(s,0) = write((s+0.0)/n)
  def avg(s,k) = read() >x> avg(s+x,k-1)

avg(0,n) >> mean(read,write)
```

Expression `(s+0.0)/n` in the first clause of `avg` ensures that the division is performed over floating point numbers.

8.4 Simple Networks

A process network [23] consists of a set of *processes* (also called *actors* in the literature [21, 3]) that are connected by a set of channels. The processes compute autonomously, and receive and send data (called *messages*) along the channels incident on them. In many networks, the set of processes, channels and their interconnections are static, but in more general networks the interconnection structure may evolve over time. In most cases, each channel has a single reader and a single writer process, that call its `get` and `put` methods respectively. A

static network can be depicted schematically by a directed graph where a node is a process, an edge a channel and the direction of the edge denotes the direction of data flow. An edge `e` may be directed to another edge `e'` to denote that writing to `e` is same as writing to `e'`.

Process network is an important paradigm in programming of concurrent systems. A network can be implemented in a distributed fashion: a process can be implemented by a physical processor and a channel by a physical channel linking the processors. Networks can also be efficiently implemented on multi-core architectures where a channel is implemented as a buffer in some memory.

Many programming tasks can be naturally decomposed into a set of nearly independent subtasks each of which can be programmed as a process; the data communicated among the subtasks travel along the channels. The decomposition results in a modular program structure that is both easier to design and comprehend. A network can also be treated as a process that is embedded within a larger network, leading to hierarchical and recursive network constructions. Finally, a process network can be depicted pictorially, which helps in understanding the task partitioning and data flow.

We study process networks in this section mainly for their use in distributed and multi-core implementations. We show how to construct networks in a systematic fashion, using site instances for processes and channel instances for channels. We need only the parallel and sequential combinators of Orc in representing a network. (We also use **`val`** for instantiating channels which is translated to a program fragment using the pruning combinator.) We use unbounded, error-free channels in our examples, but bounded or unreliable channels, or rendezvous-based communication (see Section 9.1.1) can also be programmed in Orc.

In Section 8.5, we build networks in which arbitrarily many processes are dynamically initiated, interrupted, resumed or terminated. The networks may be structured in a hierarchy where a process itself may be a network to any arbitrary depth, and connections among network components are established either statically by naming channels explicitly, or by sending a channel name as a data item as in the π-calculus [38]. The networks may also be defined recursively.

Instantiating multiple channels The following definition is useful for instantiating a set of channels; it creates a list of channels of specified length each of which is then given a name.

```
def listofChannel(0) = []
def listofChannel(n) = Channel() : listofChannel(n-1)
```

To create 3 channels, called `ch1`, `ch2` and `ch3`, we need only write

```
val [ch1,ch2,ch3] = listofChannel(3)
```

instead of enumerating each channel individually, as in

```
val ch1 = Channel()
val ch2 = Channel()
```

```
val ch3 = Channel()
```

8.4.1 Translating Orc Programs

We show a few Orc program templates that can be translated directly to process networks.

Orc Sequential Combinator Consider an Orc definition

```
def f(x) = g(x) >y> h(y)
```

where both `g` and `h` publish exactly one value for each input and neither includes a recursive call to `f`. It is required to apply `f` to all values in an input channel and publish the outputs on a given output channel in the order in which the inputs are received.

We introduce an intermediate channel `int` so that a transducer corresponding to `g` reads from the input channel and writes to channel `int`, and a transducer corresponding to `h` reads from `int` and writes to the output channel; see Figure 8.2.

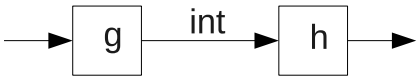

Figure 8.2: Implementing Orc Sequential Combinator

```
def seqNet(read,g,h,write) =
val int  = Channel()

  xduce(read,    g, int.put)
| xduce(int.get, h, write)
```

This is an example of a simple 2-stage pipeline. A multi-stage pipeline can be constructed by repeated application of this paradigm. Section 8.4.3 (page 222) shows how multiple copies of `g` and `h` can be used in a network so that concurrent processing is applied not merely across the two stages of the pipeline, but also for different input data items.

Fork-Join Consider an Orc definition

```
def f(x) = (g(x) , h(x))
```

where both `g` and `h` publish exactly one value for each input and neither includes a recursive call to `f`. It is required to apply `f` to all values in an input channel and publish the output tuples on an output channel in the order of inputs.

We build a network using component processes from Section 8.3.1. First, we employ the `demux` process from Section 8.3.1 to continuously read from the

input channel using `read` and copy their values into two intermediate channels, `intin` and `intin'`. Next, we have two transducer processes, for `g` and `h`, that read their inputs from `intin` and `intin'` and write the results to `intout` and `intout'`, respectively. Finally, a `tupler` process combines the inputs from `intout` and `intout'` and writes tuples to the output channel of the network. The schematic of the computation is shown in the network of Figure 8.3

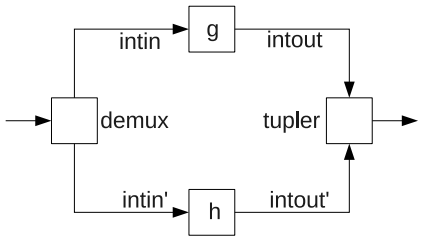

Figure 8.3: Implementing Fork-Join

The corresponding Orc program is given by:

```
def tupleNet(read,g,h,write) =

  def tupler(read, read', write) =
    write((read() , read'() )) >>
    tupler(read, read', write)

val [intin, intin', intout, intout']  = listofChannel(4)

  demux(read, [intin.put, intin'.put])
| xduce(intin.get , g, intout.put )
| xduce(intin'.get, h, intout'.put)
| tupler(intout.get, intout'.get, write)
```

If-Then-Else Consider an Orc definition

```
def f(x) = if test(x) then g(x) else h(x)
```

where `test(x)` publishes a boolean value, both `g` and `h` publish exactly one value for each input, and none of these sites includes a recursive call to `f`.

A simple translation would use a process to implement each of `test`, `f` and `g`. The process corresponding to `test` will have two output channels, one directed to `f` and the other to `g`; every input x for which `test(x)` is true is sent to `f` and the other inputs to `g`. There will be a `collector` process that merges the outputs of `f` and `g` and outputs the results on the output channel of the network.

This simple solution does not preserve the order of inputs at the output since the collector process is an unordered merge. We modify this solution as follows to fix this problem. Inputs to the network are assigned strictly increasing

integers, called *sequence numbers*, that become part of each input. The collector process applies ordered merge of Section 8.3.1, slightly modified to accommodate the tuple data structure. The program is shown below and the process network in Figure 8.4.

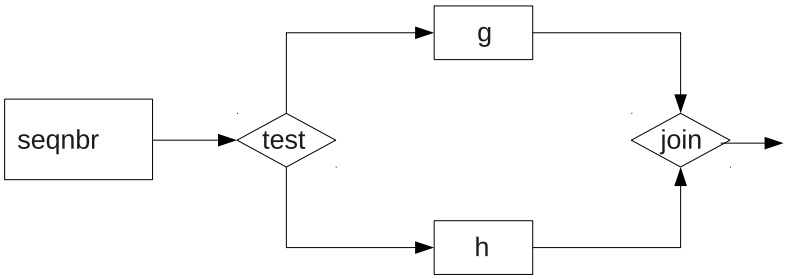

Figure 8.4: Implementing If-Then-Else

```
def ifthenelse(read,test,f,g,write) =

  def seqnbr(read,write) =
    def sequence(i) =
      read() >x> (write(i,x)) >> sequence(i+1)

  sequence(0)

  def fork(read,write,write') =
    read() >(i,x)>
    (if test(x) then write(i,x) else write'(i,x) ) >>
    fork(read,write,write')

  def transduce(read,s,write) =
    read() >(i,x)> (write(i,s(x))) >> transduce(read,s,write)

  def collector(read,read',write) =
    def merge'((i,x),(j,y)) =
        if i < j then
           write(x)  >> read()  >(i,x)> merge'((i,x),(j,y))
        else
           (write(y) >> read'() >(j,y)> merge'((i,x),(j,y)))

  (read(), read'()) >((i,x),(j,y))> merge'((i,x),(j,y))

val [ch1,ch2,ch3,ch4,ch5] = listofChannel(5)

  seqnbr(read,ch1.put)
| fork(ch1.get,ch2.put,ch3.put)
| transduce(ch2.get, f, ch4.put)
| transduce(ch3.get, g, ch5.put)
```

```
  | collector(ch4.get,ch5.get,write)
```

8.4.2 Task Decomposition

We consider a problem, due to Conway [7], mainly for its historical significance. This was one of the first problems posed and solved using coroutines, a precursor of process networks. It refers to many ancient devices, such as punch card input.

A sequence of cards, each with 80 symbols, are to be read from a card reader and the sequence of symbols in the cards to be printed on a line printer that prints lines consisting of 125 symbols. The sequence of input symbols are modified according to the following rules: (1) an extra white space is inserted at the end of each card, (2) a pair of consecutive asterisks (**) are replaced by a single ↑; more precisely, a sequence of $2 \times k$ asterisks are replaced by k ↑s and $2 \times k + 1$ asterisks by k ↑s followed by an asterisk, (3) the end of input is denoted by a special symbol # in a card column; this symbol and all following symbols are to be ignored, and (4) the last output line is padded at the end by white spaces, if needed, to fill the line.

A concise solution appears in Hoare [23] using his notation for communication sequential processes. The essence of the solution is to decompose the problem into 3 simpler tasks that are structured as a 3-stage pipeline. The processes corresponding to the tasks are `disassembler`, `squasher` and `assembler`. The `disassembler` process reads each input card as a list of 80 symbols and outputs each symbol, inserting an extra white space at the end of each card, until a # is detected; it outputs the #. The `squasher` process reads inputs from the `disassembler` and squashes pairs of asterisks to an ↑ and outputs the resulting sequence of symbols. The `assembler` process reads each symbol from `squasher` and outputs each line as a list of 125 consecutive symbols until it detects a #; then it inserts white spaces to fill the current line and outputs the result. The translation of this strategy to an Orc program is straightforward.

8.4.3 Load Balancing

Consider a transducer whose computation is time-intensive for each input. If its inputs arrive at a high rate, its computation would be unable to keep up with the arrivals. To process the inputs quickly, we may implement several copies of the transducer, where each copy reads from the input channel, applies the required transformation and writes the result on the output channel. A schematic of such a network is shown in Figure..; the corresponding program is shown below.

```
xduce(read, f, write) | xduce(read, f, write)
```

The program, though simple, has one major drawback; it does not preserve the order of the input at the output. To overcome this problem, we introduce a `distribute` process that divides the input stream among the two transducers, sending input items alternately to the two copies of the transducer. Also, a

`collect` process collects the outputs of the transducers alternately and writes them on the output channel. The process network is shown in Figure... and the program appears below.

```
def loadBalance2(read,f,write) =

  def distribute(read, intwrite, intwrite') =
    read() >x> intwrite(x)  >>
    read() >x> intwrite'(x) >>
    distribute(read, intwrite, intwrite')

  def collect(intread, intread', write)     =
    intread()  >x> write(x)  >>
    intread'() >x> write(x)  >>
    collect(intread, intread', write)

  val [inch,inch',outch,outch']  = listofChannel(4)

  distribute(read, inch.put, inch'.put)
| xduce(inch.get,  f, outch.put)
| xduce(inch'.get, f, outch'.put)
| collect(outch.get, outch'.get, write)
```

A more elaborate strategy is to have each transducer read an input whenever it is free. Each input carries a distinct sequence number (see the translation of If-Then-Else in Section 8.4.1), the collector merges the outputs according to their sequence numbers and writes the merged sequence on the output channel. Further, the number of transducers to be created could be specified as a parameter of the program. In Section 8.5, we show how to create a network with any specified number of copies of a transducer.

8.4.4 Packet Reassembly

We have seen a relatively trivial program in Section 8.3.1 for copying items from an input channel to an output channel. Here, we consider a variation where the items in the input channel may be slightly out of order, and the copying program has to reestablish the proper order. This is a problem of considerable importance in network protocols: a long message, such as a video file, is typically broken up into packets that are sent as individual messages to a recipient; the packets may not arrive in the same order in which they were sent; therefore, the recipient has to assemble the packets in the correct order to recreate the message.

A sender divides a message into packets and assigns consecutive integers as *sequence numbers* to the packets, starting with sequence number 0. A packet is thus a tuple (n, v) where n is its sequence number and v its content. The packets received by the recipient may not have retained their original order; so, the consecutive packets read by the receiver may be out of order. However, we do not expect the packets to be completely shuffled. Let p_i be the position of the packet with sequence number i in the received sequence. In a flawless

transmission, $p_i = i$. For realistic transmissions,

$$|i - p_i| \leq k, \text{ for some small positive integer } k \qquad \text{(Shuffle Constraint)}$$

We call k the shuffle *radius* and $2 \times k$ the shuffle *diameter*. The packet reassembly problem is to output the contents of the packets in order of their sequence numbers. Henceforth, we use symbol d for the shuffle diameter, $2 \times k$.

From the shuffle constraint we have $i - k \leq p_i \leq i + k$.

$p_i \leq i + k$	from the shuffle constraint	
$i + k \leq p_{i+2\times k}$	replacing i by $i + 2 \times k$ in $i - k \leq p_i$	
$p_i \leq p_{i+d}$	combining above two, using d for $2 \times k$	(S)

Since no two packets occupy the same position in the received sequence, $p_i \neq p_{i+d}$; hence, and $p_i < p_{i+d}$. Therefore, the packets whose sequence numbers are identical modulo d are received in order.

The program, below, contains two site definitions, `input` and `output`, that are executed as concurrent threads. The `input` thread receives packets and stores them internally, as follows. There are d internal channels numbered 0 through $d - 1$. A received packet with sequence number n is stored in the channel numbered n modulo d (written as $n\%d$). From (S) above, the packets in any internal channel are in order of their sequence numbers. The `output` thread scans the internal channels forever in cyclic order starting at the channel numbered 0. In each scan an item is removed from the corresponding channel and output, and the next channel is then scanned. If the channel being scanned is empty, the `output` thread merely waits until a packet is stored in that channel by the `input` thread.

```
def reassembly(read,write,d) = -- Shuffle diameter d, d > 0

val ch = Table(d, lambda(_) = Channel())

def input() = read() >(n,v)> ch(n%d).put(v) >> input()

def output(i) = -- scan internal channel i
   ch(i).get() >v> write(v) >>  output((i+1)%d)

input() | output(0)
```

The sizes of the internal channels depend on the relative speeds of `input` and `output` sites. Assuming that both sites are able to operate at about the same speed, each internal channel will rarely have more than a single packet.

Observe that for a perfect input channel, d is zero, and the program will break down. To remedy the problem, define d to be $2 \times k + 1$. The inequalities in (S) still hold, and, hence, the program is still valid.

A variation of this problem that is more realistic in practice is that a few packets may actually violate the shuffle constraint and arrive too late. It is then easy to modify the solution with time-out to account for such "straggler" packets; the `input` thread simply asks for retransmission on time-out.

8.4.5 A network of transducers; Variance computation

Consider a sensor that produces a reading of some physical device, say the temperature inside an industrial furnace, at regular intervals. The sensor readings are expected to be almost identical over time, but any significant deviation has to be checked out by a human controller. The controller could look at each sensor reading. But a better strategy is to look at the variance over n readings, where n is a given parameter. We show a process network that computes the variance for each block of n readings and outputs the value on a channel. The computation is similar to, though more elaborate than, the computation of the mean, shown in Section 8.3.2.

For a non-empty data set X, $E(X)$ is the mean, i.e., the sum of the elements of X divided by the size of the data set. The variance is given by the formula $E(X^2) - E(X)^2$, where X^2 is the data set obtained by squaring each element of X. The form of the expression suggests a network for its computation. Define two process networks, `sqmean` and `meansq`, to compute $E(X^2)$ and $E(X)^2$, respectively, over data sets of size n. A `subtract` process, similar to `tupler` of Section 8.4.1, collects $E(X^2)$ and $E(X)^2$ from `sqmean` and `meansq`, computes $E(X^2) - E(X)^2$ and outputs the result. The networks `sqmean` and `meansq` are very similar; in `sqmean`, the inputs are first squared and then fed to the network to compute the mean, whereas these two processes are applied in reverse order in `meansq`. Both `sqmean` and `meansq` are examples of 2-stage pipeline described in Section 8.4.1. The definitions of sites `demux` (from Section 8.3.1) and `mean` (from Section 8.3.2) are repeated here for completeness.

```
def variance(n,read,write) =

  def squarer(read,write) = xduce(read,lambda(x) = x*x,write)

  def demux(read,write,write') =
    read() >x> (write(x), write'(x)) >>
    demux(read,write,write')

  def mean(read,write) =
    def avg(s,0) = write((s+0.0)/n)
    def avg(s,k) = read() >x> avg(s+x,k-1)

    avg(0,n) >> mean(read,write)

  def sqmean(read,write)  = -- square the mean
    val c = Channel()
    mean(read,c.put) | squarer(c.get,write)

  def meansq(read,write)  = -- mean the square
    val c = Channel()
    squarer(read,c.put) | mean(c.get,write)

  def subtract(read,read',write) =
    (read(),read'()) >(sqm,msq)>
```

```
    write(msq-sqm) >> subtract(read,read',write)

  val [ch1,ch2,ch3,ch4]  = listofChannel(4)

    demux(read,ch1.put,ch2.put)
  | sqmean(ch1.get,ch3.put)
  | meansq(ch2.get,ch4.put)
  | subtract(ch3.get,ch4.get,write)
```

8.5 Regular and Dynamic Networks

For the process networks we have seen so far the interconnection structure has to be enumerated explicitly; we call these *enumerated* networks. In such networks, the component processes are often heterogeneous. Enumerated networks are typically small. By contrast, large networks contain a number of similar processes or have regular connection structures. For example, an n-stage pipeline will have a number of transducers connected in a linear fashion; the process behaviors are similar, though not identical because they will typically apply different transformations to data. Such networks can be described succinctly using recursion where a component process may be a network. Removing the distinction between process and network is a great convenience, both for conceptual clarity as well as for hierarchical and recursive designs. We show a number of examples of such networks in this section.

8.5.1 Pipeline

We construct an n-stage pipeline of transducers below. Site `pipe(read,fs,write)` has the structure of `xduce` from Section 8.3.2 except that `fs` is a non-empty list of sites. The sites in `fs` constitute the pipeline in the order given in the list.

```
def pipe(read, [f],   write) = xduce(read, f, write)
def pipe(read, f: fs, write) =
  val int = Channel()
  xduce(read, f, int.put) | pipe(int.get, fs, write)
```

The entire pipeline may be regarded as a single process in the hierarchical construction of a network. For example, we may extend load balancing (see Section 8.4.3) where each `xduce` process is replaced by a pipeline.

8.5.2 Multi-copy Load Balance

We construct a generalization of the load-balance network of Section 8.4.3. Below, site `loadbalance(n,read,f,write)` instantiates a network in which `n` copies of a transducer for site `f` read items from an input channel using `read` and write their outputs to the output channel using `write`. We employ the sites

`distribute` and `collect` defined in Section 8.4.3 to distribute the computation among the n copies and collect the outputs in the right order.

The simplest solution is to define the network recursively based on the value of n. For $n = 1$, the network is simply a transducer for `f`. For the general case, we use the program from Section 8.4.3, except that each copy of the transducer is replaced by a network of size $n/2$. For the moment, assume that n is a power of 2 so that the division is perfect.

```
  def loadBalance(1,read,f,write) = xduce(read,f,write)

  def loadBalance(n,read,f,write) =

    def distribute(read, intwrite, intwrite') =
      read() >x> intwrite(x)  >>
      read() >x> intwrite'(x) >>
      distribute(read, intwrite, intwrite')

    def collect(intread, intread', write)     =
      intread()  >x> write(x)  >>
      intread'() >x> write(x)  >>
      collect(intread, intread', write)

    val [inch,inch',outch,outch']  = listofChannel(4)

  -- Goal expr. Two subnetworks in place of two xduce.
    distribute(read, inch.put, inch'.put)
  | loadBalance(n/2, inch.get,  f, outch.put)
  | loadBalance(n/2, inch'.get, f, outch'.put)
  | collect(outch.get, outch'.get, write)
```

For values of n that are not powers of 2, use the ceiling and floor of $n/2$; in Orc these are $n/2$ and $n - n/2$, respectively.

The constructed network essentially consists of a pair of trees that are joined at their terminal nodes; see Figure ... Transducers are located at the terminal nodes. One tree is a distribution network that routes an input item from the root to a transducer at a terminal node and the other a collector network that collects the outputs of the transducers at its root. Therefore, an input is copied about $\log n$ times before it reaches the destination transducer and similarly an output of a transducer is copied about $\log n$ times before it becomes an output of the network. These are unacceptable costs for large n. So, we devise a network that distributes and collects directly to and from the transducers, using an array of channels.

Below, `ch` and `ch'` are arrays of n channels each. Transducer `i`, represented by `xducer(i)`, has `ch(i)` as its input channel and `ch'(i)` as its output. Site `distribute` distributes the input items among `ch` in a round-robin style; site `collect` collects the outputs from `ch'` similarly. The network consists of `distribute`, `collect` and the transducers.

```
  def loadBalance'(1,read,f,write) = xduce(read,f,write)
```

```
def loadBalance'(n,read,f,write) =

  val ch  = Table(n, lambda(_) = Channel())
  val ch' = Table(n, lambda(_) = Channel())

  def distribute(i) =
    read() >x> ch(i)?.put(x) >> distribute((i+1) % n)

  def collect(i) =
    ch'(i)?.get() >x> write(x) >> collect((i+1) % n)

  def xducer(i) =
    ch(i)?.get() >x> ch'(i)?.put(f(x)) >> xducer(i)

  distribute(0) | upto(n) >i> xducer(i) | collect(0)
```

8.5.3 Networks computing recursively defined functions

The networks defined so far compute non-recursively defined functions over a data stream. We show how to compute a recursively-defined function, Fibonacci, on a stream of argument values. The network structure mirrors the recursive definition. We emphasize yet again that such a solution is not recommended for actually computing the Fibonacci function. There are more efficient alternatives. The example merely illustrates how a definition can be systematically translated to a network.

We define a network, `fibnet(N,read,write)`, where site `read` continually reads the input arguments of Fibonacci and `write` outputs the values of the Fibonacci function at those arguments, retaining the input order. Value of `N` is the maximum value of any input number; it is necessary to specify a maximum value so that the entire network can be constructed before the computation begins (we show a network in Section 8.5.4 whose structure evolves as the computation proceeds . A similar technique can be used to eliminate N as an explicit parameter). For $N = 0$ and $N = 1$, the network consists of a single process that outputs the result immediately. For higher values of N, The network consists of 4 processes, `reader`, `writer` and and two copies of `fibnet`, where the component `fibnet` processes have $N - 1$ and $N - 2$ as arguments.

The `reader` process continuously reads input x, writes x on a channel, called `local` below, that is directed to the `writer` and and writes $x - 1$ and $x - 2$ on the channels directed to the component `fibnet` processes. The `writer` process continuously reads input x from `local`, outputs x provided it is at most 1, otherwise, waits to receive the outputs of the component `fibnet` processes, adds them and outputs the result. The order of inputs is preserved at the output, which can be shown using an inductive argument on N.

```
def fibnet(0,read,write) = copy(read,write)
```

```
def fibnet(1,read,write) = copy(read,write)

def fibnet(n,read,write) =

  val [in1,in2,out1,out2,local] = listofChannel(5)

  def reader() =
    read() >x> local.put(x) >>
    (if x :> 1 then (in1.put(x-1) , in2.put(x-2))
     else signal
    ) >>
    reader()

  def writer() =
    local.get() >x>
    (if x <= 1 then write(x)
     else ( (out1.get(), out2.get() ) >(x,y)> write(x+y))
    ) >>
    writer()

  reader()
| fibnet(n-1, in1.get, out1.put)
| fibnet(n-2, in2.get, out2.put)
| writer()
```

We can reduce the size of the network by having just one `fibnet` network as a component, instead of two. The `reader` process sends both $x-1$ and $x-2$ to the component `fibnet` and the `writer` reads two outputs of that component corresponding to the Fibonacci values of $x-1$ and $x-2$.

8.5.4 Dynamic Networks

The networks described so far have been *static*; the network structure is determined before any input is supplied to it and the course of the computation does not alter the network structure. In practice, network structures are often *dynamic* in that new processes are added and existing processes deleted during a computation. Deletion of a process is a difficult problem in general. The simplest deletion procedure is to stop supplying any data to the process, consume all its outputs and then remove the process (for garbage collection). It is difficult to ensure that a process has output all it has to output at any stage in a computation.

Here, we show how to add new processes during a computation. A process is easy to create: instantiate a site by calling it with appropriate parameters and create the channels linking this process to the existing processes in the network. We choose a relatively simple combinatorial problem to illustrate process creation, computation of *all* prime numbers by using the "sieve of Eratosthenes". At any stage the network would have produced n prime numbers, and it has n sieve processes connected as a pipeline, $n \geq 0$. The i^{th} sieve process filters the

numbers it receives by dividing them by the i^{th} prime; the ones that are divisible are discarded and the others are output on its output channel. The very first sieve receives a stream of candidates for prime numbers from a process called `cand` below, and the last sieve is connected to a `manager` process that outputs the prime numbers and creates new sieve processes as needed. The number of sieves in the network is unbounded. The structure of the network is shown in Fig...

It follows from this description that any number output by the i^{th} sieve is relatively prime to the first i primes. Consequently, the first output of the i^{th} sieve is the $(i+1)^{th}$ prime number. The `manager` reads the first number output by a sieve, it outputs this number on a specified channel `primes`, and then creates the next sieve process. It can be shown that the sequence of numbers written on `primes` is the sequence of prime numbers.

```
def Eratosthenes(primes) =

  def cand(write) = -- output 2 and positive odd integers.
    def cand'(i) = write(i) >> cand'(i+2)
    write(2) >> cand'(3)

  def sieve(read,d,write) = -- sieve using prime number d.
    read() >i>
    (if i%d = 0 then signal else write(i) ) >>
    sieve(read,d,write)

  def manager(read,write) =
  {- read a number; it is the next prime, output it,
     and create a sieve for it. -}

    val ch = Channel() -- output channel of the new sieve

    read() >i> write(i) >>
    (  sieve(read,i,ch.put)
     | manager(ch.get,write)
    )

  -- Goal of Eratosthenes(primes)
  -- initially, there is no sieve, only cand and manager.

  val candch = Channel() -- carries 2 and all odd integers
  cand(candch.put) | manager(candch.get, primes.put)

-- Usage of Eratosthenes: channel p will contain the primes

val p = Channel()

Eratosthenes(p)
```

8.6 Concluding Remarks

We have studied the process network style of programming in which the only interaction among components is through message passing over channels. A channel, though a mutable data structure, supports disciplined access to its data. So, it is simpler to develop and reason about the programs written in this style provided a given problem can be solved in this manner. In many situations other kinds of process interactions are essential. For instance, low-level synchronizations in operating system kernels are often implemented by semaphores rather than message communications. Processes, being autonomous, can not interrupt other processes, nor can the halting of a network be detected easily. Interruption and halt detection can be implemented in Orc using the pruning and otherwise combinators, respectively, though these combinators fall outside the channel-based programming paradigm. Large applications require a judicious combination of a variety of programming styles, and channel-based programming is just one such style.

Chapter 9

Synchronization

9.1 Synchronization

Perhaps the most fundamental paradigm in concurrent programming is synchronization among processes. Synchronization of a group of processes forces a process in the group to wait until some or all of the processes in that group have completed some specific activity. One of the simplest forms of synchronization is between a sender and a receiver along a channel; the receiver is forced to wait until the sender has completed putting some data in the channel. In this case and many other cases, synchronization is accompanied by transfer of data or access to shared data. Bounded channel and semaphore provide the basic ingredient for solving a variety of synchronization problems.

We study a number of synchronization problems in this section, among them *rendezvous* in which both a sender and a receiver are forced to wait for each other, *phase synchronization* in which a group of processes execute a sequence of phases in lock-step, the *readers-writers* problem in which there are two sets of processes with different privileges for access to shared data. We show solutions to the classical *dining philosophers* problem that encodes the elements of concurrent resource sharing. We consider encoding *transactions* in Orc that is suitable for the most elementary form of transactional programming.

Callback A technique of particular importance in synchronization is *callback*. It is a mechanism to decouple the site call and the site response. The caller C of a site S specifies, as part of its call, what S must do after its response is ready. Typically, C specifies the name of another site S', or a closure, that S has to execute when it is ready to deliver its response. This frees C to engage in other activities until the response becomes available; in particular, it may read the response at a more opportune moment. It also frees site S to aggregate all the calls it receives and respond to them in a different order. We show the use of callback to solve the 2-party rendezvous and the readers-writers problem.

9.1.1 Rendezvous

A *rendezvous* occurs between a pair of processes that we call the *sender* and the *receiver*. The sender has some data to send and the receiver, as usual, waits to receive the data. In channel-based communication the sender is allowed to deposit the data item in the channel and proceed without concern for whether it will ever be read. A bounded channel restricts the sender's behavior in that it can not indefinitely deposit data without a receiver reading them. Rendezvous imposes an even stricter control: the sender can proceed only after each data that is deposited has been read. Thus, reading and writing of data are synchronized. There is no need for the receiver to send an acknowledgement upon reading; acknowledgement is implicit in the rendezvous protocol. Rendezvous is the means of communication and synchronization in several process calculi [24, 36, 38].

9.1.1.1 2-party Rendezvous

We regard rendezvous as communication over a bounded channel of length 0 where a sender performs a `put` and a receiver a `get`. There may be multiple senders and receivers on the channel, as is typical for any other channel. Also, as is typical, a receiver has to wait for the data to be deposited in the channel before it can read it. But atypically, a sender has to wait until there is a waiting receiver before it can deposit a piece of data. The channel length never exceeds 0 because any deposited item is immediately consumed. We simulate rendezvous using a bounded channel of length 1. Such a channel does not quite solve the rendezvous problem because it allows a sender to proceed without waiting for a receiver. To overcome this problem, we introduce a semaphore that counts the number of waiting receivers. A sender is allowed to deposit a data item only if the semaphore value is positive. Observe that the semaphore value is initially 0 denoting that there are 0 waiting receivers.

```
def class zeroChannel() =
  val s   = Semaphore(0)
  val w   = BoundedChannel(1)

  def put(x) = s.acquire() >> w.put(x)
  def get()  = s.release() >> w.get()


stop
```

Any instance of `zeroChannel()` can be used by a group of senders and receivers for their rendezvous-based communication. Observe that both senders and receivers can not wait without one of them proceeding. Further, the bounded channel in the solution can not remain full if there is a waiting receiver, though there may be a waiting sender and an empty channel forever.

There may be a small time lag between a sender depositing an item in the channel and a waiting receiver reading it, thus allowing the sender to proceed

while the receiver has not completed its operation. This is unavoidable in any implementation that uses an asynchronous protocol.

Pairwise synchronization A special case of rendezvous is pair-wise synchronization without any data transfer. Here, the sender has no data to deposit, it engages in a rendezvous merely for synchronization. Both `put` and `get` operations publish signals upon completion. We may modify the solution above by having `put` write a signal to the bounded channel `w`. But an even simpler solution uses a semaphore in place of the bounded channel, as shown below.

```
def class pairSync() =
  val s   = Semaphore(0)
  val t   = Semaphore(0)

  def put()  = s.acquire() >> t.release()
  def get()  = s.release() >> t.acquire()


stop
```

Observe that `put` and `get` are symmetric operations, i.e., there is no distinction between a sender and a receiver.

2-party Rendezvous using Callback The rendezvous solutions given so far do not quite meet a requirement that we did not specifically pose. To see the problem, suppose that each sender *and* each receiver has a piece of data. A rendezvous results in exchanging their data. The problem is symmetric between senders and receivers. We require that when a set of senders and receivers engage in rendezvous sender s receives the data from receiver r if and only if r receives the data from s.

The class `zeroChannel`, suitably generalized, does not meet this requirement. The following scenario in `zeroChannel` describes the essence of the problem. Receiver r_1 executes `s.release()` and waits to read data and sender s_1 executes both operations in `put(x)`. But before r_1 could read the data receiver r_2 executes both operations in `get()`, thus reading the data from s_1. Even though s_1 had performed the initial part of its rendezvous with r_1 its second part is with r_2. The problem shows up more clearly in the data exchange problem because the requirement given earlier is not met: r_2 gets the data from s_1 though s_1 may not get the data from r_2.

One way to solve the problem is to ensure mutual exclusion among the senders and among the receivers, so that at most one receiver may execute a `get` operation and one sender a `put` operation. Mutual exclusion can be implemented by introducing two new semaphores, one for the `put` and the other for the `get`. We prescribe a more transparent solution using callback.

The following solution encodes data exchange. We introduce an internal channel `q` on which all receivers store callback information. Each receiver first creates two bounded channels, `u` and `v`, of size 1, stores its data in `u` and puts

the tuple `(u,v)` in channel `q`. Then it waits to read data from `v`. A sender get a tuple `(u,v)` from `q`, reads the data from u and stores its data in `v`, thus completing the operation.

```
def rendezvous2() =
  val q = Channel()

  def put(x) = q.get() >(u,v)> v.put(x) >> u.get()

  def get(x)  =
    val u = BoundedChannel(1)
    val v = BoundedChannel(1)
  u.put(x) >> q.put((u,v)) >> v.get()

stop
```

This solution creates two bounded channels for each call to `get`, though the implementation will collect the discarded channels during a garbage collection.

Note: Site put needs only write access to `v` and read access to `u`. However, all the methods of both channels are exposed by storing them in `q`. It is necessary to store only `(u.get,v.put)`, a tuple of closures, in `q`. We adopt such a convention in Section 8.3.1.

9.1.1.2 Multi-party Rendezvous

We consider a generalization of the 2-party rendezvous protocol to n parties, $n \geq 2$. Each party has a specific role in the rendezvous, analogous to the send and receive in a 2-party rendezvous. Each party contributes some data and waits to receive some data on completion of the rendezvous. The contributed data constitute the input list (of length n) of the rendezvous. After all the parties have contributed their data, a given rendezvous function f is applied to the input list to form an output list of length n. The i^{th} party in the rendezvous receives the i^{th} item of the output list, after which it may proceed with its computation.

Multi-party rendezvous is an extremely powerful synchronization mechanism. By choosing different rendezvous functions, a variety of synchronization problems can be solved. The send-receive protocol of the previous section, Section 9.1.1.1, is solved by supplying the sender's contributed data to the receiver and a signal to the sender. In a numerical computation that proceeds in rounds, every party may receive the average value of all the inputs in a round for use in the next round. For certain consensus protocols where the inputs are binary every party may receive the majority value. A synchronized broadcast may be simulated by having every party receive the value supplied by the first party, the broadcaster. In a secret sharing protocol, the inputs can be used to compute a secret which is then handed-out to all parties [45]. A dynamic rank among the parties can be established by sorting the input data and sending out the ranks to each party.

We define a class `Rendezvous` with parameters `n` and `f`, denoting the number of parties and the rendezvous function. The class contains the definition of site `go`; once the the class is instantiated, the i^{th} participant in the rendezvous may call `go(i,x)` where `x` is its input data. It receives a response, its output data, only after all parties have contributed their data and the rendezvous function has been computed. For example, the following call protocol supplies the average value of all inputs in a round to all parties.

```
  def avg([x,y,z])  =
    val av = (x+y+z+0.0)/3
  [av,av,av]

  val rg3 = Rendezvous(3,avg).go

   rg3(0,0)
|  rg3(1,10)
|  rg3(2,2)
```

In this example, `rg3` is the site that each party calls with its identity and data; parties 0, 1 and 2 contribute 0, 10 and 2 as their data. Each alternative in the parallel combinator publishes the average of all contributed values, so, there are 3 publications of 4.0 (that is, (0+10+2+0.0)/3).

Implementation The data structures in the implementation are two tables, `in` and `out`, that are used to hold the input and output lists, respectively. Each table entry is a bounded channel of length 1. Calling `go(i,x)` stores `x` in the i^{th} bounded channel in `in`, provided that entry is empty; then it waits to receive the data from the i^{th} entry of `out`. The computation revolves around `manager` that reads the entries of `in`, applies the rendezvous function, and stores the result in `out`, repeatedly. Auxiliary site `collect` forms the input list from `in`; `distribute` stores the data from the output list in `out`.

```
  def class Rendezvous(n,f) =

  val in  = Table(n, lambda(_) = BoundedChannel(1))
  val out = Table(n, lambda(_) = BoundedChannel(1))

  def go(i,x) = in(i).put(x) >> out(i).get()

{- collect(i) publishes the list of the last i items from in
-}
  def collect(0) = []
  def collect(i) = in(n-i).get() : collect(i-1)

{- distribute(vl,i) puts the first i items of vl
   in the last i positions of out.
-}
  def distribute(_,0) = signal
  def distribute(v:vl,i) =
```

```
    out(n-i).put(v) >> distribute(vl,i-1)

  def manager() =
    collect([],n) >vl> distribute(f(vl),n) >> manager()

manager()
```

9.1.2 Phase Synchronization

Consider execution of `f >> g | f' >> g'`. Executions of the two subexpressions under the parallel combinator are unsynchronized; therefore, the executions of `g` and `g'` may start at arbitrary times on completions of `f` and `f'`, respectively. Let us take `f` and `f'` to be the first *phases* of the corresponding subexpressions, and `g` and `g'` to be the second phase. We may require that the two subexpressions run their phases in lock-step so that a phase in any subexpression may start only after *both* subexpressions have completed their previous phases. This problem arises in code generation in parallelizing compilers where a phase may correspond to an iteration of a loop in the source program and each iteration of the loop may have several parallel subcomputations. The phases must be synchronized so that the values computed during an iteration are made available to all subcomputations in the following iteration.

We have considered this simple version of phase synchronization in Section 2.5.1.8 (page 32). The general version of phase synchronization has a set of n processes that each execute a sequence of phases. Each process calls the synchronizer on completing a phase (using `nextphase()` below) and receives a signal only when it is safe to start executing the next phase, i.e., when all n processes have completed the current phase. The problem is easily solved using multi-party rendezvous where the data to be shared is immaterial. We show a simpler implementation below. The implementation uses two general semaphores, `insem` and `outsem`, both initially 0. A call to `nextphase` releases `insem` and then waits to acquire `outsem`. There is a `manager` that attempts to acquire `insem` n times. Since exactly n processes call `nextphase` and each releases `insem` once , the `manager` can acquire `insem` n times only after all processes have called `nextphase`. At that point, all processes have completed their current phase and the `manager` releases `outsem` n times so that all waiting processes may enter the next phase.

```
  def class phaseSync(n) =

    val insem  = Semaphore(0)
    val outsem = Semaphore(0)

    def nextphase() = insem.release() >> outsem.acquire()

    {- Repeat the execution of f, a closure, i times -}
    def repeat(_,0) =  signal
    def repeat(f,i) = f() >> repeat(i-1,f)
```

```
    def manager() =
      repeat(insem.acquire,n)  >>
      repeat(outsem.release,n) >>
      manager()

  manager()
```

The following usage is typical for 3 processes to synchronize their phases. Each process calls `barrier()` after completing a phase.

```
val barrier = phaseSync(3).nextphase
```

9.1.3 Readers-Writers

The classical readers-writers synchronization problem [10] has a set of processes called *readers* and another set called *writers*. A process, from time to time, needs access to a shared file; the readers "read" the file and the writers "write" to the file. It is required to design an access manager that restricts access to the file as follows: (1) (safety) any number of readers may read the file concurrently, but a writer needs exclusive access; so a writer never operates concurrently with another reader or writer, and (2) (progress) any process waiting to read or write will eventually be granted permission to do so, provided every process that is granted permission eventually completes its execution.

The interface for the processes are as follows. A reader that attempts to read calls `startread()` and a writer calls `startwrite()`. When the caller receives a response, a signal, it has the permission to access the file. On completion of its execution, a reader calls `endread()` and a writer `endwrite()`. Calls to `startread()` and `startwrite()` are blocking, whereas `endread()` and `endwrite()` are non-blocking.

Implementation of the Access Manager A queue (encoded as a channel), called `req`, stores the callback information for all waiting readers and writers in the order in which they request permission for access. The callback information is a pair `(b,s)` where `b` is either `"read"` or `"write"`, and `s` is a semaphore that identifies the requester. An execution of `startread()` creates `s` with initial value 0, stores the tuple `("read",s)` in `req`, and waits to acquire `s`; execution of `startwrite()` is analogous. The semaphore is released by the access manager at the appropriate moment to grant permission to the caller.

Another data structure, called `na`, is a counter that keeps the number of active readers or writers, i.e., the ones that have been granted permission but have not yet completed their executions. This count is important because a writer can be granted access only when all readers have completed their executions, and analogously for permission for the readers. The count is incremented by the access manager when it grants access and decremented by `endread()` and `endwrite()`.

We need the standard factory site `Counter()` to implement `na`. This site instantiates a counter with initial value 0 and provides four methods:

`inc()`:	increments the counter (by 1)
`dec()`:	decrements the counter (by 1), only if the counter value is positive; halts otherwise
`value()`:	returns the current value of the counter
`onZero()`:	returns a signal whenever the counter value next becomes 0

The interface portion of the access manager is as follows.

```
val req = Channel()
val na  = Counter()

def startread() =
  val s = Semaphore(0)
  req.put(("read", s))  >> s.acquire()

def startwrite() =
  val s = Semaphore(0)
  req.put(("write", s)) >> s.acquire()

def endread()  = na.dec()

def endwrite() = na.dec()
```

The access manager loops forever removing the next item from `req` and granting permission to it.

```
def manager() =  grant(req.get()) >> manager()
```

The loop invariant for the manager is that (1) no writer is active, and (2) the value of `na` is the number of active readers. Clearly, this invariant holds initially. When the manager removes a pair `("read",s)` from `req`, given the invariant it can immediately grant permission to this reader, by releasing `s`, without violating the safety constraints. To maintain the invariant, it has to increment `na`.

```
def grant(("read", s)) = {- Reader -} na.inc() >> s.release()
```

When the manager removes a pair `("write",s)` it can grant access to the corresponding writer only when all active readers have completed their executions. It does so by calling `na.onZero()` that returns a signal when `na` becomes zero. At this point it can grant permission to this writer, by releasing `s`. However, to preserve the loop invariant it has to wait until the writer completes its execution. Again, it does so by incrementing na before granting permission to the writer and waiting until `na.onZero()` responds.

```
def grant(("write", s)) = {- Writer -}
  na.onZero() >> na.inc() >> s.release() >> na.onZero()
```

Note that `onZero()` returns a signal when the counter value is 0. But, in general, the counter value may have changed by the time the caller of `onZero()`

```
def class readerWriter1() =
  val req = Channel()
  val na  = Counter()

  def startread() =
    val s = Semaphore(0)
    req.put(("read", s))  >> s.acquire()

  def startwrite() =
    val s = Semaphore(0)
    req.put(("write", s)) >> s.acquire()

  def endread()  = na.dec()

  def endwrite() = na.dec()

  def grant(("read", s)) = {- Reader -}
    na.inc() >> s.release()

  def grant(("write", s)) = {- Writer -}
    na.onZero() >> na.inc() >> s.release() >> na.onZero()

  def manager() =  grant(req.get()) >> manager()

manager()
```

Figure 9.1: Reader-Writer1 Protocol

receives the signal. So, `onZero()` should be called only if the counter value can be changed only by its caller, which is the case in our solution.

The complete protocol is coded as a class and is shown in Figure 9.1. An instance of the class, such as,

```
val rw1 = readerWriter1()
```

creates an access manager. Then a reader may call `rw1.startread()` and `rw1.endread()`, and a writer `rw1.startwrite()` and `rw1.endwrite()`.

Correctness of `readerWriter1` The safety requirements, that the readers may have concurrent access whereas the writers must have exclusive access, is easily seen. We show the progress requirement, that every waiting process is eventually granted permission. First, observe that once the manager removes an entry from `req`, its corresponding iteration is completed in finite time: (1) if the removed entry is `("read", s)`, then `grant(("read", s))` is completed because its body includes no blocking site call, (2) similarly, if the removed entry is `("write", s)`, then `grant(("write", s))` has two blocking calls to `na.onZero()`; however, each call is guaranteed to respond because each process

```
def class readerWriter2() =
  val req = Channel()
  val na  = Counter()
  val (r,w) = (Semaphore(0), Semaphore(0))

  def startread()  = req.put("read")  >> r.acquire()

  def startwrite() = req.put("write") >> w.acquire()

  def endread()    = na.dec()

  def endwrite()   = na.dec()

  def grant("read")  = {- Reader -} na.inc() >> r.release()

  def grant("write") = {- Writer -}
    na.onZero() >> na.inc() >> w.release() >> na.onZero()

  def manager()    =  grant(req.get()) >> manager()

manager()
```

Figure 9.2: Reader-Writer2 Protocol

that is granted permission eventually completes its execution. This observation implies that every entry of `req` is eventually removed and the corresponding process granted permission.

A Refinement The previous solution is generous in its creation of semaphores; each process attempting access to the file creates one. Next, we show a solution that uses only two semaphores, `r` and `w`. Each reader waits for `r` and writer for `w`. The manager releases `r` when it grants access to a reader and `w` for writer. No semaphore is put in `req`; the entries in `req` are only booleans. The complete protocol is shown in Figure 9.2.

The correctness of this algorithm is more subtle. As before, the safety requirement is easy to see. But it is no longer as easy to see progress. For example, a waiting reader whose entry was removed from `req` may not acquire semaphore `r` when it is released; instead a waiting reader behind it in `req` may acquire it. We show progress using the properties of (strong) semaphore implemented in Orc: if semaphore `s` is released eventually as long as a caller on `s.acquire()` is waiting, the caller will eventually acquire `s`.

It is simple to show that the number of waiting readers equals the number of `"read"` entries in `req` as a loop invariant for the manager. Hence, if there is a waiting reader, there is a `"read"` entry in `req`. According to the argument for the previous protocol, each entry of `req` is removed eventually and processed. Therefore, as long as there is a waiting reader, `r` is eventually released. This

satisfies the precondition for a strong semaphore; therefore, a waiting reader will eventually acquire `r`. Similar arguments apply for writers.

A Further Refinement The refined protocol uses `req` all of whose entries are booleans. We show that it is sufficient to maintain a count of the `"read"`s and `"write"`s to encode `req`, and use a coin toss to pick the next entry of `req`. This refinement eliminates unbounded storage. Instead, we have a counter `nwr` for the number of waiting readers and `nww` for the number of waiting writers. We also use a semaphore `nwsem` whose value is the total number of waiting processes; this semaphore is necessary to determine if the manager may start an iteration of its loop.

The algorithm is as before except that `startread()` and `startwrite()` increment `nwr` and `nww` respectively and both also increase `nwsem`. The manager starts an iteration only if `nwsem` is non-zero. Then it picks a boolean value by calling `choose(nwr.value(), nww.value())`. Site `choose` publishes `"read"` if its second argument is 0 (i.e., pick a reader if there are no waiting writers), `"write"` if the first argument is 0, and randomly `"read"` or `"write"` if both arguments are non-zero. The protocol is shown in Figure 9.3.

The correctness of the protocol depends on the fairness of random number generation. As long as there is a waiting reader, the value of `nwr` is positive. So, a call to `choose` publishes either "read" (because `nww` may be 0) or a random boolean. Assuming fairness in the latter, eventually `choose` publishes "read", thus causing `r` to be released. Now, we use the argument from the previous protocol to assert that a waiting reader will acquire `r` eventually given the strong semaphore property.

9.1.4 Dining Philosophers

Dining Philosophers is a classic problem in resource sharing. We show a very simple solution, due to Hoare [23], that solves the problem by limiting the number of processes that can contend for a resource at a time. We also show the encoding of a randomized algorithm due to Lehmann and Rabin [32].

Given is a set of philosophers where each philosopher is a process. The philosophers are arranged in a ring so that each philosopher has a left and a right neighboring philosopher. Placed between each pair of neighbors is a resource called a fork; the two forks next to a philosopher are incident on that philosopher. A philosopher cycles through 3 states, *thinking*, *hungry* and *eating*, in the given order. A thinking philosopher *may* transit to hungry or remain thinking forever, and an eating philosopher `must` transit to thinking; the stimulus for transition is external and is immaterial for our purposes. An eating philosopher must hold both its incident forks, so neighbors are never simultaneously eating.

It is required to design a protocol so that every hungry philosopher eventually transits to eating. A philosopher is allowed to pick up an incident fork provided its neighbor does not already hold it; it may pick only one fork at a time. A simple algorithm for a hungry philosopher is to first pick its left and then its

```
def class ReadersWriters() =
  -- nwr   = number of waiting readers
  -- nww   = number of waiting writers
  -- na    = number of active  processes
  -- nwsem = number of waiting processes

  val (nwr,nww,na) = (Counter(),Counter(),Counter())
  val (r,w)        = (Semaphore(0), Semaphore(0))
  val nwsem        =  Semaphore(0)

  def startread()  =
    nwr.inc() >> nwsem.release() >> r.acquire()

  def startwrite() =
    nww.inc() >> nwsem.release() >> w.acquire()

  def endread()    = na.dec()

  def endwrite()   = na.dec()

  def choose(_,0) = "read"
  def choose(0,_) = "write"
  def choose(_,_) = (if Random(2) = 0 then "read" else "write
      ")

  def grant("read")  = {- Reader -}
    nwr.dec() >> na.inc() >> r.release()

  def grant("write") = {- Writer -}
    na.onZero() >> nww.dec() >> na.inc() >> w.release() >>
    na.onZero()

  def manager() =
    nwsem.acquire() >> choose(nwr.value(), nww.value()) >b>
    grant(b) >> manager()

manager()
```

Figure 9.3: Reader-Writer3 Protocol

right fork. However, if all philosophers are hungry, and they all pick up their left forks in unison then none of them can pick up the right fork and start eating. In fact, there is no deterministic algorithm that is symmetric for all processes.

9.1.4.1 Limiting the amount of contention

Given that there are n philosophers a simple solution is to limit the number of philosophers that may contend for resources to $n-1$. In that case, the deadlock scenario we have described above does not arise; some philosopher will be able to pick up both forks, by an elementary application of the pigeon-hole principle.

Denote the number of available seats at the table by semaphore `seat`. The maximum value of `seat` is $n-1$. The strategy for a hungry philosopher is to first acquire a seat by calling `seat.acquire()`. After acquiring a seat, the philosopher acquires both forks, each a semaphore, in either order. The acquisition of the seat and the forks is encoded in site `pick`. After acquiring the forks, the philosopher eats and on completion of eating, releases the forks and then the seat.

9.1.4.2 Randomized algorithm

In this solution, each hungry philosopher tosses a coin to determine the order in which to pick up the forks. After picking up the first fork if it is unable to pick up the second fork immediately, it relinquishes the first fork and starts all over. It can be shown that with extremely high probability every hungry philosopher will be able to pick up both of its incident forks and thus transit to eating.

We use the program outline from the previous solution except that there is no need for semaphore `seat`. As before, we represent a fork by a semaphore. However, we need an additional operation on this semaphore, besides the usual operations of `acquire` and `release`, to indicate that an `acquire` operation has failed. Operation `acquireD` on a semaphore is never blocked; the caller of `acquireD` either receives a signal immediately denoting that the semaphore has been acquired, or the call halts and the caller receives a negative response if the semaphore can not be currently acquired. This operation is already included for a standard Orc semaphore. It can also be programmed as a class in Orc.

The program in Figure 9.5 follows the outline we have sketched[1].

[1]This program is due to David Kitchin.

```
-- n dining philosophers in a ring with n-1 seats.
def dining(n) =

  -- allow at most n-1 philosophers at the table at a time
  val seat = Semaphore(n-1)

  -- set up the forks
  val forks = Table(n, lambda(_) = Semaphore(1))

  -- Acquire forks a and b
  def pick((a,b)) =
    seat.acquire() >> (a.acquire(), b.acquire()) >> signal

  -- Release two forks
  def drop(a,b) =
    a.release() >> b.release() >> seat.release()

  -- Start a philosopher process with forks a and b
  def phil(a,b) =

    def thinking() =
      Think() -- publish signal on completion of thinking

    def hungry() = pick((a,b))

    def eating() =
        Eat() -- publish signal on completion of eating
      >> drop(a,b)

  -- Goal expression of phil(a,b)
  thinking() >> hungry() >> eating() >> phil(a,b)

  -- Place n philosophers in a ring
  def ring(0) = stop

  def ring(i) =
      phil(forks(i%n), forks(i-1))
    | ring(i-1)

  -- goal expression of dining(n)
  ring(n)

dining(5) -- start 5 dining philosophers
```

Figure 9.4: Seat-Limited Dining Philosophers

```
-- probabilistic solution with n dining philosophers
def dining(n) =

  -- set up the forks
  val forks = Table(n, lambda(_) = Semaphore(1))

  -- Randomly swap two elements
  def shuffle(a,b) = if (Random(2) = 0) then (a,b) else (b,a)

  -- Acquire forks a and b in random order
  def pick((a,b)) =
    shuffle(a,b) >(u,v)>
    (u.acquire() >> v.acquireD() ;
     u.release() >> pick((a,b))
    )

  -- Release two forks
  def drop(a,b) = a.release() >> b.release()

  -- Start a philosopher process with forks a and b
  def phil(a,b) =

    def thinking() =
      Think() -- publish signal on completion of thinking

    def hungry() = pick((a,b))

    def eating() =
        Eat() -- publish signal on completion of eating
      >> drop(a,b)

  -- Goal expression of phil(a,b)
  thinking() >> hungry() >> eating() >> phil(a,b)

  -- Place n philosophers in a ring
  def ring(0) = stop

  def ring(i) =
      phil(forks(i%n), forks(i-1))
    | ring(i-1)

  -- goal expression of dining(n)
  ring(n)

dining(5) -- start 5 dining philosophers
```

Figure 9.5: Randomized Dining Philosophers

Chapter 10

Real-time Programming

We have seen a number of small examples using `Rwait` for time-outs in the previous chapters. We introduce a few more standard sites in connection with real-time programming in this chapter. We use these sites to program a stopwatch in Section 10.2 and an alarm clock in Section 10.3. We program a small interactive real-time game at the end of this chapter.

It is difficult to reason about real-time programs that are also concurrent. A starting point is to analyze such a program under the following simplifying assumptions: (1) execution of `Rwait(t)` consumes t units of time, (2) no other standard site consumes any time, and (3) external sites consume arbitrary amounts, and (4) events occurring at the same time will happen in some arbitrary order. In some cases the specification of an external site may include its temporal behavior and the delays in communicating with it. These specifications are not very robust as they also depend on the network communication speeds. The simplifying assumptions are of course not met in practice, because standard sites do consume time. But as long as those times are sufficiently small compared with wait times in `Rwait`, a reasonably formal proof of correctness can be constructed.

10.1 Standard Sites for Real-time Programming

Orc includes the standard site `Rwait`; `Rwait(t)` publishes a signal after passage of `t` units of real time, where the standard implementation uses a millisecond as the unit of time. `Rtime()` publishes the elapsed time since the start of the program. Both these sites refer to a standard clock.

It is often convenient to allow multiple real-time clocks to start at different times during the program execution even though they all tick at the same rate. Calling factory site `Rclock()` publishes a clock whose initial value is 0. A clock, `clk`, has two methods analogous to `Rwait` and `Rtime`: (1) `clk.wait(t)` publishes a signal after `t` units, and (2) `clk.time()` publishes the elapsed time since the creation of `clk`. It is easy to program `Rclock()`, as shown below.

```
def class Rclock() =
  val s = Rtime() -- record the start time

  def wait(t) = Rwait(t)
  def time()  = Rtime() - s
stop
```

Note that method `wait` is same as `Rwait`; it has been included only in analogy with the standard clock.

10.1.1 Device Controller: Unlocking a Car door

A car door can be unlocked by an owner using a remote key, and, typically, the owner then opens the door within some suitable time, say 30 seconds. If the door is not opened within that time, it can be assumed that the owner either changed her mind or the remote signal was received in error. In that case, the door should be relocked. We show the site that may control this piece of computation.

Site `door_controller` is called wheneever the door is unlocked using the remote key. The parmeters of the site call are: (1) site `opened` that returns a signal as soon as the door is opened, and (2) varaible `t` that specifies the amount of time that must elapse before the door is relocked. Site `door_controller` publishes a boolean, `true` if the door was opened within the specified time and `false` otherwise. The actual relocking is performed elsewhere. This site implements the most basic form of time-out.

```
def door_controller(opened,t) =

  val z = opened() >> true | Rwait(t) >> false
  z
```

10.1.2 Average Response Time

A common computation in a network is to ping a site to compute the round-trip delay through the network. This is easily done by using `Rtime()`. We show a small program that calls a given site `S`, that has no parameters, a specified number of times and computes the average response time. We use site `repeat` where `repeat(S(),n)` calls `S()` consecutively `n` times, for positive `n`, with the first call made immediately and each subsequent call made immediately after receiving the response to the previous call. Site `repeat` appears in Section 2.4.1 (page 28) and in a simpler form in Section 9.1.2 (page 238).

```
def avg_response_time(S,n) =
  Rtime() >v>
  repeat(S(),n) >>
  (Rtime() - v + 0.0)/n
```

The addition of `0.0` in the last line of the program ensures a floating point result.

10.1.3 Bounded-time Computation

The following example illustrates bounded-time computation. Let `approx` be a site that publishes the next, better, approximation in a numerical computation given an initial approximation. Computing the root of a real-valued function using Newton-Raphson method is an example. The goal is to compute the best approximation within a given time.

The program below calls `approx` initially with an approximation `v` and then successively with the last value it returns. The computation continues until some specified time at which point the last available approximation is published. More precisely, if $v_0, v_1, \cdots, v_n$ is the sequence of values where v_0 is the initial approximation, v_{i+1} the publication of `approx` given v_i as input for each i, and v_n the last publication before the time-out ($v_n = v_0$ if approx never publishes), then the goal is to publish v_n.

Site `successive`, defined below, has three formal parameters: site `approx`, the initial approximation `v` and the the time bound for computation `t`. It starts three threads simultaneously to compute: (1) `(b,v')` where `b` is a boolean and `v'` an approximation; if `approx(v)` publishes a value within time `t`, `b` is set true and `v'` set to the published value; if `approx(v)` does not publish within time `t`, `b` is set false and `v'` is set to `v`, (2) `t'` is the elapsed time for the computation of `b`, i.e., either the time taken to receive the response from `approx` or `t` if there is a time-out, and (3) the goal expression that calls `successive` with a reduced time bound for the next approximation if there is any left-over time or publishes `v` if there is none.

```
def successive(approx,v,t) =

  val (b,v') =
      (true, approx(v)) | Rwait(t) >> (false,v)

  val t' = Rtime() >s> b >> t - s

  if  t' :> 0  then successive(approx,v',t') else v
```

A more general scenario of bounded computation is for several threads to compute simultaneously until there is a time-out. At that point, each thread publishes its computed result.

10.1.4 Program Profiling

We showed a program for measuring the execution time of a computation in Section 5.4.2 (page 139) that uses a stopwatch. Here, we show a simpler program using the basic time-based sites. Site `profile` has two arguments, a site `fun` and a value `x` for the argument of `fun`. Assume that `fun` publishes at most one value. Site `profile` publishes a tuple `(v,t)` where `v` is the publication of `fun(x)` and `t` is the elapsed time for this computation.

```
def profile(fun,x) = Rclock() >clk> fun(x) >y> (y,clk.time())
```

Executing `profile(Rwait,100)` publishes `(signal, 100)`, as we would expect; the overhead associated with calls and publications within `profile` are negligible, so they do not affect the elapsed time. We can write the same program more simply using `Rtime()`.

```
def profile(fun,x) = Rtime() >s> fun(x) >y> (y,Rtime()-s)
```

Typically, we are interested in observing the behavior of a site over multiple runs. Site `multiprofile` is similar to `profile` except that it takes a list of arguments and executes the given site for each argument *in sequence*. It publishes a list of tuples, site values and elapsed times, for each argument in the input list.

```
def multiprofile(fun,[])   = []
def multiprofile(fun,x:xs) =
    profile(fun,x) >y> (y : multiprofile(fun,xs) )
```

Observe that `multiprofile` has the same structure as `seqmap` of Section 4.4.2.5; it enforces sequential execution of `fun` for each argument in the input list. This ensures that the running time of one test is published before the next test is begun.

Calling `multiprofile(Rwait,[100,200,300])` publishes

```
[(signal, 100), (signal, 200), (signal, 300)]
```

as expected. A more interesting exercise is to run two different implementations of the same site and compare their performance. Given below are two versions of the Fibonacci site.

```
def fib(0) = 0
def fib(1) = 1
def fib(n) = fib(n-1) + fib(n-2)

def fib'(n) =
  def H(0) = (0, 1)
  def H(n) = H(n - 1) >(x, y)> (y, x + y)
H(n) >(x, _)> x
```

A few calls and the corresponding publications appear below (run on a 2007 vintage desktop).

```
multiprofile(fib,[5,10,15,20,25])

  [(5, 2), (55, 17), (610, 175), (6765, 2245), (75025, 29568)]


multiprofile(fib',[5,10,15,20,25])

  [(5, 0), (55, 2), (610, 2), (6765, 2), (75025, 5)]


multiprofile(fib',[50,100,150,200])
```

```
[(12586269025, 6), (354224848179261915075, 13),
 (99692166771893033862144057602 00, 18),
 (28057117299251014003761193241303867718 9525, 23)]
```

It is known that the execution time of `fib(n)` is nearly proportional to `fib(n)`; the ratio of elapsed time to `fib(n)` for even these small values of `n` (ignoring `n = 5`) range between 0.3 and 0.4. The computation time of `fib'(n)` is linear in `n`, which is more clearly seen in the last test with `fib'` using larger values of `n`.

10.2 Stopwatch

We develop a class that implements a *stopwatch*. we show a very simple version first followed by a more realistic version.

10.2.1 A Simple Stopwatch

A stopwatch is in one of two states, *running* or *paused*, and it displays a *count* in milliseconds when it is paused. Initially, a stopwatch is paused with count 0. There are two available operations: (1) `start()` is allowed only when the stopwatch is paused; it starts the stopwatch running from its current count and immediately publishes a signal, (2) `pause()` is allowed only when the stopwatch is running; it puts the stopwatch in paused state and publishes its current count. Concurrent calls to the methods are not supported.

The implementation below maintains two `Ref` variables: (1) `laststart` has the count on the stopwatch value when `start()` was last called, and (2) `lastpause` has the count when `pause()` was last called. Initially, both variable values are 0. The states are not shown explicitly.

```
def class Stopwatch() =
  val (laststart, lastpause) = (Ref(0), Ref(0))

  def start() = laststart := Rtime()

  def pause() =
    lastpause := lastpause? + (Rtime() - laststart?) >>
    lastpause?

stop
```

In the definition of `pause`, `(Rtime()- laststart?)` is the elapsed time since the last `start`. Therefore, `lastpause? + (Rtime()- laststart?)` is the count shown by the stopwatch when it was last paused plus the elapsed time since it last started, i.e., the current count on the stopwatch.

10.2.2 A Realistic Stopwatch

A useable implementation of a stopwatch requires a more general interface. It should allow `start` operation in the state where the stopwatch is already running, and `pause` in a state where the stopwatch is already paused. Additionally, methods to reset the stopwatch, and determine its status (running or paused) should be provided. Further, concurrent calls to `start` and `pause` may interfere, so the methods should be made atomic.

The implementation supports four methods: `start`, `pause` and `reset`, `isrunning`. Execution of `reset` or `pause` causes the final state to be `paused` starting from any state. Execution of `start` ensures that the final state is `running`. Execution of `isrunning` does not cause a state change. Observe that `start` has no effect in `running` state and `pause` in `paused` state. Executing `reset` in `paused` state does not cause a state change though the count is set to 0. We encode the state explicitly in the boolean mutable variable `running` that is true if and only if the stopwatch is running.

A call to `start` or `reset` publishes a signal. A call to `pause` returns the current count on the stopwatch. A call to `isrunning` returns a boolean, `true` if and only if the state is `running`. As before, `laststart` and `lastpause` are the counts when the last calls to `start` and `pause` were made, respectively. We have `lastpause = 0` initially and also after each call to `reset`. Each method is made atomic by using a semaphore.

```
def class Stopwatch() =

  val (laststart, lastpause) = (Ref(0), Ref(0) )
  val running = Ref(false)
  val sem = Semaphore(1)

  def start() =
    sem.acquire() >>
    ( if running? then signal
      else (running := true >> laststart := Rtime() )
    ) >>
    sem.release()

  def pause() =
    sem.acquire() >>
    ( if running? then
        lastpause := lastpause? + Rtime() - laststart? >>
        running := false
      else signal
    ) >>
    sem.release() >> lastpause?

  def reset() =
    sem.acquire() >>
    running := false >> lastpause := 0 >>
    sem.release()
```

```
    def isrunning() = running? -- single atomic step

  stop

-- Usage: compute the response time for site S()

  val sw = Stopwatch()

  sw.start() >> S() >> sw.pause()
```

10.3 Alarm Clock

We develop a class to simulate an alarm clock with two methods, `set` and `cancel`. An alarm, once set, expires normally after a specified amount of time `t`, like `Rwait(t)`. Unlike `Rwait`, the alarm may be cancelled before its normal expiry by calling `cancel`.

First, we show a single-use alarm clock that is set or cancelled at most once. Next, we show a multi-alarm clock that may have multiple alarms set at different times. The alarms may even overlap in their durations.

10.3.1 Single Alarm

An instance of a single alarm can be set to ring after time `t` by calling `set(t)`. During this time period if `cancel()` is called, then the alarm is cancelled and `set` immediately responds by publishing `false`. If the alarm is not cancelled, then it expires after time `t` and `set` publishes `true` at that time. Each of these methods may be called at most once.

The `set` method in the program below may be interrupted by a call to `cancel`. The program structure closely follows the interruption technique shown in Section 2.5.2.6. The main execution thread in `set` waits to complete the alarm using `Rwait` and simultaneously waits for an interruption from `cancel`. The interruption occurs if `set` can acquire a semaphore, `off`, which `cancel` releases. `cancel` always completes immediately and publishes a signal.

```
  def class single_alarm() =

    val off = Semaphore(0)

    def set(t) =
      val b = Rwait(t) >> true | off.acquire() >> false

      b

    def cancel() = off.release()

  stop
```

```
-- Usage

  val al = single_alarm()
  al.set(8) | Rwait(5) >> al.cancel()
```

The `single_alarm` class is quite fragile. Multiple `sets` or `cancels` on the same alarm may have arbitrary effects. We modify the class so that multiple sets or cancels have no direct effect. The interface is modified such that only the first call to `set` sets the alarm and publishes `true` or `false` as before; subsequent calls to `set` simply halt. A call to `cancel` either publishes a signal if the call has the effect of cancelling the alarm, otherwise the call halts.

First, consider a modification that implements the interface for `cancel`. We use a cell, `cellC`, in place of semaphore `off`. Both `set` and `cancel` attempt to write to `cellC`, `set` attempts to write `true` when the the alarm expires normally and `cancel` attempts to write `false` when there is a call to it. Method `set` publishes whatever is written to `cellC` and `cancel` publishes a signal only if it succeeds in writing to `cellC`. Further, `cellC` ensures that at most one call to `cancel` publishes and all others halt.

Next, once-only writing to a cell, `cellS`, ensures that exactly one call to `set` executes and all others halt, provided `set` is called at all.

```
def class single_alarm() =
  val cellS = Cell()
  val cellC = Cell()

  def set(t) =
    cellS := signal >>
    (val b = Rwait(t) >> cellC := true >> stop | cellC?

    b)

  def cancel() = cellC := false

stop
```

10.3.2 Multi-Alarm Clock

We develop a `multi_alarm` class that has the same two methods, `set` and `cancel`. However, each method has an additional parameter, an `identity` of the alarm. Thus, `set(id,t)` sets the alarm with identity `id` for time period `t`, `cancel(id)` cancels the specified alarm. This class allows setting of multiple alarms simultaneously, and is robust with respect to repeated sets and cancels for the same alarm.

The interfaces are as follows. Call to `set(id,t)` halts if there is already an alarm with identity `id` that has been set; otherwise it sets such an alarm and behaves as before by publishing `true` at normal expiry or `false` if there is premature cancellation. `cancel(id)` halts if there is no alarm with identity

`id`; otherwise it cancels the specified alarm and publishes a signal. An identity may be reused after its expiration for setting an entirely different alarm.

Class `multi_alarm` creates a new instance of `single_alarm` for every alarm that is set. We make use of an identity-set (abbreviated to `idset`), a data structure that maintains the identities of the alarms that are active. Each element of `idset` stores an identity and the (closure of) the associated alarm as a pair. It supports three methods: (1) `insert((p,v))` halts if there is already an alarm whose identity is `p`; otherwise add the pair `(p,v)` to `idset` and publish a signal, (2) `remove(id)` halts if there is no entry in `idset` with identity `id`; otherwise remove the entry from `idset` and publish a signal, (3) `search(id)` halts if there is no entry in `idset` with identity `id`; otherwise publish the alarm associated with `id`. `idset` supports concurrent executions of its methods with the given semantics. We do not develop `idset` as a class here. It is similar in structure to `conc_bst` of Section 7.5.3. That class should be extended by storing a primary and secondary key at each tree node corresponding to an identity and the associated alarm, and including a method to delete nodes from the tree to support `remove`.

The `multi_alarm` class makes use of `idset` and `single_alarm` class developed in Section 10.3.1. Observe that on its termination, the `set` method has to remove the entry with the given identity from `idset`.

```
  def class multi_alarm() =
  val db = idset()

  def set(id,t) =
    val al = single_alarm()
    db.insert((id,al)) >> al.set(t) >b> db.remove(id) >> b

  def cancel(id) =
    db.search(id) >al> al.cancel()


  stop

-- Usage

  val mal = multi_alarm()

    mal.set("a",100) | mal.set("b",200)
  | mal.set("c",20) >> mal.cancel("a")
```

10.4 Response Game

We use the `Stopwatch` class in designing a small game. The ostensible goal of the game is to measure the response time of a player. To do so, the program first displays a decimal digit, called a *seed*, on the terminal. After a few seconds, the program publishes a continuous stream of random decimal digits at an even

rate. The goal for the player is to press a key on the keyboard as soon as she sees the seed in the stream. The elapsed time since the first display of the seed in the stream until the player presses a key is the player's response time. If the player presses the key too early, before the seed appears in the stream, a suitable message is displayed.

The design of the game shows the interactions of concurrent threads in real time. Two different threads are used to publish the stream and await the response from the player. The former thread is terminated as soon as the player responds. Further, a stopwatch is started as soon as the seed is published for the first time in the stream and paused when the player responds.

In the program below, we use two standard sites, `Println` and `Prompt`. Site `Println` has a single argument, a string, that it prints on the terminal and then publishes a signal. Therefore, `Println` can be sequentially followed by some computation that is started following the printing. Site `Prompt`, like `Println`, has a single argument that is a string that it prints on the terminal, but it awaits a response from the keyboard that it reads and publishes. In this game, since the player merely presses the `ENTER` key, `Prompt` merely publishes a signal after receiving the response from the player.

The variables and sites defined in the program are as follows. The main site `game` starts by choosing a random decimal digit for seed `v`. It then prompts the player to press `ENTER` when she sees `v` in the stream. An initial delay, given by variable `id`, pauses execution for a few seconds to allow the player to observe the seed carefully; below, `id` is set to 3 seconds. The random stream of digits is then printed with the delay between printing of successive digits given by variable `dd`; below, `dd` is set to 0.1 seconds. The stream is produced by site `rand_seq` which resembles a `metronome`, defined in Section 2.4 (page 27). Instead of publishing signals as in `metronome`, `rand_seq` publishes random digits. The printing thread in `game` compares every digit with the seed as it is printed and starts a stopwatch in case of a match.

The printing is terminated as soon as there is a response from the user. At that point, the state of the stopwatch is noted in variable `b`, `true` if it is running and `false` otherwise, and the count on the stopwatch in `w`. The program then publishes `w` provided `b` is `true`, that is if the seed had appeared in the stream, or a suitable message if `b` is `false`.

```
{- Response Game -}

val sw = Stopwatch()

-- id is the initial delay in starting a game.
-- dd is the delay in printing digits.

val (id,dd) = (3000,100)

def rand_seq() =
{- Publish a random sequence of digits at interval of dd -}
  Random(10) |  Rwait(dd) >> rand_seq()
```

```
def game() =
{- game() conducts one game and returns a pair (b,w).
   b is true only if the user responds after v is printed;
   then the response time is w.
-}
   val v = Random(10) -- v is the seed for one game

   val (b,w) =

    Rwait(id) >> sw.reset() >> rand_seq() >x> Println(x) >>
     Ift(x = v) >> sw.start() >> stop

  | Prompt("Press ENTER for SEED "+v) >>
    sw.isrunning() >b> sw.pause() >w> (b,w)

{- Goal expression of game() -}
   if b then
    ("Your response time = " + w + " milliseconds.")
   else ("You jumped the gun.")

game()
```

Note that a new thread is started for every digit in the stream following `Println(x)`. If the digit matches the seed `sw.start()` on stopwatch `sw` is started. A seed may be printed multiple times and the stopwatch started multiple times, but subsequent starts have no effect on a running stopwatch. In all cases the the thread for a digit is terminated without publishing. It is possible that two nearby occurrences of the seed in the stream will start two threads that execute concurrently and access the stopwatch. This is permissible because the stopwatch has been designed to accept concurrent calls on its methods.

The program can be modified easily to repeat the call to `game()` after termination of one game. And, it is equally easy to collect statistics over several games and compute the mean and variance of response times.

10.5 Calendar

An Orc program can make use of an external site to determine the absolute time and date. Below, we define a class for this purpose by importing a standard package from the Java library:

```
import class Calendar = "java.util.GregorianCalendar"
```

This creates an Orc class, `Calendar`, that includes all the methods available in the Java class for Gregorian calendar.

For ease of use, we create an Orc class, called `Orc_calendar` and shown in Figure 10.1, that includes a subset of the methods provided by the Java class. We discuss the methods in the Orc class here and in the following sections.

Method `date` publishes a 4-tuple of integers for the current date: year,

month, date of the month and day of the week, in this order. Year is a four digit integer; month is an integer between 1 and 12 inclusive, unlike Java's convention of numbering months from 0 through 11; date of the month is a two digit integer; day of the week is an integer in the range 1 through 7 where Sunday is 1. For example,

```
val cal = Orc_calendar()
cal.date()
```

may publish `(2012, 7, 9, 2)`.

Method `time` publishes a triple of two-digit integers for the current time: hour, minute and second in this order, in 24-hour format. Typically, the second value is a gross approximation.

```
val cal = Orc_calendar()
cal.time()
```

may publish `(11, 42, 31)`, denoting around 17 and half minutes before noon.

Both `date` and `time` make use of the auxiliary method `pub` that takes a calendar instance (actually, the `get` method of the calendar instance) and a string, `"date"` or `"time"`. It publishes the current date or time in the format described earlier.

Observe that a calendar instance is created with each call to `date` and `time` rather than at the time of instantiation of the Orc class. This is essential because having a fixed instance for all method calls will publish the same values no matter when the methods are called during execution of an Orc program.

10.5.1 Keeping Appointments

An appointment is usually specified by an absolute time rather than a delay from the present moment, e.g., ring a bell at noon. Method `sub_time` is used to compute the remaining time to a moment in the next 24 hours. It is called with `(fh,fm,fs)` denoting a future moment in hour, minute and second format. It computes the remaining time from the present moment, `(ch,cm,cs)`, to the future moment in the same format. The computation is carried out in mixed radix arithmetic as follows. First, `(fh-ch,fm-cm,fs-cs)` is computed. Since any of the entries in this tuple may be negative, we recompute the tuple by adding 60 to the component for second if it is negative and subtracting 1 from the minute component, then adding 60 to a negative minute component and subtracting 1 from the hour component, and adding 24 to the hour component if it is negative, as shown in Figure 10.1.

Below, site `waituntil` publishes a signal at the specified time within the next 24 hours given by `(fh,fm,fs)` for the hour, minute and second. It computes the remaining time from the present moment till the specified time,`(x,y,z)` in hour, minute and second, using `sub_time`. It then executes `Rwait(t)` where `t` is computed in milliseconds from `(x,y,z)`; for computing `t` from `(x,y,z)` we employ the site `time_conv` described in Section 4.4.2.3 as an example and repeated here for completeness. The wait time `t` could be 0.

```
def radix_Conv([],[x])              = x
def radix_Conv(_,[])                = 0
def radix_Conv(r:radixes, d:digits) =
  d + r * radix_Conv(radixes,digits)

def time_conv(ds) = radix_Conv([1000,60,60,24,7],ds)

def waituntil(fh,fm,fs) =
  val cal = Orc_calendar()

  cal.sub_time(fh,fm,fs) >(x,y,z)>
  Rwait(time_conv([0,z,y,x]) )
```

There is no counterpart of `sub_time` for date, because there is no simple way to subtract a date from another in Java.

10.5.2 Computing Absolute Dates and Times

It is sometimes required to compute the date that is 25 days from today, a typical computation in sending out a credit card bill. The computation, though straightforward given the current date, is tedious because it has to account for varying lengths of months and leap years. The Java class provides method `add` to add any integer, positive, negative or zero, to any component of a calendar. We include a method `add_date` in `Orc_calendar` that has three arguments, `(i, j,k)`; it adds `i`, `j` and `k` to the to the current year, month and date, respectively, and publishes a 4-tuple giving the resulting year, month, date and day of week. Similarly, `add_time(i,j,k)` adds `i`, `j` and `k` to the hour, minute and second, respectively, and publishes the resulting hour, minute and second as a triple, as shown in Figure 10.1.

Here is a typical computation of a future date (where the date is written in the format mm/dd/yyyy):

```
val cal = Orc_calendar()
cal.add_date(0,0,25) >(y,m,d,_)>
"Please pay your bill by " + m + "/" + d + "/" + y + "."
```

Or, computation of a future time:

```
val cal = Orc_calendar()
cal.add_time(2,0,0) >(h,m,_)>
"Meet me for lunch in two hours, at " + h + ":" + m + "."
```

Observe that `add_date(0,0,0)` publishes the current year, month, date and day and `add_time(0,0,0)` publishes the current hour, minute and second; so the methods `date()` and `time()` are unnecessary.

```
def class Orc_calendar() =

  import class Calendar = "java.util.Calendar"

  def pub(meth,"date") =
    (meth(Calendar.YEAR?),
     meth(Calendar.MONTH?)+1,
     meth(Calendar.DAY_OF_MONTH?),
     meth(Calendar.DAY_OF_WEEK?)
    )

  def pub(meth,"time") =
    (meth(Calendar.HOUR_OF_DAY?),
     meth(Calendar.MINUTE?),
     meth(Calendar.SECOND?)
    )

  def date() =
    val now = Calendar.getInstance()
    pub(now.get,"date")

  def time() =
    val now = Calendar.getInstance()
    pub(now.get,"time")

  def sub_time(fh,fm,fs) =
    time() >(ch,cm,cs)> (fh-ch,fm-cm,fs-cs) >(h,m,s)>

    (if s  <: 0 then (m-1,s+60)
                else (m,s))         >(m',s')>
    (if m' <: 0 then (h-1,m'+60)
                else (h,m'))        >(h',m'')>
    (if h' <: 0 then (h'+24,m'',s')
                else (h',m'',s'))

  def add_date(i,j,k) =
    val now = Calendar.getInstance()

    now.add(Calendar.YEAR?, i)         >>
    now.add(Calendar.MONTH?, j)        >>
    now.add(Calendar.DAY_OF_MONTH?, k) >>
    pub(now.get,"date")

  def add_time(i,j,k) =
    val now = Calendar.getInstance()

    now.add(Calendar.HOUR_OF_DAY?, i) >>
    now.add(Calendar.MINUTE?, j)      >>
    now.add(Calendar.SECOND?, k)      >>
    pub(now.get,"time")

stop
```

Figure 10.1: class `Orc_calendar`

Bibliography

[1] Will Adams. Verifying adder circuits using powerlists. Technical Report TR 94-02, Dept. of Computer Science, Univ. of Texas at Austin, Austin, Texas 78712, Mar 1994.

[2] Musab AlTurki and José Meseguer. Real-time rewriting semantics of orc. In *PPDP '07: Proceedings of the 9th ACM SIGPLAN international symposium on Principles and practice of declarative programming*, pages 131–142, New York, NY, USA, 2007. ACM Press.

[3] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent programming in ERLANG (2nd ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996.

[4] Kenneth Batcher. Sorting networks and their applications. In *Proc. AFIPS Spring Joint Computer Conference*, volume 32, pages 307–314, Reston, VA, 1968. AFIPS Press.

[5] Richard Bird, Jeremy Gibbons, and Geraint Jones. Program optimisation, naturally. In *Millennial Perspectives in Computer Science: the proceedings of the 1999 Oxford–Microsoft Symposium in honour of Sir Tony Hoare*, St. Catherine's College, Oxford, Sep 1999.

[6] K. M. Chandy and J. Misra. *Parallel Program Design. A Foundation*. Addison-Wesley, 1988.

[7] M.E. Conway. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7):396–408, July 1963.

[8] William R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University PhD Thesis, 1989.

[9] J. M. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Comp.*, 19(90):297–301, 1965.

[10] P.J. Courtois, F. Heymans, and D.L. Parnas. Concurrent control with readers and writers. *Communications of the ACM*, 14(10):667–668, Oct 1967.

[11] O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare. *Structured Programming*. Academic Press, 1972.

[12] Edsger W. Dijkstra. Guarded commands, nondeterminacy, and the formal derivation of programs. *Communications of the ACM*, 8:453–457, 1975.

[13] E.W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, 1968.

[14] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[15] Simon Marlow (editor). Haskell 2010, language report. Available at `http://www.haskell.org/onlinereport/haskell2010/haskell.html`, 2010.

[16] Jayadev Misra et. al. Orc language project. Web site. Browse at `http://orc.csres.utexas.edu`.

[17] R.A. Fisher and F. Yates. *Statistical tables for biological, agricultural and medical research*. Oliver and Boyd, London, third edition, 1948.

[18] Robert W. Floyd. Nondeterministic algorithms. *J. ACM*, 14(4):636–644, October 1967.

[19] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[20] M. P. Herlihy and J. M. Wing. Axioms for concurrent objects. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '87, pages 13–26, New York, NY, USA, 1987. ACM.

[21] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. *International Joint Conference on Artificial Intelligence*, 1973.

[22] C. A. R. Hoare. Partition: Algorithm 63, quicksort: Algorithm 64, and find: Algorithm 65. *Communications of the ACM*, 4(7):321–322, 1961.

[23] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[24] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1984.

[25] Guy L. Steele Jr. Making asynchronous parallelism safe for the world. In *POPL*, pages 218–231, 1990.

[26] D. Kapur and M. Subramaniam. Automated reasoning about parallel algorithms using powerlists. Manuscipt in preparation, 1994.

[27] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, 1998.

[28] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, third edition, 1997.

[29] Jacob Kornerup. *Data Structures for Parallel Recursion*. PhD thesis, University of Texas at Austin, 1997. Available for download at `http://www.cs.utexas.edu/users/kornerup/dis.ps.Z`.

[30] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *Journal of the Association for Computing Machinery*, 27(4):831–838, 1980.

[31] John Launchbury and Trevor Elliott. Concurrent orchestration in haskell. In *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell '10, pages 79–90, New York, NY, USA, 2010. ACM.

[32] D. J. Lehmann and M. O. Rabin. On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem. In *POPL*, pages 133–138, 1981.

[33] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.

[34] M. Douglas Mcilroy. Enumerating the strings of regular languages. *J. Funct. Program.*, 14(5):503–518, September 2004.

[35] Bertrand Meyer. *Object-oriented software construction*. Prentice Hall, second edition, 1997.

[36] R. Milner. *Communication and Concurrency*. International Series in Computer Science, C.A.R. Hoare, series editor. Prentice-Hall, 1989.

[37] R. Milner, M. Tofte, and R. Harper. *The Definition of ML*. The MIT Press, 1990.

[38] Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, May 1999.

[39] Jayadev Misra. Axioms for memory access in asynchronous hardware. *ACM Transactions on Programming Languages and Systems*, 8(1):142–153, 1986.

[40] Jayadev Misra. Loosely coupled processes. *Future Generation Computer Systems*, 8:269–286, 1992. North-Holland.

[41] Jayadev Misra. Powerlist: A structure for parallel recursion. *ACM Transactions on Programming Languages and Systems*, 16(6):1737–1767, Nov 1994.

[42] Jayadev Misra. Generating–functions of interconnection networks. In *Millennial Perspectives in Computer Science: the proceedings of the 1999 Oxford–Microsoft Symposium in honour of Sir Tony Hoare*, St. Catherine's College, Oxford, Sep 1999.

[43] David Park. Concurrency and automata on infinite sequences. In *Proceedings of the 5th GI-Conference on Theoretical Computer Science*, pages 167–183, London, UK, 1981. Springer-Verlag.

[44] R. Sedgewick. *Quicksort*. PhD thesis, Stanford University, 1975.

[45] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, November 1979.

[46] Stephen Warshall. A theorem on boolean matrices. *J. ACM*, 9(1):11–12, January 1962.