The Dissertation Committee for John Adam Thywissen
certifies that this is the approved version of the following dissertation:

Implicitly Distributing Pervasively
Concurrent Programs

Committee:

Christopher J. Rossbach, Supervisor

William R. Cook, Co-supervisor

Jayadev Misra

Simon Peter

Milos Gligoric

Implicitly Distributing Pervasively Concurrent Programs

by

John Adam Thywissen

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 2020

To my mother and my father.

I am extremely fortunate and happy to be your son.

Acknowledgments

This work resulted from the collaboration of the members of the Orc research group, namely Arthur Peters, David Kitchin, Adrian Quark, and Professors Christopher Rossbach, Jayadev Misra, and William Cook, among others. Arthur Peters, David Kitchin, and Adrian Quark, along with this author are co-designers and developers of the Orc language and the software systems that implement it. Jayadev Misra originated the Orc process calculus and programming language, and has guided its evolution. William Cook implemented the first iteration of the Orc programming language, and has provided programming language design and development expertise for much of the Orc research program. Christopher Rossbach provided systems expertise and support for making Orc practical, and provided leadership to bring the work to its conclusion.

This work was supported in part by funding from the National Science Foundation and other U.S. Government agencies, and by The University of Texas at Austin. Any opinions, findings, and conclusions or recommendations expressed in this document are those of the author and do not necessarily reflect the views of the National Science Foundation, the U.S. Government, or The University of Texas at Austin.

I benefitted from not just the intellectual contributions of my fellow graduate students, but also their inspiration, debate, and companionship. The faculty of the Department of Computer Science at the University of Texas at Austin is an extraordinarily generous and caring group of people, and have provided opportunities and support to me on many occasions. Professor William Cook started me on the Orc journey. Professor Jayadev Misra's generosity on many occasions was deeply appreciated. Professor Christopher Rossbach patiently provided guidance and support to get "over the finish line". I am exceptionally fortunate to have such a group of colleagues, mentors, and friends.

My partner, Andrea Weissenbuehler, has been encouraging, supportive, and helpful throughout the many years of graduate school. I truly would not be here without her.

Abstract

Implicitly Distributing Pervasively Concurrent Programs

by

John Adam Thywissen, PhD

The University of Texas at Austin, 2020

Supervisor: Christopher J. Rossbach

Co-supervisor: William R. Cook

Distributed programs are often written as a collection of communicating modules. For example, to use Java RMI, programs are divided into objects which can be remotely referenced. Yet, in many cases, it would be desirable to write the program without the program structure being driven by distribution decisions. If distribution is decoupled from program structure, the programming language must allow communication throughout a program's structure, instead of at a few known points. This situation is simplified if the programming language provides a uniform programming model for local and remote values (location transparency). We present Distributed Orc, which offers location transparency, and distributes program operations automatically in cooperation with the execution environment. By eliminating any special semantics of remote values, Distributed Orc enables programmers to write cohesive distributed programs, rather than programs artificially divided at distribution boundaries. Distributed Orc is derived from the Orc language, a (centralized) concurrent orchestration language.

Contents

Figures

Tables

Listings

1.
Introduction

1.1. Background and Motivation: Why Implicit Distribution?

Modern applications are increasingly distributed, meaning that applications operate concurrently in multiple places. Sometimes, distribution is necessary to access physical aspects of the environment in various locations, for example a sensor, an output device, or the user. But, predominately, distribution is used to take advantage of system resource availability—storage space, processing power, networking capacity, etc.

In current general-purpose distributed programming models, Java RMI for example, distribution is manual: Programmers explicitly partition programs into modules, explicitly specify their interaction, and then explicitly place them in the locations where they will execute.

Consider a robot team given the task of photographing an object. The team needs to decide which members should attempt the task. This decision could be implemented as a centralized program by simply scanning the list of robots and selecting robots close to the object. Alternatively, to decentralize this program, the programmer could write code from the point of view of each robot, enabling the robots to exchange distances to the object with each other and self-select. The centralized variant is simpler and easier to write correctly, since the global behavior is directly specified in the program, rather than derived from the behavior of multiple program instances.

When a program is written in the centralized style, without explicit partitioning or distribution boundaries, we call it *cohesive*. The system-wide viewpoint of a cohesive program seems to imply that the team is controlled by a central controller, or at least an elected leader. This centralization is antithetical to the goals of team robotics, and many other distributed systems. Consequently, many distributed systems are written in the decentralized (non-cohesive) style, despite the increased complexity. In these cases,

the imperative to avoid a central controller drives the overall application architecture, as well as module and distribution boundaries.

This distribution-induced architecture fragmentation has multiple adverse effects:

- The program is less comprehensible to developers. Program logic is fragmented among various distribution-imposed modules, requiring developers to understand the decomposition for distribution in order to locate basic program logic.

- Software engineering goals that should drive the program's modularization are subverted by the needs of the distribution decisions. Modularization constructs exist in programming languages to support software engineering concerns. Principles such as information hiding, avoidance of adverse coupling, module coherence, and so forth, are what should drive a program's module structure. When modules are used as units of distribution, the requirements for distribution override the "proper" use of modules to address software engineering desiderata.

- Some program properties that are explicit in a cohesive program become emergent in the non-cohesive variant. Since the program's logic is fragmented among modules, some behavior that would have been explicit in a cohesive program emerges only from the interaction of the distributed modules.

- Changing distribution placement requires program architecture changes. If the modularization of a program was determined by distribution boundaries, moving those boundaries necessitates re-modularization.

- Interface or communication code must be developed and maintained. Since the non-cohesive program consists of communicating modules, the communications interface among these modules becomes part of the software development and maintenance burden.

- Program enhancement, adaptation, debugging, and verification is hampered by larger, fragmented, and more complicated programs. All of these effects add up to a program that is harder and slower to build, understand, and change.

Many alternative distributed programming models, such as map–reduce [21], improve this situation by abstracting a specific communication pattern. However, the program's modularization must conform to the given pattern, such as a map and reduce function with fixed parameters.

In this work, we consider distribution of general purpose programs. As mentioned, in current general-purpose distributed programming models, such as Java RMI [64], CORBA [43], Microsoft .NET Remoting [40], Akka Actors [1], and gRPC [27], programmers explicitly partition programs into a collection of modules, where each module instance executes at a single location, and then specify each interaction. While some workloads can benefit from this explicit decomposition, others are less well-served by it.

For many workloads, distribution is merely a technique to take advantage of resources available in the execution environment. A common case is a networked cluster, which may have changing available resources as the program executes. In these cases, the interfaces between distributed parts of a program are immaterial to the developers and users. Some distributed programming languages have attempted to allow cohesive development by abstracting away concerns of where each part of the program executes [14, 16, 30, 34, 60]. This language facility is called *location transparency*. Languages that provide location transparency often include not just transparent remote call invocation, but also transparent migration of executing threads among locations (for example, Black et al. [14], Jul [34]).

Cohesive distributed programming, enabled by location transparency, mitigates the adverse effects listed above. However, by enabling distribution to occur throughout a cohesive program, the realities of distributed computing are also pervasive throughout the program. These distributed computing realities include concurrency and failure. That is to say, location transparency implies pervasive concurrency and pervasive failure-proneness.

1.2. Problem Statement

Programming languages that provide location transparency often include not just transparent remote call invocation, but also transparent migration of executing threads among locations. This abstraction has been sharply criticized [61], and location transparency is not in widespread use. The essence of the criticism of location transparency is that semantics of a remote operation should not be "hidden" by appearing as an operation with local semantics. We agree, and propose the reverse: *Make the semantics of local operations consistent with the semantics of remote operations.*

Two important concerns with remote operations in a distributed system are the possibility of failure and the latency of communication. These concerns are addressed by detecting and responding to failure, and by overlapping operations using concurrency. It may seem excessive to impose these concerns on local operations, instead of just remote operations.

Thesis 1. *Distributed programs can be written in a location-transparent style, where local and remote semantics are uniform. This uniformity does not cause the local semantics to be awkward for programmers.*

Result. We found support for this thesis, through experience with Distributed Orc and through a usability evaluation.

Most current distributed languages use the language's modularity construct (usually objects) as the entities that are distributed. However, modularization of programs is for the benefit of the developers, and is intended to mitigate conceptual complexity and improve adaptability to change. Distribution of programs, in contrast, is driven by platform characteristics (such as performance and capabilities) and communication costs. We propose that the reuse of the units of a module decomposition (objects) as the units of deployment in a distributed system is a conflation of concerns. It is not the case that, universally, a desirable module decomposition yields the desired units for deployment.

Thesis 2. *Distributed programs should be able to communicate as part of any expression,*

not just when crossing object/module boundaries. Language modularity constructs (such as objects) should be used for software engineering concerns, and not be co-opted by the distribution mechanism.

Result. We found support for this thesis through experience with Distributed Orc and through a usability evaluation.

The elimination of explicit local/remote semantic distinctions and the elimination of explicit distribution boundaries from the program delegates many responsibilities to the language system. The system must track locations of values used in the program, and infer where communications should occur.

Thesis 3. *The language system can automate the distribution of these location-transparent distributed programs.*

Result. We found support for this thesis, by constructing a system that does so.

The current practice of manual placement of program operations can involve significant engineering optimization studies. A program's aggregate network performance often can only be determined post hoc. Delegating placement responsibility to the language system requires that it make acceptable placement choices, so that communications costs do not become prohibitive.

Thesis 4. *The performance of programs under this automated distribution is adequate.*

Result. We did not find support for this thesis at present. Excessive communications costs have caused poor performance in our evaluation. Efforts to reduce these costs have been ineffective.

1.3. Significance and Implications

Computing's value to people depends on the benefits it provides by delivering relevant services at helpful times, with minimal encumbrances. As the use of computing grows, the effort required to develop and maintain computing-based services must decrease, and

their quality must increase. Advances in programing language design are crucial to these improvements.

Much of the progress made across the history of programming languages has shifted programmer responsibilities to the language system. For example, explicit memory management has been supplanted by language-managed deallocation in modern languages. By offloading these types of mechanical details, the programmer is left free to focus on the application specifics. This results in quicker outcomes, lower costs, and higher quality software systems.

We view implicitly distributed programs as a similar step in the evolution of programming languages. Distributed programming conventional wisdom is that distributed programs *must* be written with explicit distribution, and current practice reflects this. Some situations call for detailed explicit specification of the interfaces and protocols among distributed system components. In these cases, current practice may be appropriate.

However, this work argues that there is a significant proportion of distributed software development where *implicit* distribution is applicable, contrary to conventional wisdom. Languages such as Distributed Orc eliminate the effort of explicit development of interfaces and protocols, and leave the language system free to optimize for, and adapt to, the execution environment. By unburdening software developers in these cases, implicit distribution enables more rapid development of new applications and accelerated adaptation to changing needs, lower development and maintenance costs, higher quality systems, and dynamic adaptation to execution environments as they change.

2.
Literature Review

The area of programming models for distributed computing has been an active one for many years. Here we cover some of the most closely related work. We also refer the reader to Weisenburger et al. [62] for a recent survey of cohesive, but not transparent ("multi-tier") programming languages, and Bal et al. [7] for a historical survey of earlier programming languages for distributed computing systems.

Later, in chapter 8, we present Distributed Orc, which offers location transparency, and distributes program operations automatically in cooperation with the execution environment. As we present related work here, we make some comparisons to Distributed Orc.

2.1. Distributed Objects with Location Transparency

There are a number of distributed languages with features similar to Distributed Orc. However, all of these languages operate by migrating objects or making RMI-like calls, rather than Distributed Orc's migrating tokens. Further, these languages do not have the asynchrony- and failure-awareness combinators inherent in Orc.

The Emerald language [14, 34] was an impressive early leader in this area. Emerald provided many features: location transparency, migration of objects among nodes, remote references, etc. Emerald argued for a cohesive programming style, with no explicit communications (RPC, RMI) code. There are multiple systems that extend from Emerald's ideas, for example ProActive [10] and the Sapphire system [68]. Obliq [16] was a influential language that provided location transparency, but without migration of objects among nodes. Distributed Oz [30, 60] builds on the concurrent language Oz. Distributed Oz adds location transparency and migration to Oz. This has strong similarities to the Distributed Orc approach; however, Orc has distinctly different combinators, and does

not have the consistency needs implied by Oz's constraint store.

2.2. Distributed JVMs

J-Orchestra [59] and JavaSplit [23, 24] implement a software distributed shared memory abstraction in the Java VM. Threads are placed on nodes when they are created. References to remote objects and distributed locks are transparently handled. Since J-Orchestra and JavaSplit operate on programs via bytecode rewriting, distribution-unaware applications can be run without programmer or user modification or reconfiguration. These systems, as described, do not attempt to place threads based on any cost model, nor do they support migration. Factor et al. [23] includes a survey of eight distributed JVM tools.

2.3. Tightly-Coupled (Local) Distributed Models

Much of the work under monikers similar to "automated distributed programming models" targets distributed shared memory architectures, such as non-uniform memory architecture (NUMA) systems or tightly-coupled local clusters, for example André et al. [5], Banâtre et al. [8], Bareau et al. [9], Bauer et al. [11], Chen et al. [17], Haines [29]. They typically assume homogeneous hardware and regular data. Some work takes an explicitly single-instruction, multiple-data (SIMD) approach, despite their "distributed" label. Other work targets software distributed shared memory (S-DSM) systems, which still is generally constrained to local clusters.

In contrast, Setälä et al. [54] present a system that is not a homogeneous DSM-style system, but targets heterogeneous co-processors (such as FPGAs) in a single device.

Some aspects of this work may apply in heterogeneous or truly distributed systems.

2.4. Distributed Dataflow

Dryad [32] is a dataflow-style distributed language with a set of combinators that have some similarities to Orc. Unlike Orc, Dryad's execution engine contains a job manager that maintains a global state of the distributed computation graph to schedule work on cluster nodes. The job manager attempts to execute computations close to their data by using metadata that specifies data locations used by jobs and cost functions [33]. The job-level aggregate costs are minimized by solving for a min-cost flow over the network infrastructure.

Other dataflow frameworks such as MapReduce [21], Spark [66], and Spark Streaming [67] target computations that can be refactored to fit the frameworks' fixed architecture. For example, MapReduce requires computations to be decomposed into map and reduce stages. if the computations do not fit these frameworks' architecture, the decomposition becomes awkward, and can result in inefficiencies.

Some dataflow languages such as CIEL [42] generalize the architecture, but still have a centralized scheduler.

2.5. Distributed Program Slicing

In Distributed Orc, various program slices will be executed on the nodes of the distributed system. Distributed dependence graphs [22] use slicing in a distributed setting, but where the distribution and communication is explicitly specified. The work extends the conventional program dependence graph (PDG) to include inter-node communications dependences. This work is extended in Garg and Mittal [26].

To analyze parts of the program that affect a point of interest, backward slicing may be used. Danicic and Laurence [19] does so for non-deterministic programs.

2.6. Program Fragmentation

Demaq/Transscale [15], at compile time, fragments cohesive programs into per-node sub-programs that then do not need a distributed runtime system. Examining distributed system state is expressed as a query against the history of locally received messages. During fragmentation, the per-node sub-programs are refactored to ensure the each node's local message history captures all system state that the node may query. This is done by duplicating messages and memoizing via derived messages.

2.7. Tierless Web Development

Several languages (Hop [53], RED [41], etc.) address distribution in the Web development setting. STIP.JS [49] instead uses JavaScript and program slicing techniques to generate the per-tier codebase. These languages require explicit annotation of which parts of the program run on which tiers.

2.8. Migration

Antoniu et al. [6] dynamically migrates threads in High Performance FORTRAN among nodes. JESSICA2 [69] distributes Java programs' threads across a cluster, and migrates Java objects among threads. However, the migration decision is simply based on observed access counts during the current execution.

2.9. C-omega/LINQ and Remote Batch Invocation

Cω and Remote Batch Invocation provide a cohesive programming model for certain distributed operations. Cω [13], now implemented as Language Integrated Query (LINQ), unifies distributed computation for the case of database queries. Both the database code and the surrounding code are written as a single program using the same programming language and model. Cω partitions the data operations from the rest of the program, translates the data operations to a database language, and migrates them to the database

server. The database operations are not requested one at a time, but a small database procedure is transmitted. The script corresponds to a stream of operations in the program, saving many network round-trips. Dandelion [52], leverages the LINQ-style language semantics to target clusters of GPUs and FPGAs.

Relatedly, Remote Batch Invocation [31] handles code blocks of computation on a mix of local and remote objects. Like Cω, Remote Batch Invocation partitions the program into local and remote operations, and migrates a script to the remote node that corresponds to multiple program steps, rather than issue a series of one-at-a-time RMI calls. Notably, loops and branches in these blocks are "remoted" (migrated) when they operate on remote data.

2.10. Distributed Process Calculi

The π-calculus has inspired a number of distributed process calculi. Two notable examples are Distributed Join-Calculus [25] and Nomadic Pict [55]. See the Related Work section of Sewell et al. [55] for an extensive overview of this area. Orc itself started as a concurrent calculus, and evolved into a full language.

2.11. Related Orc-Based Work

As part of work on architectural descriptions of grid computing applications, O2J [2] proposes using Orc to model the orchestration aspects of these applications. O2J implements some of the Orc combinators in Java, so that an orchestration modeled in Orc can be directly translated to Java for further development. Because of its intended use, O2J does not attempt to provide migration or full location transparency.

Dist-Orc [4] builds on the authors' previous rewriting logic specification of Orc to provide communication mechanisms to otherwise independent Orc instances. One may then formally verify correctness properties of distributed *real-time* Orc systems. Because of this focus on correctness, Dist-Orc is a conservative extension to Orc and does not include migration or location transparency.

Quality of service (QOS) modeling of orchestrations, using Orc as a specification language, is developed in Benveniste et al. [12].

3.

Requisites of a Transparently Distributed Language

If distribution concerns are to be delegated to the language system, then the language semantics must accommodate the realities of distributed programming. For example, a vector addition `x + y` operation for 5-element `x` and `y` vectors would not be worth executing remotely. However, if `x` and `y` are billion-element vectors, distributed processing would be preferable. In a transparently distributed language, this transition from a purely local to a distributed operation is not visible to the program. The language needs to provide either local or distributed semantics for the vector add + operation.

The primary changes from purely local programming to distributed programming are accommodating (1) concurrency, and (2) failures. Previous transparently distributed languages have attempted to hide concurrency and failures to make remote operations seem as if they were local. Our work takes the opposite approach, and embraces pervasive concurrency and failures, even for local operations.

3.1. Pervasive Concurrency

Distributed execution of a program is inherently multi-processor, since the program is running on various network-connected machines. Non-transparently distributed languages expose this concurrency to programs at the distribution boundaries. For example, languages that use objects as the unit of distribution often permit treating method invocations as message sends, rather than synchronous calls.

When the language system is transparently managing distribution, these concurrency effects then may appear throughout the program wherever the language system chooses to distribute execution. In other words, any operation could be concurrently executed (within necessary constraints) with others.

Suppose a distributed language were to attempt to make remote (concurrent) oper-

ations seem as if they were local (sequential). To preserve the appearance of a conventional execution model, i.e., that of a single, in-order processor, the machines executing the program would need to coordinate extensively. The needed coordination would include aspects similar to that of cache coherence features found in (local) multiprocessor systems and effect reordering features found in pipelined processors. However, in this distributed setting, inter-processor coordination is far more expensive than in the local multiprocessor case, because coordination transits the network, rather than processor-internal busses or inter-processor interconnects. This expense impels the relaxation of any execution model that attempts to preserve the appearance of in-order program execution. This relaxed execution model will reveal concurrency effects to some extent.

This is why our work embraces pervasive concurrency: Instead of attempting to "paper over" the inherent concurrency of distributed systems, and then retreating to partially concurrent semantics, we adopt full pervasive concurrency.

3.2. Pervasive Failure Handling

Because of the current level of reliability of contemporary networks, distributed systems encounter network failures frequently, relative to failures of other components.

Many distributed programming languages provide some failure-awareness facilities. Resilient X10 [18], an explicitly distributed language, will throw a `DeadPlaceException` when it detects that a targeted location is inaccessible. It is then up to the application to respond appropriately.

Failures in distributed systems can take many forms, but the most common failures appear as a lack of response from a remote machine. This could be caused by a machine failure or a network outage, for example. This common *fail-stop* type of failure is what most distributed programming environments address. An alternative form of failure, *Byzantine failure*, where failing machines could act arbitrarily (even maliciously), is less commonly addressed by distributed programming environments, and is out of scope for this work.

Just as with concurrency, if the language system is transparently managing distribution, these failures then may appear throughout the program wherever the language system chooses to distribute execution. This means arbitrary operations in the program could fail-stop. A transparently distributed language must provide a means of detecting fail-stop failures of any operation.

4.
Orc: A Basis for a Transparently Distributed Language

As a starting point for our exploration of transparent distribution, we adopt the Orc programming language [35, 38]. Orc provides the pervasive concurrency needed for transparent distribution.

4.1. Overview of the Orc Language

Orc is both a process calculus and a programming language which implements it. The Orc language and calculus are designed to facilitate concurrent orchestration, meaning that the Orc program manages various other modules that may or may not be written in Orc. These modules are called *sites* and they are used to represent everything from arithmetic operations to user interface devices. Sites are called with arguments and *publish* (return) a stream of zero or more values to the calling Orc program. How these values are computed and the timing of their publication is outside the purview of Orc. This allows enormous flexibility in how Orc programs interact with their environment. Sites are first-class values.

To utilize this weakly constrained site behavior, Orc programs are pervasively concurrent. Operation ordering constraints are far less common than in conventional languages. The Orc combinators, described below, offer concurrency structures that take the place of conventional control structures. Orc variables are immutable in order to reduce the potential for unwanted data races. When needed, mutable data structures are available through site calls.

Orc extends its immutable variables with transparent futures. A variable can be bound to a future, which is a value that will be available at some later time. When the variable is read, the reader waits for the value (blocks) if it has not been computed yet. This enables concurrent, race-free execution of the writer and reader, but does not require

explicit synchronization code.

Orc provides five *combinators* to process streams of publications from site calls or nested expressions, e_1 and e_2.

Parallel, written e_1 `|` e_2, runs both e_1 and e_2 and combines the publication streams into a single stream.

Sequential, written e_1 `>`*x*`>` e_2, provides a "fan-out" or for-each facility. It runs e_1, and for each publication, runs an instance of e_2 with the variable *x* bound to the publication. Note that e_2 is run immediately upon each publication of e_1, even while e_1 continues to run. The publications of the overall sequential expression are the combined publications of all the e_2 instances.

Trim, written `{|` e `|}`, provides a "pick the first available result" facility. When e publishes a result, that result is published by the trim combinator, and further execution of e is terminated. An execution of the trim combinator is guaranteed to publish at most once.

Graft, written **`val`** *x* `=` e_1 followed by e_2, provides future facilities. Graft runs both e_1 and e_2, with the variable *x* bound to a future in expression e_2. The future will be bound to the *first* publication of e_1, and further publications of e_1 will be ignored. If e_2 refers to the variable *x* before it has been bound, the reference to *x* will wait for the publication. If e_1 halts without publishing, references to *x* will halt as well. The resulting publications of the overall graft expression are the publications of e_2.

Otherwise, written e_1 `;` e_2, provides a "fallback" facility. Otherwise runs expression e_1, and then if e_1 terminates without publishing, runs e_2. If e_1 publishes one or more times, e_2 is not run. The overall publications of the otherwise expression are either those of e_1 or e_2.

Orc imposes no ordering constraints on the interleaving of publications from the subexpressions of the parallel or sequential combinators.

As an example of Orc, consider the following problem: Download a file from the fastest of two servers, and return an error if no server works. In Orc:

```
{| download(A) | download(B) |} ; error()
```

The expression `download(A) | download(B)` runs the downloads concurrently. When a download completes the trim combinator publishes the result and terminates the other download. Finally, the expression `...; error()` publishes an error if *d* is not bound to a value. This example shows how Orc simplifies the implementation of orchestration problems that in many systems would require complicated event handling.

The error handling in this example can be extended with timeouts easily:

```
{| download(A) | download(B) | (Rwait(10 * seconds) >> error() |}
```

The expression `Rwait(10 * seconds) >> error()` publishes the error after ten seconds, causing all the downloads to abort. This will occur if all of the downloads fail or are delayed for any reason including network latency or failure.

Here, we have presented the core Orc calculus. The "surface language" of Orc contains additional constructs, such as if-then-else statements, pattern matching, and list literals, that are "syntactic sugar". During compilation of an Orc program, these language constructs are translated into core Orc calculus expressions that make use of standard Orc library sites.

Orc also includes an object system [45].

Full details of the Orc language are available at Orc language project Web site https://orc.csres.utexas.edu/, which provides an interactive "Try Orc" Web interface, a user guide [39], reference manual [38], examples, and downloadable implementations of Orc. Excerpts of the user guide are presented in the remainder of this chapter.

4.2. Simple Expressions in Orc

An Orc program is an expression. Complex Orc expressions are built up recursively from simpler expressions. Orc expressions are *executed*; an execution may interact with external services, and *publish* some number of values (possibly zero). Publishing a value

is similar to returning a value with a `return` statement in an imperative language, or evaluating an expression in a functional language, except that an execution may publish many times, at different times, or might not publish at all. An expression which does not publish is called *silent*.

An execution *halts* when it is finished; it will not interact with any more services, publish any more values, or have any other effects.

Different executions of the same expression may have completely different behaviors; they may call different services, may receive different responses from the same site, and may publish different values.

This section shows how to write some simple Orc expressions. Simple expressions publish at most one value, and do not recursively contain other expressions. We will see later how some of these cases may also be used as complex expressions.

The simplest expression one can write is a literal value. Executing that expression simply publishes the value.

Orc has four kinds of literal values:

- Booleans: **`true`** and **`false`**
- Numbers: `5`, `-1`, `2.71828`, ...
- Strings: `"orc"`, `"ceci n'est pas une |"`
- A special value **`signal`**.

Orc has a standard set of arithmetic, logical, and comparison operators. As in most other programming languages, they are written in the usual infix style. They have Java-like operator precedence, which can be overridden by adding parentheses.

- `1 + 2` publishes `3`.
- `(98 + 2) * 17` publishes `1700`.
- `4 = 20 / 5` publishes **`true`**.

- `3-5 >= 5-3` publishes **`false`**.
- **`true`** && (**`false`** || **`true`**) publishes **`true`**.
- `"leap" + "frog"` publishes `"leapfrog"`.
- `3 / 0` halts, publishing nothing.

An Orc program interacts with the external world by calling *sites*. Sites are one of the two fundamental concepts of Orc programming, the other being combinators which we discuss later when covering complex expressions.

A site call in Orc looks like a method, subroutine, or function call in other programming languages. A site call might publish a useful value, or it might just publish a signal, or it might halt, refusing to publish anything, or it might even wait indefinitely. Here are some examples:

- `Println("hello world")` prints hello world to the console and publishes a **`signal`**.
- `Random(10)` publishes a random integer from 0 to 9, uniformly distributed.
- `Browse("http://orc.csres.utexas.edu/")` opens a browser window pointing to the Orc home page and publishes a **`signal`**.
- `Error("I AM ERROR")` reports an error message on the console, and halts. It publishes nothing.
- `Rwait(42)` waits for 42 milliseconds, then publishes a signal.
- `Prompt("Username:")` requests some input from the user, then publishes the user's response as a string. If the user never responds, the site waits forever.

Even the most basic operations in Orc are sites. For example, all of the operators are actually sites; `2+3` is just another way of writing the site call `(+)(2,3)`.

By convention, alphabetic site names begin with a capital letter.

4.3. Orc Combinators

Complex expressions recursively contain other expressions. They may be formed in a number of ways: using one of Orc's five combinators, adding a declaration, adding a conditional expression, or using an expression as an operand or site call argument.

The concurrency combinators are one of the two fundamental concepts of Orc programming, the other being sites. They provide the core orchestration capabilities of Orc: parallel execution, sequential execution, blocking on future values, terminating a computation, and trying an alternative if some computation halts.

4.3.1. Parallel Combinator

Orc's simplest combinator is **|**, the parallel combinator. Execution of the complex expression F **|** G, where F and G are Orc expressions, executes F and G concurrently. Whenever a value is published during the execution of F or G, the execution of F **|** G publishes that value. Note that the publications of F and G are interleaved arbitrarily.

```
{- Publishes 1 and 2, in either order -}
1 | 1+1
```

The brackets *{- -}* enclose comments, which are present only for documentation and are ignored by the compiler.

4.3.2. Sequential Combinator

Now that we have expressions which publish multiple values, what can we do with those publications? The sequential combinator, written F >x> G, combines the expression F, which may publish some values, with another expression G, which will use the values as they are published; the variable x transmits the values from F to G.

The execution of F >x> G starts by executing F. Whenever F publishes a value, a new execution of G begins in parallel with F (and with other executions of G). In that instance of G, variable x is bound to the value published by F. Values published by the executions of

G are published by the whole expression, but the values published by F are not published by the whole expression; they are consumed by the variable binding.

```
{- Publishes 1 and 2 in parallel (in either order) -}
(0 | 1) >n> n+1

{- Publishes 3 and 4 in parallel (in either order) -}
2 >n> (n+1 | n+2)

{- Publishes 5 -}
2 >x> 3 >y> x+y
```

The sequential combinator may also be written without a variable, as in F **>>** G. This has the same behavior, except that no variable name is given to the values published by F. When F publishes only one value, this is similar to a sequential execution in an imperative language.

```
{- Print three messages in sequence -}
Println("Yes") >>
Println("We") >>
Println("Can") >>
stop
```

The simple expression **stop** does nothing and halts immediately. In conjunction with **>>**, it can be used to ignore unneeded publications, such as the signal that would be published by `Println("Can")`.

4.3.3. Trim Combinator

The trim combinator, written `{| F |}`, allows us to terminate a computation. Whenever F publishes a value, that value is published by the entire execution, then the execution of F is immediately *killed*. A killed expression cannot make any more site calls or publish any values.

```
{- Publishes 3 or 4, but not both -}
{| 3 | 4 |}
```

Though a terminated execution may not make any new calls, the calls that it has already made will continue normally; their responses are simply ignored. This may have surprising consequences when a call has side effects, as in the following example.

```
{- This example might actually print both "uh" and "oh" to the
   console, regardless of which call responds first. -}
{| Println("uh") | Println("oh") |} >> stop
```

Both of the `Println` calls could be initiated before either one of them publishes a value and terminates the expression. Once the expression is terminated, no new calls occur, but the other `Println` call still proceeds and still has the effect of printing its message to the console.

4.3.4. Graft Combinator

The graft combinator, written `val x = F # G`, allows us to block a computation waiting for a result. The execution of `val x = F # G` starts by executing F and G in parallel. Whenever G publishes a value, that value is published by the entire execution. When F publishes its first value, that value is bound to `x` in G. Further publications of F are ignored.

During the execution of G, any part of the execution that depends on `x` will be blocked until `x` is bound. If F never publishes a value, parts of G may be blocked forever. If F halts silently (without publishing), references to `x` will halt silently too.

```
{- Publishes 5 or 6, but not both -}
val x = 3 | 4
x+2
```

4.3.5. Otherwise Combinator

Orc's fifth concurrency combinator, the otherwise combinator, is written F `;` G. The execution of F `;` G proceeds as follows. First, F is executed. If F halts, and has not published any values, then G executes. If F did publish one or more values, then G is ignored.

4.4. Other Orc Compound Expressions

4.4.1. Conditional Expressions

Orc has a conditional expression, written **`if`** E **`then`** F **`else`** G. The **`else`** branch is required. Execution of **`if`** E **`then`** F **`else`** G first executes E. If E publishes **`true`**, E is terminated and F executes. If E publishes **`false`**, E is terminated and G executes.

4.4.2. Nested Expressions

The execution of an Orc expression may publish many values. What if we want to use such an expression in a context where only one value is expected? For example, what does `2 + (3 | 4)` publish?

Whenever an Orc expression appears as an argument (or operand), it executes until it publishes its first value, and then it is terminated. The published value is then used in the context. This allows any expression to be used as an operand of an operator expression or an argument to a site call.

```
{- Publishes 5 or 6 -}
2 + (3 | 4)

{- Publishes exactly one of 0, 1, 2 or 3 -}
(0 | 2) + (0 | 1)
```

To be precise, whenever an Orc expression appears in an argument position (of a call or operator expression), it is treated as if it were the value subexpression of a graft combinator, using a fresh variable name to fill in the hole. This is called *deflation*. For example, `Println(1 | 2)` deflates to **`val`** `x = 1 | 2 # Println(x)`.

5.
Notation: Token-Style Operational Semantics

The Orc programming language is defined by its formal semantics. In this work, Orc's semantics are formalized using our variant of structural operational semantics [50] that we call token semantics.[1] In this chapter, we describe token semantics in general, and then in later chapters, we utilize token semantics to specify language semantics under study here.

We use this formalization to: (1) improve the precision of the language definition, (2) guide the implementation, and (3) provide a basis for proving properties of the language.

Token semantics make steps just as structural operational semantics do, but represent execution state differently. Threads of execution are represented as tokens which move through the program abstract syntax, carrying the state of the thread of execution. A program's original abstract syntax is always available by simply erasing all tokens from an executing instance.

In particular, instead of carrying environments on the left of a turnstile symbol, environments are carried in tokens. Also, when a subexpression is evaluated, instead of rewriting the program's abstract syntax to its evaluated value, the resulting value is carried in the token.

Here, we first review structural operational semantics, and then describe the notation used by the Orc token semantics, which will be used in semantic rules in subsequent chapters.

5.1. Structured Operational Semantics

In Structured Operational Semantics [50] (also called small-step semantics), a set of *configurations* is given, along with a reduces relation ($\rightarrow$) among the configurations.

[1] Previous work on Orc has used a more traditional structural operational semantics, but here we use this style of semantics, anticipating the extension to the distributed semantics.

Conventionally, the reduces relation is specified via inference rules in the form

$$\frac{\gamma_1 \to \gamma_1' \quad \gamma_2 \to \gamma_2' \quad \cdots}{\gamma \to \gamma'}$$

The judgement $\gamma \to \gamma'$ asserts that the configurations γ and γ' are in the reduces relation.

The reduces relation is sometimes read as "reduces in one step" to be more explicit. Note that the reduces relation is irreflexive—configurations do not reduce to themselves in one step.

Configurations represent the state of the reduction. This can be a program (language term), or perhaps a program and the state of memory (the store).

The transitive-reflexive closure of reduces, *i.e.* reduces in zero or more steps, is written $\to^*$.

Configurations are evaluated by applying series of reduction steps:

$$\gamma \to \gamma' \to \gamma'' \to \gamma''' \to \gamma''''$$

this is also

$$\gamma \to^* \gamma''''$$

To justify each step, a derivation of that step can be exhibited using the reduction rules of the language. Justification of the reduction sequence $\gamma \to \gamma' \to \gamma'' \to \gamma'''$ would have three derivation trees, with roots $\gamma \to \gamma'$, $\gamma' \to \gamma''$, and $\gamma'' \to \gamma'''$.

$$\frac{\dfrac{\dfrac{\cdots}{\gamma_1 \to \gamma_1'} \quad \dfrac{\cdots}{\gamma_2 \to \gamma_2'}}{\gamma \to \gamma'} \quad \dfrac{\dfrac{\cdots}{\cdots} \quad \dfrac{\cdots}{\cdots}}{\gamma' \to \gamma''}}{\cdots}$$

5.1.1. SOS Example

A simple grammar of addition:

$v ::= 0|1|2|3|4|5|6$

$e ::= v \mid e + e$

along with a reduction relation specified by the single rule:

$$\frac{}{v_1 + v_2 \to v_3} \text{ (Add)}$$
$$\text{where } v_3 = v_1 + v_2$$

will permit the reduction of $1 + 2$ to 3 in one step. Reducing $1 + 2 + 3$ requires a *congruence rule* which permits reducing subexpressions of the + operator:

$$\frac{e_1 \to e_2}{e_1 + e_3 \to e_2 + e_3} \text{ (Add-Left)}$$

Now, $1 + 2 + 3$ can be reduced in two steps:

$$\frac{\dfrac{}{1 + 2 \to 3}}{1 + 2 + 3 \to 3 + 3} \quad \frac{}{3 + 3 \to 6}$$

5.1.2. Adding State to Configurations

When evaluating program expressions, there is often necessary evaluation state beyond the program abstract syntax. This may be variable bindings, the memory state, and so forth. In operational semantics, this state is represented in the configurations that the rules operate upon. In the above example, the configurations are simply expressions of the language (nonterminal e). To add variable bindings, the configurations can become a pair containing an environment and the program abstract syntax. The environment maps variables to values $\rho(x) = v$. The conventional notation presents environment to the left of a turnstile: $\rho \vdash e_1 \to e_2$ denotes $\langle \rho, e_1 \rangle \to \langle \rho, e_2 \rangle$ is in the reduces relation $\to$.

Evaluation of a variable x retrieves its mapping in the current environment:

$$\frac{}{\rho \vdash x \to v} \text{ (Var)}$$
$$\text{where } v = \rho(x)$$

Binding a variable x to a value v adds the mapping $x \mapsto v$ to the environment ρ for

evaluation in its scope, for example:

$$\frac{\rho' \vdash e_1 \to e_2}{\rho \vdash \texttt{let}\ x = v\ \texttt{in}\ e_1 \to\ \texttt{let}\ x = v\ \texttt{in}\ e_2}\ \text{(Def)}$$
$$\text{where } \rho' = \rho[x \mapsto v]$$

5.2. Token Semantics

In token semantics, the configurations are different from traditional Structured Operational Semantics. Rather than the configuration being tuples with execution state and the program abstract syntax, the execution state is carried in "tokens" inserted in to the program abstract syntax. This facilitates repeated and concurrent execution of expressions while maintaining notational concision.

The presence of a token $\bullet$ in the program abstract syntax indicates that execution is occurring at that location. For example, the expression $\bullet\ (1+2)$ indicates the addition expression is ready for execution. The results of an execution, i.e. a publication in Orc, are carried by a token. We indicate that a token is carrying a published value v by placing a superscript on the token $\bullet^v$. For example, $(1+2)\ \bullet^3$ indicates the addition expression's execution resulted in (published) 3.

Tokens carry an environment ρ, which is indicated as a subscript on the token $\bullet_\rho$. A token's environment is extended with a new mapping as it enters a binding construct, for example:

$$\frac{}{\bullet_{\rho,}\ \texttt{let}\ x = v\ \texttt{in}\ e \to\ \texttt{let}\ x = v\ \texttt{in}\ \bullet_{\rho',}\ e}\ \text{(Def)}$$
$$\text{where } \rho' = \rho[x \mapsto v]$$

Additionally, tokens carry a *tag*, which identifies the token's membership in sets of tokens. This membership is used, for example, to terminate all member tokens of a particular set of tokens. A token's tag θ is indicated as a second subscript on the token $\bullet_{\rho,\theta}$.

5.2.1. Token Sematics Example: Orc Parallel Combinator

Remember that the semantics for Orc's parallel combinator $l \mid r$, are that both sides of the combinator are executed, and any publications from either side are published by the combinator. The token semantics rules for the parallel combinator are:

$$\frac{}{\bullet_{\rho,\theta}\ (l \mid r) \rightarrow (\bullet_{\rho,\theta}\ l) \mid (\bullet_{\rho,\theta}\ r)}\ \text{(Parallel-Enter)}$$

$$\frac{}{(l \bullet^{v}_{\rho,\theta}) \mid r \rightarrow (l \mid r) \bullet^{v}_{\rho,\theta}}\ \text{(Parallel-PubL)}$$

$$\frac{}{l \mid (r \bullet^{v}_{\rho,\theta}) \rightarrow (l \mid r) \bullet^{v}_{\rho,\theta}}\ \text{(Parallel-PubR)}$$

The Parallel-Enter rule takes a token that is ready to execute the parallel combinator expression, and replaces it with two tokens within the combinator, to execute the left and right subexpressions of the combinator. The Parallel-PubL and Parallel-PubR rules take a token published from the left or right subexpressions (respectively), and replaces it with a token published from the overall parallel combinator.

In order to make progress executing the parallel combinator's left and right subexpressions l and r, *congruence* rules are required. These rules permit progress in each subexpression as if it was a stand-alone expression:

$$\frac{\phi \vdash l \rightarrow \phi \vdash l'}{\phi \vdash l \mid r \rightarrow \phi \vdash l' \mid r}\ \text{(Parallel-CongruL)}$$

$$\frac{\phi \vdash r \rightarrow \phi \vdash r'}{\phi \vdash l \mid r \rightarrow \phi \vdash l \mid r'}\ \text{(Parallel-CongruR)}$$

6.
Orc Semantics

Since Distributed Orc uses the non-distributed Orc language as its basis, we first describe the semantics of Orc, which is extended to Distributed Orc in chapter 9.

6.1. Object Language Symbols

The Orc token semantics operate over an object language that includes lexemes of the source program, plus tokens and prototokens. Tokens and prototokens incorporate values, environments, and tag sets. These are described below:

Abstract lexeme. The abstract syntax of the source program is composed from the following symbols: literal values, identifiers, `stop`, (,), >, `|`, {|, |}, $\overset{\text{val}}{=}$, `#`, `;`, and site declaration bodies.

For simplicity (without loss of generality), assume all identifiers are unique—shadowing does not occur.

A "site declaration body" specifies the location of the interface to an Orc site. This may reference a site written in Orc, or a site provided externally. Further details are implementation-specific, and not elaborated further here.

Value. A value is either a literal value, a value returned by a site call, or a closure. Orc literal values are integer literals, float literals, string literals, boolean literals, `signal`, and `null`. Many values in Orc programs are values that were published by site calls. Literals can be viewed as simple sites that always publish a constant value. Orc closures are a pair consisting of a reference to a site body, and the site's required argument cardinality.

Environment with futures. An environment is a mapping from identifiers to values or futures. A *future* indicates a value that has not yet been computed. We write futures

Expressions. e, l, r
Literal values (Constants). c
Variables. x; $\overline{x}_m$ is a sequence of m (≥ 0) variables, separated by commas
Arguments. a – a variable or literal; $\overline{a}_n$ is a sequence of n (≥ 0) arguments, separated by commas
Site declaration bodies. s
Closures. Pairs of the form $\langle s, m \rangle$
Values. v, v'
Environments. $\rho, \rho' \in \text{Variable} \rightarrow \text{Value}$
Cardinalities. $m, n \in \mathbb{N}$
Tag sets. θ, θ'

Figure 6.1. Metavariables used in Distributed Orc abstract grammar

as $\Diamond_\theta$, where θ is a unique *tag* that identifies the future. We write just $\Diamond$ to mean any future. The bound value of a future with tag θ is given by the function $\phi(\theta)$. This function will have no mapping for a tag until that future is bound. Once bound, the value is placed in the ϕ mapping and never changed. I.e., the ϕ mapping can be viewed as a partial map that grows towards a final state. Futures and the return values of sites are also allowed to have the distinguished value `stop` which signals that the computation has halted with publishing any value, so there will never be a value.

Tag set. A Tag set is an ordered set of opaque values (tags) which are used to identify groups of tokens. Sub-group membership is indicated by superset relationships.

Token. A token is a structure used by the rewrite rules that carries state. Tokens represent a current point of execution. The notation $\bullet^{v}_{\rho,\theta}$ represents a token with environment ρ, tag set θ, and (optionally) result v.

Prototoken. A prototoken is another structure used by the rewrite rules that carries state. Prototokens represent "suspended" tokens which cannot currently execute. The notation $\odot_{\rho,\theta}$ represents a prototoken with environment ρ and tag set θ.

6.2. Metavariables, Abstract Grammar, and Auxiliary Functions

The abstract grammar uses metavariables specified in figure 6.1. The semantic rules make use of the abstract grammar in figure 6.2. Language features omitted for brevity

$e ::= c$ – literal value
| x – variable
| `stop` – silent expression
| $x(\overline{a}_n)$ – site or function call
| $l >x> r$ – sequential
| $l \mid r$ – parallel
| $x \overset{\text{val}}{=} e_1 \# e_2$ – graft
| $\{\!|e|\!\}$ – trim
| $l \,;\, r$ – otherwise
| $\overline{\bullet}\, e$ – e is ready for execution
| $e\, \overline{\bullet}$ – execution of e completed
| $x\overline{\bullet}(\overline{a}_n)$ – call to x in progress

where $\overline{\bullet}$ is a possibly empty sequence of tokens or prototokens.

$a ::= c \mid x$ – argument to a call

Figure 6.2. Distributed Orc abstract grammar, as used in semantics rules

from the abstract grammar: (1) typing, (2) function declarations, (3) objects, and (4) virtual time.

The semantics use two auxiliary functions to manipulate tags: pushTag(θ) makes a new child of a tag θ. popTag(θ) returns the immediate parent of a tag θ.

$$\text{pushTag}(\theta) = \langle \text{fresh id}, \theta \rangle \qquad \text{popTag}(\langle \theta_1, \theta_2 \rangle) = \theta_2$$

The semantics also use two auxiliary functions to observe and erase groups of tokens: isLive(θ, e) is true if and only if there are tokens or prototokens in expression e that are tagged with θ or a descendant of θ.

$$\neg\, \text{isLive}(\theta, e) \Leftrightarrow \text{eraseTokens}(\theta, e) = e$$

eraseTokens(θ, e) returns the expression e after erasing all tokens and prototokens in e that are tagged with θ or a descendant of θ.

6.3. Rewrite Rules

Figure 6.3, figure 6.4, and figure 6.5 show the semantics' rewrite rules.

The rules describe members of the reduces relation $\rightarrow$, written as $\phi \vdash e \rightarrow \phi' \vdash e'$. The elements of the reduces relation (configurations) are pairs of the future-to-value mapping

ϕ and expressions e.

For site calls and returns, there are two special notations placed on the reduces relation arrow. The notation $\xrightarrow{s!\langle\theta,v_{1\ldots n}\rangle}$ indicates a call to the site s with arguments $v_{1\ldots n}$ and a call id θ is emitted to the "environment" to process. The id θ is solely used to associate calls to returned values. The notation $\xrightarrow{\theta?v}$ indicates that the site call with id θ returns value v. Site returns v can also be `stop`, indicating that the site call with id θ halted silently (i.e., without a return value).

The action of the rules can be summarized as follows: Site calls wait for any arguments that are futures to be bound, then notify the environment to execute the call (rule SiteCall-Issue). When the environment returns a result, that result is published (rule SiteCall-Return). If the site call halts silently, then the token is halted (SiteCall-Silent). The parallel combinator simply executes both subexpressions and publishes the publications of both (rules Parallel-Enter, Parallel-PubL and Parallel-PubR). The sequential combinator runs the left-hand subexpression (rule Sequential-Enter), and for each publication v, runs the right-hand subexpression with the variable bound to v (rule Sequential-PubL). The overall publications of the sequential combinator are those of the right-hand expression executions (rule Sequential-PubR).

The trim combinator begins execution (rule Trim-Enter); and then upon the first publication, publishes and halts further execution (rule Trim-Pub). The graft combinator begins execution of the value expression e_1, and body e_2 with the future unbound (rule Graft-Enter). Upon the first publication of the value expression e_1, the graft combinator binds the future (rule Graft-FirstPubL). Subsequent publications of the value expression are disregarded (rule Graft-SubsPubL). If the value expression halts without publishing, the future is bound to `stop` (rule Graft-NoPubL). The overall publications of the graft combinator are those of its body's execution (rule Graft-PubR). The otherwise combinator begins execution of its lefthand side (rule Otherwise-Enter), and if that side publishes one or more times, publishes those results and ignores its righthand side (rule Otherwise-PubL). However, if the lefthand halts silently (without publishing), the righthand side is

executed (rule Otherwise-NoPub), and those results are published (rule Otherwise-PubR).

The rules presented here omit Orc's function call facility, for simplicity of presentation. For our purposes here, function calls can be treated as equivalent to inlining the function body at call sites. However, Orc is capable of general mutual recursion. See the Orc references [35, 38] for details.

An Orc program begins execution with a single token at the root of the abstract syntax tree. Tokens that return to the root carry values that are considered publications of the whole program.

6.4. Order of Execution

Orc's semantics give the language runtime significant scheduling flexibility. The semantics specify possibly many next steps that the runtime is permitted to execute in any order convenient to it. For example, it is permissible to execute a parallel combinator by executing its lefthand side to completion, and only then starting to execute its righthand side. More generally, publication order is not guaranteed in Orc. If an expression publishes, for example, the values 1 then 2, the semantics do not require the 1 to be propagated to surrounding expressions before the 2.

Similarly, when multiple publications occur in a graft combinator's value expression, one will be used to bind the graft's future, but there are no requirements that the language implementation use any particular notion of a "first" publication to select for binding. This obviates the need for synchronization among tokens executing within a graft combinator.

When a publication occurs in a trim combinator, termination of execution does not need to occur promptly. The language implementation need only ensure that it appears so—namely, that only one publication will emerge from the trim combinator and that no further site calls are made. The termination, shown in the Trim-Pub rule's application of the eraseTokens function, may be performed without blocking other progress, provided that the language runtime maintains additional state to prevent further publications or

site calls.

6.5. Locality

Anticipating the use of Orc as the basis for Distributed Orc, we discuss the locality, or more importantly, the non-locality of operations specified by the Orc semantics.

The scope of observations made by operations in the Orc semantics can be placed in one or more of four categories:

Token-local observations. Observing the values within tokens or the position (node) of tokens in the program AST.

First publication detection. Detecting the first publication by any token in an identified group of tokens located in a given AST subtree.

Future look-up. Looking up a given future in the future-to-value mapping ϕ to find its bound value, or that it has not been bound yet.

Group liveness observation. Determining if any token in an identified group of tokens is still executing within a given AST subtree. This appears in the rules as applications of the isLive function.

The scope of effects of operations in the Orc semantics can be placed in one or more of five categories:

Single token effects. Effects of the operation affect state or location of one token. These effects have simple atomicity requirements.

Double token effects. Effects of the operation affect two tokens, or a token and a prototoken. Each instance of these effects require that one token's updates are applied before (or at least simultaneously with) the other token's updates. The effects on the two involved tokens do *not* require synchronization.

Table 6.1. Scope of observations and scope of effects (locality/non-locality) of the Orc combinators. Token-local observations and single token effects are assumed for all combinators, and are not shown here.

Combinator	Scope of observations	Scope of effects
Parallel	—	Double token effects
Sequential	—	—
Trim	First publication observation	Group kill
Graft	First publication detection and Group liveness observation	Future binding
Otherwise	Group liveness observation	Double token effects

Future binding. The future-to-value mapping ϕ has one new entry added. This is the only type of change that is made to the future-to-value mapping ϕ. The changes create an increasing sequence of maps—all updates of ϕ to ϕ' preserve $\phi \subset \phi'$.

Group kill. An AST subtree has a group of tokens (identified by a tag) erased. This appears in the rules as applications of the eraseTokens function. This class is the largest-scale scope of effects, but the Orc semantics are preserved even when the erasures are performed non-atomically (as described above for the trim combinator). (Note: The rules in figure 6.4 are written assuming eager erasure.)

The scope of observations and scope of effects of the Orc combinators is given in table 6.1. The parallel and sequential combinators require no coordination beyond simple atomicity, but the trim, graft, and otherwise combinators require coordination beyond the involved tokens. Execution of a trim combinator requires that the first publication from the execution trigger a group kill. Execution of a graft combinator requires that the first published value from the execution be used to bind the future. Execution of an otherwise combinator requires that silent halting be detected.

When a future is looked up in the future-to-value mapping ϕ, this is a non-local observation, but the semantics only require that binding of a future *eventually* be observed. The semantics are preserved if there is a delay between binding of a future and the value being available to an observer.

$$\frac{}{\phi \vdash \bullet_{\rho,\theta}\ c \to \phi \vdash c\ \bullet^{c}_{\rho,\theta}} \quad \text{(Literal)}$$

$$\frac{x \in \rho \qquad \rho(x) \neq \Diamond}{\phi \vdash \bullet_{\rho,\theta}\ x \to \phi \vdash x\ \bullet^{\rho(x)}_{\rho,\theta}} \quad \text{(Variable)}$$

$$\frac{}{\phi \vdash \bullet_{\rho,\theta}\ \texttt{stop} \to \phi \vdash \texttt{stop}} \quad \text{(Silent)}$$

$$\frac{\rho(x) = s \qquad v_{1\ldots n} \neq \Diamond}{\phi \vdash \bullet_{\rho,\theta}\ x(a_{1\ldots n}) \xrightarrow{s!\langle \theta', v_{1\ldots n}\rangle} \phi \vdash x\ \bullet_{\rho,\theta'}\ (a_{1\ldots n})} \quad \text{(SiteCall-Issue)}$$
$$\text{where } \theta' = \text{pushTag}(\theta) \quad v_i = \rho(a_i)$$

$$\frac{v \neq \texttt{stop}}{\phi \vdash x \bullet_{\rho,\theta}(a_{1\ldots n}) \xrightarrow{\theta?v} \phi \vdash x(a_{1\ldots n}) \bullet^{v}_{\rho,\theta'}} \quad \text{(SiteCall-Return)}$$
$$\text{where } \theta' = \text{popTag}(\theta)$$

$$\frac{v = \texttt{stop}}{\phi \vdash x\ \bullet_{\rho,\theta}\ (a_{1\ldots n}) \xrightarrow{\theta?v} \phi \vdash x(a_{1\ldots n})} \quad \text{(SiteCall-Silent)}$$

$$\frac{}{\phi \vdash \bullet_{\rho,\theta}\ (l \mid r) \to \phi \vdash (\bullet_{\rho,\theta}\ l) \mid (\bullet_{\rho,\theta}\ r)} \quad \text{(Parallel-Enter)}$$

$$\frac{}{\phi \vdash \bullet_{\rho,\theta}(l >x> r) \to \phi \vdash (\bullet_{\rho,\theta}\ l) >x> r} \quad \text{(Sequential-Enter)}$$

$$\frac{}{\phi \vdash (l \bullet^{v}_{\rho,\theta}) \mid r \to \phi \vdash (l \mid r) \bullet^{v}_{\rho,\theta}} \quad \text{(Parallel-PubL)}$$

$$\frac{}{\phi \vdash l \bullet^{v}_{\rho,\theta}\ >x> r \to \phi \vdash l >x>\ \bullet_{\rho',\theta}\ r} \quad \text{(Sequential-PubL)}$$
$$\text{where } \rho' = \rho[x \mapsto v]$$

$$\frac{}{\phi \vdash l \mid (r \bullet^{v}_{\rho,\theta}) \to \phi \vdash (l \mid r) \bullet^{v}_{\rho,\theta}} \quad \text{(Parallel-PubR)}$$

$$\frac{}{\phi \vdash l >x> (r \bullet^{v}_{\rho,\theta}) \to \phi \vdash (l >x> r) \bullet^{v}_{\rho',\theta}} \quad \text{(Sequential-PubR)}$$
$$\text{where } \rho' = \rho \setminus \{x\}$$

$$\frac{\rho(x) = \Diamond_{\theta'} \qquad \phi(\theta') \neq \texttt{stop} \qquad \phi(\theta') = v}{\begin{array}{ll}\phi \vdash \bullet_{\rho,\theta,k}\ x & \to \phi \vdash \bullet_{\rho',\theta,k}\ x \\ \phi \vdash \bullet_{\rho,\theta,k}\ m(\ldots,x,\ldots) & \to \phi \vdash \bullet_{\rho',\theta,k}\ m(\ldots,x,\ldots)\end{array}} \quad \text{(Future)}$$
$$\text{where } \rho' = \rho[x \mapsto v]$$

$$\frac{\rho(x) = \Diamond_{\theta'} \qquad \phi(\theta') = \texttt{stop}}{\begin{array}{ll}\phi \vdash \bullet_{\rho,\theta,k}\ x & \to \phi \vdash\ x \\ \phi \vdash \bullet_{\rho,\theta,k}\ x(\ldots) & \to \phi \vdash\ x(\ldots) \\ \phi \vdash \bullet_{\rho,\theta,k}\ m(\ldots,x,\ldots) & \to \phi \vdash\ m(\ldots,x,\ldots)\end{array}} \quad \text{(Future-Stop)}$$

Figure 6.3. Orc semantics, part 1 of 3. Rules for trim, graft, and otherwise are in part 2.

$$\frac{}{\phi \vdash \bullet_{\rho,\theta}\ \{|e|\} \to \phi \vdash \{|\ \bullet_{\rho,\theta'}\ e|\}}\ \text{(Trim-Enter)}$$
where $\theta' = \text{pushTag}(\theta)$

$$\frac{}{\phi \vdash \{|e\ \bullet^{v}_{\rho,\theta}\ |\} \to \phi \vdash \{|e'|\}\ \bullet^{v}_{\rho,\theta'}}\ \text{(Trim-Pub)}$$
where $e' = \text{eraseTokens}(\theta, e)$ $\quad \theta' = \text{popTag}(\theta)$

$$\frac{}{\phi \vdash \bullet_{\rho,\theta}\ (x \overset{\text{val}}{=} e_1\ \#\ e_2) \to \phi \vdash x \overset{\text{val}}{=} (\bullet_{\rho,\theta'}\ e_1)\ \#\ (\bullet_{\rho',\theta}\ e_2)}\ \text{(Graft-Enter)}$$
where $\theta' = \text{pushTag}(\theta)$ $\quad \rho' = \rho[x \mapsto \Diamond_{\theta'}]$

$$\frac{\theta \notin \phi}{\phi \vdash x \overset{\text{val}}{=} (e_1\ \bullet^{v}_{\rho,\theta})\ \#\ e_2 \to \phi' \vdash x \overset{\text{val}}{=} e_1\ \#\ e_2}\ \text{(Graft-FirstPubL)}$$
where $\phi' = \phi[\theta \mapsto v]$

$$\frac{\theta \in \phi}{\phi \vdash x \overset{\text{val}}{=} (e_1\ \bullet^{v}_{\rho,\theta})\ \#\ e_2 \to \phi \vdash x \overset{\text{val}}{=} e_1\ \#\ e_2}\ \text{(Graft-SubsPubL)}$$

$$\frac{\neg\,\text{isLive}(\theta, e_1)}{\phi \vdash x \overset{\text{val}}{=} e_1\ \#\ e_2 \to \phi' \vdash x \overset{\text{val}}{=} e_1\ \#\ e_2}\ \text{(Graft-NoPubL)}$$
where $\phi' = \phi[\theta \mapsto \texttt{stop}]$

$$\frac{}{\phi \vdash x \overset{\text{val}}{=} e_1\ \#\ (e_2\ \bullet^{v}_{\rho,\theta}) \to \phi \vdash (x \overset{\text{val}}{=} e_1\ \#\ e_2)\ \bullet^{v}_{\rho',\theta}}\ \text{(Graft-PubR)}$$
where $\rho' = \rho \setminus \{x\}$

$$\frac{}{\phi \vdash \bullet_{\rho,\theta}\ (l\ ;\ r) \to \phi \vdash (\bullet_{\rho,\theta'}\ l)\ ;\ (\odot_{\rho,\theta'}\ r)}\ \text{(Otherwise-Enter)}$$
where $\theta' = \text{pushTag}(\theta)$

$$\frac{\neg\,\text{isLive}(\theta, l)}{\phi \vdash l\ ;\ \odot_{\rho,\theta}\ r \to \phi \vdash l\ ;\ \bullet_{\rho,\theta}\ r}\ \text{(Otherwise-NoPub)}$$

$$\frac{}{\phi \vdash (l\ \bullet^{v}_{\rho,\theta})\ ;\ r \to \phi \vdash (l\ ;\ r')\ \bullet^{v}_{\rho,\theta'}}\ \text{(Otherwise-PubL)}$$
where $r' = \text{eraseTokens}(\theta, r)$ $\quad \theta' = \text{popTag}(\theta)$

$$\frac{}{\phi \vdash l\ ;\ (r\ \bullet^{v}_{\rho,\theta}) \to \phi \vdash (l\ ;\ r)\ \bullet^{v}_{\rho,\theta'}}\ \text{(Otherwise-PubR)}$$
where $\theta' = \text{popTag}(\theta)$

Figure 6.4. Orc semantics, part 2 of 3. Rules for trim, graft, and otherwise combinators.

$$\frac{\phi \vdash l \to \phi' \vdash l'}{\phi \vdash l >x> r \to \phi' \vdash l' >x> r} \text{ (Sequential-CongruL)}$$

$$\frac{\phi \vdash r \to \phi' \vdash r'}{\phi \vdash l >x> r \to \phi' \vdash l >x> r'} \text{ (Sequential-CongruR)}$$

$$\frac{\phi \vdash l \to \phi' \vdash l'}{\phi \vdash l \mid r \to \phi' \vdash l' \mid r} \text{ (Parallel-CongruL)}$$

$$\frac{\phi \vdash r \to \phi' \vdash r'}{\phi \vdash l \mid r \to \phi' \vdash l \mid r'} \text{ (Parallel-CongruR)}$$

$$\frac{\phi \vdash e_1 \to \phi' \vdash e_1'}{\phi \vdash x \overset{\text{val}}{=} e_1 \ \# \ e_2 \to \phi' \vdash x \overset{\text{val}}{=} e_1' \ \# \ e_2} \text{ (Graft-CongruL)}$$

$$\frac{\phi \vdash e_2 \to \phi' \vdash e_2'}{\phi \vdash x \overset{\text{val}}{=} e_1 \ \# \ e_2 \to \phi' \vdash x \overset{\text{val}}{=} e_1 \ \# \ e_2'} \text{ (Graft-CongruR)}$$

$$\frac{\phi \vdash e \to \phi' \vdash e'}{\phi \vdash \{\!|e|\!\} \to \phi' \vdash \{\!|e'|\!\}} \text{ (Trim-Congru)}$$

$$\frac{\phi \vdash l \to \phi' \vdash l'}{\phi \vdash l \ ; r \to \phi' \vdash l' \ ; r} \text{ (Otherwise-CongruL)}$$

$$\frac{\phi \vdash r \to \phi' \vdash r'}{\phi \vdash l \ ; r \to \phi' \vdash l \ ; r'} \text{ (Otherwise-CongruR)}$$

Figure 6.5. Orc semantics, part 3 of 3. Congruence rules.

7.
Current Orc Implementations

At present, the Orc research group at the University of Texas at Austin maintains two implementations of Orc. Several other groups have also implemented Orc, for example Aldinucci et al. [2], AlTurki and Meseguer [3], Launchbury and Elliott [36].

7.1. Orc Compiler

The UT Austin Orc compiler is architecturally conventional—it is composed of phases that, in aggregate, transform input program text into Orc Intermediate Language (OIL). Each phase passes an abstract syntax tree (AST) to its successor (see figure 7.1). Initially, the input program text is processed by a conventional lexical scanner and parser, resulting in an AST very close to the input language. This *extended* AST contains many language constructs that can be reduced to primitive forms, such as if–then–else statements. The extended AST is then desugared and transformed into the Orc Intermediate Language (OIL). OIL is the essential language that is specified by the Orc formal semantics. OIL is an AST representation of the language shown in figure 6.2.

The compiler processes the OIL AST in several additional phases. The program can optionally be type checked. Various errors and warnings are generated, such as possible infinite recursion. Unused functions are pruned from the program. Finally, the OIL AST is transformed to a nameless OIL AST form, in the spirit of de Bruijn's nameless dummies [20], to simplify execution.

7.2. Token-Driving Interpreter

The standard Orc runtime engine is an AST interpreter. It takes the output of the standard Orc compiler, namely an OIL AST, and applies the formal semantics rules of figure 6.3, figure 6.4, and figure 6.5.

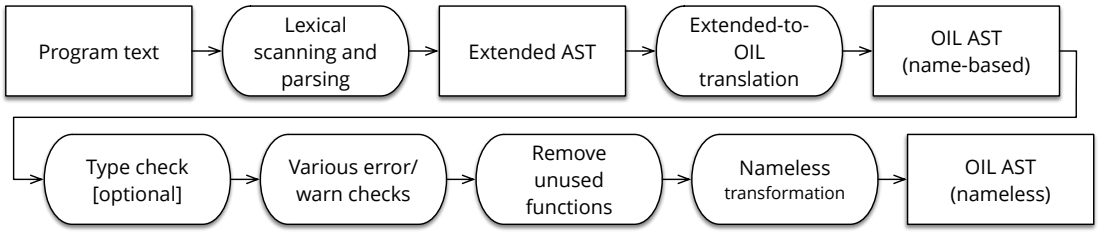


Figure 7.1. Orc compiler major phases

Since the OIL AST is a representation of the abstract grammar of the formal semantics, and the execution state is represented in similar tokens, the execution steps that the token-driving interpreter makes are very similar to the rules of the token semantics. One can examine the token semantics rules and the interpreter code and produce a fairly direct mapping between them.

The token-driving interpreter is implemented in the Scala programming language, and runs in the Java runtime environment version 8 or later.

To begin execution, the OIL AST is loaded. During OIL AST loading, the sites are loaded from the site class path, and references to sites are bound. The Orc runtime scheduler is initialized. A top-level token group is created to contain the initial token. Publications that reach the top-level group are publications of the program, and handed to the environment to process (typically, print to `stdout`). The initial token is created, positioned to begin execution at the root of the loaded AST, and scheduled for execution.

When a token from the scheduling queue is run, it firstly checks its enclosing group's status, to determine if it has been killed. If the token is blocked, it checks to see if its needed values are available. Otherwise, the token is "live", and it performs the action specified by the AST node where the token is currently located (see figure 7.2). For example, if the current AST node is a parallel combinator, the token copies itself and places the token copies on the left and right child subexpression nodes.

When a token enters certain contexts, the context is pushed on the token's context stack. When the token publishes a value, the token is propagated to successive enclosing

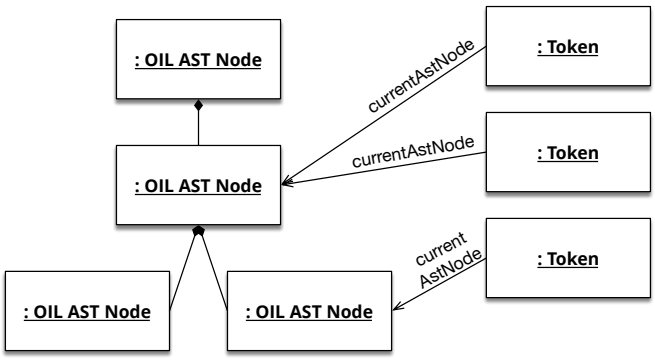


Figure 7.2. A small OIL AST with 3 tokens executing

contexts (i.e., up the stack) until the publishing token is consumed. The types of contexts tracked on the token's stack are: a binding scope, a lefthand subexpression of a sequence combinator, a graft combinator value expression, a trim combinator subexpression, and a lefthand subexpression of an otherwise combinator. Publications propagating out of a binding's scope have that binding popped off its environment stack, and publishing propagation continues. Publications propagating out of the combinator contexts listed above are consumed and used in the operation of the combinator's semantics. For example, a publication in a graft combinator's value expression is used to bind the future to the published value.

Certain combinators, namely graft, trim, and otherwise, need to track when execution halts within the combinator's subexpressions. For example, when the lefthand subexpression of an otherwise combinator halts silently (without publishing), the combinator's righthand subexpression must be executed. This halt tracking is done using token groups (see figure 7.3). Every token is a member of a group, and groups have a parent, forming a program's group tree. If a token halts, it notifies its parent group. When all tokens in a group have halted, the group's parent group is notified that the group has halted. Groups are also used to propagate forced token terminations (kills), for example, by the trim combinator.

Orc utilizes services in its execution environment by making calls to sites. When a

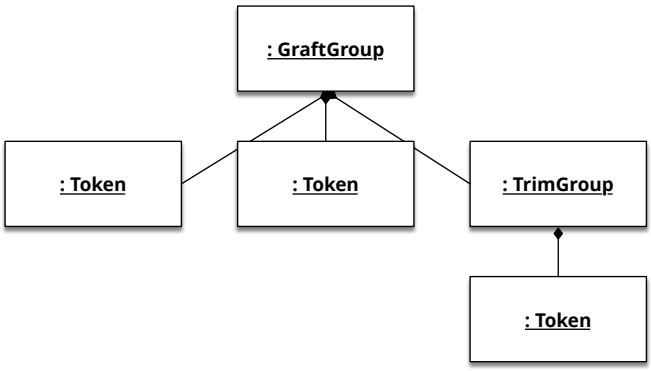


Figure 7.3. A token group tree while an expression such as `val x = f() | g() | {| h() |}` is executing.

site call AST node is reached by a token, the call waits (blocks) until the value of the site call target and all call arguments are available. (The site call target or arguments may be futures that have not been bound yet.) Upon availability of all the values, the site call is dispatched to the site. Orc sites are a JVM class that implements the Orc `Site` interface. Orc also has an Orc-to-Java bridge that enables execution of plain Java objects' constructors and methods as Orc site calls. Orc site calls can publish zero, one, or multiple times. Publications can occur over time as the site call is executing—i.e., a site call produces a (possibly empty) stream of publications. The site may inform Orc when the call has halted. If the called site throws an exception, the site call halts.

When an Orc expression evaluates a future, if the future is unbound, the executing token will block, awaiting the future's value. The blocked token will be added to the future's list of waiting readers. When the future is bound, all waiting readers are rescheduled to continue their execution. If a future is bound to **`stop`**, i.e. there is no value, then readers will halt silently.

7.3. PorcE

A higher-performance Orc implementation is PorcE [47, 48], which implements Orc using a static compiler, a dynamic compiler, and a work-stealing runtime with future support.

The compiler performs a series of translations to produce the PorcE execution language, PEL.

The PorcE static compiler uses a value flow analysis to determine which futures and future force operations (waits for values) can be eliminated. The compiler then translates OIL into a simple functional (with continuation-passing) language, which is much closer to typical target platforms, such as the Java VM. The PorcE runtime then uses profile-based concurrent optimizations and traditional dynamic optimizations, such as speculative inlining. Finally, the running code is just-in-time (JIT) compiled to native code by partially evaluating the interpreter and dynamically optimized using the Truffle language framework and the Graal virtual machine (VM) [65].

PorcE is implemented in the Scala programming language, and runs in the GraalVM high-performance Java runtime environment.

8.
Extending Orc with Transparent Distribution

Distributed Orc implements the same language and semantics as Orc, but allows programs to execute across multiple locations. To allow this, Distributed Orc tracks the locations on which data reside and uses this information to determine locations on which to execute program operations. Some data may reside on sets of locations, instead of just a single location.

As a simple example, take `f() + g()`, where the site `f` is available at locations in set A and site `g` is available at locations in set B. Any location in the intersection $A \cap B$ can execute the whole expression without any need for communication. If $A \cap B = \varnothing$ or if the current execution is not in $A \cap B$, then communication is required to execute the expression. For example, `f()` could be run on a location in A, and the program could migrate to a location in B, where the `g()` and `+` operations can be performed.

8.1. Additions to Orc

Distributed Orc builds on the asynchrony and failure-awareness of Orc, and makes additions that can be grouped into three areas:

Migration. Migrating points of execution in the program among distributed locations.

Value–location relation. Identifying the locations of values in the distributed environment. Distributed Orc tracks both the current locations of a value and the permitted locations of a value.

Revised semantics. Extending existing Orc language features to handle remote values, namely: site calls, the trim combinator, the graft combinator, and the otherwise combinator. All other Orc language features operate locally, and therefore are unchanged in Distributed Orc.

Migration of execution in Distributed Orc is driven by the locations of values used in site calls, namely the site and the call arguments. If a site call is to be executed, but the current location does not have local copies of these values, Distributed Orc can copy values between locations (if allowed), migrate execution to another location, or some combination thereof. The locations of existing copies, costs of communication, and any policy constraints on allowed locations of copies may be taken into account by the Distributed Orc implementation when choosing which actions to take.

The trim, graft, and otherwise combinators must be distribution-aware. Execution of a trim combinator `{| e |}` needs to coordinate the selection of the first publication from its subexpression `e` among all locations executing parts of `e`, and notify other affected points of execution of this selection. Execution of a graft combinator **`val`** `x =` e_1 `#` e_2 needs to coordinate the selection of the first publication from its subexpression e_1 among all locations executing parts of e_1. Similarly, execution of an otherwise combinator e_1 `;` e_2 must monitor execution, across any involved locations, of its lefthand expression e_1 for publications and termination.

We present a formalization of Distributed Orc semantics in chapter 9.

Earlier, in-progress results of this work have been reported in several workshops [46, 56–58].

8.2. Examples of Distributed Orc

We present three examples of using Distributed Orc as demonstrations of transparent distribution. An evaluation of the usability of Distributed Orc appears in section 12.1.

8.2.1. Web Site Registration Form

A Web application is an example of a simple distributed system, with the locations being the user's Web browser and the Web server. Most operations of the Web application must execute in part on the browser and in part on the server. For example, a Web form asking a user to select a username needs to validate the username in multiple ways, some of

Listing 8.1. A user account creation process expressed in Distributed Orc.

```
askUser("Pick a username")  >username>
if isLegal(username) && isUnique(username) then
  createNewUser(username)
else
  displayError(username)
```

which must be done on the server. Listing 8.1 shows how such a form is implemented in Orc. This program is simple and cohesive, but when it is executed in Distributed Orc, location constraints cause the execution to be implicitly distributed. The `askUser` and `displayError` sites must execute in the browser's environment, and the `isUnique` and `createNewUser` sites must execute on the Web server. `isLegal` may execute anywhere. These constraints are supplied as part of the declaration of these sites. The constraints for this example are summarized in table 8.1.

Execution of the user account creation program starts with a call to `askUser`. Since the `askUser` site is located at the browser, the execution migrates there and prompts the user to enter a username. The call to `isLegal` is executed on the browser. If `isLegal` produces **`true`** execution migrates to the server (with all needed values) to execute `isUnique` and the logical-and (`&&`). Otherwise, execution continues on the browser, because logical-and is shortcutting and can execute on the browser. Based on the result, either the `createNewUser` call is made without migrating, or execution calls `displayError` migrating back to the browser if execution is not already on the browser.

Table 8.1. Permissible locations of site values in the user account creation process example

Site name	Permissible locations
`askUser`	`webBrowser`
`isLegal`	`webBrowser`, `webServer`
`isUnique`	`webServer`
`createNewUser`	`webServer`
`displayError`	`webBrowser`

Listing 8.2. User account creation process: Browser side (JavaScript)

```
function attemptCreateNewUser(username, callback) {
  ...  RPC call ...
}
var username = askUser("Pick a username");
function displayErrorIfNeeded(success) { // Callback
  if (!success) displayError(username);
}
if (isLegal(username))
  attemptCreateNewUser(username, displayErrorIfNeeded);
else displayErrorIfNeeded(true);
```

Listing 8.3. User account creation process: Server side (Scala)

```
def attemptCreateNewUser(username) = {
  if (isUnique(username)) {
    createNewUser(username)
    true
  } else false
}
```

To implement the same execution in a typical Web programming environment, the programmer needs to divide the program into two parts and handle the remote calls differently from local calls. Listing 8.2 and Listing 8.3 show this pattern using JavaScript on the browser and Scala on the server. This manually implements the same execution and migration pattern as the Distributed Orc implementation produces. The browser begins the process by requesting a username and checking if it is legal. If the username is legal, the browser calls the server with a callback to handle the result. If the username is illegal, the browser calls the callback directly. The server checks that the username is unique and creates the user if it is. Finally, the server returns whether it succeeded and the browser callback displays the result.

Distributed Orc has simplified the remote calls by providing native support for concurrency. However, local calls in Distributed Orc have not become more complex due to their

Listing 8.4. Collaborative route finding in Distributed Orc

```
def findRegionsOfInterestOnRoverIfPossible(img) =
  {|
    at(rover) >> findRegionsOfInterest(img) |
    Rwait(5 * seconds) >> findRegionsOfInterest(img)
  |}
def findRoute(from, to) =
  val img = photographPosition(from)
  val regionData = map(lambda(r) = (r, additionalMeasurements(r)),
                       findRegionsOfInterestOnRoverIfPossible(img))
  val route = planRoute(from, regionData, to)
  if getEndPoint(route) /= to then
    concatenate(route, findRoute(getEndPoint(route), end))
  else
    route
```

asynchronous semantics. Instead, Orc's semantics allows programmers to easily control the asynchrony when needed. In addition, Distributed Orc has allowed the programmer to write the program cohesively instead of as two communicating modules.

8.2.2. Robot Collaboration

In autonomous robotics, all programs need to execute without a reliable central controller. However, many collaborations between robots are more easily represented as a single cohesive program. In traditional programming environments, a cohesive program must execute in a single location. Distributed Orc's location transparency and migration support enables us to resolve this conflict. As an example, consider a rover and an Unmanned Aerial Vehicle (UAV) collaborating to determine a safe route for the rover to drive from one point to another. The rover has greater computing power and battery capacity than the UAV, but the UAV may not always be in the communications range of the rover to make use of those resources.

Listing 8.5. Collaborative route finding in Scala: Rover code

```
def measureForRoute(from, to) = ... RPC call ...
def findRoute(from, to) = {
  measureForRoute(from, to) onSuccessful { regionData =>
    val route = planRoute(from, regionData, to)
    if (getEndPoint(route) != to)
      concatenate(route, findRoute(getEndPoint(route), end))
    else
      route
  }
}
```

Listing 8.6. Collaborative route finding in Scala: UAV code

```
def findRegionsOfInterestOnRover(img, terminator) = ... RPC call ...
def findRegionsOfInterestOnRoverIfPossible(img) = {
  val result = Promise()
  val terminator = TerminationFlag()
  val roverCall = findRegionsOfInterestOnRover(img, terminator)
  roverCall.onSuccessful(v => result.success(v))
  timer.schedule(5 * seconds, { () =>
    result.success(findRegionsOfInterest(img, terminator))
  })
  result.future.onSuccessful(_ => terminator.set())
  result.future
}
def measureForRoute(from, to) = {
  val img = photographPosition(from)
  findRegionsOfInterestOnRoverIfPossible(img) onSuccessful { rois =>
    rois map { reg => (reg, additionalMeasurements(reg)) }
  }
}
```

Listing 8.4 shows the Distributed Orc implementation of this collaborative route planning problem in terms of a set of high-level primitive sites. Two sites, `photographPosition` and `additionalMeasurements`, take a photograph of a location and take measurements of a region, respectively. These sites use sensors on the UAV, so they must run on the UAV. The `planRoute` site computes a route, given starting and ending points and information about regions in between. The site `planRoute` requires too much memory to run on the UAV, so it is only available on the rover. Finally, the `findRegionsOfInterest`

site searches an image for regions that are worth further investigation. It is a fast approximate algorithm that can run on either the UAV or the rover, but it is faster on the rover and consumes enough power to be a significant drain on the UAV's limited power capacity. So, calls to it should be run on the rover when possible.

The `findRoute` function captures an image of the starting area and other measurements and then computes a route based on the resulting data. Then, `findRoute` recursively calls itself to find the rest of the route to the target if the whole route is not visible in one aerial photograph. The auxiliary function `findRegionsOfInterestOnRoverIfPossible` migrates to the rover and calls `findRegionsOfInterest`. However, if this migration is delayed for more than 5 seconds or fails (for instance, because the rover is out of reliable communication range), the program calls `findRegionsOfInterest` on the UAV. The first result to arrive is used and then the other computation is terminated.

The local and remote operations in Listing 8.4 are handled consistently within a single cohesive module. This allows the programmer to handle errors consistently as well.

Listing 8.5 and Listing 8.6 show an implementation of the same collaboration in Scala using Scala's standard concurrency primitives, a high level RPC library, and a distributed flag `TerminationFlag` (to communicate termination requests between locations). Unlike the Distributed Orc implementation, the `findRoute` operation is no longer implemented cohesively. In addition, the remote calls in Scala require future handling while local calls to not allow it, forcing the programmer to manage two mixed programming styles. Finally, manual concurrency and termination handling is needed throughout the program.

8.2.3. Data Processing Patterns Imply Distribution

In Distributed Orc, unmodified functional data processing operations, including map and fold, can execute in a distributed fashion. The example of a distributed data processor in Listing 8.7, shows this in action. The function `combineDataAndUpdate` takes a

Listing 8.7. Distributed data processing in Distributed Orc

```
type Data = Block BlobRef
          | Alternatives [Data]
def combineDataAndUpdate(combine, ctx, Block(item1), Block(item2)) =
  combine(ctx, item1, item2)
def combineDataAndUpdate(combine, ctx, Alternatives(alts), block@Block(_)) =
  foldl(lambda(c, a) = combineDataAndUpdate(combine, c, a, block), ctx, alts
    )
-- Symmetrical case for Alternatives on the right.
```

combining function, an initial processing context, and two datasets. It recursively combines the datasets using the function. Each call to the function produces a new context that is passed to the next call (like the accumulator in a fold operation). An example of a combine function would be a log analysis task which needs to compare and merge the events that occurred in multiple logs.

If the dataset storage is distributed across multiple data centers, the traversal defined by `combineDataAndUpdate` will migrate from data center to data center as needed. The

Listing 8.8. Distributed data processing in Scala

```
def combineDataAndUpdate(locationFor, combine, ctx, d1, d2) =
  (d1, d2) match {
    case (Block(item1), Block(item2)) => combine(ctx, item1, item2)
    case (Alternatives(alts), block@Block(_)) => {
      val result = Promise()
      def movingFold(ctx, alts) = alts match {
        case a :: rest =>
          locationFor(block, a).runAt {
            combineDataAndUpdate(locationFor, combine, ctx, a, block)
              .onSuccessful { ctx1 => movingFold(ctx1, rest) }
          }
        case Seq() => result.success(ctx)
      }
      movingFold(ctx, alts)
      result.future
    }
    // Symmetrical case for Alternatives on the right.
  }
```

context will only be copied between data centers when migration is required to bring the computation near the new data that it is processing.

A Scala implementation of the same program in Listing 8.8 produces the same execution pattern manually. Since most of the calls may be remote, the programmer needs to handle failure and asynchrony throughout the program. In addition, the `combineDataAndUpdate` function needs to take an additional argument `locationFor` to tell it where each computation should occur.

8.3. Emulating Map–Reduce Without a Map–Reduce Framework

Listing 8.9 is the complete source code of a program that counts words in a given list of text files. It operates by mapping the word counting functions over the file list. This program operates as a local Orc program.

However, by running the program in Distributed Orc, and supplying a location policy for the files (Listing 8.10), the processing occurs with the same communication pattern as a program written within a map–reduce framework [21], such as Hadoop [28]. Note that this occurs *with no changes to the local-only variant of the program*.

The word count program running on a Distributed Orc cluster has the message communication pattern illustrated in figure 8.1.

Listing 8.9. An Orc program that counts words in a give list of text files.

```
val inputList = ["file1.txt", "file2.txt", "file3.txt", "file4.txt"]

def countLine(line) =
  import class BreakIterator = "java.text.BreakIterator"
  import class Character = "java.lang.Character"
  def containsAlphabetic(s, startPos, endPos) =
    Character.isAlphabetic(s.codePointAt(startPos)) || (if startPos+1 <:
    endPos then containsAlphabetic(s, startPos+1, endPos) else false)
  def wordCount'(startPos, wb, accumCount) =
    wb.next()  >endPos>
    (if endPos <: 0 then accumCount else (if containsAlphabetic(line,
    startPos, endPos) then wordCount'(endPos, wb, accumCount + 1) else
    wordCount'(endPos, wb, accumCount))) #
  BreakIterator.getWordInstance() >wb>
  wb.setText(line)  >>
  wordCount'(0, wb, 0)

def countFile(file) =
  import class Files = "java.nio.file.Files"
  def countLinesFrom(in, accumCount) =
    (in.readLine() ; null)  >nextLine>
    (if nextLine = null then accumCount else countLinesFrom(in, accumCount +
     countLine(nextLine))) #
  Files.newBufferedReader(file)  >in>
  countLinesFrom(in, 0)  >count>
  in.close()  >>
  count

def countFilename(pathname) =
  import class Paths = "java.nio.file.Paths"
  def sumN(n, f) = if (n :> 0) then f() + sumN(n-1, f) else 0

  Paths.get(pathname)  >file>
  countFile(file)

map(countFilename, inputList)  >wordCountList>
afold((+), wordCountList)
```

Listing 8.10. File location policy for the word count example (excerpt) (Scala).

```
class FileValueLocator[L <: AbstractLocation](locations: ClusterLocations[L
    ]) extends ValueLocator[L] with CallLocationOverrider[L] {
  private val numLocations = locations.allLocations.size

  /* Map files to locations 1 ... n-1, skipping leader (0) */
  val filenameLocationMap = {
    val locIndex = locations.allLocations.toSeq.sortWith({ (x, y) => x.
    hashCode.compareTo(y.hashCode) < 0 })
    1.to(4).map(i => (s"file$i.txt", locIndex((i % numLocations - 1) + 1)))
    .toMap ++
  }
// . . . . . .
  override val permittedLocations: PartialFunction[Any, Set[L]] = {
    case f: File if filenameLocationMap.contains(f.getName) => Set(
    filenameLocationMap(f.getName))
    case p: Path if filenameLocationMap.contains(p.getFileName.toString) =>
    Set(filenameLocationMap(p.getFileName.toString))
  }
// . . . . . .
}
```

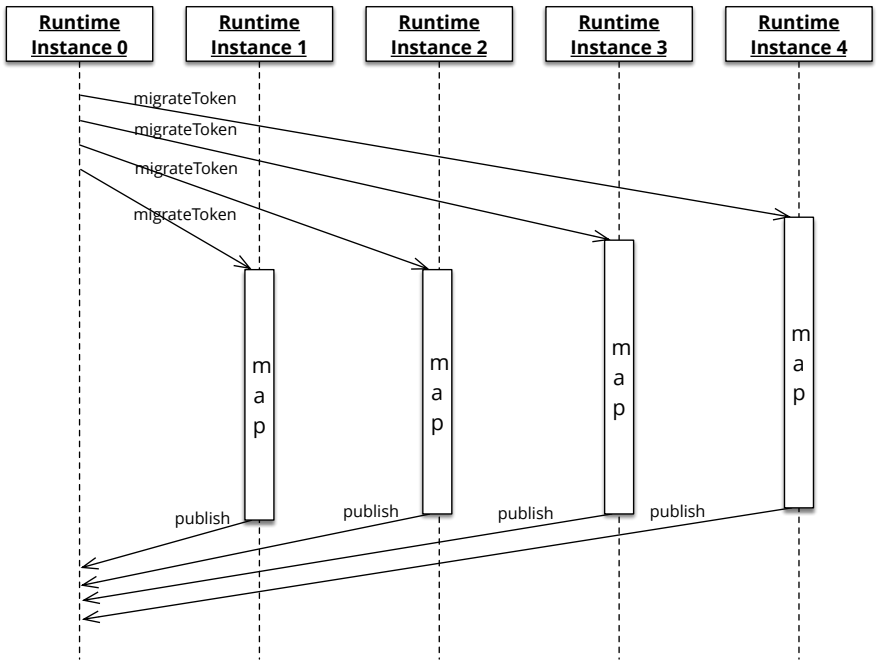


Figure 8.1. Communication pattern for word count running on a Distributed Orc cluster. Includes token migration messages; elides future and halt messaging.

9.
Distributed Orc Operational Semantics

The Distributed Orc semantics modify the Orc semantics of figure 6.3, figure 6.4, and figure 6.5 with enhanced notation, an enhanced reduces relation, and new rewrite rules.

9.1. Extended Notation

We add locations to tokens and prototokens, and introduce new policy and location maps.

Location. Locations are written k and come from the set of locations Loc.

Token. A token represents a current point of execution. Token notation is extended with a location k to $\bullet^{v}_{\rho,\theta,k}$, which represents a token with an optional published value v, environment ρ, and tag θ, at location k.

Prototoken. A prototoken represents a "suspended" token which cannot currently execute. The notation $\odot_{\rho,\theta,k}$ represents a prototoken with environment ρ and tag θ at location k.

Policy for values. $\mathscr{P}(v) \in 2^{\text{Loc}}$ is the policy set for value v — the permitted locations for copies of v.

Location of values. $\mathscr{L}(v) \in 2^{\text{Loc}}$ is a set of locations with copies of value v. This is permitted to be an under-approximation, allowing some locations' view of $\mathscr{L}(v)$ to be incomplete.

9.2. Enhanced Reduces Relation

We enhance the reduces relation $\rightarrow$ to add the location mapping $\mathscr{L}$. Reductions are now written as $\mathscr{L}, \phi \vdash e \rightarrow \mathscr{L}', \phi' \vdash e'$.

9.3. New Rewrite Rules

Distributed Orc's rewrite rules reuse the Orc rewrite rules in a distributed setting. For every rule in figure 6.3 and figure 6.5 (but not figure 6.4), every token in the rule must be at the same location k. Formally, a local rule of the form:

$$\frac{\Gamma}{\phi \vdash \Delta \to \phi' \vdash \Delta'}$$

is an abbreviation for:

$$\frac{k \in \bigcap_{u \in U} \mathscr{L}(u) \qquad \Gamma}{\mathscr{L}, \phi \vdash \Delta[\bullet^{v}_{\rho,\theta} \mapsto \bullet^{v}_{\rho,\theta,k}] \to \mathscr{L}, \phi' \vdash \Delta'[\bullet^{v}_{\rho,\theta} \mapsto \bullet^{v}_{\rho,\theta,k}]}$$

where U is the set of all Distributed Orc values used in the rule. Congruence rules must be extended to carry $\mathscr{L}$ in the configurations: Congruence rule premises and conclusions of the form $\phi \vdash e \to \phi' \vdash e'$ become $\mathscr{L}, \phi \vdash e \to \mathscr{L}', \phi' \vdash e'$

New rules for Distributed Orc are shown in figure 9.1. Replacement rules for figure 6.4 are in figure 9.2.

If the location handling is erased from the non-local semantics, it is equivalent to a purely local semantics of Orc. Appendix A demonstrates this.

9.3.1. Token Migration

When a Distributed Orc token executing in one location needs to continue its execution in another location, the token must be migrated. In the rewrite rules, a token migration from location k to k' is represented as a relabeling of the token $\bullet^{\theta}_{\rho,v,k}$ to $\bullet^{\theta}_{\rho,v,k'}$.

9.3.2. Value–Location Relation

Values in a Distributed Orc execution may exist at one or more locations, because of copying in preparation for site calls (rule SiteCall-CopyVal), or because the environment pre-positioned copies at certain locations. Further, some values may have restrictions on copying to other locations. In the sematic rules, the value–location relation is represented as a function $\mathscr{L}$ from a value v to set of locations 2^{Loc} with copies of value v. The

restriction on where copies of a value v are permitted are represented similarly through a function $\mathscr{P} : v \to 2^{\text{Loc}}$.

9.3.3. Site Calls

To call a site, Distributed Orc requires the site and argument values to all be at the token's current location. Distributed Orc can copy values as needed to any location permitted by the values' location policies (rule SiteCall-CopyVal). Also, Distributed Orc can migrate the token to another location (rule SiteCall-Migrate). Once the token is at a location that has the requisite local values, the site call is invoked (local rule SiteCall-Issue in figure 6.3). The SiteCall-Migrate and SiteCall-CopyVal rules are nondeterministic and permit divergence. The SiteCall-Migrate rule permits migration to any location, which is clearly under-constrained. This is to give the implementation semantic flexibility to execute copying and migration steps as it sees fit.

9.3.4. Trim Combinator

The Orc trim combinator terminates all execution in an expression when the first value is published. A Distributed Orc expression may have tokens executing in multiple locations.

A publication by a token executing in the scope of a trim combinator causes other tokens to be terminated (rule Trim-Pub). This termination is represented in the semantics as an application of the eraseTokens function. This eager erasure of tokens is unnecessary. As long as there are no visible effects from the to-be-erased tokens, namely additional publications or site calls, an implementation can lazily erase tokens.

9.3.5. Graft Combinator

The Orc graft combinator binds a future to a result. A Distributed Orc expression may have tokens executing in multiple locations.

When a future is bound, the future-to-value mapping ϕ is updated (rule Graft-PubL), and expressions waiting for the value can proceed using it (local rule Future). If the value

$$\frac{v \in \{x, a_{1\ldots n}\} \qquad v \neq \Diamond \qquad k' \in \mathscr{P}(v) \setminus \mathscr{L}(v)}{\mathscr{L}, \phi \vdash \bullet_{\rho,\theta,k}\, x(a_{1\ldots n}) \rightarrow \mathscr{L}', \phi \vdash \bullet_{\rho,\theta,k}\, x(a_{1\ldots n})} \text{ (SiteCall-CopyVal)}$$

where $\mathscr{L}' = \mathscr{L} \cup \{\langle v, k' \rangle\}$

$$\frac{\rho(x) = s \qquad k' \in \mathrm{Loc}}{\mathscr{L}, \phi \vdash \bullet_{\rho,\theta,k}\, x(a_{1\ldots n}) \rightarrow \mathscr{L}, \phi \vdash \bullet_{\rho,\theta,k'}\, x(a_{1\ldots n})} \text{ (SiteCall-Migrate)}$$

Figure 9.1. Distributed Orc non-local semantics: added site call rules

expression halts without publishing, the future-to-value mapping ϕ is updated to record this (rule Graft-NoPub), and expressions waiting for the future's value will halt (local rule Future-Stop).

9.3.6. Otherwise Combinator

The Orc otherwise combinator monitors execution of an expression. When the otherwise combinator begins executing, state is saved in a prototoken in the righthand subexperssion for possible use (rule Otherwise-Enter). If the left subexpression publishes, the publication is published by the otherwise combinator, and the prototoken is erased (rule Otherwise-PubL). If all tokens in scope terminate, and there have been no publications, the alternate expression is run (rule Otherwise-NoPub).

9.3.7. Program Startup

At program startup, the execution environment supplies: (1) the location of the initial token, (2) the policy function $\mathscr{P}$, and (3) the initial location function $\mathscr{L}$. The manner of specifying these is not part of the language semantics, and is application-specific.

$$\frac{}{\mathscr{L}, \phi \vdash \bullet_{\rho,\theta,k} \{|e|\} \to \mathscr{L}, \phi \vdash \{| \bullet_{\rho,\theta',k} e|\}} \text{(Trim-Enter)}$$
where $\theta' = \text{pushTag}(\theta)$

$$\frac{}{\mathscr{L}, \phi \vdash \{|e \bullet^{v}_{\rho,\theta,k}|\} \to \mathscr{L}, \phi \vdash \{|e'|\} \bullet^{v}_{\rho,\theta',k}} \text{(Trim-Pub)}$$
where $e' = \text{eraseTokens}(\theta, e) \quad \theta' = \text{popTag}(\theta)$

$$\frac{}{\mathscr{L}, \phi \vdash \bullet_{\rho,\theta,k} (x \overset{\text{val}}{=} e_1 \ \# \ e_2) \to \mathscr{L}, \phi \vdash x \overset{\text{val}}{=} (\bullet_{\rho,\theta',k} e_1) \ \# \ (\bullet_{\rho',\theta,k} e_2)} \text{(Graft-Enter)}$$
where $\theta' = \text{pushTag}(\theta) \quad \rho' = \rho[x \mapsto \Diamond_{\theta'}]$

$$\frac{\theta \notin \phi}{\mathscr{L}, \phi \vdash x \overset{\text{val}}{=} (e_1 \bullet^{v}_{\rho,\theta,k}) \ \# \ e_2 \to \mathscr{L}, \phi' \vdash x \overset{\text{val}}{=} e_1 \ \# \ e_2} \text{(Graft-FirstPubL)}$$
where $\phi' = \phi[\theta \mapsto v]$

$$\frac{\theta \in \phi}{\mathscr{L}, \phi \vdash x \overset{\text{val}}{=} (e_1 \bullet^{v}_{\rho,\theta,k}) \ \# \ e_2 \to \mathscr{L}, \phi \vdash x \overset{\text{val}}{=} e_1 \ \# \ e_2} \text{(Graft-SubsPubL)}$$

$$\frac{\neg \, \text{isLive}(\theta, e_1)}{\mathscr{L}, \phi \vdash x \overset{\text{val}}{=} e_1 \ \# \ e_2 \to \mathscr{L}, \phi' \vdash x \overset{\text{val}}{=} e_1 \ \# \ e_2} \text{(Graft-NoPubL)}$$
where $\phi' = \phi[\theta \mapsto \texttt{stop}]$

$$\frac{}{\mathscr{L}, \phi \vdash x \overset{\text{val}}{=} e_1 \ \# \ (e_2 \bullet^{v}_{\rho,\theta,k}) \to \mathscr{L}, \phi \vdash (x \overset{\text{val}}{=} e_1 \ \# \ e_2) \bullet^{v}_{\rho',\theta,k}} \text{(Graft-PubR)}$$
where $\rho' = \rho \setminus \{x\}$

$$\frac{}{\mathscr{L}, \phi \vdash \bullet_{\rho,\theta,k} (l \ ; \ r) \to \mathscr{L}, \phi \vdash (\bullet_{\rho,\theta',k} l) \ ; \ (\odot_{\rho,\theta',k} r)} \text{(Otherwise-Enter)}$$
where $\theta' = \text{pushTag}(\theta)$

$$\frac{\neg \, \text{isLive}(\theta, l)}{\mathscr{L}, \phi \vdash l \ ; \ \odot_{\rho,\theta,k} r \to \mathscr{L}, \phi \vdash l \ ; \ \bullet_{\rho,\theta,k} r} \text{(Otherwise-NoPub)}$$

$$\frac{}{\mathscr{L}, \phi \vdash (l \bullet^{v}_{\rho,\theta,k}) \ ; \ r \to \mathscr{L}, \phi \vdash (l \ ; \ r') \bullet^{v}_{\rho,\theta',k}} \text{(Otherwise-PubL)}$$
where $r' = \text{eraseTokens}(\theta, r) \quad \theta' = \text{popTag}(\theta)$

$$\frac{}{\mathscr{L}, \phi \vdash l \ ; (r \bullet^{v}_{\rho,\theta,k}) \to \mathscr{L}, \phi \vdash (l \ ; \ r) \bullet^{v}_{\rho,\theta',k}} \text{(Otherwise-PubR)}$$
where $\theta' = \text{popTag}(\theta)$

Figure 9.2. Distributed Orc non-local semantics: trim, graft, and otherwise combinators

10.
Distributed Orc Design Rationale

There are a number of decisions made in the Distributed Orc design that are crucial to its usability and tractability. The goals that drove these decisions include:

1. Provide location transparency, while preserving the local Orc semantics.
2. Exclude the need for any global state driven by the language semantics.
3. Exclude the need for any global atomicity or synchronization.
4. Leave the language system as much freedom as possible for many different program execution tactics (i.e. optimization or adaptation to other platforms).

Some of the major design decisions are discussed here.

10.1. Migration Not Limited to Component Boundaries

Most transparently distributed languages only permit migration at the boundaries of software engineering constructs, such as objects, modules, functions, etc. This overloads the purpose of these constructs—what was intended as a language feature to manage functional complexity becomes entangled with distribution choices. This is "adverse coupling" in the language design and has several undesirable implications. The adverse effects of this were enumerated in section 1.1, so we will not reiterate them here.

Distributed Orc's view is that software engineering constructs should remain as such, and migration should be allowed throughout the program.

10.2. Selection of Orc as Base Language

The Orc language fulfills requisites of chapter 3, namely pervasive concurrency and pervasive failure awareness.

10.3. Migration Triggered by Site Calls

Since the "work" in Orc is performed by external site calls, Distributed Orc is only obligated to reify values at a location when a site call is executed. Other language constructs, since they don't examine or manipulate values, only need to track the location of values, but not have access them as local values. Therefore, token migration is required only during the preparation for site call execution.

10.4. Migration Underspecified

The language semantics permit migration that is not strictly necessary, and permit the arbitrary selection of locations for migration. This leaves open a wide range of migration behaviors, and permits the language system to optimize migration decisions.

10.5. Coherence Not a Language Feature

Base Orc treats values that it manages as immutable, but permits mutability in sites. This keeps the language simple, and permits each site to handle the consistency concerns in the manner most appropriate for the site's semantics. For example, a counter site could accommodate unrestricted concurrent access, while a file I/O site may need some form of concurrency control to prevent concurrent writers from interfering with each other.

Since Distributed Orc's values are immutable, coherence concerns are similarly delegated to sites. Sites can elect to interact with the Distributed Orc distribution mechanism to provide mutability and appropriate coherence semantics.

10.6. Distribution Obliviousness

Because the need for token migration is restricted to site calls, and because Distributed Orc treats values as opaque in most situations, the vast majority of a Distributed Orc implementation can be oblivious to distribution. This includes the sequential and parallel

combinators. The trim, graft, and otherwise combinators require communication, but not token migration.

10.7. Synchronization Requirements

The locality discussion in section 6.5 foreshadowed considerations of synchronization requirements for a cluster executing Distributed Orc programs.

The semantics in chapter 9 were very deliberately designed to not require any global stores or global synchronization. Several operations require coordination and some require eventual consistency, as discussed below.

The four categories of observations in section 6.5, along with their distributed runtime implications for Distributed Orc are:

Token-local observations. No distributed relevance.

First publication detection. There must be consensus on a selected "first" publication from a token group. It is not essential that this be the real-time first publication.

Future look-up. Looking up a given future in the future-to-value mapping ϕ must *eventually* see a value, if one is placed in the mapping, but timeliness is a performance concern, not a correctness requirement.

Group liveness observation. Observing that a token group has halted (i.e., is no longer has executing tokens) only need occur eventually. Delayed reporting of a true-to-false transition for the semantic isLive function does not cause any correctness problems.

The four categories of effects in section 6.5, along with their distributed runtime implications for Distributed Orc are:

Single token effects. No distributed relevance.

Double token effects. Actions are performed at the first token, then the second token (possibly at a remote location) is notified to perform its actions.

Future binding. When a future is bound, any blocked readers of the future must be notified of the binding. The value must be kept for subsequent readers of the future. As an optimization, this entry may be dropped from the mapping when the future is no longer in scope for any token. Orc does not permit escape of future references from the body of the graft combinator, so this optimization does not necessitate distributed garbage collection, but merely an isLive test of the graft combinator's body.

Group kill. Group kills may be performed non-atomically, so synchronization is not required.

The result of this semantic design is that the parallel and sequential combinators require no engineering effort for distribution. The trim, graft, and otherwise combinators require an extension to their existing token group responsibilities. Futures require only an eventually-consistent map, which optionally can be optimized to purge no-longer-needed entries.

11.
Distributed Orc Implementation

11.1. Reuse of Existing Orc Implementation

Our Distributed Orc implementation extends the existing UT Austin Orc implementation to support distribution and migration according to the formal semantics. As part of this work, we implemented Distributed Orc three times, initially as a preliminary experimental extension to the token-driving interpreter, then as an extension to PorcE, and again as an extension to the token-driving interpreter.

Extensions needed for Distributed Orc are localized in a small number of new packages, with only two existing Orc interpreter source files modified to "hook" into the existing system, adding only approximately 2000 lines of Scala (12%) to the existing Orc compiler and runtime system.

The two modifications to the existing Orc local-only runtime engines are: (1) possible migration is considered at site calls, and (2) declaration of the Distributed Orc as an alternative runtime vs. the local-only runtime. The remainder of the Distributed Orc implementation consists of separate, non-invasive additions to the Orc implementation.

11.2. Communication Channels

Distributed Orc runtime instances communicate messages to perform the following actions during execution (excluding startup and shutdown messages):

1. Migrate an executing token (thread) to another runtime instance.
2. Notify of a published value.
3. Terminate execution of an expression (group kill).
4. Notify of an expression execution's halting.

5. Read a future's value.

These messages are sent via Java serialization streams over TCP connections. Distributed Orc modifies the Java serialization behavior (1) to comply with the value location policy, (2) to use the Distributed Orc remote reference facility (described later), (3) to correctly handle closures, (4) to prevent serialization of runtime engine internal structures, and (5) to fix-up certain implementation-internal context during marshaling/unmarshaling. A large proportion of the Distributed Orc codebase and implementation effort is devoted to network connection management and marshaling/unmarshaling management, including working around bugs and limitations in the Java serialization facility.

11.3. Cluster Startup/Shutdown and Program Startup/Shutdown

At cluster startup time, a complete graph of connections among the cluster's Distributed Orc runtime instances is established. These connections are Java serialization streams over TCP. When program execution starts, a copy of the program AST is sent to all runtime instances. Then, execution proceeds normally, with placement of the initial token at the root node of the program AST.

When the top-level token group halts, indicating the program has terminated, runtime instances are notified to unload the AST. When the cluster is shutdown, the Java serialization streams and TCP connections are closed in an orderly manner.

11.4. Token Migration

When migration is needed to execute a site call, the current execution state of a token (including stack, environment, and certain interpreter state values) is marshaled and sent to the destination Distributed Orc interpreter. A proxy for the token is left behind on the source Distributed Orc interpreter, so that the non-Distributed Orc parts of the Orc runtime can operate unaware of distribution.

Implementation of the transfer of a token to a new location primarily consists of copying four major token state components: the point of execution in the program, the token group that the token is currently a member of, the stack, and the variable binding environment. The point of execution is marshaled as an index into the AST. Group membership is managed using the remote group proxy facility described below. Stack frames are marshaled using serialization replacement objects, and reconstructed during unmarshaling.

Values in the variable binding environment are marshaled by copying or by replacement with a remote reference. Orc's values are immutable, as in functional languages. So, in many cases, bound values of variables can be simply copied to the new location. However, in some cases, variables may be bound to values that cannot be copied to the new location, for example, because the value is mutable. In this case, a remote reference is bound to the variable in the environment at the new location. Closures in the environment are marshaled so that their bodies are correctly referenced at the destination. As an optimization, futures in the environment with a known value are marshaled as if they are simply bound values, eliminating the need to read the future from the destination.

Lastly, if the token is currently carrying a published value, that value is marshaled, too.

11.5. Remote References

When a value cannot be copied to a migration destination, because it is mutable, because policy prevents it, or for some other reason, it is replaced at the destination with a remote object reference. A remote object reference is not necessarily a proxy; it may simply be an identifier of an object on another node. Remote object references are uniquely identified, and are replaced with their referent when unmarshaled on their origin runtime instance.

When the location is being selected for a site call, remote references are treated as being located at their origin runtime instance. The token will migrate to the origin before the site call is executed. This ensures Orc sites do not see remote references as argument

values.

11.6. Remote Group and Group Member Proxies

In order to interoperate with the non-distribution-aware parts of the language system, Distributed Orc creates group proxies and group member proxies. When a token migrates away from its parent group, a group member proxy is left in its place at the token's origin location. As the migrated token continues to execute, a single group member proxy may come to represent multiple tokens, and/or subgroups. When all remote members halt, this proxy will remove itself from the parent. If this proxy is killed, it will pass the kill on to its remote members.

When a token migrates to a new destination, a group proxy is created at the destination to stand in for the token's parent group. The proxies maintain a remote reference to their principal objects, and send the appropriate messages to them as the non-distribution-aware code interacts with the proxies. When all local group proxy members halt, the remote (principal) group will be notified. If the remote group is killed, this proxy will pass the kill on to the local members.

11.7. Future Reads and Binding

In the Distributed Orc implementation, futures are created normally, as local futures. However, when referenced by tokens that are migrated to other runtime instances, the references become remote future references. If a remote future reference is read, the reader blocks, and a read request is sent to the Distributed Orc runtime instance hosting the future. If the future is not yet bound, this runtime instance adds the read request to the set of blocked readers on the future. When the future is bound, all blocked readers are notified of the value, which they locally cache for future reads. If a read request is received for an already bound future, the value is sent immediately. If a graft combinator's value expression halts silently (without publishing), indicating the future will never be bound to a value, readers of the future are notified of this.

11.8. Value Location Tracking and Policies

Distributed Orc runtime instances track locations of values, in order to determine the appropriate locations to execute site calls. The implementation uses a set of value locator services to find locations for a given value and the locations permitted by policy. In other words, value locators implement the semantics' policy function $\mathscr{P}$ and location function $\mathscr{L}$. Multiple value locator services may exist per execution, each handling some subset of Orc values.

The Distributed Orc implementation and standard library provide default value locator services and placement policies.

11.9. Distribution-Aware Sites

As an extension to the formal semantics, value locators may override the location decision for certain site calls. This is useful for ensuring constructor calls are migrated to their preferred location. As another extension to the formal semantics, Orc values can request to be notified as they are marshaled and unmarshaled. Values can also supply a marshalable replacement for themselves when they are marshaled for serialization to another location. These extensions let distribution-aware sites participate in or even control the migration and marshaling processes. Sites may use this to, for example, implement a consistency invariant for their mutable state.

11.10. Example Message Sequence

To provide a qualitative sense of messages exchanged among Distributed Orc runtime instances, figure 11.1 presents the message sequence for execution of the following program.

```
def times10(x) = x * 10
val theList = [shard1, shard2]
map(times10, theList)
```

The program represents the message flow of a map–reduce problem in essential form. The function `times10` is mapped to a two-element list, `theList`. The `map` function is an Orc standard library function. The program executes in a 3-runtime instance cluster, with a leader and two follower runtime instances. (Leader and follower designations are simply for coordination of startup/shutdown and program top-level user interface, and have no semantic or other operational significance.) The list elements `shard1` is constrained by policy to reside on runtime 1, and `shard2` on runtime 2. The elements are single-element list shards, which are referred to as shard1 and shard2 in the diagram.

Orc lists are conventional LISP-style cons lists, consisting of a head–tail pair, where the distinguished `Nil` (empty list) value terminates the list. The Orc deflation process for site calls transforms the list construction `[shard1, shard2]` into `Nil() >elem0> Cons(shard2, elem0) >elem1> Cons(shard1, elem1)`.

The depicted sequence of messages begins after leader and followers establish network connections and mutually identify each other.

1. Initially, the leader transmits a copy of the program AST to all followers.

2. Program execution begins on runtime 0. The first site call is construction of the list cons pair `shard2` and `Nil`, the empty list. `shard2` can only reside on runtime 2, so token migration begins.

3. The token arrives at runtime 2, and the site call is executed locally. Runtime 2 is nominated as the coordinator for the `theList` future.

4. The next site call is construction of the list cons pair `shard1` with the previously-constructed cons pair. `shard1` can only reside on runtime 1, so token migration begins.

5. The token arrives at runtime 1. Note that the reference to the previously-constructed cons pair's head is now a remote reference. The site call is executed locally.

6. This site call's result is published, and runtime 1 propagates the publication to the token's enclosing context ("group"), which resides on runtime 2.

7. Runtime 1 notifies the enclosing group on runtime 2 that the token halted (completed execution).

8. Runtime 2 propagates the publication to the next enclosing group, which resides on runtime 0.

9. The enclosing group on runtime 0 is notified that the token halted (completed execution).

10. Clausal def matching occurs, which is an inessential language feature not discussed here. After clausal def matching, execution continues on runtime 1.

11. Runtime 1 requests the value of the `theList` future from the coordinator, runtime 2.

12. Concurrently, the `map` function proceeds, publishing the `Nil` at the end of the mapped list to runtime 2.

13. Runtime 2 sends the value of the `theList` future to runtime 1.

14. Runtime 1 notifies the enclosing group on runtime 2 that the token halted (completed execution).

15. Runtime 2 applies the mapped function `times10` to its local part of the list, and publishes the result to runtime 1.

16. Runtime 2 notifies the enclosing group on runtime 1 that the token halted (completed execution).

17. Runtime 1 applies the mapped function `times10` to its local part of the list, and publishes the result, along with runtime 2's results, to runtime 0.

18. Runtime 1 notifies the enclosing group on runtime 0 that the token halted (completed execution).

19. Runtime 0 publishes the resulting mapped list as a program result.

20. The program halts.

21. The leader notifies all followers that the program AST may be unloaded.

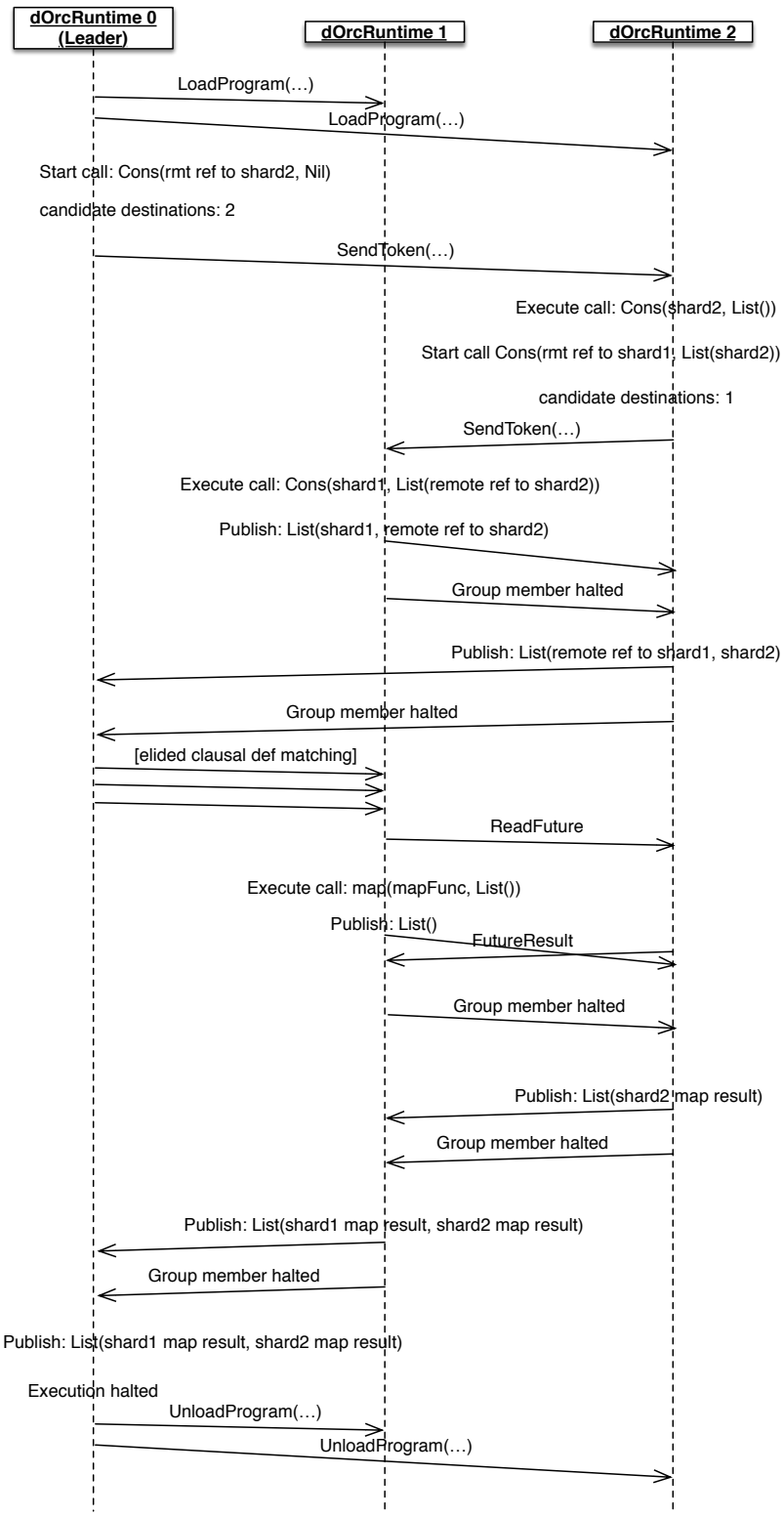


Figure 11.1. Example message sequence.

12.
Evaluation and Results

While we consider the operational semantics for Distributed Orc presented in chapter 9 to be the primary contribution of this work, we evaluate Distributed Orc to characterize programmability benefits and to provide preliminary evidence of the potential performance gains enabled by transparent migration of execution.

12.1. Programmability

We compared six pairs of small programs to evaluate the programmability advantages of Distributed Orc.

Method. Each pair included an Orc program written by members of the Orc research group and a Java program written based on a publicly available implementation and modified by the members of the Orc research group where necessary to ensure functional comparability. Each pair of programs uses the same algorithms and, where applicable, the same kind of synchronization and approach to partitioning application logic across distributed execution resources; all use communication frameworks that enable distribution across multiple machines.

Results. Results are shown in table 12.1. We report the number of lines of code for both Java and Distributed Orc implementations, along with the percentage of those lines of code devoted to functionality other than core application logic, for example, launching worker processes or configuration of RMI registries. The remaining columns include a check mark where distribution and concurrency force the use of programming idioms in the Java implementations that are absent for Distributed Orc. The *sched/data-xfer* column indicates whether or not explicit scheduling and data transfer code is present. The *RMI* column is checked where Java RMI is used to invoke remote work. The *synchronized*

Table 12.1. Comparison of Distributed Orc versus Java implementations for a group of benchmarks. LoC were measured with SLOCCount [63], and are reported along with the percentage of those lines devoted to things other than core application logic. Source code for the programs used for comparisons is in Appendix B.

	Orc LoC	Java LoC	sched/data-xfer RMI	synchronized	cond-sync	futures	sockets/msg
Map–Reduce	31 (3%)	403 (74%)	✓	✓		✓	✓	
Randomized Byzantine Agreement	43 (0%)	718 (90%)	✓		✓	✓		✓
Dining Philosophers	65 (0%)	332 (36%)	✓	✓	✓	✓		
Breadth-First Search	28 (0%)	260 (60%)	✓	✓	✓	✓		✓
Depth-First Search	23 (0%)	138 (40%)	✓	✓	✓			
Sudoku	60 (13%)	152 (35%)	✓	✓				

column is checked where the implementation requires programmer-managed mutual exclusion, either through explicit locks or use of the `synchronized` keyword. The *cond-sync* column indicates cases where `wait/notify*` is used. The *futures* column indicates implementations that use futures. The *sockets/msg* column is checked for programs that communicate explicitly using sockets or message passing.

The results show that Distributed Orc has huge potential to simplify distributed programming with respect to Java. The Orc versions are consistently shorter, with a geometric mean of 82% fewer lines of code. The Orc versions also devote a much larger fraction of their code to the application logic than the Java versions, with 97% of the code devoted to the application logic for Orc vs. 44% for Java. Any code that does not implement some element of the application logic is additional conceptual load for the programmer, and is code that must be maintained. Moreover, the vast majority of the additional code is devoted to functionality such as synchronization that is known to be challenging and bug-prone [44, 51].

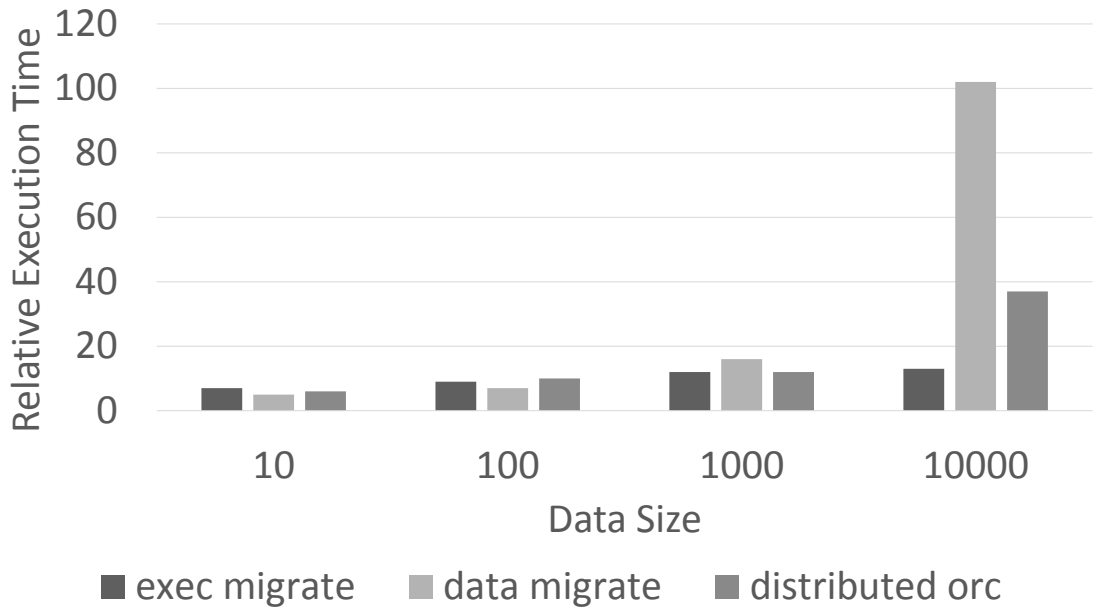


Figure 12.1. Performance of policies which migrate data or execution only, as well as a dynamic heuristic policy, relative to an all-local case for varying user database sizes. "Data size" is banned user record count.

12.2. Execution Migration

To provide preliminary evidence of the potential for performance gains, we ran the example from section 8.2.1, using a number of different migration and copying policies. The results show the importance of such policy for performance and show how Distributed Orc's decoupling of policy from mechanism enables dynamic change without requiring changes to the program code.

We evaluated the following location policies, using the performance of a non-distributed (single-process) scenario as a baseline.

1. User interaction (prompts and output) is executed in one location, and database operations are executed in a second location. This simulates a Web browser and Web server interacting, where the Web server holds the database. Tokens are migrated across two locations as needed.

2. All interaction is executed in a single location, with data copied/migrated to that

location on demand.

3. A heuristic combination of data transfer and execution migration policies: when data transfer costs are likely to be small, execution remains in a single location; conversely, execution is migrated when transfer costs are likely to be high, approximated with a threshold of 50 rows in the user database.

Method. Experiments were run on a MacBook Pro with an Intel Core i7-3720QM CPU, with 16 GB 1600 MHz DDR3 RAM. The OS was macOS Sierra version 10.12.1 (16B2555) with the Java SE Runtime Environment (build 1.8.0_60-b27) running the Java HotSpot 64-Bit Server VM (build 25.60-b23, mixed mode).

The Distributed Orc runtime instances executed on the same physical machine, but communicated using TCP/IP sockets. Database tables were simulated with Java `ConcurrentSkipListSets`, pre-loaded with a varying numbers of user e-mail addresses and banned e-mail addresses.

Results. We reported execution times from when the new user's e-mail address is supplied to when the "user added successfully" indication is returned to the UI, averaged over three runs, relative to the all local baseline. The results are shown in figure 12.1.

These results demonstrate location policies that depend on particulars of the data, in this case the database table size. For small tables, it is faster to copy the data to the current location of execution and eliminate token migrations. For larger tables, the cost of copying the table outweighs the cost of token migrations. In Distributed Orc, this consideration is kept out of the program body, and is handled by the location policy. The policies evaluated here were simple to implement and did not affect the program text.

12.3. Word Count

As preliminary evidence that location transparency can be used to implement real systems, we investigate a map–reduce workload, with the goal of having the Distributed Orc

Table 12.2. Source code size of the map–reduce word count implementations. SLOC by language is the number of physical non-blank, non-comment source lines of code of the application logic, excluding test harness overhead code.

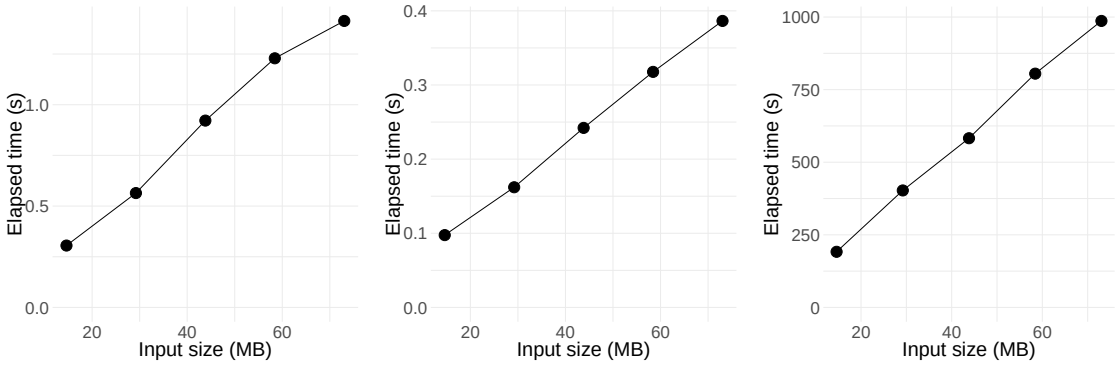
Implementation variant	SLOC by language	Local-to-distributed adaptation effort
Single-threaded conventional Java	99 Java	Complete rewrite
Mixed Orc–Java	21 Orc, 42 Java	No source changes
Concurrent pure Orc	36 Orc	No source changes

runtime transparently perform the work that requires an explicitly coded job manager and scheduler in a conventional implementation. We consider a map–reduce distributed word count application, presenting source code size and performance results for three variants:

1. A conventional, pure Java, single-threaded implementation;
2. A concurrent pure Orc variant, where each file is counted in parallel; and
3. A mixed Orc–Java variant, where each file is counted in parallel, but the Orc program orchestrates concurrent calls to a Java word count routine.

See table 12.2 for source code sizes. The pure Orc variant and mixed Orc–Java variant were written as single-machine Orc programs. Adapting them to a distributed setting requires *no* changes to the single-machine Orc programs' source code. Instead of changes to the program source, an execution placement policy is given, which in this case specifies a file-to-machine mapping. With this placement policy, the word count file operations migrate to the location appropriate for the file they are processing. When the counts are summed, these reduce operations communicate in a tree pattern.

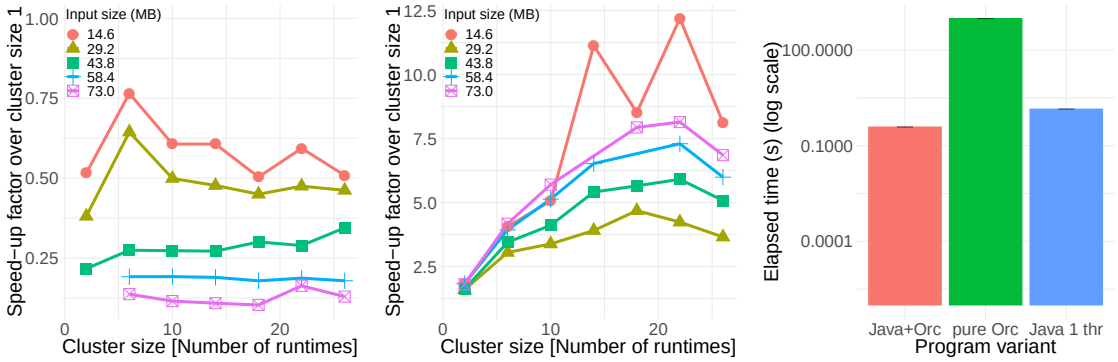
Method. We executed these variants on a cluster of 26 Dell Precision Tower 3620 machines, each with an Intel Xeon E3-1270 CPU (4 cores each) at 3.80 GHz, with 16 GiB of physical RAM. The OS was Ubuntu 18.04.3 LTS with Java SE Runtime Environment 1.8.0_8u222-b10. The JVM maximum heap size was set to 12 GiB.



(a) Elapsed time, Single-threaded conventional Java variant

(b) Elapsed time, Distributed Orc cluster size of 1, Mixed Orc–Java variant

(c) Elapsed time, Distributed Orc cluster size of 1, Concurrent pure Orc variant

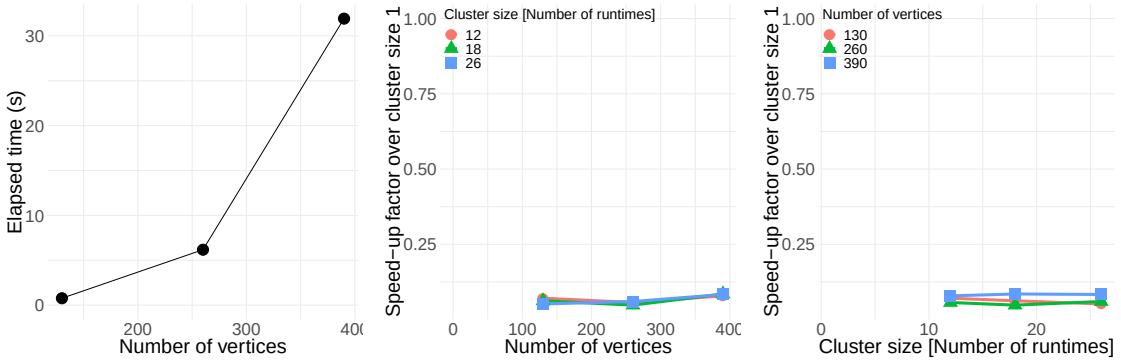


(d) Relative speed-up of Distributed Orc over a single runtime instance, for various Distributed Orc cluster sizes. Mixed Orc–Java variant.

(e) Relative speed-up of Distributed Orc over a single runtime instance, Concurrent pure Orc variant.

(f) Elapsed time on 73 MB of text. Single runtime instance (cluster of one).

Figure 12.2. WordCount: Performance of counting words in the indicated size input



(a) Elapsed time, Distributed Orc cluster size of 1

(b) Relative speed-up over a single runtime instance, for various number of vertices

(c) Relative speed-up over a single runtime instance, for various Distributed Orc cluster sizes

Figure 12.3. SSSP: Performance of computing the shortest path in the indicated size input

Each experimental condition (combination of program, input size, and cluster size) is repeated 20 times. The first 9 repetitions are discarded as warm-up, and the remaining 11 are used to compute the mean.

Results. Figure 12.2f shows the elapsed times for three of the variants to process 73 MB of input (split into 130 files of 562 kB each) in a single-machine (cluster of one) configuration. Figure 12.2d shows the relative speed-up of the mixed Orc–Java variant for various cluster sizes. The current Orc runtime engine has an undesirably high overhead for managing the interpreter's internal data structure (stacks, environments, etc.). In the pure Orc variant, this overhead dominates its performance. See Peters et al. [48] for an effort to improve Orc's local performance.

12.4. SSSP

As a second location transparency workload, we also investigated a graph-based "think like a vertex" workload. As in WordCount, here the Distributed Orc runtime transparently performs the work that requires an explicitly coded job manager and scheduler in a

conventional implementation. We implemented a single source shortest path (SSSP) application, using the Pregel SSSP algorithm [37].

The SSSP application program's text is simply the Pregel SSSP algorithm, with no explicit distribution specification. We supply the Distributed Orc runtime with an execution placement policy that maps graph vertices, based on id, to machines on the Distributed Orc cluster.

Method. The tests were run in the same environment as Word Count (section 12.3). Each experimental condition (combination of input size and cluster size) is repeated 20 times. The first 9 repetitions are discarded as warm-up, and the remaining 11 are used to compute the mean.

Results. Figure 12.3c shows the relative speed-up of the over a single runtime instance for various cluster sizes. The aforementioned undesirably high overhead of the current Orc runtime engine produces a slow baseline (single-machine) performance for the SSSP application. However, the results show a lack of scaling for the distributed runs. These runs are communication-bound, showing no improvement for a given problem size when more machines are used in a run. This is a result of the programmer-supplied execution placement policy adversely interacting with the program. In this case, the execution placement policy did not constrain the placement of the edge instances, and their far suboptimal placement causes excessive communication. The programmer could include edges in the execution placement policy, but ideally Distributed Orc should be able to infer this placement. This inference facility is the next step in the evolution of Distributed Orc (see section 13.4).

12.5. SSSP Message Count

The communication-bound performance of SSSP can be understood by observing the message count growth vs. graph size.

Method. SSSP tests were run in the same environment as previous experiments, but with smaller graph sizes (20–80 vertices). A communications trace log was written by each Distributed Orc runtime, collected, and the messages were counted by type.

Results. The message counts vs. graph size are presented in figure 12.4. Only the message types with counts over approximately 1500 are shown (3 of 9 types). The message count growth is approximately cubic (degree 2.7–3.4) vs. graph size.

There are three types of messages that dominate the message traffic. They are the messages exchanged for (1) token migration, (2) future binding, and (3) group member halt notification. These messages correspond to the following actions in the token semantics: (1) A migrate token message corresponds to changing the location label of a token, which only occurs in rule SiteCall-Migrate. (2) A future result message corresponds to adding an element to the future-to-value map ϕ, or reading that new element. (3) A group member halt notification message corresponds to the token group tracking needed for the isLive semantic function.

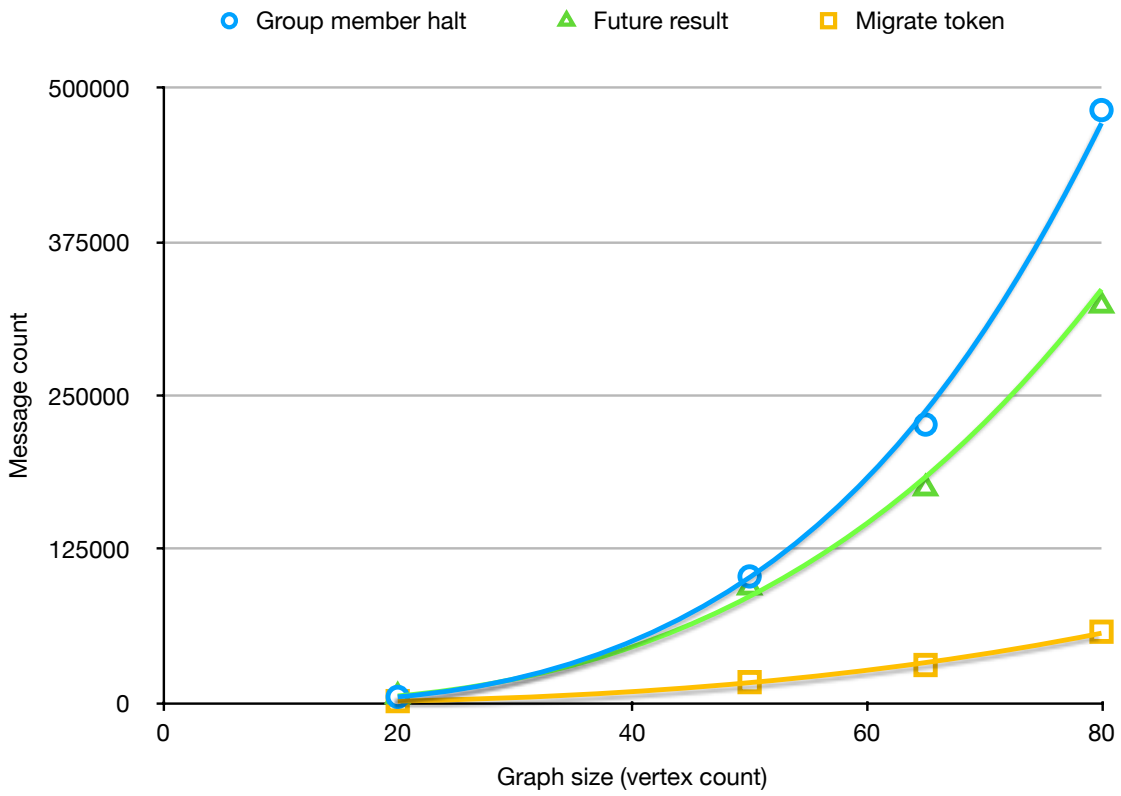


Figure 12.4. SSSP message count. Count of messages, by message type, of messages exchanged among Distributed Orc runtime instances for SSSP executions on small graphs. Only the message types with counts over approximately 1500 are shown (3 of 9 types).

13.
Discussion

13.1. Key Findings

This work set out to demonstrate that distributed software can be written in an implicitly distributed manner. Specifically, we seek to validate the four theses in section 1.2, reiterated below. We have found support for three of the four thesis, but the last remains unproven.

Thesis 1. *Distributed programs can be written in a location-transparent style, where local and remote semantics are uniform. This uniformity does not cause the local semantics to be awkward for programmers.*

This thesis is supported. The concern that this thesis responds to is the following: The concurrency and failure concerns introduced by remote operations, when they are made pervasive throughout the program, might become overwhelming to reason about, and cause the program to become difficult to write and comprehend.

However, years of experience with the Orc language demonstrate that *pervasive concurrency* does not result in turbid code. Orc has been used by several research groups, in university courses, and even some commercial use. The experiences of the Orc users in this case argue that Orc is a usable language. Distributed Orc is able to provide semantics that are indistinguishable from Orc, so the usability of the Orc semantics conveys to Distributed Orc. The results in section 12.1 provide evidence for Distributed Orc's usability for common distributed programs.

Thesis 2. *Distributed programs should be able to communicate as part of any expression, not just when crossing object/module boundaries. Language modularity constructs (such as objects) should be used for software engineering concerns, and not be co-opted by the distribution mechanism.*

This thesis is supported. In some sense, this is a position that this work takes as a premise, in order to eliminate the adverse effects of distribution-induced architecture fragmentation enumerated in section 1.1. Support for this position is provided by experience in using Distributed Orc, such as that presented in section 12.1.

Thesis 3. *The language system can automate the distribution of these location-transparent distributed programs.*

This thesis is supported. The current Distributed Orc implementation automate the distribution of programs, for the full, unrestricted Distributed Orc language. This has been demonstrated for various programs, in a variety of environments: from multiple processes on a single laptop to clusters of dozens of hosts, as shown in chapter 12.

Optimal placement of work and data in distributed systems is a complex question. Distributed Orc does not claim to eliminate this complexity, but it does, at a minimum, externalize placement, rather than leave it intertwined with the program. Placement decisions are managed in the language system, location policy specifications, and distribution-aware sites.

Thesis 4. *The performance of programs under this automated distribution is adequate.*

Not supported at present. There are some workloads where performance might be considered adequate, but for most that we evaluated, the communications costs quickly become overwhelming with scale. The language system's current communications needs and placement decisions produce executions that become network-bound, and execute orders of magnitude too slowly. See section 12.5 for a characterization of these costs. See section 13.4 for potential future work that may mitigate this problem.

13.2. Assumptions and Limitations

1. Both Orc and Distributed Orc are not currently targeted at high-performance computing (HPC) applications, which perform "massively" data-parallel operations.

These applications often require detailed manual performance tuning of both their compute and communications operations.

2. We take Orc, and its results, as given, and do not develop a motivation for Orc, nor evaluate Orc itself. There is a significant body of work addressing Orc already. See https://orc.csres.utexas.edu/research.shtml.

3. Security considerations here are similar to existing distributed systems' security considerations. Therefore, they are outside the scope of this work.

4. The Distributed Orc evaluations are limited at present, because scaling problems appeared in early results. There was no need to demonstrate Distributed Orc's lack of performance in a multitude of benchmarks.

13.3. Generalizability

Distributed Orc was fully implemented as an extension to the PorcE Orc implementation, and then re-implemented as an extension to the token-driving Orc implementation. The performance measured for both implementations was approximately equivalent.

The critical placement decisions exposed by the poor performance in SSSP, for example, are primarily related to two Orc language features: future values and halting notifications. This raises the question: are these problems an artifact of adopting Orc, or are they inherent to transparent distribution? Many programs utilize some type of means of concurrently executing an expression while computing an input to that expression, which is what futures provide. If futures were not available, these programs would essentially reconstruct them using the available language features and libraries. Perhaps futures would not be as widely used, but they are essential, in our view. On the other hand, halting notifications are essential to the semantics of 3 of the 5 Orc combinators, but a different language may be able to obviate the need for them.

13.4. Potential Future Work

Placement optimization. The formal semantics of Distributed Orc do not include any explicit optimizations. However, the semantics admit (through their non-determinism) a wide range of static and dynamic optimizations of communication and execution. For example, an optimizing Distributed Orc implementation could perform a backwards program slicing analysis to take into account destination of results of operations, and choose execution devices that reduce the need for communications later in the execution. As another example, an optimizing Distributed Orc runtime could measure the communications performance to/from the other Distributed Orc runtime instances in the cluster, and use estimated communications costs as an input to placement decisions.

Distributed consensus and multicast. When a trim, graft, or otherwise combinator is executed, the current Distributed Orc implementation sets the location as a coordination point for that combinator instance. For example, when a graft combinator's value subexpression publishes, the coordination point selects the "first" publication. This is a single point of failure and possibly a bottleneck. However, the semantics do not require such a coordination point. These coordination points may be eliminated by multicast, distributed consensus, etc.

Context pruning. When an expression executes on a different device than its surrounding or "parent" expression, how much of the context needs to be transferred? For example, the expression `isUnique(`*`username`*`)` needs the bound value for the name *`username`*. Transmitting the whole environment with each call isn't optimal, so some pruning must happen. The current Distributed Orc implementation lightly prunes environments, but could do much better.

Pervasive failure awareness. Orc's pervasive failure awareness capabilities could be strengthened. One research proposal addressing this is Chromatic Orc (http://orc.csres.utexas.edu/papers/OrcExceptionSemantics.pdf). Chromatic Orc proposes

assigning “colors” to publications in Orc, which combinators can selectively interact with or ignore. An exception-handling facility would be syntactic sugar on top of Chromatic Orc. This would simplify failure handling.

Eliminate halting notifications. Is there an alternative to using the Orc language as a basis that provides pervasive concurrency, but without the need for halting notifications? The halting notifications are one of the two dominant drivers of the excessive communications observed in Distributed Orc evaluations, so eliminating the need for them may be fruitful.

A.

Equivalence of Distributed Orc and Local Orc Token Semantics

Here, we demonstrate the equivalence of the Distributed Orc token semantics rules to the local Orc rules. This is a simple exercise in erasing the location data from the Distributed Orc rules, and then verifying that the result is the local Orc rules.

The erasure is:

- Erase the current location from tokens and prototokens: $\bullet^{v}_{\rho,\theta,k}$ becomes $\bullet^{v}_{\rho,\theta}$
- Erase $\mathscr{L}$ from configurations: $\mathscr{L}, \phi \vdash e \to \mathscr{L}', \phi' \vdash e'$ becomes $\phi \vdash e \to \phi' \vdash e'$
- Erase $\mathscr{L}$ and $\mathscr{P}$ conditions

Distributed Orc

$$\frac{v \in \{x, a_{1\ldots n}\} \qquad v \neq \Diamond \qquad k' \in \mathscr{P}(v) \setminus \mathscr{L}(v)}{\mathscr{L}, \phi \vdash \bullet_{\rho,\theta,k}\, x(a_{1\ldots n}) \to \mathscr{L}', \phi \vdash \bullet_{\rho,\theta,k}\, x(a_{1\ldots n})} \text{ (SiteCall-CopyVal)}$$

$$\text{where } \mathscr{L}' = \mathscr{L} \cup \{\langle v, k' \rangle\}$$

after erasure, becomes tautologous and can be disregarded.

Distributed Orc

$$\frac{\rho(x) = s \qquad k' \in \mathrm{Loc}}{\mathscr{L}, \phi \vdash \bullet_{\rho,\theta,k}\, x(a_{1\ldots n}) \to \mathscr{L}, \phi \vdash \bullet_{\rho,\theta,k'}\, x(a_{1\ldots n})} \text{ (SiteCall-Migrate)}$$

after erasure, becomes tautologous and can be disregarded.

Distributed Orc

$$\frac{}{\mathscr{L}, \phi \vdash \bullet_{\rho,\theta,k}\, \{|e|\} \to \mathscr{L}, \phi \vdash \{|\, \bullet_{\rho,\theta',k}\, e|\}} \text{ (Trim-Enter)}$$

$$\text{where } \theta' = \mathrm{pushTag}(\theta)$$

after erasure, becomes Orc

$$\frac{}{\phi \vdash \bullet_{\rho,\theta}\, \{|e|\} \to \phi \vdash \{|\, \bullet_{\rho,\theta'}\, e|\}} \text{ (Trim-Enter)}$$

$$\text{where } \theta' = \mathrm{pushTag}(\theta)$$

Distributed Orc

$$\frac{}{\mathscr{L}, \phi \vdash \{\!|e \bullet^{v}_{\rho,\theta,k}|\!\} \to \mathscr{L}, \phi \vdash \{\!|e'|\!\} \bullet^{v}_{\rho,\theta',k}} \text{(Trim-Pub)}$$

$$\text{where } e' = \text{eraseTokens}(\theta, e) \quad \theta' = \text{popTag}(\theta)$$

after erasure, becomes Orc

$$\frac{}{\phi \vdash \{\!|e \bullet^{v}_{\rho,\theta}|\!\} \to \phi \vdash \{\!|e'|\!\} \bullet^{v}_{\rho,\theta'}} \text{(Trim-Pub)}$$

$$\text{where } e' = \text{eraseTokens}(\theta, e) \quad \theta' = \text{popTag}(\theta)$$

Distributed Orc

$$\frac{}{\mathscr{L}, \phi \vdash \bullet_{\rho,\theta,k} (x \overset{\text{val}}{=} e_1 \mathbin{\#} e_2) \to \mathscr{L}, \phi \vdash x \overset{\text{val}}{=} (\bullet_{\rho,\theta',k}\, e_1) \mathbin{\#} (\bullet_{\rho',\theta,k}\, e_2)} \text{(Graft-Enter)}$$

$$\text{where } \theta' = \text{pushTag}(\theta) \quad \rho' = \rho[x \mapsto \Diamond_{\theta'}]$$

after erasure, becomes Orc

$$\frac{}{\phi \vdash \bullet_{\rho,\theta} (x \overset{\text{val}}{=} e_1 \mathbin{\#} e_2) \to \phi \vdash x \overset{\text{val}}{=} (\bullet_{\rho,\theta'}\, e_1) \mathbin{\#} (\bullet_{\rho',\theta}\, e_2)} \text{(Graft-Enter)}$$

$$\text{where } \theta' = \text{pushTag}(\theta) \quad \rho' = \rho[x \mapsto \Diamond_{\theta'}]$$

Distributed Orc

$$\frac{\theta \notin \phi}{\mathscr{L}, \phi \vdash x \overset{\text{val}}{=} (e_1 \bullet^{v}_{\rho,\theta,k}) \mathbin{\#} e_2 \to \mathscr{L}, \phi' \vdash x \overset{\text{val}}{=} e_1 \mathbin{\#} e_2} \text{(Graft-FirstPubL)}$$

$$\text{where } \phi' = \phi[\theta \mapsto v]$$

after erasure, becomes Orc

$$\frac{\theta \notin \phi}{\phi \vdash x \overset{\text{val}}{=} (e_1 \bullet^{v}_{\rho,\theta}) \mathbin{\#} e_2 \to \phi' \vdash x \overset{\text{val}}{=} e_1 \mathbin{\#} e_2} \text{(Graft-FirstPubL)}$$

$$\text{where } \phi' = \phi[\theta \mapsto v]$$

Distributed Orc

$$\frac{\theta \in \phi}{\mathscr{L}, \phi \vdash x \overset{\text{val}}{=} (e_1 \bullet^{v}_{\rho,\theta,k}) \mathbin{\#} e_2 \to \mathscr{L}, \phi \vdash x \overset{\text{val}}{=} e_1 \mathbin{\#} e_2} \text{(Graft-SubsPubL)}$$

after erasure, becomes Orc

$$\frac{\theta \in \phi}{\phi \vdash x \overset{\text{val}}{=} (e_1 \bullet^{v}_{\rho,\theta}) \mathbin{\#} e_2 \to \phi \vdash x \overset{\text{val}}{=} e_1 \mathbin{\#} e_2} \text{ (Graft-SubsPubL)}$$

Distributed Orc

$$\frac{\neg\, \text{isLive}(\theta, e_1)}{\mathscr{L}, \phi \vdash x \overset{\text{val}}{=} e_1 \mathbin{\#} e_2 \to \mathscr{L}, \phi' \vdash x \overset{\text{val}}{=} e_1 \mathbin{\#} e_2} \text{ (Graft-NoPubL)}$$

where $\phi' = \phi[\theta \mapsto \texttt{stop}]$

after erasure, becomes Orc

$$\frac{\neg\, \text{isLive}(\theta, e_1)}{\phi \vdash x \overset{\text{val}}{=} e_1 \mathbin{\#} e_2 \to \phi' \vdash x \overset{\text{val}}{=} e_1 \mathbin{\#} e_2} \text{ (Graft-NoPubL)}$$

where $\phi' = \phi[\theta \mapsto \texttt{stop}]$

Distributed Orc

$$\frac{}{\mathscr{L}, \phi \vdash x \overset{\text{val}}{=} e_1 \mathbin{\#} (e_2 \bullet^{v}_{\rho,\theta,k}) \to \mathscr{L}, \phi \vdash (x \overset{\text{val}}{=} e_1 \mathbin{\#} e_2) \bullet^{v}_{\rho',\theta,k}} \text{ (Graft-PubR)}$$

where $\rho' = \rho \setminus \{x\}$

after erasure, becomes Orc

$$\frac{}{\phi \vdash x \overset{\text{val}}{=} e_1 \mathbin{\#} (e_2 \bullet^{v}_{\rho,\theta}) \to \phi \vdash (x \overset{\text{val}}{=} e_1 \mathbin{\#} e_2) \bullet^{v}_{\rho',\theta}} \text{ (Graft-PubR)}$$

where $\rho' = \rho \setminus \{x\}$

Distributed Orc

$$\frac{}{\mathscr{L}, \phi \vdash \bullet_{\rho,\theta,k} (l \mathbin{;} r) \to \mathscr{L}, \phi \vdash (\bullet_{\rho,\theta',k}\, l) \mathbin{;} (\odot_{\rho,\theta',k}\, r)} \text{ (Otherwise-Enter)}$$

where $\theta' = \text{pushTag}(\theta)$

after erasure, becomes Orc

$$\frac{}{\phi \vdash \bullet_{\rho,\theta} (l \mathbin{;} r) \to \phi \vdash (\bullet_{\rho,\theta'}\, l) \mathbin{;} (\odot_{\rho,\theta'}\, r)} \text{ (Otherwise-Enter)}$$

where $\theta' = \text{pushTag}(\theta)$

Distributed Orc

$$\frac{\neg\,\mathrm{isLive}(\theta, l)}{\mathscr{L}, \phi \vdash l \;;\; \odot_{\rho,\theta,k}\, r \to \mathscr{L}, \phi \vdash l \;;\; \bullet_{\rho,\theta,k}\, r} \text{ (Otherwise-NoPub)}$$

after erasure, becomes Orc

$$\frac{\neg\,\mathrm{isLive}(\theta, l)}{\phi \vdash l \;;\; \odot_{\rho,\theta}\, r \to \phi \vdash l \;;\; \bullet_{\rho,\theta}\, r} \text{ (Otherwise-NoPub)}$$

Distributed Orc

$$\frac{}{\mathscr{L}, \phi \vdash (l \bullet^{v}_{\rho,\theta,k}) \;;\; r \to \mathscr{L}, \phi \vdash (l \;;\; r') \bullet^{v}_{\rho,\theta',k}} \text{ (Otherwise-PubL)}$$

$$\text{where } r' = \mathrm{eraseTokens}(\theta, r) \quad \theta' = \mathrm{popTag}(\theta)$$

after erasure, becomes Orc

$$\frac{}{\phi \vdash (l \bullet^{v}_{\rho,\theta}) \;;\; r \to \phi \vdash (l \;;\; r') \bullet^{v}_{\rho,\theta'}} \text{ (Otherwise-PubL)}$$

$$\text{where } r' = \mathrm{eraseTokens}(\theta, r) \quad \theta' = \mathrm{popTag}(\theta)$$

Distributed Orc

$$\frac{}{\mathscr{L}, \phi \vdash l \;;\; (r \bullet^{v}_{\rho,\theta,k}) \to \mathscr{L}, \phi \vdash (l \;;\; r) \bullet^{v}_{\rho,\theta',k}} \text{ (Otherwise-PubR)}$$

$$\text{where } \theta' = \mathrm{popTag}(\theta)$$

after erasure, becomes Orc

$$\frac{}{\phi \vdash l \;;\; (r \bullet^{v}_{\rho,\theta}) \to \phi \vdash (l \;;\; r) \bullet^{v}_{\rho,\theta'}} \text{ (Otherwise-PubR)}$$

$$\text{where } \theta' = \mathrm{popTag}(\theta)$$

B.

Source of Distributed Orc vs. Java Implementations

The following pages provide source code listings of the programs used for the evaluation in section 12.1. The programs were obtained from open sources, or written by members of the Orc research team.

B.1. Map–Reduce

B.1.1. Orc Map–Reduce

Listing B.1. Orc Map–Reduce

```
{-
A demonstration of the map/reduce idiom expressed in Orc.
This program is not actually allocating the map/reduce
tasks to different machines, but it shows how cleanly the
high-level data flow can be expressed in Orc; all that's
needed to actually distribute the computation is suitable
sites to which the data can be passed at each phase.
-}

import class ConcurrentMap = "java.util.concurrent.
    ConcurrentHashMap"

-- Publish every element of a Java iterator
def eachIterable(it) = repeat(IterableToStream(it))

{-
The first half of the program is the map/reduce framework,
which is assumed fixed.
-}

-- Build a map of mapper output buffers. One for each key.
val mapOutputBuffers = ConcurrentMap()
def getMapOutputBuffer(k) =
  mapOutputBuffers.putIfAbsent(k, Channel()) >> mapOutputBuffers.
    get(k)

-- Given a key, return a function which stores
-- that key.  Here we store all keys in a single
-- buffer, but a real implementation would hash
-- keys to a buffer on the machine where they would
-- be reduced
def storeForReduce(k, v) =
  val out = getMapOutputBuffer(k)
  out.put(v)

def getForReduce() =
  eachIterable(mapOutputBuffers.entrySet()) >entry> (entry.getKey
    (), entry.getValue().getAll())

-- Implement a retry policy; if the site f
-- doesn't respond in 1 second, call it again
```

```
def retry(f, a, b) =
  val opt = Some(f(a,b)) | (Rwait(1000) >> None)
  opt >Some(value)> value |
  opt >None()> retry(f, a, b)

-- The map phase reads data, maps it,
-- partitions it, and stores it
def Map(mapper, data) =
  each(data) >(k1,v1)>
  retry(mapper, k1, v1) >kvs>
  each(kvs) >(k2, v2)>
  storeForReduce(k2, v2) >> stop

-- The reduce phase sorts data,
-- groups it, reduces it, and writes it
def Reduce(reducer) =
  -- For each mapper output buffer reduce that list.
  getForReduce() >(k, vs)>  (k, retry(reducer, k, vs))

def MapReduce(mapper, reducer, data) =
  Map(mapper, data) ; Reduce(reducer)

{-
The second half of the program is the user-provided
mapper and reducer functions. For this example,
our data will be lists of numbers, and we will
count the total number of times each number appears.
-}

def mapper(name, numbers) =
  def makeTuple(v) = (v, 1)
  map(makeTuple, numbers)

def reducer(number, counts) =
  foldl((+), 0, counts)

val data = [
  ("primes", [2, 3, 5, 7, 11]),
  ("odd", [1, 3, 5, 7, 9, 11]),
  ("trees", [1, 2, 3, 6, 11]) ]

MapReduce(mapper, reducer, data)
```

B.1.2. Java Map–Reduce

Listing B.2. Java Map–Reduce `IContext` interface

```
package edu.utexas.orc;

import java.io.IOException;

public interface IContext<TKI, TVI, TKO, TVO> {

    /**
     * Advance to the next key, value pair, returning null if at
    end.
     * @return the key object that was read into, or null if no
    more
     */
    public boolean nextKeyValue() throws IOException,
    InterruptedException;

    /**
     * Get the current key.
     * @return the current key object or null if there isn't one
     * @throws IOException
     * @throws InterruptedException
     */
    public TKI getCurrentKey() throws IOException,
    InterruptedException;

    /**
     * Get the current value.
     * @return the value object that was read into
```

```
     * @throws IOException
     * @throws InterruptedException
     */
    public TVI getCurrentValue() throws IOException,
     InterruptedException;

    /**
     * Generate an output key/value pair.
     */
    public void write(TKO key, TVO value)
            throws IOException, InterruptedException;

}
```

Listing B.3. Java Map–Reduce `ITaskRunner` interface

```
package edu.utexas.orc;
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.List;
import java.util.Map;

/**
 * Created by crossbach on 12/7/2016.
 */
public interface ITaskRunner<TMapKI, TMapVI, TMapKO, TMapVO,
     TReduceKO, TReduceVO> extends Remote {
    Map<TMapKO, Iterable<TMapVO>> runMapper(Mapper<TMapKI, TMapVI
     , TMapKO, TMapVO> mapper, Map<TMapKI, Iterable<TMapVI>> data)
      throws RemoteException;
    Map<TMapKO, Iterable<TMapVO>> runCombiner(List<Map<TMapKO,
     Iterable<TMapVO>>> shards) throws RemoteException;
    Map<TReduceKO, TReduceVO> runReducer(Reducer<TMapKO, TMapVO,
     TReduceKO, TReduceVO> reducer, Map<TMapKO, Iterable<TMapVO>>
     data) throws RemoteException;
}
```

Listing B.4. Java Map–Reduce `Main` class

```
package edu.utexas.orc;

import java.io.File;
import java.io.IOException;
import java.rmi.NotBoundException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.util.Arrays;
import java.util.List;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

public class Main {

    public static void main(String[] args) throws Exception {

        Map<String, List<Integer>> data = new ConcurrentHashMap<
     String, List<Integer>>();
        data.put("primes", Arrays.asList(2, 3, 5, 7, 11));
        data.put("odds", Arrays.asList(1,3,5,7,9,11));
        data.put("trees", Arrays.asList(1,2,3,6,11));
        runMapReduce(data);
    }

    static Process startProcess(Class<?> cls, String... arguments
     )
            throws IOException {
        String[] args = new String[arguments.length + 2];
        args[0] = "java";
        args[1] = cls.getName();
        System.arraycopy(arguments, 0, args, 2, arguments.length)
     ;
        // System.out.println("Starting process with args list: "
      +
        // Arrays.asList(args));
        ProcessBuilder pb = new ProcessBuilder(args);
        pb.directory(new File(System.getProperty("user.dir") + "/
```

```
out/production/DistributedMapReduce"));
    pb.inheritIO();
    return pb.start();
}

public static void runMapReduce(Map<String, List<Integer>>
data) throws Exception {

    int nProcs = data.size();
    Registry registry = LocateRegistry.createRegistry(1099);
    Process[] procs = new Process[nProcs];
    ITaskRunner[] runners = new ITaskRunner[nProcs];

    try {
        for (int i = 0; i < nProcs; i++) {
            String name = "tasker" + i;
            procs[i] = startProcess(MapReduceTasker.class,
name);
        }

        for (int i = 0; i < nProcs; i++) {
            String name = "tasker" + i;
            while (runners[i] == null) {
                try {
                    System.out.print("."); System.out.flush()
;
                    runners[i] = (ITaskRunner) registry.
lookup(name);
                } catch (NotBoundException e) {
                    // Give the server 1 second more to
register.
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e1) {
                    }
                }
            }
        }
```

```
        Mapper<String, Integer, Integer, Integer> mapper =
    new Mapper();
        Reducer<Integer, Integer, Integer, Integer> reducer =
     new Reducer();
        MapReduce<String, Integer, Integer, Integer, Integer,
     Integer> mr = new MapReduce(mapper, reducer, data);
        Map<Integer, Integer> result = mr.run(runners);
        for(Integer k : result.keySet()) {
            System.out.println(k + " --> " + result.get(k));
        }
        System.out.println("done!");
        Thread.sleep(1000);
      } finally {
        for (Process proc : procs) {
            if (proc != null)
                proc.destroyForcibly();
        }
    }
}
```

Listing B.5. Java Map–Reduce `MapContext` class

```
package edu.utexas.orc;

import javafx.util.Pair;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

/**
 * Created by crossbach on 12/7/2016.
 */
public class MapContext<TKI, TVI, TKO, TVO> implements IContext<
    TKI, TVI, TKO, TVO> {
```

```
ArrayList<Pair<TKI, TVI>> input = null;
ArrayList<Pair<TKO, TVO>> intermediate = null;
Iterator<Pair<TKI, TVI>> keyIterator = null;
Pair<TKI, TVI> currentInputPair = null;

public MapContext(Map<TKI, Iterable<TVI>> data) {
    super();
    input = new ArrayList<Pair<TKI, TVI>>();
    for(TKI k : data.keySet()) {
        Iterable<TVI> vs = data.get(k);
        for(TVI v : vs) {
            input.add(new Pair(k, v));
        }
    }
    intermediate = new ArrayList<Pair<TKO, TVO>>();
}

public Map<TKO, Iterable<TVO>> getOutput() {
    Map<TKO, Iterable<TVO>> result = new ConcurrentHashMap<
TKO, Iterable<TVO>>();
    for(Pair<TKO, TVO> p : intermediate) {
        ArrayList<TVO> vs = (ArrayList<TVO>)result.
getOrDefault(p.getKey(), null);
        if(vs == null) {
            vs = new ArrayList<TVO>();
            result.put(p.getKey(), vs);
        }
        vs.add(p.getValue());
        System.out.println("ctxt.getOutput: " + p.getKey() +
", " + vs.get(0) + "("+vs.size()+"x)");
    }
    return result;
}

public void write(TKO k, TVO v) {
    intermediate.add(new Pair(k, v));
}

public boolean nextKeyValue() {
    if(keyIterator == null) {
        keyIterator = input.iterator();
    }
    return keyIterator.hasNext();
}

public TKI getCurrentKey() {
    if(keyIterator != null) {
        if(currentInputPair == null)
            currentInputPair = keyIterator.next();
        return currentInputPair.getKey();
    }
    throw new RuntimeException("broken iterator!");
}

public TVI getCurrentValue() {
    if(keyIterator != null) {
        if(currentInputPair == null)
            currentInputPair = keyIterator.next();
        return currentInputPair.getValue();
    }
    throw new RuntimeException("broken iterator!");
}

public void advance() {
    if(keyIterator != null) {
        currentInputPair = null;
        return;
    }
    throw new RuntimeException("broken iterator!");
}


}
```

Listing B.6. Java Map–Reduce `Mapper` class

```
package edu.utexas.orc;
import java.io.IOException;
import java.io.Serializable;

/**
 * Created by crossbach on 12/7/2016.
 */
public class Mapper<TKI, TVI, TKO, TVO> implements Serializable {

    /**
     * Called once for each key/value pair in the input split.
    Most applications
     * should override this, but the default is the identity
    function.
     */
    @SuppressWarnings("unchecked")
    protected void map(TKI key, TVI value, MapContext<TKI, TVI,
    TKO, TVO> context) throws IOException, InterruptedException {
        // default just dumps the input to the output
        System.out.println("Mapper: " + key + " --> " + value);
        context.write((TKO) value, (TVO) new Integer(1));
    }

    /**
     * Called once at the end of the task.
     */
    protected void cleanup() throws IOException,
    InterruptedException {
    }

    public void run(MapContext<TKI, TVI, TKO, TVO> context)
    throws IOException, InterruptedException {
        try {
            while (context.nextKeyValue()) {
                map(context.getCurrentKey(),
                        context.getCurrentValue(),
                        context);
                context.advance();
            }
        } finally {
        }
    }
}
```

Listing B.7. Java Map–Reduce `MapReduce` class

```
package edu.utexas.orc;

import java.util.*;
import java.util.concurrent.*;

/**
 * Created by crossbach on 12/7/2016.
 */
public class MapReduce<TMapKI, TMapVI, TMapKO, TMapVO, TReduceKO,
     TReduceVO> {

    private Mapper<TMapKI, TMapVI, TMapKO, TMapVO> m_mapper;
    private Reducer<TMapKO, TMapVO, TReduceKO, TReduceVO>
    m_reducer;
    Map<TMapKI, Iterable<TMapVI>> m_data;
    List<Map<TMapKI, Iterable<TMapVI>>> m_shards;

    public Map<TReduceKO, TReduceVO> run(ITaskRunner<TMapKI,
    TMapVI, TMapKO, TMapVO, TReduceKO, TReduceVO>[] _taskers) {

        int threads = _taskers.length;
        Map<TReduceKO, TReduceVO> result = null;
        final ITaskRunner<TMapKI, TMapVI, TMapKO, TMapVO,
    TReduceKO, TReduceVO>[] taskers = _taskers;
        ITaskRunner<TMapKI, TMapVI, TMapKO, TMapVO, TReduceKO,
    TReduceVO> reduceTasker = _taskers[0];
        ITaskRunner<TMapKI, TMapVI, TMapKO, TMapVO, TReduceKO,
    TReduceVO> combinerTasker = _taskers[0];
```

```
    ExecutorService pool = Executors.newFixedThreadPool(
threads);
    CompletionService<Map<TMapKO, Iterable<TMapVO>>>
futurePool =
            new ExecutorCompletionService<Map<TMapKO,
Iterable<TMapVO>>>(pool);
    Set<Future<Map<TMapKO, Iterable<TMapVO>>>> futureSet =
new HashSet<Future<Map<TMapKO, Iterable<TMapVO>>>>();
    m_shards = new ArrayList<Map<TMapKI, Iterable<TMapVI>>>()
;
    for(TMapKI k : m_data.keySet()) {
        ConcurrentHashMap<TMapKI, Iterable<TMapVI>> shard =
new ConcurrentHashMap<TMapKI, Iterable<TMapVI>>();
        shard.put(k, m_data.get(k));
        m_shards.add(shard);
    }

    int _idx=0;
    for (Map<TMapKI, Iterable<TMapVI>> shard : m_shards) {
        final int idx = _idx++;
        final Map<TMapKI, Iterable<TMapVI>> _shard = shard;
        final Mapper<TMapKI, TMapVI, TMapKO, TMapVO> m =
m_mapper;
        Callable<Map<TMapKO, Iterable<TMapVO>>> callable =
new Callable<Map<TMapKO, Iterable<TMapVO>>>() {
            @Override
            public Map<TMapKO, Iterable<TMapVO>> call()
throws Exception {
                return taskers[idx].runMapper(m, _shard);
            }
        };
        futureSet.add(futurePool.submit(callable));
    }
    pool.shutdown();
    int n = futureSet.size();
    System.out.println("futureSet size is " + n);
    ReducerContext<TMapKO, TMapVO, TReduceKO, TReduceVO>
reducerContext = null;
    Map<TMapKO, Iterable<TMapVO>> reducerIn = null;
    List<Map<TMapKO, Iterable<TMapVO>>> combinerInput = new
ArrayList<Map<TMapKO, Iterable<TMapVO>>>();
    try {
        for (int i = 0; i < n; i++) {
            combinerInput.add(futurePool.take().get());
        }
        reducerIn = combinerTasker.runCombiner(combinerInput)
;
        result = reduceTasker.runReducer(m_reducer, reducerIn
);
    } catch(Exception e) {
        System.out.println(e.getMessage());
    }
    return result;
}

public MapReduce(Mapper<TMapKI, TMapVI, TMapKO, TMapVO> mapfn
,
                 Reducer<TMapKO, TMapVO, TReduceKO, TReduceVO>
reducefn,
                 Map<TMapKI, Iterable<TMapVI>> data) {
    m_mapper = mapfn;
    m_reducer = reducefn;
    m_data = data;
}

}
```

Listing B.8. Java Map–Reduce `MapReduceTasker` class

```
package edu.utexas.orc;

import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;
import java.rmi.registry.LocateRegistry;
```

```
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import java.util.*;
import java.util.concurrent.*;

/**
 * Created by crossbach on 12/7/2016.
 */
public class MapReduceTasker<TMapKI, TMapVI, TMapKO, TMapVO,
    TReduceKO, TReduceVO>
        implements ITaskRunner<TMapKI, TMapVI, TMapKO, TMapVO,
    TReduceKO, TReduceVO> {

    private Mapper<TMapKI, TMapVI, TMapKO, TMapVO> m_mapper;
    private Reducer<TMapKO, TMapVO, TReduceKO, TReduceVO>
    m_reducer;
    MapContext<TMapKI, TMapVI, TMapKO, TMapVO> m_mapContext;
    ReducerContext<TMapKO, TMapVO, TReduceKO, TReduceVO>
    m_reducerContext;

    public Map<TMapKO, Iterable<TMapVO>> runCombiner(List<Map<
    TMapKO, Iterable<TMapVO>>> shards) {

        Map<TMapKO, Iterable<TMapVO>> result = new
    ConcurrentHashMap<TMapKO, Iterable<TMapVO>>();
        for(Map<TMapKO, Iterable<TMapVO>> shard : shards) {
            for(TMapKO ko : shard.keySet()) {
                ArrayList<TMapVO> vs = (ArrayList<TMapVO>)result.
    getOrDefault(ko, null);
                if(vs == null) {
                    vs = new ArrayList<TMapVO>();
                    result.put(ko, vs);
                }
                for(TMapVO vo : shard.get(ko)) {
                    vs.add(vo);
                }
            }
        }
        return result;
    }

    public Map<TMapKO, Iterable<TMapVO>> runMapper(Mapper<TMapKI,
     TMapVI, TMapKO, TMapVO> mapper, Map<TMapKI, Iterable<TMapVI
    >> data) {
        try {
            m_mapper = mapper;
            m_mapContext = new MapContext<TMapKI, TMapVI, TMapKO,
     TMapVO>(data);
            m_mapper.run(m_mapContext);
            return m_mapContext.getOutput();
        } catch (Exception e) {
            System.out.println(e.getMessage());
            e.printStackTrace();
            return null;
        }
    }

    public Map<TReduceKO, TReduceVO>
    runReducer(Reducer<TMapKO, TMapVO, TReduceKO, TReduceVO>
    reducer,
        Map<TMapKO, Iterable<TMapVO>> data) {
        try {
            m_reducer = reducer;
            m_reducerContext = new ReducerContext<TMapKO, TMapVO,
     TReduceKO, TReduceVO>(data);
            m_reducer.run(m_reducerContext);
            return m_reducerContext.getOutput();
        } catch (Exception e) {
            System.out.println(e.getMessage());
            e.printStackTrace();
            return null;
        }
    }

    public MapReduceTasker() {
    }
```

```
    public static void main(String[] args) throws Exception {

        String name = args[0];
        Registry registry = LocateRegistry.getRegistry();
        MapReduceTasker tasker = new MapReduceTasker();
        ITaskRunner stub = (ITaskRunner) UnicastRemoteObject.
    exportObject(tasker, 0);
        registry.rebind(name, stub);
    }

}
```

Listing B.9. Java Map–Reduce `Reducer` class

```
package edu.utexas.orc;
import java.io.IOException;
import java.io.Serializable;
import java.util.Map;

public class Reducer<TInputKey, TInputValue, TOutputKey,
    TOutputValue>
    implements Serializable {

    ReducerContext m_context;

    /**
     * This method is called once for each key. Most applications
     will define
     * their reduce class by overriding this method. The default
    implementation
     * is an identity function.
     */
    @SuppressWarnings("unchecked")
    protected void reduce(TInputKey key, Iterable<TInputValue>
    values, ReducerContext context
    ) throws IOException, InterruptedException {
        m_context = context;
        Integer sum = new Integer(0);
        for(TInputValue value: values) {
            System.out.println("Reducer: " + key + " --> " +
    value);
            sum += (Integer) value;
        }
        context.write((TOutputKey) key, (TOutputValue) sum);
    }

    Map<TOutputKey, TOutputValue> getOutput() {
        return m_context.getOutput();
    }

    /**
     * Called once at the end of the task.
     */
    protected void cleanup(ReducerContext context
    ) throws IOException, InterruptedException {
        // NOTHING
    }


    public void run(ReducerContext context) throws IOException,
    InterruptedException {

        try {
            while (context.nextKey()) {
                reduce((TInputKey)context.getCurrentKey(),
    context.getValues(), context);
                context.advance();
            }
        } finally {

        }
    }
}
```

Listing B.10. Java Map–Reduce `ReducerContext` class

```
package edu.utexas.orc;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

/**
 * Created by crossbach on 12/7/2016.
 */
public class ReducerContext<TKI, TVI, TKO, TVO> implements
     IContext<TKI, TVI, TKO, TVO> {

    Map<TKI, Iterable<TVI>> input;
    ConcurrentHashMap<TKO, TVO> output;
    Iterator<TKI> keyIterator = null;
    TKI currentKey = null;
    Iterable<TVI> currentValues = null;


    public ReducerContext(Map<TKI, Iterable<TVI>> data) {
        super();
        input = data;
        output = new ConcurrentHashMap<TKO,TVO>();
    }

    public Map<TKO, TVO> getOutput() {
        return output;
    }

    public void write(TKO k, TVO v) {
        output.put(k, v);
    }

    public boolean nextKeyValue() {
        return nextKey();
    }

    public boolean nextKey() {
        if(keyIterator == null) {
            keyIterator = input.keySet().iterator();
        }
        return keyIterator.hasNext();
    }

    public TKI getCurrentKey() {
        if(keyIterator != null) {
            if(currentKey == null)
                currentKey = keyIterator.next();
            return currentKey;
        }
        throw new RuntimeException("broken iterator!");
    }

    public TVI getCurrentValue() {
        throw new RuntimeException("broken iterator!");
    }

    public Iterable<TVI> getValues() {
        if(keyIterator != null) {
            if(currentValues == null)
                currentValues = input.get(getCurrentKey());
            return currentValues;
        }
        throw new RuntimeException("broken iterator!");
    }

    public void advance() {
        if(keyIterator != null) {
            currentKey = null;
            currentValues = null;
            return;
        }
```

```
        throw new RuntimeException("broken iterator!");
    }


}
```

B.2. Randomized Byzantine Agreement

B.2.1. Orc Randomized Byzantine Agreement

Listing B.11. Orc Randomized Byzantine Agreement

```
{- randomized-byzantine.orc -- Orc program with a randomized
    solution to Byzantine agreement
 -}

import class Map = "scala.collection.mutable.HashMap"

-- number of bad processes
val t = 2
-- total number of processes
val p = 8*t+1
-- list of process out channels
val channels = collect(lambda () = upto(p) >> Channel[Integer]())

-- Return a random boolean
def coin() = Random(2) :> 0

-- Return a default value if the first argument is None
def default[A](Option[A], A) :: A
def default(None(), v) = v
def default(Some(v), _) = v

-- Tally a list of votes and return
-- (majority vote value, majority vote count)
def tallyVotes(votes :: List[Integer]) =
  val table = Map[Integer, Integer]()
  def tallyVote((mv, mt) :: (Integer, Integer), v :: Integer) =
    val t = default(table.get(v) :!: Option[Integer], 0)
    val newt = t+1
    table.put(v, newt) >>
    (
    if newt :> mt then
      (v, newt)
    else
      (mv, mt)
    )
  foldl(tallyVote, (0, 0), votes)

-- decision algorithm for a good process
def good(maj :: Integer, tally :: Integer) =
  val threshold = (if coin() then 5*t else 6*t)
  if tally :> threshold then maj else 0

-- decision algorithm for a bad process
def bad(_ :: Integer, _ :: Integer) = Random(2)

val nRounds = Ref[Integer](0)

-- generic process; pick is the decision algorithm
def process(pick :: lambda(Integer, Integer) :: Integer) ::
    lambda(Channel[Integer]) :: Integer =
  lambda (out) = (
    -- vote for a value
    def vote(value :: Integer) = map(lambda (_::Top) = out.put(
    value), channels)
    -- receive votes
    def receive() = map(lambda (c :: Channel[Integer]) = c.get(),
     channels)
    -- execute one round
    def round(value :: Integer, n :: Integer) :: Integer =
      -- count the round
```

```
      nRounds := n >>
      -- collect and tally votes from every process
      tallyVotes(receive()) >(maj, tally)>
      pick(maj, tally) >newValue>
      vote(newValue) >>
      ( if tally :> 7*t then newValue else round(newValue, n+1) )
      #
    Random(2) >value>
    vote(value) >>
    round(value, 1)
  )

  Println("Bad: " + map(process(bad), take(t, channels))) >> stop
| Println("Good: " + map(process(good), drop(t, channels))) >>
    stop
; Println("Rounds: " + nRounds?) >> stop
```


B.2.2. Java Randomized Byzantine Agreement

Listing B.12. Java Randomized Byzantine Agreement ByzantineAgreementParticipant class

```
package edu.utexas.orc;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketException;
import java.util.StringTokenizer;

public class ByzantineAgreementParticipant {

    //Each instance will open a socket on (START_PORT + myId)
    private static final int START_PORT = 8000;
    private static final String LOCAL_HOST = "127.0.0.1";

    private static int myId;
    private static int numProcesses;
    private static int myPort;
    private static int testerPort; //the port number of the
    tester process

    public static void main(final String[] args) throws Exception
     {

        myId = Integer.parseInt(args[0]);
        numProcesses = Integer.parseInt(args[1]);
        testerPort = Integer.parseInt(args[2]);
        myPort = START_PORT + myId;

        System.out.println("BA " + myId);

        // Create a server socket for myself
        final ServerSocket serverSocket = new ServerSocket(myPort
    );

        // Initialize MessageHandler
        final MessageManager msgManager = MessageManager.
    getInstance();
        msgManager.init(myId, numProcesses);

        // Create instance of the Byzantine algorithm
        // and set it to handle incoming messages
        final RandomizedByzantineAgreement wbga = new
    RandomizedByzantineAgreement(myId, numProcesses, msgManager);
        msgManager.setMessageHandler(wbga);

        // Accept connections from all processes with a lower ID
        for (int i = 0; i < myId; i++) {
            final Socket s = serverSocket.accept();
            final BufferedReader dIn = new BufferedReader(new
    InputStreamReader(s.getInputStream()));
```

```
        final String getLine = dIn.readLine();
        final StringTokenizer st = new StringTokenizer(
getLine);
        final int theirId = Integer.parseInt(st.nextToken());
        st.nextToken(); //read destId, but don't use it
        final String tag = st.nextToken();
        if (tag.equals("hello")) {
            // Save connection
            msgManager.setChannelSocket(theirId, s);
            msgManager.setChannelInput(theirId, dIn);
            msgManager.setChannelOutput(theirId, new
PrintWriter(s.getOutputStream()));
            //Start thread to listen on this port
            new ListenerThread(theirId, msgManager).start();
        }
    }

    // Connect to processes with a higher ID
    for (int i = myId + 1; i < numProcesses; i++) {
        Socket s = null;
        //try to connect to process i, until the connection
is made
        while (s == null) {
            try {
                s = new Socket(LOCAL_HOST, START_PORT + i);
            } catch (final SocketException e) {
            }
        }
        final PrintWriter pr = new PrintWriter(s.
getOutputStream());
        final BufferedReader br = new BufferedReader(new
InputStreamReader(s.getInputStream()));
        // send hello message to the process
        pr.println(myId + " " + i + " " + "hello" + " " + "
null");
        pr.flush();
        //Save connection
        msgManager.setChannelSocket(i, s);
            msgManager.setChannelInput(i, br);
            msgManager.setChannelOutput(i, pr);
            //Start thread to listen on this port
            new ListenerThread(i, msgManager).start();
        }

        // Create connection to the tester program
        final Socket testerSocket = new Socket(LOCAL_HOST,
    testerPort);
        msgManager.setTesterSocket(testerSocket);
        final PrintWriter testerPr = new PrintWriter(testerSocket
    .getOutputStream());
        testerPr.println(myId + " " + "hello" + " " + "null"); //
     Send connection message
        msgManager.setTesterInput(new BufferedReader(new
    InputStreamReader(testerSocket.getInputStream())));
        msgManager.setTesterOutput(testerPr);
        // Start thread to listen on this port
        new ListenerThread(-1, msgManager).start();

        // Inform tester process that we are initialized
        msgManager.sendTesterMessage("ready", null);

        // Wait forever, ignore new connections
        // Change this?
        while (true) {
            serverSocket.accept();
        }
    }

}
```

Listing B.13. Java Randomized Byzantine Agreement `ConnectionManager` class

```
package edu.utexas.orc;

import java.io.BufferedReader;
import java.io.IOException;
```

```
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.StringTokenizer;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.TimeUnit;

public class ConnectionManager extends Thread {

    private final int port;

    // Incoming message from all processes are placed in this
     queue
    private final BlockingQueue<TesterMessage> queue;

    private final Socket[] sockets;
    private final PrintWriter[] dataOut;

    public ConnectionManager(final int port, final int
     numProcesses) {
        this.port = port;

        queue = new LinkedBlockingQueue<>();
        sockets = new Socket[numProcesses];
        dataOut = new PrintWriter[numProcesses];

        this.start();
    }

    @Override
    public void run() {
        try {
            final ServerSocket serverSocket = new ServerSocket(
     port);

            //Wait for a new connection and start a
ListenerThread for it
            while (true) {
                //Get connection
                final Socket s = serverSocket.accept();

                //Read first message to determine ID
                final BufferedReader dIn = new BufferedReader(new
 InputStreamReader(s.getInputStream()));
                final String getLine = dIn.readLine();
                final StringTokenizer st = new StringTokenizer(
getLine);
                final int theirId = Integer.parseInt(st.nextToken
());
                final String tag = st.nextToken();
                if (tag.equals("hello")) {
                    // Save connection
                    sockets[theirId] = s;
                    dataOut[theirId] = new PrintWriter(s.
getOutputStream());
                    // Start listener thread
                    new TestListenerThread(dIn, queue).start();
                }
            }
        } catch (final IOException e) {
            e.printStackTrace();
        }
    }

    /**
     * Sends a message to a launched process.
     *
     * Message format: "<srcId> <destId> <tag> <message>#"
     * srcId is -1, to indicate the message is from the tester
    process
     */
    public void sendMessage(final int destId, final String tag,
     final String message) {
        dataOut[destId].println(Integer.toString(-1) + " " +
```

```
    Integer.toString(destId) + " " + tag + " " + message + "#");
        dataOut[destId].flush();
    }

    /**
     * Attempts to retrieve message from queue. Will block for up
     to 1 second.
     *
     * Returns a TesterMessage if successful, or null if it times
     out.
     *
     */
    public TesterMessage getNextMessage() throws
     InterruptedException {
        return queue.poll(1, TimeUnit.SECONDS);
    }
}
```

Listing B.14. Java Randomized Byzantine Agreement `IMessageHandler` interface

```
package edu.utexas.orc;

public interface IMessageHandler {
    public void _handleMessage(Message m, int srcId, String tag);
}
```

Listing B.15. Java Randomized Byzantine Agreement `ListenerThread` class

```
package edu.utexas.orc;

import java.io.BufferedReader;
import java.io.IOException;
import java.net.SocketException;

public class ListenerThread extends Thread {

    private final int channelId;
    private final MessageManager msgManager;

    public ListenerThread(final int channelId, final
     MessageManager msgManager) {
        this.channelId = channelId;
        this.msgManager = msgManager;
    }

    @Override
    public void run() {
        BufferedReader dataIn;
        if (channelId == -1) {
            dataIn = msgManager.getTesterInput();
        } else {
            dataIn = msgManager.getChannelInput(channelId);
        }

        while (true) {
            try {
                final String line = dataIn.readLine();
                msgManager.receiveMessage(channelId, line);
            } catch (final SocketException e) {
                //Lost connection to process, probably because
     process terminated.
                //No reason to keep listening, so return, killing
      thread.
                return;
            } catch (final IOException e) {
                e.printStackTrace();
            }
        }
    }

}
```

Listing B.16. Java Randomized Byzantine Agreement `Main` class

```
package edu.utexas.orc;
```

```
import java.io.File;
import java.util.ArrayList;

public class Main {

    /* Adapted/simplified to make Orc-comparable by cjr based on:
        https://code.google.com/archive/p/weighted-byzantine-
    agreement/
     */

    private static final int NUM_PROCESSES = 6;
    private static String CURRENT_DIR = "C:\\cygwin64\\home\\
    crossbach\\papers\\papers-dorc-1\\evaluation\\
    RandomizedByzantineAgreement\\out\\production\\
    RandomizedByzantineAgreement";
    private static int PORT = 7999;
    private static int MAX_WAIT = 10;
    static StopWatch stopWatch = new StopWatch();

    public static void main(final String[] args) throws Exception
     {
        final Process processes[] = new Process[NUM_PROCESSES];
        final ConnectionManager connManager = new
    ConnectionManager(PORT, NUM_PROCESSES);

        //Launch processes
        System.out.println("Launching " + NUM_PROCESSES + "
    processes.");
        for (int i = 0; i < NUM_PROCESSES; i++) {
            final ProcessBuilder pb = new ProcessBuilder("java",
    "edu/utexas/orc/ByzantineAgreementParticipant", Integer.
    toString(i), Integer.toString(NUM_PROCESSES), Integer.
    toString(PORT));
            pb.directory(new File(CURRENT_DIR));
            pb.redirectErrorStream(true);
            processes[i] = pb.start();
            final StreamGobbler sg = new StreamGobbler(i,
processes[i].getInputStream());
            sg.start();
        }
        System.out.println("Finished launching " + NUM_PROCESSES
+ " processes.");

        //Wait for "ready" messages from all launched processes
        //Wait for MAX_WAIT, then fail
        System.out.println("Waiting for 'ready' message from all
launched processes");
        int numReadysReceived = 0;
        stopWatch.start();
        while (numReadysReceived < NUM_PROCESSES && stopWatch.
getElapsedTimeSecs() < MAX_WAIT) {
            final TesterMessage msg = connManager.getNextMessage
();
            if (msg != null) {
                if (msg.getTag().equals("ready")) {
                    numReadysReceived++;
                }
            }
        }

        //Check if all processes sent 'ready' message
        if (numReadysReceived == NUM_PROCESSES) {
            System.out.println("Received 'ready' from all
processes.");
        } else {
            System.out.println("Only received 'ready' from " +
numReadysReceived + " processes.");
            System.out.println("Exiting.");
            killProcessesAndExit(processes);
        }

        //Start the agreement process
        stopWatch.start();
        connManager.sendMessage(0, "initAgreement", null);
```

```
    //Wait for all processes to decide, time out after
MAX_WAIT
    System.out.println("Waiting for 'decide' message from all
 launched proceseses");
    int numDecidesReceived = 0;
    final ArrayList<Integer> decisions = new ArrayList<>();
    while (numDecidesReceived < NUM_PROCESSES && stopWatch.
getElapsedTimeSecs() < MAX_WAIT) {
        final TesterMessage msg = connManager.getNextMessage
();
        if (msg != null) {
            if (msg.getTag().equals("decide")) {
                numDecidesReceived++;
                decisions.add(Integer.parseInt(msg.getMessage
().substring(1)));
            }
        }
    }
    stopWatch.stop();

    //Display results
    if (numDecidesReceived == NUM_PROCESSES) {
        System.out.println("Received 'decide' from all
processes.");
        System.out.println("Agreement took " + stopWatch.
getElapsedTime() + " ms");
    } else {
        System.out.println("Agreement failed. Only received "
 + numDecidesReceived + "'decide' messages.");
    }

    //Verify all processes made same decision
    if (decisions.size() > 0) {
        final int first = decisions.get(0);
        boolean match = true;
        for (int i = 1; i < decisions.size(); i++) {
            if (decisions.get(i) != first) {
                match = false;
            }
        }
        if (match) {
            System.out.println("All processes decided the
 same value: " + first);
        } else {
            System.out.print("The processes did not all
 decide the same value:");
            System.out.println(decisions);
        }
    }

    killProcessesAndExit(processes);
}

/**
 * Kills the launched processes and exits.
 */
private static void killProcessesAndExit(final Process[]
 processes) {
    //kill processes
    System.out.println("Killing all processes.");
    for (int i = 0; i < NUM_PROCESSES; i++) {
        if (processes[i] != null) {
            processes[i].destroy();
        }
    }
    System.out.println("Finished killing all processes.");

    System.exit(0);
}

}
```

Listing B.17. Java Randomized Byzantine Agreement `Message` class

```
package edu.utexas.orc;
```

```
import java.util.StringTokenizer;

public class Message {
    private final int srcId;
    private final int destId;
    private final String tag;
    private final String msg;

    public Message(final int srcId, final int destId, final
     String tag, final String msg) {
        this.srcId = srcId;
        this.destId = destId;
        this.tag = tag;
        this.msg = msg;
    }

    public int getSrcId() {
        return srcId;
    }

    public int getDestId() {
        return destId;
    }

    public String getTag() {
        return tag;
    }

    public String getMessage() {
        return msg;
    }

    public static Message parseMessage(final String s) {
        final StringTokenizer st = new StringTokenizer(s);
        final int srcId = Integer.parseInt(st.nextToken());
        final int destId = Integer.parseInt(st.nextToken());
        final String tag = st.nextToken();
        final String msg = st.nextToken("#");
        return new Message(srcId, destId, tag, msg);
    }

    @Override
    public String toString() {
        final String s = String.valueOf(srcId) + " " + String.
     valueOf(destId) + " " + tag + " " + msg + "#";
        return s;
    }
}
```

Listing B.18. Java Randomized Byzantine Agreement MessageManager class

```
package edu.utexas.orc;

import java.io.BufferedReader;
import java.io.PrintWriter;
import java.net.Socket;

public class MessageManager {

    //Singleton class
    private static MessageManager instance = null;

    private int myId;

    private Socket[] sockets;
    private PrintWriter[] dataOut;
    private BufferedReader[] dataIn;

    private Socket testerSocket;
    private PrintWriter testerDataOut;
    private BufferedReader testerDataIn;

    private IMessageHandler msgHandler;

    protected MessageManager() {
        //Singleton class
```

```
}

public static MessageManager getInstance() {
    if (instance == null) {
        instance = new MessageManager();
    }
    return instance;
}

public void init(final int myId, final int numProcesses) {
    this.myId = myId;
    sockets = new Socket[numProcesses];
    dataOut = new PrintWriter[numProcesses];
    dataIn = new BufferedReader[numProcesses];
}

public void setMessageHandler(final IMessageHandler
 msgHandler) {
    this.msgHandler = msgHandler;
}

public Socket getChannelSocket(final int id) {
    return sockets[id];
}

public void setChannelSocket(final int id, final Socket s) {
    sockets[id] = s;
}

public BufferedReader getChannelInput(final int id) {
    return dataIn[id];
}

public void setChannelInput(final int id, final
 BufferedReader br) {
    dataIn[id] = br;
}

public PrintWriter getChannelOutput(final int id) {
    return dataOut[id];
}

public void setChannelOutput(final int id, final PrintWriter
 pr) {
    dataOut[id] = pr;
}

public Socket getTesterSocket() {
    return testerSocket;
}

public void setTesterSocket(final Socket s) {
    testerSocket = s;
}

public BufferedReader getTesterInput() {
    return testerDataIn;
}

public void setTesterInput(final BufferedReader br) {
    testerDataIn = br;
}

public PrintWriter getTesterOutput() {
    return testerDataOut;
}

public void setTesterOutput(final PrintWriter pr) {
    testerDataOut = pr;
}

/**
 * Process message format: "<srcId> <destId> <tag> <message
>#"
 */
public void sendMessage(final int destId, final String tag,
```

```
        final String message) {
            if (destId != myId) {
                dataOut[destId].println(myId + " " + destId + " " +
        tag + " " + message + "#");
                dataOut[destId].flush();
            } else { //sending message to self
                receiveMessage(myId, myId + " " + destId + " " + tag
        + " " + message + "#");
            }
        }

        /**
         * Tester message format: "<srcId> <tag> <message>#"
         */
        public void sendTesterMessage(final String tag, final String
         message) {
            testerDataOut.println(myId + " " + tag + " " + message +
         "#");
            testerDataOut.flush();
        }

        public void receiveMessage(final int fromId, final String
         line) {
            final Message m = Message.parseMessage(line);
            msgHandler._handleMessage(m, m.getSrcId(), m.getTag());
        }

    }
```

Listing B.19. Java Randomized Byzantine Agreement RandomizedByzantineAgreement class

```
    package edu.utexas.orc;

    import java.util.ArrayList;
    import java.util.LinkedList;
    import java.util.Random;

    public class RandomizedByzantineAgreement implements
     IMessageHandler, Runnable {

        protected int myId;
        protected int numProcesses;
        private final MessageManager msgManager;

        public RandomizedByzantineAgreement(final int myId, final int
         numProcesses, final MessageManager msgManager) {
            this.myId = myId;
            this.numProcesses = numProcesses;
            this.msgManager = msgManager;
            queues = new ArrayList<>();
            hasDecided = new boolean[numProcesses];
            decidedValue = new byte[numProcesses];
            tossValue = new byte[numProcesses];
            for (int i = 0; i < numProcesses; ++i) {
                queues.add(new LinkedList<Byte>());
                hasDecided[i] = false;
                decidedValue[i] = UNDECIDED;
                tossValue[i] = UNDECIDED;
            }
        }

        private final byte UNDECIDED = 2;

        private volatile byte V;
        private volatile int g;
        private volatile int t;

        private volatile Random rand;
        private volatile ArrayList<LinkedList<Byte>> queues;
        private volatile boolean hasDecided[];
        private volatile byte decidedValue[];
        private volatile byte tossValue[];

        private int group(final int p) {
            return p / g;
        }
```

```
private byte tossCoin() {
    return (byte) rand.nextInt(2);
}

public void initAgreement() {
    System.out.println("RandomizedByzantineAgreement:
initAgreement");
    if (myId == 0) {
        //Everyone is symmetric in this algorithm,
        //so go ahead and init everything
        for (int i = 1; i < numProcesses; ++i) {
            sendMessage(i, "initAgreement", null);
        }
    }
    rand = new Random();
    V = (byte) rand.nextInt(2);
    t = (numProcesses - 1) / 3;
    g = 1;
    new Thread(this).start();
}

public void handleMessage(final Message m, final int src,
final String tag) {
    System.out.println("handleMessage: " + src + ", " + tag +
 ", " + m.getMessage());

    if (tag.equals("V")) {
        synchronized (queues.get(src)) {
            queues.get(src).addLast(Byte.parseByte(m.
getMessage().substring(1)));
        }
    } else if (tag.equals("V_toss")) {
        synchronized (queues.get(src)) {
            final String values[] = m.getMessage().substring
(1).split(" ");
            queues.get(src).addLast(Byte.parseByte(values[0])
);
            queues.get(src).addLast(Byte.parseByte(values[1])
);
        }
    } else if (tag.equals("last")) {
        final String values[] = m.getMessage().substring(1).
split(" ");
        decidedValue[src] = Byte.parseByte(values[0]);
        tossValue[src] = Byte.parseByte(values[1]);
        hasDecided[src] = true;
    } else {
        //shouldn't get here...
        throw new RuntimeException("Unknown Tag Received: " +
 tag);
    }
}

@Override
public void run() {
    byte ans;
    byte toss;
    double num;
    final double increment = 1 / (double) numProcesses;
    for (long e = 1; e < Long.MAX_VALUE; ++e) {
        double s0 = 0, s1 = 0;
        //First Phase:
        //send V to all processes
        for (int i = 0; i < numProcesses; i++) {
            sendMessage(i, "V", String.valueOf(V | 16));
        }
        for (int i = 0; i < numProcesses; ++i) {
            boolean gotValue = false;
            byte theValue = -1;
            while (!gotValue) {
                if (hasDecided[i]) {
                    theValue = decidedValue[i];
                    gotValue = true;
                } else {
                    synchronized (queues.get(i)) {
```

```
                        if (!queues.get(i).isEmpty()) {
                            theValue = queues.get(i).
removeFirst();
                            gotValue = true;
                        }
                    }
                }
            }
            theValue = (byte) (theValue & 0x0F);
            if (theValue == 1) {
                s1 += increment;
            } else if (theValue == 0) {
                s0 += increment;
            } else if (theValue == UNDECIDED) {
                throw new RuntimeException("Got UNDECIDED;
expected otherwise");
            } else {
                throw new RuntimeException("Unknown Value
Failure: " + theValue);
            }
        }
        if (s1 >= (double) (numProcesses - t) / (double)
numProcesses) {
            V = 1;
        } else if (s0 >= (double) (numProcesses - t) / (
double) numProcesses) {
            V = 0;
        } else {
            V = UNDECIDED;
        }
        if (group(myId) == e % (numProcesses / g)) {
            toss = tossCoin();
        } else {
            toss = 0;
        }

        s0 = 0;
        s1 = 0;
        double t0 = 0, t1 = 0;
        //Second Phase:
        //send V, toss to all processes
        for (int i = 0; i < numProcesses; i++) {
            sendMessage(i, "V_toss", String.valueOf(V) + " "
+ String.valueOf(toss));
        }
        for (int i = 0; i < numProcesses; ++i) {
            boolean gotValue = false;
            byte theValue = -1;
            byte theToss = -1;
            while (!gotValue) {
                if (hasDecided[i]) {
                    theValue = decidedValue[i];
                    theToss = tossValue[i];
                    gotValue = true;
                } else {
                    synchronized (queues.get(i)) {
                        if (!queues.get(i).isEmpty()) {
                            theValue = queues.get(i).
removeFirst();
                            theToss = queues.get(i).
removeFirst();
                            gotValue = true;
                        }
                    }
                }
            }
            theValue = (byte) (theValue & 0x0F);
            if (theValue == 1) {
                s1 += increment;
            } else if (theValue == 0) {
                s0 += increment;
            } else if (theValue == UNDECIDED) {
                //do nothing
            } else {
                throw new RuntimeException("Unknown Value
Failure: " + theValue);
```

```
            }
            if (group(i) == e % (numProcesses / g)) {
                theToss = (byte) (theToss & 0x0F);
                if (theToss == 1) {
                    t1 += increment;
                } else if (theToss == 0) {
                    t0 += increment;
                } else if (theToss == UNDECIDED) {
                    throw new RuntimeException("Got UNDECIDED
; expected otherwise");
                } else {
                    throw new RuntimeException("Unknown Toss
Failure: " + theToss);
                }
            }
        }

        if (s1 >= s0) {
            ans = 1;
            num = s1;
        } else {
            ans = 0;
            num = s0;
        }
        if (num >= (double) (numProcesses - t) / (double)
numProcesses) {
            for (int i = 0; i < numProcesses; i++) {
                sendMessage(i, "last", String.valueOf(V) + "
" + String.valueOf(toss));
            }
            V = ans;
            decide(V);
            break;
        } else if (num >= (double) (t + 1) / (double)
numProcesses) {
            V = ans;
        } else {
            if (t1 >= t0) {
                V = 1;
            } else {
                V = 0;
            }
        }
    }
}

@Override
public void _handleMessage(final Message m, final int srcId,
 final String tag) {
    if (tag.equals("initAgreement")) {
        initAgreement();
    } else {
        handleMessage(m, srcId, tag);
    }
}

/**
 * Sends an application message to another process.
 */
protected void sendMessage(final int destId, final String tag
, final String message) {
    msgManager.sendMessage(destId, tag, message);
}

/**
 * Call this function once the process has decided on a value
.
 *
 * Sends a message to the tester program, letting the tester
process know
 * that a decision was reached. This message includes the
decided value.
 *
 * @param value the value decided
 */
protected void decide(final int value) {
```

```
        msgManager.sendTesterMessage("decide", Integer.toString(
    value));
    }
}
```

Listing B.20. Java Randomized Byzantine Agreement `StopWatch` class

```
package edu.utexas.orc;

/*
Copyright (c) 2005, Corey Goldberg

StopWatch.java is free software; you can redistribute it and/or
    modify
it under the terms of the GNU General Public License as published
     by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.
*/

public class StopWatch {

    private long startTime = 0;
    private long stopTime = 0;
    private boolean running = false;

    public void start() {
        this.startTime = System.currentTimeMillis();
        this.running = true;
    }

    public void stop() {
        this.stopTime = System.currentTimeMillis();
        this.running = false;
    }

    //elaspsed time in milliseconds
    public long getElapsedTime() {
        long elapsed;
        if (running) {
            elapsed = System.currentTimeMillis() - startTime;
        } else {
            elapsed = stopTime - startTime;
        }
        return elapsed;
    }

    //elaspsed time in seconds
    public long getElapsedTimeSecs() {
        long elapsed;
        if (running) {
            elapsed = (System.currentTimeMillis() - startTime) /
    1000;
        } else {
            elapsed = (stopTime - startTime) / 1000;
        }
        return elapsed;
    }

    //sample usage
    public static void main(final String[] args) {
        final StopWatch s = new StopWatch();
        s.start();
        //code you want to time goes here
        s.stop();
        System.out.println("elapsed time in milliseconds: " + s.
    getElapsedTime());
    }
}
```

Listing B.21. Java Randomized Byzantine Agreement `StreamGobbler` class

```
package edu.utexas.orc;

/**
 * A thread that outputs an InputStream, line by line.
```

```
 *
 * Used for capturing the standard output of another process and
    displaying it
 * in this process' standard output.
 */

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;

public class StreamGobbler extends Thread {

    private final InputStream is;
    private final int pid;

    public StreamGobbler(final int pid, final InputStream is) {
        this.pid = pid;
        this.is = is;
    }

    @Override
    public void run() {
        try {
            final InputStreamReader isr = new InputStreamReader(
    is);
            final BufferedReader br = new BufferedReader(isr);
            String line = null;
            while ((line = br.readLine()) != null) {
                System.out.println("pid " + pid + "> " + line);
            }
        } catch (final IOException ioe) {
            ioe.printStackTrace();
        }
    }
}
```


Listing B.22. Java Randomized Byzantine Agreement `TestListenerThread` class

```
package edu.utexas.orc;

import java.io.BufferedReader;
import java.io.IOException;
import java.net.SocketException;
import java.util.concurrent.BlockingQueue;

public class TestListenerThread extends Thread {

    private final BufferedReader dataIn;
    private final BlockingQueue<TesterMessage> queue;

    public TestListenerThread(final BufferedReader br, final
     BlockingQueue<TesterMessage> q) {
        this.dataIn = br;
        this.queue = q;
    }

    @Override
    public void run() {
        while (true) {
            try {
                final String line = dataIn.readLine();
                final TesterMessage msg = TesterMessage.
    parseMessage(line);
                queue.put(msg);
            } catch (final SocketException e) {
                // Connection is lost, probably due to the
    connected process terminating.
                // No need to continue listening, so return.
                return;
            } catch (final IOException e) {
                e.printStackTrace();
            } catch (final InterruptedException e) {
                e.printStackTrace();
```

```
            }
        }
    }
}
```

Listing B.23. Java Randomized Byzantine Agreement `TesterMessage` class

```
package edu.utexas.orc;

import java.util.StringTokenizer;

public class TesterMessage {
    private final int srcId;
    private final String tag;
    private final String msg;

    public TesterMessage(final int srcId, final String tag, final
     String msg) {
        this.srcId = srcId;
        this.tag = tag;
        this.msg = msg;
    }

    public int getSrcId() {
        return srcId;
    }

    public String getTag() {
        return tag;
    }

    public String getMessage() {
        return msg;
    }

    public static TesterMessage parseMessage(final String s) {
        final StringTokenizer st = new StringTokenizer(s);
        final int srcId = Integer.parseInt(st.nextToken());
        final String tag = st.nextToken();
        final String msg = st.nextToken("#");
        return new TesterMessage(srcId, tag, msg);
    }

}
```

B.3. Dining Philosophers

B.3.1. Orc Dining Philosophers

Listing B.24. Orc Dining Philosophers

```
{- misra-philosopher.orc
 -
 - The "hygenic solution to the diners problem", described in
 - K. M. Chandy and J. Misra. 1984. The drinking philosophers
    problem.
 - ACM Trans. Program. Lang. Syst. 6, 4 (October 1984), 632-646.
 -}

-- Use a Scala set implementation.
-- Operations on this set are _not_ synchronized.
import class ScalaSet = "scala.collection.mutable.HashSet"

{-
Make a set initialized to contain
the items in the given list.
-}
def Set[A](items :: List[A]) = ScalaSet[A]() >s> joinMap(s.add,
    items) >> s
```

```
type Message = (String, lambda((String, lambda(Top) :: Signal))
    :: Signal)
type Xmitter = lambda(Message) :: Signal

{-
Start a philosopher process; never publishes.

name: identify this process in status messages
mbox: our mailbox
missing: set of neighboring philosophers holding our fork
-}
def philosopher(name :: (Integer, Integer), mbox :: Channel[
    Message], missing :: ScalaSet[Xmitter]) :: Bot =
  val send = mbox.put
  val receive = mbox.get
  -- deferred requests for forks
  val deferred = Channel[Xmitter]()
  -- forks we hold which are clean
  val clean = Set[Xmitter]([])

  def sendFork(p :: Xmitter) =
    missing.add(p) >>
    p(("fork", send))

  def requestFork(p :: Xmitter) =
    clean.add(p) >>
    p(("request", send))

  -- While thinking, start a timer which
  -- will tell us when we're hungry
  def digesting() :: Bot =
      Println(name + " thinking") >>
      thinking()
    | Rwait(Random(30)) >>
      send(("rumble", send)) >>
      stop

  def thinking() :: Bot =
    def on(("rumble", _) :: Message) =
      Println(name + " hungry") >>
      map(requestFork, missing.toList() :!: List[Xmitter]) >>
      hungry()
    def on(("request", p)) =
      sendFork(p :!: Xmitter) >>
      thinking()
    on(receive())

  def hungry() :: Bot =
    def on(("fork", p) :: Message) =
      missing.remove(p :!: Xmitter) >>
      (
        if missing.isEmpty() then
          Println(name + " eating") >>
          eating()
        else hungry()
      )
    def on(("request", p)) =
      if clean.contains(p :!: Xmitter) then
        deferred.put(p :!: Xmitter) >>
        hungry()
      else
        sendFork(p :!: Xmitter) >>
        requestFork(p :!: Xmitter) >>
        hungry()
    on(receive())

  def eating() :: Bot =
    clean.clear() >>
    Rwait(Random(10)) >>
    map(sendFork, deferred.getAll()) >>
    digesting()

  digesting()

{-
```

```
Create an NxN 4-connected grid of philosophers.  Each philosopher
     holds the
fork for the connections below and to the right (so the top left
    philosopher
holds both its forks).
-}
def philosophers(n :: Integer) =
  {- channels -}
  val cs = uncurry(Table(n, lambda (_::Top) = Table(n, lambda(
    _::Top) = Channel[Message]())))

  {- first row -}
  philosopher((0,0), cs(0,0), Set[Xmitter]([]))
  | for(1, n) >j>
    philosopher((0,j), cs(0,j), Set[Xmitter]([cs(0,j-1).put]))

  {- remaining rows -}
  | for(1, n) >i> (
      philosopher((i,0), cs(i,0), Set[Xmitter]([cs(i-1,0).put]))
      | for(1, n) >j>
        philosopher((i,j), cs(i,j), Set[Xmitter]([cs(i-1,j).put,
    cs(i,j-1).put]))
    )

philosophers(2)
--val cs = Table(2, lambda(_::Top) = Channel[Message]())
--philosopher((0,0), cs(0), Set[Xmitter]([])) |
--philosopher((1,0), cs(1), Set[Xmitter]([cs(0).put]))
```

B.3.2. Java Dining Philosophers

Listing B.25. Java Dining Philosophers `Channel` class

```
package edu.utexas.orc.channel;

import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import java.util.UUID;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;

public class Channel<T> extends PortableChannelPutter<T>
    implements ChannelPutter<T>, ChannelGetter<T> {
    static Registry registry;
    static {
        try {
            registry = LocateRegistry.getRegistry();
        } catch (final RemoteException e) {
            registry = null;
            throw new Error(e);
        }
    }

    private final BlockingQueue<T> queue = new
    LinkedBlockingQueue<>();

    public Channel() {
        super(UUID.randomUUID());
        final ChannelReceiver<T> server = new ChannelReceiver<T
    >() {
            @Override
            public void put(final T message) throws
    RemoteException {
                //System.out.println("Message sent on channel " +
     uuid + ": " + message);
                queue.add(message);
            }
        };
        try {
            @SuppressWarnings("unchecked")
            final ChannelReceiver<T> stub = (ChannelReceiver<T>)
    UnicastRemoteObject.exportObject(server, 0);
```

```
            registry.rebind(getChannelName(), stub);
        } catch (final RemoteException e) {
            throw new Error(e);
        }
    }

    public ChannelPutter<T> getPutter() {
        return new PortableChannelPutter<>(uuid);
    }

    @Override
    public T get() throws InterruptedException {
        return queue.take();
    }
}
```

Listing B.26. Java Dining Philosophers `ChannelGetter` interface

```
package edu.utexas.orc.channel;

public interface ChannelGetter<T> {

    T get() throws InterruptedException;

}
```

Listing B.27. Java Dining Philosophers `ChannelPutter` interface

```
package edu.utexas.orc.channel;

public interface ChannelPutter<T> {
    void put(T message);
}
```

Listing B.28. Java Dining Philosophers `ChannelReceiver` interface

```
package edu.utexas.orc.channel;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface ChannelReceiver<T> extends Remote {
    public void put(T message) throws RemoteException;
}
```

Listing B.29. Java Dining Philosophers `Main` class

```
package edu.utexas.orc.channel;

import java.io.File;
import java.io.IOException;
import java.rmi.NotBoundException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class Main {
    private static final int nProcs = 15;

    static Process startProcess(final Class<?> cls, final String
    ... arguments) throws IOException {
        final String[] args = new String[arguments.length + 2];
        args[0] = "java";
        args[1] = cls.getName();
        System.arraycopy(arguments, 0, args, 2, arguments.length)
    ;
        // System.out.println("Starting process with args list: "
     +
        // Arrays.asList(args));
        final ProcessBuilder pb = new ProcessBuilder(args);
        pb.directory(new File(System.getProperty("user.dir") + "/
    bin"));
        pb.inheritIO();
        return pb.start();
    }
```

```
public static void main(final String[] args) throws
IOException, InterruptedException {
    Registry registry;
    registry = LocateRegistry.createRegistry(1099);

    final Process[] procs = new Process[nProcs];
    final Philosopher[] philosophers = new Philosopher[nProcs
];
    try {
        for (int i = 0; i < nProcs; i++) {
            final String name = "ph" + i;
            procs[i] = startProcess(PhilosopherImpl.class,
name);
        }

        for (int i = 0; i < nProcs; i++) {
            final String name = "ph" + i;
            while (philosophers[i] == null) {
                try {
                    // System.out.print("."); System.out.
flush();
                    philosophers[i] = (Philosopher) registry.
lookup(name);
                } catch (final NotBoundException e) {
                    // Give the server 1 second more to
register.
                    try {
                        Thread.sleep(1000);
                    } catch (final InterruptedException e1) {
                    }
                }
            }

            if (i > 0) {
                philosophers[i].addConnection(philosophers[i
- 1].getPutter());
            }
            if (i > 1) {
                philosophers[i].addConnection(philosophers[i
- 2].getPutter());
            }
            if (i > 5) {
                philosophers[i].addConnection(philosophers[i
- 6].getPutter());
            }
            philosophers[i].start();
        }

        Thread.sleep(10000);
    } finally {
        for (final Process proc : procs) {
            if (proc != null) {
                proc.destroyForcibly();
            }
        }
    }
}
```

Listing B.30. Java Dining Philosophers `Philosopher` interface

```
package edu.utexas.orc.channel;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Philosopher extends Remote {
    ChannelPutter<Message> getPutter() throws RemoteException;

    void addConnection(ChannelPutter<Message> putter) throws
    RemoteException;

    void start() throws RemoteException;
}
```

Listing B.31. Java Dining Philosophers `PhilosopherImpl` class

```
package edu.utexas.orc.channel;

import java.io.Serializable;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.Queue;
import java.util.Set;
import java.util.Timer;
import java.util.TimerTask;
import java.util.concurrent.Callable;

class Message implements Serializable {
    private static final long serialVersionUID =
     -7539323123578933274L;

    public final ChannelPutter<Message> sender;
    public final String command;

    public Message(final String command, final ChannelPutter<
     Message> sender) {
        this.command = command;
        this.sender = sender;
        // TODO Auto-generated constructor stub
    }

    @Override
    public String toString() {
        return "Message [sender=" + sender + ", command=" +
     command + "]";
    }
}
```

```
@FunctionalInterface
interface PhilosopherState extends Callable<PhilosopherState> {
}

public class PhilosopherImpl implements Philosopher {
    private static Timer timer = new Timer(true);

    //  val send = mbox.put
    //  val receive = mbox.get
    //  -- deferred requests for forks
    //  val deferred = Channel[Xmitter]()
    //  -- forks we hold which are clean
    //  val clean = Set[Xmitter]([])

    private final String name;

    private final Channel<Message> mbox = new Channel<>();
    private final Set<ChannelPutter<Message>> missing = new
     HashSet<>();
    private final Queue<ChannelPutter<Message>> deferred = new
     LinkedList<>();
    private final Set<ChannelPutter<Message>> clean = new HashSet
     <>();
    private final ChannelPutter<Message> myPutter;

    private volatile PhilosopherState state;

    //def philosopher(name :: (Integer, Integer), mbox :: Channel
     [Message], missing :: ScalaSet[Xmitter]) :: Bot =
    public PhilosopherImpl(final String name) {
        this.name = name;
        myPutter = mbox.getPutter();
    }

    //  def sendFork(p :: Xmitter) =
    //    missing.add(p) >>
    //    p(("fork", send))
```

```
private void sendFork(final ChannelPutter<Message> other) {
    missing.add(other);
    // System.out.println(name + " giving fork to " + other +
 "(" + missing + " " + deferred + " " + clean + ")");
    other.put(new Message("fork", myPutter));
}

//
//  def requestFork(p :: Xmitter) =
//    clean.add(p) >>
//    p(("request", send))

private void requestFork(final ChannelPutter<Message> other)
{
    clean.add(other);
    // System.out.println(name + " requesting fork from " +
other + "(" + missing + " " + deferred + " " + clean + ")");
    other.put(new Message("request", myPutter));
}

//
//  -- While thinking, start a timer which
//  -- will tell us when we're hungry
//  def digesting() :: Bot =
//      Println(name + " thinking") >>
//      thinking()
//    | Rwait(Random(30)) >>
//      send(("rumble", send)) >>
//      stop

private PhilosopherState digesting() {
    System.out.println(name + " " + "thinking");
    timer.schedule(new TimerTask() {
        @Override
        public void run() {
            myPutter.put(new Message("rumble", myPutter));
        }
    }, 20);
    return this::thinking;
}

//
//  def thinking() :: Bot =
//    def on(("rumble", _) :: Message) =
//      Println(name + " hungry") >>
//      map(requestFork, missing.toList() :!: List[Xmitter])
>>
//      hungry()
//    def on(("request", p)) =
//      sendFork(p :!: Xmitter) >>
//      thinking()
//    on(receive())

private PhilosopherState thinking() throws
InterruptedException {
    final Message msg = mbox.get();
    switch (msg.command) {
    case "rumble":
        System.out.println(name + " " + "hungry");
        for (final ChannelPutter<Message> p : missing) {
            requestFork(p);
        }
        return this::hungry;
    case "request":
        sendFork(msg.sender);
        return this::thinking;
    default:
        throw new RuntimeException("Illegal message: " + msg)
;
    }
}

//
//  def hungry() :: Bot =
//    def on(("fork", p) :: Message) =
//      missing.remove(p :!: Xmitter) >>
```

```
//      (
//        if missing.isEmpty() then
//          Println(name + " eating") >>
//          eating()
//        else hungry()
//      )
//    def on(("request", p)) =
//      if clean.contains(p :!: Xmitter) then
//        deferred.put(p :!: Xmitter) >>
//        hungry()
//      else
//        sendFork(p :!: Xmitter) >>
//        requestFork(p :!: Xmitter) >>
//        hungry()
//    on(receive())

private PhilosopherState hungry() throws InterruptedException
 {
    if (missing.isEmpty()) {
        System.out.println(name + " " + "eating");
        // System.err.println(name + "[");
        return this::eating;
    } else {
        final Message msg = mbox.get();
        switch (msg.command) {
        case "fork":
            missing.remove(msg.sender);
            return this::hungry;
        case "request":
            if (clean.contains(msg.sender)) {
                deferred.add(msg.sender);
            } else {
                sendFork(msg.sender);
                requestFork(msg.sender);
            }
            return this::hungry;
        default:
            throw new RuntimeException("Illegal message: " +
msg);
        }
    }
}

//
//  def eating() :: Bot =
//    clean.clear() >>
//    Rwait(Random(10)) >>
//    map(sendFork, deferred.getAll()) >>
//    digesting()

private PhilosopherState eating() {
    clean.clear();
    try {
        Thread.sleep((long) (Math.random() * 200));
    } catch (final InterruptedException e) {
        e.printStackTrace();
    }
    // System.err.println(name + "]");
    for (final ChannelPutter<Message> p : deferred) {
        sendFork(p);
    }
    deferred.clear();
    return this::digesting;
}

synchronized private PhilosopherState waiting() throws
InterruptedException {
    wait();
    return this::digesting;
}

public void run() {
    state = this::waiting;
    try {
        while (true) {
            try {
```

```
                state = state.call();
            } catch (final InterruptedException e) {
                // Just retry
            }
        }
    } catch (final Exception e) {
        //System.err.println("Philosopher " + name + " " +
myPutter + " exception:");
        e.printStackTrace();
        //System.err.println("State " + name + " " + myPutter
 + ":" + missing + " " + deferred + " " + clean);
    }
}

@Override
public ChannelPutter<Message> getPutter() throws
RemoteException {
    return myPutter;
}

@Override
public void addConnection(final ChannelPutter<Message> putter
) throws RemoteException {
    missing.add(putter);
}

@Override
public synchronized void start() throws RemoteException {
    System.out.println(name + " starting with putter " +
getPutter());
    System.out.println(name + " starting with missing " +
missing);

    notifyAll();
}

public static void main(final String[] args) throws Exception
 {
    final String name = args[0];

    final PhilosopherImpl server = new PhilosopherImpl(name);
    final Philosopher stub = (Philosopher)
UnicastRemoteObject.exportObject(server, 0);
    final Registry registry = LocateRegistry.getRegistry();
    registry.rebind(name, stub);

    server.run();
  }

}
```

Listing B.32. Java Dining Philosophers `PortableChannelPutter` class

```
package edu.utexas.orc.channel;

import java.io.Serializable;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.util.UUID;

public class PortableChannelPutter<T> implements ChannelPutter<T
    >, Serializable {
    private static final long serialVersionUID =
    4314633803229318677L;

    protected UUID uuid;

    public PortableChannelPutter(final UUID uuid) {
        this.uuid = uuid;
    }

    @SuppressWarnings("unchecked")
    @Override
    public void put(final T message) {
        try {
            ChannelReceiver<T> recv = null;
```

```
        while (recv == null) {
            try {
                recv = (ChannelReceiver<T>) Channel.registry.
lookup(getChannelName());
            } catch (final NotBoundException e1) {
                try {
                    Thread.sleep(1000);
                } catch (final InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
        }
        recv.put(message);
    } catch (final RemoteException e) {
        throw new Error(e);
    }
}

protected String getChannelName() {
    return "channel_" + uuid;
}

@Override
public String toString() {
    return "Putter [" + uuid + "]";
}

@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + (uuid == null ? 0 : uuid.
hashCode());
    return result;
}

@Override
    public boolean equals(final Object obj) {
        if (obj == null) {
            return false;
        }
        if (getClass() != obj.getClass()) {
            return false;
        }
        final PortableChannelPutter other = (
     PortableChannelPutter) obj;
        if (!uuid.equals(other.uuid)) {
            return false;
        }
        return true;
    }
}
```


B.4. Breadth-First Search

B.4.1. Orc Breadth-First Search

Listing B.33. Orc Breadth-First Search

```
{- bfs.orc

EXERCISE:

You are given a data type for binary trees with the
constructors <code>Tree(left, value, right)</code> and
<code>Leaf()</code>. Write a function which, given a tree,
returns a list of values in the tree ordered by
depth. I.e. the first element should be the root
value, followed by the root's children, followed by
its grandchildren, and so on.
```

```
SOLUTION:
--}

type Tree[A] = Node(Tree[A], A, Tree[A]) | Leaf()

def levels[A](Tree[A]) :: List[A]
def levels(tree) =
  def helper(List[Tree[A]]) :: List[A]
  def helper([]) = []
  def helper(Leaf():rest) = helper(rest)
  def helper(Node(l,v,r):rest) =
    v:helper(append(rest, [l, r]))
  helper([tree])

levels(Node(
  Node(
    Node(
      Leaf(),
      "F3",
      Leaf()),
    "B2",
    Node(
      Node(Leaf(), "L4", Leaf()),
      "H3",
      Node(Leaf(), "M4", Leaf()))),
  "A1",
  Node(
    Node(Node(Leaf(), "I4", Leaf()), "D3", Leaf()),
    "C2",
    Node(
      Node(Node(Leaf(), "N5", Leaf()), "J4", Leaf()),
      "E3",
      Node(Leaf(), "K4", Leaf())))))

{-
OUTPUT:
[1, 2, 2, 3, 3, 3, 3]
-}
```


B.4.2. Java Breadth-First Search

Listing B.34. Java Breadth-First Search `DistBFSServer` interface

```
package edu.utexas.orc.tree;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface DistBFSServer extends Remote {
    void searchWithState(SearchState state) throws
     RemoteException;

    void addTree(Tree tree) throws RemoteException;

    void shutdown() throws RemoteException;
}
```

Listing B.35. Java Breadth-First Search `DistBFSServerImpl` class

```
package edu.utexas.orc.tree;

import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.Map;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;

/**
 * Examples of various kinds of tree traversals and searches.
```

```
 *
 * @author Dave Matuszek
 * @author Arthur Peters
 * @version 12/08/2016
 *
 */
public class DistBFSServerImpl implements DistBFSServer {
    private static Registry registry;
    private static SearchState SENTINAL_STATE = new SearchState(
     null, null, null);

    public static void main(final String[] args) {
        final String name = args[0];

        //System.out.println("Starting server " + name);
        try {
            final DistBFSServerImpl server = new
     DistBFSServerImpl(name);
            final DistBFSServer stub = (DistBFSServer)
     UnicastRemoteObject.exportObject(server, 0);
            registry = LocateRegistry.getRegistry();
            registry.rebind(name, stub);
            //System.out.println("Server " + name + " registered
     .");

            server.run();
        } catch (final Exception e) {
            //System.err.println("Server exception:");
            e.printStackTrace();
        }
    }

    private final BlockingQueue<SearchState> commandQueue = new
     LinkedBlockingQueue<>();
    private final String name;
    private final Map<Integer, Tree> treeMap = new HashMap<>();

    public DistBFSServerImpl(final String name) {
        this.name = name;
    }

    public void run() {
        try {
            while (true) {
                final SearchState state = commandQueue.take();
                if (state == SENTINAL_STATE) {
                    break;
                }

                //System.out.println("Working from " + state + "
    as " + name);
                final LinkedList<TreeRef> queue = state.queue;
                final ArrayList<String> result = state.result;
                while (!queue.isEmpty()) {
                    final TreeRef noderef = queue.peek();
                    if (noderef.location.equals(name)) {
                        queue.poll();
                        final Tree node = treeMap.get(noderef.id)
    ;
                        //System.out.println(node.value);
                        result.add(node.value);
                        queue.addAll(node.children);
                    } else {
                        final DistBFSServer next = (DistBFSServer
    ) registry.lookup(noderef.location);
                        next.searchWithState(new SearchState(
    queue, result, state.requester));
                        break;
                    }
                }

                if (queue.isEmpty()) {
                    state.requester.complete(result);
                }
            }
        } catch (final InterruptedException e) {
```

```
            // Interrupted. Just give up.
        } catch (final Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }

    @Override
    public void addTree(final Tree tree) {
        if (!tree.location.equals(name)) {
            throw new IllegalArgumentException("Name must be " +
     name + ": " + tree);
        }
        treeMap.put(tree.id, tree);
        for (final TreeRef t : tree.children) {
            if (t instanceof Tree) {
                addTree((Tree) t);
            }
        }
    }

    @Override
    public void searchWithState(final SearchState state) {
        commandQueue.add(state);
    }

    @Override
    synchronized public void shutdown() {
        try {
            registry.unbind(name);
            UnicastRemoteObject.unexportObject(this, true);
        } catch (RemoteException | NotBoundException e) {
            throw new RuntimeException(e);
        }
        commandQueue.add(SENTINAL_STATE);
    }
}
```

Listing B.36. Java Breadth-First Search `Requester` interface

```
package edu.utexas.orc.tree;

import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.ArrayList;

public interface Requester extends Remote {
    void complete(ArrayList<String> result) throws
     RemoteException;
}
```

Listing B.37. Java Breadth-First Search `SearchState` class

```
package edu.utexas.orc.tree;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.LinkedList;

public class SearchState implements Serializable {
    private static final long serialVersionUID =
     -7034389379306468787L;

    public final LinkedList<TreeRef> queue;
    public final ArrayList<String> result;
    public final Requester requester;

    public SearchState(final LinkedList<TreeRef> queue, final
     ArrayList<String> result, final Requester requester) {
        this.queue = queue;
        this.result = result;
        this.requester = requester;
    }

    //  @Override
    //  public String toString() {
    //          return "SearchState [queue=" + queue + ", result
```

```
    =" + result + "]";
    //  }
}

/**
 * @author amp
 */
class TreeRef implements Serializable {
    private static final long serialVersionUID =
    4476907093676031702L;

    public final String location;
    public final int id;

    public TreeRef(final String location, final int id) {
        this.location = location;
        this.id = id;
    }

    //  @Override
    //  public String toString() {
    //          return "TreeRef [location=" + location + ", id="
    + id + "]";
    //  }
}
```

Listing B.38. Java Breadth-First Search Tree class

```
package edu.utexas.orc.tree;

import java.util.Vector;

/**
 * Absolutely minimal implementation of a tree data type,
    suitable only
 * for creating an example tree. All data is unprotected.
 *
 * @author Dave Matuszek
 * @version 3/31/2003
 */
public class Tree extends TreeRef {
    static int nextID = 0;
    String value;
    Vector<TreeRef> children;

    /** Constructor to create a node containing the given value.
     */
    Tree(final String value, final String location) {
        super(location, nextID++);
        this.value = value;
        children = new Vector<>();
    }

    /** Add a node containing the given value as a child of this
     node. */
    Tree add(final String value) {
        final Tree tree = new Tree(value, location);
        add(tree);
        return tree;
    }

    Tree add(final String value, final String location) {
        final Tree tree = new Tree(value, location);
        add(tree);
        return tree;
    }

    /** Adds the given tree as a child of this node. */
    TreeRef add(final TreeRef tree) {
        children.add(tree);
        return tree;
    }

    TreeRef getRef() {
        return new TreeRef(location, id);
    }
```

```
    @Override
    public String toString() {
        return "<" + value + ", " + location + ">";
    }
}
```

Listing B.39. Java Breadth-First Search `TreeTraversalsDist` class

```
package edu.utexas.orc.tree;

import java.io.File;
import java.io.IOException;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import java.util.ArrayList;
import java.util.LinkedList;

/**
 * Examples of various kinds of tree traversals and searches.
 *
 * @author Dave Matuszek
 * @version 3/31/2003
 */
public class TreeTraversalsDist {
    static int nProcs = 2;
    private static Registry registry;
    private static Process[] procs = new Process[nProcs];
    private static DistBFSServer[] servers = new DistBFSServer[
    nProcs];

    static Process startProcess(final Class<?> cls, final String
    ... arguments) throws IOException {
        final String[] args = new String[arguments.length + 2];
        args[0] = "java";
        args[1] = cls.getName();
        System.arraycopy(arguments, 0, args, 2, arguments.length)
;
        //System.out.println("Starting process with args list: "
+ Arrays.asList(args));
        final ProcessBuilder pb = new ProcessBuilder(args);
        pb.directory(new File(System.getProperty("user.dir") + "/
bin"));
        pb.inheritIO();
        return pb.start();
    }

    public static void main(final String[] args) throws
    IOException {
        try {
            registry = LocateRegistry.createRegistry(1099);
        } catch (final Exception e) {
            System.err.println("Server exception:");
            e.printStackTrace();
        }

        try {
            for (int i = 0; i < nProcs; i++) {
                final String name = "proc" + i;
                procs[i] = startProcess(DistBFSServerImpl.class,
name);
                while (true) {
                    try {
                        //System.out.print("."); System.out.flush
();
                        servers[i] = (DistBFSServer) registry.
lookup(name);
                        break;
                    } catch (final NotBoundException e) {
                        // Give the server 1 second more to
register.
                        try {
                            Thread.sleep(1000);
```

```
                    } catch (final InterruptedException e1) {
                    }
                }
            }
        }

        final Tree tree = createTree();

        System.out.println(bfs(tree));
    } finally {
        for (final DistBFSServer server : servers) {
            if (server != null) {
                server.shutdown();
            }
        }
    }
}

/** Creates and returns a tree for use in testing the other
 methods.
 * @throws RemoteException */
static Tree createTree() throws RemoteException {
    final Tree root = new Tree("A1", "proc0");
    final Tree left = root.add("B2");
    left.add("F3");
    Tree temp = left.add("H3");
    temp.add("L4");
    temp.add("M4");

    final Tree right = new Tree("C2", "proc1");
    temp = right.add("D3");
    temp.add("I4");
    temp = right.add("E3");
    final Tree temp2 = temp.add("J4");
    temp2.add("N5");
    temp.add("K4");

    servers[1].addTree(right);
    root.add(right.getRef());

    servers[0].addTree(root);

    return root;
}

/**
 * Performs a breadth-first search, starting from the given
node, for any
 * node that satisfies the condition isGoal(node).
 *
 * @param root
 *            The root of the tree or subtree to be searched.
 * @return True if a goal node could be found.
 * @throws RemoteException
 */
static ArrayList<String> bfs(final Tree root) throws
RemoteException {
    final LinkedList<TreeRef> queue = new LinkedList<>();
    queue.add(root);

    final RequesterImpl req = new RequesterImpl();
    final Requester stub = (Requester) UnicastRemoteObject.
exportObject(req, 0);

    servers[0].searchWithState(new SearchState(queue, new
ArrayList<String>(), stub));
    try {
        synchronized (req) {
            req.wait();
        }
    } catch (final InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
```

```
        UnicastRemoteObject.unexportObject(req, true);

        return req.result;
    }

    private static final class RequesterImpl implements Requester
     {
        public ArrayList<String> result = null;

        @Override
        synchronized public void complete(final ArrayList<String>
     result) {
            this.result = result;
            notify();
        }
    }
}
```


B.5. Depth-First Search

B.5.1. Orc Depth-First Search

Listing B.40. Orc Depth-First Search

```
{- dfs.orc -- Orc program: Perform a depth-first search of a
    graph

This is the depth first search algorithm for a connected
    undirected
graph. The graph structure is embedded in an immutable array
    conn; conn(i), 0<= i<
N, is the list of neighbors of i. The output of the algorithm is
    the
depth-first search tree, which is represented by array
parent: parent(i) is N if i is the root (node 0), and 0<= j< N,
    otherwise.

Since the graph is connected, conn(i) is never []. array parent
    is
mutable. An invariant of the algorithm is: parent(i) is negative
    if i
is not the root(node 0) nor has a parent, N if i is the root (
    node 0),
and j, 0<= j< N, if i has parent j.

Edge (i,j) is a tree edge if i is j's parent, i.e., parent(j) =
    i;
(i,j) is backward if j is an ancestor, possibly parent, of i,
i.e., parent(j) =/ i
-}

val N = 6
val conn = Array[List[Integer]](N)
val parent = fillArray(Array[Integer](N), lambda(_ :: Integer) =
    -1)

def dfs(Integer) :: Signal
def dfs(i) =
  def scan(List[Integer]) :: Signal
  def scan([]) = signal
  def scan(y:ys) =
    if parent(y)? <: 0 then
      ( parent(y) := i >> dfs(y) >> scan(ys) )
    else
      scan(ys)
  scan(conn(i)?)

-- Goal expression. First specify the graph structure.
#
( conn(0) := [1,2,3,4]
, conn(1) := [0,5]
, conn(2) := [0,4]
```

```
, conn(3) := [0,5]
, conn(4) := [0,2]
, conn(5) := [1,3]
)
>>
parent(0) := N >> dfs(0) >> upto(N) >i> (i,parent(i)?)

-- Note that this test may occasionally spontaneously fail due to
     nondeterminism,
-- since some pairs might occur in a different order.
-- This is just a weakness of the test harness.

{-
OUTPUT:PERMUTABLE:
(0, 6)
(1, 0)
(2, 0)
(3, 5)
(4, 2)
(5, 1)
-}
```

B.5.2. Java Depth-First Search

Listing B.41. Java Depth-First Search `DepthFirstSearch` class

```
/********
 *  Compilation:  javac DepthFirstSearch.java
 *  Execution:    java DepthFirstSearch filename.txt s
 *  Dependencies: Graph.java StdOut.java
 *  Data files:   http://algs4.cs.princeton.edu/41graph/tinyG.txt
 *                http://algs4.cs.princeton.edu/41graph/mediumG.
    txt
 *
 *  Run depth first search on an undirected graph.
 *  Runs in O(E + V) time.
 *
 *  % java DepthFirstSearch tinyG.txt 0
 *  0 1 2 3 4 5 6
 *  NOT connected
 *
 *  % java DepthFirstSearch tinyG.txt 9
 *  9 10 11 12
 *  NOT connected
 *
 ********/

package edu.utexas.orc.graph;

import java.io.File;
import java.io.IOException;
import java.rmi.NotBoundException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

/**
 *  The {@code DepthFirstSearch} class represents a data type for
 *  determining the vertices connected to a given source vertex <
    em>s</em>
 *  in an undirected graph. For versions that find the paths, see
 *  {@link DepthFirstPaths} and {@link BreadthFirstPaths}.
 *  <p>
 *  This implementation uses depth-first search.
 *  The constructor takes time proportional to <em>V</em> + <em>E
    </em>
 *  (in the worst case),
 *  where <em>V</em> is the number of vertices and <em>E</em> is
    the number of edges.
 *  It uses extra space (not including the graph) proportional to
     <em>V</em>.
 *  <p>
 *  For additional documentation, see <a href="http://algs4.cs.
    princeton.edu/41graph">Section 4.1</a>
```

```
 *  of <i>Algorithms, 4th Edition</i> by Robert Sedgewick and
    Kevin Wayne.
 *
 *  @author Robert Sedgewick
 *  @author Kevin Wayne
 */
public class DepthFirstSearch {
    static Process startProcess(final Class<?> cls, final String
    ... arguments) throws IOException {
        final String[] args = new String[arguments.length + 2];
        args[0] = "java";
        args[1] = cls.getName();
        System.arraycopy(arguments, 0, args, 2, arguments.length)
    ;
        //System.out.println("Starting process with args list: "
    + Arrays.asList(args));
        final ProcessBuilder pb = new ProcessBuilder(args);
        pb.directory(new File(System.getProperty("user.dir") + "/
    bin"));
        pb.inheritIO();
        return pb.start();
    }

    public static void main(final String[] args) throws
    IOException {
        Registry registry;
        registry = LocateRegistry.createRegistry(1099);

        DepthFirstSearchServer server = null;
        try {
            startProcess(DepthFirstSearchServerImpl.class, "
    server");
            while (true) {
                try {
                    //System.out.print("."); System.out.flush();
                    server = (DepthFirstSearchServer) registry.
    lookup("server");
                    break;
                } catch (final NotBoundException e) {
                    // Give the server 1 second more to register.
                    try {
                        Thread.sleep(1000);
                    } catch (final InterruptedException e1) {
                    }
                }
            }

            final int s = Integer.parseInt(args[1]);
            final int[] parent = server.dfs(s);
            for (int v = 0; v < parent.length; v++) {
                System.out.println(v + " " + parent[v]);
            }
        } finally {
            if (server != null) {
                server.shutdown();
            }
        }
    }
}
```

Listing B.42. Java Depth-First Search `DepthFirstSearchServer` interface

```
package edu.utexas.orc.graph;

import java.rmi.Remote;
import java.rmi.RemoteException;

/**
 *
 *  @author Robert Sedgewick
 *  @author Kevin Wayne
 *  @author Arthur Peters
 */
public interface DepthFirstSearchServer extends Remote {
    // depth first search from v
    public int[] dfs(int v) throws RemoteException;
```

```
    public void shutdown() throws RemoteException;
}
```

Listing B.43. Java Depth-First Search `DepthFirstSearchServerImpl` class

```
/********
 *  Compilation:  javac DepthFirstSearch.java
 *  Execution:    java DepthFirstSearch filename.txt s
 *  Dependencies: Graph.java StdOut.java
 *  Data files:   http://algs4.cs.princeton.edu/41graph/tinyG.txt
 *                http://algs4.cs.princeton.edu/41graph/mediumG.
    txt
 *
 *  Run depth first search on an undirected graph.
 *  Runs in O(E + V) time.
 *
 *  % java DepthFirstSearch tinyG.txt 0
 *  0 1 2 3 4 5 6
 *  NOT connected
 *
 *  % java DepthFirstSearch tinyG.txt 9
 *  9 10 11 12
 *  NOT connected
 *
 ********/

package edu.utexas.orc.graph;

import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import java.util.Arrays;

/**
 *  The {@code DepthFirstSearch} class represents a data type for
 *  determining the vertices connected to a given source vertex <
    em>s</em>
 *  in an undirected graph. For versions that find the paths, see
 *  {@link DepthFirstPaths} and {@link BreadthFirstPaths}.
 *  <p>
 *  This implementation uses depth-first search.
 *  The constructor takes time proportional to <em>V</em> + <em>E
    </em>
 *  (in the worst case),
 *  where <em>V</em> is the number of vertices and <em>E</em> is
    the number of edges.
 *  It uses extra space (not including the graph) proportional to
     <em>V</em>.
 *  <p>
 *  For additional documentation, see <a href="http://algs4.cs.
    princeton.edu/41graph">Section 4.1</a>
 *  of <i>Algorithms, 4th Edition</i> by Robert Sedgewick and
    Kevin Wayne.
 *
 *  @author Robert Sedgewick
 *  @author Kevin Wayne
 */
public class DepthFirstSearchServerImpl implements
    DepthFirstSearchServer {
    private static Registry registry;
    public int[] parent; // marked[v] = is there an s-v path?
    private final Graph G;
    private static String name;

    public DepthFirstSearchServerImpl(final Graph G) {
        this.G = G;
    }

    /**
     * Computes the vertices in graph {@code G} that are
     * connected to the source vertex {@code s}.
     * @param G the graph
     * @param s the source vertex
```

```
     * @throws IllegalArgumentException unless {@code 0 <= s < V}
     */
    @Override
    public synchronized int[] dfs(final int s) {
        parent = new int[G.V()];
        Arrays.fill(parent, -1);
        parent[s] = G.V();
        dfsInner(G, s);
        return parent;
    }

    private void dfsInner(final Graph G, final int v) {
        for (final int w : G.adj(v)) {
            if (parent[w] < 0) {
                parent[w] = v;
                dfsInner(G, w);
            }
        }
    }

    @Override
    public void shutdown() {
        try {
            registry.unbind(name);
            UnicastRemoteObject.unexportObject(this, true);
        } catch (RemoteException | NotBoundException e) {
            throw new RuntimeException(e);
        }
    }

    /**
     * Unit tests the {@code DepthFirstSearch} data type.
     *
     * @param args the command-line arguments
     */
    public static void main(final String[] args) {
        name = args[0];
        final Graph G = new Graph(6);

        G.addEdge(0, 1);
        G.addEdge(0, 2);
        G.addEdge(0, 3);
        G.addEdge(0, 4);
        G.addEdge(1, 0);
        G.addEdge(1, 5);
        G.addEdge(2, 0);
        G.addEdge(2, 4);
        G.addEdge(3, 0);
        G.addEdge(3, 5);
        G.addEdge(4, 0);
        G.addEdge(4, 2);
        G.addEdge(5, 1);
        G.addEdge(5, 3);

        try {
            final DepthFirstSearchServerImpl server = new
    DepthFirstSearchServerImpl(G);
            final DepthFirstSearchServer stub = (
    DepthFirstSearchServer) UnicastRemoteObject.exportObject(
    server, 0);
            registry = LocateRegistry.getRegistry();
            registry.rebind(name, stub);
        } catch (final Exception e) {
            //System.err.println("Server exception:");
            e.printStackTrace();
        }
    }

}
```

Listing B.44. Java Depth-First Search `Graph` class

```
/*******
 *  Compilation:  javac Graph.java
 *  Execution:    java Graph input.txt
```

```
 *  Dependencies: Bag.java Stack.java In.java StdOut.java
 *  Data files:   http://algs4.cs.princeton.edu/41graph/tinyG.txt
 *                http://algs4.cs.princeton.edu/41graph/mediumG.
    txt
 *                http://algs4.cs.princeton.edu/41graph/largeG.
    txt
 *
 *  A graph, implemented using an array of sets.
 *  Parallel edges and self-loops allowed.
 *
 *  % java Graph tinyG.txt
 *  13 vertices, 13 edges
 *  0: 6 2 1 5
 *  1: 0
 *  2: 0
 *  3: 5 4
 *  4: 5 6 3
 *  5: 3 4 0
 *  6: 0 4
 *  7: 8
 *  8: 7
 *  9: 11 10 12
 *  10: 9
 *  11: 9 12
 *  12: 11 9
 *
 *  % java Graph mediumG.txt
 *  250 vertices, 1273 edges
 *  0: 225 222 211 209 204 202 191 176 163 160 149 114 97 80 68
    59 58 49 44 24 15
 *  1: 220 203 200 194 189 164 150 130 107 72
 *  2: 141 110 108 86 79 51 42 18 14
 *  ...
 *
 ********/

package edu.utexas.orc.graph;

import java.io.Serializable;
import java.util.ArrayList;

/**
 *  The {@code Graph} class represents an undirected graph of
    vertices
 *  named 0 through <em>V</em> - 1.
 *  It supports the following two primary operations: add an edge
     to the graph,
 *  iterate over all of the vertices adjacent to a vertex. It
    also provides
 *  methods for returning the number of vertices <em>V</em> and
    the number
 *  of edges <em>E</em>. Parallel edges and self-loops are
    permitted.
 *  By convention, a self-loop <em>v</em>-<em>v</em> appears in
    the
 *  adjacency list of <em>v</em> twice and contributes two to the
     degree
 *  of <em>v</em>.
 *  <p>
 *  This implementation uses an adjacency-lists representation,
    which
 *  is a vertex-indexed array of {@link Bag} objects.
 *  All operations take constant time (in the worst case) except
 *  iterating over the vertices adjacent to a given vertex, which
     takes
 *  time proportional to the number of such vertices.
 *  <p>
 *  For additional documentation, see <a href="http://algs4.cs.
    princeton.edu/41graph">Section 4.1</a>
 *  of <i>Algorithms, 4th Edition</i> by Robert Sedgewick and
    Kevin Wayne.
 *
 *  @author Robert Sedgewick
 *  @author Kevin Wayne
 */
public class Graph implements Serializable {
```

```
private static final long serialVersionUID =
-6156113248147121955L;

private final int V;
private final ArrayList<Integer>[] adj;

/**
 * Initializes an empty graph with {@code V} vertices and 0
edges.
 * param V the number of vertices
 *
 * @param  V number of vertices
 * @throws IllegalArgumentException if {@code V < 0}
 */
public Graph(final int V) {
    if (V < 0) {
        throw new IllegalArgumentException("Number of
vertices must be nonnegative");
    }
    this.V = V;
    adj = new ArrayList[V];
    for (int v = 0; v < V; v++) {
        adj[v] = new ArrayList<>();
    }
}

/**
 * Returns the number of vertices in this graph.
 *
 * @return the number of vertices in this graph
 */
public int V() {
    return V;
}

/**
 * Adds the undirected edge v-w to this graph.
 *
 * @param  v one vertex in the edge
 * @param  w the other vertex in the edge
 * @throws IllegalArgumentException unless both {@code 0 <= v
 < V} and {@code 0 <= w < V}
 */
public void addEdge(final int v, final int w) {
    adj[v].add(w);
    adj[w].add(v);
}

/**
 * Returns the vertices adjacent to vertex {@code v}.
 *
 * @param  v the vertex
 * @return the vertices adjacent to vertex {@code v}, as an
iterable
 * @throws IllegalArgumentException unless {@code 0 <= v < V}
 */
public Iterable<Integer> adj(final int v) {
    return adj[v];
}
}
```

B.6. Sudoku

B.6.1. Orc Sudoku

Listing B.45. Orc Sudoku

```
{- Sudoku.orc -- Orc program Sudoku
 -}

type Board = List[ List[ Option[ Integer ] ] ]
```

```
def replace(l:ls, v, 0) = v : ls
def replace(l:ls, v, n) = l : replace(ls, v, n-1)

def update(b, v, x, y) = replace(b, replace(index(b, y), v, x), y
    )

def remove(_, []) = []
def remove(e, x:xs) = if e = x then xs else x : remove(e, xs)

def getCell(x, y, b) =
        index(index(b, y), x)

def getBlock(x, y, b) =
        val top = (y/3) * 3
        val left = (x/3) * 3
        upto(3) >x'> upto(3) >y'>
        getCell(left + x', top + y', b)

def isValid(v, x, y, b) =
        (upto(10) >x'> getCell(x', y, b) = Some(v) |
         upto(10) >y'> getCell(x, y', b) = Some(v) |
         getBlock(x, y, b) >v> v         = Some(v)) >true> false
    ; true

def solveCell(x, y, b) =
        val c = getCell(x, y, b) #
        (c >None> for(1, 10) >v> Ift(isValid(v, x, y, b)) >>
    update(b, Some(v), x, y)) |
        (c >Some(_)> b)

def ndfoldl[A,B](lambda (B, A) :: B, B, List[A]) :: B
def ndfoldl(f,z,[]) = z
def ndfoldl(f,z,x:xs) = f(z,x) >z'> ndfoldl(f,z',xs)

def solveRow(y, b) = ndfoldl(lambda(acc, x) = solveCell(x, y, acc
    ), b, range(0, 9))
def solveAlgo0(b) = ndfoldl(lambda(acc, y) = solveRow(y, acc), b,
     range(0, 9))
```

```
def load(s) =
  def parseCell(c) =
          Ift(c = " ") >> None()
        | (val v = Read[Integer](c)
          if v :> 0 && v <: 10 then Some(v) else stop)
  def parseLine(l) = map(parseCell, characters(l))
  map(parseLine, lines(s))

def printRow([]) = ""
def printRow(Some(v):cs) = " " + Write(v) + printRow(cs)
def printRow(None():cs) = " _" + printRow(cs)

def printBoard([]) = ""
def printBoard(r : rs) = printRow(r) + "\n" + printBoard(rs)

-- This does not work
def printBoard2(b) = unlines(map(printRow, b))

val ex0 = "4 2  9 83\n  84 2   \n3968   7 \n  1   75 \n9   6   2\
    n 57   8  \n 2   6395\n   7 56  \n86 9  4 7"
val ex1 = "  7  5  2\n    31 8 \n 3    7  \n  8 2   3\n 73   82 \
    n5   6 9  \n  5    6 \n 1 59    \n8  4  2  "
val ex2 = " 437  9 8\n  5 3    \n 1    3  \n6   27   \n4 7   1 3\
    n   54   9\n  2    3 \n    5 4  \n5 4  126 "
val ex3 = "2437 6958\n98523 7 6\n71689 342\n6 912758 \n4579 8123\
    n8  543  9\n1 24 9 3 \n37  5 4 1\n5 4381267"

val b = load(ex3)

Println(printBoard(b)) >>
Println("==================") >>
solveAlgo0(b) >x>
Println(printBoard(x))

{-
BENCHMARK
-}
```

B.6.2. Java Sudoku

Listing B.46. Java Sudoku Sudoku class

```
package edu.utexas.orc;

import java.io.File;
import java.io.IOException;
import java.rmi.NotBoundException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

/**
 * The <code>Sudoku</code> class povides a static <code>main</
    code> method
 * allowing it to be called from the command line to print the
    solution to a
 * specified Sudoku problem.
 *
 * <p>
 * The following is an example of a Sudoku problem:
 *
 * <pre>
 *  ----------------------
 * |   8   | 4   2 |   6   |
 * |   3 4 |       | 9 1   |
 * | 9 6   |       |   8 4 |
 *  ----------------------
 * |       | 2 1 6 |       |
 * |       |       |       |
 * |       | 3 5 7 |       |
 *  ----------------------
 * | 8 4   |       |   7 5 |
 * |   2 6 |       | 1 3   |
 * |   9   | 7   1 |   4   |
 *  ----------------------
 * </pre>
 *
 * The goal is to fill in the missing numbers so that every row,
    column and box
 * contains each of the numbers <code>1-9</code>. Here is the
    solution to the
 * problem above:
 *
 * <pre>
 *  ----------------------
 * | 1 8 7 | 4 9 2 | 5 6 3 |
 * | 5 3 4 | 6 7 8 | 9 1 2 |
 * | 9 6 2 | 1 3 5 | 7 8 4 |
 *  ----------------------
 * | 4 5 8 | 2 1 6 | 3 9 7 |
 * | 2 7 3 | 8 4 9 | 6 5 1 |
 * | 6 1 9 | 3 5 7 | 4 2 8 |
 *  ----------------------
 * | 8 4 1 | 9 6 3 | 2 7 5 |
 * | 7 2 6 | 5 8 4 | 1 3 9 |
 * | 3 9 5 | 7 2 1 | 8 4 6 |
 *  ----------------------
 * </pre>
 *
 * Note that the first row <code>187492563</code> contains each
    number exactly
 * once, as does the first column <code>159426873</code>, the
    upper-left box
 * <code>187534962</code>, and every other row, column and box.
 *
 * <p>
 * The {@link #main(String[])} method encodes a problem as an
    array of strings,
 * with one string encoding each constraint in the problem in row
    -column-value
 * format. Here is the problem again with the indices indicated:
 *
 * <pre>
 *     0 1 2   3 4 5   6 7 8
 *    ----------------------
```

```
 * 0 |   8   | 4   2 |   6   |
 * 1 |   3 4 |       | 9 1   |
 * 2 | 9 6   |       |   8 4 |
 *    ----------------------
 * 3 |       | 2 1 6 |       |
 * 4 |       |       |       |
 * 5 |       | 3 5 7 |       |
 *   ----------------------
 * 6 | 8 4   |       |   7 5 |
 * 7 |   2 6 |       | 1 3   |
 * 8 |   9   | 7   1 |   4   |
 *    ----------------------
 * </pre>
 *
 * The <code>8</code> in the upper left box of the puzzle is encoded as
 * <code>018</code> (<code>0</code> for the row, <code>1</code> for the column,
 * and <code>8</code> for the value). The <code>4</code> in the lower right box
 * is encoded as <code>874</code>.
 *
 * <p>
 * The full command-line invocation for the above puzzle is:
 *
 * <pre>
 * % java -cp . Sudoku 018 034 052 076 \
 *                     113 124 169 171 \
 *                     209 216 278 284 \
 *                     332 341 356     \
 *                     533 545 557     \
 *                     608 614 677 685 \
 *                     712 726 761 773 \
 *                     819 837 851 874 \
 * </pre>
 *
 * <p>
 * See <a href="http://en.wikipedia.org/wiki/Sudoku">Wikipedia: Sudoku</a> for
 * more information on Sudoku.
 *
 * <p>
 * The algorithm employed is similar to the standard backtracking <a
 * href="http://en.wikipedia.org/wiki/Eight_queens_puzzle">eight queens
 * algorithm</a>.
 *
 * @version 1.0
 * @author <a href="http://www.colloquial.com/carp">Bob Carpenter</a>
 */
public class Sudoku {
    static Process startProcess(final Class<?> cls, final String
    ... arguments) throws IOException {
        final String[] args = new String[arguments.length + 2];
        args[0] = "java";
        args[1] = cls.getName();
        System.arraycopy(arguments, 0, args, 2, arguments.length)
    ;
        // System.out.println("Starting process with args list: "
     +
        // Arrays.asList(args));
        final ProcessBuilder pb = new ProcessBuilder(args);
        pb.directory(new File(System.getProperty("user.dir") + "/
    bin"));
        pb.inheritIO();
        return pb.start();
    }

    /**
     * Print the specified Sudoku problem and its solution. The
    problem is
     * encoded as specified in the class documentation above.
     *
```

```
     * @param args
     *            The command-line arguments encoding the problem
    .
     * @throws IOException
     */
    public static void main(final String[] args) throws
    IOException {
        Registry registry;
        registry = LocateRegistry.createRegistry(1099);

        SudokuSolver server = null;
        try {
            startProcess(SudokuSolverImpl.class, "server");
            while (true) {
                try {
                    // System.out.print("."); System.out.flush();
                    server = (SudokuSolver) registry.lookup("
    server");
                    break;
                } catch (final NotBoundException e) {
                    // Give the server 1 second more to register.
                    try {
                        Thread.sleep(1000);
                    } catch (final InterruptedException e1) {
                    }
                }
            }
            int[][] matrix = parseProblem(args);
            writeMatrix(matrix);
            matrix = server.solve(matrix);
            if (matrix != null) { // solves in place
                writeMatrix(matrix);
            } else {
                System.out.println("NONE");
            }
        } finally {
            if (server != null) {
                server.shutdown();
            }
        }
    }

    static int[][] parseProblem(final String[] args) {
        final int[][] problem = new int[9][9]; // default 0 vals
        for (final String arg : args) {
            final int i = Integer.parseInt(arg.substring(0, 1));
            final int j = Integer.parseInt(arg.substring(1, 2));
            final int val = Integer.parseInt(arg.substring(2, 3))
    ;
            problem[i][j] = val;
        }
        return problem;
    }

    static void writeMatrix(final int[][] solution) {
        for (int i = 0; i < 9; ++i) {
            if (i % 3 == 0) {
                System.out.println(" ----------------------");
            }
            for (int j = 0; j < 9; ++j) {
                if (j % 3 == 0) {
                    System.out.print("| ");
                }
                System.out.print(solution[i][j] == 0 ? " " :
     Integer.toString(solution[i][j]));

                System.out.print(' ');
            }
            System.out.println("|");
        }
        System.out.println(" ----------------------");
    }

    /*
    static void writeMatrixForOrc(int[][] solution) {
    for (int i = 0; i < 9; ++i) {
```

```
        // if (i % 3 == 0)
        // System.out.println(" ----------------------");
        for (int j = 0; j < 9; ++j) {
                // if (j % 3 == 0)
                // System.out.print("| ");
                System.out.print(solution[i][j] == 0 ? " " :
    Integer
                                .toString(solution[i][j]));

                // System.out.print(' ');
        }
        // System.out.println("|");
        System.out.print("\\n");
    }
    // System.out.println(" ----------------------");
    }
    */

}
```

Listing B.47. Java Sudoku `SudokuSolver` interface

```
package edu.utexas.orc;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface SudokuSolver extends Remote {

    public abstract void shutdown() throws RemoteException;

    public abstract int[][] solve(int[][] cells) throws
     RemoteException;

}
```

Listing B.48. Java Sudoku `SudokuSolverImpl` class

```
package edu.utexas.orc;

import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

/**
 * The <code>Sudoku</code> class povides a static <code>main</
    code> method
 * allowing it to be called from the command line to print the
    solution to a
 * specified Sudoku problem.
 *
 * <p>
 * The following is an example of a Sudoku problem:
 *
 * <pre>
 * ----------------------
 * |   8   | 4   2 |   6   |
 * |   3 4 |       | 9 1   |
 * | 9 6   |       |   8 4 |
 *  ----------------------
 * |       | 2 1 6 |       |
 * |       |       |       |
 * |       | 3 5 7 |       |
 *  ----------------------
 * | 8 4   |       |   7 5 |
 * |   2 6 |       | 1 3   |
 * |   9   | 7   1 |   4   |
 *  ----------------------
 * </pre>
 *
 * The goal is to fill in the missing numbers so that every row,
    column and box
 * contains each of the numbers <code>1-9</code>. Here is the
    solution to the
```

```
* problem above:
*
* <pre>
*  ----------------------
* | 1 8 7 | 4 9 2 | 5 6 3 |
* | 5 3 4 | 6 7 8 | 9 1 2 |
* | 9 6 2 | 1 3 5 | 7 8 4 |
*  ----------------------
* | 4 5 8 | 2 1 6 | 3 9 7 |
* | 2 7 3 | 8 4 9 | 6 5 1 |
* | 6 1 9 | 3 5 7 | 4 2 8 |
*  ----------------------
* | 8 4 1 | 9 6 3 | 2 7 5 |
* | 7 2 6 | 5 8 4 | 1 3 9 |
* | 3 9 5 | 7 2 1 | 8 4 6 |
*  ----------------------
* </pre>
*
* Note that the first row <code>187492563</code> contains each
   number exactly
* once, as does the first column <code>159426873</code>, the
   upper-left box
* <code>187534962</code>, and every other row, column and box.
*
* <p>
* The {@link #main(String[])} method encodes a problem as an
   array of strings,
* with one string encoding each constraint in the problem in row
   -column-value
* format. Here is the problem again with the indices indicated:
*
* <pre>
*     0 1 2   3 4 5   6 7 8
*    ----------------------
* 0 |   8   | 4   2 |   6   |
* 1 |   3 4 |       | 9 1   |
* 2 | 9 6   |       |   8 4 |
*    ----------------------
* 3 |       | 2 1 6 |       |
* 4 |       |       |       |
* 5 |       | 3 5 7 |       |
*   ----------------------
* 6 | 8 4   |       |   7 5 |
* 7 |   2 6 |       | 1 3   |
* 8 |   9   | 7   1 |   4   |
*    ----------------------
* </pre>
*
* The <code>8</code> in the upper left box of the puzzle is
   encoded as
* <code>018</code> (<code>0</code> for the row, <code>1</code>
   for the column,
* and <code>8</code> for the value). The <code>4</code> in the
   lower right box
* is encoded as <code>874</code>.
*
* <p>
* The full command-line invocation for the above puzzle is:
*
* <pre>
* % java -cp . Sudoku 018 034 052 076 \
*                     113 124 169 171 \
*                     209 216 278 284 \
*                     332 341 356     \
*                     533 545 557     \
*                     608 614 677 685 \
*                     712 726 761 773 \
*                     819 837 851 874 \
* </pre>
*
* <p>
* See <a href="http://en.wikipedia.org/wiki/Sudoku">Wikipedia:
   Sudoku</a> for
* more information on Sudoku.
*
* <p>
```

```
 * The algorithm employed is similar to the standard backtracking
     <a
 * href="http://en.wikipedia.org/wiki/Eight_queens_puzzle">eight
    queens
 * algorithm</a>.
 *
 * @version 1.0
 * @author <a href="http://www.colloquial.com/carp">Bob Carpenter
    </a>
 */
public class SudokuSolverImpl implements SudokuSolver {
    private static Registry registry;
    private static String name;

    @Override
    public void shutdown() {
        try {
            registry.unbind(name);
            UnicastRemoteObject.unexportObject(this, true);
        } catch (RemoteException | NotBoundException e) {
            throw new RuntimeException(e);
        }
    }

    /**
     * Unit tests the {@code DepthFirstSearch} data type.
     *
     * @param args the command-line arguments
     */
    public static void main(final String[] args) {
        name = args[0];

        try {
            final SudokuSolverImpl server = new SudokuSolverImpl
    ();
            final SudokuSolver stub = (SudokuSolver)
    UnicastRemoteObject.exportObject(server, 0);
            registry = LocateRegistry.getRegistry();
            registry.rebind(name, stub);
        } catch (final Exception e) {
            //System.err.println("Server exception:");
            e.printStackTrace();
        }
    }

    @Override
    public int[][] solve(final int[][] cells) {
        if (solve(0, 0, cells)) {
            return cells;
        } else {
            return null;
        }
    }

    private boolean solve(int i, int j, final int[][] cells) {
        if (i == 9) {
            i = 0;
            if (++j == 9) {
                return true;
            }
        }
        if (cells[i][j] != 0) {
            return solve(i + 1, j, cells);
        }

        for (int val = 1; val <= 9; ++val) {
            if (legal(i, j, val, cells)) {
                cells[i][j] = val;
                if (solve(i + 1, j, cells)) {
                    return true;
                }
            }
        }
        cells[i][j] = 0; // reset on backtrack
        return false;
    }
```

```
private boolean legal(final int i, final int j, final int val
, final int[][] cells) {
    for (int k = 0; k < 9; ++k) {
        // row
        if (val == cells[k][j]) {
            return false;
        }
    }

    for (int k = 0; k < 9; ++k) {
        // col
        if (val == cells[i][k]) {
            return false;
        }
    }
        final int boxRowOffset = i / 3 * 3;
        final int boxColOffset = j / 3 * 3;
        for (int k = 0; k < 3; ++k) {
            // box
            for (int m = 0; m < 3; ++m) {
                if (val == cells[boxRowOffset + k][boxColOffset +
 m]) {
                    return false;
                }
            }
        }

        return true; // no violations, so it's legal
    }
}
```

References

[1] Akka [Internet]. Typesafe Inc.; c2011–2020 [cited 2020 Oct 13]. Available from: http://akka.io/.

[2] Aldinucci, Marco; Danelutto, Marco; Kilpatrick, Peter; Dazzi, Patrizio. From Orc Models To Distributed Grid Java Code. In: Gorlatch, Sergei; Fragopoulou, Paraskevi; Priol, Thierry, eds.. *Grid Computing: Achievements and Prospects*; 2008 Apr 2–4; Hersonissos, Crete, Greece. New York: Springer; 2008. p. 13–24. doi:10.1007/978-0-387-09457-1_2. Presented at CoreGRID Integration Workshop 2008 (CGIW'2008).

[3] AlTurki, Musab; Meseguer, José. Real-Time Rewriting Semantics of Orc. In: Leuschel, Michael; Podelski, Andreas, eds.. *PPDP'07: Proceedings of the 9th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*; 2007 Jul 14–16; Wrocław, Poland. New York: ACM; 2007. p. 131–142. doi:10.1145/1273920.1273938.

[4] AlTurki, Musab; Meseguer, José. Dist-Orc: A Rewriting-based Distributed Implementation of Orc with Formal Analysis. In: Ölveczky, Peter Csaba, editor. *Proceedings First International Workshop on Rewriting Techniques for Real-Time Systems*; 2010 Apr 6–9; Longyearbyen, Norway. Electronic Proceedings in Theoretical Computer Science; 2010. p. 26–45. doi:10.4204/EPTCS.36.2.

[5] André, Françoise; Pazat, Jean-Louis; Thomas, Henry. Pandore: A System to Manage Data Distribution. In: *ICS'90: Proceedings of the 4th International Conference on Supercomputing*; 1990 Jun 11–15; Amsterdam. New York: ACM; 1990. p. 380–388. doi:10.1145/77726.255179.

[6] Antoniu, Gabriel; Bougé, Luc; Namyst, Raymond; Pérez, Christian. Compiling Data-Parallel Programs To A Distributed Runtime Environment With Thread Isomigration. *Parallel Processing Lett*. 2000 Jun/Sep; **10**(2/3):201–214. doi:10.1142/S0129626400000202.

[7] Bal, Henri E.; Steiner, Jennifer G.; Tanenbaum, Andrew S. Programming Languages for Distributed Computing Systems. *ACM Comput Surv*. 1989 Sep; **21**(3):261–322. doi:10.1145/72551.72552.

[8] Banâtre, Michel; Belhamissi, Yasmina; Issarny, Valérie; Puaut, Isabelle; Routeau, Jean-Paul. Adaptive Placement of Method Executions within a Customizable Distributed Object-Based Runtime System: Design, implementation and performance. In: *Proceedings of the 15th International Conference on Distributed Computing Systems*; 1995 May 30–Jun 02; Vancouver, BC. New York: IEEE; 1995. p. 279–286. doi:10.1109/ICDCS.1995.500030.

[9] Bareau, Cyrille; Caillaud, Benoît; Jard, Claude; Thoraval, René. Correctness of Automated Distribution of Sequential Programs. In: *PARLE'93: Parallel Architectures*

and Languages Europe: 5th International PARLE Conference; 1993 Jun 14–17; Munich. Berlin: Springer; 1993. p. 517–528. (Lecture Notes in Computer Science; vol. 694). doi:10.1007/3-540-56891-3_41.

[10] Baude, Françoise; Caromel, Denis; Huet, Fabrice; Vayssière, Julien. Communicating Mobile Active Objects in Java. In: Bubak, Marian; Afsarmanesh, Hamideh; Williams, Roy; Hertzberger, Bob, eds.. *High Performance Computing and Networking: 8th International Conference (HPCN Europe 2000)*; 2000 May 8–10; Amsterdam. Berlin: Springer; 2000. p. 633–643. (Lecture Notes in Computer Science; vol. 1823). doi:10.1007/3-540-45492-6_79.

[11] Bauer, Michael; Treichler, Sean; Slaughter, Elliott; Aiken, Alex. Legion: Expressing locality and independence with logical regions. In: *SC'12: 2012 International Conference for High Performance Computing, Networking, Storage and Analysis*; 2012 Nov 10–16; Salt Lake City, UT. New York: IEEE; 2012. p. 1–11. doi:10.1109/SC.2012.71.

[12] Benveniste, Albert; Jard, Claude; Kattepur, Ajay; Rosario, Sidney; Thywissen, John A. QoS-Aware Management of Monotonic Service Orchestrations. *Formal Methods in Systems Design*. 2013 Jul; **44**(1):1–43. doi:10.1007/s10703-013-0191-7.

[13] Bierman, Gavin; Meijer, Erik; Schulte, Wolfram. The Essence of Data Access in Cω: The Power is in the Dot! In: Black, Andrew P., editor. *ECOOP 2005 — Object-Oriented Programming*; 2005 Jul 25–29; Glasgow, UK. Berlin: Springer; 2005. p. 287–311. (Lecture Notes in Computer Science; vol. 3586). doi:10.1007/11531142_13.

[14] Black, Andrew; Hutchinson, Norm; Jul, Eric; Levy, Henry; Carter, Larry. Distribution and Abstract Types in Emerald. *IEEE Trans Softw Eng*. 1987 Jan; **SE-13**(1):65–76. doi:10.1109/TSE.1987.232836.

[15] Böhm, Alexander; Kanne, Carl-Christian. Demaq/Transscale: Automated distribution and scalability for declarative applications. *Information Syst*. 2011 May; **36**(3):565–578. doi:10.1016/j.is.2010.07.007.

[16] Cardelli, Luca. A Language with Distributed Scope. In: *POPL'95: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*; 1995 Jan 23–25; San Francisco. New York: ACM; 1995. p. 286–297. doi:10.1145/199448.199516.

[17] Chen, DeQing; Tang, Chunqiang; Sanders, Brandon; Dwarkadas, Sandhya; Scott, Michael L. Exploiting High-level Coherence Information to Optimize Distributed Shared State. In: *PPoPP'03: Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*; 2003 Jun 11–13; San Diego. New York: ACM; 2003. p. 131–142. doi:10.1145/781498.781518.

[18] Cunningham, David; Grove, David; Herta, Benjamin; Iyengar, Arun; Kawachiya, Kiyokuni; Murata, Hiroki; Saraswat, Vijay; Takeuchi, Mikio; Tardieu, Olivier. Resilient X10: Efficient Failure-Aware Programming. In: *PPoPP'14: Proceedings*

of the 2014 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming; 2014 Feb 15–19; Orlando, FL. New York: ACM; 2014. p. 67–80. doi:10.1145/2555243.2555248.

[19] Danicic, Sebastian; Laurence, Michael R. Static Backward Slicing of Non-Deterministic Programs and Systems. *ACM Trans Program Lang Syst*. 2018 Aug; **40**(3):Article 11. doi:10.1145/2886098.

[20] de Bruijn, N. G. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Proceedings of the Koninklijke Nederlandse Akademie van Wetenschappen, Series A, Mathematical Sciences*. 1972; **75**(5):381–392. Also published as Indagationes Mathematicae, vol. 34.

[21] Dean, Jeffrey; Ghemawat, Sanjay. MapReduce: Simplified Data Processing on Large Clusters. *Commun ACM*. 2008 Jan; **51**(1):107–113. doi:10.1145/1327452.1327492.

[22] Duesterwald, E.; Gupta, R.; Soffa, M. Distributed Slicing and Partial Re-execution for Distributed Programs. In: *Languages and Compilers for Parallel Computing: 5th International Workshop*; 1992 Aug 3–5; New Haven, CT. Berlin: Springer; 1993. p. 497–511. (Lecture Notes in Computer Science; vol. 757). doi:10.1007/3-540-57502-2_67.

[23] Factor, Michael; Schuster, Assaf; Shagin, Konstantin. A Distributed Runtime for Java: Yesterday and Today. In: *18th International Parallel and Distributed Processing Symposium: IPDPS 2004*; 2004 Apr 26–30; Santa Fe, NM. New York: IEEE; 2004. p. 159–165. doi:10.1109/IPDPS.2004.1303149. IPDPS workshop: Sixth International Workshop on Java for Parallel and Distributed Computing — JAVAPDC.

[24] Factor, Michael; Schuster, Assaf; Shagin, Konstantin. A Platform-Independent Distributed Runtime for Standard Multithreaded Java. *International J Parallel Programming*. 2006 Apr; **34**(2):113–142. doi:10.1007/s10766-006-0007-0.

[25] Fournet, Cédric; Gonthier, Georges; Levy, Jean-Jacques; Maranget, Luc; Rémy, Didier. A Calculus of Mobile Agents. In: Montanari, Ugo; Sassone, Vladimiro, eds.. *CONCUR '96: Concurrency Theory: 7th International Conference*; 1996 Aug 26–29; Pisa, Italy. Berlin: Springer; 1996. p. 406–421. (Lecture Notes in Computer Science; vol. 1119). doi:10.1007/3-540-61604-7_67.

[26] Garg, Vijay K.; Mittal, Neeraj. On Slicing a Distributed Computation. In: *21st International Conference on Distributed Computing Systems (ICDCS 2001)*; 2001 Apr 16–19; Mesa, AZ. New York: IEEE; 2001. p. 322–329. doi:10.1109/ICDSC.2001.918962.

[27] Google, Inc. gRPC Motivation and Design Principles [Internet]. 2015 Sep. Available from: http://www.grpc.io/posts/principles.

[28] Apache Hadoop [Internet]. The Apache Software Foundation; c2006–2020 [cited 2020 Oct 13]. Available from: https://hadoop.apache.org/.

[29] Haines, Matthew Dennis. Distributed Runtime Support for Task and Data Management. Boulder: Colorado State University; 1993 Aug. Report No.: CS-93-110. Available from: http://www.cs.colostate.edu/TechReports/Reports/1993/tr-110.pdf.

[30] Haridi, Seif; Van Roy, Peter; Smolka, Gert. An Overview of the Design of Distributed Oz. In: *Second International Symposium on Parallel Symbolic Computation: PASCO '97*; 1997 Jul 20–22; Kihei, HI. New York: ACM; 1997. p. 176–187. doi:10.1145/266670.266726.

[31] Ibrahim, Ali; Jiao, Yang; Tilevich, Eli; Cook, William R. Remote Batch Invocation for Compositional Object Services. In: Drossopoulou, Sophia, editor. *ECOOP 2009 — Object-Oriented Programming*; 2009 Jul 6–10; Genoa, Italy. Berlin: Springer; 2009. p. 595–617. (Lecture Notes in Computer Science; vol. 5653). doi:10.1007/978-3-642-03013-0_27.

[32] Isard, Michael; Budiu, Mihai; Yu, Yuan; Birrell, Andrew; Fetterly, Dennis. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In: *Proceedings of EuroSys 2007 Conference*; 2007 Mar 21–23; Lisbon, Portugal. New York: ACM; 2007. p. 59–72. doi:10.1145/1272996.1273005.

[33] Isard, Michael; Prabhakaran, Vijayan; Currey, Jon; Wieder, Udi; Talwar, Kunal; Goldberg, Andrew. Quincy: Fair Scheduling for Distributed Computing Clusters. In: *SOSP'09: Proceedings of the Twenty-Second ACM SIGOPS Symposium on Operating Systems Principles*; 2009 Oct 11–14; Big Sky, MT. New York: ACM; 2009. p. 261–276. doi:10.1145/1629575.1629601.

[34] Jul, Eric. Separation of Distribution and Objects. In: Guerraoui, Rachid; Nierstrasz, Oscar; Riveill, Michel, eds.. *Object-Based Distributed Programming: ECOOP '93 Workshop*; 1993 Jul 26–27; Kaiserslautern, Germany. Berlin: Springer; 1993. p. 47–54. (Lecture Notes in Computer Science; vol. 791). doi:10.1007/BFb0017533.

[35] Kitchin, David; Quark, Adrian; Cook, William R.; Misra, Jayadev. The Orc Programming Language. In: Lee, David; Lopes, Antónia; Poetzsch-Heffter, Arnd, eds.. *Proceedings of FMOODS/FORTE 2009*; 9–11 Jun 2009; Lisbon, Portugal. Berlin: Springer; 2009. p. 1–25. (Lecture Notes in Computer Science; vol. 5522). doi:10.1007/978-3-642-02138-1_1.

[36] Launchbury, John; Elliott, Trevor. Concurrent Orchestration in Haskell. In: *Haskell'10: Proceedings of the 2010 ACM SIGPLAN Haskell Symposium*; 2010 Sep 30; Baltimore, Maryland, USA. New York: ACM; 2010. p. 79–90. doi:10.1145/1863523.1863534.

[37] Malewicz, Grzegorz; Austern, Matthew H.; Bik, Aart J. C.; Dehnert, James C.; Horn, Ilan; Leiser, Naty; Czajkowski, Grzegorz. Pregel: A System for Large-Scale Graph Processing. In: *SIGMOD'10: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*; 2010 Jun 06–10; Indianapolis, IN. New York: ACM; 2010. p. 135–146. doi:10.1145/1807167.1807184.

[38] McCord, Brian; Kitchin, David; Thywissen, John A.; Misra, Jayadev. Orc Reference Manual v2.1.0. Austin, TX: The University of Texas at Austin, Department of Computer Science; 2013 Sep. Regular tech report. Report No.: TR-13-24. Available from: https://apps.cs.utexas.edu/tech_reports/reports/tr/TR-2209.pdf.

[39] McCord, Brian; Kitchin, David; Thywissen, John A.; Misra, Jayadev. Orc User Guide v2.1.0. Austin, TX: The University of Texas at Austin, Dept. of Computer Science; 2013 Sep. Regular tech report. Report No.: TR-13-23. Available from: https://apps.cs.utexas.edu/tech_reports/reports/tr/TR-2208.pdf.

[40] Microsoft Corporation. An Introduction to Microsoft .NET Remoting Framework [Internet]. 2007 Apr [cited 2020 Oct 13]. Available from: https://msdn.microsoft.com/en-us/library/ms973864.aspx.

[41] Milicevic, Aleksandar; Jackson, Daniel; Gligoric, Milos; Marinov, Darko. Model-Based, Event-Driven Programming Paradigm for Interactive Web Applications. In: *Onward! 2013: The Proceedings of the 2013 International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*; 2013 Oct 29–31; Indianapolis, IN. New York: ACM; 2013. p. 17–36. doi:10.1145/2509578.2509588.

[42] Murray, Derek G.; Schwarzkopf, Malte; Smowton, Christopher; Smith, Steven; Madhavapeddy, Anil; Hand, Steven. CIEL: A Universal Execution Engine for Distributed Data-Flow Computing. In: *NSDI '11: 8th USENIX Symposium on Networked Systems Design and Implementation*; 2011 Mar 30–Apr 1; Boston, MA. USENIX Association; 2011. p. 113–126. Available from: https://www.usenix.org/legacy/events/nsdi11/tech/full_papers/Murray.pdf.

[43] Object Management Group. *The Common Object Request Broker (CORBA): Architecture and Specification*. Object Management Group; 1995.

[44] Pankratius, Victor; Adl-Tabatabai, Ali-Reza. A Study of Transactional Memory vs. Locks in Practice. In: *SPAA'11: Proceedings of the Twenty-Third Annual Symposium on Parallelism in Algorithms and Architectures*; 2011 Jun 4–6; San Jose, CA. New York: ACM; 2011. p. 43–52. doi:10.1145/1989493.1989500.

[45] Peters, Arthur Michener; Kitchin, David; Thywissen, John A.; Cook, William R. OrcO: A Concurrency-First Approach to Objects. In: *OOPSLA'16: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*; 2016 Oct 30–Nov 4; Amsterdam. New York: ACM; 2016. p. 548–567. doi:10.1145/2983990.2984022.

[46] Peters, Arthur Michener; Thywissen, John A.; Cook, William R.; Rossbach, Christopher J. PITCHFORC: Concurrent Programming at Rack Scale. In: *MARS'17: 7th Workshop on Multi-core and Rack Scale Systems*; 2017 Apr 23; Belgrade, Serbia. 2017.

[47] Peters, Arthur Michener; Thywissen, John A.; Rossbach, Christopher J. PorcE: A Deparallelizing Compiler. In: *SFMA'18: 8th Workshop on Systems for Multi-core and Heterogeneous Architectures*; 2018 Apr 23; Porto, Portugal. 2018.

[48] Peters, Arthur Michener; Thywissen, John A.; Rossbach, Christopher J. PorcE: A Deparallelizing Compiler. In: *MPLR'19: Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*; 2019 Oct 21–22; Athens, Greece. New York: ACM; 2019. p. 117–130. doi:10.1145/3357390.3361023.

[49] Philips, Laure; De Roover, Coen; Van Cutsem, Tom; De Meuter, Wolfgang. Towards Tierless Web Development Without Tierless Languages. In: *Onward! 2014: Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*; 2014 Oct 20–24; Portland, OR. New York: ACM; 2014. p. 69–81. doi:10.1145/2661136.2661146.

[50] Plotkin, Gordon D. A structural approach to operational semantics. *J Logic Algebraic Programming*. 2004 Jul–Dec; **60–61**:17–139. doi:10.1016/j.jlap.2004.05.001. Reprint of: Plotkin, Gordon D. A structural approach to operational semantics. Aarhus University; 1981. Report No.: DAIMI FN-19.

[51] Rossbach, Christopher J.; Hofmann, Owen S.; Witchel, Emmett. Is Transactional Programming Actually Easier? In: *PPoPP'10: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*; 2010 Jan 9–14; Bangalore, India. New York: ACM; 2010. p. 47–56. doi:10.1145/1693453.1693462.

[52] Rossbach, Christopher J.; Yu, Yuan; Currey, Jon; Martin, Jean-Philippe; Fetterly, Dennis. Dandelion: A Compiler and Runtime for Heterogeneous Systems. In: *SOSP'13: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*; 2013 Nov 3–6; Farmington, Pennsylvania. New York: ACM; 2013. p. 49–68. doi:10.1145/2517349.2522715.

[53] Serrano, Manuel. Programming Web Multimedia Applications with Hop. In: *MM'07: Proceedings of the 15th International Conference on Multimedia*; 2007 Sep 23–28; Augsburg, Germany. New York: ACM; 2007. p. 1001–1004. doi:10.1145/1291233.1291450.

[54] Setälä, Mikko; Kukkala, Petri; Arpinen, Tero; Hännikäinen, Marko; Hämäläinen, Timo D. Automated Distribution of UML 2.0 Designed Applications to a Configurable Multiprocessor Platform. In: *Embedded Computer Systems: Architectures, Modeling, and Simulation: 6th International Workshop, SAMOS 2006*; 2006 Jul 17–20; Samos, Greece. Berlin: Springer; 2006. p. 27–38. (Lecture Notes in Computer Science; vol. 4017). doi:10.1007/11796435_5.

[55] Sewell, Peter; Wojciechowski, Paweł T.; Unyapoth, Asis. Nomadic Pict: Programming Languages, Communication Infrastructure Overlays, and Semantics for Mobile Computation. *ACM Trans Programming Languages Syst*. 2010 Apr; **32**(4):Article 12. doi:10.1145/1734206.1734209.

[56] Thywissen, John A.; Peters, Arthur Michener; Cook, William R. Implicitly Distributing Pervasively Concurrent Programs. The University of Texas at Austin, Department of Computer Science; 2016 Jul. Regular tech report. Report No.: TR-16-06. Available from: http://apps.cs.utexas.edu/tech_reports/reports/tr/TR-2232.pdf.

[57] Thywissen, John A.; Peters, Arthur Michener; Cook, William R. Implicitly Distributing Pervasively Concurrent Programs [extended abstract]. In: *PMLDC'16: First Workshop on Programming Models and Languages for Distributed Computing*; 2016 Jul 17; Rome, Italy. New York: ACM; 2016. p. Article 1. doi:10.1145/2957319.2957370.

[58] Thywissen, John A.; Peters, Arthur Michener; Rossbach, Christopher J. Local Operations Should Appear to Be Remote: Consistent Semantics Enable Transparent Distribution. In: *SFMA'18: 8th Workshop on Systems for Multi-core and Heterogeneous Architectures*; 2018 Apr 23; Porto, Portugal. 2018.

[59] Tilevich, Eli; Smaragdakis, Yannis. J-Orchestra: Automatic Java Application Partitioning. In: Magnusson, Boris, editor. *ECOOP 2002 — Object-Oriented Programming: 16th European Conference*; 2002 Jun 10–14; Málaga, Spain. Berlin: Springer; 2002. p. 178–204. (Lecture Notes in Computer Science; vol. 2374). doi:10.1007/3-540-47993-7_8.

[60] Van Roy, Peter; Haridi, Seif; Brand, Per; Smolka, Gert; Mehl, Michael; Scheidhauer, Ralf. Mobile Objects in Distributed Oz. *ACM Trans Programming Languages Syst*. 1997 Sep; **19**(5):804–851. doi:10.1145/265943.265972.

[61] Waldo, Jim; Wyant, Geoff; Wollrath, Ann; Kendall, Sam. A Note on Distributed Computing. In: Vitek, Jan; Tschudin, Christian, eds.. *Mobile Object Systems Towards the Programmable Internet: Second International Workshop, MOS'96*; 1996 Jul 8–9; Linz, Austria. Berlin: Springer; 1996. p. 49–66. (Lecture Notes in Computer Science; vol. 1222). doi:10.1007/3-540-62852-5_6.

[62] Weisenburger, Pascal; Wirth, Johannes; Salvaneschi, Guido. A Survey of Multi-tier Programming. *ACM Comput Surv*. 2020 Sep; **53**(4):Article 81. doi:10.1145/3397495.

[63] Wheeler, David. SLOCCount [Internet; cited 2020 Oct 13]. Available from: http://www.dwheeler.com/sloccount/.

[64] Wollrath, Ann; Riggs, Roger; Waldo, Jim. A Distributed Object Model for the Java System. In: *Proceedings of the USENIX 1996 Conference on Object-Oriented Technologies (COOTS)*; 1996 Jun 17–21; Toronto, Canada. Berkeley, CA: USENIX; 1996. p. 219–232. Available from: https://www.usenix.org/legacy/publications/library/proceedings/coots96/wollrath.html.

[65] Würthinger, Thomas; Wimmer, Christian; Wöß, Andreas; Stadler, Lukas; Duboscq, Gilles; Humer, Christian; Richards, Gregor; Simon, Doug; Wolczko, Mario. One VM to Rule Them All. In: *Onward! 2013: The Proceedings of the 2013 International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*; 2013 Oct 29–31; Indianapolis, IN. New York: ACM; 2013. p. 187–204. doi:10.1145/2509578.2509581.

[66] Zaharia, Matei; Chowdhury, Mosharaf; Das, Tathagata; Dave, Ankur; Ma, Justin; McCauley, Murphy; Franklin, Michael J.; Shenker, Scott; Stoica, Ion. Resilient

Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In: *NSDI '12: 9th USENIX Symposium on Networked Systems Design and Implementation*; 2012 Apr 25–27; San Jose, CA. USENIX Association; 2012. p. 15–28. Available from: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia.

[67] Zaharia, Matei; Das, Tathagata; Li, Haoyuan; Hunter, Timothy; Shenker, Scott; Stoica, Ion. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In: *SOSP'13: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*; 2013 Nov 3–6; Farmington, PA. New York: ACM; 2013. p. 423–438. doi:10.1145/2517349.2522737.

[68] Zhang, Irene; Szekeres, Adriana; Van Aken, Dana; Ackerman, Isaac; Gribble, Steven D.; Krishnamurthy, Arvind; Levy, Henry M. Customizable and Extensible Deployment for Mobile/Cloud Applications. In: *OSDI'14: Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*; 2014 Oct 6–8; Broomfield, CO. Berkeley, CA: USENIX Association; 2014. p. 97–112.

[69] Zhu, Wengzhan; Wang, Cho-Li; Lau, Francis C. M. JESSICA2: A Distibuted Java Virtual Machine with Transparent Migration Support. In: *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER 2002)*; 2002 Sep 23–26; Chicago. New York: IEEE; 2002. p. 381–388. doi:10.1109/CLUSTR.2002.1137770.

Vita

John Thywissen received a Master of Science in Computer Science from The University of Texas at Austin, a Master of Science in Engineering Management from Southern Methodist University, and a Bachelor of Science in Computer Science from the University of Kansas. He also is a graduate of London Business School's Accelerated Development Programme.

During his PhD studies at UT Austin, he was an Assistant Instructor, Teaching Assistant, and Graduate Research Assistant. He also completed a research internship at Caltech/NASA JPL (Jet Propulsion Laboratory).

Prior to his time at UT Austin, he worked extensively in technology services, most recently at EDS (Electronic Data Systems Corp., later merged with HP), where he had many roles, including Senior Systems Engineer, Enterprise Programme Director, and Managing Consultant. At EDS and other firms, he provided technology expertise, advice, and education to business and government clients, and designed and led development of numerous large technology systems. He is the inventor on two U.S. patents, related to encryption and to pricing project risk.

At UT Austin, he received a Dean's Excellence Award and a Teaching Assistant Excellence Award. He was selected as a member of the Phi Kappa Phi national scholastic honor society and the Sigma Xi scientific research honor society. He also is a member of the ACM and a Senior Member of the IEEE.

He also holds a Commercial Pilot license and is an active Flight Instructor.

Email address: jthywiss@cs.utexas.edu

This manuscript was typed by the author.