OrcO: A Concurrency-First Approach to Objects

Arthur Michener Peters
The University of Texas at Austin, USA
amp@cs.utexas.edu

David Kitchin
Google Inc., USA
dkitchin@google.com

John A. Thywissen
The University of Texas at Austin, USA
jthywiss@cs.utexas.edu

William R. Cook
The University of Texas at Austin, USA
wcook@cs.utexas.edu

Abstract

The majority of modern programming languages provide concurrency and object-orientation in some form. However, object-oriented concurrency remains cumbersome in many situations. We introduce the language OrcO, Orc with concurrent Objects, which enables a flexible style of concurrent object-oriented programming. OrcO extends the Orc programming language by adding abstractions for programming-in-the-large; namely objects, classes, and inheritance. OrcO objects are designed to be orthogonal to concurrency, allowing the concurrent structure and object structure of a program to evolve independently. This paper describes OrcO's goals and design and provides examples of how OrcO can be used to deftly handle events, object management, and object composition.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Concurrent programming structures; D.3.2 [*Programming Languages*]: Language Classifications—Object-oriented languages

Keywords Composition, futures, language design, mixin inheritance, object supervision, Orc

1. Introduction

The move to multi-core computers and the rise of network services have made concurrent programming ubiquitous. Since object-oriented programming languages are prevalent today, concurrent programming in object-oriented languages is increasingly significant. Language designers have developed a wide range of techniques to combine concurrency and object-oriented programming, but object-oriented concurrency remains challenging.

The Orc programming language [15, 22] enables a *concurrency-first* style of programming, in which programmers start with a concurrent program, instead of adding concurrency only when it is required. While the concurrency-first approach is unorthodox, it merits consideration for today's highly concurrent programs. This paper introduces *OrcO*, Orc with concurrent Objects, a language extension which adds object-oriented programming constructs to Orc while enhancing the concurrency-first style of programming.

Many popular languages implement concurrency using a library of primitives, such as threads and locks. This allows the programmer to choose between many different concurrency tools and libraries. However, the explicit encoding of concurrency also tends to cause problems when concurrency requirements change, or when multiple different concurrent libraries need to work together. OrcO offers a similar freedom of choice, but provides common underlying concurrency primitives through which varied styles of concurrent code can interact and evolve.

Another common approach is to integrate concurrency into the existing object-oriented abstractions of a language. This integration ties the concurrent structure of the program to its object-oriented design and therefore limits where concurrency can appear in the program. For instance, in pure active object systems concurrency can only exist between objects so a new object must be introduced to add concurrency even if that object will be adversely coupled to other objects.

OrcO objects are designed to be orthogonal to concurrency, allowing the concurrent structure and the object structure of a program to evolve independently. Sequential objects, such as those provided by Java, are not orthogonal to concurrency because they require sequential execution in many parts of their semantics. For instance, they use sequential initialization, and thereby prevent concurrent initialization without a

OOPSLA'16, November 2–4, 2016, Amsterdam, Netherlands
ACM. 978-1-4503-4444-9/16/11...
http://dx.doi.org/10.1145/2983990.2984022

Site call	`f(`e_1`,..., `e_n`)`
Parallel	`f	g`
Graft	`val x = f # g`
Branch	`f >x> g`
Trim	`{| f |}`
Otherwise	`f ; g`
If–then–else	`if e then f else g`
Function definition	`def F(`e_1`, ..., `e_n`) = f # g`
Anonymous function	`{ f }`
Function call	`f(`e_1`,..., `e_n`)`

(a) The Orc syntax

Class definition	`class C extends D { ... } g`
Instantiation	`new C`, `new { ... }`, or `new C { ... }`
Field access	`o.x`
Mixin	`C with D`

(b) Additional syntax in OrcO

Figure 1. The Orc and OrcO combinators and declarations. The meta-variables `e`, `f`, and `g` represent an expression. The meta-variable `x` represents a variable name. The meta-variables `C` and `D` represent classes.

special encoding. Such manual encodings affect both the object being initialized and its clients, preventing independent evolution.

In this paper, we: (1) Propose a new language, OrcO, in which objects and concurrency are orthogonal but coordinated (Section 3); (2) Discuss the design of OrcO and how it relates to previous approaches (Section 4); (3) Show how OrcO enables new programming techniques and allows new uses for existing ones (Section 5). This paper does not address issues related to OrcO's type system, or advanced class features such as member visibility, since OrcO is usable as a simple untyped language and its type system and class features will not be novel. We have developed a prototype implementation of OrcO. It is available at `https://github.com/orc-lang/orc/tree/ODO/OrcScala`.

2. Background on Orc

OrcO is based on the Orc programming language. In Orc, concurrent computations are built by composing primitive operations using Orc's concurrency combinators and declarations. Orc is pervasively concurrent, because it replaces conventional sequential control structures with concurrency combinators. Figure 1 shows the important elements of the Orc and OrcO syntax. This section and the next introduce Orc and OrcO informally. A formal semantics of OrcO is provided in Appendix A.

Executing an Orc expression results in the expression *publishing* values. Expressions may publish zero, one, or multiple separate values. These publications may occur at different times, meaning that Orc naturally supports asynchronous events. An expression's publications pass to the combinator enclosing the expression, which handles the publications as described below. Publication can be viewed as a generalization of the conventional notion of an expression returning a value, or as putting the value in an unordered stream.

The simplest examples of publishing expressions are `stop` and `signal`. The expression `stop` never publishes, and *halts*—finishes executing—immediately. The expression `signal` immediately publishes the value `signal` and halts. The value `signal` is used to represent an event with no information associated with it, similar to unit in functional languages.

Sites The primitive operations in an Orc program are *sites*. Sites are callable values whose execution is outside the Orc semantics and can be implemented in any language. They can publish any number of values when called, including zero. Sites allow an Orc program to interact with the outside world. For example, the site call `Prompt("Query:")` prompts the user by displaying the message "Query:" and then publishes the response, and `Rwait(1000)` publishes `signal` after 1000 ms. We discuss the semantics of site calls in more detail after defining the combinators.

The values published by sites are immutable, but can reference mutable state, allowing sites to implement mutable structures such as references or channels. For example, the site call `Ref()` publishes a reference `r` to a mutable value that can be set with `Write(r, v)` and read with `Read(r)`. Note that these site-created values are not Orc objects. They are opaque values that Orc can pass from one site call to another. However, Orc can use externally implemented objects which have fields and methods, such as Java objects. These external object use an object-like syntax, so the `Write` call from above can be written as `r.write(v)` even though `r` is not an OrcO object.

Orc provides common primitive values and types including strings, numbers, booleans, and tuples. Orc also supports standard operators on these primitive types, such as `+` for string concatenation and numeric addition. These operators are treated as site calls with the operands as arguments. In addition, Orc provides a range of flow-control primitives, including if–then–else. These primitives are internally encoded as site calls [22], but the encoding is omitted for brevity.

Parallel An application of the parallel combinator, `f | g`, executes `f` and `g` concurrently and publishes all publications of `f` and `g` as they are produced. As a running example, consider the following expression:

```
Prompt("Query:") | VoicePrompt("Query")
```

`Prompt` will publish responses entered as text by the user and `VoicePrompt` will use voice recognition to produce text from spoken input and publish the result. The overall expression will publish responses received from either subexpression; the publications can appear in any order.

Graft A use of the graft combinator,[1] `val x = f # g`, declares a new variable `x` whose scope is `g` and whose value is the first publication of `f`. Unlike sequential variable bindings, graft executes `f` and `g` concurrently, resolves `x` to the first publication of `f`, and allows `f` to continue executing. The variable `x` is a transparent future, so accesses to `x` in `g` block until `x` is resolved. If `f` halts, meaning all subexpressions finish executing, without publishing, so do accesses to `x`. Publications of `f` after the first are silently discarded. The `#` in the syntax is optional. It separates `f` from `g` in cases when the parsing is ambiguous. Graft can be used, in the running example, to prompt the user for multiple inputs at the same time, and then combine the results when they become available:

```
val q1 = Prompt("Query:")
val q2 = VoicePrompt("Query")
q1 + ", " + q2
```

Because `q1` (resp. `q2`) halts without publishing if `Prompt` (resp. `VoicePrompt`) halts without publishing, the final expression will only publish a value if both prompts publish.

Branch A use of the branch combinator,[2] `f >x> g`, executes `f` immediately, and executes a new copy of `g` for every value published by `f`. In each execution of `g`, the variable `x` is bound to the associated publication. The executions of `g` are concurrent with one another and with `f`. Branch provides a way to "fan-out" execution to handle many values published concurrently. The following example performs two concurrent queries for a user based on the text and voice inputs respectively:

```
(Prompt("Query:") | VoicePrompt("Query")) >s>
    PerformQuery(s)
```

An instance of `ProcessQuery` begins executing as soon as a prompt response is received; if no response is received then `PerformQuery` never executes. Branch is useful for event handling, since the publications that trigger executions of `g` may be spread over time. When `x` is not used in `g` it can be omitted, simply writing `f >> g`.

Trim An application of the trim combinator, `{| f |}`, executes `f` until it publishes for the first time, and then terminates the execution of `f`. This first publication of `f` is the only publication of the expression `{| f |}`. Applying trim to the running example allows it to query based on the text or voice input, depending on which is provided first:

```
{| Prompt("Query:") | VoicePrompt("Query") |}
    >s> PerformQuery(s)
```

This code will close both prompts as soon as the user gives an answer to either.

The *trim scope* of a trim combinator is `f` and all its dynamic subexpressions, including those in function bodies called from `f`. All expressions in a trim combinator's trim scope will be terminated when the expression in the combinator publishes. Trim scopes are nested whenever a trim combinator appears inside another trim scope.

Implementing complete termination of external site calls is difficult, if not impossible, so the semantics of trim do not require external sites to be terminated. Instead if the Orc call to a site is terminated, any responses from the external site are ignored. In current Orc implementations, external sites are notified when the Orc call to them has been terminated. The site may decide to halt or ignore the termination, depending on the requirements and capabilities of the external system. For example, the `Prompt` site will close the prompt when the site call is terminated. However, calls into normal Java code may ignore termination.

Otherwise An application of the otherwise combinator, `f ; g`, executes `f` initially. If `f` halts without publishing any value, then `f ; g` will execute `g`. If `f` publishes, then `g` will never execute. The publications of `f` or `g` become the publications of `f ; g`. The otherwise combinator is the only truly sequential combinator in Orc. It is used to detect halting of an expression and sometimes to detect failure. In the running example, the program will halt if the user closes the prompts or if the query returns no results. With otherwise, the program can handle these cases by showing a message:

```
{| Prompt("Query:") | VoicePrompt("Query") |}
    >s> PerformQuery(s) ;
Display("No query given!")
```

Functions Functions in Orc are defined using the syntax: `def f(`x_1`, ..., `x_n`) = g # h`. This defines the, possibly recursive, function `f` with a body `g` in the scope `h`. The `#` is an optional separator. A function can encapsulate our running example:

```
def queryRetry() =
  {| Prompt("Query:") | VoicePrompt("Query") |} ;
  queryRetry() #
...
```

This function prompts for a query repeatedly until it gets a result either by voice or as text. Functions are first class values and can be published or stored like any other value.

Orc also provides a syntax for anonymous functions, `{ e }`, similar to blocks in Smalltalk [19, p. 31]. These are used primarily to pass a block of code to another function to be executed. For instance, a synchronization function similar to a Java synchronized block can be called as follows: `synchronized({ ... })`.

Calls Calls take the usual form, `f(`e_1`, ..., `e_n`)` where `f` is a site or a function, but calls have concurrent semantics. Site calls and function calls execute differently. For site calls, the arguments are executed concurrently and the site is called once every argument has published a value. So, site calls act as join points or barriers. For function calls, the arguments are executed concurrently with the the body of the function. During function execution, accesses to an argument will

[1] Previous papers on Orc had a related combinator called "prune." Unlike prune, graft does not terminate `f` when it publishes.

[2] Branch was called "sequence" in previous papers.

```
def findAirline(name) =
  { Read(Prompt("Enter "+name+"'s price")) }

def askAirline(airline) =
  val response =
    {| (airline() ; 99999) |
       (Rwait(15000) >> 99999)
    |}
  if ~(response = 99999) then response else stop

def betterPrice(best, airline) =
  val price = askAirline(airline)
  min(best, price) ; best

betterPrice(
  betterPrice(
    betterPrice(99999, findAirline("Delta")),
    findAirline("United")
  ), findAirline("Southwest")
)
```

Figure 2. Finding the lowest-priced airline flight within time limits and publishing that price.

block until that argument publishes. This is similar to laziness except that the arguments begin executing concurrently with the call instead of only as needed.

Mechanically, every argument e_i to a call is replaced with a variable x_i which is bound to e_i using graft. For example, `f(Prompt("Query:"), VoicePrompt("Query"))` is expanded to:

```
val x1 = Prompt("Query:")
val x2 = VoicePrompt("Query")
f(x1, x2)
```

If the called `f` is a site, the call does not proceed until all parameters are resolved. If any site call argument halts without publishing, the whole site call will halt. If the called `f` is a function, the body will execute immediately and block as needed. The function body executes in the same trim scope as the call, so trim can terminate execution inside the function.

Example Figure 2 shows an example Orc program that gets price quotes from airlines in parallel, and returns the lowest price that is received within a time limit. For the purposes of this example, we define a function `findAirline` which, given the name of an airline, returns a function representing that airline. When the airline function is called, it will simulate a request for a quote by prompting the user for a quote. The `askAirline` function requests a price from an airline, and concurrently waits for 15 seconds using `Rwait`. If the airline function publishes a price within 15 seconds, the enclosing trim combinator terminates `Rwait`. If the airline function halts without publishing, for instance if the airline refuses to provide a quote, the overall expression publishes a distinguished value (99999) using the otherwise combinator and terminates Rwait. However, after 15 seconds with no response from the airline function, `Rwait` publishes and the overall expression publishes a distinguished value. This publication causes the trim combinator to terminate the request to the airline. If the distinguished value is published, the subsequent **`if`** causes the `askAirline` function to halt without publishing. The `betterPrice` function is given a current best price, and an airline. `betterPrice` calls `askAirline` and returns the lower of the airline's returned price and the current best price. If `askAirline` doesn't publish a result, `betterPrice` returns the current best price. Finally, we invoke `betterPrice` repeatedly, starting with our distinguished value as the current "best" price and passing the result of each `betterPrice` call to the next `betterPrice` call. Each nested `betterPrice` call is concurrent with its enclosing call, so by extension, the entire nested stack of calls will start concurrently. Therefore, all `betterPrice` calls are concurrent with one another and with the `findAirline` calls.

3. Introduction to OrcO

OrcO introduces objects to Orc in a manner that embraces the concurrency-first approach of Orc. In particular, object boundaries are not used to structure concurrency and concurrency within objects is the same as concurrency among objects. To introduce and control concurrency, OrcO uses the Orc combinators. Since objects and concurrency are not controlled using the same constructs, OrcO allows the concurrent structure and the object-oriented structure of a program to be orthogonal. Compared to Orc, OrcO adds tools for high-level program architecture and globally accessible futures. These differences and their implications are discussed in Section 4.

Objects OrcO objects are recursive, immutable records with an OrcO expression providing the value for each field. We call these expressions *field bodies* instead of initializers because the latter term is misleading in OrcO. Field bodies can continue executing after they publish a value. Objects are implicitly concurrent: all field bodies in an object are executed concurrently, and any access to a field blocks until the field's body publishes a value. This allows fields to represent ongoing computation, delayed initialization, and simple values all in one construct without interfering with the natural concurrency of Orc.

OrcO objects are created using the syntax:

`new { val` x_1 `=` e_1 `#` ... `#` **`val`** x_n `=` e_n **`}`**

This creates an object `o` with fields $x_1, \ldots, x_n$, and publishes it immediately. The **`new`** expression executes $e_1, \ldots, e_n$ and binds the respective fields $x_1, \ldots, x_n$ to the first publication of each. Any access to $o.x_i$ will block until e_i has published a value, and halt if e_i halts without publishing. For fields that represent ongoing computation, we will write **`val`** `_ =` ... to represent an unnamed field, since the value of the field will never be used. The expressions e_i can recursively access the object `o` as **`self`**. A field access **`self.x`** is abbreviated as `x` as usual. The `#` is an optional separator.

All the expressions e_i are executed in the same trim scope as the `new` expression, so a trim can be used to terminate all of an object's field bodies. This is similar to the "stop" or "poison pill" messages supported by many actor systems. Any field that was already resolved to a value before the object was terminated will still be accessible, but those that were not resolved will halt when accessed.

The fields of an OrcO object cannot be modified after they are resolved. However, a field may be resolved to a mutable value, such as a `Ref` as described in Section 2, to allow the object to have mutable state.

Methods in OrcO are simply fields with function values. OrcO methods are defined using `def f(...) = g`. This creates a function value and assigns it to a field named `f`. Methods can still be called on a terminated object, but the method may not be able to complete successfully if it accesses fields that are were never resolved.

As an example, the following object stores a database which is loaded at start up, and is reloaded every time the file changes:

```
new {
  def fileChanged() = ...
  def loadDB() = ...
  val db = Ref(loadDB())
  val _ = fileChanged() >> db.write(loadDB())
  def query(u) = ... db.read() ...
}
```

The method `loadDB` loads the database from a file and publishes it. The method `fileChanged` publishes `signal` every time the data file is modified. The field `db` contains a mutable reference to a database that is initialized to the result of `loadDB()`. The ongoing computation reloads the database and updates the value of `db` every time `fileChanged()` publishes, using `db.write` to set the `Ref` value. The method `query` uses the value of `db` to perform a query, and hence will block if it is called before the reference is created and assigned an initial value.

Classes Classes describe an object implementation that can be used repeatedly. OrcO's class mechanism is similar to the core trait system of the Scala programming language [30]. A class is defined using the syntax:

```
class C { val x1 = e1 # ... # val xn = en }
g
```

Once `C` is declared, `new C` is equivalent to `new { ... }` with the same fields. Classes may recursively instantiate themselves in their field bodies. The database object from above can be converted to a class without changing the body of the object: `class DB { ... }`. Now, a new database object can be created with `new DB`.

Inheritance OrcO supports class inheritance, which allows the programmer to build a new subclass by overriding fields in, and adding fields to, an existing superclass. The subclass can access the superclass implementation of any field `x` as `super.x`.

A subclass definition is in the form `class C extends D { ... }`. This defines the class `C` which inherits from `D` and extends and overrides fields. Instances of `C` will expose all fields of `D` in addition to the fields of `C`. As a convenience, the programmer can create new objects of anonymous subclasses using the syntax: `new C { ... }`. This form is the same as creating a subclass of `C` with the given fields and instantiating it.

A new class can extend the `DB` class by inheriting from it:

```
class RemoteDB extends DB { ...
  def query(u) = {| remoteDB.query(u) |
                    super.query(u) |}
}
```

This overrides `query` with a new implementation that queries both local and remote databases, and returns the first response.

In OrcO, field bodies in the superclass execute concurrently with field bodies in the subclass. This also applies to overridden fields. Any access to a field will wait until the field is bound, even if the implementation is in a subclass. Similarly, if the superclass calls a method overridden in the subclass, the subclass implementation will wait for any required subclass initialization to complete before publishing values back to the superclass. This is discussed in more detail in Section 4.

Mixins OrcO allows a class to inherit from more than one other class using mixin composition [6, 29]. Mixin composition combines multiple classes, each implementing one feature of an object, into a single class. In OrcO, wherever a class is required, the programmer can provide an expression `D with E` which mixes the classes `D` and `E` together.

OrcO mixins use a linearization algorithm derived from Scala's linearization [28, ch. 5] to align with the Scala-based class system. This choice is not fundamental to OrcO. A linearization is a list of classes such that every class appears after all of its superclasses and no class appears more than once. In OrcO, the linearization of a class or class expression is defined by the recursive function $\mathcal{L}(C)$.

$$\begin{array}{lcl} \mathcal{L}(O) & = & \langle\rangle \\ \mathcal{L}(E_1 \texttt{ with } E_2) & = & \mathcal{L}(E_1) \overleftarrow{+} \mathcal{L}(E_2) \\ \mathcal{L}(C \texttt{ extends } E\ \{\ldots\}) & = & \mathcal{L}(E) \overleftarrow{+} \langle C \rangle \end{array}$$

where O is the superclass of classes without a specified superclass in the program text. Here, $A \overleftarrow{+} B$ represents the concatenation operation where any elements of B that already appear in A are omitted. As an example, given the code

```
class A { val field = 0 }
class B extends A { val field = 2 }
class C extends A { val field = 1 }
class D extends B with C { val field = 3 }
```

the linearization of D is $\mathcal{L}(\texttt{D}) = \langle \texttt{A}, \texttt{C}, \texttt{B}, \texttt{D} \rangle$. The linearization function is refined and formalized in Appendix A.3.

A reference to `super.x` in a class E will access `x` as implemented by the closest class which implements `x` before E in the linearization. This means `super` will reference different classes depending on the computed linearization for `self`. As an example, consider the object `o = new D`. If `self` is `o`, then a reference to `super.field` in `D` will publish `2`, and in `B` `super.field` will publish `1`.

Like any OrcO object, instances of mixed classes execute all fields concurrently. This concurrency makes `with` similar to the parallel combinator, except that `with` operates at the class level. Like parallel, `with` allows multiple concurrent computations to be combined without conflict.

Implementation We have implemented a prototype OrcO interpreter, and used it to develop a few programs. Our current implementation of OrcO is dynamically typed, since we have not yet extended Orc's type system to include objects. However, we expect to be able to extend the type system, including Orc's local type inference, to use an object and class type system similar to Scala's. We expect this extended type system to be expressive enough to statically type all of the examples in this paper. In addition, we are developing an optimizing compiler for Orc programs.

4. Design Decisions for OrcO

The goal of OrcO is to provide tools to allow better object-oriented software engineering in the presence of pervasive concurrency. To do this, OrcO uses a novel combination of language design choices:

- OrcO does not restrict the use of concurrency combinators anywhere in the program.
- OrcO fields are implicit futures and are initialized concurrently.
- OrcO objects are controlled using the same primitives used for any Orc expression.

As in previous work on concurrent objects, OrcO objects execute concurrently with their environment. We hope these core design choices will allow better object-oriented and concurrent design by allowing the object-oriented and concurrent structures of the program to evolve independently of one another.

Previous approaches to concurrent object-oriented programming have focused on using sequential objects to structure concurrent programs. Briot, Guerraoui, and Lohr [9] developed a taxonomy describing approaches to concurrent object-oriented programming. The approaches are divided into three basic categories: library, integrative, and reflective. The *library approach* is the best known approach and appears in most modern object-oriented languages. It provides a library of concurrency primitives, such as threads and locks, represented as objects. This is the approach taken by languages such as Scala, Java, and C++. The *integrative approach* is seen in languages with active objects [23], actors [1], or any other language structure which is given special concurrent properties. This approach is used in languages such as ProActive [11], Encore [7], and Erlang [2]. The *reflective approach* defines the concurrent properties of objects using meta-programs which modify aspects of specific objects' behavior. The programmer can build custom concurrency models without changes to the core language. This approach was used to add concurrency support to Smalltalk in Actalk [8] and later by Yonezawa [40] and McAffer [24].

OrcO falls between the integrative and the library approaches and provides some of the advantages of each. OrcO provides a concurrency model like an integrative language, but the OrcO concurrency model is less restrictive than most integrative languages. So, OrcO libraries can implement almost any concurrency model using Orc's combinators (see Section 5.2). However, for libraries using the Orc combinators, interactions between dissimilar concurrent libraries are less complicated than in many library concurrent languages. OrcO does not attempt to provide the level of semantic flexibility of reflexive frameworks.

4.1 OrcO Objects

Prior to the introduction of objects, Orc did not provide any mechanism for modularizing large programs. This limited Orc's usefulness for large programs. OrcO provides objects that are specifically designed to enable modularization of concurrent programs and hopefully ease large-scale concurrent programming.

Objects are not the only approach to providing modularity. We considered using first-class modules as available in ML-derived languages. However, the object metaphor appears to be better suited to concurrent programming, because objects explicitly combine computation and data. Modules provide similar abstraction, but they don't encapsulate computation, represented by the module, with its data, represented by the values created by the module. This missing encapsulation would hide the connection between ongoing computation and data in concurrent programs.

To allow the Orc combinators to control objects, object field bodies execute in the scope of the instantiation of the object. This allows the programmer to use the Orc trim and otherwise combinators to control and monitor object execution (see Section 5.3). Work on Orc objects prior to OrcO included a form of objects that executed in their own protected scope, so that they could not be affected by the rest of the program. However, protecting objects in this way made such objects fundamentally different from other expressions in the language, and prevented the use of Orc's combinators to control these objects.

OrcO does not distinguish active and passive objects. Passive objects simply have no ongoing computation, though they will generally have concurrent initialization. In addition,

terminating an active object using trim turns that object into a passive object with no associated computation.

OrcO's uniform object model simplifies the program design process by insulating the rest of a program from changes in the amount of ongoing computation and concurrency in an object. Many integrative languages, such as Encore, have different semantics for calls to active and passive objects which makes this conversion difficult.

4.2 OrcO Fields and Futures

An OrcO field's state can only transition from unresolved (waiting for a value) to halted or resolved to a value, and cannot change after that, making them monotonic. Fields, like graft futures, can only be resolved to a publication of their associated field body. When an object is terminated with trim, resolved fields retain their value, instead of halting like unresolved fields. Together these properties simplify reasoning about the field values, because they can only take on one value or halt, not both, and the value can only come from one expression. This is in contrast to futures from many libraries, such as ECMAScript's Promise [18, sec. 25.4.3], which can be resolved from any point in the program which has a reference to the future. OrcO does not guarantee freedom from data races.

These choices make fields behave the same as Orc's graft variables, thus reducing the number of different kinds of interaction the programmer needs to worry about. There are several other options that we have considered, however each introduces complexity for little advantage.

- If fields publish the most recent value published by the field implementation, then every use of a field would be the result of a race between the use and the implementation. This would complicate reasoning about field accesses.
- If fields publish all values published by the field implementation after the field read occurs, then fields would no longer behave like variables. Instead, it would be possible to miss the value of a variable if the read came too late. This would result in race conditions even in simple situations.
- If fields publish all values published by the field implementation, then the programmer would need to handle multiple publications in a wider range of locations. We deemed that requiring the programmer to manually encode single value fields using trim or graft was not worth the added complexity.

These alternative field semantics can be encoded concisely in OrcO. For instance, the last option above can be encoded using an auxiliary function `AllValues` which captures all publications of it's argument.

```
val x = AllValues({ f })
```

The publications of `f` are accessed with `x.read()`.

The field futures in OrcO are a new primitive that was not available in Orc. The graft futures in Orc and OrcO are dynamically scoped and the future itself cannot escape that scope, since any access to the future will block until it is resolved. This prevents graft futures from being blocked on from other parts of the program. However, OrcO fields are transparent futures that can be forced from any part of the program with access to the object. This enables new usage patterns, such as using fields as flags to notify other parts of the program of some event (see Section 5.1). These global futures also directly support lenient data structures—data structures which are computed eagerly, but concurrently with their use. Lenient data structure require special encoding in the original Orc or strict languages, such as Erlang.

OrcO's use of futures for fields addresses some forms of uninitialized field access problems which can occur in sequential object-oriented languages [31]. Since all field bodies run concurrently, any access to a field which is not initialized will block, regardless of which class is providing the value. Therefore, a superclass can wait for subclass initialization to compute a value by using a field that the subclass has overridden. No field will ever publish an uninitialized value. It is possible for initialization to deadlock if several field implementations mutually block on one another. However, this is comparatively straightforward to debug since the execution will be blocked at the exact set of expressions that cause the problem, unlike `null` uninitialized values, which often cause problems that are only apparent later in the program's execution.

Finally, OrcO replaced one of the Orc combinators to avoid unexpected behavior in object-oriented programming. Previous work on Orc used a combinator called "prune" equivalent to the following in OrcO:

```
val x = {| f |} # g
```

This combination of trim and graft is useful in a functional programming style since if only one value will be bound to `x` then there is no reason to allow `f` to continue executing. However, in a concurrent object-oriented setting, the programmer will generally want objects instantiated by `f` to continue executing since the publication of `f` may contain references to them. So, OrcO eliminated the implicit termination of prune in favor of explicit termination with trim.

4.3 Internal Sites

OrcO adds support for *internal sites*, which are simply sites implemented in OrcO. They have the same semantics as sites implemented outside of OrcO. They can be called and provide no other interface. Where functions are locally executable values, sites are references to remote services. Internal sites are declared in OrcO similarly to functions, but any calls to them execute where the site is declared, not where it is called. This means the site body may execute in a different trim scope than its call. If the declaration of a site is terminated, all calls to the site halt.

Internal sites declared as members of an object will execute in the same trim scope as the object. Sites can expose functionality that cannot be invoked once the object has been terminated or whose execution should be in the trim scope of the object instead of the caller. Internal sites and functions coexist because methods of both types are useful and neither can be easily simulated by the other. Fully exploring the applications and implications of internal sites is beyond the scope of this paper.

5. Examples of OrcO

The following four examples demonstrate and discuss a number of salient characteristics of OrcO. Some of the examples provide comparisons between an OrcO and a conventional implementation. We use the Scala programming language as a state-of-the-art language for comparison purposes. In so doing, we mean no critique of Scala in particular. Instead, we use Scala because it is an excellent modern language that sets a high standard, and has inspired parts of OrcO's object model.

5.1 Event Handling

Event handling is a fundamentally asynchronous problem that appears in almost every application domain. Using the Orc combinators within an object provides a powerful way to select and handle events from multiple event streams.

An example of complex event handling is a GUI application that performs asynchronous operations such as database queries. The `GUI` class, in Figure 3, shows the core of an application which manages a simple GUI. It performs database queries based on GUI events and updates the GUI based on the results. The `Database` class implements a naïve "database", which loads data on start-up and then performs queries by examining every element of the database. Figure 4 shows the same classes implemented in Scala. Figure 5 describes two utility functions that simplify the Scala implementation. These functions are not part of the Scala standard library but can be implemented in a few lines of code. This example could be implemented elegantly in an actor system such as Akka [38] by wrapping the GUI and the database in actors. Here we compare against a more conventional approach to show the advantages of OrcO over the library-based approach to concurrency and asynchrony.

The `GUI` class, in Figure 3, shows an elegant way to handle events using Orc's combinators. Instead of installing a callback, the `GUI` class calls a method on the GUI component that publishes events as they occur. The program waits for either a button click or a text field activation, and as soon as either event happens, the GUI stops listening. The trim combinator guarantees that only one event will be processed, even if both events happen at the same time. The example uses an OrcO library function `repeat(f)` which calls `f` immediately and then calls `f` again each time the previous call publishes. In this case, `repeat` restarts the event handling

```
class GUI {
  val db
  val queryEntry = TextField()
  val queryButton = Button("Query")
  val resultsList = ListComponent()
  -- GUI event handling
  val _ = repeat({
    {| queryButton.onAction() |
       queryEntry.onAction() |} >>
    queryButton.setEnabled(false) >>
    resultsList.clear() >>
    db.query(queryEntry.getText()) >r>
    resultsList.add(r) >> stop ;
    queryButton.setEnabled(true)
  })
  -- Initialization event handling
  val _ = queryButton.setEnabled(false) >>
          {| db.ready | Rwait(5*seconds) |} >>
          queryButton.setEnabled(true)
}
class Database {
  val data = new MutableList
  val ready =
    loadData() >s> data.add(s) >> stop ; signal
  def loadData() = ...
  def query(query) =
    ready >> data.each() >s> (
    if s.matches(query) then s else stop)
}
```

Figure 3. Event handling example code in OrcO.

once the previous event is handled. In Scala, the GUI needs to explicitly track when it is making a query, to avoid starting another query while the previous one is still in progress. This is because the Scala code cannot detect when the previous query has finished asynchronously processing the event without some explicit form of communication.

During database initialization, the `GUI` class disables the `queryButton`. The database provides the field `ready`, which is only resolved when the database is initialized. The GUI accesses this field to wait for the database to initialize, but also sets a limit on how long it will wait. When either the database is ready or the timeout occurs, the GUI enables the `queryButton`. This pattern uses local concurrency to express the start-up pattern concisely. The Scala implementation must register a callback to be called in both the ready and timeout cases and make sure the action does not occur twice by using a flag variable.

The `Database` class loads and queries data in parallel using the branch combinator on the multiple publications from `loadData()`, which publishes records as it loads them, and `data.each()`, which publishes all the records in `data`. This approach makes loading data and returning results into asynchronous events. These events are handled in the `GUI` class, which displays query results as they become

```
class GUI(db: Database) {
  val queryEntry = new JTextField()
  val queryButton = new JButton("Query")
  val resultsList = new JList()
  // GUI event handling
  var isQuerying = false
  val queryActionListener = new ActionListener {
    def actionPerformed(e: ActionEvent): Unit = {
      if(isQuerying) return
      isQuerying = true
      queryButton.setEnabled(false)
      listModel.clear()
      db.query(queryEntry.getText()) { r =>
        onEDT { resultsList.addElement(r) }
      }.onComplete { _ =>
        onEDT {
          queryButton.setEnabled(true)
          isQuerying = false
        }
      }
    }
  }
  queryButton.addActionListener(
    queryActionListener)
  queryEntry.addActionListener(
    queryActionListener)
  // Initialization event handling
  queryButton.setEnabled(false)
  var isButtonEnabledFirstTime = false
  def initialEnableButton() = {
    onEDT {
      if (!isButtonEnabledFirstTime) {
        isButtonEnabledFirstTime = true
        queryButton.setEnabled(true)
      }
    }
  }
  onDelay(5000) { initialEnableButton() }
  db.ready.onComplete{_ => initialEnableButton()}
}

class Database {
  val data = new mutable.ArrayBuffer[String]()
  val ready = Future {
    for (s <- loadData()) { data += s }
  }
  def loadData() = ...
  def query(query: String)(k: (Result => Unit))={
    Future {
      Await.ready(ready, Duration.Inf)
      for (s <- data if s.matches(query)) {
        k(Result(s))
      }
    }
  }
}
```

Figure 4. Event handling example code in Scala.

```
// Run a block on the GUI event dispatch thread
def onEDT(f: => Unit) = ...
// Run a block after a timeout (on the EDT)
def onDelay(ms: Int)(f: => Unit) = ...
```

Figure 5. Utilities for Scala event handling example.

available. The Scala implementation uses a thread spawned with the Scala `Future` library to load and query the data, and callbacks to send the data to the application. Some degree of concurrency is lost relative to the OrcO implementation, since the matching is not performed in parallel or concurrently with the callback. An actor-based implementation would also lose this concurrency. The Scala GUI implementation handles query completion using a callback as well.

This example has two independently asynchronous systems: the GUI and the database. The OrcO implementation can use all events and operations homogeneously because OrcO abstracts the scheduling of the various systems. The OrcO wrapper for the GUI components integrates the GUI calls into OrcO's scheduling system. This is analogous to integrating a concurrent library into a Scala framework's scheduling system. However, a framework integration is specific to the framework, while an OrcO integration is usable in any OrcO program.

5.2 Embedding Other Concurrency Models

OrcO's flexibility allows programmers to embed other concurrency models in a way similar to library-based concurrent languages, such as Scala. This section shows an OrcO implementation of active objects and its use. This demonstrates that other concurrency models can be concisely embedded in OrcO, without being restricted by the underlying concurrency model as can happen in integrative languages. In addition, OrcO's transparent futures simplify usage of the resulting concurrent objects in cases where a Scala implementation would require explicit future handling.

Figure 6a shows a complete implementation of a sequential executor and its use to implement active object method dispatch. `SequentialExecutor` contains a queue object to manage the execution requests, which are represented as functions, and a `schedule` method which adds a function to the queue. The ongoing computation repeatedly takes a function out of the queue, calls it, and waits for it to complete. Once the otherwise combinator detects that the function has halted, **`signal`** publishes a value to cause the next function in the queue to execute. The function may be internally concurrent. `ActiveObjectBase` contains an executor representing the unique thread of control of the active object. The `scheduleMethod` method handles scheduling a method to execute and forwarding the publications of the call using a `Channel`. The function submitted to the scheduler puts each publication of the method into the channel and closes the channel when the method halts. The parallel `repeat(c.get)` expression at the end of `ActiveObjectBase`

```
class SequentialExecutor {
  val queue = BlockingQueue()
  def schedule(f) = queue.put(f)
  val _ = repeat({
    val f = queue.take()
    f() >> stop ; signal
  })
}
class ActiveObjectBase {
  val exec = SequentialExecutor()
  def scheduleMethod(f) =
    val c = Channel()
    exec.schedule({
      f() >x> c.put(x) >> stop ; c.close()
    }) >> stop | repeat(c.get)
}
```

(a) A mixin for active objects and the scheduler used to execute method bodies.

```
class Counter extends ActiveObjectBase {
  val v = Ref(0)
  def incr(x) = scheduleMethod({
    v.write(v.read() + x)
  })
  def read() = v.read()
}
```

(b) An active object implemented using the above library.

Figure 6. Implementing active objects in OrcO.

publishes each item added to the channel and halts once the channel is closed. This implementation may seem to only implement synchronous calls to methods, but the caller can continue executing while waiting for the result, because of the concurrent semantics of Orc.

Figure 6b shows an atomic counter implemented as an active object. Active object style methods are implemented by scheduling their body for later execution using `scheduleMethod`. The method `incr` uses this technique to guarantee sequential execution of updates to the `Ref`. However, the method `read` does not use sequential execution since the underlying `read` operation of `Ref` is already atomic.

OrcO also allows similar implementations of the actor model and monitors, among others. The out of order message processing allowed by the actor model requires a more complicated "mailbox" in place of the queue. However, the execution scheduling is very simple regardless of whether actors are allowed to receive messages synchronously as in Erlang or must provide a continuation as in Akka. For monitors, critical sections are implemented similarly to `scheduleMethod` above except using a lock instead of a scheduling queue. Condition notification in the monitor is implemented using standard techniques.

5.3 Controlling Objects

The Orc combinators and the semantics of OrcO objects allow OrcO libraries to control and monitor objects without restricting the range of concurrency patterns that the programmer can use in the objects. This is an example of the use of the trim and otherwise combinators on objects as discussed in Section 4. In addition, this example shows how these tools enable useful program control patterns developed for other programming models.

As an example, we present a simple supervisor library inspired by Erlang/OTP [39]. Erlang/OTP allows the programmer to construct a hierarchy of supervisors which manage failures in their subordinate objects. Figure 7a shows a simple OrcO supervision library, which provides superclasses for two different restart policies: `AllForOneSupervisor`, which restarts all supervised objects if any one of them fails, and `OneForOneSupervisor`, which only restarts the failed object.

The `SubordinateRef` class implements an object manager that creates the object and provides access to it. The `SubordinateRef` class calls its `constructor` method to create new instances. `SubordinateRef.current` is a `ClearableRef` which can be set to a value or `cleared` to make all subsequent `read()` calls block until it is set again. The `def SubordinateRef` defines a constructor function for `SubordinateRef` to simplify its use. The `Supervisor` class declares the common interface for supervisors. In this case, the interface is just a list of `subordinates` to supervise. `AllForOneSupervisor` and `OneForOneSupervisor` implement their distinct supervision strategies using ongoing computation. `AllForOneSupervisor` runs all the objects using branch and if any halts, as detected with the otherwise combinator, terminates the objects with trim. This process is repeated indefinitely. `OneForOneSupervisor` runs all the objects independently and uses the same technique to restart each object if it halts.

Figure 7b shows a small group of servers that are managed by two nested supervisors. If the database server fails, all the web servers will be shutdown and restarted; but if a single web server fails, it will be restarted without affecting other objects. This whole process can be started by simply instantiating `App`. `DbServer` and `WebServer` are constructor functions for the database and web servers.

This implementation has some interesting properties that distinguish it from Erlang/OTP:

- The objects in the supervisors can directly reference other objects in related supervisors.
- The use of transparent futures means that any dependent objects will block for their dependencies to be created exactly when that dependency is needed.
- The managed objects can be any OrcO expression with ongoing computation.

A Scala implementation of supervision would be complicated, because Scala lacks universal halt detection and termi-

```
class SubordinateRef {
  val constructor
  val current = ClearableRef()
  def get() = current.read()
  def run() = (current.write(constructor()) >>
               stop) ; current.clear()
}
def SubordinateRef(c) =
  new SubordinateRef { val constructor = c }

class Supervisor { val subordinates }
class AllForOneSupervisor extends Supervisor {
  val _ = repeat({ {|
    subordinates.each() >m>
    (m.run() >> stop ; signal)
  |} })
}
class OneForOneSupervisor extends Supervisor {
  val _ = subordinates.each() >m> repeat({
    m.run() >> stop ; signal
  })
}
```

(a) An implementation of object supervision.

```
class App extends AllForOneSupervisor {
  val subordinates = [webServers, dbServer]
  val webServers = SubordinateRef({
    new OneForOneSupervisor {
      val subordinates = [webSv1, webSv2]
      val webSv1 = SubordinateRef({
                WebServer(dbServer.get()) })
      val webSv2 = SubordinateRef({
                WebServer(dbServer.get()) })
    }
  })
  val dbServer = SubordinateRef({ DbServer() })
}
```

(b) Using the above classes to manage the execution of objects.

Figure 7. Object management example code in OrcO.

```
class Handler {
  val db
  def handle(r)
  def sendQuery(q) =
        val msg = new Query { val query = q }
        db.query(msg) >> stop |
        msg.reply.read()
}
class Query {
  val query
  val reply = Cell()
}
class Database {
  val queryChannel = Channel()
  def query(q) = queryChannel.put(q)
  val _ = repeat(queryChannel.get) >q>
          q.reply.write(doQuery(q))
  def doQuery(q) = ...
}
val database = new Database
onRequest() >r> {|
  r.reply((new HandlerImpl {
      val db = database
    }).handle(r)) |
  r.onDisconnect()
|}
```

(a) A request handling framework with a database.

```
class HandlerImpl extends Handler {
  def extractQuery(r) = ...
  def displayResults(r) = ...
  def handle(r) =
        val res = sendQuery(extractQuery(r))
        displayResults(res)
}
```

(b) An implementation of `Handler`.

Figure 8. A set of classes that implement a request handler and framework in OrcO.

nation. Every manageable object would need to provide an API to terminate it and to notify the `SubordinateRef` when it halts. In many cases these termination and halt monitoring tools would need to be custom written for each object to support the object's internal structure and concurrency. This requirement of a common interface would complicate using supervisors on objects not implemented with supervision in mind.

5.4 Composing Objects and Behaviors

The combination of objects with Orc's combinators in OrcO allows mixins to extend classes with concurrent or asynchronous behavior. This can be used to implement superposed computations [13, sec. 7.3] on top of an underlying computation. A *superposed computation* monitors, controls, or augments an underlying computation without interfering with its execution. To superpose computation over a group of interacting classes, the programmer can extend each class and communicate between the extensions.

One example of a superposed computation is logging in a web server. For each request, the server instantiates a `Handler` and calls its `handle` method. If the client closes the connection, the `Handler` object is killed. This is shown in Figure 8a along with the definitions of the `Handler`, `Query`, and `Database` classes. The specific example `Handler`, `HandlerImpl`, handles the request by parsing it, then dispatching a command to a database, and then formatting the result. The database command is handled by sending a message, so that terminating the `HandlerImpl` does not terminate the

```
class LogRecord {
  def add(v) = ...
}
class LoggingQuery extends Query {
  val logRec
}
class LoggingDatabase extends Database {
  def doQuery(q) =
    q.logRec.add(q) |
    (val v = super.doQuery(q)
    q.logRec.add(v) >> stop | v)
}
class LoggingHandler extends Handler {
  val logRec = new LogRecord
  val _ = Logger.submit(logRec)
  def handle(r) =
    logRec.add(r) >> stop |
    (val v = super.handle(r)
    logRec.add(v) >> stop | v)
  def sendQuery(q) =
    val msg = new LoggingQuery {
        val query = q
        val logRec = LoggingHandler.self.logRec
    }
    db.query(msg) >> stop |
    msg.reply.read()
}
```

(a) Logging superposed onto the handler and database.

```
class Timeout extends Handler {
  def handle(r) = {|
    super.handle(r) |
    Rwait(timeout) >> TimeoutError()
  |}
}
```

(b) A selection of features added to handlers.

```
class LoggingHandlerImpl extends HandlerImpl
    with Timeout with LoggingHandler {}
val database = new LoggingDatabase

onRequest() >r> {|
  r.reply((new LoggingHandlerImpl {
      val db = database
    }).handle(r)) |
  r.onDisconnect()
|}
```

(c) Using logging, timeout, and asynchrony mixins.

Figure 9. A set of classes that extend the OrcO request handler (Figure 8) with logging, timeouts, and asynchronous initialization.

database operation. This can be implemented more clearly with internal sites as described in Section 4.3. The message includes a `Cell` object to hold the reply from the database. A `Cell` is a write-once reference. The `Handler` will wait for the reply by reading the `Cell` and the database will send it by writing to the `Cell`. This is shown in Figure 8b.

To improve log comprehensibility, information about a single request and its database query are logged together in a single log record. To do this, the programmer superposes logging computation on top of `Handler`, `DB`, and `Query`. The superposed computation does not interfere with the execution of the handler or the database.

As shown in Figure 9a, the mixin class `LoggingHandler` creates a logging record and submits it to the logging framework without completing it. This guarantees that the request will be logged if any part of it was actually processed. `LoggingHandler.handle` adds the relevant values to the logger. `LoggingHandler` also overrides `sendQuery` to pass the log record to the `Query` object when it is created. `LoggingDB`'s `doQuery` method logs the query and the results using the log record added to `LoggingQuery`. Figure 9c shows how to instantiate and use the logging version of the database and handler.

In other cases, the developer may need to control the execution of the original class's code. Figure 9b shows `Timeout` which controls the execution of the superclass. The call `super.handle(r)` is wrapped in a trim combinator which is triggered to kill the handler and report a timeout after a specific amount of time. `Timeout` builds on other `Handler` classes, without needing to know anything about the concurrency or scheduling requirements of the superclass. Therefore, `Timeout` can be applied to any `Handler` as shown in Figure 9c.

To implement superposition and control of this kind in Scala, the programmer would need to explicitly implement concurrency support in almost every object. `HandlerImpl` and the objects it uses would need to provide support for termination at any point in their execution. Many methods would need to be written in an asynchronous style to avoid blocking the framework.

Superposition similar to this example can be implemented using an actor system such as Akka. An actor implementation would use a wrapper actor that represented the superposed computation. The wrapper would need to intercept and process messages both entering and leaving the underlying actor. This precludes the use of the automatic routing facilities Akka provides, such as message forwarding. The termination of `Timeout` would also be difficult to implement, because in Akka termination only affects one actor at a time not a group. So, while an Akka implementation would be manageable, the OrcO implementation shows some notable advantages.

6. Related Work

Active Objects Many integrative languages require sequential execution inside objects or groups of objects. Examples of this model include Asynchronous Sequential Processes [10], E [27], JCoBox [32], Panini [3], and ProActive [11]. ProActive and Asynchronous Sequential Processes provide transparent futures. In all of these languages, futures are only introduced at asynchronous calls to active or remote objects. This limits the flexibility of programming in these languages because asynchrony cannot be introduced without an object to support it. These languages focus on enabling interactions between sequential processes.

Emerald [4, 5], like OrcO, enables concurrency both between and within objects. Like many active object languages, all concurrency in Emerald is provided by a single sequential process attached to each object. However, Emerald allows concurrency within objects by allowing multiple operations on an object to be invoked at the same time and allowing these invocations to execute concurrently with the object's process. Shared data inside an object is protected with monitors. All invocations in Emerald are synchronous, which limits concurrency between objects.

Encore [7] is an active object language which tries to solve many of the limitations of the active object model by integrating other techniques without changing the core model. While OrcO and Encore seek to solve related programs, Encore's approach is very different. Encore includes multiple ways to express concurrency each suited to a different use case, unlike OrcO which uses the same concurrency primitives in all cases. For instance, Encore makes a strong distinction between object parallelism and data parallelism, using active objects for object parallelism and parallel types and Orc-inspired combinators for data parallelism. Encore lacks transparent futures.

Several object-oriented languages have actor libraries which allow actors to be used along with objects in various ways. These libraries are often used to emulate, loosely or strictly, the restrictions and properties of active object languages. A notable example is the Akka actor library [38].

Extended Actors Parallel actor monitors [33] provide a way to describe allowable concurrency within a single actor. This is implemented as a separate monitor that dispatches messages to the actor. The parallel actor monitor is allowed to dispatch messages to execute concurrently with other messages in the same actor. This addresses the lack of concurrency and parallelism within an actor, but does not allow the level of flexibility that OrcO provides.

Several projects have addressed the limitations of pure message passing models by introducing a safe shared memory model. Domains [16] provide an abstraction of memory regions which can be shared in various ways between actors. Accesses to domains enforce the use of locks to prevent race conditions. Domains do not change the execution limitations of the actors involved, and hence do not enable pervasively concurrent programming.

ABCL/1 [41] extends actors with non-transparent futures and new message passing capabilities that utilize them. This eliminates the need to split actor bodies for message receipt in the midst of a computation. AmbientTalk [17] uses the ABCL/1 object model as the basis of a distributed language.

Transparent Futures Multilisp [20] extends Scheme with explicitly created transparent futures and asynchronously executing expressions to resolve those futures. Multilisp does address some of the same adaptability problems as OrcO. However, Multilisp does not attempt to address object-orientation and does not encourage concurrency-first programming since concurrency is never the default.

Object-Oriented Distributed Computation X10 [14] and the derived Habanero-Java [12] provide tools for location-aware distributed programming over distributed data. These languages are not designed for concurrent asynchronous computing so much as parallel computing over large datasets. They provide a range of mutual exclusion and signaling primitives, but those primitives are limited with respect to nested concurrency. For instance, atomic code blocks cannot contain parallelism. OrcO does not have these restrictions and targets higher-level concurrency control and orchestration instead of parallel computation.

Concurrency Control Languages Several languages have focused on expressing the concurrency constraints of objects or programs. This helps alleviate some of the problems with using concurrent patterns in object-oriented languages, especially in the presence of inheritance. Jeeg [26] uses a temporal-logic-based language to describe concurrency constraints of Java programs. Jeeg and related approaches provide a language for restricting concurrency, but they do not enable flexible concurrent programming and generally still require the use of explicit threads.

Asynchrony Extensions Several modern languages have added asynchronous programming support, including F# [36], C# [25], and Python [34]. The asynchrony support comes in the form of a future type and a way to return control to a scheduler without losing the state of the local execution. These features allow the program to perform blocking tasks without blocking the event or request handling thread, as long as the blocking code is written with asynchrony in mind. Specifically, asynchronous functions return a future to represent their eventual result and other asynchronous functions block on the future using a special construct that allows other tasks to run until the future is resolved. F# and C# also provide tools for scheduling tasks to run on another thread to allow true parallelism.

Asynchronous programming addresses the problems of manual event-based programming, such as inversion of control [36]. Specifically, it avoids the need for the programmer to manually construct the continuation to be invoked when a

value becomes available. However, asynchronous programming does not address the general problem of writing concurrent or parallel systems. These asynchronous features are not needed in OrcO, because concurrency obviates the need for special handling of asynchronous events.

Concurrent Logic Programming Oz [35] provides concurrent object-oriented programming in a constraint logic programming language. All variables in Oz—including fields—are logic variables and assignment is bidirectional unification. These logic variables take the place of futures in other concurrent languages. However, logic variables can be assigned from anywhere in the program instead of a single position, which complicates reasoning about future resolution. In addition, bidirectional unification complicates efficient concurrent and distributed implementations of Oz. Oz is not concurrency-first, since to introduce concurrency an Oz program must execute a special thread construct and Oz method and function calls are synchronous by default.

7. Conclusion

OrcO provides an approach to concurrent object-oriented programming that occupies the space between the traditional integrative and library approaches. It eases concurrency-first programming through its novel pervasively concurrent design. This design may simplify reasoning about concurrency compared to the library approach while not limiting or hiding the concurrency of the program. OrcO provides potential advantages in event handling and asynchronous programming compared to traditional thread-based programming. It also enables the use of actor-style supervision in heterogeneous environments where different objects may have different concurrency properties. Finally, the concurrent semantics of inheritance in OrcO enable new uses for inheritance, such as superposition and execution control.

Orc and, by extension, OrcO have potential for transparent distributed execution [37]. This would allow OrcO objects to provide all of their flexibility and power in a distributed environment without the programmer needing to explicitly manage the distribution. Just as OrcO decouples concurrency from objects, it also decouples distribution from objects.

The techniques and tools provided by OrcO are applicable in a wide range of areas, including GUI programming, web programming, and even robotics. Concurrency-first programming will be more and more attractive as the number of purely sequential elements of programming and operating environments dwindles. Because of this, the pervasive concurrency of Orc and the flexible objects of OrcO may be very useful as the state of the art adjusts to the new requirements of modern applications.

We do not expect OrcO to replace all other languages for concurrent object-oriented programming. However, because OrcO uses a standard class model and is not tied to any type system, the important elements of its design could be used in a wide range of contexts. We hope that OrcO's design will inform the development and design of future languages.

Acknowledgments

Thanks to the anonymous reviewers for their useful and insightful comments. Special thanks to Jayadev Misra and Don Batory for their guidance, advice, and insights throughout this work.

References

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Dec. 1986. ISBN 0-262-01092-5.

[2] J. Armstrong. Erlang – A survey of the language and its industrial applications. In *The Ninth Exhibition and Symposium on Industrial Applications of Prolog (INAP)*, 1996. URL `http://www.erlang.se/publications/inap96.pdf`.

[3] M. Bagherzadeh and H. Rajan. Panini: A concurrent programming model for solving pervasive and oblivious interference. In *14th International Conference on Modularity (MODULARITY 2015)*, pages 93–108. ACM, 2015. doi:10.1145/2724525.2724568.

[4] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object structure in the Emerald system. In *OOPSLA '86: Object-Oriented Programming Systems, Languages, and Applications: Conference Proceedings*, pages 78–86, New York, 1986. ACM. doi:10.1145/28697.28706.

[5] A. P. Black, N. C. Hutchinson, E. Jul, and H. M. Levy. The development of the Emerald programming language. In *Proceedings: The Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III)*, pages 11–1–11–51, New York, 2007. ACM. doi:10.1145/1238844.1238855.

[6] G. Bracha and W. Cook. Mixin-based inheritance. In *OOPSLA/ECOOP '90: Proceedings of Joint Conference on Object-Oriented Programming Systems, Languages, and Applications / European Conference on Object-Oriented Programming*, pages 303–311. ACM, 1990. doi:10.1145/97945.97982.

[7] S. Brandauer, E. Castegren, D. Clarke, K. Fernandez-Reyes, E. B. Johnsen, K. I. Pun, S. L. T. Tarifa, T. Wrigstad, and A. M. Yang. Parallel objects for multicores: A glimpse at the parallel language Encore. In M. Bernardo and B. E. Johnsen, editors, *Formal Methods for Multicore Programming: 15th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2015*, pages 1–56, Cham, 2015. Springer International Publishing. doi:10.1007/978-3-319-18941-3_1.

[8] J.-P. Briot. Actalk: A testbed for classifying and designing actor languages in the Smalltalk-80 environment. In *ECOOP 89: Proceedings of the Third European Conference on Object-Oriented Programming*, pages 109–129. Cambridge University Press, 1989. ISBN 0-521-38232-7.

[9] J.-P. Briot, R. Guerraoui, and K.-P. Lohr. Concurrency and distribution in object-oriented programming. *ACM Comput. Surv.*, 30(3):291–329, Sept. 1998. doi:10.1145/292469.292470.

[10] D. Caromel and L. Henrio. *A Theory of Distributed Objects: Asynchrony — Mobility — Groups — Components*, chapter

Asynchronous Sequential Processes, pages 69–74. Springer, 2005. doi:10.1007/3-540-27245-3_4.

[11] D. Caromel, C. Delbé, A. Di Costanzo, M. Leyton, and Others. ProActive: An integrated platform for programming and running applications on grids and P2P systems. *Comput. Methods Sci. Technol.*, 12(1):69–77, 2006. doi:10.12921/cmst.2006.12.01.69-77.

[12] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-Java: The new adventures of old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java (PPPJ 2011)*, pages 51–61. ACM, 2011. doi:10.1145/2093157.2093165.

[13] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988. ISBN 0-201-05866-9.

[14] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: 20th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 519–538. ACM, 2005. doi:10.1145/1094811.1094852.

[15] W. Cook and J. Misra. Structured interacting computations. In *Software-Intensive Systems and New Computing Paradigms: Challenges and Visions*, volume 5380 of *Lecture Notes in Computer Science*, pages 139–145. Springer, 2008. doi:10.1007/978-3-540-89437-7_9.

[16] J. De Koster, S. Marr, T. Van Cutsem, and T. D'Hondt. Domains: Sharing state in the communicating event-loop actor model. *Comput. Lang. Syst. Struct.*, 45:132–160, 2016. doi:10.1016/j.cl.2016.01.003.

[17] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter. Ambient-oriented programming in AmbientTalk. In D. Thomas, editor, *ECOOP 2006 – Object-Oriented Programming: 20th European Conference, Nantes, France, July 3-7, 2006. Proceedings*, volume 4067 of *Lecture Notes in Computer Science*, pages 230–254, Berlin, 2006. Springer. doi:10.1007/11785477_16.

[18] Ecma International. ECMAScript 2015 language specification. Standard ECMA-262, 6th Edition, Ecma International, Geneva, June 2015.

[19] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983. ISBN 0-201-11371-6.

[20] R. H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, Oct. 1985. doi:10.1145/4472.4478.

[21] D. Kitchin, W. R. Cook, and J. Misra. A language for task orchestration and its semantic properties. In *CONCUR 2006 – Concurrency Theory: 17th International Conference: Proceedings*, volume 4137 of *Lecture Notes in Computer Science*, pages 477–491. Springer, 2006. doi:10.1007/11817949_32.

[22] D. Kitchin, A. Quark, W. Cook, and J. Misra. The Orc programming language. In *Formal Techniques for Distributed Systems: Joint 11th IFIP WG 6.1 International Conference FMOODS 2009 and 29th IFIP WG 6.1 International Conference FORTE 2009: Proceedings*, volume 5522 of *Lecture Notes in Computer Science*, pages 1–25. Springer, 2009. doi:10.1007/978-3-642-02138-1_1.

[23] R. G. Lavender and D. C. Schmidt. Active Object: An object behavioral pattern for concurrent programming. In *Pattern Languages of Program Design 2 (PLoP'95)*, pages 483–499. Addison-Wesley, 1996. ISBN 0-201-895277.

[24] J. McAffer. Meta-level programming with CodA. In M. Tokoro and R. Pareschi, editors, *ECOOP '95 — Object-Oriented Programming: 9th European Conference*, pages 190–214, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg. doi:10.1007/3-540-49538-X_10.

[25] Microsoft Corp. Asynchronous programming with async and await (C# and Visual Basic), 2015. URL `https://msdn.microsoft.com/en-us/library/hh191443.aspx`.

[26] G. Milicia and V. Sassone. Jeeg: A programming language for concurrent objects synchronization. In *JGI'02: Proceedings of the 2002 Joint ACM-ISCOPE Conference on Java Grande*, pages 212–221. ACM, 2002. doi:10.1145/583810.583834.

[27] M. S. Miller, E. D. Tribble, and J. Shapiro. Concurrency among strangers: Programming in E as plan coordination. In *Trustworthy Global Computing: International Symposium, TGC 2005: Revised Selected Papers*, volume 3705, pages 195–229. Springer, 2005. doi:10.1007/11580850_12.

[28] M. Odersky. The Scala language specification: Version 2.9, June 2014. URL `http://www.scala-lang.org/docu/files/ScalaReference.pdf`.

[29] M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A nominal theory of objects with dependent types. In *ECOOP 2003 – Object-Oriented Programming: 17th European Conference: Proceedings*, volume 2743 of *Lecture Notes in Computer Science*, pages 201–224. Springer, 2003. doi:10.1007/978-3-540-45070-2_10.

[30] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the Scala programming language. Technical Report IC/2004/64, EPFL, Lausanne, Switzerland, 2004.

[31] X. Qi and A. C. Myers. Masked types for sound object initialization. In *POPL'09: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 53–65, New York, NY, USA, 2009. ACM. doi:10.1145/1480881.1480890.

[32] J. Schäfer and A. Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In *ECOOP 2010 – Object-Oriented Programming: 24th European Conference: Proceedings*, volume 6183 of *Lecture Notes in Computer Science*, pages 275–299. Springer, 2010. doi:10.1007/978-3-642-14107-2_13.

[33] C. Scholliers, É. Tanter, and W. De Meuter. Parallel actor monitors: Disentangling task-level parallelism from data partitioning in the actor model. *Sci. Comput. Program.*, 80:52–64, Feb. 2014. doi:10.1016/j.scico.2013.03.011.

[34] Y. Selivanov. PEP 492 — Coroutines with async and await syntax, 2015.

[35] G. Smolka, M. Henz, and J. Würtz. Object-oriented concurrent constraint programming in Oz. In *Grundlagen und Anwendungen der Künstlichen Intelligenz: 17. Fachtagung für Künstliche*

Intelligenz, pages 44–59. Springer, 1993. doi:10.1007/978-3-642-78545-0_3.

[36] D. Syme, T. Petricek, and D. Lomov. The F# asynchronous programming model. In *Practical Aspects of Declarative Languages: 13th International Symposium, PADL 2011: Proceedings*, volume 6539 of *Lecture Notes in Computer Science*, pages 175–189. Springer, 2011. doi:10.1007/978-3-642-18378-2_15.

[37] J. A. Thywissen, A. M. Peters, and W. R. Cook. Implicitly distributing pervasively concurrent programs: Extended abstract. In *First Workshop on Programming Models and Languages for Distributed Computing*, PMLDC '16, pages 1:1–1:4, New York, NY, USA, 2016. ACM. doi:10.1145/2957319.2957370.

[38] Typesafe Inc. Akka, 2016. URL `http://akka.io/`.

[39] S. Vinoski. Reliability with Erlang. *IEEE Internet Computing*, 11(6):79–81, Nov. 2007. doi:10.1109/MIC.2007.132.

[40] A. Yonezawa. A reflective object oriented concurrent language ABCL/R. In T. Ito and R. H. Halstead, editors, *Parallel Lisp: Languages and Systems: US/Japan Workshop on Parallel Lisp: Proceedings*, pages 254–256, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg. doi:10.1007/BFb0024158.

[41] A. Yonezawa, J.-P. Briot, and E. Shibayama. Object-oriented concurrent programming in ABCL/1. In *OOPSLA '86: Object-oriented Programming Systems, Languages, and Applications: Conference Proceedings*, pages 258–268, New York, 1986. ACM. doi:10.1145/28697.28722.

A. Formal Semantics

We have extended the formal semantics of Orc without objects [21] to include the additional features of OrcO. The semantics are included here to demonstrate, through their relative brevity, that OrcO's expressiveness does not come at the cost of high complexity. The formal semantics are divided into two parts: an internal semantics, in Figure 11, which describes the behavior of OrcO programs, and an external semantics, in Figure 13, which describes how OrcO programs interact with the global store and the external environment. Figure 10 defines the abstract syntax of each of the constructs used in the internal and external semantics. Expression halting is formally defined in Figure 12 and used in the internal semantics.

A.1 Internal Semantics

The internal semantics defines how an OrcO expression f transitions to a new expression f' by emitting the label l; denoted $f \xrightarrow{l} f'$. The most important of these labels is $!v$, a publication of the value v, with which many of the combinators interact. Labels other than publications are denoted by n and are emitted by expressions in the internal semantics, but do not interact with OrcO expressions. Instead, the (PROPAGATE) rule propagates these labels up to the top-level expression, where they are interpreted by the external semantics. Expressions can transition to a new form without side effects using a special label τ.

$$
\begin{array}{rcll}
x, y, z & \in & \textit{Variable} & \\
V & \in & \textit{Value} & \\
k & \in & \textit{Handle} & \\
q & \in & \textit{Object} & \\
\\
F & \in & \textit{Function} & ::= \texttt{def}\ y(\bar{x}) = f \\
D & \in & \textit{Declaration} & ::= F \\
& & & \mid \texttt{val}\ x = f \\
\\
f, g & \in & \textit{Expression} & ::= p \qquad \text{Bare parameter} \\
& & & \mid p.x \qquad \text{Object member} \\
& & & \mid p(\bar{p}) \qquad \text{Call} \\
& & & \mid k? \qquad \text{Call handle} \\
& & & \mid f \mid g \qquad \text{Parallel} \\
& & & \mid f \gg x \gg g \qquad \text{Branch } x \text{ to } g \\
& & & \mid f \ll x - g \qquad \text{Graft } x \text{ from } g \\
& & & \mid f\ ;\ g \qquad \text{Otherwise} \\
& & & \mid \{\!|\ f\ |\!\} \qquad \text{Trim} \\
& & & \mid D \,\#\, f \qquad \text{Scoped declaration} \\
& & & \mid k \triangleleft p \qquad \text{Store} \\
& & & \mid \texttt{new}\ \{z\ \bar{D}\} \qquad \text{New object} \\
\\
v & \in & \textit{Orc value} & ::= V \mid F \mid q \\
w & \in & \textit{Response} & ::= v \mid \texttt{stop} \\
p & \in & \textit{Parameter} & ::= w \mid x \\
\\
& & \textit{Future} & ::= w \mid k \\
\Sigma & \in & \textit{Variable} \to \textit{Future} & \\
\\
n & \in & \textit{Non-pub Label} & ::= k \triangleright V(\bar{v}) \\
& & & \mid k \triangleleft w \\
& & & \mid \texttt{new}\ \Sigma \\
& & & \mid \tau \\
l & \in & \textit{Label} & ::= !v \mid n \\
\\
E & \in & \textit{Execution context} & ::= \square \\
& & & \mid E \mid g \\
& & & \mid f \mid E \\
& & & \mid E \gg x \gg g \\
& & & \mid E \ll x - g \\
& & & \mid f \ll x - E \\
& & & \mid E\ ;\ g \\
& & & \mid \{\!|\ E\ |\!\} \\
& & & \mid D \,\#\, E \\
\\
& & E[f] & \equiv [\square \mapsto f]E
\end{array}
$$

Figure 10. Formal syntax of OrcO.

$$\frac{f \xrightarrow{n} f'}{E[f] \xrightarrow{n} E[f']} \quad \text{(PROPAGATE)}$$

$$v \xrightarrow{!v} \mathbf{stop} \quad \text{(PUBLISH)}$$

$$\frac{k \text{ fresh}}{V(\bar{v}) \xrightarrow{k \triangleright V(\bar{v})} k?} \quad \text{(SITECALL)}$$

$$\frac{F = \mathbf{def}\ y(\bar{x}) = g \qquad FV(F) = \emptyset}{F \# f \xrightarrow{\tau} [y \mapsto F]f} \quad \text{(DEFCREATE)}$$

$$\frac{F = \mathbf{def}\ y(\bar{x}) = g}{F(\bar{p}) \xrightarrow{\tau} [y \mapsto F][\bar{x} \mapsto \bar{p}]g} \quad \text{(DEFCALL)}$$

$$(\mathbf{val}\ x = g) \# f \xrightarrow{\tau} f \lessdot x - g \quad \text{(VAL)}$$

$$\frac{f \xrightarrow{!v} f'}{f \mid g \xrightarrow{l} f' \mid g} \quad \text{(PARL)}$$

$$\frac{g \xrightarrow{!v} g'}{f \mid g \xrightarrow{l} f \mid g'} \quad \text{(PARR)}$$

$$\frac{f \xrightarrow{!v} f'}{f \gg x \ll g \xrightarrow{\tau} f' \gg x \ll g \mid [x \mapsto v]g} \quad \text{(BRANCH)}$$

$$\frac{f \xrightarrow{!v} f'}{f \lessdot x - g \xrightarrow{!v} f' \lessdot x - g} \quad \text{(GRAFTL)}$$

$$\frac{g \xrightarrow{!v} g'}{f \lessdot x - g \xrightarrow{\tau} [x \mapsto v]f \mid \mathbf{stop} \lessdot x - g'} \quad \text{(GRAFTV)}$$

$$\frac{g \text{ halted}}{f \lessdot x - g \xrightarrow{\tau} [x \mapsto \mathbf{stop}]f} \quad \text{(GRAFTSTOP)}$$

$$\frac{f \xrightarrow{!v} f'}{f \,;\, g \xrightarrow{!v} f'} \quad \text{(OTHERV)}$$

$$\frac{f \text{ halted}}{f \,;\, g \xrightarrow{\tau} g} \quad \text{(OTHERSTOP)}$$

$$\frac{f \xrightarrow{!v} f'}{\{\!|\, f \,|\!\} \xrightarrow{!v} \mathbf{stop}} \quad \text{(TRIM)}$$

$$\frac{\begin{array}{c}\text{dom}(\bar{D}) = \{y_0, \ldots, y_n\} \\ \{k_0, \ldots, k_n\} \text{ fresh} \\ q \text{ fresh} \\ f = \bar{D} \# (k_0 \triangleleft y_0 \mid \ldots \mid k_n \triangleleft y_n) \\ \Sigma = \{q.y_0 \mapsto k_0, \ldots, q.y_n \mapsto k_n\}\end{array}}{\mathbf{new}\ \{z\ \bar{D}\} \xrightarrow{\mathbf{new}\ \Sigma} q \mid [z \mapsto q]f} \quad \text{(NEW)}$$

$$k \triangleleft w \xrightarrow{k \triangleleft w} \mathbf{stop} \quad \text{(STORE)}$$

Figure 11. Internal semantics of OrcO.

$$\frac{f \text{ halted} \qquad D \neq \mathbf{val}\ x = g}{D \# f \text{ halted}} \quad \text{(HALTDEF)}$$

$$\frac{f \text{ halted} \qquad g \text{ halted}}{f \mid g \text{ halted}} \quad \text{(HALTPAR)}$$

$$\frac{f \text{ halted}}{f \gg x \ll g \text{ halted}} \quad \text{(HALTBRANCH)}$$

$$\frac{f \text{ halted}}{\{\!|\, f \,|\!\} \text{ halted}} \quad \text{(HALTTRIM)}$$

$$\mathbf{stop}(\bar{p}) \text{ halted} \quad \text{(HALTCALL)}$$

$$V(\ldots, \mathbf{stop}, \ldots) \text{ halted} \quad \text{(HALTARG)}$$

$$\mathbf{stop}.x \text{ halted} \quad \text{(HALTMEMBER)}$$

$$\mathbf{stop} \text{ halted} \quad \text{(HALTSTOP)}$$

Figure 12. Definition of halted expressions.

The rule (SITECALL) implements site calls by notifying the environment and waiting for a response. The rule emits a site call label, attaching a unique handle k to the label that indicates where the response to the site call should go. The call transitions to the expression $k?$, which waits for the response. Note that this rule only applies if the entity being called is an external value V; functions F are handled by (DEFCALL), and objects q cannot be called. The arguments to the call must be values as site calls are strict.

The rules (DEFCREATE) and (DEFCALL) implement function definition and invocation. (DEFCREATE) takes a scoped function definition and turns it into a closure F, substituting the closure for each occurrence of the defined name y in the scoped expression. (DEFCALL) calls a closure, replacing the call expression with the function body, replacing the arguments $\bar{x}$ with the call parameters $\bar{p}$ and replacing the function name y with a copy of the closure itself to allow recursion. Unlike site calls, function calls are lenient: the arguments to a function call may be any parameters p, including unbound variables or **stop**.

The rule (VAL) defines the semantics of the **val** definition by transforming it into a graft combinator. Normally this would be syntactic sugar, but because of the way objects are interpreted in the semantics, it must be transformed explicitly using a transition rule.

The eight combinator rules define how publication and halting are handled inside subexpressions. The rules (PARL) and (PARR) allow publications to propagate from either side of a parallel combinator. (BRANCH) captures a publication $!v$ from the left side of a branch combinator, consumes the publication, and creates a parallel copy of the right expression g where x has been replaced by v. (GRAFTL) allows publications to propagate from the left side of a graft combinator. The rule (GRAFTV) captures a publication $!v$ from the right side of a graft combinator, consumes the publication, and then splits the graft combinator into two parallel expressions: the left expression with x replaced by v, and the right expression with all subsequent publications suppressed by **>> stop**. The rule (GRAFTSTOP) detects that the right side of a graft combinator has halted with no publications, discards the right side, and replaces x with **stop** in the left side to indicate that the variable x will never be bound. The rule (OTHERV) drops the right side of an otherwise combinator when the left side publishes. The rule (OTHERSTOP) executes the right side of an otherwise combinator when the left side halts. The rule (TRIM) replaces the trim combinator with **stop** (terminating all subexpressions) when the contained expression publishes, and emits the publication.

The object semantics are defined using rules for object instantiation, and field binding. Classes are defined in Section A.3 as a translation to pure objects. The rule (NEW) instantiates an object defined by a set of declarations. It determines the field names y_i of the object to be created; these are just the domain of the definitions $\bar{D}$. It then creates a unique handle k_i corresponding to each field y_i, and a unique object value q to represent the new object. The instantiation emits the object store Σ, and transitions to the object value q, with the object body f executing in parallel. The object body expression f waits, in parallel, for each defined field y_i to become bound, and stores its value to the corresponding future k_i. In f, all occurrences of the object's self reference z have been replaced by q. The object store Σ associates each field on the object with its corresponding future. (STORE) handles a resolved field value by emitting a storage event when a response w is available to be stored to a handle k. This storage event will replace k with w in the global store.

A.2 External Semantics

The external semantics defines how the environment η, the global store Σ, and the OrcO expression f all transition to their new counterparts η', Σ', and f'; denoted $\eta, \Sigma, f \hookrightarrow \eta', \Sigma', f'$. Unlike the internal semantics, this is not a labeled transition. The external semantics rules are given in Figure 13.

The store Σ is a map from a field of an object to a either a response w if the field is resolved, or a handle k if it is not resolved. The environment η is an abstract representation of all behavior outside of the OrcO program.

Five rules allow the internal semantics to execute the OrcO program and use the label to update the environment and store. The rule (STEP) allows an expression to make a silent transition. This has no effect on the environment or store. (STEPV) allows an expression to make a transition and emit a publication. The publication is added to the environment, perhaps as console output, but it has no other effect. (CALL) allows an expression to send a site call to the environment. (ALLOC) allows an expression to append a newly instantiated object Σ', represented as a store containing its fields, to the global store. (STORE) allows an expression to bind k to a value, replacing all occurrences of k in the store with w.

The rules (MEMBER) and (RETURN) transfer information from the environment and store into the program. (MEMBER) allows an expression to retrieve a resolved future from the store by substitution into the expression. The execution context E allows the substitution to occur on any subexpression that is currently executing. (RETURN) allows the environment to report the result of a site call by mapping its associated handle k to some response w. The expression accepts this response from the environment by replacing $k?$ with w.

The rule (EXTERNAL) allows the environment to make arbitrary transitions of its own. This represents the externally implemented semantics of sites.

The rule (NOSENDER) allows the store to detect that a handle k referenced in the store has disappeared from the program. This occurs when an expression containing an executing object is trimmed before all of the object's futures are resolved. In this case, the handle k resolves to **stop** in the store, so that field accesses do not block forever on an object that is no longer executing.

$$\frac{f \xrightarrow{\tau} f'}{\eta, \Sigma, f \hookrightarrow \eta, \Sigma, f'} \quad \text{(STEP)}$$

$$\frac{f \xrightarrow{!v} f'}{\eta, \Sigma, f \hookrightarrow \eta \cup \{!v\}, \Sigma, f'} \quad \text{(STEPV)}$$

$$\frac{f \xrightarrow{k \triangleright V(\bar{v})} f'}{\eta, \Sigma, f \hookrightarrow \eta \cup \{k \triangleright V(\bar{v})\}, \Sigma, f'} \quad \text{(CALL)}$$

$$\frac{\eta(k) = w}{\eta, \Sigma, f \hookrightarrow \eta, \Sigma, [k? \mapsto w]f} \quad \text{(RETURN)}$$

$$\frac{\eta \to \eta'}{\eta, \Sigma, f \hookrightarrow \eta', \Sigma, f} \quad \text{(EXTERNAL)}$$

$$\frac{f \xrightarrow{\texttt{new}\ \Sigma'} f'}{\eta, \Sigma, f \hookrightarrow \eta, \Sigma \cup \Sigma', f'} \quad \text{(ALLOC)}$$

$$\frac{f \xrightarrow{k \triangleleft w} f'}{\eta, \Sigma, f \hookrightarrow \eta, [k \mapsto w]\Sigma, f'} \quad \text{(STORE)}$$

$$\frac{\Sigma(q.x) = w}{\eta, \Sigma, E[q.x] \hookrightarrow \eta, \Sigma, E[w]} \quad \text{(MEMBER)}$$

$$\frac{k \in FV(\Sigma) \qquad k \notin FV(f)}{\eta, \Sigma, f \hookrightarrow \eta, [k \mapsto \texttt{stop}]\Sigma, f} \quad \text{(NOSENDER)}$$

Figure 13. External semantics of OrcO.

$$\begin{array}{lcl}
\mathcal{FF}(\langle\rangle, z) & = & \texttt{new}\ \{w\ \} \\
\mathcal{FF}(\langle L, C\{\bar{D}\}\rangle, z) & = & \texttt{new}\ \{w\ \texttt{val}\ s = \mathcal{FF}(L, z) \,\#\, \bar{S}(L, \bar{D}, s) \,\#\, [\texttt{super} \mapsto s][\texttt{self} \mapsto z]\bar{D}\ \} \\
\mathcal{FC}(\langle L, C\{\bar{D}\}\rangle) & = & \texttt{def}\ \texttt{newC}() = \texttt{new}\ \{z\ \texttt{val}\ s = \mathcal{FF}(L, z) \,\#\, \bar{S}(L, \bar{D}, s) \,\# \\
 & & \qquad [\texttt{new}\ \texttt{C} \mapsto \texttt{newC}()][\texttt{super} \mapsto s][\texttt{self} \mapsto z]\bar{D}\ \} \\
\bar{S}(L, \bar{D}, s) & = & \{\texttt{val}\ x = s.x \text{ where } x \in \mathrm{dom}(L) \wedge x \notin \mathrm{dom}(\bar{D})\} \\
 & & \qquad \text{where } \mathrm{dom}(L) = \bigcup_{C\{\bar{D}'\} \in L} \mathrm{dom}(\bar{D}')
\end{array}$$

Figure 14. Translation functions for OrcO classes.

A.3 Class Semantics

The underlying OrcO calculus does not support inheritance or mixins. Instead, classes are converted into constructor functions which create instances based on the linearization of the class. The **super** reference is a specialized instance of the superclass which uses **self** from the subclass object instead of its own **self**. This technique would not work for sequential objects, because superclass initialization and subclass initialization would be ordered preventing the superclass from observing the subclass initialization or visa-versa. OrcO objects are initialized concurrently, so the superclass and subclass can block on any fields they need without affecting the rest of the initialization.

Formally, class inheritance in OrcO is defined as a translation from surface language class constructs (Figure 1) into the abstract syntax (Figure 10). The translation replaces each class definition **class** C **{** ... **}** with a function newC and each instantiation **new** C with a call newC().

A class E expression can be one of three forms: a subclass C **extends** E $\{\bar{D}\}$ where C is a class name, two mixed classes E_1 **with** E_2, or the empty class O. A surface language class **class** C **extends** E **{** ... **}** is translated to a subclass C **extends** E $\{\ldots\}$. A surface language class with no subclass is treated as a subclass of O. Mixins translate directly.

A class expression E is translated into a sequence of sets of fields labeled with the class in which they are declared. This sequence is called the linearization $\mathcal{L}(E)$ and written as $\langle C_1\{\bar{D}_1\}, C_2\{\bar{D}_2\}, \ldots\rangle$. This is a refinement of the linearization function in Section 3. $\mathcal{L}(E)$ is defined as a recursive function on class expressions:

$$\begin{array}{lcl}
\mathcal{L}(O) & = & \langle\rangle \\
\mathcal{L}(E_1\ \texttt{with}\ E_2) & = & \mathcal{L}(E_1) \overleftarrow{+} \mathcal{L}(E_2) \\
\mathcal{L}(C\ \texttt{extends}\ E\ \{\bar{D}\}) & = & \mathcal{L}(E) \overleftarrow{+} \langle C\{\bar{D}\}\rangle
\end{array}$$

Here $A \overleftarrow{+} B$ represents the concatenation operation where any elements of B that already appear in A are omitted.

$$\begin{array}{lcll}
A \overleftarrow{+} \langle B, b\rangle & = & \langle (A \overleftarrow{+} B), b\rangle & \text{if } b \notin A \\
 & = & A \overleftarrow{+} B & \text{if } b \in A
\end{array}$$

As an example, the class **class** C **extends** A **with** B **{ }** (assuming A and B have no superclasses) has linearization $\langle A\{\ldots\}, B\{\ldots\}, C\{\}\rangle$.

From a linearization, we can build an instantiation expression in the OrcO calculus. Building this expression takes two functions defined in Figure 14: $\mathcal{FF}(\langle L, C\{\bar{D}\}\rangle, z)$ (flatten fragment) translates a linearization (where L is the prefix of the linearization) into an expression that instantiates a flatten variant of C and all classes in L with a given self reference z; and $\mathcal{FC}(\texttt{C})$ (flatten class) which performs the same translation using its own self reference and building a recursive function. $\bar{S}(L, \bar{D}, s)$ is an auxiliary function (also defined in Figure 14) which constructs a set of fields which forwards every field in L which is not present in $\bar{D}$. The function $\mathcal{FF}$ recursively builds a chain of nested superclass instances all sharing the given self reference. $\mathcal{FF}$ does not use the self reference w of the object it creates. The function $\mathcal{FC}$ builds a recursive class constructor, using $\mathcal{FF}$ to construct the super reference. Each object forwards any fields it does not define to its super reference.

The translation for a surface language expression `class C extends E { `$\bar{D}$` } # e` is:

$$\mathcal{FC}(\mathcal{L}(C\ \texttt{extends}\ E\{\bar{D}\}))\ \#\ [\texttt{new C} \mapsto \texttt{newC}()]e$$

with any class names in E resolved to the appropriate class expressions. This translation supports recursive classes by using recursive functions.