Table of Contents

Chapter 1. Introducing Orc

Chapter 2. The Orc Programming Language

In this chapter, we describe the full capabilities of the Orc programming language. This is intended to be a comprehensive reference for the Orc programmer.

Introduction

Orc more strongly resembles functional programming languages such as Haskell and ML than imperative languages such as Java and C. The reader is assumed to have at least a passing familiarity with the concept of variable binding (as opposed to variable assignment), as encountered in functional languages, as well as lexical scope. It may also help to be familiar with the concepts of lexical closure and pattern matching.

An Orc program is a structured expressions which does computation, invokes services, and then *publishes* some number of values. Publishing a value is similar to returning a value in other languages, except that an expression may publish multiple times, or it might never publish at all. An expression which never publishes is called *silent*.

An Orc program is built up from base expressions, the simplest programs. These are connected by combinators to form larger expressions. Such expressions may also be enclosed by declarations which define functions, compute values, or introduce new sites. Orc expressions are always compositional: two expressions joined by a combinator form an expression, and an expression prefixed by a declaration also forms an expression.

Base Expressions

An Orc program is built up from base expressions. A base expression on its own is always a valid Orc program.

Constants

The simplest program one can write is a constant value. Orc supports the following set of constants:

- Integer constants: `( ... -1, 0, 1 ... )`
- Booleans: `true` and `false`
- Strings: `"orc"`, `"ceci n'est pas une |"`
- A special value `signal`, which carries no information (analagous to the unit value `()` in ML).

When evaluated, a constant simply publishes that value immediately.

Variables

An identifier on its own is a valid expression; it represents a variable. It waits for that variable to be bound, and then publishes that bound value. If the variable is already bound when the expression is evaluated, its bound value is published immediately. The expression never publishes if the variable never becomes bound.

Call

An identifier followed by a parenthesized sequence of *arguments* is a call. A call may have zero arguments, in which case we write an identifier followed by empty parens `()`. The identifier is a variable which names the value that will be called. For now, we assume that the arguments are either variables or constants.

A call does nothing until the identifier is bound to a value (the *target*). Once the target is known, the behavior of the call depends on whether the target is a site or a defined function.

- If the target is a site, the call subsequently waits for each variable argument to be bound; we say that calls to sites are *strict* since they must wait for all of their arguments to be bound. Once the values of all of the arguments are known (constants are always known and variables are known once they have bound values), the service associated with that site is invoked with those values. The call waits for the service to respond with a value. If the service responds, the call publishes that value. The call acknowledges at most one response from the service, so at most one publication occurs.
- If the target is a defined function, the call is replaced by the body of the function, where each of the call's arguments is substituted for the function's formal parameters in the body (even if those arguments are unbound variables). We say that function calls are *lenient* because they do not need to wait for their arguments to be bound.

Note

Any Orc expression can be used as an argument; see the pull combinator [5] for more details.

Operators

Orc supports a standard set of arithmetic, logical, and comparison operators. They are written in the usual infix style, and have C-like precedence.

Arithmetic		**Comparison**		**Logical**	
`+`	addition	`=`	equality	`&&`	logical and
`-`	subtraction	`/=`	inequality	`||`	logical or
`*`	multiplication	`<`	less than	`~`	logical not
`/`	division	`>`	greater than		
`%`	modulus	`<=`	less than or equal		
		`>=`	greater than or equal		

These operators are actually sites, and behave like sites; they differ only in syntax. Accordingly, any Orc expression can be used as an operand; it undergoes the same translation [5] as an argument to a call.

If

The expression `if(b)`, where `b` is some boolean expression, will publish a signal if `b` evaluates to true, and otherwise remains silent. `if` is used together with the push combinator to form conditional expressions.

`if` is a site; it is called a *fundamental site* because it is always available.

Data structures

Orc supports two basic data structures:

- A *tuple* is a comma-separated sequence of arguments enclosed by parentheses. It looks like a call without a preceding identifier: `(x,2,y)`. A tuple must have at least two arguments. Like a site call, a tuple waits for each of its arguments to become bound. Then, it publishes a tuple value containing each of those bound values.
- A *list* is a comma-separated sequence of arguments enclosed by square brackets: `[2,11,0]`. It may be of any length, including zero; the empty list is written `[]`. A list can also be extended using the *cons* operation, written `h:t`, which publishes a new list whose first element is `h` and whose tail is `t`.

Again, as with operators, each data structuring operation is actually a site call.

To learn more about inspecting and using these data structures after they have been created, read about patterns.

Messages

In many object-oriented programming languages, one calls a method or accesses a field of an object using the dot operator; for example, `obj.m()` calls the method `m` of the object `obj`.

There is a special kind of call in Orc which serves a similar purpose. One may write `x.msg`, for any identifiers `x` and `msg`. This attempts to send the *message* 'msg' to the value bound to `x`. The message may not be understood, in which case no publication occurs.

Typically this capability is used so that sites may be treated like objects, with multiple methods and fields. For example, a channel `c` might understand the messages `get` and `put`, to get values from and put values on that channel, respectively. Such calls would be written `c.get()`, or `c.put(6)`.

> **Note**
> Such calls actually occur in two steps: first `c.put` sends the message `put` to the value `c`; this publishes a site whose only purpose is to put values to that channel. Then, that 'put site' is called on the argument `6`, sending 6 on that channel. Readers familiar with functional programming will recognize this technique as *currying*.

Stop

`stop` is a special base expression which is silent and does no computation. It is typically used together with a push to silence the publications of another expression.

Combinators

Bar (|)

The bar combinator `|` executes two expressions in parallel, and publishes each of their publications as they occur. It is associative and commutative.

Push (>>)

The push combinator `>x>` executes its left side; for each publication on the left side, a new copy of the right side starts, with the variable `x` bound to that published value.

A push can be written as `>>`, with no variable name, which behaves similarly except that no variable binding occurs.

The variable name within the combinator may be replaced with an arbitrary pattern (see Patterns).

Pushing is right-associative, and it has higher precedence than bar.

Pull (<<)

The pull combinator executes its left and right sides in parallel. Calls using `x` on the left side block until it is bound. The first publication of the right side is bound to `x`; this causes the rest of the right side to stop executing. That expression may not evaluate further; any site calls already in progress will continue, but their return values will be ignored.

A pull can be written as `<<`, with no variable name, which behaves similarly except that no variable binding occurs.

The variable name within the combinator may be replaced with an arbitrary pattern (see Patterns).

Pulling is left-associative, and it has lower precedence than bar.

A call may be written with complex expressions as arguments. This is actually a shorthand for using a pull to evaluate each of those expressions and bind its result to a variable. For example,

```
M(x+5, 4, N() | R())
```

is just another way of writing

```
M(y, 4, z)
 <y< x+5
 <z< N() | R()
```

Thus, all of the rules governing pull apply to using an expression as an argument: it may publish at most one value, and it terminates when it publishes a value.

Before (;)

The before combinator executes its left side, publishing each of its publications as they occur. When the left side has completely finished executing (i.e. it is equivalent to `stop`), then the right side executes.

Note

The before combinator is intended to capture as closely as possible the notion of sequential processing, as denoted by `;` in other languages. It was not present in the original formulation of the Orc concurrency calculus; it has been added to support computation and iteration over strictly finite data.

Patterns

A *pattern* is a special construct that may take the place of a variable binding. It examines the structure of a published value, rather than simply binding that value to a single variable. It may *match* the value, capturing components of the value, and then bind those component values to variables or publish them. Or, it may *refuse* the value, silencing that publication entirely. A pattern is called *refutable* if it can refuse a value; otherwise it is *irrefutable*.

Note

Whenever possible, the structure of a pattern mimics the structure of an expression that would publish a value recognized by that pattern. For example, the pattern `(x,[y,z])` matches the

value published by the expression `(true,[3,4])`. However, the converse is not true: if a pattern matches a value, the source of that value is not necessarily an expression with the same structure. This is particularly true when views are used.

Syntax

This section describes the syntax of patterns. All patterns are assumed to be refutable unless stated otherwise.

Variable

Variables are the most basic patterns. A variable pattern simply binds the matched value to that variable. It is an irrefutable pattern.

Important

Patterns must be *linear*, meaning that a pattern may not mention the same variable more than once.

Wildcard

A wildcard pattern, written _, matches any value, and does not bind any variables. It is the same as binding a variable which is not used anywhere. Like variables, wildcards are irrefutable.

Note

The empty push (>>) and empty pull (<<) combinator forms are just shorthand for (>_>) and (<_<) respectively.

Literal

A literal pattern can be any constant value. It will only match that value, and will refuse any others.

Tuple

A tuple of patterns matches only a tuple value of exactly the same size; it refuses any other value. It recursively matches the elements of the tuple against its own member patterns. A tuple pattern must contain at least two subpatterns.

For example, the expression

```
( (1,2) | (1,3) | (5,4) )  >(1,x)>  x
```

will publish `2` and `3` but not `4`, because the pattern `(1,x)` refuses the value `(5,4)`.

List

A list of patterns matches a list value of exactly the same length, refusing any other value. It recursively matches each item of the list value against each of its member patterns. A list pattern may have any number of subpatterns, including zero.

For example, the expression

```
( [4] | [5,6] | [7,8,9] )  >[x,y]>  x+y
```

publishes only `11`; the pattern refuses the other two lists because they are of the wrong length.

Cons

It is also possible to use the cons (`:`) operator in a pattern, to split a list into its head and tail.

```
[1,2,3]  >h:t>  ( h | t )
```

This publishes `1` and `[2,3]`. Note that a cons pattern refuses the empty list.

Bang

A bang pattern, written `!p`, will publish the value that matches the pattern `p` if the match is successful. This pattern is refutable only if `p` is refutable.

```
(1,2,3)  >(x,!y,!z)> stop
```

This publishes `2` and `3`.

Note that a bang pattern will not publish if the overall match fails, even if its particular match succeeds:

```
( (1,2,3) | (4,5,6) )  >(1,!x,y)>  stop
```

This publishes only `2`. Even though the pattern `x` matches the value `5`, the overall pattern `(1,!x,y)` refuses the value `(4,5,6)`, so `5` is not published.

As

The pattern `p as x` can be used to capture a whole subpattern `p` and bind it to the variable `x`. This pattern is refutable only if `p` is refutable.

```
(1,(2,3))  >(x,(2,z) as w)>  w
```

This publishes `(2,3)`.

Site

A site pattern, written `M(p,...,p)`, matches any value which was published by a call to the site `M` with arguments that match the tuple `(p,...,p)`.

Site patterns provide a generalized version of datatype matching, as seen in the `case .. of` or `match .. with` constructs provided by Haskell and ML.

Occurrences

These are the syntactic contexts in which a pattern may replace a normal variable binding. The effect of a pattern may be subtly different depending on the syntactic context in which it appears.

In a push

A pattern may replace the variable name in a push combinator. When the left side publishes a value, that value is checked against the pattern. If it matches, a new copy of the right hand side starts with all of the pattern's variables bound in it. If the pattern refuses the value, that publication is ignored; no new copy of the right hand side is created.

In a pull

A pattern may replace the variable name in a pull combinator. When the right side publishes a value, that value is checked against the pattern. If it matches, the pattern's variables become bound in the left side and the right side terminates, as expected. However, if the pattern refuses the value, then the publication is ignored and the right side continues to execute as normal.

In a `val` declaration

A pattern may replace the variable name in a `val` declaration. Since a val is just a shorthand for writing a pull, its behavior is exactly the same: whenever the expression publishes a value, it is checked against the pattern; if the pattern refuses, the publication is ignored and the expression continues executing, whereas if if matches, the pattern's variables are bound and the expression terminates.

As function arguments

A pattern may replace any formal parameter in a function clause. Whenever that function is called, its arguments are matched against the argument patterns of each clause in linear order. If every argument pattern of a clause matches, then the patterns' variables are bound in that clause's body and the body executes. Otherwise, the next clause is tried.

> **Note**
> Since function calls in Orc are not strict, some of the arguments may not be bound when they are matched against a pattern. If an unbound argument is matched against a *refutable* pattern, then that match waits for the variable to become bound, since it is impossible to tell if the pattern will refuse without knowing the bound value.

Patterns as views

In addition to matching data structures constructed within Orc, patterns can also be used to match external data structures and values. In this setting, a pattern does not necessarily observe the true structure of the data, but instead provides a *view* of that data.

For example, a list pattern can be used to view some sequentially structured data (such as lines in a file) as if it were a list. The views allowed on a value are determined by the value itself; for example, a value which does not support a list view will be refused by any list pattern it is matched against.

Declarations

In addition to using combinators, Orc expressions can also be built up by adding declarations. A declaration is a directive that precedes an expression. It does not publish any values on its own; instead introducing one or more new identifiers so that they can be used in that expression. The expression following a declaration is called the *scoped expression* of that declaration.

Unless specifically stated otherwise, the scoped expression begins executing immediately, without waiting for the declaration to do any computation. If the scoped expression reaches an identifier introduced by that declaration before the declaration has given it a value, the identifier is considered unbound.

Val

The simplest declaration is a value declaration. It binds the first publication of an expression to a variable. It is written:

`val x =` *expr*

This evaluates the expression *expr* until it publishes a value, and then binds that value to the variable *x* and terminates *expr*. It is a shorthand for a pull combinator; if the scoped expression is *g*, then writing this declaration is exactly the same as writing:

g `<x<` *expr*

Functions

Introduction

Like most other programming languages, Orc has the capability to define *functions*, which take arguments and are invoked in the same way as sites. Functions are defined using the keyword `def`, in the following way:

`def F(`*args*`) =` *expr*
`F` is the name of the function. It may have any number of arguments; if it has zero arguments, empty parentheses are still written `()`. *expr* is the body of the expression.

For example, the expression

```
def E(x,y) = x | y | x+y
E(2,3)
```

publishes the values 2, 3, and 5.

Leniency

As discussed in the section on calls, functions are lenient, or 'call-by-name'. Function calls do not need to wait for all of their arguments to be bound:

```
E(a,b)
    <a< stop
    <b< 4
```

This will publish 4, even though the variable `a` is never bound. The call `E(a,b)` occurs immediately, effectively executing the expression `a | b | a+b` in its place.

Recursion

Definitions can be recursive; that is, the name of a definition is bound in its own body.

```
def countdown(n) = if(n > 0) >> ( n | countdown(n-1) )
countdown(3)
```

This publishes 3, 2, and 1.

Mutual Recursion

Mutual recursion is also supported:

```
def even(n) =
  if(n > 0) >> odd(n-1)
| if(n < 0) >> odd(n+1)
| if(n = 0) >> true
def odd(n) =
  if(n > 0) >> even(n-1)
| if(n < 0) >> even(n+1)
| if(n = 0) >> false
```

There is no special keyword for mutual recursion; any contiguous sequence of definitions is assumed to be mutually recursive.

Closure

Definitions are actually values, just like any other value. Defining an expression creates a special value called a *closure* and binds it. Thus, it can be published, or put into a data structure, like any other value:

```
f(1) | f(3)
<f<
(def E(x) = x+1
 E)
```

This publishes 2 and 4. It defines the expression `E`, then publishes that definition as a closure, which is then bound to the variable `f` and called twice.

Lexical Closure

Like any other declaration, a function definition may appear within the context of any expression. This means that variables from that context may appear in the body of the function:

```
val pi = 3.14159
def area(r) = pi * r * r
```

Functions are also values in Orc, so the function `area` might be published, and then called in some other context. What happens to the variable `pi`? It 'remembers' the value that it had when the function `area` was defined (3.14159), and uses that value, even if `pi` has a different value in the scope where the function is called:

```
val pi = 2.71828
area(10)
```

This is called lexical closure: the body of the function is a closure because it remembers the values of all of its free variables, and the closure is 'lexical' because it occurs when the function is defined, not when it is used.

Note

Lexical closure has an interesting subtlety when combined with variables that are in scope but have not yet been bound to a value (such as pull variables). Since the closure must capture the values of each of those variables, the closure operation itself must actually wait for all of those variables to become bound. Thus, the function name itself is unbound until the closure can be created.

Anonymouns Functions

Sometimes one would like to create a closure directly without bothering to name the defined expression. There is a special keyword `lambda` for this purpose:

```
lambda (x) = x+1
{-    equivalent to (def E(x) = x+1  E)    -}
```

Patterns

A defined expression, just like the push and pull combinators, may have patterns instead of variables as its arguments:

```
def E([x,y]) = x | y
```

This definition publishes two elements of a list, but only if its argument is in fact a two-element list. Otherwise, the call remains silent. Note that using patterns as arguments interacts with the call-by-name semantics of defined expressions: if a pattern is refutable, then the body of the expression cannot execute until that argument is bound.

Clauses

The combination of functions and pattern matching offers a powerful capability: *clausal* definition of functions. We can define expressions which execute different code depending on the structure of their arguments. Here's an example:

```
def sum([]) = 0
def sum(h:t) = h + sum(t)
```

`sum(L)` publishes the sum of the numbers in the list `L`. It has two clauses: one which matches the empty list, and one which matches any nonempty list. If its argument is an empty list, it returns 0, the appropriate sum for an empty list. If the argument is a nonempty list, it adds the first element of that list to the sum of all of the other elements. In this way, it recursively finds the sum of the list.

A function may have multiple clauses, each of which has a sequence of patterns to match each formal argument, and a body expression. All clauses of a function must have the same number of arguments. Any contiguous sequence of definitions with the same name and different arguments is interpreted as a clausal definition, where each individual definition is a clause of the larger function.

When the function is called, the clauses are tried in the order in which they appear until a match is found. If no clause matches, the call remains silent.

Mutual recursion and clausal definitions are allowed to occur together. For example, this definition takes a list and publishes a new list with every other element repeated:

```
def stutter([]) = []
def stutter(h:t) = h:h:mutter(t)
def mutter([]) = []
def mutter(h:t) = h:stutter(t)
```

Services

Certain declarations are used to provide access to external services.

Sites

A `site` declaration instantiates a Java object to be used as a site in an Orc program:

```
{-  Define the Buffer site  -}
site Buffer = orc.lib.state.Buffer
```

Classes

A `class` declaration looks very similar to a `site` declaration, but serves a different purpose. Rather than creating a Java object to be used as a site, this declaration actually creates a proxy for the class's constructor, which can then be invoked, and its publications used like Java objects by using the dot syntax:

```
class Str = java.lang.String
val s = Str("foo")
s.concat("bar")
```

Thus, it makes all of the capabilities of sequential Java programming, including its large standard library, available to the Orc programmer.

Includes

An include declaration names a file containing a sequence of declarations. It loads all of those declarations in the order given in the file, as if they had been written directly into the program at that point. This is a convenient way to organize large groups of declarations.

```
{-  include some useful list functions  -}
include "inc/orc/list.inc"
```

Comments

The implementation supports two kinds of comments.

A line which begins with two dashes (`--`), preceded only by whitespace, is a single line comment. The region from the two dashes to the next encountered newline, inclusive, is ignored.

```
-- This is a single line comment.
  -- This is also a single line comment.
```

Multiline comments are enclosed by matching braces of the form `{- -}`. Multiline comments may be nested. They may appear anywhere, even in the middle of an expression.

```
{-
   This is a
   multiline comment.
-}

{- Multiline comments {- can be nested -} -}
```

```
1 | 2 {- They may appear anywhere, -} >x> {- even in the middle of an expression.
```

Commented regions are ignored by the compiler. They are often eliminated entirely by the parser, and do not appear in intermediate representations of a program.