Table of Contents

Chapter 1. The Orc Programming Language

Introduction

Orc is a programming language designed to make distributed and concurrent programs simple and intuitive to write. Orc expresses orchestration, a type of structured concurrency. It emphasizes the flow of control and gives a global view of a concurrent system. Most other concurrency-oriented languages describe the individual pieces of a system and their local interactions, but without a global view. Orc is particularly well-suited for task orchestration, a form of concurrent programming with applications in workflow, business process management, and web service orchestration. Orc provides constructs to orchestrate the concurrent invocation of services while managing time-outs, priorities, and failures of services or communication. To learn more about Orc and its community of users, visit our website: http://orc.csres.utexas.edu

This chapter describes the Orc programming language. The language is built around *expressions*. Given that `f`, `g` and `h` are Orc expressions, an expression is defined by `f ::=` base expression | `g | h` | `g >x> h` | `g <x< h` The language supports a variety of other ways of writing expressions in structured manner, but they are all based on the constructs shown above. The combinators (`|`, >x> and <x<) are concurrency combinators that allow, respectively, concurrent computations, (a general form of) sequential execution with spawning of computations and concurrency combined with computation termination. We describe the language features that support the concurrency combinators in Section ... . Next, we describe base expressions.

Base Expression

Base expressions are either calls to external *services* (described in Section ..), or simple (functional) expressions that can evaluated locally, (next Section ..).

Cor

Cor is a functional subset of Orc. The original Orc calculus [reference] did not include a functional subset; thus to compute 3+4, an external service for addition was called with the arguments 3 and 4. We retain the underlying philosophy of Orc, but we also allow the user to simply write ***`3+4`***, or even functional programs in a restricted language, which are then translated to calls on appropriate services.

Cor

In this section we introduce *Cor*, a subset of the Orc language. Users of functional programming languages such as Haskell and ML will already be familiar with many of the key concepts of Cor.

A Cor program is called an *expression*. Cor expressions are built up recursively from smaller expressions. Cor *evaluates* an expression to reduce it to some simple *value* which cannot be evaluated further, for example a list of numbers or a Boolean truth value. This value is called the *result* of the expression.

In the following subsections we'll introduce the concepts of Cor. First, we'll talk about simple constants, such as numbers and truth values, and the operations that we can perform on those values. Then we'll introduce conditionals (**`if/then/else`**). Then we'll introduce variables and binding, as a way to give a name to the value of an expression. After that, we'll talk about constructing data structures, and examining those structures using patterns. Lastly, we'll introduce functions.

Constants

The simplest expression one can write is a constant. It evaluates trivially to that constant value.

Cor has three types of constants, and thus for the moment three types of values:

- Integers: `... -1, 0, 1 ...`
- Booleans: `true` and `false`
- Strings: `"orc"`, `"ceci n'est pas une |"`

Operators

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

Examples

- `1+2` evaluates to `3`.
- `(98+2)*17` evaluates to `1700`.
- `4 = 20 / 5` evaluates to `true`.
- `3-5 >= 5-3` evaluates to `false`.
- `true && (false || true)` evaluates to `true`.
- `"leap" + "frog"` evaluates to `"leapfrog"`.

Here is the full set of operators that Cor supports:

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

The = operator can compare values of any type.

Examples

- `10 = true` evaluates to `false`.

Sometimes, there are situations where an expression is stuck, because it is attempting to perform some impossible operation and cannot reach a value. If this is the case, we say that the expression is *silent*. An expression is also silent if it depends on the result of a silent subexpression.

Examples

- ***10 / 0*** is silent.
- ***6* + *false*** is silent.
- ***3* + *1/0*** is silent.
- ***4* + *true* = *5*** is silent.

Note that Cor is a dynamically typed language. It does not statically check the type correctness of an expression; instead, an expression with a type error is simply silent when evaluated.

Conditionals

A conditional expression in Cor is of the form **`if`** *b* **`then`**> *f* **`else`** *g*. Its meaning is similar to that in other languages: the value of the expression is the value of *f* if *b* is true, or the value of *g* if *b* is false. Note that *f* is evaluated only if *b* is true and *g* only if *b* is false. Thus, evaluation of **`if`** ***3* = *3*** **`then`** ***5*** **`else`** ***1/0*** does not cause any error.

Unlike other languages, expressions in Cor may be silent. If *b* is silent, then the entire expression is silent. And, if *b* is true but *f* is silent then also the expression is silent (similarly, if *b* is false and *g* is silent).

The behavior of conditionals is summarized by the following table, where *v* denotes a value.

bexp	***texp***	***fexp***	***result***
true	*v*	-	*v*
true	silent	-	silent
false	-	*v*	*v*
false	-	silent	silent
silent	-	-	silent

Examples

- **if *true* then *4* else *5*** evaluates to ***5***.
- **if *2* < *3*** && ***5* < *4* then *"blue"* else *"green"*** evaluates to "green".
- **if *true* || *"fish"* then *"yes"* else *"no"*** is silent.
- **if *false* || *false* then *4*+*true* else *4*+*5*** is silent.
- **if *0* < *5* then *0/5* else *5/0*** evaluates to 0.

Variables

A *variable* is a way of naming the value of some expression so that we can use it later. Expression `x`, where x is some variable name, evaluates to the result *bound* to the variable x.

Variables are bound using a *declaration*. A declaration is a statement that has no value of its own but instead binds one or more variables. The simplest declaration is **`val`**, which evaluates an expression and binds its result to a variable. Declarations follow the rules of lexical scoping [http://en.wikipedia.org/wiki/Lexical_scope].

```
val x = 1 + 2 + 3 + 4 + 5
val y = x + x
```

These declarations bind variable `x` to 15 and variable `y` to 30.

If the expression on the right side of a **`val`** is silent, then the variable is not bound, but evaluation of other declarations and expressions can continue. If an evaluated expression depends on that variable, that expression is silent.

```
val x = 1/0
val y = 4+5
if false then x else y
```

Evaluation of the declaration and `val y = 4+5` and the expression `if false then x else y` may continue even though `x` is not bound. The expression evaluates to 9.

Note

`val` expresses a limited form of concurrency called *declarative concurrency* (add biblio reference to CTMCP). A declaration may be evaluated in parallel with the declarations and expressions that follow it; any expression which uses a variable introduced by a declaration that is still evaluating will wait for that evaluation to complete. However, since Cor is a purely declarative language, and cannot express state changes or the progress of time, it is impossible to distinguish a parallel evaluation from a sequential evaluation within Cor.

Data Structures

Cor supports two basic data structures, *tuples* and *lists*.

A *tuple expression* is a comma-separated sequence of at least two expressions, enclosed by parentheses. Each expression is evaluated; the value of the whole tuple expression is a tuple value containing each of these values in order. If any of the expressions is silent, then the whole tuple expression is silent.

Examples

- `(1+2, 7)` evaluates to `(3,7)`.
- `("true" + "false", true || false, true && false)` evaluates to `("truefalse", true, false)`.
- `(2/2, 2/1, 2/0)` is silent.

A *list expression* is a comma-separated sequence of expressions enclosed by square brackets. It may be of any length, including zero. Each expression is evaluated; the value of the whole list expression is a list value containing each of these values in order. If any of the expressions is silent, then the whole list expression is silent.

Examples

- `[1,2+3]` evaluates to `[1,5]`.
- `[true && true]` evaluates to `[true]`.
- `[]` evaluates trivially to `[]`, the empty list.
- `[5, 5 + true, 5]` is silent.

There is also a concatenation (*cons*) operation on lists, written *h*`:`*t*, where *h* and *t* are expressions. Its result is a new list whose first element is the result of *h* and whose remaining elements are the list result of *t*.

Examples

- `(1+3):[2+5,6]` evaluates to `[4,7,6]`.
- `2:2:5:[]` evaluates to `[2,2,5]`.
- Suppose `t` is bound to [3,5]. Then `1:t` evaluates to `[1,3,5]`.
- `2:3` is silent, because `3` is not a list.

Patterns

We have seen how to construct data structures. But how do we examine them, and use them? We use *patterns*.

A pattern is a powerful way to bind variables. When writing **`val`** declarations, instead of just binding one variable, we can replace the variable name with a more complex pattern that follows the structure of the value, and matches its components. A pattern's structure is called its *shape*; a pattern may take the shape of any structured value. A pattern can hierarchically match a value, going deep into its structure. It can also bind an entire structure to a variable.

Examples

- `val (x,y) = (2+3,2*3)` binds `x` to 5 and `y` to 6.
- `val [a,b,c] = ["one", "two", "three"]` binds `a` to "one", `b` to "two", and `c` to "three".
- `val ((a,b),c) = ((1, true), (2, false))` binds `a` to 1, `b` to `true`, and `c` to `(2,false)`.

A pattern must not use a variable name more than once; patterns are *linear*. For example, `(x,y,x)` is not a valid pattern.

Note that a pattern may fail to match a value, if it does not have the same shape as that value. In a **`val`** declaration, this has the same effect as evaluating a silent expression. No variable in the pattern is bound, and if any one of those variables is later evalauted, it is silent.

It is often useful to ignore parts of the value that are not relevant. We can use the wildcard pattern, written _, to do this; it matches any shape and binds no variables.

Examples

- `val (x,_,_) = (1,(2,2),[3,3,3])` binds `x` to 1.
- `val [[_,x],[_,y]] = [[1,3],[2,4]]` binds `x` to 3 and `y` to 4.

Functions

Like most other programming languages, Cor has the capability to define *functions*, which are expressions that have a defined name, and have some number of parameters. Functions are declared using the keyword **`def`**, in the following way.

```
def add(x,y) = x+y
```

The expression on the right of the `=` is called the *body* of the function.

After defining the function, we can *call* it. A call looks just like the left side of the declaration except that the variables have been replaced by expressions. To evaluate a call, first we evaluate each of its arguments. Then, we evaluate the body of the function with each of the argument names bound to the values of the arguments. The result of the call is the result of the function body.

Examples

- `add(10,10*10)` evaluates to `110`.
- `add(add(5,3),5)` evaluates to `13`.

A call may have zero arguments, in which case we write `()` for the arguments.

```
def zero() = 0
```

Non-strict evaluation

Function calls are not strict in their arguments. Even if some of the argument expressions to a call are silent, the function body will still be evaluated. Any argument variable that corresponds to a silent expression will be silent if it is used in the function body.

Recursion

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

```
def sumto(n) = if n < 1 then 0 else n + sumto(n-1)
```

Then, `sumto(5)` evaluates to 15.

Mutual Recursion

Mutual recursion is also supported:

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

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

Closure

Functions are actually values, just like any other value. Defining a function creates a special value called a *closure*; the name of the function is a variable and its bound value is the closure. Thus, a closure can be put into a data structure, or bound to some other variable, just like any other value.

```
def a(x) = x-3
def b(y) = y*4
val funs = (a,b)
```

Like any other value, a closure can be passed as an argument to another function. This means that Cor has *higher-order* functions.

```
def onetwosum(f) = f(1) + f(2)
def triple(x) = x * 3
onetwosum(triple)
```

Then, `onetwosum(triple)` is `triple(1) + triple(2)`, which is `1 * 3 + 2 * 3`, which evaluates to `9`.

Note

Since all declarations (including function declarations) in Cor are lexically scoped, these closures are *lexical closures*.

Lambda

Sometimes one would like to create a closure directly, without bothering to give it a name. There is a special keyword **`lambda`** for this purpose. By writing a function definition with the keyword **`lambda`** instead of a function name, that definition becomes an expression which evaluates to a closure.

```
def onetwosum(f) = f(1) + f(2)
onetwosum( lambda(x) = x * 3 )
{-
  identical to:

  def onetwosum(f) = f(1) + f(2)
  def triple(x) = x * 3
  onetwosum(triple)
-}
```

Then, `onetwosum( lambda(x) = x * 3 )` evaluates to 9.

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 argument, and a body expression. Naturally, 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 declaration 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.

We allow a new form of pattern which is very useful in clausal definition of functions: a constant pattern. A constant pattern is a match only for the same constant value. We can use this to define the "base case" of a recursive function in a straightforward way.

```
{- Fibonacci numbers -}
def fib(1) = 1
def fib(n) = if (n < 1) then 0 else fib(n-1) + fib(n-2)
```

```
{- Take up to the first n elements from a list -}
def take(0,_) = []
def take(_,[]) = []
def take(n,h:t) = h:(take(n-1,t))
```

Mutual recursion and clausal definitions are allowed to occur together. For example, this function takes a list and evaluates to 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)
```

`stutter([1,2,3])` evaluates to `[1,1,2,3,3]`.

Clauses of mutually recursive functions may also be interleaved, to make them easier to read.

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

Comments

Cor has 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 -} -}

{- They may appear anywhere, -}
1 + {- even in the middle of an expression. -} 2 + 3
```

Communicating with external services

Cor is a pure declarative language. It has no state, since variables are bound at most once and cannot be reassigned. It cannot communicate with the outside world except by producing a value. Clearly, the full Orc language must transcend this limitation, because the orchestration of external services is critical to Orc's purpose.

We now introduce *sites*: Orc's interface to external services. Sites are called using the same syntax as a function call, but with a slightly different meaning. Sites are values, introduced and bound to variables by a special declaration.

Calling a site

Suppose that the variable `Google` is bound to a site which sends a string to the Google search engine and returns the URL of the top result. A call to `Google` looks just like a function call.

```
{- Get the top search result for "computation orchestration" -}
Google("computation orchestration")
```

When the Google service determines the top result for this search, it responds with some URL. The site call then evaluates to that URL value. Note that the service might never respond: Google's servers might be down, the network might be down, or the search might yield no result URL. If the service fails to respond, the site call remains silent.

A site call involves only one roundtrip communication with an external service. All of the information needed for the call must be present before contacting the service. Thus, site calls are strict; all arguments must be bound before the call can proceed. If any argument is silent, the call never occurs.

A site is sometimes called purely for its effect on the external world; its return value is irrelevant. Many sites which do not need to return a meaningful value will instead return a *signal*: a special value which carries no information (analogous to the unit value `()` in ML). The signal value can be written as **`signal`** within Orc programs.

```
-- Print a string to some output console
-- The return value of this site call is a signal
println("Hello, World!")
```

Declaring sites

Sites are bound to variables by a **`site`** declaration. This declaration makes some external service available as a site and binds that site to the given variable. This example instantiates a Java object to be used as a site in an Orc program, specifically a Java object which provides an asynchronous buffer service.

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

The concurrency combinators of Orc

The Cor language has no concurrency. We extend Cor's simple model with powerful concurrent capabilities by adding three *combinators*: special operators that connect expressions together so that they can evaluate concurrently in useful ways. We also generalize the simple notion of evaluation to a more powerful notion that makes more sense in a concurrent context.

Bar: Parallelism and publications

The first combinator we introduce is the bar combinator, written `|`. Two expressions combined by `|` evaluate in parallel with each other. But what is the value of such an expression? For example, what is the value of `1+2 | 3+4` ?

At this point, we can no longer talk about evaluation of an expression, because expressions may not have a single unique value. Instead, we *execute* an expression, which *publishes* some number of values. Whenever a Cor expression is executed, it is evaluated, and its value is published. If the expression is silent, then no value is published. An expression whose execution publishes no values is also called silent.

So, two expressions combined by `|` execute in parallel, and whenever one of those execution publishes a value, the combined execution publishes that value.

Examples

- `3+4` publishes 7.
- `3/0 | 3/1` publishes `3`.
- `1 | 2 | 1+2` publishes 1, 2, and 3, in unspecified order.

Push: Capturing publications

Now that we have expressions which publish multiple values, what can we do with those publications? The push combinator, written `>x>`, combines an expression which publishes some values with another expression which will use those values.

f `>x>` *g* executes only the expression *f*. Each time that *f* publishes a value *v*, a new parallel copy of *g* executes, in which the variable `x` is bound to that value *v*. The combined execution does not publish the value *v*; it is *hidden*. However, whenever some copy of *g* publishes a value, the combined execution also publishes that value.

The variable is optional; a push without a variable is written `>>`.

`stop` is a special expression which is always silent. It is typically used together with a push to silence the publications of another expression.

Pull: Making `val` concurrent

The pull combinator, written *g* `<x<` *f*, allows us to block a computation waiting for a result, or terminate a computation. In fact, we have already seen this combinator: it is often useful to write it as a declaration of the form **`val`** `x =` *f* preceding an expression *g*. In a functional setting, its behavior is exactly as previously described. However, in a concurrent setting, **`val`** takes on two new properties.

First, we cannot simply bind the variable to the result of the expression, because an expression may now publish many values as it executes. Instead, the variable is bound to the first value published by the execution of the expression. Additionally, as soon as that expression publishes its first value, it *terminates*; its execution immediately stops.

Second, execution of subsequent statements and expressions continues immediately, without waiting for the expression to publish a value and bind the variable. If the variable is used before it is bound, the execution using that variable *blocks* until the variable is bound. If the variable is never bound, then that execution blocks forever and becomes silent.

Additional Features

Orc has some advanced features that are useful in writing certain kinds of programs.

Special call forms

The `.` notation

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*.

Currying

It is sometimes useful to *stage* the arguments to a function; that is, rather than writing a function on two arguments, one instead writes a function on one argument which returns a function taking the second argument and performing the remainder of the evaluation. (form of curried applications)

This technique is known as *currying* and it is common in functional programming languages. It is obviously supported in Orc since it is possible to define closures. However, there is additional support in defining functions directly in a curried way.

For example, instead of writing

```
def f(x) = lambda(y) = x+y
```

it is also permissible to simply write

```
def f(x)(y) = x+y
```

Separate stages of application are separated by parentheses around their argument lists.

Special patterns

Bang pattern

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 pattern

In taking apart a value with a pattern, it is often useful to capture intermediate results.

```
val (a,b) = ((1,2),(3,4))
val (ax,ay) = a
val (bx,by) = b
```

We can use the **`as`** keyword to simplify this process, giving a name to an entire subpattern. Here is an equivalent version of the above code.

```
val ((ax,ay) as a, (bx,by) as b) = ((1,2),(3,4))
```

Site patterns

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.

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.

Note

The concept of views and view patterns is not new in programming languages; it has been demonstrated in other languages, most notably Scala [http://www.scala-lang.org/]. In fact, the underlying implementation strategy for using sites as view patterns is very similar to the use of extractor objects and the `unapply` method in Scala.

The before combinator

Orc has a fourth concurrent combinator: the *before* combinator, written `f ; g`. 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.

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. Sequential programs conflate the concept of producing a value with the concept of termination. Orc separates these two concepts; variable binding combinators like `>x>` and `<x<` handle values, whereas `;` detects the completion of an execution.