Orc Reference Manual v2.0.1

Orc Reference Manual v2.0.1

Publication date 2011-04-27

License and Grant Information

This material is based upon work supported by the National Science Foundation under Grant No. CCF-0811536. Any opinions, findings, and conclusions or recommendations expressed in this document are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Table of Contents

List of Tables

List of Examples

Chapter 1. Data Values

The primitive data types in Orc are Booleans, numbers, strings, and a unit value `signal`. Orc also has structured values: lists, tuples, records, and algebraic datatypes. Functions are values, called closures. Additionally, sites are themselves values, and may be passed to sites or returned by sites.

All of the preceding values are immutable. Orc also has access to mutable state through sites. The Standard Library includes many such sites, e.g. `Ref` and `Channel`.

Orc sites may create and manipulate many more kinds of values, such as Java objects, XML documents, video files, relational databases, etc. In fact, all of the built-in datatypes could be implemented by external sites.

1.1. Booleans

Orc supports *Boolean* values, `true` and `false`.

1.1.1. Syntax

[53] BooleanLiteral `::=` `true` | `false`

1.1.2. Operations

Notable Boolean operations include:

- Logical negation (not): `~`
- Logical and: `&&`
- Logical or: `||`

as well as the comparison operators, which yield Boolean results:

- Less than: `<:`
- Greater than: `:>`
- Less than or equal to: `<=`
- Greater than or equal to: `>=`
- Equal to: `=`
- Not equal to: `/=`

Note: Unlike the typical symbols `<` and `>` for arithmetic relations, Orc uses `<:` and `:>` respectively. This usage avoids ambiguity with the sequential combinator and the pruning combinator.

1.1.3. Type

A Boolean value has type `Boolean`.

1.1.4. Java calls

Orc Boolean values are passed in calls to and returns from Java code as `java.lang.Boolean`, which is boxed and unboxed per *The Java Language Specification [http://java.sun.com/docs/books/jls/third_edition/html/conversions.html#5.1.7]* as `boolean`.

1.1.5. Examples

Example 1.1. Boolean XOR

```
{- Define exclusive or -}

def xor(a,b) = (a || b) && ~(a && b)

xor(true, true) | xor(false, false)

{-
OUTPUT:
false
false
-}
```

1.1.6. Related Links

Related Reference Topics

- `if then else`
- `Ift` [147]
- `Iff` [147]

Related Tutorial Sections

- Literal Values
- Conditionals

1.2. Numerics

Orc includes two numeric types, *integers* and *numbers*. Orc's integers are arbitrary-precision two's complement integers. Orc's numbers are floating point numbers, with an arbitrary-precision two's complement significand and a 32-bit two's complement exponent (base 10). These numbers behaves in accordance with the ANSI INCITS 274-1996 subset of ANSI/IEEE Std 754-2008. Namely, "infinite, NaN, or subnormal results are always treated as errors, and –0 results are hidden".

Note that the divide operation [149] on Orc numbers can encounter a non-terminating decimal expansion, where there is no exact representable decimal result. In this case, the divide operation falls back to division using IEEE 754 binary64 (formerly called double precision) binary floating-point operands. This fall back may result in a loss of precision for this operation.

Similarly, the exponent operation [149] may fall back to IEEE 754 binary64 binary floating-point operands in the case of a fractional exponent. This fall back may result in a loss of precision for this operation.

1.2.1. Literals

1.2.1.1. Syntax

[54] IntegerLiteral ::= DecimalDigit+

[55] NumberLiteral ::= IntegerLiteral DecimalPart**?** ExponentPart**?**

[56] DecimalPart ::= `.` IntegerLiteral

[57] ExponentPart ::= `E` IntegerLiteral
| `E+` IntegerLiteral
| `E-` IntegerLiteral
| `e` IntegerLiteral
| `e+` IntegerLiteral
| `e-` IntegerLiteral

Numeric literals in Orc are specified in decimal. Leading zeros are allowed, but have no significance. Trailing zeros after a decimal point also have no significance. If a numeric literal contains a decimal point or an "`E`", it is a number (floating point) literal, otherwise it is an integer literal.

1.2.2. Operations

Notable operations on integers and numbers include:

- Add: `+`
- Subtract: `-`
- Negate (unary minus): `-`
- Multiply: `*`
- Divide: `/`
- Exponent: `**`
- Remainder: `%`
- Absolute value: `abs`

- Signum: `signum`
- Floor: `Floor`
- Ceiling: `Ceil`

Arithmetic operators with two integer arguments will perform an integer operation and return an integer result; for example, `5 / 2` performs integer division and returns 2. However, if either argument to an operator has a decimal part (even if it is trivial, as in `3.0`), the other argument will be promoted, and a decimal operation will be performed. For example, `5 / 2.0` and `5.0 / 2` both perform decimal division and return 2.5.

1.2.3. Java calls

Orc integer values are passed in calls to and returns from Java code as `java.math.BigInteger`, or if the callee method expects a specific numeric type, an Orc integer will be converted to a `java.lang.Byte`, `java.lang.Short`, `java.lang.Integer`, `java.lang.Long`, `java.lang.Float`, or `java.lang.Double`, as appropriate. These values are boxed and unboxed per *The Java Language Specification [http://java.sun.com/docs/books/jls/third_edition/html/conversions.html#5.1.7]*.

Orc number values are passed in calls to and returns from Java code as `java.math.BigDecimal`, or if the callee method expects a specific numeric type, an Orc number will be converted to a `java.lang.Float` or `java.lang.Double`, as appropriate. These values are boxed and unboxed per *The Java Language Specification [http://java.sun.com/docs/books/jls/third_edition/html/conversions.html#5.1.7]*.

1.2.4. Type

All numeric values (integers and numbers) have type `Number`. Integer values also have the more specific type `Integer`, which is a subtype of `Number`.

1.2.5. Examples

Example 1.2. Operator Precedence

```
{- Operators follow common precedence practice.
   Combinators have lower precedence than operators.
-}

4 + 15 / 3 * 2 >result> result

{-
OUTPUT:
14
-}
```

See the Orc precedence table.

Example 1.3. Integer to Number Promotion

```
{- Arithmetic operations, when all arguments are integers,
   operate as integer operations.  If any argument is a floating-
   point number, then the operation treats all arguments as
   floating-point numbers and operates in that domain.
-}

16 - 8 + 50.0 / 3 * 1 >a>
16 - Floor(8.5) + Ceil(4e1 + 9.99) / 3 * 1 >b>
(a, b)

{-
OUTPUT:
(24.666666666666668, 24)
-}
```

Example 1.4. Square Roots and Squares

```
{- Calculate the square roots and squares of
   a list of numbers
-}

val nums = [1, 2, 3, 4.0, 5.00]

each(nums) >x> (
  x**(1.0/2) >sqrt>
  x**2 >sq>
  (x,sqrt,sq)
)

{-
OUTPUT:PERMUTABLE
(1, 1.0, 1)
(2, 1.4142135623730951, 4)
(3, 1.7320508075688772, 9)
(4.0, 2.0, 16.00)
(5.00, 2.23606797749979, 25.0000)
-}
```

1.2.6. Related Links

Related Reference Topics

- + operator [148]
- - operator [148]
- unary - operator [148]
- * operator [148]

- `**` (exponentiation) operator [149]
- `/` operator [149]
- `%` (modulus) operator [149]
- `abs` function [151]
- `signum` function [151]
- `Floor` site [152]
- `Ceil` site [152]

Related Tutorial Sections

- Literal Values
- Operators

1.3. Character Strings

Orc's character sequence representation is a *string*. Characters in Orc strings are drawn from the Unicode character set, and are encoded in the UTF-16 encoding form. (Note that Orc source code files are encoded in the UTF-8 encoding form.) Orc strings have an explicit length and are not zero-terminated. They are limited to a maximum length of 2^31-1 (2 147 483 647) characters.

1.3.1. Literals

1.3.1.1. Syntax

[58] CharacterStringLiteral `::= "Character+" | ""`

String literals are enclosed in quotation marks (U+0022). Any Unicode character can be placed between the quotation marks, except: line terminators (CR, LF, NEL [U+0085], LS [U+2028], FF, or PS [U+2029]) or a quotation mark. A reverse solidus (backslash) and its subsequent character are treated as follows:

- `\f`: represents form feed (FF) (U+000C)
- `\n`: represents line feed (LF) (U+000A)
- `\r`: represents carriage return (CR) (U+000D)
- `\t`: represents character tabulation (HT) (U+0009)
- `\\`: represents reverse solidus (backslash) (U+005C)
- `\"`: represents quotation mark (U+0022)
- `\` followed by any other character: represents that character

1.3.2. Operations

Notable string operations include:

- All `java.lang.String` operations
- Convert an Orc value to a string: `write` [195]
- Split a string into a list of strings corresponding to its lines: `lines` [195]
- Combine a list of strings into one multi-line string: `unlines` [196]
- Split a string into a list of strings corresponding to its words: `words` [196]
- Combine a list of strings into one multi-word strings: `unwords` [196]

1.3.3. Type

A string value has type `String`.

1.3.4. Java calls

Orc strings are passed in calls to and returns from Java code as `java.lang.String`.

1.3.5. Examples

Example 1.5. String Concatenation

```
{- Concatenate two strings -}

"hello" + " world"

{-
OUTPUT:
"hello world"
-}
```

Example 1.6. String Functions

```
{- Showcase some Orc library string functions -}

Println("I go to the console") >>
unwords(["I", "like", "Orc"])

{-
OUTPUT:
I go to the console
"I like Orc"
-}
```

1.3.6. Related Links

Related Reference Topics

- `Write` site [195]
- `lines` function [195]
- `unlines` function [196]
- `words` function [196]
- `unwords` function [196]

Related Tutorial Sections

- Literal Values

1.4. `signal`

A *signal* is a value that carries no information. It typically indicates completion of a call, when there is no other meaningful value to return. It is analogous to `void` in Java or `()` in ML.

1.4.1. Syntax

[52] SignalLiteral `::= signal`

1.4.2. Operations

Notable signal operations include:

- Publish a signal if an expression is true: `Ift` [147]
- Publish a signal if an expression is false: `Iff` [147]
- Publish some number of signals simultaneously: `signals` [200]

1.4.3. Type

The `signal` value has type `Signal`.

1.4.4. Java calls

Orc signals don't correspond to any Java value, so if a signal is passed to Java code, it will be as an `java.lang.Object` of a type not specified here. A return of type `void` from Java code is converted into a signal return value.

1.4.5. Related Links

Related Reference Topics

- `Ift` site [147]
- `Iff` site [147]
- `metronome` function [198]
- `signals` function [200]

Related Tutorial Sections

- Sites

1.5. Lists

A *list* consists of a finite sequence of values. The empty list is written as `[]`.

1.5.1. Syntax

[4] List ::= [Expression , ... , Expression]

1.5.2. Constructors

The list expression [E_0 , … , E_n] publishes the list [v_0 , … , v_n] only if each expression E_i deflates to value v_i. Otherwise, it halts silently.

1.5.3. Operations

Notable list operations include:

- *Cons* (construct) a list with first element h and remaining elements t: `h:t`
- Publish `true` iff the list l has no elements: `empty(l)`
- Publish the length of list l: `length(l)`
- Publish the first element of list l: `head(l)`
- Publish a list with every element in list l except the first: `tail(l)`
- Publish all but the last element of list l: `init(l)`
- Publish the last element of list l: `last(l)`
- Publish the nth element of a list, counting from 0: `index(l, n)`
- Publish a list with the first n elements of the list l: `take(n, l)`
- Publish a list with all but the first n elements of the list l: `drop(n, l)`
- Publish every value in list l, simultaneously: `each(l)`
- Concatenate list a and list b: `append(a, b)`
- Publish a list with the elements of list l in reverse order: `reverse(l)`
- Publish a list containing only those elements of l which satisfy the function f: `filter(f, l)`
- Apply unary function f to every element of list l (in parallel), and return a list of the results: `map(f, l)`
- Apply a binary function to every element of a list: `foldl`, `foldr`, and many variations thereof.
- Combine two lists into a list of pairs, and its reverse: `zip` and `unzip`
- Concatenate a list of lists l into a single list: `concat(l)`
- Publish `true` if item x is a member of list l: `member(l)`

1.5.4. Type

The type of a list is `List[`*U*`]`, where *U* is the join of the types of each of its elements. In particular, if all of the elements have the same type *T*, then the list will have type `List[`*T*`]`.

1.5.5. Java calls

Orc lists do not correspond to any Java value, so if a list is passed to Java code, it will be as a `java.lang.Object` of a type not specified here.

1.5.6. Examples

Example 1.7. Sample Lists

```
  []                        -- empty list
| [1, 2, 3]                 -- a list of integers
| [(1, 2), (2, 3), (3, 4)] -- a list of tuples of integers
| [1, 1 * 2, 1 * 2 * 3]    -- a list of the first 3 factorials
| [[1], [2, 2], [3, 3, 3]] -- a list of lists of integers

{-
OUTPUT:PERMUTABLE:
[]
[1, 2, 3]
[(1, 2), (2, 3), (3, 4)]
[1, 2, 6]
[[1], [2, 2], [3, 3, 3]]
-}
```

Example 1.8. Lists Are Not Sets

```
{-
  Lists do not behave like sets.
  The order and number of elements in a list do matter.
-}

  [2,3] /= [3,2]
| [2] /= [2,2]

{-
OUTPUT:
true
true
-}
```

Example 1.9. Building Lists with Cons

```
  3:[]
| 4:3:[2,1]

{-
OUTPUT:PERMUTABLE:
[3]
[4, 3, 2, 1]
-}
```

1.5.7. Related Links

Related Reference Topics

- List pattern
- Cons pattern
- Standard library list functions
- `each` function [163]
- `map` function [163]
- `reverse` function [163]
- `filter` function [164]
- `head` function [164]
- `tail` function [164]
- `init` function [165]
- `last` function [165]
- `empty` function [165]
- `index` function [166]
- `append` function [166]
- `foldl` function [166]
- `foldl1` function [167]
- `foldr` function [167]
- `foldr1` function [168]
- `afold` function [168]
- `cfold` function [168]
- `zip` function [169]

- `unzip` function [169]
- `concat` function [170]
- `length` function [170]
- `take` function [170]
- `drop` function [171]
- `member` function [171]
- `merge` function [172]
- `mergeBy` function [172]
- `mergeUnique` function [173]
- `mergeUniqueBy` function [174]
- `sort` function [172]
- `sortBy` function [173]
- `sortUnique` function [174]
- `sortUniqueBy` function [175]
- `group` function [175]
- `groupBy` function [176]
- `range` function [176]
- `rangeBy` function [176]
- `any` function [177]
- `all` function [177]
- `sum` function [178]
- `product` function [178]
- `and` function [178]
- `or` function [179]
- `minimum` function [179]
- `maximum` function [179]

Related Tutorial Sections

- Lists
- Idioms: Lists

1.6. Tuples

A *tuple* is a sequence of at least two values. Orc does not have 0-tuples or 1-tuples.

Tuples are intended to be used for sequences with a fixed length and varying element types, whereas lists are intended to be used for sequences with varying length and a fixed element type.

1.6.1. Syntax

[3] Tuple `::=` `(` Expression `,` `...` `,` Expression `)`

1.6.2. Constructors

The tuple expression (E_0 , … , E_n) publishes the tuple value (v_0 , … , v_n) only if each expression E_i deflates to value v_i. Otherwise, it halts silently.

1.6.3. Operations

Notable tuple operations include:

- Return the tuple element at position *`index`*, starting from 0: `tuple(index)`
- Return the first element of a pair: `fst(tuple)`
- Return the second element of a pair: `snd(tuple)`

1.6.4. Type

The type of a tuple value (v_0 , … , v_n) where v_i has type T_i, is a tuple type, written (T_0 , … , T_n).

1.6.5. Java calls

Orc tuples don't correspond to any Java value, so if a tuple is passed to Java code, it will be as a `java.lang.Object` of a type not specified here.

1.6.6. Examples

Example 1.10. Tuple Selection

```
{- Unzip a list of tuples into a tuple of lists -}

val squares = [(1,1), (2,4), (3,9), (4,16)]

signal >> ( map(fst,squares) , map(snd,squares) )

{-
OUTPUT:
([1, 2, 3, 4], [1, 4, 9, 16])
-}
```

Example 1.11. Fork-Join

```
{- Print "fork", but wait at least 500ms before printing "join" -}

( Println("fork"), Rwait(500) ) >> Println("join") >> stop

{-
OUTPUT:
fork
join
-}
```

1.6.7. Related Links

Related Reference Topics

- Tuple pattern
- `fst` function [192]
- `snd` function [192]

Related Tutorial Sections

- Tuples
- Fork-Join

1.7. Records

A *record* is an unordered finite map from keys to values. The empty record is written as `{. .}`.

1.7.1. Syntax

[5] Record `::=` `{.` Key = Expression `,` `...` `,` Key = Expression `.}`

1.7.2. Constructors

A record is constructed from a comma-separated sequence of bindings enclosed by dotted braces. Each binding associates a key with an expression.

The record expression `{.` K_0 = E_0 `,` … `,` K_n = E_n `.}` publishes the record `{.` K_0 = v_0 `,` … `,` K_n = v_n `.}` only if each expression E_i deflates to value v_i. Otherwise, it halts silently.

Duplicate bindings for the same key are allowed, but only the rightmost binding will be used in the resulting record value.

1.7.3. Operations

Notable record operations include:

- Publish the value bound to key k in record r (a dot access): r`.`k
- Extend a record r with new entries : r `+` s

A record extension r `+` s publishes a new record with all of the bindings of s, plus all of the bindings in r which do not bind keys mentioned in s. In other words, s overrides r. Record extension is associative, but not commutative. The expression `{.` K_0 = E_0 `,` … `,` K_n = E_n `.}` is equivalent to the expression `{.` K_0 = E_0 `.}` + … + `{.` K_n = E_n `.}`.

1.7.4. Special Keys

There are two record keys that have special meanings:

- If a record has a binding for the `apply` key, then the record may be called like a site or function.
- If a record has a binding for the `unapply` key, then the record may be used in a call pattern.

1.7.5. Type

The type of a record value `{.` K_0 = v_0 `,` … `,` K_n = v_n `.}` where v_i has type T_i, is a record type, written `{.` K_0 `::` T_0 `,` … `,` K_n `::` T_n `.}`.

1.7.6. Java calls

Orc records do not correspond to any Java value, so if a record is passed to Java code, it will be as a `java.lang.Object` of a type not specified here.

1.7.7. Examples

Example 1.12. Normalizing Vectors

```
{- Normalize a given integer vector using records -}

def magnitude(v) =
  (v.x * v.x +
   v.y * v.y +
   v.z * v.z) ** 0.5

def norm(v) =
  val m = magnitude(v)
  {. x = v.x / m,
     y = v.y / m,
     z = v.z / m .}

val velocity = {. x = 3.0, y = 0.0, z = 4.0 .}
norm(velocity)

{-
OUTPUT:
{. x = 0.6, y = 0, z = 0.8 .}
-}
```

Example 1.13. Record Extension

```
{- Add an alpha field to an rgb color record -}

val rgb = {. red = 60, green = 230, blue = 5 .}
val rgba = rgb + {. alpha = 128 .}
rgba.alpha

{-
OUTPUT:
128
-}
```

1.7.8. Related Links

Related Reference Topics

- Record pattern
- Dot access

Related Tutorial Sections

- Records

1.8. Algebraic Data Types

An Orc *datatype* is an algebraic data type, or "tagged union". A datatype value contains a sequence of values enclosed by a *tag*. This value can be matched by a pattern.

Datatypes are defined by a `type` declaration. Each *constructor* in the declaration introduces a new tag, followed by a sequence of *slots*. The `|` separator allows multiple constructors to be defined at the same time.

In an untyped program, slots are written as _. In a typed program, slots contain types.

1.8.1. Syntax

[32] DeclareDatatype ::= `type` TypeVariable TypeParameters**?** = Constructor `|` **...** `|` Constructor

[33] Constructor ::= Variable `(` Slot `,` **...** `,` Slot `)`

[34] Slot ::= Type ▮ _

1.8.2. Constructors

Each constructor is a site, which takes one argument for each slot of the constructor, and publishes a datatype value consisting of the sequence of argument values, tagged with that constructor's unique tag.

Each constructor also has a corresponding `unapply` member, which takes a value with that constructor's tag, removes the tag, and publishes the sequence of values as a tuple. If its argument does not have the tag, it halts. Thus, constructors can be used in pattern matching.

1.8.3. Type

A datatype declaration defines a new *sum type*.

Each constructor defined by the datatype has the function type `lambda[`X_0`, …, `X_n`](`T_0`, …, `T_n`) :: `S, where X_i are the type parameters of the datatype, T_i are the types in the slots of the constructor, and S is the sum type.

Each constructor also has a corresponding `unapply` member, with type `lambda[`X_0`, …, `X_n`](`S`) :: (`T_0`, …, `T_n`)`.

A datatype declaration may define a *recursive type*: the name of the type may be used in the definition of the type itself. In fact, this is the only way to declare a recursive type in Orc.

1.8.4. Examples

Example 1.14. Enumeration

```
{- An enumeration, such as Java's enum, can be represented by a datatype
   whose constructors have no arguments. Note that an empty argument list,
   (), is still needed.
-}
type objective = Primary() | Secondary() | Tertiary()

[Secondary(), Tertiary()]

{-
OUTPUT:
[Secondary(), Tertiary()]
-}
```

Example 1.15. Geometric shape datatype

```
{- A Shape type with three data constructors -}
type Shape = Rectangle(_, _) | Circle (_)

def area(Rectangle(width,height)) = width * height
def area(Circle(radius)) = 3.1415926535897 * radius ** 2

area(Rectangle(2, 3)) | area(Circle(1))

{-
OUTPUT:PERMUTABLE:
6
3.14159265358970
-}
```

Example 1.16. Binary tree node

```
{- This is a binary tree datatype
   Leaf nodes carry integer values
-}

type Node = LeafNode(Integer) | InnerNode(Node, Node)

{- Constructing a simple tree
    /\
   1 /\
    2  3
-}
InnerNode(LeafNode(1),InnerNode(LeafNode(2),LeafNode(3)))

{-
OUTPUT:
InnerNode(LeafNode(1), InnerNode(LeafNode(2), LeafNode(3)))
-}
```

Example 1.17. Polymorphic binary tree node

```
{- This is a binary tree datatype
   Leaf nodes carry values of type T
-}

type Node[T] = LeafNode(T) | InnerNode(Node, Node)

{- Constructing a simple tree
      /\
   "A" /\
    "B"  "C"
-}
InnerNode[String](LeafNode("A"),InnerNode(LeafNode("B"),LeafNode("C")))

{-
OUTPUT:
InnerNode(LeafNode("A"), InnerNode(LeafNode("B"), LeafNode("C")))
-}
```

Example 1.18. Orc built-in Option type

```
{- A datatype for optional values of type T -}

type Option[T] = Some(T) | None

{-
NONRUNNABLE
-}
```

1.8.5. Related Links

Related Reference Topics

- Call pattern
- `type`: Declare Type

Related Tutorial Sections

- Datatypes
- Patterns

1.9. Closures

Functions are first-class values in Orc. Defining a function creates a special value called a *closure*; the defined name of the function is a variable and the value bound to it is the closure. A closure can be published, passed as an argument to a call, or put into a data structure, just like any other value.

Since all declarations — including function declarations — are lexically scoped, these closures are lexical closures [http://en.wikipedia.org/wiki/Lexical_closure]. When a closure is created, if the body of the function contains any variables other than the formal parameters, closure creation blocks until those variables are bound, and then the values bound to those variables are stored as part of the closure. Then, when the closure is called, the evaluation of the function body uses those stored bindings.

1.9.1. Type

The type of a closure is a *function type* `lambda [`X_0 `, … ,` X_m`](`T_0 `, … ,` T_n`) :: `R, where T_i are the argument types of the closure, R is its return type, and X_j are the type parameters if the function is polymorphic. This type is derived from the original definition of the function.

1.9.2. Examples

Example 1.19. Staged Addition

```
{- Create a closure using inc, and then apply it -}

def inc(n) =
  def addnto(x) = x + n
  addnto

val f = inc(3)
f(4)

{-
OUTPUT:
7
-}
```

Example 1.20. One Two Sum

```
{- The function triple() is used as a closure -}

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

{-
OUTPUT:
9
-}
```

1.9.3. Related Links

Related Reference Topics

- `def`: Define Function
- `lambda` Expression
- Function Calls
- Adding Type Information to Functons

Related Tutorial Sections

- First-Class Functions

1.10. Mutable State

A mutable object is a value on which operations may yield different results when called at different times. Mutable objects are common in imperative programming languages. They reflect the concept of a storage location in a von Neumann style architecture.

Most of Orc's value types are immutable. The standard library provides the following kinds of mutable objects:

- `Ref`: Reference to a value that can be changed
- `Cell`: A write-once `Ref`
- `Channel`: A queue of values
- `BoundedChannel`: A queue of values with a maximum size
- `Array`: A mutable integer-indexed sequence of values
- `Counter`: A value that can be incremented and decremented atomically
- `Dictionary`: A mutable dynamic record

1.10.1. Related Links

Related Reference Topics

- `Ref` site [183]
- `Cell` site [182]
- `Channel` site [185]
- `BoundedChannel` site [186]
- `Array` site [188]
- `Counter` site [190]
- `Dictionary` site [191]

Related Tutorial Sections

- Mutable References
- Channels

1.11. External Values

Orc's value set is "open" — any value returned from a site call can be used in an Orc execution. For example, Orc programs running on a Java virtual machine (JVM) can interact with any Java object.

1.11.1. Type

External values usually have types that cannot be represented using Orc's primitive types or structured types. An `import type` declaration may be used to import such a type into the Orc type system.

1.11.2. Examples

Example 1.21. Java BitSet

```
{- Using the Java class java.util.BitSet,
   construct a new instance, and then
   set bits 2 and 4 in the BitSet instance
-}

import class BitSet = "java.util.BitSet"

BitSet() >b> b.set(2) >> b.set(4) >> b.toString()

{-
OUTPUT:
"{2, 4}"
-}
```

Example 1.22. Java HashMap

```
{- Using the Java class java.util.HashMap, construct a new instance,
   and then put and get some entries
-}

import class HashMap = "java.util.HashMap"

HashMap[String, String]() >m> m.put("Mickey", "Minnie") >>
m.put("Donald", "Daisy") >> m.get("Mickey")

{-
OUTPUT:
"Minnie"
-}
```

1.11.3. Related Links

Related Reference Topics

- Java Sites

- Web Services
- Custom Sites
- Typing of Sites

Related Tutorial Sections

- Importing Resources

Chapter 2. Expressions

Orc expressions are executed, rather than evaluated. An execution may call external services and publish some number of values (possibly zero). Different executions of the same expression may have completely different behaviors; they may call different services, may receive different responses from the same site, and may publish different values. This chapter shows how various expressions are executed in Orc.

2.1. Literal Value

A literal value may occur on its own as an expression. Execution of a literal value immediately publishes that value and then halts.

2.1.1. Syntax

[51] Literal ::= SignalLiteral
| BooleanLiteral
| NumberLiteral
| CharacterStringLiteral
| `null`

2.1.2. Type

The type for each literal value is given in the Data Values chapter.

2.1.3. Examples

Example 2.1. String Literal as Expression

```
{-
  http://en.wikipedia.org/wiki/The_Treachery_of_Images
-}

"ceci n'est pas une |"

{-
OUTPUT:
"ceci n'est pas une |"
-}
```

2.1.4. Related Links

Related Reference Topics

- Data Values
- Publication

Related Tutorial Sections

- Literal Values

2.2. Variable

A variable may occur on its own as an expression. Execution of a variable publishes the value bound to that variable, and then halts.

The variable might be executed before it is bound to a value. This could occur if the variable was introduced by a pruning combinator, or if it is the name of a defined function whose body contains unbound variables. In this case, execution of that variable blocks until the variable is bound.

If the variable was introduced by a pruning combinator, and the right side of that combinator halts before the variable becomes bound, execution of the variable also halts.

2.2.1. Syntax

[59] Variable `::=` Identifier

2.2.2. Type

The type of a variable expression is the type given to it by the current typing context.

2.2.3. Examples

Example 2.2. Blocking on a Variable

```
{-
  Publish the values bound to two variables.
  One of the bindings occurs only after some time has passed,
  so execution of that variable blocks.
-}

val x = 0
val y = Rwait(1000) >> 1

x | y

{-
OUTPUT:
0
1
-}
```

2.2.4. Related Links

Related Reference Topics

- Publication
- Halting
- Blocking

- Combinators

Related Tutorial Sections

- Sequential Combinator

2.3. `stop`

The `stop` expression does nothing. Execution of `stop` immediately halts silently.

2.3.1. Syntax

[2] Stop `::= stop`

2.3.2. Type

`stop` has type `Bot`.

2.3.3. Examples

Example 2.3. Print Silently

```
{-
  Print three numbers in sequence, and then halt without publishing
  a signal from Println.
-}

Println("1") >>
Println("2") >>
Println("3") >>
stop

{-
OUTPUT:
1
2
3
-}
```

Example 2.4. Square Root

```
{-  Define a square root function which halts silently if
    its argument is negative.
-}

def squareroot(n) = if (n <: 0) then stop else (n ** 0.5)

squareroot(-1)

{-
OUTPUT:
-}
```

2.3.4. Related Links

Related Reference Topics

- Halting
- Silence
- `Bot`
- Otherwise combinator

Related Tutorial Sections

- Sequential Combinator

2.4. Site and Function Calls

A *call* is an invocation of a *target* expression on a sequence of *argument* expressions. First, the target expression is deflated to a value g. If g is a site, then the call is a *site call*. If g is a closure, then the call is a *function call*. If g is a record, then the call is a *record application*. Each of these cases is described below.

2.4.1. Syntax

[6] Call ::= Expression TypeArguments**?** Arguments
[68] Arguments ::= (Expression , ... , Expression)
[69] TypeArguments ::= [Type , ... , Type]

2.4.2. Site Calls

A site call is *strict*. Execution of a site call $g($ `(` E_0 `,` $\ldots$ `,` E_n `)` deflates each expression E_i to a value v_i. The site g is then invoked with arguments v_i.

A site call publishes at most one value. Once it publishes a value, it immediately halts. If the site is helpful, the call may halt without publishing a value.

2.4.3. Function Calls

Unlike a site call, a function call is *lenient*. Execution of a function call $g($ `(` E_0 `,` $\ldots$ `,` E_n `)` deflates each expression E_i, but does not wait for values to be published; instead, the body of the function executes immediately.

Since a function call executes an arbitrary expression, it may publish any number of values.

2.4.4. Record Application

A record may be called if it has a member named `apply`. In this case, the value v bound to `apply` in the record is retrieved, and then v is called on the arguments. v may be a site, a closure, or even another record. The call behaves exactly as if v had been the original call target.

If a record without an `apply` member is called, the call halts silently.

2.4.5. Type

In addition to argument expressions, a call may also have type arguments A_j, since the target may have a polymorphic type.

Suppose the target expression E has type S, and each argument expression E_i has type S_i.

If S is a function type `lambda [`X_0 `,` $\ldots$ `,` X_m`](`T_0 `,` $\ldots$ `,` T_n`) ::` R, then there must be exactly as many arguments as argument types, and each argument type S_i must be a subtype of the corresponding type T_i (with all X_j replaced by A_j). Then the type of the whole call is R, the return type.

If S is the type of a site, S itself behaves like a site during typehcecking. The typechecker passes the argument types S_i and type arguments A_j to S, which then responds with the return type R, or indicates that the argument types are not appropriate. If S does respond, then the type of the call expression is the return type R.

2.4.6. Examples

Example 2.5. Sites are Strict

```
{- A site call is strict. -}

Println(Rwait(500) >> "Waited 0.5 seconds" | Rwait(1000) >> "Waited 1 second") >>

{-
OUTPUT:
Waited 0.5 seconds
-}
```

Example 2.6. Functions are Lenient

```
{-
  A function call is lenient.
  Parts of the function that do not need the arguments can execute immediately.
  However, any part that uses the arguments must wait.
-}

def Printfn(s) =
    Println("Immediate")
  | s >> Println("Waiting")

Printfn(Rwait(1000) >> signal) >> stop

{-
OUTPUT:
Immediate
Waiting
-}
```

Example 2.7. Function with Domain and Range

```
{-
  Use a record with an apply member to create
  a function enhanced with .domain and .range members.
-}

val implies =
  def imp(true,false) = false
  def imp(_,_) = true
  {.
    apply = imp,
    domain = [(true,true),(true,false),(false,true),(false,false)],
    range = [true, false]
  .}

each(implies.domain) >(x,y)>
implies(x,y) >z>
member(z, implies.range)

{-
OUTPUT:
true
true
true
true
-}
```

2.4.7. Related Links

Related Reference Topics

- Call Pattern
- Dot Access
- Operators
- Sites and Services
- Records
- Closures
- `def`: Define Function
- Deflation
- Publication
- Silence
- Blocking
- Halting

- Killing
- Helpful Sites
- Polymorphic Calls

Related Tutorial Sections

- Importing Resources

2.5. Dot Access

A *dot access* is an expression that retrieves a named *member* of a value. It consists of a *target* expression E and a *key* K. First, E is deflated to a value v. If the value v has a member named K, that member is published. Otherwise, the expression halts.

Not all values have members. Records have members, as do many sites. A value created by an imported class has a member for each method and field of the corresponding class. A value created by a defined class has a member for each `def` and `def class` declaration in the class.

Like many expressions in Orc, a dot access is simply another form of site call. The key is converted to a special value and passed to the site call as the only argument. The site call publishes the member named by the key if it exists, and halts otherwise.

2.5.1. Syntax

[7] DotAccess ::= Expression `.` Key

2.5.2. Type

If the target expression E has the record type $\{. K_0 :: T_0, \ldots, K_n :: T_n .\}$, then the dot access $E.K_i$ has type T_i.

If the target has a Java class or object type, the usual Java typing rules apply.

If the target is a site, then the type of the dot access is determined entirely by the site, like any other site call.

2.5.3. Examples

Example 2.8. Record Access

```
{- Add two members of a record -}

val displacement = {. dx = 3.0, dy = 0.3, dz = 1.1 .}

displacement.dx + displacement.dy

{-
OUTPUT:
3.3
-}
```

Example 2.9. Channel Operations

```
{- Create a channel, and perform some operations on it. -}

val b = Channel()

  b.get() >x> b.put(x+1) >> stop
| b.get() >y> b.put(y*2) >> stop
| b.put(3) >> stop
  ;
  b.get()

{-
OUTPUT:
7
-}
{-
OUTPUT:
8
-}
```

Example 2.10. Capture Channel Methods

```
{- Access the 'put' and 'get' members of a channel, and use them as separate sites

val c = Channel()
val send = c.put
val receive = c.get

map(send, [1, 2, 3]) >>
signals(3) >>
receive() >x>
Println("Received: " + x) >>
stop

{-
OUTPUT:PERMUTABLE:
Received: 1
Received: 2
Received: 3
-}
```

2.5.4. Related Links

Related Reference Topics

- Records
- `def class`: Define Site in Orc
- `import class`: Import Class from Java

- Java Sites

Related Tutorial Sections

- The . notation

2.6. Operators

Orc has a standard set of infix, prefix, and postfix *operators*. Operators in Orc are syntactic sugar for site calls; they publish at most once, and their operands are deflated. For example, `1 + 2` is equivalent to `(+)(1, 2)` and `1 + 2 * 3` is equivalent to `(+)(1, (*)(2, 3))`.

Operators are parsed using the precedence and associativity rules of Orc. In particular, these rules describe the relationship between operators and combinators.

2.6.1. Syntax

[8] PrefixOperation ::= PrefixOperator Expression
[9] InfixOperation ::= Expression InfixOperator Expression
[10] PostfixOperation ::= Expression PostfixOperator

2.6.2. Standard Orc Operators

2.6.2.1. Arithmetic Operators

`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division
`%`	Modulus
`**`	Exponentiation
`-` (unary prefix form)	Arithmetic negation

Numeric literals with no decimal part, such as `3`, are treated as integers. Arithmetic operators with two integer arguments will perform an integer operation and return an integer result; for example, `5 / 2` performs integer division and returns `2`. However, if either argument to an operator has a decimal part (even if it is trivial, as in `3.0`), the other argument will be promoted, and a decimal operation will be performed. For example, `5 / 2.0` and `5.0 / 2` both perform decimal division and return `2.5`.

2.6.2.2. Comparison Operators

`=`	Equal to
`/=`	Not equal to
`<:`	Less than
`:>`	Greater than
`<=`	Less than or equal to
`>=`	Greater than or equal to

The `=` operator can compare values of any type. Values of different type are always unequal; for example, `10 = true` publishes `false`.

2.6.2.3. Logical Operators

`&&`	Logical and
`||`	Logical or
`~`	Logical negation

2.6.2.4. String Operators

`+`	Concatenation

2.6.2.5. List Operators

`:`	List construction

2.6.2.6. Record Operators

`+`	Extension

2.6.2.7. Reference Operators

`?`	Dereference
`:=`	Assignment

2.6.3. Redefining Operators

The operators' syntax (fixity, precedence, and associativity) is fixed by the Orc grammar; however, the site called is defined by the library and can be changed by a `def` or `import site` declaration.

Orc permits a special identifier form for operators: the operator name surrounded by parenthesis, as in `(+)`. To disambiguate the unary prefix operator `-` from the binary infix operator `-`, Orc translates the unary prefix operator as `(0-)`. Binding a value to such an identifier redefines the corresponding operator in the scope of that binding.

2.6.4. Type

Assuming that an operator has not been redefined, it obeys the following typing rules:

- An arithmetic operation publishes an `Integer` if all of its operands have type `Integer`. If any operand has type `Number` (but not `Integer`), the operation publishes a `Number`.
- A comparison operation allows operands of any type, and publishes a `Boolean`.
- A logical operation allows only `Boolean` operands, and publishes a `Boolean`.
- A string concatenation requires at least one `String` operand. Other operands may be of type `Top`; they are converted to strings. String concatenation publishes a `String`.
- List construction is polymorphic. It takes an operand of type *`T`* and an operand of type `List[`*`T`*`]`. It returns a value of type `List[`*`T`*`]`.
- Record extension takes two records and publishes a record.

- Dereference takes an operand of type `Ref[T]` and publishes a *T*.
- Assignment takes operands of types `Ref[T]` and *T*, and publishes a `Signal`.

2.6.5. Examples

Example 2.11. Operator precedence

```
1 + 2 * 3

{-
OUTPUT:
7
-}
```

Example 2.12. Operators publish once

```
1 + (2 | 3)

{-
OUTPUT:
3
-}
{-
OUTPUT:
4
-}
```

Example 2.13. Redefine an operator

```
{- Redefine the "||", "~", and "?" operators -}

def (||)(x,y) = x + ", or " + y
def (~)(x) = "not " + x
def (?)(r) = r + ", that is the question."

signal >> ("To be" || ~ "to be")?

{-
OUTPUT:
"To be, or not to be, that is the question."
-}
```

2.6.6. Related Links

Related Reference Topics

- Precedence and Associativity Table
- Site Calls

- Numerics
- Strings
- Booleans
- Lists
- Records
- `Ref` [183]

Related Tutorial Sections

- Operators

2.7. `if then else`

The expression `if` E_c `then` E_t `else` E_f is a conditional expression. It executes as follows:

- If E_c deflates to `true`, execute E_t.
- If E_c deflates to `false`, execute E_f.
- If E_c deflates to a non-Boolean value, halt.
- If E_c halts silently, halt.

2.7.1. Syntax

[15] Conditional `::=` `if` Expression `then` Expression `else` Expression

2.7.2. Type

The type of `if` E_c `then` E_t `else` E_f is the join of the types of E_t and E_f. Additionally, E_c must have type `Boolean`.

2.7.3. Examples

Example 2.14. Binary Search in a Sorted Array

```
{-
  Binary search in a sorted array.
-}

def binary_search(x,a) =
  def searchIn(lo, hi) =
    if (lo >= hi) then
      false
    else (
      val mid = (lo+hi) / 2
      val y = a(mid)?
      if (x = y) then
        true
      else if (x <: y) then
        searchIn(lo, mid)
      else
        searchIn(mid+1, hi)
    )
  searchIn(0, a.length?)

val a = Array(15)

for(0, 15) >i>
a(i) := 2*i >>
stop ;

binary_search(19, a) | binary_search(22, a)

{-
OUTPUT:PERMUTABLE
true
false
-}
```

2.7.4. Related Links

Related Reference Topics

- Booleans
- `Ift` [147]
- `Iff` [147]

2.8. `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 without the keyword `def` and replacing the function name with the keyword `lambda`, that definition becomes an expression which publishes a closure.

Note that a `lambda` cannot create a recursive function, since the function is not given a name in the body.

2.8.1. Syntax

[16] Lambda **::=** `lambda` TypeParameters**?** Parameters ReturnType**?** **=** Expression

2.8.2. Examples

Example 2.15. One Two Sum

```
{- Define a function that sums the results of its argument function
   evaluated with arguments 1 and 2
-}

def onetwosum(f) = f(1) + f(2)

onetwosum( lambda(x) = x * 3 )

{-
  identical to:
  def triple(x) = x * 3
  onetwosum(triple)
-}

{-
OUTPUT:
9
-}
```

2.8.3. Type

The type of a `lambda` expression is exactly the type of the closure it creates.

2.8.4. Related Links

Related Reference Topics

- Closures
- `def`: Define Function
- Higher-Order Idioms

- `map` function [163]
- `filter` function [164]

Related Tutorial Sections

- First-Class Functions

Chapter 3. Combinators

Orc has four *combinators*: parallel, sequential, pruning, and otherwise. A combinator forms an expression from two component expressions. Each combinator captures a different aspect of concurrency. Syntactically, the combinators are written infix, and have lower precedence than operators or calls, but higher precedence than other expression forms.

3.1. Parallel Combinator

F | *G*
: Execution of expression *F* | *G* occurs by executing *F* and *G* concurrently. Whenever *F* or *G* communicates with a service or publishes a value, *F* | *G* does so as well. Therefore, *F* | *G* interleaves the publications of *F* and *G* arbitrarily.

3.1.1. Syntax

[11] Parallel ::= Expression | Expression

Combinator Precedence Level: sequential > parallel > pruning > otherwise [Full Table]

3.1.2. Notable Identities

F | *G* | *H* = (*F* | *G*) | *H* (Left Associative)

F | *G* | *H* = *F* | (*G* | *H*) (Right Associative)

F | *G* = *G* | *F* (Commutative)

3.1.3. Type

The type of *F* | *G* is the join of the types of *F* and *G*.

3.1.4. Examples

Example 3.1. Parallel Publication

```
{- Publish 1 and 2 in parallel -}

1 | 1+1

{-
OUTPUT:PERMUTABLE
1
2
-}
```

Example 3.2. Parallel Site Calls

```
include "search.inc"

{- Access two search sites, Google and Yahoo, in parallel.

   Publish any results they return.

   Since each call may publish a value, the expression
   may publish up to two values.
-}

Google("cupcake") | Yahoo("cupcake")
```

3.1.5. Related Links

Related Reference Topics

- Publication
- Join

Related Tutorial Sections

- Parallel Combinator
- Patterns

3.2. Sequential Combinator

F >x> G
: The execution of *F >x> G* starts by executing *F*. Whenever *F* publishes a value, a new execution of *G* begins in parallel with *F* (and with any previous executions of *G*); in that execution of *G*, variable *x* is bound to the value published by *F*. Any value published by any executions of *G* is published by the whole expression, but the values published by *F* are not published by the whole expression; they are consumed by the variable binding.

F >P> G
: The sequential combinator may be written as *F >P> G*, where *P* is a pattern instead of just a variable name. Any value published by *F* is matched against the pattern *P*. If this match is successful, a new execution of *G* begins, with all of the bindings from the match. Otherwise, the published value is simply ignored, and no new execution of *G* is begun.

F >> G
: This is equivalent to using a wildcard pattern: *F >_> G*. Every publication of *F* will match the combinator pattern, causing an execution of *G* for every individual publication of *F*. No bindings will be made in *G* from these publications.

3.2.1. Syntax

[12] Sequence **::=** Expression **>**Pattern**?>** Expression

Combinator Precedence Level: sequential > parallel > pruning > otherwise [Full Table]

3.2.2. Notable Identities

F >P> G >P> H = F >P> (G >P> H) (Right Associative)

3.2.3. Type

The type of *F >P> G* is the type of *G* in the context Γ_F, where Γ_F is the result of matching the pattern *P* against the type of *F*.

3.2.4. Examples

Example 3.3. Variable Binding

```
{- Publish 1 and 2 in parallel -}

(0 | 1) >n> n+1

{-
OUTPUT:PERMUTABLE
1
2
-}
```

Example 3.4. Filtering

```
{- Filter out values of the form (_,false) -}

( (4,true) | (5,false) | (6,true) )  >(x,true)> x

{-
OUTPUT:PERMUTABLE
4
6
-}
```

Example 3.5. Suppressed Publication

```
{- Print two strings to the console,
   but don't publish the return values of the calls.
-}

Println("goodbye") >>
Println("world") >>
stop

{-
OUTPUT:
goodbye
world
-}
```

3.2.5. Related Links

Related Reference Topics

- Publication
- Silence
- Patterns
- `signal`

Related Tutorial Sections

- Sequential Combinator
- Patterns
- Sites

3.3. Pruning Combinator

F <*x*< *G*
: The execution of *F* <*x*< *G* starts by executing *F* and *G* in parallel. Whenever *F* publishes a value, that value is published by the entire execution. When *G* publishes its first value, that value is bound to *x* in *F*, and then the execution of *G* is immediately killed. A killed expression cannot call any sites or publish any values. During the execution of *F*, any part of the execution that depends on *x* will be blocked until *x* is bound (to the first value published by *G*). If *G* never publishes a value, those parts remain blocked forever.

F <*P*< *G*
: The pruning combinator may include a full pattern *P* instead of just a variable name. Any value published by *G* is matched against the pattern *P*. If this match is successful, then *G* is killed and all of the bindings of pattern *P* are made in *F*. Otherwise, the published value is simply ignored and *G* continues to execute.

F << *G*
: This is equivalent to using a wildcard pattern, *F* <_< *G*. *G* continues to execute until it publishes a value. Any value published by *G* will match the pattern. After the successful match, *G* is killed, but no bindings are made in *F*. No part of execution of *F* is suspended by the pruning combinator since there is no variable to be bound.

3.3.1. Syntax

[13] Prune ::= Expression <Pattern?< Expression

Combinator Precedence Level: sequential > parallel > pruning > otherwise [Full Table]

3.3.2. Notable Identities

F <*P*< *G* <*P*< *H* = (*F* <*P*< *G*) <*P*< *H* (Left Associative)

3.3.3. Type

The type of *F* <*P*< *G* is the type of *F* in the context Γ_G, where Γ_G is the result of matching the pattern *P* against the type of *G*.

3.3.4. Examples

Example 3.6. Exclusive Publication

```
{- Simulate a coin toss by publishing either "heads" or "tails" arbitrarily -}

x <x< ("heads" | "tails")

{-
OUTPUT:
"heads"
-}
{-
OUTPUT:
"tails"
-}
```

Example 3.7. Print First Result

```
include "search.inc"

{- Query Google and Yahoo for a search result
   Print out the result that arrives first; ignore the other result
-}

Println(result) <result< ( Google("cupcake") | Yahoo("cupcake") )
```

Example 3.8. Pattern Publication

```
{- Publish either 9 or 25, but not 16. -}

x*x <(x,true)< ( (3,true) | (4,false) | (5,true) )

{-
OUTPUT:
9
-}
{-
OUTPUT:
25
-}
```

Example 3.9. Timed Termination

```
{- Print all publications of the metronome function for 90 msec
   (after the execution of metronome starts).
   Then kill metronome.  Note that metronome(20) publishes a
   signal every 20 msec.
-}

stop << (metronome(20) >x> Println(x) >> stop | Rwait(90) )

{-
OUTPUT:
signal
signal
signal
signal
signal
-}
```

3.3.5. Related Links

Related Reference Topics

- Publication
- Patterns
- `val` Declaration
- Blocking
- Killing
- Halting
- Deflation

Related Tutorial Sections

- Pruning Combinator
- Patterns
- Sites

3.4. Otherwise Combinator

F ; *G*
: The execution of *F* ; *G* proceeds as follows. First, *F* is executed. If *F* halts, and has not published any value, then *G* executes. If *F* publishes one or more values, then *G* is ignored. The publications of *F* ; *G* are those of *F* if *F* publishes, or those of *G* if *F* is silent.

3.4.1. Syntax

[14] Otherwise ::= Expression ; Expression

Combinator Precedence Level: sequential > parallel > pruning > otherwise [Full Table]

3.4.2. Notable Identities

F ; *G* ; *H* = (*F* ; *G*) ; *H* (Left Associative)

F ; *G* ; *H* = *F* ; (*G* ; *H*) (Right Associative)

3.4.3. Type

The type of *F* ; *G* is the join of the types of *F* and *G*.

3.4.4. Examples

Example 3.10. Fall-back Search

```
include "search.inc"

{- Attempt to retrieve search results from Google.
   If Google does not respond, then use Yahoo.
-}

Google("cupcake") ; Yahoo("cupcake")
```

Example 3.11. Lexicographic sublist

```
{- A call to sum(n, xs), where n is an integer and xs is a list
   of integers, find the first sublist of xs lexicographically
   whose elements add up to n.  The call publishes nothing if
   there is no solution
-}

def sum(0,[]) = []
def sum(n,[]) = stop
def sum(n, x:xs) =
  x:sum(n-x, xs) ; sum(n, xs)

sum(-5,[-2,5,1,4,8,-7])

{-
OUTPUT:
[-2, 4, -7]
-}
```

Example 3.12. Channel Transfer

```
{- Transfer all items from a channel to a list.  Assume that the
   process has exclusive access to the channel, so that no other
   process is adding or removing items.
-}

def xfer(ch) =
  ch.getD() >x> x:xfer(ch) ; []

val ch = Channel()

ch.put(1) >> ch.put(2) >> ch.put(3) >> ch.put(4) >> xfer(ch)

{-
OUTPUT:
[1, 2, 3, 4]
-}
```

Example 3.13. Helpful publication

```
{- Publish a list of all publications of f.
   Assume f is helpful.  Assume you have xfer() from above.
-}

f() >x> b.put(x) >> stop ; xfer(b)

{-
NONRUNNABLE
-}

{- (1 | 2 | 3) >x> c.put(x) >> stop ; xfer(c) outputs
   [1,2,3]
-}
```

3.4.5. Related Links

Related Reference Topics

- Halting
- Publication
- Silence
- Helpful Sites
- `stop`
- Join

Related Tutorial Sections

- Otherwise Combinator
- Patterns
- Sites

Chapter 4. Declarations

A declaration binds variables to values, or type variables to types, in an expression. The values may be as simple as integers or booleans, or as complex as functions, sites, or classes. Declarations follow the rules of lexical scoping [http://en.wikipedia.org/wiki/Lexical_scoping#Lexical_scoping].

4.1. `val`: Bind Value

The `val` declaration binds variables to values. The declaration `val` *P* = *G*, followed by expression *F*, is a different way of writing the expression *F* <*P*< *G*. Thus, `val` shares all of the behavior of the pruning combinator.

4.1.1. Syntax

[21] DeclareVal ::= `val` Pattern = Expression

4.1.2. Examples

Example 4.1. Binding variables to values

```
{- Bind x to true or false arbitrarily
   Bind y to 2
-}

val x = true | false
val y = Rwait(200) >> 2 | Rwait(300) >> 3

"x is " + x + ", " +
"y is " + y

{-
OUTPUT:
"x is true, y is 2"
-}
{-
OUTPUT:
"x is false, y is 2"
-}
```

Example 4.2. Timeout

```
include "search.inc"

{- Publish the result of a Google search.
   If it takes more than 5 seconds, time out.
-}

val result = Google("impatience") | Rwait(5000) >> "Search timed out."

result
```

Example 4.3. Roll Die

```
{- Bind an arbitrary integer between 1 and 6, inclusive,
   to a variable in order to simulate a 6-sided die roll.
-}

val roll = (1 | 2 | 3 | 4 | 5 | 6)
"You rolled " + roll

{-
OUTPUT:
"You rolled 1"
-}
{-
OUTPUT:
"You rolled 2"
-}
{-
OUTPUT:
"You rolled 3"
-}
{-
OUTPUT:
"You rolled 4"
-}
{-
OUTPUT:
"You rolled 5"
-}
{-
OUTPUT:
"You rolled 6"
-}
```

4.1.3. Related Links

Related Reference Topics

- Pruning Combinator
- Publication
- Patterns
- Blocking
- Killing
- Halting

Related Tutorial Sections

- `val` Declaration

- Val
- Sequential Combinator
- Pruning Combinator
- Patterns

4.2. `def`: Define Function

The `def` declaration defines a function. A function definition consists of an identifier, a sequence of *parameters*, and a *body* expression.

An Orc function behaves much like a function, method, procedure, or subroutine in other programming languages. However, there are two key differences:

- Orc functions are lenient: when a function is called, the argument expressions and the function body are evaluated in parallel.
- Orc functions may publish any number of values, including zero, during the execution of the function body.

Orc functions have additional features, many of them adopted from typed functional programming languages:

- Orc functions may be recursive. A group of functions may be mutually recursive.
- Patterns may be used as function parameters.
- A function may have a guard, which allows the function body to be executed only if a given condition holds.
- A function may be defined by multiple clauses.

4.2.1. Syntax

[22] DeclareDefinition **::=** `def` Variable TypeParameters**?** Parameters ReturnType**?** Guard**?** `=` Expression

[64] Parameters **::=** `(` Pattern `,` **...** `,` Pattern `)`

[24] Guard **::=** `if` `(` Expression `)`

4.2.2. Function Execution

The simplest case of argument binding uses only variables and wildcards as parameters.

When a function is called, the function body executes, and in parallel the argument expressions of the call are deflated.

If the execution of the body encounters a use of a parameter whose corresponding argument expression has not yet published a value, that use blocks until the argument value is available, but the rest of the body continues to execute.

Whenever the execution of the body would publish a value, the function call publishes that value. If execution of the body halts, the function call also halts. As a result, a function call might publish any number of times, including zero.

If the function call is killed, execution of the body expression is also immediately killed.

Because functions are lenient, the following two programs are equivalent:

```
def fn(x,y) = E
fn(G,H)
```

```
val x = G
val y = H
E
```

4.2.3. Patterns as Parameters

A function parameter may be any pattern. A *lenient pattern* is either a variable pattern or a wildcard pattern; such patterns will never fail to match. Any other pattern is a *strict pattern*, which could fail to match.

When a function is called, the call blocks until a value is available for each strict pattern. The values are then matched against the strict patterns. If all of these matches succeed, then the function call executes as described earlier. If any strict pattern match fails, or if any of the argument expressions corresponding to a strict pattern halts, then the function call halts.

Suppose `P` is a strict pattern. The following two programs are equivalent:

```
def fn(x,P) = E
fn(G,H)
```

```
val x = G
val z = H
z >P> E
```

4.2.4. Recursion

Functions can be recursive; that is, the name of a function may be used in its own body.

```
{- A recursive factorial function -}
def fact(n) = if (n <= 1) then 1 else n * fact(n-1)
```

A recursive function might continue executing indefinitely, producing an infinite number of publications.

```
{- Publishes a signal every second, forever -}
def metronome() = signal | Rwait(1000) >> metronome()
```

A set of functions may be mutually recursive by naming each other in their bodies. There is no special keyword for mutual recursion; whenever two or more function definitions are adjacent, they are allowed to mutually recurse.

4.2.5. Guards

A function definition may include a guard, of the form `if (` *E* `)`. When a function is called, and each strict pattern matches successfuly as described earlier, then the guard expression *E* is deflated. If *E* deflates to `true`, then the function body is executed. If *E* deflates to some other value, or halts without publishing a value, then the function call halts silently.

Suppose `P` is a strict pattern and `Gd` is a guard expression. The following two programs are equivalent:

```
def fn(x,P) if (Gd) = E
```

```
fn(G,H)


val x = G
val z = H
z >P> Ift(Gd) >> E
```

4.2.6. Clausal Definition

A function can be defined by a sequence of *clauses*: repeated function definitions with the same identifier but different parameters. Each clause must have the same number of parameters.

When a function with multiple clauses is called, the argument expressions are deflated, and in parallel the first clause is executed. The clause will *fail* under any of the following conditions:

1. One of the parameters is a strict pattern, and that pattern fails to match.

2. One of the parameters is a strict pattern, and the corresponding argument expression has halted silently.

3. There is a guard expression, and it did not deflate to `true`.

If each strict pattern matches successfully, and the guard expression (if present) deflates to `true`, then the corresponding function body executes. If the clause fails, then the next clause is executed. If the last clause fails, then the function call halts silently.

4.2.7. Type

When a function is defined, the function identifier is bound to a closure. A definition must be given additional type information so that the typechecker can deduce the correct function type for the identifier.

4.2.8. Examples

Example 4.4. Available Pairs

```
{-
  Publish pairs of results from three computations, as they become available.
-}
def pairs(x, y, z) = (x, y) | (x, z) | (y, z)

pairs(Rwait(2000) >> 0, 1, Rwait(1000) >> 2)

{-
OUTPUT:
(1, 2)
(0, 1)
(0, 2)
-}
{-
OUTPUT:
(1, 2)
(0, 2)
(0, 1)
-}
```

Example 4.5. Parallel-Or Function

```
{-
  Define a parallel-or function.
-}
def parallelor(x,y) =
  val first = Ift(x) >> true | Ift(y) >> true | (x || y)
  first

parallelor(false, Rwait(1000) >> true)

{-
OUTPUT:
true
-}
```

Example 4.6. Even/Odd Using Mutual Recursion

```
{- Test if a number is even or odd, using mutual recursion -}
def even(n) = Ift(n = 0) >> true
            | Ift(n <: 0) >> odd(n+1)
            | Ift(n :> 0) >> odd(n-1)

def odd(n) = Ift(n = 0) >> false
           | Ift(n <: 0) >> even(n+1)
           | Ift(n :> 0) >> even(n-1)

odd(-4)

{-
OUTPUT:
false
-}
```

Example 4.7. List Head

```
{-
  Publish the head of a nonempty list.
  If the list is empty, halt silently.
-}
def head(h:_) = h

head([2, 3]) | head([])

{-
OUTPUT:
2
-}
```

Example 4.8. List Length

```
{- Find the length of a list -}
def length([]) = 0
def length(_:rest) = length(rest) + 1

length([1, 2, 4])

{-
OUTPUT:
3
-}
```

Example 4.9. List Sum

```
{- Sum the elements of a list -}

def sum([]) = 0
def sum(h:t) = h + sum(t)
sum([1, 2, 3])

{-
OUTPUT:
6
-}
```

Example 4.10. Same-Length Zip

```
{-
  "Zip" a pair of lists together into a list of pairs.
  If the lists are of inequal length, halt silently.
-}
def zip(x:xs, y:ys) = (x, y):zip(xs, ys)
def zip([], []) = []

  zip([0, 1], [false, true])
| zip([1, 2, 3], signal)

{-
OUTPUT:
[(0, false), (1, true)]
-}
```

Example 4.11. Fibonacci Function

```
{- Fibonacci numbers -}

def fib(0) = 1
def fib(1) = 1
def fib(n) if (n :> 1) = fib(n-1) + fib(n-2)

fib(5)

{-
OUTPUT:
8
-}
```

4.2.9. Related Links

Related Reference Topics

- Function Calls
- Patterns
- Publication
- Blocking
- Deflation
- `lambda` Expressions
- Closures
- Adding Type Information to Functions
- `def class`: Define Site in Orc

Related Tutorial Sections

- Functions
- First-class Functions

4.3. `def class`: Define Site in Orc

The `def class` declaration provides the capability to implement sites within Orc itself. It is syntactically similar to the `def` declaration, but serves a different purpose.

The body of a `def class` consists of a sequence of declarations followed by a *goal* expression. Each `def` or `def class` declaration in this sequence is called a *method*; a class must have at least one method.

Orc's `def class` provides encapsulation in a manner similar to classes in object-oriented languages. In addition to colocating methods and the shared resources on which they operate, an Orc class may also encapsulate some computation via the goal expression, colocating methods and resources with some orchestration which manages or monitors those resources.

4.3.1. Syntax

[23] DeclareDefclass `::=` `def class` Variable TypeParameters**?** Parameters ReturnType**?** Guard**?** = Declaration+ Expression

4.3.2. Creating Instances

A `def class` declaration binds its identifier to a new site, which we call an Orc class. When this site is called, it creates and publishes an *instance*, a site which has all of the methods defined in the class.

In parallel, execution of the goal expression begins. Any value published by the goal expression is discarded. The class is a site, meaning that the execution of the goal expression cannot be killed by Orc; it continues independently of the rest of the program.

When an instance is created, each `val` declaration begins to execute, and each method is captured as a closure. This has two important implications:

- Different instances will have different values for each `val` declaration, created by separate computations, and these bindings are captured by the method closures. In particular, this means that an instance could create some mutable resource, such as a mutable cell or a semaphore, which is then manipulated by the methods; different instances will have different underlying mutable resources. Instances are thus very similar to objects in object-oriented languages, and `val` declarations are analogous to private fields.

- Recall that creation of a closure is strict in the closure's free variables. Thus, the creation of the instance may block on some free variable within the class (i.e. a variable bound by `val`), or a free variable in the scope of the class. In practice, this means that instance creation will block on every `val` declaration whose name is mentioned in any method. Furthermore, the goal expression might begin executing even before the instance is published.

4.3.3. Calling Methods

Each instance returned by a call to the class has a method for each `def` or `def class` declaration in the class body. A method is accessed using a dot access; the identifier name of the declaration is used as the key. When a method is called, the corresponding definition executes.

Each method behaves as a site, in particular a helpful site. Therefore,

- A method call, unlike a function call, is strict in all of its arguments.
- A method call publishes at most one value. If the method is a `def`, its first publication is used, and execution of the body continues, but subsequent publications are ignored.
- If the method is a `def`, and execution of its body expression halts silently, then the method call also halts silently, exactly like a helpful site.
- A method call cannot be killed by the pruning combinator. Once the method has been called, its execution continues independently of the rest of the Orc program.

The methods of an instance may be executed concurrently through concurrent invocations; there is no synchronization or serialization mechanism to prevent this. Different calls to the same method may even execute concurrently with each other. If any method uses a shared resource, care must be taken to ensure that different method invocations do not interfere with each other in unexpected ways.

4.3.4. Type

A `def class` declaration is typechecked in the same way as a `def` declaration, with one exception. The return type of a closure created by `def class`, rather than being the type of the body expression, is instead a record type $\{. m_0 :: T_0 , \ldots , m_n :: T_n .\}$, where each m_i is the name of a method, and T_i is its type.

4.3.5. Examples

Example 4.12. Matrix Definition

Orc's standard library supports only one dimensional arrays, and array indices always start at 0. We define a template for a 2-dimensional matrix whose row and column indices range over arbitrary intervals.

```
{- Create a matrix whose indices range from
   (rowlo, collo) to (rowhi, colhi) inclusive,
   with a method to access its elements.
-}

def class Matrix((rowlo, rowhi), (collo, colhi)) =
 val mat = Array((rowhi - rowlo + 1) * (colhi - collo + 1))

 def access(i, j) = mat((i - rowlo) * (colhi - collo + 1) + j)

stop

{- Usage -}
val A = Matrix((-2, 0), (-1, 3)).access
A(-1, 2) := 5 >> A(-1, 2) := 3 >> A(-1, 2)?

{-
OUTPUT:
3
-}
```

Note: We have defined `A` as the "access" method of the defined matrix. This allows us to retrieve and update the matrix elements using traditional notation.

Example 4.13. Create a Write-Once Site

We create a site, `Cell`, that defines a write-once variable. It supports two methods: `read` blocks until the variable has been written, and then it publishes its value; `write(v)` blocks forever if a value has already been written, otherwise it writes `v` as the value and publishes a `signal`.

We use two library sites, `Semaphore` (to ensure blocking of write if a value has been written) and `Ref` to store the value.

```
{- Create a mutable cell site -}

def class Cell() =
  val s = Semaphore(1)
  val r = Ref()

  def write(v) = s.acquire() >> r := v
  def read() = r?

  stop

val c = Cell()

c := 42 >> c?

{-
OUTPUT:
42
-}
```

Example 4.14. Extend Functionality of an Existing Site

The `Channel` site implements an unbounded channel. We add a new method, `length`, that returns the number of items in the channel.

```
{- Extend the pre existing channel site with a length field -}

def class CustomChannel() =
  val ch = Channel()
  val chlen = Counter(0)

  def put(x) = ch.put(x) >> chlen.inc()
  def get() = ch.get() >x> chlen.dec() >> x
  def length() = chlen.value()

  stop

val cc = CustomChannel()

  signals(10) >> cc.put("item") >> stop
| signals(5) >> cc.get() >> stop
; cc.length()

{-
OUTPUT:
5
-}
```

Example 4.15. Managing Concurrent Access

The methods of a class instance may be executed concurrently through concurrent invocations. Concurrent execution may cause interference, as in the example of the `Newset` example. Typically, semaphores are used to restrict access to methods and/or data. We rewrite `Newset` in which all accesses to shared data `ne` are protected using a semaphore.

```
{- Manage access to a set via a site -}

def class Newset(n) =
 val b = BoundedChannel(n)
 val (s , ne) = (Semaphore(1) , Ref(0))

 {- Add an element to the set if it is non-full.
   If the set is full, wait until the set becomes non-full.
   Return a signal on completion.
 -}
 def add(x) = b.put(x) >> s.acquire() >> ne := ne? + 1 >> s.release()

 {- Remove some element from the set if it is non-empty.
   If the set is empty, wait until the set becomes non-empty.
   Return the removed value.
 -}
 def remove() = b.get() >x> s.acquire() >> ne := ne? - 1 >>
   s.release() >> x

 {- Return the size, i.e., the number of elements currently in the set
 -}
 def size() = s.acquire() ne? >x> s.release() >> x

stop

{-
OUTPUT:
-}
```

Example 4.16. Computing with the Goal Expression

All the goal expressions shown so far have been merely `stop`. In the following example, the goal expression of the class initiates an computation in which a number is printed every second; all the publications of the goal expression are ignored.

```
{- Perform a print action with the goal expression
   of a class: counting down from n to 0.
-}

def class countdown(n) =

 def downfrom(i) if (i >= 0) = i | Rwait(1000) >> downfrom(i-1)

 {- Goal -} downfrom(n) >i> Println(i)

val _ = countdown(5)
{- Goal of the whole program -}  stop

{-
OUTPUT:
5
4
3
2
1
0
-}
```

Example 4.17. Stopwatch

```
{-
  An instance of Stopwatch has the following operations:

    start():
      Start a paused stopwatch and publish a signal.
      If the stopwatch is already running, halt instead.
    finish():
      Pause a running stopwatch and publish the time
      that has passed since it was last started.
      If the stopwatch is already paused, halt instead.
-}

def class Stopwatch() =
  val now = Rclock().time
  val checkpoint = Ref(now())
  val (startlock, finishlock) = (Semaphore(0), Semaphore(0))
  def start() =
    startlock.acquireD() >>
      now() >starttime>
      checkpoint := starttime >>
    finishlock.release()
  def finish() =
    finishlock.acquireD() >>
      checkpoint? >starttime>
      now() >endtime>
      checkpoint := endtime >>
    startlock.release() >>
    endtime - starttime

  startlock.release()


val watch = Stopwatch()

watch.start() >> Rwait(400) >> watch.finish() >i>
Rwait(200) >>
watch.start() >> Rwait(100) >> watch.finish() >j>
(i+j >= 500) && (i+j <= 525)

{-
OUTPUT:
true
-}
```

4.3.6. Related Links

Related Reference Topics

- Site Calls
- Dot Access

- Publication
- Record Types
- Helpful Sites
- `def`: Define Function

Related Tutorial Sections

- Defining Classes

4.4. `import site`: Import Site

A *site declaration* makes an Orc site that is present in the environment available for use by the program.

Orc sites provide operations such as arithmetic functions, platform facilities, or Web services for Orc programs to invoke. The Orc standard prelude, which is implicitly included in Orc programs by the compiler, contains declarations for all items in the Orc standard library. Sites beyond those in the standard library are made available by a site declaration. A site declaration names a variable to bind to the site and specifies a location from with the site is obtained (site resolution). In the present Orc implementation, this location is specified as a fully qualified name of a class that can be loaded from the JVM classpath.

4.4.1. Syntax

[26] DeclareSite ::= `import site` Variable = ClassName

4.4.2. Site Resolution

A site is resolved as follows: The site class loader is requested to load the class of the fully qualified name specified in the site declaration. The resolver verifies that the loaded class is a subtype of `orc.values.site.Site`. If so, that class is bound to the variable name in the site declaration. If it is not, the resolver attempts to load the Scala companion module for this class. If this exists and is a subtype of `orc.values.site.Site`, it is bound to the variable name in the site declaration. If it is not found, or is not the expected type, site resolution fails.

The site class loader referred to above is the usual JVM stack of delegating class loaders. In the Orc implementation, the stack is initially the first non-`null` of: 1) the current thread's context class loader, 2) the class loader of the site resolver class, or 3) the JVM system class loader. This loader has a URL class loader stacked on it if a site classpath is configured. Thus, if a site classpath is configured, is is searched first. If no class is found on the site classpath, or the site classpath is not configured, then the normal classpath is searched.

4.4.3. Type

The type of a site is determined entirely by the site itself. It may be representable in Orc, for example as a function type, or it may be an entirely opaque type. The type of a site behaves very much like a site itself during typechecking; it is invoked with argument types and responds with a return type.

The Orc typechecker cannot independently verify that a site's stated type is correct. If the site misrepresents its type, this may result in a runtime type error.

4.4.4. Examples

Example 4.18. Declaring a user-supplied site

```
{-
 - Assume a JVM class file named com.example.MySite is
 - availalble on the classpath, and that it implements
 - the orc.values.site.Site trait.
 -}

import site MySite = "com.example.MySite"

MySite()
```

4.4.5. Related Links

Related Reference Topics

- Standard Library

Related Tutorial Sections

- Importing Resources

4.5. `import class`: Import Class from Java

Orc programs can use services of the platform on which the Orc runtime engine runs. In this implementation of Orc, this platform is the Java Runtime Environment (JRE). Java classes may be imported into Orc programs and called as Orc sites.

The declaration `import class S = "J"` binds the variable `S` to a site representing the Java class `J`. Calls to `S` invoke the corresponding constructor of the Java class `J`. The created class instance is published from the site call. Instance methods may be invoked on this published value. Calls to `S.m` invoke the corresponding class static method of the Java class `J`. The return value of the method is published from the site call. Further details of the Orc-Java interface are in the Java sites section.

4.5.1. Syntax

[27] DeclareClass `::=` `import class` Variable = ClassName

4.5.2. Type

When typechecking declared classes, the Orc typechecker interfaces with the Java type system, so that operations on Java objects typecheck as one would expect in a Java program, with a few exceptions.

- Methods with return type `void` in Java have return type `Signal` in Orc.
- Interfaces are not fully supported.
- Type parameter bounds are not supported.

In addition to declaring a site, the `import class` declaration also declares a type with the same name. It is the Orc type for instances of that Java class. For example, the declaration `import class File = java.io.File` declares a type `File`, which is the type of all instances of `java.io.File`.

If a `import class` declaration binds a generic Java class, then the corresponding Orc type is instead a type operator, and the constructor site takes type parameters.

Subtyping between Java object types is determined by Java's subtyping relation: if one class is a subclass of another, then that type will be a subtype of the other. The typechecker does not implement a true join operation for Java types; it will find a common ancestor, but not the least common ancestor.

4.5.3. Examples

Example 4.19. Invoke a Java constructor and instance method

```
{- Use a Java class to count the number of tokens in a string -}

import class StringTokenizer = "java.util.StringTokenizer"

-- Invoke a constructor:
val st = StringTokenizer("Elegance is not a dispensable luxury, but a quality that

-- Invoke a method
st.countTokens()

{-
OUTPUT:
15
-}
```

Example 4.20. Invoke a Java static method

```
{- Use a Java class to calculate a log value -}

import class JavaMath = "java.lang.Math"

-- Invoke a static method:
JavaMath.log10(42.0)

{-
OUTPUT:
1.6232492903979006
-}
```

Example 4.21. Declaring and instantiating a Java generic class

```
{- Use a Java class to represent tree data structures -}

import class TreeSet = "java.util.TreeSet"

val s = TreeSet[String]()
s.add("Orc") >> s.add("Java") >> s.add("Orc") >> s.size()

{-
OUTPUT:
2
-}
```

4.5.4. Related Links

Related Reference Topics

- Java Sites
- Dot Access
- Site Calls

Related Tutorial Sections

- Importing Resources

4.6. `type`: Declare Type

A type declaration binds a type variable to a type. There are three kinds of type declarations: *aliases*, *imports*, and *datatypes*.

4.6.1. Syntax

[29] DeclareType ::= DeclareTypeAlias
| DeclareTypeImport
| DeclareDatatype

[30] DeclareTypeAlias ::= `type` TypeVariable TypeParameters**?** = Type

[31] DeclareTypeImport ::= `import type` TypeVariable = ClassName

4.6.2. Type Alias

A *type alias* gives an existing type a new name, for the programmer's ease of use. There is no distinction between the alias and the aliased type; they can be used interchangeably. A type alias may not be recursive.

A type alias may have type parameters, in which case it defines a type operator which can be applied to other types.

4.6.3. Type Import

A *type import* gives a name to a type described by some external service. This allows new types to be added to Orc, in much the same way that the `import site` declaration declaration allows new sites to be added to Orc. Type imports are often used in conjunction with the `import site`, to give a name to the type of values produced by the site.

4.6.4. Datatype

A *datatype declaration* defines a new type, called a *sum type*, which is the type of all values produced by any of the declared constructors.

A datatype declaration may be polymorphic, introducing new type variables which may be used within the slots of the constructors. It may also be recursive; the type name itself can be used within the slots of the constructors. A datatype declaration is the only way to define type recursively in Orc.

The datatype declaration also assigns types to each of the constructors that it defines.

4.6.5. Examples

Example 4.22. Aliasing Types

```
{- Define a three-dimensional vector type, and a polymorphic relation type  -}

type Vector = {. x :: Number, y :: Number, z :: Number .}
type Relation[R] = (R,R) => Boolean

{-
NONRUNNABLE
-}
```

Example 4.23. Importing Types

```
{- Import the Cell type of write-once cells, and import Java's URI type -}

import type Cell = "orc.lib.state.types.CellType"
import type URI = "java.net.URI"

{-
NONRUNNABLE
-}
```

Example 4.24. Binary Tree Type

```
{-
   Define a polymorphic binary tree datatype,
   then construct a tree of integers and a
   tree of booleans.
-}

type Tree[T] = Node(Tree[T], T, Tree[T]) | Empty()

val intTree =
  val left = Node(Empty(), 0, Empty())
  val right = Node(Empty(), 2, Empty())
  Node(left, 1, right)

val boolTree =
  val left = Node(Empty(), false, Empty())
  Node(left, true, Empty())

intTree | boolTree

{-
OUTPUT:PERMUTABLE
Node(Node(Empty(), false, Empty()), true, Empty())
Node(Node(Empty(), 0, Empty()), 1, Node(Empty(), 2, Empty()))
-}
```

4.6.6. Related Links

Related Reference Topics

- Algebraic Data Types
- Parametric Types
- Type System Metatheory

Related Tutorial Sections

- Datatypes

4.7. `include`: Include Orc File

It is often convenient to group related declarations into units that can be shared between programs. The `include` declaration offers a simple way to do this. It names a source file containing a sequence of Orc declarations; those declarations are incorporated into the program as if they had textually replaced the include declaration. An include declaration may occur wherever any other declaration occurs, even in a nested scope. An included file may itself contain `include` declarations. Included files may come from local files, any URI recognized by the Java library (http, https, ftp, etc.), and include resources found in the Orc JAR files.

4.7.1. Syntax

[28] DeclareInclude `::=include` FileName

4.7.2. Examples

Example 4.25. fold.inc,used below

```
{- Contents of fold.inc -}

def foldl(f,[],s) = s
def foldl(f,h:t,s) = foldl(f,t,f(h,s))

def foldr(f,l,s) = foldl(f,rev(l),s)
```

Example 4.26. Include a separate file

```
{- This is the same as inserting the contents of fold.inc here -}
include "fold.inc"

def sum(L) = foldl(lambda(a,b) = a+b, L, 0)

sum([1,2,3])
```

4.7.3. Related Links

Related Reference Topics

- Declarations

Related Tutorial Sections

- Importing Resources

Chapter 5. Patterns

Patterns are used in combinators, in `val` declarations, and in clausal definitions of functions, to select values and bind variables to values. A pattern is given by a shape and a set of variables. A *shape* is either a tuple, a list, a record, a call, a literal value, or wildcard (written as _). If the shape describes a structured value (such as a tuple), its components may also be shapes. For example, the shape `(_,3)` describes all pairs whose second element is `3`, and the pattern `(x,3)` binds `x` to the first element of all such pairs.

Note that a pattern may fail to match a value, if it does not have the same shape as that value. When this occurs, the unmatched value is simply discarded.

A pattern such as `(x,y)` may bind multiple variables. However, patterns are *linear*, meaning that a pattern may mention a variable name at most once. For example, `(x,y,x)` is not a valid pattern.

During typechecking, a pattern is matched against a type instead of a value. This match produces a typing context, which associates a type with each variable that occurs in the pattern.

5.1. Literal Pattern

A *literal pattern* matches only the same literal value. It is often used in clausal definitions of functions.

5.1.1. Syntax

[37] LiteralPattern ::= Literal

5.1.2. Type

When a literal pattern is matched against a type T , the type of the literal value must be a subtype of the type T . The match produces an empty typing context, since it binds no variables.

5.1.3. Examples

Example 5.1. Implication by Cases

```
{- Defining logical implication by cases -}

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

implies(false, false)

{-
OUTPUT:
true
-}
```

5.1.4. Related Links

Related Reference Topics

- Boolean literals
- Numeric literals
- String literals

Related Tutorial Sections

- Patterns

5.2. Variable Pattern

When a *variable pattern* *x* is matched against a value *v*, the variable *x* is bound to the value *v*.

5.2.1. Syntax

[38] VariablePattern ::= Variable

5.2.2. Type

When a variable pattern *x* is matched against a type *T*, it produces the typing context { *x* has type *T* }.

5.2.3. Examples

Example 5.2. Sum Pair

```
{- Sum the elements of a pair -}
val (x, y) = (3, 4)

x + y

{-
OUTPUT:
7
-}
```

Example 5.3. Pair to List

```
{- Convert pairs to lists -}

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

{-
OUTPUT:PERMUTABLE:
[1, 5]
[2, 6]
[3, 4]
-}
```

5.2.4. Related Links

Related Reference Topics

- Sequential Combinator
- Pruning Combinator
- `val`: Bind Value

Related Tutorial Sections

- Patterns

5.3. Tuple Pattern

Each element of a *tuple pattern* matches the corresponding element of a tuple value.

5.3.1. Syntax

[39] TuplePattern ::= (Pattern , ... , Pattern)

5.3.2. Type

When a tuple pattern (P_0 , … , P_n) is matched against a tuple type (T_0 , … , T_n), each P_i is matched against the corresponding T_i, producing typing contexts Γ_i. The typing context produced by the whole match is the union of the contexts Γ_i.

5.3.3. Examples

Example 5.4. Filtering

```
{- Publish a signal for each tuple with a first value of true -}

((false, true) | (true, false) | (false, false)) >(true, _)> signal

{-
OUTPUT:
signal
-}
```

Example 5.5. Pattern Publication

```
{- Publish 3, 6, and 9 in arbitrary order -}

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

{-
OUTPUT:PERMUTABLE
3
6
9
-}
```

5.3.4. Related Links

Related Reference Topics

- Tuples

Related Tutorial Sections

- Patterns

5.4. List Pattern

Each element of a *list pattern* matches the corresponding element of a list value.

5.4.1. Syntax

[40] ListPattern ::= `[` Pattern `,` ... `,` Pattern `]`

5.4.2. Type

When a list pattern `[` P_0 `,` … `,` P_n `]` is matched against a list type `List[`T`]`, each P_i is matched against the type T, producing typing contexts Γ_i. The typing context produced by the whole match is the union of the contexts Γ_i.

5.4.3. Examples

Example 5.6. Insertion Sort

```
{- Insertion Sort -}

def insert(x, []) = [x]
def insert(x, y:ys) = if (x <: y) then x:y:ys else y:insert(x,ys)

def sort([]) = []
def sort([x]) = [x]
def sort([x,y]) = if (x <: y) then [x,y] else [y,x]
def sort(x:xs) = insert(x, sort(xs))

sort([3, 1, 4, 1, 5, 9])

{-
OUTPUT:
[1, 1, 3, 4, 5, 9]
-}
```

5.4.4. Related Links

Related Reference Topics

- Cons Pattern
- Lists

Related Tutorial Sections

- Patterns

5.5. Record Pattern

When a *record pattern* is matched against a record value, each key mentioned in the record pattern must have a mapping in the record value, and each such mapped value must match its corresponding pattern. The record value may contain additional keys not mentioned by the record pattern.

5.5.1. Syntax

[41] RecordPattern ::= `{.` Key = Pattern `,` ... `,` Key = Pattern `.}`

5.5.2. Type

When a record pattern `{.` $K_0 = P_0$ `,` … `,` $K_n = P_n$ `.}` is matched against a record type R, each K_i must have a binding in R. Then each P_i is matched against the type bound to K_i in R, producing a typing context Γ_i. The typing context produced by the whole match is the union of the contexts Γ_i.

5.5.3. Examples

Example 5.7. Student Applications

```
{- Use records to test for a given string -}

val applicants =
[
  {. name = "Harry Q. Bovik", college = "Carnegie Mellon University", status = "ac
  {. name = "Fred Hacker", college = "Massachusetts Institute of Technology", stat
  {. name = "D. F. Automaton", college = "Final State College", status = "accepted
]
each(applicants) >a>
(
    a >{. name = n, status = "accepted" .}> Println(n + "'s application was accept
  | a >{. name = n, status = "rejected" .}> Println(n + "'s application was not ac
)

{-
OUTPUT:PERMUTABLE
Fred Hacker's application was not accepted
D. F. Automaton's application was accepted
Harry Q. Bovik's application was accepted
-}
```

5.5.4. Related Links

Related Reference Topics

- Records

Related Tutorial Sections

- Patterns

5.6. Call Pattern

A *call pattern* allows a call to be made within a pattern match.

A pattern $x(P_0, \ldots, P_n)$, is matched against a value v by calling $x.\texttt{unapply}(v)$, and matching each value published by that call against the tuple pattern $(P_0, \ldots, P_n)$. If there is only one pattern P, then P is matched on its own, instead of using a tuple pattern. If there are no patterns, a wildcard pattern is used.

If $x.\texttt{unapply}(v)$ halts silently, or halts without producing any matching values, then the match fails.

If multiple values are published and successfully match, then a *multimatch* occurs: the entire pattern succeeds multiple times. In a function call, the matching clause is executed multiple times, once for each match. In a sequential combinator, the right hand side is executed multiple times, once for each match. In a pruning combinator, one of the matches is chosen arbitrarily.

5.6.1. Syntax

[43] CallPattern ::= Variable (Pattern , ... , Pattern)

5.6.2. Type

When a call pattern is matched against a type S, the `unapply` member of the type S must have the function type `lambda (`T`) :: (`T_0`, ... ,` T_n`)`, where S is a subtype of T. Then each argument pattern P_i is matched against the corresponding type T_i, producing typing contexts Γ_i. The typing context produced by the whole match is the union of the contexts Γ_i.

5.6.3. Examples

Example 5.8. Trees

```
{-
   Build up a small binary tree, then use call patterns to deconstruct the tree an
-}

type Tree = Node(_,_,_) | Empty()

val l = Node(Empty(), 0, Empty())
val r = Node(Empty(), 2, Empty())
val t = Node(l,1,r)

t >Node(l,j,r)>
l >Node(_,i,_)>
r >Node(_,k,_)>
( i | j | k )

{-
OUTPUT:PERMUTABLE
0
1
2
-}
```

Example 5.9. Integer square root

```
{-
   A user-defined call pattern match, using a record with an unapply member.

   The integer square root function, isqrt, returns the square root of a
   perfect square, and halts on any input that is not a perfect square.

   isqrt is then used to define a value 'square' that matches perfect squares.
-}

def isqrt(n) =
  if (n <: 0)
    then stop
    else (
      val root = Floor(n ** 0.5)
      if (n = root*root)
        then root
        else stop
    )

val square = {. unapply = isqrt .}

each([9, 12, 16, 24, 25]) >square(n)> n

{-
OUTPUT:PERMUTABLE:
3
4
5
-}
```

Example 5.10. Factoring Using Multimatch

```
{-
   A user-defined call pattern match, using a record with an unapply member.

   The factors function publishes all nontrivial positive factors of its argument
   (any factor greater than 1 and less than n)

   factors is then used to define a value 'multipleOf' that matches all
   nontrivial positive factors of an integer.
-}

def factors(n) if (n <: 0) = factors(-n)
def factors(n) = for(2, n/2 + 1) >i> Ift(n % i = 0) >> i

val multipleOf = {. unapply = factors .}

30 >multipleOf(n)> n

{-
OUTPUT:PERMUTABLE:
2
3
5
6
10
15
-}
```

5.6.4. Related Links

Related Reference Topics

- Algebraic Data Types
- `unapply` key
- Site and Function Calls

Related Tutorial Sections

- Patterns

5.7. Cons Pattern

The *cons pattern* matches the head and the tail of a nonempty list.

5.7.1. Syntax

[42] ConsPattern `::=` Pattern `:` Pattern

5.7.2. Type

When a cons pattern P_h`:`P_t is matched against a list type `List[`T`]`, pattern P_h is matched against type T, producing typing context Γ_h, and pattern P_t is matched against type `List[`T`]`, producing typing context Γ_t. The typing context produced by the whole match is $\Gamma_h \cup \Gamma_t$.

5.7.3. Examples

Example 5.11. List Deconstruction

```
{- Publish the head and tail of a list as a tuple -}

val a = [1, 2, 3]
a >x:y> (x, y)

{-
OUTPUT:
(1, [2, 3])
-}
```

5.7.4. Related Links

Related Reference Topics

- List Pattern
- Lists

Related Tutorial Sections

- Patterns

5.8. As Pattern

The `as` keyword binds a variable to the value matched by a pattern. It is a more general way of binding variables than using variable patterns alone.

5.8.1. Syntax

[44] AsPattern `::=` Pattern `as` Variable

5.8.2. Type

When *P* `as` *x* is matched against a type *T*, the pattern *P* is matched against *T*, producing typing context Γ. The typing context produced by the whole match is $\Gamma \cup \{x \text{ has type } T\}$.

5.8.3. Examples

Example 5.12. Simplified Fragment

```
{- Consider this initial program fragment, without an 'as' pattern -}
val (a,b) = ((1,2),(3,4))
val (ax,ay) = a
val (bx,by) = b

{- Compared to the following fragment -}
val ((ax,ay) as a, (bx,by) as b) = ((1,2),(3,4))

[ax, ay, a] | [bx, by, b]

{-
OUTPUT:PERMUTABLE
[1, 2, (1, 2)]
[3, 4, (3, 4)]
-}
```

5.8.4. Related Links

Related Reference Topics

- Variable Patterns

Related Tutorial Sections

- Patterns

5.9. Wildcard Pattern

The *wildcard pattern* matches any value and binds no variables.

5.9.1. Syntax

[36] WildcardPattern `::=_`

5.9.2. Type

A wildcard pattern matches any type. The matching produces an empty typing context.

5.9.3. Examples

Example 5.13. Wildcard Assignments

```
{- Showcase various wildcard assignments -}

val (_,(_,x),_) = (0,(2,2),[5,5,5])
val [[_,y],[_,z]] = [[1,3],[2,4]]

[x, y, z]

{-
OUTPUT:
[2, 3, 4]
-}
```

Example 5.14. Implication by Fewer Cases

```
{-
  Defining logical implication by cases,
  using wildcard to abbreviate the 'true' cases.
-}

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

implies(true, true)

{-
OUTPUT:
true
-}
```

5.9.4. Related Links

Related Reference Topics

- Sequential Combinator, `>>` Form

Related Tutorial Sections

- Patterns

Chapter 6. Sites and Services

Orc programs communicate with their environment by calling *sites*. Sites perform services, which the Orc program orchestrates. A site call may return a single value, or may be silent, returning no value. Beyond this, there are no restrictions on sites' behavior. In prticular, site calls may interact with other site calls, have side effects, or continue to run after returning a value.

Orc has no built-in services, so even operations as simple as addition are performed by sites. Sites may also provide complex services, such as a database management system.

Sites may be provided by various types of software components, such as Java classes, Web services, or custom Orc sites. An Orc program's views of sites, however, remains uniform across these types.

Some sites, called *classes*, create other sites when called. Orc programs may import previously-defined sites and classes for use, through use of the `import site` and `import class` declarations. Programs may also define classes directly in Orc code, using the `def class` declaration.

6.1. Library sites

The Orc Standard Library provides a variety of sites, functions, and types for use in Orc programs. These sites are declared in the Orc *prelude*, a sequence of declarations which is implicitly included in all Orc programs.

Orc programs are expected to rely on the host language and environment for all but the most essential sites. For example, in the Java implementation of Orc, the entire Java standard library is available to Orc programs via `import class` declarations. Therefore, the Orc standard library aims only to provide convenience for the most common Orc idioms, not the complete set of features needed for general-purpose programming.

6.1.1. Examples

Example 6.1. Using a standard library site

```
{- The Floor site, also called the greatest integer function or integer value -}

Floor(2.5)

{-
OUTPUT:
2
-}
```

Example 6.2. Another standard library site: Ift

```
{- Orc's conditional site: Ift(), publishes a signal if the argument is true -}

Ift(1 :> 0) >> "correct"

{-
OUTPUT:
"correct"
-}
```

6.1.2. Related Links

Related Reference Topics

- Standard Library
- Helpful Sites

6.2. Java sites

Essential to Orc's role as an orchestration language is Orc's interaction with its host platform, which in this implementation is the Java virtual machine (JVM). In addition to calling Orc sites *per se*, Orc programs can access arbitrary Java classes using the `import class` declaration. Values in Java fields, values returned from Java methods, and values returned from Java constructors may be used in Orc programs just as any other Orc value.

6.2.1. Java classes

Java classes are named in an Orc program using the `import class` declaration. In the scope of such a declaration, the declared name is bound to a value that acts as a proxy for the Java class. This class proxy may be called as an Orc site, which invokes the Java class constructor corresponding to the arguments of the site call. The class proxy also presents the class static methods and class static fields in the same manner as an Orc record, with keys corresponding to the Java class members' names. Class methods appear as record elements that are sites, and are invoked by projecting the record element using the dot notation and calling the site. Fields appear as record elements that are references [183], and are accessed and assigned by projecting the record element using the dot notation and using Orc's `?` and `:=` operators.

Note that Java allows fields and methods with identical names to be members of the same class. In this case, Orc's Java class proxy resolves from the usage of a record element whether to access the field or method.

Orc Java class proxies are not true Orc record values, although they appear syntactically as if they are.

Java classes are loaded from the Orc site classpath, which may be specified as a setting of the Orc runtime engine; for example via a command-line option or an Eclipse project property. If the class is not found on the Orc site classpath, the loading attempt continues using the normal Java classpath.

6.2.2. Java objects

Java objects may be returned by any Orc site call, including constructor or class static method invocations of an Orc Java class proxy. Java objects' methods and fields appear in the same manner as an Orc record with keys corresponding to the members' names. Methods appear as record elements that are sites, and are invoked by projecting the record element using the dot notation and calling the site. Fields appear as record elements that are references, and are accessed and assigned by projecting the record element using the dot notation and using Orc's `?` and `:=` operators. If a Java object is called as a site without projecting a member, the method name `apply` is implicitly used.

Note that Java allows fields and methods with identical names to be members of the same class. In this case, Orc's Java object proxy attempts to resolve from the usage of a record element whether to access the field or method.

If a field's value is a class with a member named `read` this member will be invoked when an Orc program accesses that field with the `?` operator. Similarly, if a field's value is a class with a member named `write`, this member will be invoked when an Orc program assigns a new value to that field with the `:=` operator. Note that this is a potentially surprising name conflict.

6.2.3. Java value conversions

When interacting with Java classes and objects, Orc performs some conversions of values passed to and from the Java code. Specifically, Orc applies conversions to the following:

- Arguments of invoked constructors and methods
- Return values of invoked constructors and methods
- Accessed values from fields
- Assigned values to fields
- Accessed values from array components
- Assigned values to array components

The conversions applied are the following:

- `void` Java methods return `signal` in Orc.
- Orc integers are converted to Java `Byte`, `Short`, `Integer`, `Long`, `Float`, or `Double`, as needed.
- Orc numbers are converted to Java `Float` or `Double`, as needed.
- Java `Byte`, `Short`, `Integer`, and `Long` are converted to Orc integers.
- Java `Float` and `Double` are converted to Orc numbers.
- Java primitive values are boxed and unboxed as needed, per *The Java Language Specification [http://java.sun.com/docs/books/jls/third_edition/html/conversions.html#5.1.7]* §5.1.7 and §5.1.8.
- Java widening primitive conversions are applied as needed, per *The Java Language Specification [http://java.sun.com/docs/books/jls/third_edition/html/conversions.html#5.1.2]* §5.1.2.

Note that Orc integers and numbers represent a larger range of values than the standard Java numeric types. If an integer conversion is applied to an out-of-range value, the result is the least significant bits of the out-of-range value. This will change the magnitude of the number and may change its sign. If a floating-point conversion is applied to an out-of-range value, the result is positive or negative infinity, as appropriate.

6.2.4. Java method and constructor invocation

Orc invokes a Java method or constructor by closely approximating the Java method invocation rules specified in *The Java Language Specification [http://java.sun.com/docs/books/jls/third_edition/html/expressions.html#15.12]* §15.12. This is only an approximation, because Orc performs Java compile-time steps at run time using Java's reflection facilities, and therefore has a different view of the types of values than the Java compiler. However, in most cases, this difference has no effect. At present, Orc does not recognize Java variable arity methods. (In practice, this is a very rarely used feature of Java.)

Orc has a Java stack trace option, which may be specified as a setting to the Orc runtime engine (for example, via a command-line option or an Eclipse run configuration setting). This may be helpful when debugging Java invocations.

6.2.5. Java arrays

Java arrays are accessible in Orc. To refer to a component of a Java array, follow the array value with a nonnegative integer-valued expression in parenthesis. Java arrays are zero-origin. Array components appear as references in Orc. Their values may be accessed and assigned using Orc's `?` and `:=` operators. The Java `length` pseudo-field of arrays is available in Orc as `length?`.

6.2.6. Examples

Example 6.3. Construct a Java object, and invoke methods

```
{- Name a Java class and use it to invoke object methods -}
import class URL = "java.net.URL"

{- Create an instance-}
val googleUrl = URL("http://www.google.com/")

{- Invoke some methods -}
googleUrl.openConnection().getResponseMessage()

{-
OUTPUT:
"OK"
-}
```

Example 6.4. Accessing class members

```
{- Access Java's String and Boolean classes -}

import class JavaBoolean = "java.lang.Boolean"
import class JavaString = "java.lang.String"

-- Static field access
JavaBoolean.TRUE? |

-- Constructor invocation
JavaBoolean("true") |

-- Static method access
JavaBoolean.parseBoolean("true") |

-- Overloading and Orc-Java conversion -- String.valueOf(double) is invoked
JavaString.valueOf(2e108)

{-
OUTPUT:PERMUTABLE
true
true
true
"2E+108"
-}
```

Example 6.5. No-arg constructor invocation, field assignment, and field dereference

```
{- Constructor invocation -}

import class FieldTestClass = "org.omg.CORBA.portable.ServantObject"

-- No-arg constructor invocation
FieldTestClass() >testInstance>

-- Field assignment
testInstance.servant := "test 4" >>

-- Field derefence
testInstance.servant?

{-
OUTPUT:
"test 4"
-}
```

Example 6.6. Integer conversion overflow

```
{- Demonstrate an Orc-to-Java integer conversion of an out-of-range value -}

import class JavaInteger = "java.lang.Integer"

val x = 1230000000000000000456789

JavaInteger.valueOf(x)

{-
OUTPUT:
-1530464171
-}
```

6.2.7. Related Links

Related Reference Topics

- `import class` declaration
- External data values
- Custom sites
- Interacting with Java Types

Related Tutorial Sections

- `import class` declaration

6.3. Web Services

Some sites provide access to remote services, rather than performing local computation. In particular, a site could provide access to a Web service. The Orc library provides sites which perform basic HTTP requests and manipulate JSON and XML data representations; these capabilities are sufficient to interact with many simple web services, especially RESTful services.

6.3.1. HTTP

The `HTTP` [205] site provides a simple mechanism to send GET and POST requests to a URL.

- `HTTP(`*`U`*`)`, where *`U`* is a `java.net.URL`, publishes a site which accepts HTTP requests on the URL *`U`*.
- `HTTP(`*`S`*`)` parses the string *`S`* as an absolute URL *`U`*, and then behaves as `HTTP(`*`U`*`)`.
- `HTTP(`*`S`*`,`*`Q`*`)` maps the record *`Q`* to a URL query string *`QS`* by translating each record binding to a query pair, escaping characters if necessary. The call then behaves as `HTTP(`*`S`*`+`*`QS`*`)`.

The `HTTP` site publishes a site (call it *`H`*) with methods `get` and `post`:

- *`H`*`.get()` performs a `GET` request on the URL used to create *`H`*.
- *`H`*`.post(`*`P`*`)` performs a `POST` request with payload *`P`* on the URL used to create *`H`*.

If the request is successful, the response is published as a single string, which may be translated to a more useful representation by other library sites.

6.3.2. Data Processing

Web services often expect inputs or provide outputs in a particular format, such as XML or JSON. The Orc library provides sites which convert between string representations of these data formats and structured representations which Orc can manipulate.

6.3.2.1. JSON

JSON [http://www.json.org] (JavaScript Object Notation) is a lightweight data format often used by Web services, especially by services with simple interfaces. The structure of JSON values maps onto Orc values, using nested lists and records.

The library sites `ReadJSON` [206] and `WriteJSON` [206] convert between string representations of JSON values and Orc representations of those values. `ReadJSON` parses a string representation of JSON and creates an Orc value, where each JSON array becomes an Orc list, each JSON object becomes an Orc record, and literal values map to their Orc counterparts. `WriteJSON` performs the inverse operation, taking a structured value of this form and producing a string representation of an equivalent JSON value. If *`j`* is an Orc representation of a JSON value, then `ReadJSON(WriteJSON(`*`j`*`))` is equal to *`j`*.

6.3.2.2. XML

XML [http://www.w3.org/XML/] (Extensible Markup Language) is a structured data format used in many contexts, including Web services.

The library sites `ReadXML` [207] and `WriteXML` [207] convert between string representations of XML values and Orc representations of those values. `ReadXML` parses a string representation of XML and

creates an Orc representation, which can be manipulated by Orc's XML library sites. `WriteXML` performs the inverse operation, taking an Orc representation of XML and serializing it to a string. If *x* is an Orc representation of an XML fragment, then `ReadXML(WriteXML(`*x*`))` is equal to *x*.

Unlike JSON, the structure of XML does not map directly to Orc's structured values. Instead, Orc provides a set of library sites and functions which manipulate XML in a manner similar to Orc's datatypes. Orc's XML manipulation capabilities are currently incomplete; in particular, it does not handle namespaces. All constructed elements have the default namespace, element and attribute tags have no prefixes, and element matching discards namespace and prefix information. For more comprehensive XML manipulation, use of underlying platform capabilities (such as Scala or Java libraries) is recommended.

6.3.2.2.1. Primitive XML Sites

There are three library sites that construct XML nodes directly. Each of these sites may also be used in a call pattern to match the corresponding constructed value.

- `XMLElement [207](`*tag*`,` *attr*`,` *children*`)` creates a new XML element with the tag given by string *tag*, the set of attributes specified by the record *attr*, and the sequence of child elements given by the list *children*. Each value mapped by *attr* is converted to a string in the XML representation. *attr* may be an empty record. Each element of *children* must be an XML node. *children* may be an empty list.
- `XMLText [207](`*txt*`)` creates a new XML text node whose content is the string *txt*. Characters in *txt* which are not permitted in XML text will be encoded.
- `XMLCData [207](`*txt*`)` creates a new XML CDATA node whose content is the string *txt*. Characters in *txt* are *not* encoded.

The library site `IsXML` [207] verifies that its argument is an XML node of some type. `IsXML(`*x*`)` publishes *x* if *x* is an XML node; otherwise it halts silently.

6.3.2.2.2. Manipulating XML

The library provides two records, `xml` [208] and `xattr` [208], to manipulate XML nodes more conveniently than by using the primitive XML sites. Each record has two members, `apply` and `unapply`, that are both functions. The apply function builds XML values, and the unapply function is used in pattern matching.

`xml` is convenient when working only with the tree structure of XML, and not with the attributes of the element nodes. Note that `xml`'s apply and unapply members are not inverses of each other.

- `xml(`*t*`,` *cs*`)` returns a new XML element with the tag *t*, the list of children *cs*, and no attributes. If any element of *cs* is not an XML node, it is converted to a string and enclosed in a text node.
- The pattern `xml(`p_{tag}`,` p_{child}`)` matches an XML element. The pattern p_{tag} is matched against the element tag. The pattern p_{child} is matched against each child of the element; it is a multimatch, so it may succeed multiple times and produces a result on each success. Additionally, when p_{child} matches a text or CDATA node rather than an element node, it is bound to the string contents of the node, rather than the node itself.

`xattr` is convenient when working only with the attributes of XML element nodes, and not with the overall tree structure.

- `xattr(`*x*`,` *attr*`)` returns a new XML element which is the same as *x* except that all bindings in the record *attr* have been added to the attributes of the element. If *attr* binds any attribute names already bound in *x*, *attr* takes precedence and overwrites those bindings.

- The pattern `xattr(`p_e`, `p_a`)` matches an XML element. The pattern p_e is matched against a copy of the element with no attributes. The pattern p_a is matched against a record containing the attributes of the element.

6.3.3. SOAP Web Services

Though Orc does not provide direct bindings to SOAP web services, it is possible to access such services through Java bindings. Frameworks such as JAX-WS map SOAP web service functionality to Java classes; these classes can then be used as sites in Orc.

6.3.4. Examples

Example 6.7. Random Bytes from Fourmilab

```
{-
   Make a request to the Fourmilab HotBits service
   to produce a sequence of 4 random bytes in hex.
-}

{- Returns a string of n random hexadecimal bytes -}
def randombytes(n) =
  val query = {. nbytes = n, fmt = "xml" .}
  val location = "https://www.fourmilab.ch/cgi-bin/Hotbits"
  val response = HTTP(location, query).get()
  val xml("hotbits", xml("random-data", data)) = ReadXML(response)
  data.trim()

randombytes(4)
```

6.3.5. Related Links

Related Reference Topics

- Standard Library: Web
- Standard Library: XML
- Lists
- Records

6.4. Custom sites

For services beyond those available as library sites, Java sites, and Web service sites, Orc programs may call sites developed specifically as Orc sites. The calling Orc program names the site with an `import site` declaration. This declaration causes the Orc engine to load the Java class of the given name from the Orc site classpath or JVM classpath. Calls to this site are dispatched via Orc's site interface, which permit the site to interact with the calling program natively.

6.4.1. Implementing new custom sites

Orc sites may be implemented in any language that can produce Java class files. However, the Orc team rec ommends the Scala programming language from Martin Odersky and his team at EPFL (École Polytechnique Fédérale de Lausanne). For information on Scala, see http://www.scala-lang.org/.

The Orc runtime engine provides a basic `Site` interface which a site must implement. This interface specifies an abstract `call` method that receives a call handle for the site to respond to the site call. The `Handle` interface provides methods to respond by either:

- Publishing a value
- Halting silently
- Throwing an exception

Additionally, sites may notify the Orc runtime engine of events via the `Handle`, such as the need to write a string to the standard output stream.

Site call arguments and return values are not subject to Orc-Java conversions, so the site must work with native Orc values. Sites are also responsible for enforcing arity and type requirements on the argument lists of calls.

Orc provides a number of convenience mix-in traits for site implementors:

Site interface mix-in traits

`orc.values.sites.Site`
: The basic site trait -- a site that can be called. Implement the `call` method.

`orc.values.sites.TotalSite`
: A site that always publishes. Implement the `evaluate` method instead of the `call` method.

`orc.values.sites.PartialSite`
: A site that sometimes publishes and sometimes halts silently. Implement the `evaluate` method instead of the `call` method. `evaluate` should return an `Option`, `Some(x)` to publish `x`, or `None` to halt.

`orc.values.sites.UnimplementedSite`
: A site that throws `orc.error.NotYetImplementedException` when called or type checked.

`orc.values.sites.TypedSite`
: A site that declares its type to the Orc type checker.

`orc.values.sites.UntypedSite`
: A site that does not participate in type checking. Use sparingly.

For a detailed description of the Orc-site interface, refer to the Orc implementation Javadoc [http://orc.csres.utexas.edu/javadoc/STABLE/index.html].

6.4.2. Related Links

Related Reference Topics

- `import site` declaration

Related Tutorial Sections

- `import site`

Chapter 7. Time

Orc is designed to communicate with the external world, and one of the most important characteristics of the external world is the passage of time. Orc implicitly interacts with the passage of time by calling external services which take time to respond. However, Orc can also explicitly wait for some amount of time, using the special site `Rwait`.

7.1. Real Time

7.1.1. Rwait

The site `Rwait` is a relative timer. It takes as a nonnegative integer argument a number of milliseconds to wait. It waits for approximately that amount of time, and then responds with a `signal`.

7.1.2. Rtime

The site `Rtime` is a program clock. When called, it returns the number of milliseconds that have elapsed since the program began executing.

7.1.3. Rclock

The site `Rclock` creates new clock instances. When called, it returns a new clock, with two members: `wait` and `time`. The `wait` member behaves exactly like `Rwait`. The `time` member returns the number of milliseconds that have passed since the clock was created.

7.1.4. Examples

Example 7.1. Simple example of using Rwait

```
{- Print "red", wait for 3 seconds (3000 ms), and then print "green" -}

Println("red") >> Rwait(3000) >> Println("green") >> stop

{-
OUTPUT:
red
green
-}
```

Example 7.2. Timeout

```
include "search.inc"

{- Publish the result of a Google search.
   If it takes more than 5 seconds, time out.
-}
result
  <result< ( Google("impatience")
           | Rwait(5000) >> "Search timed out.")
```

7.1.5. Related Links

Related Reference Topics

- Standard Library: Time

- `metronome` function [198]

Related Tutorial Sections

- Timeout

Chapter 8. Concepts

These are a few concepts which are essential to Orc, and are used frequently in other parts of the reference manual.

8.1. Publication

An Orc expression *publishes* a value when the expression produces a value to be used in the context enclosing the expression. An extremely simple example of publication is a literal, for example the expression `1`, which publishes a single integer value. Orc expressions may publish zero or more values during their execution. The effect of publication depends on the context. For example, each publication of *F* in the sequential combinator expression *F* `>x>` *G* causes *G* to be run with `x` bound to the published value. An expression that never publishes is said to be silent.

The following is the publication behavior of a few common forms of Orc expressions.

- Site calls publish only zero or one value.
- Function calls may publish any number of values.
- A literal expression publishes exactly one value.
- `stop` never publishes.
- A variable expression, when executed, publishes its value, if bound. If not, execution of the variable is blocked until it is bound.
- Each of the Orc combinators has a unique publication behavior.

8.1.1. Examples

Example 8.1. Publish no values

```
(1 | 2) >> stop
{-
OUTPUT:
-}
```

Example 8.2. Publish one value

```
1
{-
OUTPUT:
1
-}
```

Example 8.3. Publish two values

```
(1 | 2) >x> x + 30
{-
OUTPUT:PERMUTABLE
31
32
-}
```

Example 8.4. Publish an unbounded number of values (metronome)

```
def metronome() = signal | Rwait(1000) >> metronome()

metronome()
```

8.1.2. Related Links

Related Reference Topics

- Combinators
- Site Call
- Function Call
- `stop`
- Silent

Related Tutorial Sections

- Orc expressions
- Combinators
- Nesting expressions
- Sites

8.2. Silence

8.2.1. Definition

An expression is *silent* in an execution if it never publishes in that execution. An expression is always silent if it does not publish in any execution. A silent expression may call sites. The type of a silent expression is `Bot`.

8.2.2. Examples

Example 8.5. Silence with Side Effects

```
{- Silent expressions never publish but may have side effects
   such as output to the console.
-}

  Println("print but don't publish") >> stop
| stop >> Println("never runs")

{-
OUTPUT:
print but don't publish
-}
```

Example 8.6. Conditional Silence

```
{-
  Ift(x) is silent if x is bound to false
  Ift(y) is silent because y is always bound to false
  Ift(z) is silent because z is never bound
-}

val x = true | false
val y = false
val z = stop

Ift(x) | Ift(y) | Ift(z)

{-
OUTPUT:
signal
-}
{-
OUTPUT:
-}
```

8.2.3. Related Links

Related Reference Topics

- Publish

- `stop`
- `Bot`
- Otherwise combinator

8.3. Expression States

A site call, from the site's perspective, has three possible outcomes: the site will definitely publish a value, the site knows that it will never publish a value, and the site does not know if it will ever publish a value. For example, a call to `Ift(true)` publishes a signal, `Ift(false)` never publishes, and `c.get()` on channel `c` that is currently empty may eventually publish or remain non-responsive, depending on whether a value is put in `c` in the future.

A site call, from the caller's perspective, is ready, blocked or halted. A call is ready if all its arguments have deflated to values, so that a call can be made. A call is blocked if either (1) the caller can not make the call because not all argument values are bound (since site calls are strict); this is an *internally blocked* call, or (2) the caller is waiting for the response after having called the site; this is an *externally blocked* call. A site call is halted if the caller has already received a response or the site has indicated that it will never send a response. A site call is killed if it is part of *G* in *F* <x< *G* and *G* publishes.

An internally blocked call becomes ready when all its arguments are bound to values. An internally blocked call halts if one of its arguments will never be bound because the expression that computes its value has halted or has been killed. A ready call becomes externally blocked once the call is made. A blocked call transitions to halted if it receives a response or if the called site can determine that it will never respond; a blocked call may remain blocked forever if the called site can not determine if if it will ever respond. Note that a halted call stays halted unless it is killed.

We extend these concepts to execution of an expression. At any moment, an expression has an associated set of site calls under consideration for execution; if the expression has begun additional executions, as in *F* >x> *G*, all the versions of *G* being executed contribute to this set; if the expression includes function calls, those calls initiate execution of the function bodies which are also expressions that contribute to this set; any variable *x* used as an expression on its own is equivalent to a site call `Let [147](x)`. If some site call in this set is ready, the expression is ready; if all calls are blocked, the expression is blocked, and if all calls are halted the expression is halted.

8.3.1. Ready

A site call is *ready* if all its argument variables are bound to values, so that a call can be made. An expression is ready if some site call in its set of associated site calls is ready.

8.3.2. Blocked

A site call is *blocked* if (1) the call can not be made because some argument of the call is unbound, the call is then internally blocked, or (2) the caller is waiting for a response, the call is then externally blocked. An expression is blocked if its set of associated site calls are all blocked. All component expressions of a blocked expression are blocked. A blocked expression stays blocked unless (1) an internally blocked site call is made ready by the bindings of its arguments, or (2) it is halted, or (3) killed. A halted expression never makes site calls nor publishes any value.

8.3.3. Halted

A site call is *halted* if (1) it was internally blocked and one of its arguments will never be bound because the expression that computes its value has been halted or killed, or (2) it was externally blocked and either a response has been received or an indication that there never will be a response. An expression is halted if the set of associated site calls have all halted. All component expressions of a halted expression are halted. A halted expression stays halted unless it is killed. A halted expression never makes site calls nor publishes any value.

8.3.4. Killed

Expression *G* is *killed* in *F* `<x<` *G* if *G* has published. All component expressions of a killed expression are killed. A killed expression stays killed. A killed expression never makes site calls nor publishes any value.

8.3.5. Helpful Sites

Sites that may indicate absence of response are called *helpful* (see Helpful Sites). Not all sites are helpful.

8.3.6. Examples

Example 8.7. Parallel site calls; ready and blocked states

Let `c` be a channel. Consider expression *G* given by

```
c.get() | Rwait(1000)
```

The expression is ready because it can make both calls. After both calls have been made, the expression is blocked waiting for their responses. Suppose `Rwait(1000)` responds first. Then the expression stays blocked waiting for the response to `c.get()`. If a response is received, the expression halts; if `c` is empty and another caller closes `c`, then `c.get()` indicates that there will be no response, causing *G* to halt; otherwise, *G* stays blocked forever.

Example 8.8. Pruning combinator; killed state

Consider the expression

F `<x<` *G*

where *G* from the previous example is

```
c.get() | Rwait(1000)
```

As we have shown, *G* will definitely publish. Then *G* is killed, and so are its sub-expressions.

Example 8.9. Sequential and otherwise combinators

In the previous example let *F* be

```
x >> c.get() >> true ; false
```

In *F* `<x<` *G*, expression *F* is blocked until `x` is bound to a value. Since *G* eventually publishes, `x` will be bound eventually. Then the call `c.get()` is made in *F*. As we have discussed, this call (1) may receive a response, in which case true will be published, and the entire expression halts, (2) the call receives an indication that there will be no response (in case `c` is empty and it is closed) in which case `x >> c.get() >> true` halts silently, causing false to be published, or (3) the call remains blocked forever, causing *F* to remain blocked.

8.3.7. Related Links

Related Reference Topics

- Expressions

- Site and Function Calls
- `stop`
- Combinators
- Helpful Sites

8.4. Deflation

8.4.1. Definition

Deflation is an execution mechanism which extracts a single value from an expression that might publish many values, so that such an expression can be executed in a context that expects at most one value.

A expression *E* is *deflated* to a value *v* in a context *C* by rewriting

C{*E*}

to

C{*x*} <*x*< *E*

When *E* publishes a value *v*, that value is bound to x and used in the context *C*, and *E* is killed.

The context *C* may contain multiple expressions to be deflated, so this transformation may be applied multiple times. For example, the expression

(*E*, *F*, *G*)

rewrites to

(x, y, z) <*x*< *E* <*y*< *F* <*z*< *G*

If any deflated expression halts silently, then the enclosing expression also halts silently. In the example above, if *F* halted silently, then the expression (*E*, *F*, *G*) would also halt silently.

However, there is one exception to this rule: if *C* is a function call it does not halt, since function evaluation is lenient.

Notice that each deflated expression is evaluated concurrently, due to the behavior of the pruning combinator. This is what makes Orc a functional concurrent language: when an expression is evaluated recursively, all such evaluations take place simultaneously.

8.4.2. Examples

Example 8.10. Search Comparison

```
include "search.inc"

{-
  Return search results from two major search engines in the form of a record.
  Each search has a timeout; if the search engine does not respond by the timeout,
  the result is instead "no result".
-}

{.
  google = Google("Jack Burton") | Rwait(5000) >> "no result",
  yahoo = Yahoo("Jack Burton") | Rwait(7000) >> "no result"
.}
```

8.4.3. Related Links

Related Reference Topics

- Pruning Combinator
- Publication
- Silence
- Killing
- Site and Function Calls
- Operators
- Tuples
- Lists
- Records

Related Tutorial Sections

- Fork-join

8.5. Helpful Sites

A *helpful site* responds with an indication that it will never publish, when it knows this outcome for a call.

All library sites in Orc are helpful. Thus, `Ift` [147] is a helpful site, and `Ift(false)` sends an indication to the caller that it will never publish. A `Channel` [185] is a helpful site, and some of its methods will send an indication that they will never publish. In particular, the `get` method is blocked on an empty channel, but it sends an indication it will never publish if the channel is closed.

The indication from a helpful site that it will never publish causes the call to halt. This can be exploited by the caller by using the otherwise combinator.

We do not expect arbitrary services to be helpful.

8.5.1. Related Links

Related Reference Topics

- Otherwise Combinator
- Halting
- Standard Library

8.6. Approximation in Orc Implementation

This reference manual describes the ideal behavior of Orc programs. In particular, it assumes unbounded memory, an arbitrary amount of concurrent processing power, and any level of speed of computation demanded by the program. Therefore, we have not treated aspects such as floating point overflow, overhead for platform-level thread creation, nor the exact speeds of computation.

For example, the program

```
val z = Rwait(2) >> true | Rwait(3) >> false
```

is ideally expected to bind value `true` to `z`. In practice, we can only make a best effort to implement the ideal behavior of `Rwait`. Given that `Rwait(2)` and `Rwait(3)` are very close in real time, an actual implementation may indeed assign false to `z`. The programmer should be aware of such limitations, and program around them.

8.6.1. Related Links

Related Reference Topics

- `Rwait`
- Numerics

Chapter 9. Type System

The Orc language is dynamically typed. If an operation occurs at runtime which is not type correct, that operation halts silently, and reports an error on the console.

Orc also has an optional static typechecker, which will guarantee that a program is free of type errors before the program is run. For every expression in the program, the typechecker tries to find the types of values that the expression could publish, and then checks that all such types are consistent. The typechecker performs a limited form of type inference, so it can discover many of these types automatically. However, the programmer must provide additional type information for function definitions and for a few other specific cases.

The typechecker is disabled by default, though typed syntax is still permitted (and types are still checked for syntax errors) even when the typechecker is not used. It may be enabled as a project property in the Eclipse plugin, or by using the `--typecheck` switch on the command line.

If the typechecker can verify that a program is correctly typed, it will display the message

`Program type checks as` *`T`*

This means that the program has no type errors, and that every value published by the program will be of type *`T`*.

9.1. Metatheory

The typechecker uses the *local type inference* algorithm described by Pierce and Turner in the paper Local Type Inference [http://www.cis.upenn.edu/~bcpierce/papers/lti-toplas.pdf]. The typechecker extends this algorithm with polymorphic type operators (e.g. `List` or `Channel`), forming a second-order type system. It also includes polymorphic user-defined datatypes and a typing procedure for site calls.

The typechecker supports both generics and subtyping, though it does not currently implement *bounded polymorphism*, which combines the two. Sites may also be overloaded (*ad-hoc polymorphism*), but the programmer cannot write overloaded functions within Orc itself.

The Orc type system currently uses an *erasure* semantics, meaning that type information does not affect runtime behavior, and may be removed from a program after typechecking is finished. In the case of datatype declarations, the type information and constructor bindings are separated by the compiler, so that datatypes may be used in an untyped setting.

9.1.1. Related Links

Related Reference Topics

- Parametric Polymorphism
- Subtyping
- Datatype Declaration

Related Tutorial Sections

- Type Checking

9.2. Parametric Polymorphism

The Orc type system supports *parametric polymorphism*: functions, sites, and types may accept type parameters. This is the same theory that underlies Java's generics.

Parametric polymorphism occurs in three contexts: the declaration of a parametric type, the definition of a polymorphic function, or a call to a polymorphic function or site.

9.2.1. Parametric types

[50] TypeApplication `::=` Type TypeArguments

Orc has a special form of type, called a *type application*, which applies a type operator to a sequence of type arguments. This permits the instantiation of multiple variations of the same type. The simplest example of this feature is the `List` type operator. For example, the list value `[1,2,3]` has the type `List[Integer]`, whereas the list value `[true, false]` has the type `List[Boolean]`.

Lists are not the only parametric type. The standard library includes other parametric types, such as `Option`, `Channel` [185], and `Cell` [182]. A type alias may have type parameters, thus defining a parametric type. Similarly, a datatype declaration may also have type parameters.

9.2.2. Parametric functions

A function may be polymorphic, taking one or more type parameters. Such functions can be operate generically over different types. Consider the following definition of the `append` function, which appends two lists:

```
def append[T](List[T], List[T]) :: List[T]
def append([], l) = l
def append(h::t, l) = h::append(t,l)
```

The function `append` has a type parameter *`T`* in its signature. The type *`T`* is the type of the elements in the lists being appended. Notice that both argument lists must contain the same type of elements. The resulting list contains elements of that same type.

9.2.3. Polymorphic calls

When calling the `append` function, in addition to providing its normal arguments, we must also provide its type argument:

```
append[Integer]([1,2,3], [4,5])
```

However, it would be very burdensome and verbose to provide type arguments to all such calls. Fortunately, in most cases, the type checker can infer the correct type arguments, in the same way that it infers the correct type for many expressions without any additional information. So in this case, we can simply write:

```
append([1,2,3], [4,5])
```

and the typechecker infers that the parameter *`T`* is `Integer`, since both argument lists are of type `List[Integer]`. For a more thorough explanation of how this inference occurs, please refer to the typing algorithm [127] on which the Orc typechecker is based.

Inference of type arguments will always fail on certain kinds of calls, because the typechecker does not have enough information to infer the correct type. The most common case is a site call which constructs a parametric type without taking any arguments. For example, the call `Channel()` will never typecheck, since there is no way for the typechecker to know what type of elements the channel should contain. In other languages such as ML, the typechecker might be able to infer this information from the rest of the program, but Orc's typechecker is based on *local* type inference, which must find the information locally, such as from the types of the arguments. So, to construct a channel that will contain numbers, a type argument must be given: `Channel[Number]()`.

9.2.4. Related Links

Related Reference Topics

- Type System Metatheory
- Variance
- Site and Function Calls
- Adding Type Information to Functions
- Datatype Declaration

Related Tutorial Sections

- Type Checking

9.3. Subtyping

A type S is a *subtype* of a type T, written $S \leq T$, if a value of type S can be used in any context expecting a value of type T.

A type U is a *supertype* of a type T if $T \leq U$.

Subtyping is reflexive: $S \leq S$. Subtyping is also transitive: if $S \leq T$ and $T \leq U$, then $S \leq U$.

If an expression has type S, and $S \leq T$, then that expression also has type T. This is called *subsumption*.

Types in Orc form a bounded lattice. The lattice is ordered by the subtyping relation, its maximal element is the special type `Top`, and its minimal element is the special type `Bot`.

9.3.1. Top

`Top` is the universal type. Every value has type `Top`. Every type is a subtype of `Top`.

9.3.2. Bot

`Bot` is the empty type. No value has type `Bot`. `Bot` is a subtype of every type. An expression has type `Bot` only if it is silent.

`Bot` has an interesting status in Orc. In other typed languages, if an expression has type `Bot`, this usually indicates a guaranteed error, infinite loop, or other failure to return a value. Since sequential programming rarely involves subexpressions that are guaranteed never to return, `Bot` is usually just a curiosity or a formal artifact of the type system, and indeed many type systems do not have a `Bot` type at all.

In Orc, however, `Bot` is very useful, since it is frequently the case that Orc expressions are written to carry out ongoing concurrent activities but never publish any values, and the type system can use the type `Bot` to indicate that no publications will ever be seen from such expressions.

9.3.3. Join

A *common supertype* of two types S and T is any type U such that $S \leq U$ and $T \leq U$. The *join* of S and T is the *least* common supertype of S and T: it is a subtype of every common supertype of S and T.

Some common cases:

- The join of T and T is T.
- If $S \leq T$, then the join of S and T is T.
- The join of T and `Top` is `Top`.
- The join of T and `Bot` is T.

The join of two unrelated types is usually `Top`.

9.3.4. Meet

A *common subtype* of two types S and T is any type U such that $U \leq S$ and $U \leq T$. The *meet* of S and T is the *greatest* common subtype of S and T: it is a supertype of every common subtype of S and T.

Some common cases:

- The meet of T and T is T.
- If $S \leq T$, then the meet of S and T is S.
- The meet of T and `Top` is T.
- The meet of T and `Bot` is `Bot`.

The meet of two unrelated types is usually `Bot`.

9.3.5. Variance

When types contain other types as components, such as a tuple type or a polymorphic type, the subtype relationship between these composite types depends on the subtype relationships between their components. Suppose we have a composite type of the form $C\{T\}$, where T is a type and C is the context in which it appears. The *variance* of the context C is defined in the standard way:

- C is *covariant* if $S \leq T$ implies that $C\{S\} \leq C\{T\}$
- C is *contravariant* if $S \leq T$ implies that $C\{T\} \leq C\{S\}$
- C is *invariant* if $S = T$ implies that $C\{S\} = C\{T\}$.

Tuple types, record types, and list types are all covariant contexts. In a function type, the return type is a covariant context, but the argument types are contravariant contexts. The type parameters of mutable object types are invariant contexts.

The variance of the type parameters in an aliased type or datatype is determined from the declaration itself, by observing the contexts in which the parameters appear. If a parameter appears only in covariant contexts, it is covariant. If it appears only in contravariant contexts, it is contravariant. If it appears in both contexts, it is invariant.

9.3.6. Related Links

Related Reference Topics

- Type System Metatheory

9.4. Adding Type Information

The Orc typechecker uses a type inference algorithm [127] to deduce type information from a program without any help from the programmer. In many contexts, the typechecker can find all of the type information it needs; however, there are some cases where extra information is needed. For this purpose, there are four kinds of type information that may be added to an Orc program.

Note that due to the erasure [127] property of the Orc type system, adding type information will never change the runtime behavior of the program.

9.4.1. Explicit Type Arguments

Type information may be added to a polymorphic site or function call by providing explicit type arguments. The typechecker can usually infer type arguments, but there are certain cases where it does not have enough information, such as when calling polymorphic factory sites like `Channel` [185] or `Ref` [183].

9.4.2. Function Type Information

Whenever a function is defined, in order for the typechecker to determine its type, the definition must be accompanied by information about the argument types and return type of the function. If the function is polymorphic, the names of its type parameters must also be given. This is the same information that a function type carries. There are multiple ways to provide this information, and some of it can be inferred under certain conditions.

The most comprehensive way to provide type information about a function is through a *signature*. A signature precedes a function definition, providing a sequence of type parameters, a sequence of argument types, and a return type.

[25] DeclareSignature ::= `def` Variable TypeParameters**?** ArgumentTypes ReturnType
[22] DeclareDefinition ::= `def` Variable TypeParameters**?** Parameters ReturnType**?** Guard**?** = Expression

Type information may also be written directly into a clause of a function definition. For example, the following definitions are equivalent:

```
{- Adding type information using a signature -}
def min(Number, Number) :: Number
def min(x,y) = if (x <: y) then x else y

{- Inline type information -}
def min(x :: Number, y :: Number) :: Number = if (x <: y) then x else y
```

If the function is not recursive, then the inline return type is optional, because the typechecker can infer the return type from the body expression.

When writing a `lambda` expression, type information must be included in this way, since there is no way to write a separate signature. The return type is not needed, since a `lambda` will never be recursive. The parameter types are required and cannot be inferred, except in one context: when a `lambda` expression appears as an argument to a call which requires no other inference, then the argument types can be inferred from the type of the target.

9.4.3. Pattern Type Information

[45] PatternWithTypeInformation ::= Pattern `::` Type

A pattern may specify the type of values against which it may be matched. The typechecker can then verify this stated type, rather than attempting to infer it, which may provide enough type information to make other inferences possible or resolve ambiguities. Furthermore, adding extra type information makes it easier to pinpoint the source of a typechecking failure. Note that this type information has no effect on the runtime match behavior of the pattern.

9.4.4. Expression Type Information

[18] WithTypeInformation ::= Expression `::` Type

An expression may specify the type of values that it will publish. The typechecker can then verify this stated type, rather than attempting to infer it, which may provide enough type information to make other inferences possible or resolve ambiguities. For example, the typechecker may not be able to infer the correct join type for a parallel combinator, but it is always able to check that both branches are subtypes of an already provided type. Furthermore, adding extra type information makes it easier to pinpoint the source of a typechecking failure.

9.4.5. Related Links

Related Reference Topics

- `def`: Define Function
- `lambda` Expressions
- Patterns

Related Tutorial Sections

- Adding Type Information

9.5. Type Override

[19] TypeOverride ::= Expression :!: Type

While the typechecker can be helpful, it will not accept programs that are not typesafe according to its algorithm. This can be burdensome when the programmer knows that an expression will have a certain type but the typechecker cannot verify it.

Since the typechecker is optional, it can always be turned off in these cases. But this is often too drastic a solution: typechecking difficulties often arise from small segments of a much larger program, and the rest of the program still benefits from typechecking.

The typechecker may be selectively disabled for parts of a program. For this purpose, the typechecker allows an *override* of the type of an expression. Overriding is like adding type information to an expression, but rather than verifying that an expression has the stated type, the typechecker instead assumes that the stated type is correct, without examining the expression at all. Thus, the programmer can supply any type without being restricted by the typechecking algorithm.

This feature should be used sparingly, with the knowledge that it does compromise the integrity of the typechecking algorithm. If the supplied type is wrong, runtime type errors could propagate to any part of the program that depends on that type. Overrides are useful for rapid prototyping, but they are not recommended for production code.

9.5.1. Related Links

Related Reference Topics

- Adding Type Information
- Metatheory

9.6. Typing Contexts

Whenever one or more variables is bound, the type checker records information from that binding to typecheck the uses of that variable in its scope. This information is stored in a *typing context*, which is just a sequence of variable names and their types. When an expression is in the scope of many bindings, the typechecker will carry a composite context of all of the typing contexts created by all of those bindings.

Declarations, combinators, and function calls all bind variables. Often, only a single variable is bound, so only one binding is generated. However, pattern matching may generate an entire set of variable bindings.

When a variable is typechecked, its type is found by first looking in the most recent context (i.e. from the nearest binding), and if its type is not found there, proceeding further and further out in scope. If the variable is not in the typing context, then it is a free variable, which is a syntactic error.

9.6.1. Related Links

Related Reference Topics

- Patterns

Chapter 10. Syntax

These sections explain the Orc syntax and keywords.

10.1. EBNF Grammar

The Orc language grammar is expressed in Extended Backus-Naur Form (EBNF), with the following meta-notation:

G**?** denotes zero or one occurrences of G.

G+ denotes one or more occurrences of G.

G *sep* **...** *sep* G denotes zero or more occurrences of G, separated by *sep*.

[1] Expression ::= Literal
| Variable
| Stop
| Tuple
| List
| Record
| Call
| DotAccess
| PrefixOperation
| InfixOperation
| PostfixOperation
| Parallel
| Sequence
| Prune
| Otherwise
| Conditional
| Lambda
| WithDeclaration
| WithTypeInformation
| TypeOverride
| `(` Expression `)`

[2] Stop ::= `stop`

[3] Tuple ::= `(` Expression `,` **...** `,` Expression `)`

[4] List ::= `[` Expression `,` **...** `,` Expression `]`

[5] Record ::= `{.` Key = Expression `,` **...** `,` Key = Expression `.}`

[6] Call ::= Expression TypeArguments**?** Arguments

[7] DotAccess ::= Expression`.`Key

[8] PrefixOperation ::= PrefixOperator Expression

[9] InfixOperation ::= Expression InfixOperator Expression

[10] PostfixOperation ::= Expression PostfixOperator

[11] Parallel ::= Expression `|` Expression

[12] Sequence ::= Expression >Pattern**?**> Expression

[13] Prune ::= Expression <Pattern**?**< Expression

[14] Otherwise ::= Expression `;` Expression

[15] Conditional ::= `if` Expression `then` Expression `else` Expression

[16] Lambda ::= `lambda` TypeParameters**?** Parameters ReturnType**?** = Expression

[17] WithDeclaration ::= Declaration Expression

[18] WithTypeInformation ::= Expression `::` Type

[19] TypeOverride ::= Expression `:!:` Type

[20] Declaration ::= DeclareVal
| DeclareDefinition
| DeclareDefclass
| DeclareSignature

			❙ DeclareSite
			❙ DeclareClass
			❙ DeclareInclude
			❙ DeclareType
[21]	DeclareVal	: : =	`val` Pattern = Expression
[22]	DeclareDefinition	: : =	`def` Variable TypeParameters**?** Parameters ReturnType**?** Guard**?** = Expression
[23]	DeclareDefclass	: : =	`def class` Variable TypeParameters**?** Parameters ReturnType**?** Guard**?** = Declaration+ Expression
[24]	Guard	: : =	`if` (Expression)
[25]	DeclareSignature	: : =	`def` Variable TypeParameters**?** ArgumentTypes ReturnType
[26]	DeclareSite	: : =	`import site` Variable = ClassName
[27]	DeclareClass	: : =	`import class` Variable = ClassName
[28]	DeclareInclude	: : =	`include` FileName
[29]	DeclareType	: : =	DeclareTypeAlias
			❙ DeclareTypeImport
			❙ DeclareDatatype
[30]	DeclareTypeAlias	: : =	`type` TypeVariable TypeParameters**?** = Type
[31]	DeclareTypeImport	: : =	`import type` TypeVariable = ClassName
[32]	DeclareDatatype	: : =	`type` TypeVariable TypeParameters**?** = Constructor `|` **…** `|` Constructor
[33]	Constructor	: : =	Variable (Slot , **…** , Slot)
[34]	Slot	: : =	Type ❙ _
[35]	Pattern	: : =	WildcardPattern
			❙ LiteralPattern
			❙ VariablePattern
			❙ TuplePattern
			❙ ListPattern
			❙ RecordPattern
			❙ ConsPattern
			❙ CallPattern
			❙ AsPattern
			❙ PatternWithTypeInformation
			❙ (Pattern)
[36]	WildcardPattern	: : =	_
[37]	LiteralPattern	: : =	Literal
[38]	VariablePattern	: : =	Variable
[39]	TuplePattern	: : =	(Pattern , **…** , Pattern)
[40]	ListPattern	: : =	[Pattern , **…** , Pattern]
[41]	RecordPattern	: : =	{. Key = Pattern , **…** , Key = Pattern .}
[42]	ConsPattern	: : =	Pattern : Pattern
[43]	CallPattern	: : =	Variable (Pattern , **…** , Pattern)
[44]	AsPattern	: : =	Pattern `as` Variable
[45]	PatternWithTypeInformation	: : =	Pattern : : Type
[46]	Type	: : =	TypeVariable
			❙ TupleType
			❙ RecordType
			❙ FunctionType
			❙ TypeApplication
			❙ (Type)
[47]	TupleType	: : =	(Type , **…** , Type)
[48]	RecordType	: : =	{. Key = Type , **…** , Key = Type .}
[49]	FunctionType	: : =	`lambda` TypeParameters**?** ArgumentTypes ReturnType
[50]	TypeApplication	: : =	Type TypeArguments

```
[51]                  Literal ::= SignalLiteral
                                | BooleanLiteral
                                | NumberLiteral
                                | CharacterStringLiteral
                                | null
[52]            SignalLiteral ::= signal
[53]           BooleanLiteral ::= true | false
[54]           IntegerLiteral ::= DecimalDigit+
[55]           NumberLiteral  ::= IntegerLiteral DecimalPart? ExponentPart?
[56]            DecimalPart   ::= . IntegerLiteral
[57]            ExponentPart  ::= E IntegerLiteral
                                | E+ IntegerLiteral
                                | E- IntegerLiteral
                                | e IntegerLiteral
                                | e+ IntegerLiteral
                                | e- IntegerLiteral
[58]  CharacterStringLiteral  ::= "Character+" | ""
[59]              Variable    ::= Identifier
[60]          TypeVariable    ::= Identifier
[61]                 Key      ::= Identifier
[62]             ClassName    ::= CharacterStringLiteral
[63]              FileName    ::= CharacterStringLiteral
[64]            Parameters    ::= ( Pattern , ... , Pattern )
[65]        TypeParameters    ::= [ TypeVariable , ... , TypeVariable ]
[66]        ArgumentTypes     ::= ( Type , ... , Type )
[67]            ReturnType    ::= :: Type
[68]             Arguments    ::= ( Expression , ... , Expression )
[69]        TypeArguments     ::= [ Type , ... , Type ]
```

Tuple Size ≥ 2

A tuple must contain at least two elements. No 0-tuples or 1-tuples are permitted.

10.1.1. Related Links

Related Reference Topics

- Lexical Specifications

Related Tutorial Sections

- Syntactic and Stylistic Conventions

10.2. Lexical Specifications

This page specifies Orc's processing of an input byte stream into an Orc lexical token sequence. This token sequence is the input to the Orc parsing procedure.

10.2.1. Input Byte Stream

Orc can read source code input byte streams from a number of types of sources. For example, Orc 2.0 running on Java SE 6 accepts input from local files, FTP, Gopher, HTTP, and JAR files. Orc source code input byte streams must encode a Unicode character sequence using the UTF-8 [http://www.unicode.org/versions/Unicode5.2.0/ch03.pdf] encoding form. No other encoding is supported. HTTP headers specifying other charsets are ignored.

10.2.2. Lexical Tokens

Orc lexical scanning reads Unicode characters and emits corresponding Orc lexical tokens. Orc uses seven lexical token types: identifier, keyword, operator, delimiter, integer literal, floating-point literal, string literal. Orc comments and whitespace are scanned as separators among other tokens and disregarded.

10.2.2.1. Identifier

Orc scans identifiers per Unicode Standard Annex #31, Unicode Identifier and Pattern Syntax [http://unicode.org/reports/tr31/], namely:

- Identifiers start with "Characters having the Unicode General_Category of uppercase letters (Lu), lowercase letters (Ll), titlecase letters (Lt), modifier letters (Lm), other letters (Lo), letter numbers (Nl)".
- Identifiers can continue with "All of the above, plus characters having the Unicode General_Category of nonspacing marks (Mn), spacing combining marks (Mc), decimal number (Nd), connector punctuations (Pc)", plus Orc's addition of apostrophe as a "prime" mark.
- All identifiers are normalized to Unicode Normalization Form C as they are parsed.

Examples of allowed Orc identifiers: orchestrate, iscenesætte, ενορχηστρώνω, امانسجام, ##, #####, ####_####, оркестровать, ##########_####.

Also, mathematical letter-like characters are allowed, such as #, #, $\aleph_0$, and, of course, Greek letters.

Orc also treats an operator (defined below) placed in parenthesis, such as `(+)`, as an identifier with the name of the operator (without the parenthesis).

An identifier cannot match a keyword; see the following section.

10.2.2.2. Keyword

Any token that otherwise follows the rules for identifiers, but matches an entry in the following list is not treated as an identifier, but as a keyword instead.

```
as def else if import include lambda signal stop then type val true
false null _
```

Note that _ is a special case: Identifiers cannot start with an underscore, but _ scans as a keyword nonetheless.

10.2.2.3. Operator

An Orc operator is a character sequence that matches one of the following. This match is greedy — for example, ** is matched in preference to two * operators, if possible.)

```
+ - * / % ** && || ~ < > = <: :> <= >= /= : . ? :=
```

10.2.2.4. Delimiters

An Orc delimiter is a character sequence that matches one of the following.

```
( ) [ ] {. .} , | ; :: :!:
```

10.2.2.5. Literals

Numeric literals and character string literals are recognized per the syntax given in those sections.

10.2.2.6. Comments

Orc comments take two forms:

1. `--to end of the line (see newlines, below)`
2. `{- multi-line comment body -}`

A multi-line comment body is any character sequence (possibly empty), where {- and -} have lexical significance. Orc comments can be nested, so {- starts a nested multi-line comment, and -} ends the current multi-line comment.

10.2.2.7. Whitespace

Per Unicode Standard Annex #31, section 4, recommendation R3 [http://unicode.org/reports/tr31/#Pattern_Syntax], Orc treats all Unicode Pattern_White_Space characters as whitespace.

- U+0009 HT Character Tabulation
- U+000B VT Line Tabulation
- U+0020 Space
- U+200E Left-to-Right Mark
- U+200F Right-to-Left Mark
- And the six newline characters (below)

10.2.2.8. Newlines

Orc follows the newline definition of Unicode standard section 5.8, Newline Guidelines, recommendation R4 [http://unicode.org/versions/Unicode5.2.0/ch05.pdf].

Namely, Orc stops reading a line when it encounters one of the following characters:

- U+000D CR Carriage Return
- U+000A LF Line Feed

- U+0085 NEL Next Line
- U+2028 LS Line Separator
- U+000C FF Form Feed
- U+2029 PS Paragraph Separator

10.2.3. Related Links

Related Reference Topics

- EBNF Grammar
- Numeric values
- Character Strings

Related Tutorial Sections

- Syntactic and Stylistic Conventions

10.3. Precedence, Fixity, and Associativity

Precedence rules specify the order in which parts of an expression are parsed, in absence of parenthesis. For example, in `1 + 2 * 3`, Orc's precedence rules prescribe that the multiplication be parsed as a sub-expression of the addition. Operators or combinators listed in the table below with higher precedence will be parsed in preference to lower precedence operations or combinators.

Fixity specifies the relative position of an operator and it operands. For example, `+` is infix, so the `+` operator is written in between its operands. However, `?` is postfix, so it is written after its operand.

Associativity specifies the grouping of a series of infix operator (or combinator) expressions, in absence of parenthesis. For example, division is left associative, so `40 / 4 / 2` is grouped as `(40 / 4) / 2` which is 5. It is not `40 / (4 / 2)` = 20. Similarly, the sequential combinator is right associative, so `e >x> f >y> g` is equivalent to `e >x> (f >y> g)`.

Table 10.1. Orc Combinator/Operator Precedence, Fixity, and Associativity

Precedence Level	*Name*	*Symbol*	*Fixity*	*Associativity*
13	Call	`()`	Postfix	--
	Dot Access	`.`	Postfix	--
	Dereference	`?`	Postfix	--
12	Arithmetic Negation	`-`	Prefix	--
	Negation (Logical Complement)	`~`	Prefix	--
11	Exponentiation	`**`	Infix	Right
10	Multiplication	`*`	Infix	Left
	Division	`/`	Infix	Left
	Modulus	`%`	Infix	Left
9	Addition/Concatenation	`+`	Infix	Left
	Subtraction	`-`	Infix	Left
8	List Construction	`:`	Infix	Right
7	Equal To	`=`	Infix	None
	Not Equal To	`/=`	Infix	None
	Less Than	`<:`	Infix	None
	Greater Than	`:>`	Infix	None
	Less Than or Equal To	`<=`	Infix	None
	Greater Than or Equal To	`>=`	Infix	None
6	Logical OR	`||`	Infix	Left
	Logical AND	`&&`	Infix	Left
5	Assignment	`:=`	Infix	None
4	Sequential	`>`*P*`>`	Infix	Right
3	Parallel	`|`	Infix	Left
2	Pruning	`<`*P*`<`	Infix	Left

Precedence Level	*Name*	*Symbol*	*Fixity*	*Associativity*
1	Otherwise	`;`	Infix	Left
0	Type Information	`::`	Infix	Left
	Type Override	`:!:`	Infix	Left
	Closure	`lambda`	Prefix	--
	Conditional	`if then else`	Prefix	--
	Declaration	`val, def, import, include, type`	Prefix	--

10.3.1. Related Links

Related Reference Topics

- Operators
- Combinators

Related Tutorial Sections

- Operators

Chapter 11. Standard Library

11.1. Introduction

The standard library is a set of declarations implicitly available to all Orc programs. In this section we give an informal description of the standard library, including the type of each declaration and a short explanation of its use.

Orc programs are expected to rely on the host language and environment for all but the most essential sites. For example, in the Java implementation of Orc, the entire Java standard library is available to Orc programs via `import class` declarations. Therefore the Orc standard library aims only to provide convenience for the most common Orc idioms, not the complete set of features needed for general-purpose programming.

The documentation of library functions uses special notation for types that have dot-accessible members. Member names of an instance of `Type` are written in the form `type.member`, e.g. `foo.get` refers to the `get` member of an object of type `Foo`. The object type can include type parameters which are referenced by the member type, so for example `@method channel[A].get() :: A` means that when the `get` method is called on a value of type `Channel[A]`, it will return a value of type `A`.

The Standard Library makes use of colored tags to quickly convey properties of library sites. The tags and their definitions are as follows:

Site Property Set

Indefinite
A call to this site may block execution of an expression, since it is not guaranteed to always eventually publish a value or halt.

Definite
A call to this site will never block execution of an expression, since it is guaranteed to always eventually publish a value or halt.

Pure
Any call to this site is pure, meaning that it is deterministic, responds immediately, and has no side effects. A call to a pure site may be textually replaced with its return value (or replaced with `stop` if the call halts) in any program context.

Idempotent
The site is idempotent; calling it more than once on the same arguments is equivalent to calling it once on those arguments.

11.2. core: Fundamental sites and operators.

Fundamental sites and operators.

These declarations include both prefix and infix sites (operators). For consistency, all declarations are written in prefix form, with the site name followed by the operands. When the site name is surrounded in parentheses, as in `(+)`, it denotes an infix operator.

For a more complete description of the built-in operators and their syntax, see the Operators article.

11.2.1. `Let`

```
site Let() :: Signal
```

Definite **Pure**

When called with no arguments, returns a signal.

11.2.2. `Let`

```
site Let(A) :: A
```

When called with a single argument, returns that argument (behaving as the identity function).

11.2.3. `Let`

```
site Let(A, ...) :: (A, ...)
```

When called with two or more arguments, returns the arguments as a tuple.

11.2.4. `Ift`

```
site Ift(Boolean) :: Signal
```

Definite **Pure**

Returns a signal if the argument is true, otherwise halts silently.

Example:

```
-- Publishes: "Always publishes"
  Ift(false) >> "Never publishes"
| Ift(true) >> "Always publishes"
```

11.2.5. `Iff`

```
site Iff(Boolean) :: Signal
```

Definite **Pure**

Returns a signal if the argument is false, otherwise halts silently.

Example:

```
-- Publishes: "Always publishes"
  Iff(false) >> "Always publishes"
| Iff(true) >> "Never publishes"
```

11.2.6. `Error`

```
site Error(String) :: Bot
```

Definite

Emits the given string as an error message, then halt silently.

Example, using `Error` to implement assertions:

```
def assert(b) =
  if b then signal else Error("assertion failed")
-- Fail with the error message: "assertion failed"
assert(false)
```

11.2.7. `(+)`

```
site (+)(Number, Number) :: Number
```

`a+b` returns the sum of `a` and `b`.

11.2.8. `(-)`

```
site (-)(Number, Number) :: Number
```

`a-b` returns the value of `a` minus the value of `b`.

11.2.9. `(0-)`

```
site (0-)(Number) :: Number
```

Return the additive inverse of the argument. When this site appears as an operator, it is written in prefix form without the zero, i.e. `-a`

11.2.10. `(*)`

```
site (*)(Number, Number) :: Number
```

`a*b` returns the product of `a` and `b`.

11.2.11. (**)

```
site (**)(Number, Number) :: Number
```

`a ** b` returns a^b, i.e. `a` raised to the `b`th power.

11.2.12. (/)

```
site (/)(Number, Number) :: Number
```

`a/b` returns `a` divided by `b`. If both arguments have integral types, `(/)` performs integral division, rounding towards zero. Otherwise, it performs floating-point division. If `b=0`, `a/b` halts with an error.

Example:

```
  7/3   -- publishes 2
| 7/3.0 -- publishes 2.333...
```

11.2.13. (%)

```
site (%)(Number, Number) :: Number
```

`a%b` computes the remainder of `a/b`. If `a` and `b` have integral types, then the remainder is given by the expression `a - (a/b)*b`. For a full description, see the Java Language Specification, 3rd edition [http://java.sun.com/docs/books/jls/third_edition/html/expressions.html#15.17.3].

11.2.14. (<:)

```
site (<:)(Top, Top) :: Boolean
```

`a <: b` returns true if `a` is less than `b`, and false otherwise.

11.2.15. (<=)

```
site (<=)(Top, Top) :: Boolean
```

`a <= b` returns true if `a` is less than or equal to `b`, and false otherwise.

11.2.16. (:>)

```
site (:>)(Top, Top) :: Boolean
```

`a :> b` returns true if `a` is greater than `b`, and false otherwise.

11.2.17. (>=)

```
site (>=)(Top, Top) :: Boolean
```

`a >= b` returns true if `a` is greater than or equal to `b`, and false otherwise.

11.2.18. (=)

```
site (=)(Top, Top) :: Boolean
```

`a = b` returns true if `a` is equal to `b`, and false otherwise. The precise definition of "equal" depends on the values being compared, but always obeys the rule that if two values are considered equal, then one may be substituted locally for the other without affecting the behavior of the program.

Two values with the same object identity are always considered equal. Orc data structures, such as tuples, are equal if their contents are equal. Other types are free to implement their own equality relationship provided it conforms to the rules given here.

Note that although values of different types may be compared with `=`, the substitutability principle requires that such values are always considered inequal, i.e. the comparison will return `false`.

11.2.19. (/=)

```
site (/=)(Top, Top) :: Boolean
```

`a/=b` returns false if `a=b`, and true otherwise.

11.2.20. (~)

```
site (~)(Boolean) :: Boolean
```

Return the logical negation of the argument.

11.2.21. (&&)

```
site (&&)(Boolean, Boolean) :: Boolean
```

Return the logical conjunction of the arguments. This is not a short-circuiting operator; both arguments must publish before the result is computed.

11.2.22. (||)

```
site (||)(Boolean, Boolean) :: Boolean
```

Return the logical disjunction of the arguments. This is not a short-circuiting operator; both arguments must publish before the result is computed.

11.2.23. (:)

```
site (:)[A](A, List[A]) :: List[A]
```

The list `a:b` is formed by prepending the element `a` to the list `b`.

Example:

```
-- Publishes: (3, [4, 5])
3:4:5:[] >x:xs> (x,xs)
```

11.2.24. abs

```
def abs(Number) :: Number
```

Publishes the absolute value of the argument.

Implementation.

```
def abs(Number) :: Number
def abs(x) = if x <: 0 then -x else x
```

11.2.25. signum

```
def signum(Number) :: Number
```

`signum(a)` publishes `-1` if `a<0`, `1` if `a>0`, and `0` if `a=0`.

Implementation.

```
def signum(Number) :: Number
def signum(x) =
  if x <: 0 then -1
  else if x :> 0 then 1
  else 0
```

11.2.26. min

```
def min[A](A,A) :: A
```

Publishes the lesser of the arguments. If the arguments are equal, publishes the first argument.

Implementation.

```
def min[A](A,A) :: A
def min(x,y) = if y <: x then y else x
```

11.2.27. `max`

```
def max[A](A,A) :: A
```

Publishes the greater of the arguments. If the arguments are equal, publishes the second argument.

Implementation.

```
def max[A](A,A) :: A
def max(x,y) = if x :> y then x else y
```

11.2.28. `Floor`

```
site Floor(Number) :: Integer
```

Return the greatest integer less than or equal to this number.

11.2.29. `Ceil`

```
site Ceil(Number) :: Integer
```

Return the least integer greater than or equal to this number.

11.2.30. `sqrt`

```
def sqrt(Number) :: Integer
```

Publish the square root of this number. If the number is negative, halt silently.

Implementation.

```
def sqrt(Number) :: Number
def sqrt(n) = n ** 0.5
```

11.3. idioms: Higher-order Orc programming idioms.

Higher-order Orc programming idioms. Many of these are standard functional-programming combinators borrowed from Haskell or Scheme.

11.3.1. curry

```
def curry[A,B,C](lambda (A,B) :: C) :: lambda(A) :: lambda(B) :: C
```

Curry a function of two arguments.

Implementation.

```
def curry[A,B,C](lambda (A,B) :: C) :: lambda(A) :: lambda(B) :: C
def curry(f) = lambda(x) = lambda(y) = f(x,y)
```

11.3.2. curry3

```
def curry3[A,B,C,D](lambda (A,B,C) :: D) :: lambda(A) :: lambda(B) ::
lambda(C) :: D
```

Curry a function of three arguments.

Implementation.

```
def curry3[A,B,C,D](lambda (A,B,C) :: D) :: lambda(A) :: lambda(B) :: lambda(C) ::
def curry3(f) = lambda(x) = lambda(y) = lambda(z) = f(x,y,z)
```

11.3.3. uncurry

```
def uncurry[A,B,C](lambda (A) :: lambda(B) :: C) :: lambda(A, B) :: C
```

Uncurry a function of two arguments.

Implementation.

```
def uncurry[A,B,C](lambda (A) :: lambda(B) :: C) :: lambda(A, B) :: C
```

```
def uncurry(f) = lambda(x,y) = f(x)(y)
```

11.3.4. uncurry3

```
def uncurry3[A,B,C,D](lambda (A)(B)(C) :: D) :: lambda(A,B,C) :: D
```

Uncurry a function of three arguments.

Implementation.

```
def uncurry3[A,B,C,D](lambda (A) :: lambda(B) :: lambda(C) :: D) :: lambda(A,B,C)
def uncurry3(f) = lambda(x,y,z) = f(x)(y)(z)
```

11.3.5. flip

```
def flip[A,B,C](lambda (A, B) :: C) :: lambda(B, A) :: C
```

Flip the order of parameters of a two-argument function.

Implementation.

```
def flip[A,B,C](lambda (A, B) :: C) :: lambda(B, A) :: C
def flip(f) = lambda(x,y) = f(y,x)
```

11.3.6. constant

```
def constant[A](A) :: lambda() :: A
```

Create a function which returns a constant value.

Implementation.

```
def constant[A](A) :: lambda() :: A
def constant(x) = lambda() = x
```

11.3.7. defer

```
def defer[A,B](lambda (A) :: B, A) :: lambda() :: B
```

Given a function and its argument, return a thunk which applies the function.

Implementation.

```
def defer[A,B](lambda (A) :: B, A) :: lambda() :: B
def defer(f, x) = lambda() = f(x)
```

11.3.8. defer2

```
def defer2[A,B,C](lambda (A,B) :: C, A, B) :: lambda() :: C
```

Given a function and its arguments, return a thunk which applies the function.

Implementation.

```
def defer2[A,B,C](lambda (A,B) :: C, A, B) :: lambda() :: C
def defer2(f, x, y) = lambda() = f(x, y)
```

11.3.9. ignore

```
def ignore[A](lambda () :: A) :: lambda(Top) :: B
```

From a function of no arguments, create a function of one argument, which is ignored.

Implementation.

```
def ignore[A](lambda () :: A) :: lambda(Top) :: A
def ignore(f) = lambda(_) = f()
```

11.3.10. ignore2

```
def ignore2[A,B,C](lambda () :: C) :: lambda(A, B) :: C
```

From a function of no arguments, create a function of two arguments, which are ignored.

Implementation.

```
def ignore2[A,B,C](lambda () :: C) :: lambda(A, B) :: C
def ignore2(f) = lambda(_, _) = f()
```

11.3.11. compose

```
def compose[A,B,C](lambda (B) :: C, lambda (A) :: B) :: lambda(A) :: C
```

Compose two single-argument functions.

Implementation.

```
def compose[A,B,C](lambda (B) :: C, lambda (A) :: B) :: lambda (A) :: C
def compose(f,g) = lambda(x) = f(g(x))
```

11.3.12. while

```
def while[A](lambda (A) :: Boolean, lambda (A) :: A) :: lambda(A) :: A
```

Iterate a function while a predicate is satisfied, publishing each value passed to the function.

Example:

```
-- Publishes: 0 1 2 3 4 5
while(
  lambda (n) = (n <= 5),
  lambda (n) = n+1
)(0)
```

Implementation.

```
def while[A](lambda (A) :: Boolean, lambda (A) :: A) :: lambda(A) :: A
def while(p,f) =
  def loop(A) :: A
  def loop(x) = Ift(p(x)) >> ( x | loop(f(x)) )
```

```
loop
```

11.3.13. repeat

```
def repeat[A](lambda () :: A) :: A
```

Call a function sequentially, publishing each value returned by the function. The expression `repeat(f)` is equivalent to the infinite expression `f() >x> ( x | f() >x> ( x | f() >x> ... ) )`

Implementation.

```
def repeat[A](lambda () :: A) :: A
def repeat(f) = f() >x> (x | repeat(f))
```

11.3.14. fork

```
def fork[A](List[lambda () :: A]) :: A
```

Call a list of functions in parallel, publishing all values published by the functions.

The expression `fork([f,g,h])` is equivalent to the expression `f() | g() | h()`

Implementation.

```
def fork[A](List[lambda () :: A]) :: A
def fork([]) = stop
def fork(p:ps) = p() | fork(ps)
```

11.3.15. forkMap

```
def forkMap[A,B](lambda (A) :: B, List[A]) :: B
```

Apply a function to a list in parallel, publishing all values published by the applications.

The expression `forkMap(f, [a,b,c])` is equivalent to the expression `f(a) | f(b) | f(c)`

Implementation.

```
def forkMap[A,B](lambda (A) :: B, List[A]) :: B
def forkMap(f, []) = stop
def forkMap(f, x:xs) = f(x) | forkMap(f, xs)
```

11.3.16. seq

```
def seq[A](List[lambda () :: A]) :: Signal
```

Call a list of functions in sequence, publishing a signal whenever the last function publishes. The actual publications of the given functions are not published.

The expression `seq([f,g,h])` is equivalent to the expression `f() >> g() >> h() >> signal`

Implementation.

```
def seq[A](List[lambda () :: A]) :: Signal
def seq([]) = signal
def seq(p:ps) = p() >> seq(ps)
```

11.3.17. seqMap

```
def seqMap[A,B](lambda (A) :: B, List[A]) :: Signal
```

Apply a function to a list in sequence, publishing a signal whenever the last application publishes. The actual publications of the given functions are not published.

The expression `seqMap(f, [a,b,c])` is equivalent to the expression `f(a) >> f(b) >> f(c) >> signal`

Implementation.

```
def seqMap[A,B](lambda (A) :: B, List[A]) :: Signal
def seqMap(f, []) = signal
def seqMap(f, x:xs) = f(x) >> seqMap(f, xs)
```

11.3.18. join

```
def join[A](List[lambda () :: A]) :: Signal
```

Call a list of functions in parallel and publish a signal once all functions have completed.

The expression `join([f,g,h])` is equivalent to the expression `(f(), g(), h()) >> signal`

Implementation.

```
def join[A](List[lambda () :: A]) :: Signal
def join([]) = signal
def join(p:ps) = (p(), join(ps)) >> signal
```

11.3.19. `joinMap`

```
def joinMap[A](lambda (A) :: Top, List[A]) :: Signal
```

Apply a function to a list in parallel and publish a signal once all applications have completed.

The expression `joinMap(f, [a,b,c])` is equivalent to the expression `(f(a), f(b), f(c)) >> signal`

Implementation.

```
def joinMap[A](lambda (A) :: Top, List[A]) :: Signal
def joinMap(f, []) = signal
def joinMap(f, x:xs) = (f(x), joinMap(f, xs)) >> signal
```

11.3.20. `alt`

```
def alt[A](List[lambda () :: A]) :: A
```

Call each function in the list until one of them publishes.

The expression `alt([f,g,h])` is equivalent to the expression `f() ; g() ; h()`

Implementation.

```
def alt[A](List[lambda () :: A]) :: A
def alt([]) = stop
def alt(p:ps) = p() ; alt(ps)
```

11.3.21. `altMap`

```
def altMap[A,B](lambda (A) :: B, List[A]) :: B
```

Apply the function to each element in the list until one publishes.

The expression `altMap(f, [a,b,c])` is equivalent to the expression `f(a) ; f(b) ; f(c)`

Implementation.

```
def altMap[A,B](lambda (A) :: B, List[A]) :: B
def altMap(f, []) = stop
def altMap(f, x:xs) = f(x) ; altMap(f, xs)
```

11.3.22. `por`

```
def por(List[lambda () :: Boolean]) :: Boolean
```

Parallel or. Execute a list of boolean functions in parallel, publishing a value as soon as possible, and killing any unnecessary ongoing computation.

Implementation.

```
def por(List[lambda () :: Boolean]) :: Boolean
def por([]) = false
def por(p:ps) =
  Let(
    val b1 = p()
    val b2 = por(ps)
    Ift(b1) >> true | Ift(b2) >> true | (b1 || b2)
  )
```

11.3.23. `pand`

```
def pand(List[lambda () :: Boolean]) :: Boolean
```

Parallel and. Execute a list of boolean functions in parallel, publishing a value as soon as possible, and killing any unnecessary ongoing computation.

Implementation.

```
def pand(List[lambda () :: Boolean]) :: Boolean
def pand([]) = true
def pand(p:ps) =
  Let(
    val b1 = p()
    val b2 = pand(ps)
    Iff(b1) >> false | Iff(b2) >> false | (b1 && b2)
  )
```

11.3.24. `collect`

```
def collect[A](lambda () :: A) :: List[A]
```

Run a function, collecting all publications in a list. Publish the list when the function halts.

Example:

```
-- Publishes: [signal, signal, signal, signal, signal]
collect(defer(signals, 5))
```

Implementation.

```
def collect[A](lambda () :: A) :: List[A]
def collect(p) =
  val b = Channel[A]()
  p() >x> b.put(x) >> stop
  ; b.getAll()
```

11.4. list: Operations on lists.

Operations on lists.

Many of these functions are similar to those in the Haskell prelude, but operate on the elements of a list in parallel.

11.4.1. each

```
def each[A](List[A]) :: A
```

Publish every value in a list, simultaneously.

Implementation.

```
def each[A](List[A]) :: A
def each([]) = stop
def each(h:t) = h | each(t)
```

11.4.2. map

```
def map[A,B](lambda (A) :: B, List[A]) :: List[B]
```

Apply a function to every element of a list (in parallel), publishing a list of the results.

Implementation.

```
def map[A,B](lambda (A) :: B, List[A]) :: List[B]
def map(f,[]) = []
def map(f,h:t) = f(h):map(f,t)
```

11.4.3. reverse

```
def reverse[A](List[A]) :: List[A]
```

Publish the reverse of the given list.

Implementation.

```
def reverse[A](List[A]) :: List[A]
def reverse(l) =
  def tailrev(List[A], List[A]) :: List[A]
  def tailrev([],x) = x
  def tailrev(h:t,x) = tailrev(t,h:x)
  tailrev(l,[])
```

11.4.4. `filter`

```
def filter[A](lambda (A) :: Boolean, List[A]) :: List[A]
```

Publish a list containing only those elements which satisfy the predicate. The filter is applied to all list elements in parallel.

Implementation.

```
def filter[A](lambda (A) :: Boolean, List[A]) :: List[A]
def filter(p,[]) = []
def filter(p,x:xs) =
  val fxs = filter(p, xs)
  if p(x) then x:fxs else fxs
```

11.4.5. `head`

```
def head[A](List[A]) :: A
```

Publish the first element of a list.

Implementation.

```
def head[A](List[A]) :: A
def head(x:xs) = x
```

11.4.6. `tail`

```
def tail[A](List[A]) :: List[A]
```

Publish all but the first element of a list.

Implementation.

```
def tail[A](List[A]) :: List[A]
def tail(x:xs) = xs
```

11.4.7. `init`

```
def init[A](List[A]) :: List[A]
```

Publish all but the last element of a list.

Implementation.

```
def init[A](List[A]) :: List[A]
def init([x]) = []
def init(x:xs) = x:init(xs)
```

11.4.8. `last`

```
def last[A](List[A]) :: A
```

Publish the last element of a list.

Implementation.

```
def last[A](List[A]) :: A
def last([x]) = x
def last(x:xs) = last(xs)
```

11.4.9. `empty`

```
def empty[A](List[A]) :: Boolean
```

Is the list empty?

Implementation.

```
def empty[A](List[A]) :: Boolean
def empty([]) = true
def empty(_) = false
```

11.4.10. `index`

```
def index[A](List[A], Integer) :: A
```

Publish the nth element of a list, counting from 0.

Implementation.

```
def index[A](List[A], Integer) :: A
def index(h:t, 0) = h
def index(h:t, n) = index(t, n-1)
```

11.4.11. `append`

```
def append[A](List[A], List[A]) :: List[A]
```

Publish the first list concatenated with the second.

Implementation.

```
def append[A](List[A], List[A]) :: List[A]
def append([],l) = l
def append(h:t,l) = h:append(t,l)
```

11.4.12. `foldl`

```
def foldl[A,B](lambda (B, A) :: B, B, List[A]) :: B
```

Reduce a list using the given left-associative binary operation and initial value. Given the list `[x1, x2, x3, ...]` and initial value `x0`, returns `f(... f(f(f(x0, x1), x2), x3) ...)`

Example using `foldl` to reverse a list:

```
-- Publishes: [3, 2, 1]
foldl(flip((:)), [], [1,2,3])
```

Implementation.

```
def foldl[A,B](lambda (B, A) :: B, B, List[A]) :: B
def foldl(f,z,[]) = z
def foldl(f,z,x:xs) = foldl(f,f(z,x),xs)
```

11.4.13. `foldl1`

```
def foldl1[A](lambda (A, A) :: A, List[A]) :: A
```

A special case of `foldl` which uses the first element of the list as the initial value. If called on an empty list, halt.

Implementation.

```
def foldl1[A](lambda (A, A) :: A, List[A]) :: A
def foldl1(f,x:xs) = foldl(f,x,xs)
```

11.4.14. `foldr`

```
def foldr[A,B](lambda (A, B) :: B, B, List[A]) :: B
```

Reduce a list using the given right-associative binary operation and initial value. Given the list `[..., x3, x2, x1]` and initial value `x0`, returns `f(... f(x3, f(x2, f(x1, x0))) ...)`

Example summing the numbers in a list:

```
-- Publishes: 6
foldr((+), 0, [1,2,3])
```

Implementation.

```
def foldr[A,B](lambda (A, B) :: B, B, List[A]) :: B
def foldr(f,z,xs) = foldl(flip(f),z,reverse(xs))
```

11.4.15. `foldr1`

```
def foldr1[A](lambda (A, A) :: A, List[A]) :: A
```

A special case of `foldr` which uses the last element of the list as the initial value. If called on an empty list, halt.

Implementation.

```
def foldr1[A](lambda (A, A) :: A, List[A]) :: A
def foldr1(f,xs) = foldl1(flip(f),reverse(xs))
```

11.4.16. `afold`

```
def afold[A](lambda (A, A) :: A, List[A]) :: A
```

Reduce a non-empty list using the given associative binary operation. This function reduces independent subexpressions in parallel; the calls exhibit a balanced tree structure, so the number of sequential reductions performed is O(log n). For expensive reductions, this is much more efficient than `foldl` or `foldr`.

Implementation.

```
def afold[A](lambda (A, A) :: A, List[A]) :: A
def afold(f, [x]) = x
{- Here's the interesting part -}
def afold(f, xs) =
  def afold'(List[A]) :: List[A]
  def afold'([]) = []
  def afold'([x]) = [x]
  def afold'(x:y:xs) = f(x,y):afold'(xs)
  afold(f, afold'(xs))
```

11.4.17. `cfold`

```
def cfold[A](lambda (A, A) :: A, List[A]) :: A
```

Reduce a non-empty list using the given associative and commutative binary operation. This function opportunistically reduces independent subexpressions in parallel, so the number of sequential reductions

performed is as small as possible. For expensive reductions, this is much more efficient than `foldl` or `foldr`. In cases where the reduction does not always take the same amount of time to complete, it is also more efficient than `afold`.

Implementation.

```
def cfold[A](lambda (A, A) :: A, List[A]) :: A
def cfold(f, []) = stop
def cfold(f, [x]) = x
def cfold(f, [x,y]) = f(x,y)
def cfold(f, L) =
  val c = Channel[A]()
  def work(Number, List[A]) :: A
  def work(i, x:y:rest) =
    c.put(f(x,y)) >> stop | work(i+1, rest)
  def work(i, [x]) = c.put(x) >> stop | work(i+1, [])
  def work(i, []) =
    if (i <: 2) then c.get()
    else c.get() >x> c.get() >y>
         ( c.put(f(x,y)) >> stop | work(i-1,[]) )
  work(0, L)
```

11.4.18. `zip`

```
def zip[A,B](List[A], List[B]) :: List[(A,B)]
```

Combine a pair of lists into a list of pairs. The length of the shortest list determines the length of the result.

Implementation.

```
def zip[A,B](List[A], List[B]) :: List[(A,B)]
def zip([],_) = []
def zip(_,[]) = []
def zip(x:xs,y:ys) = (x,y):zip(xs,ys)
```

11.4.19. `unzip`

```
def unzip[A,B](List[(A,B)]) :: (List[A], List[B])
```

Split a list of pairs into a pair of lists.

Implementation.

```
def unzip[A,B](List[(A,B)]) :: (List[A], List[B])
def unzip([]) = ([],[])
def unzip((x,y):z) = (x:xs,y:ys) <(xs,ys)< unzip(z)
```

11.4.20. concat

```
def concat[A](List[List[A]]) :: List[A]
```

Concatenate a list of lists into a single list.

Implementation.

```
def concat[A](List[List[A]]) :: List[A]
def concat([]) = []
def concat(h:t) = append(h,concat(t))
```

11.4.21. length

```
def length[A](List[A]) :: Integer
```

Publish the number of elements in a list.

Implementation.

```
def length[A](List[A]) :: Integer
def length([]) = 0
def length(h:t) = 1 + length(t)
```

11.4.22. take

```
def take[A](Integer, List[A]) :: List[A]
```

Given a number `n` and a list `l`, publish a list of the first `n` elements of `l`. If `n` exceeds the length of `l`, or `n < 0`, take halts with an error.

Implementation.

```
def take[A](Integer, List[A]) :: List[A]
def take(0, _) = []
def take(n, x:xs) =
  if n :> 0 then x:take(n-1, xs)
  else Error("Cannot take(" + n + ", _)")
```

11.4.23. `drop`

```
def drop[A](Integer, List[A]) :: List[A]
```

Given a number `n` and a list `l`, publish a list of the elements of `l` after the first `n`. If `n` exceeds the length of `l`, or `n < 0`, drop halts with an error.

Implementation.

```
def drop[A](Integer, List[A]) :: List[A]
def drop(0, xs) = xs
def drop(n, x:xs) =
  if n :> 0 then drop(n-1, xs)
  else Error("Cannot drop(" + n + ", _)")
```

11.4.24. `member`

```
def member[A](A, List[A]) :: Boolean
```

Publish true if the given item is a member of the given list, and false otherwise.

Implementation.

```
def member[A](A, List[A]) :: Boolean
def member(item, []) = false
def member(item, h:t) =
  if item = h then true
  else member(item, t)
```

11.4.25. merge

```
def merge[A](List[A], List[A]) :: List[A]
```

Merge two sorted lists.

Example:

```
-- Publishes: [1, 2, 2, 3, 4, 5]
merge([1,2,3], [2,4,5])
```

Implementation.

```
def merge[A](List[A], List[A]) :: List[A]
def merge(xs,ys) = mergeBy((<:), xs, ys)
```

11.4.26. mergeBy

```
def mergeBy[A](lambda (A,A) :: Boolean, List[A], List[A]) :: List[A]
```

Merge two lists using the given less-than relation.

Implementation.

```
def mergeBy[A](lambda (A,A) :: Boolean,
               List[A], List[A]) :: List[A]
def mergeBy(lt, xs, []) = xs
def mergeBy(lt, [], ys) = ys
def mergeBy(lt, x:xs, y:ys) =
  if lt(y,x) then y:mergeBy(lt,x:xs,ys)
  else x:mergeBy(lt,xs,y:ys)
```

11.4.27. sort

```
def sort[A](List[A]) :: List[A]
```

Sort a list.

Example:

```
-- Publishes: [1, 2, 3]
sort([1,3,2])
```

Implementation.

```
def sort[A](List[A]) :: List[A]
def sort(xs) = sortBy((<:), xs)
```

11.4.28. `sortBy`

```
def sortBy[A](lambda (A,A) :: Boolean, List[A]) :: List[A]
```

Sort a list using the given less-than relation.

Implementation.

```
def sortBy[A](lambda (A,A) :: Boolean, List[A]) :: List[A]
def sortBy(lt, []) = []
def sortBy(lt, [x]) = [x]
def sortBy(lt, xs) =
  val half = Floor(length(xs)/2)
  val front = take(half, xs)
  val back = drop(half, xs)
  mergeBy(lt, sortBy(lt, front), sortBy(lt, back))
```

11.4.29. `mergeUnique`

```
def mergeUnique[A](List[A], List[A]) :: List[A]
```

Merge two sorted lists, discarding duplicates.

Example:

```
-- Publishes: [1, 2, 3, 4, 5]
mergeUnique([1,2,3], [2,4,5])
```

Implementation.

```
def mergeUnique[A](List[A], List[A]) :: List[A]
def mergeUnique(xs,ys) = mergeUniqueBy((=), (<:), xs, ys)
```

11.4.30. `mergeUniqueBy`

```
def mergeUniqueBy[A](lambda (A,A) :: Boolean, lambda (A,A) :: Boolean,
List[A], List[A]) :: List[A]
```

Merge two lists, discarding duplicates, using the given equality and less-than relations.

Implementation.

```
def mergeUniqueBy[A](lambda (A,A) :: Boolean,
                     lambda (A,A) :: Boolean,
                      List[A], List[A])
  :: List[A]
def mergeUniqueBy(eq, lt, xs, []) = xs
def mergeUniqueBy(eq, lt, [], ys) = ys
def mergeUniqueBy(eq, lt, x:xs, y:ys) =
  if eq(y,x) then mergeUniqueBy(eq, lt, xs, y:ys)
  else if lt(y,x) then y:mergeUniqueBy(eq,lt,x:xs,ys)
  else x:mergeUniqueBy(eq,lt,xs,y:ys)
```

11.4.31. `sortUnique`

```
def sortUnique[A](List[A]) :: List[A]
```

Sort a list, discarding duplicates.

Example:

```
-- Publishes: [1, 2, 3]
sortUnique([1,3,2,3])
```

Implementation.

```
def sortUnique[A](List[A]) :: List[A]
def sortUnique(xs) = sortUniqueBy((=), (<:), xs)
```

11.4.32. sortUniqueBy

```
def sortUniqueBy[A](lambda (A,A) :: Boolean, lambda (A,A) :: Boolean,
List[A]) :: List[A]
```

Sort a list, discarding duplicates, using the given equality and less-than relations.

Implementation.

```
def sortUniqueBy[A](lambda (A,A) :: Boolean,
                    lambda (A,A) :: Boolean,
                    List[A])
  :: List[A]
def sortUniqueBy(eq, lt, []) = []
def sortUniqueBy(eq, lt, [x]) = [x]
def sortUniqueBy(eq, lt, xs) =
  val half = Floor(length(xs)/2)
  val front = take(half, xs)
  val back = drop(half, xs)
  mergeUniqueBy(eq, lt,
    sortUniqueBy(eq, lt, front),
    sortUniqueBy(eq, lt, back))
```

11.4.33. group

```
def group[A,B](List[(A,B)]) :: List[(A,List[B])]
```

Given a list of pairs, group together the second elements of consecutive pairs with equal first elements.

Example:

```
-- Publishes: [(1, [1, 2]), (2, [3]), (3, [4]), (1, [3])]
group([(1,1), (1,2), (2,3), (3,4), (1,3)])
```

Implementation.

```
def group[A,B](List[(A,B)]) :: List[(A,List[B])]
def group(xs) = groupBy((=), xs)
```

11.4.34. groupBy

```
def groupBy[A,B](lambda (A,A) :: Boolean, List[(A,B)]) ::
List[(A,List[B])]
```

Given a list of pairs, group together the second elements of consecutive pairs with equal first elements, using the given equality relation.

Implementation.

```
def groupBy[A,B](lambda (A,A) :: Boolean,
                 List[(A,B)])
  :: List[(A,List[B])]
def groupBy(eq, []) = []
def groupBy(eq, (k,v):kvs) =
  def helper(A, List[B], List[(A,B)]) :: List[(A,List[B])]
  def helper(k,vs, []) = [(k,vs)]
  def helper(k,vs, (k2,v):kvs) =
    if eq(k2,k) then helper(k, v:vs, kvs)
    else (k,vs):helper(k2, [v], kvs)
  helper(k,[v], kvs)
```

11.4.35. `rangeBy`

```
def rangeBy(Number, Number, Number) :: List[Number]
```

`rangeBy(low, high, skip)` returns a sorted list of numbers `n` which satisfy `n = low + skip*i` (for some integer `i`), `n >= low`, and `n < high`.

Implementation.

```
def rangeBy(Number, Number, Number) :: List[Number]
def rangeBy(low, high, skip) =
  if low <: high
  then low:rangeBy(low+skip, high, skip)
  else []
```

11.4.36. `range`

```
def range(Number, Number) :: List[Number]
```

Generate a list of numbers in the given half-open range.

Implementation.

```
def range(Number, Number) :: List[Number]
def range(low, high) = rangeBy(low, high, 1)
```

11.4.37. any

```
def any[A](lambda (A) :: Boolean, List[A]) :: Boolean
```

Publish true if any of the elements of the list match the predicate, and false otherwise. The predicate is applied to all elements of the list in parallel; the result is returned as soon as it is known and any unnecessary execution of the predicate killed.

Implementation.

```
def any[A](lambda (A) :: Boolean, List[A]) :: Boolean
def any(p, []) = false
def any(p, x:xs) =
  Let(
    val b1 = p(x)
    val b2 = any(p, xs)
    Ift(b1) >> true | Ift(b2) >> true | (b1 || b2)
  )
```

11.4.38. all

```
def all[A](lambda (A) :: Boolean, List[A]) :: Boolean
```

Publish true if all of the elements of the list match the predicate, and false otherwise. The predicate is applied to all elements of the list in parallel; the result is returned as soon as it is known and any unnecessary execution of the predicate killed.

Implementation.

```
def all[A](lambda (A) :: Boolean, List[A]) :: Boolean
def all(p, []) = true
def all(p, x:xs) =
  Let(
    val b1 = p(x)
    val b2 = all(p, xs)
    Iff(b1) >> false | Iff(b2) >> false | (b1 && b2)
  )
```

11.4.39. sum

```
def sum(List[Number]) :: Number
```

Publish the sum of all numbers in a list. The sum of an empty list is 0.

Implementation.

```
def sum(List[Number]) :: Number
def sum(xs) = foldl(
  (+) :: lambda (Number, Number) :: Number,
  0, xs)
```

11.4.40. product

```
def product(List[Number]) :: Number
```

Publish the product of all numbers in a list. The product of an empty list is 1.

Implementation.

```
def product(List[Number]) :: Number
def product(xs) = foldl(
  (*) :: lambda (Number, Number) :: Number,
  1, xs)
```

11.4.41. and

```
def and(List[Boolean]) :: Boolean
```

Publish the boolean conjunction of all boolean values in the list. The conjunction of an empty list is `true`.

Implementation.

```
def and(List[Boolean]) :: Boolean
```

```
def and([]) = true
def and(false:xs) = false
def and(true:xs) = and(xs)
```

11.4.42. or

```
def or(List[Boolean]) :: Boolean
```

Publish the boolean disjunction of all boolean values in the list. The disjunction of an empty list is `false`.

Implementation.

```
def or(List[Boolean]) :: Boolean
def or([]) = false
def or(true:xs) = true
def or(false:xs) = or(xs)
```

11.4.43. minimum

```
def minimum[A](List[A]) :: A
```

Publish the minimum element of a non-empty list.

Implementation.

```
def minimum[A](List[A]) :: A
def minimum(xs) =
  def minA(x :: A, y :: A) = min(x,y)
  foldl1(minA, xs)
```

11.4.44. maximum

```
def maximum[A](List[A]) :: A
```

Publish the maximum element of a non-empty list.

Implementation.

```
def maximum[A](List[A]) :: A
def maximum(xs) =
  def maxA(x :: A, y :: A) = max(x,y)
  foldl1(maxA, xs)
```

11.5. reflect: Metalanguage operations.

Metalanguage operations.

11.5.1. `MakeSite`

```
site MakeSite[A](A) :: A
```

This site promotes an Orc closure to a site; when the site is called, the closure is executed on those arguments. This execution behaves like a site call; in particular, the following four properties hold:

- The site, like all sites, is strict in its arguments.
- The site returns only the first value published by the executed closure. The closure continues to run, but its subsequent publications are discarded.
- The execution of the closure is protected from being killed. If the site call is killed, the closure still runs, and its publications are simply ignored.
- If the execution of the closure halts, so does the site call.

The typical usage of `MakeSite` looks like:

```
def foo(...) = ...
val Foo = MakeSite(foo)
```

The typing of `MakeSite` enforces the additional condition that the type A is a function type.

11.6. state: General-purpose supplemental data structures.

General-purpose supplemental data structures.

11.6.1. Some

```
site Some[A](A) :: Option[A]
```

An optional value which is available. This site may also be used in a pattern.

Example:

```
-- Publishes: (3, 4)
Some((3,4)) >s> (
    s >Some((x,y))> (x,y)
  | s >None()> signal
)
```

11.6.2. None

```
site None[A]() :: Option[A]
```

An optional value which is not available. This site may also be used in a pattern.

11.6.3. Cell

```
site Cell[A]() :: Cell[A]
```

Create a write-once storage location.

Example:

```
-- Publishes: 5 5
val c = Cell()
  c.write(5) >> c.read()
| Rwait(1) >> ( c.write(10) ; c.read() )
```

```
cell[A].read() :: A
```

Indefinite **Idempotent**

Read a value from the cell. If the cell does not yet have a value, block until it receives one.

```
cell[A].readD() :: A
```

Definite **Idempotent**

Read a value from the cell. If the cell does not yet have a value, halt silently.

```
cell[A].write(A) :: Signal
```

Definite **Idempotent**

Write a value to the cell. If the cell already has a value, halt silently.

11.6.4. `Ref`

```
site Ref[A]() :: Ref[A]
```

Create a rewritable storage location without an initial value.

Example:

```
val r = Ref()
Rwait(1000) >> r := 5 >> stop
| Println(r?) >>
  r := 10 >>
  Println(r?) >>
  stop
```

11.6.5. `Ref`

```
site Ref[A](A) :: Ref[A]
```

Create a rewritable storage location initialized to the provided value.

```
ref[A].read() :: A
```

Read the value of the ref. If the ref does not yet have a value, block until it receives one.

```
ref[A].readD() :: A
```

Read the value of the ref. If the ref does not yet have a value, halt silently.

```
ref[A].write(A) :: Signal
```

Write a value to the ref, then return a `signal`.

11.6.6. `(?)`

```
def (?)[A](Ref[A]) :: A
```

Get the value held by a reference. `x?` is equivalent to `x.read()`.

Implementation.

```
def (?)[A](Ref[A]) :: A
def (?)(r) = r.read()
```

11.6.7. (:=)

```
def (:=)[A](Ref[A], A) :: Signal
```

Set the value held by a reference. `x := y` is equivalent to `x.write(y)`.

Implementation.

```
def (:=)[A](Ref[A], A) :: Signal
def (:=)(r,v) = r.write(v)
```

11.6.8. swap

```
def swap[A](Ref[A], Ref[A]) :: Signal
```

Swap the values in two references, then return a signal.

Implementation.

```
def swap[A](Ref[A], Ref[A]) :: Signal
def swap(r,s) = (r?,s?) >(rval,sval)> (r := sval, s := rval) >> signal
```

11.6.9. Semaphore

```
site Semaphore(Integer) :: Semaphore
```

Return a semaphore with the given value. The semaphore maintains the invariant that its value is always non-negative.

An example using a semaphore as a lock for a critical section:

```
-- Prints:
-- Entering critical section
```

```
-- Leaving critical section
val lock = Semaphore(1)
lock.acquire() >>
Println("Entering critical section") >>
Println("Leaving critical section") >>
lock.release()
```

```
semaphore.acquire() :: Signal
```

Indefinite

If the semaphore's value is greater than `0`, decrement the semaphore and return a signal. If the semaphore's value is `0`, block until it becomes greater than `0`.

```
semaphore.acquireD() :: Signal
```

Definite

If the semaphore's value is greater than `0`, decrement the semaphore and return a signal. If the semaphore's value is `0`, halt silently.

```
semaphore.release() :: Signal
```

Definite

If any calls to `acquire` are blocked, allow the oldest such call to return. Otherwise, increment the value of the semaphore. This may increment the value beyond that with which the semaphore was constructed.

```
semaphore.snoop() :: Signal
```

Indefinite

If any calls to `acquire` are blocked, return a signal. Otherwise, block until some call to `acquire` blocks.

```
semaphore.snoopD() :: Signal
```

Definite

If any calls to `acquire` are blocked, return a signal. Otherwise, halt silently.

11.6.10. `Channel`

```
site Channel[A]() :: Channel[A]
```

Create a new asyncrhonous FIFO channel of unlimited size. A channel supports get, put and close operations.

A channel may be either empty or non-empty, and either open or closed. When empty and open, calls to `get` block. When empty and closed, calls to `get` halt silently. When closed, calls to `put` halt silently. In all other cases, calls return normally.

Example:

```
-- Publishes: 10
val b = Channel()
```

```
  Rwait(1000) >> b.put(10) >> stop
| b.get()
```

```
channel[A].get() :: A
```

Indefinite

Get an item from the channel. If the channel is open and no items are available, block until one becomes available. If the channel is closed and no items are available, halt silently.

```
channel[A].getD() :: A
```

Definite

Get an item from the channel. If no items are available, halt silently.

```
channel[A].put(A) :: Signal
```

Definite

Put an item in the channel. If the channel is closed, halt silently.

```
channel[A].close() :: Signal
```

Indefinite **Idempotent**

Close the channel and block until it is empty. This has the effect of immediately causing any blocked calls to `get` to halt silently. In addition, any subsequent calls to `put` will halt silently, and once the channel becomes empty, any subsequent calls to `get` will halt silently.

When the channel is empty, return a signal.

```
channel[A].closeD() :: Signal
```

Definite **Idempotent**

Close the channel and return a signal immediately. This has the effect of immediately causing any blocked calls to `get` to halt silently. In addition, any subsequent calls to `put` will halt silently, and once the channel becomes empty, any subsequent calls to `get` will halt silently.

```
channel[A].isClosed() :: Boolean
```

Definite

If the channel is currently closed, return true, otherwise return false.

```
channel[A].getAll() :: List[A]
```

Definite

Get all of the items currently in the channel, emptying the channel and returning a list of the items in the order they were added. If there are no items in the channel, return an empty list.

11.6.11. BoundedChannel

```
site BoundedChannel[A](Integer) :: BoundedChannel[A]
```

Create a new asynchronous FIFO channel with the given number of slots. Putting an item into the channel fills a slot, and getting an item opens a slot. A channel with zero slots is equivalent to a synchronous channel.

A bounded channel may be empty, partly filled, or full, and either open or closed. When empty and open, calls to `get` block. When empty and closed, calls to `get` halt silently. When full and open, calls to `put` block. When closed, calls to `put` halt silently. In all other cases, calls return normally.

Example:

```
-- Publishes: "Put 1" "Got 1" "Put 2" "Got 2"
val c = BoundedChannel(1)
  c.put(1) >> "Put " + 1
| c.put(2) >> "Put " + 2
| Rwait(1000) >> (
    c.get() >n> "Got " + n
  | c.get() >n> "Got " + n
  )
```

```
boundedChannel[A].get() :: A
```

Indefinite

Get an item from the channel. If the channel is open and no items are available, block until one becomes available. If the channel is closed and no items are available, halt silently.

```
boundedChannel[A].getD() :: A
```

Definite

Get an item from the channel. If no items are available, halt silently.

```
boundedChannel[A].put(A) :: Signal
```

Indefinite

Put an item in the channel. If no slots are open, block until one becomes open. If the channel is closed, halt silently.

```
boundedChannel[A].putD(A) :: Signal
```

Definite

Put an item in the channel. If no slots are open, halt silently. If the channel is closed, halt silently.

```
boundedChannel[A].close() :: Signal
```

Indefinite **Idempotent**

Close the channel and block until it is empty. This has the effect of immediately causing any blocked calls to `get` to halt silently. In addition, any subsequent calls to `put` will halt silently, and once the channel becomes empty, any subsequent calls to `get` will halt silently. Note that any blocked calls to `put` initiated prior to closing the channel may still be allowed to return as usual.

```
boundedChannel[A].closeD() :: Signal
```

Definite **Idempotent**

Close the channel and return a signal immediately. This has the effect of immediately causing any blocked calls to `get` to halt silently. In addition, any subsequent calls to `put` will halt silently, and once the channel becomes empty, any subsequent calls to `get` will halt silently. Note that any blocked calls to `put` initiated prior to closing the channel may still be allowed to return as usual.

```
boundedChannel[A].isClosed() :: Boolean
```

Definite

If the channel is currently closed, return true, otherwise return false.

```
boundedChannel[A].getOpen() :: Integer
```

Definite

Return the number of open slots in the channel. Because of concurrency this value may become out-of-date so it should only be used for debugging or statistical measurements.

```
boundedChannel[A].getBound() :: Integer
```

Definite **Pure**

Return the total number of slots (open or filled) in the channel.

```
boundedChannel[A].getAll() :: List[A]
```

Definite

Get all of the items currently in the channel or waiting to be added, emptying the channel and returning a list of the items in the order they were added. If there are no items in the channel or waiting to be added, return an empty list.

11.6.12. `Array`

```
site Array[A](Integer) :: Array[A]
```

Create a new native array of the given size. The array is initialized to contain `null`s.

The resulting array can be called directly with an index, as if its type were `lambda (Integer) :: Ref[A]`. In this case, it returns a Ref [183] pointing to the element of the array specified by an index, counting from 0. Changes to the array are reflected immediately in the ref and visa versa.

Simple example:

```
-- Publishes: 3
val a = Array(1)
a(0) := 3 >>
a(0)?
```

More complex example:

```
-- Publishes: 0 1 2
val a = Array(3)
```

```
for(0, a.length?) >i>
a(i) := i >>
stop
; a(0)? | a(1)? | a(2)?
```

```
Array[A](Integer, String) :: Array[A]
```

Create a new primitive array of the given size with the given primitive type. The initial values in the array depend on the primitive type: for numeric types, it is `0`; for booleans, `false`; for chars, the character with codepoint `0`.

The element type of the array should be the appropriate wrapper type for the given primitive type, although a typechecker may not be able to verify this. This constructor is only necessary when interfacing with certain Java libraries; most programs will just use the `Array(Integer)` constructor.

```
array[A].length? :: Integer
```

Definite **Pure**

Return the size of the array.

11.6.13. `arrayToList`

```
def arrayToList[A](Array[A]) :: List[A]
```

Given an array, create an Orc list whose elements are exactly the contents of the array, in the same order.

Implementation.

```
def arrayToList[A](Array[A]) :: List[A]
def arrayToList(a) =
  def walk(Integer, List[A]) :: List[A]
  def walk(0,acc) = acc
  def walk(i,acc) = walk(i-1, a(i-1)? : acc)
  walk(a.length?, [])
```

11.6.14. `Table`

```
def Table[A](Integer, lambda (Integer) :: A)(Integer) :: A
```

The call `Table(n,f)`, where `n` is a natural number and `f` a total function over natural numbers, creates and returns a partial, pre-computed version of `f` restricted to the range (0, `n`-1). `Table` does not return a value until all calls to `f` have completed. Consequently, if `f` halts silently on any call, the call to `Table` will halt silently.

The user may also think of the call as returning an immutable array whose `i`th element is accessed by calling `f(i)`.

This function provides a simple form of memoisation; we avoid recomputing the value of `f(i)` by internally storing the result in an array.

Example:

```
val a = Table(5, fib)
-- Publishes the 4th number of the fibonnaci sequence: 5
a(3)
```

Implementation.

```
def Table[A](Integer, lambda (Integer) :: A) :: (lambda(Integer) :: A)
def Table(n, f) =
  val a = Array[A](n) :: Array[A]
  def fill(Integer, lambda (Integer) :: A) :: Signal
  def fill(i, f) =
    if i <: 0 then signal
    else ((a(i) := f(i)), fill(i-1, f)) >> signal
  fill(n-1, f) >> (lambda (i :: Integer) = a(i)?)
```

11.6.15. `Counter`

```
site Counter(Integer) :: Counter
```

Create a new counter initialized to the given value.

11.6.16. `Counter`

```
site Counter() :: Counter
```

Create a new counter initialized to zero.

```
counter.inc() :: Signal
```

Definite

Increment the counter.

```
counter.dec() :: Signal
```

Definite

If the counter is already at zero, halt silently. Otherwise, decrement the counter and return a signal.

```
counter.onZero() :: Signal
```

Indefinite

If the counter is at zero, return a signal. Otherwise, block until the counter reaches zero.

```
counter.value() :: Integer
```

Definite

Return the current value of the counter.

Example:

```
-- Publishes five signals
val c = Counter(5)
repeat(c.dec)
```

11.6.17. `Dictionary`

```
site Dictionary() :: Dictionary
```

Create a new dictionary (a mutable map from field names to values), initially empty. The first time a field of the dictionary is accessed (using a dot access), the dictionary creates and returns a new empty Ref [183] which will also be returned on subsequent accesses of the same field. A dictionary differs from a record in that it is both mutable and dynamically extensible.

Example:

```
-- Prints: 1 2
val d = Dictionary()
  Println(d.one.read()) >>
  Println(d.two.read()) >>
  stop
| d.one.write(1) >>
  d.two.write(2) >>
  stop
```

Here is the same example rewritten using Orc's reference syntax to improve clarity:

```
-- Prints: 1 2
val d = Dictionary()
  Println(d.one?) >>
  Println(d.two?) >>
  stop
| d.one := 1 >>
  d.two := 2 >>
  stop
```

To create a multi-level dictionary, one must explicitly create sub-dictionaries for each field. For example:

```
-- Prints: 2
val d = Dictionary()
d.one := Dictionary() >>
d.one?.two := 2 >>
```

```
Println(d.one?.two?) >>
stop
```

Note that `d.one.two` is not valid: because `d.one` is a reference to a dictionary, and not simply a dictionary, a dereference step is needed before accessing a field, as in `d.one? >x> x.two`. For readers familiar with the C language, this is the same reason one must write `s->field` instead of `s.field` when `s` is a pointer to a struct.

11.6.18. `fst`

```
def fst[A,B]((A,B)) :: A
```

Return the first element of a pair.

Implementation.

```
def fst[A,B]((A,B)) :: A
def fst((x,_)) = x
```

11.6.19. `snd`

```
def snd[A,B]((A,B)) :: B
```

Return the second element of a pair.

Implementation.

```
def snd[A,B]((A,B)) :: B
def snd((_,y)) = y
```

11.6.20. `Interval`

```
site Interval[A](A, A) :: Interval[A]
```

`Interval(a,b)` returns an object representing the half-open interval [`a`,`b`).

```
interval[A].isEmpty() :: Boolean
```

Return true if this interval is empty.

```
interval[A].spans(A) :: Boolean
```

Return true if the interval spans the given point, false otherwise.

```
interval[A].intersects(Interval[A]) :: Boolean
```

Return true if the given interval has a non-empty intersection with this one, and false otherwise.

```
interval[A].intersect(Interval[A]) :: Interval[A]
```

Return the intersection of this interval with another. If the two intervals do not intersect, returns an empty interval.

```
interval[A].contiguous(Interval[A]) :: Boolean
```

Return true if the given interval is contiguous with this one (overlaps or abuts), and false otherwise.

```
interval[A].union(Interval[A]) :: Interval[A]
```

Return the union of this interval with another. Halts with an error if the two intervals are not contiguous.

11.6.21. `Intervals`

```
site Intervals[A]() :: Intervals[A]
```

Return an empty set of intervals. An Intervals object is iterable; iterating over the set returns disjoint intervals in increasing order.

```
intervals[A].isEmpty() :: Boolean
```

Return true if this set of intervals is empty.

```
intervals[A].spans(A) :: Boolean
```

Return true if this set of intervals spans the given point, and false otherwise.

```
intervals[A].intersect(Intervals[A]) :: Intervals[A]
```

Return the intersection of this set of intervals with another.

```
intervals[A].union(Interval[A]) :: Intervals[A]
```

Return the union of this set of intervals with the given interval. This method is most efficient when the given interval is before most of the intervals in the set.

11.7. text: Operations on strings.

Operations on strings.

11.7.1. `String`

```
site String
```

Strings themselves have a set of methods associated with them. These methods can be invoked on any string literal or any variable bound to a string.

The methods documented here are only a subset of those available in the Java implementation. In practice, strings in the Java implementation support all methods provided by Java's `String` class.

```
string.length() :: Integer
```

Return the length of the string.

Example:

```
-- Publishes: 4
"four".length()
```

```
string.substring(Integer, Integer) :: String
```

Return the substring of this string covered by the given half-open range.

Example:

```
-- Publishes: "orc"
val s = "apple orchard"
s.substring(6,9)
```

```
string.indexOf(String) :: Integer
```

Return the starting index of the first occurrence of the given string.

Example:

```
-- Publishes: 6
"apple orchard".indexOf("orc")
```

11.7.2. `Print`

```
site Print(Top) :: Signal
```

Definite

Print a value as a string to standard output. For Java objects, this will call `toString()` to convert the object to a String.

11.7.3. `Println`

```
site Println(Top) :: Signal
```

Definite

Same as `Print`, appending a newline.

11.7.4. `Read`

```
site Read[A](String) :: A
```

Definite **Pure**

Given a string representing an Orc value (using standard Orc literal syntax), return the corresponding value. If the argument does not conform to Orc literal syntax, halt with an error.

Example:

```
  Read("true") -- publishes the boolean true
| Read("1") -- publishes the integer 1
| Read("(3.0, [])") -- publishes the tuple (3.0, [])
| Read("\"hi\"") -- publishes the string "hi"
```

11.7.5. `Write`

```
site Write(Top) :: String
```

Definite **Pure**

Given an Orc value, return its string representation using standard Orc literal syntax. If the value is of a type with no literal syntax, (for example, it is a site), return an arbitrary string representation which is intended to be human-readable.

Example:

```
  Write(true) -- publishes "true"
| Write(1) -- publishes "1"
| Write((3.0, [])) -- publishes "(3.0, [])"
| Write("hi") -- publishes "\"hi\""
```

11.7.6. `lines`

```
def lines(String) :: List[String]
```

Split a string into lines, which are substrings terminated by an endline or the end of the string. DOS, Mac, and Unix endline conventions are all accepted. Endline characters are not included in the result.

Implementation.

```
def lines(String) :: List[String]
def lines(text) = arrayToList(text.split("\n|\r\n|\r"))
```

11.7.7. unlines

```
def unlines(List[String]) :: String
```

Append a linefeed, "\n", to each string in the sequence and concatenate the results.

Implementation.

```
def unlines(List[String]) :: String
def unlines(line:lines) = line + "\n" + unlines(lines)
def unlines([]) = ""
```

11.7.8. words

```
def words(String) :: List[String]
```

Split a string into words, which are sequences of non-whitespace characters separated by whitespace.

Implementation.

```
def words(String) :: List[String]
def words(text) = arrayToList(text.trim().split("\\s+"))
```

11.7.9. unwords

```
def unwords(List[String]) :: String
```

Concatenate a sequence of strings with a single space between each string.

Implementation.

```
def unwords(List[String]) :: String
def unwords([]) = ""
def unwords([word]) = word
def unwords(word:words) = word + " " + unwords(words)
```

11.7.10. characters

```
def characters(String) :: List[String]
```

Convert a string to a list of strings of length 1, where the list items are the characters of the given string.

Example: characters("Hello") = ["H", "e", "l", "l", "o"]

Implementation.

```
def characters(String) :: List[String]
def characters(s) =
  def rest(Integer) :: List[String]
  def rest(i) =
      Ift (i >= s.length()) >> []
    | Ift (i <: s.length()) >> s.substring(i,i+1):rest(i+1)
  rest(0)
```

11.8. time: Real time.

Real time.

11.8.1. `Rclock`

```
site Rclock() :: {. time :: (lambda () :: Integer), wait :: (lambda (Integer) :: Signal) .}
```

Create a new realtime clock instance.

```
rclock.wait(Integer) :: Signal
```

Return a signal after the given number of milliseconds.

```
rclock.time() :: Integer
```

Return the total number of milliseconds that have passed since this clock was created. Ranges from 0 to `Long.MAX_VALUE`.

11.8.2. `Rwait`

```
site Rwait(Integer) :: Signal
```

Return a signal after the given number of milliseconds.

11.8.3. `Rtime`

```
site Rtime() :: Integer
```

Return the total number of milliseconds that have passed since this program began executing. Ranges from 0 to `Long.MAX_VALUE`.

11.8.4. `metronome`

```
def metronome(Integer) :: Signal
```

Publish a signal at regular intervals, indefinitely. The period is given by the argument, in milliseconds.

11.9. util: Miscellaneous utility functions.

Miscellaneous utility functions.

11.9.1. Random

```
site Random() :: Integer
```

Return a random integer chosen from the range of all possible 32-bit integers.

11.9.2. Random

```
site Random(Integer) :: Integer
```

Return a pseudorandom, uniformly distributed integer between 0 (inclusive) and the specified value (exclusive). If the argument is 0, halt silently.

11.9.3. URandom

```
site URandom() :: Number
```

Return a pseudorandom, uniformly distributed number between 0.0 (inclusive) and 1.0 (exclusive).

11.9.4. UUID

```
site UUID() :: String
```

Return a random (type 4) UUID represented as a string.

11.9.5. Prompt

```
site Prompt(String) :: String
```

Indefinite

Prompt the user for some input. The user may cancel the prompt, in which case the site fails silently. Otherwise their response is returned as soon as it is received.

Example:

```
-- Publishes the user's name
Prompt("What is your name?")
```

The user response is always taken to be a string. Thus, integer 3 as a response will be treated as "3". To convert the response to its appropriate data type, use the library function `Read` [195]:

```
-- Prompts the user to enter an integer, then parses the response.
Prompt("Enter an integer:") >r> Read(r)
```

11.9.6. signals

```
def signals(Integer) :: Signal
```

Publish the given number of signals, simultaneously.

Example:

```
-- Publishes five signals
signals(5)
```

Implementation.

```
def signals(Integer) :: Signal
def signals(n) = if n :> 0 then (signal | signals(n-1)) else stop
```

11.9.7. for

```
def for(Integer, Integer) :: Integer
```

Publish all values in the given half-open range, simultaneously.

Example:

```
-- Publishes: 1 2 3 4 5
for(1,6)
```

Implementation.

```
def for(Integer, Integer) :: Integer
def for(low, high) =
  if low >= high then stop
  else ( low | for(low+1, high) )
```

11.9.8. upto

```
def upto(Integer) :: Integer
```

`upto(n)` publishes all values in the range `(0..n-1)` simultaneously.

Example:

```
-- Publishes: 0 1 2 3 4
upto(5)
```

Implementation.

```
def upto(Integer) :: Integer
def upto(high) = for(0, high)
```

```
import class Iterable = "java.lang.Iterable"
```

11.9.9. `IterableToStream`

```
site IterableToStream[A](Iterable[A]) :: lambda () :: A
```

Converts a Java object implementing the Iterable interface into an Orc stream backed by the object's iterator. When the site is called, if the iterator has items remaining, the next item is returned. If the iterator has no items remaining, the call halts.

11.9.10. `iterableToList`

```
def iterableToList[A](Iterable[A]) :: List[A]
```

Given a Java object implementing the Iterable interface, create an Orc list whose elements are the values produced by the object's iterator, in the same order.

Implementation.

```
def iterableToList[A](Iterable[A]) :: List[A]
def iterableToList(iterable) =
  val s = IterableToStream[A](iterable)
  def walk(List[A]) :: List[A]
  def walk(l) = s():l >m> walk(m) ; reverse(l)
  walk([])
```

11.9.11. `fillArray`

```
def fillArray[A](Array[A], lambda (Integer) :: A) :: Array[A]
```

Given an array and a function from indices to values, populate the array by calling the function for each index in the array. Publish the array once it has been populated.

For example, to set all elements of an array to zero:

```
-- Publishes: 0 0 0
val a = fillArray(Array(3), lambda (_) = 0)
a(0)? | a(1)? | a(2)?
```

Implementation.

```
def fillArray[A](Array[A], lambda (Integer) :: A)
  :: Array[A]
def fillArray(a, f) =
  val n = a.length?
  def fill(Integer, lambda(Integer) :: A) :: Bot
  def fill(i, f) =
    if i = n then stop
    else ( a(i) := f(i) >> stop
         | fill(i+1, f) )
  fill(0, f) ; a
```

11.9.12. `sliceArray`

```
def sliceArray[A](Array[A], Integer, Integer) :: Array[A]
```

Given an array and a half-open index range, create a new array which contains the elements of the original array in that index range.

Implementation.

```
def sliceArray[A](Array[A], Integer, Integer) :: Array[A]
def sliceArray(orig, from, until) =
  val size = until - from
  val a = Array[A](size)
  def copy(i :: Integer) :: Bot =
    Ift(i <: size) >>
    a(i) := orig(from + i)? >>
    copy(i+1)
  copy(0) ; a
```

11.9.13. `takePubs`

```
def takePubs[A](Integer, lambda () :: A) :: A
```

`takePubs(n, f)` calls `f()`, publishes the first `n` values published by `f()` (as they are published), and then halts.

Implementation.

```
def takePubs[A](Integer, lambda () :: A) :: A
def takePubs(n, f) =
  val out = Channel[A]()
  val c = Counter(n)
  Let(
    f() >x>
    Ift(c.dec() >> out.put(x) >> false
       ; out.closeD() >> true)
  ) >> stop | repeat(out.get)
```

11.9.14. `withLock`

```
def withLock[A](Semaphore, lambda () :: A) :: A
```

Acquire the semaphore and run a thunk which is expected to publish no more than one value. Publishes the value published by the thunk and releases the semaphore.

Implementation.

```
def withLock[A](Semaphore, lambda () :: A) :: A
def withLock(s, f) =
  s.acquire() >> (
    Let(f()) >x>
    s.release() >>
    x
    ; s.release() >> stop
  )
```

11.9.15. synchronized

```
def synchronized[A](Semaphore, lambda () :: A)() :: A
```

Given a lock and thunk, return a new thunk which is serialized on the lock. Similar to Java's synchronized keyword.

Implementation.

```
def synchronized[A](Semaphore, lambda () :: A) :: lambda() :: A
def synchronized(s,f) = lambda() = withLock(s, f)
```

11.10. web: Web browsing, HTTP, and JSON capabilities.

Web browsing, HTTP, and JSON capabilities.

11.10.1. Browse

```
site Browse(String) :: Signal
```

Attempts to open a browser window in whatever user context is available, pointing to the URL given by the string argument. The URL must be absolute and well-formed. If the URL is malformed or unreachable, the call halts silently.

Example:

```
-- Open a browser window to the Google home page
Browse("http://www.google.com")
```

The HTTP site provides a simple mechanism to send GET and POST requests to a URL.

11.10.2. HTTP

```
site HTTP(java.net.URL) :: HTTP
```

Publishes an HTTP site which accepts HTTP requests on the given URL.

```
http.get() :: String
```

Performs a GET request on the URL used to create this HTTP site. The payload is empty. The response is published as a single string.

```
http.post(String) :: String
```

Performs a POST request on the URL used to create this HTTP site. The string argument is used as a UTF-16 encoded payload. The response is published as a single string.

```
http.url :: String
```

Publishes the URL used to create this HTTP site. If query parameters were used, they will be displayed in the query part of the URL, with the appropriate encoding.

11.10.3. HTTP

```
site HTTP(String) :: HTTP
```

Converts the given string to a URL U and then behaves as HTTP(U).

11.10.4. HTTP

```
site HTTP(String, {..}) :: HTTP
```

Takes a string S and a query record Q. Maps Q to a URL query string QS by translating each record binding to a query pair, escaping characters if necessary, and then behaves as HTTP(S+QS).

11.10.5. `ReadJSON`

```
site ReadJSON(String) :: Top
```

Parses a string representation of a JSON value, producing an Orc representation of that JSON value.

An Orc representation of a JSON value, called an OrcJSON value, is either a record of OrcJSON values (representing a JSON object), a list of OrcJSON values (representing a JSON array), or a literal value (representing a JSON primitive value).

ReadJSON and WriteJSON are mutual inverses. If *S* is a string representation of a JSON value, then `WriteJSON(ReadJSON(`*S*`))` = *S*, modulo whitespace and object member ordering.

11.10.6. `WriteJSON`

```
site WriteJSON(Top) :: String
```

Serializes an Orc representation of a JSON value, producing a string representation of that JSON value.

An Orc representation of a JSON value, called an OrcJSON value, is either a record of OrcJSON values (representing a JSON object), a list of OrcJSON values (representing a JSON array), or a literal value (representing a JSON primitive value).

WriteJSON and ReadJSON are mutual inverses. If *V* is an Orc representation of a JSON value, then `ReadJSON(WriteJSON(`*V*`))` = *V*.

11.11. xml: XML manipulation.

XML manipulation.

11.11.1. ReadXML

```
site ReadXML(String) :: XML
```

Parses a string representation of XML into a structured form that Orc can manipulate.

11.11.2. WriteXML

```
site WriteXML(XML) :: String
```

Serializes Orc's representation of XML to a string.

11.11.3. XMLElement

```
site XMLElement(String, {. .}, List[XML]) :: XML
```

Creates an XML element node with the given tag, attributes, and children. This site may also be used for matching.

11.11.4. XMLText

```
site XMLText(String) :: XML
```

Creates an XML text node with the given contents. Encoding may occur. This site may also be used for matching.

11.11.5. XMLCData

```
site XMLCData(String) :: XML
```

Creates an XML text node with the given contents. Contents will not be encoded. This site may also be used for matching.

11.11.6. IsXML

```
site IsXML(Top) :: XML
```

Acts as the identity function for any XML node, and halts silently for any non-XML argument.

11.11.7. xml

```
def xml(String, List[XML]) :: XML
```

Creates an XML element with the given tag and children, and no attributes. May also be used for matching. When matching, the second argument is a multimatch on each child, rather than a match on the list of children.

Implementation.

```
val xml =
  def toxml(String, List[Top]) :: XML
  def toxml(tag, children) =
    def liftChild(Top) :: XML
    def liftChild(x) = IsXML(x) ; XMLText("" + x)
    XMLElement(tag, {. .}, map(liftChild, children))
  def fromxml(XML) :: Top
  def fromxml(XMLElement(tag,attr,children)) =
    each(children) >c>
      ( c >XMLElement(_,_,_)> (tag,c)
      | c >XMLText(s)> (tag,s)
      | c >XMLCData(s)> (tag,s) )
  def fromxml(XMLText(s)) = s
  def fromxml(XMLCData(s)) = s
  {. apply = toxml, unapply = fromxml .}
```

11.11.8. `xattr`

```
def xattr(XML, {. .}) :: XML
```

Creates a copy of the XML element, adding new attributes as given by the record argument. If there is a conflict, the new attributes override the old ones. May also be used for matching.

Implementation.

```
val xattr =
  def toattr(XML, {. .}) :: XML
  def toattr(XMLElement(tag, attr, children), moreattr) =
    XMLElement(tag, attr + moreattr, children)

  def fromattr(XML) :: (XML, {. .})
  def fromattr(XMLElement(tag, attr, children)) =
    (XMLElement(tag, {. .}, children), attr)
  {. apply = toattr, unapply = fromattr .}
```

Index of Key Terms

This index is meant to direct the reader to key terms and concepts in this reference manual. It is not a comprehensive index that lists every occurrence of every term.

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Z