A TREE SEMANTICS OF AN ORCHESTRATION LANGUAGE

Tony Hoare, Microsoft Research Labs, Cambridge, U.K.
Galen Menzel, University of Texas, Austin, Texas 78712, USA
Jayadev Misra, University of Texas, Austin, Texas 78712, USA*
email: thoare@microsoft.com, galen@alumni.utexas.net, misra@cs.utexas.edu

Abstract This paper presents a formal semantics of a language, called Orc, which is described in a companion paper[3] in this volume. There are many styles of presentation of programming language semantics. The more operational styles give more concrete guidance to the implementer on how a program should be executed. The more abstract styles are more helpful in proving the correctness of particular programs. The style adopted in this paper is neutral between implementer and programmer. Its main achievement is to permit simple proofs of familiar algebraic identities that hold between programs with different syntactic forms.

1. Introduction

This paper presents a formal semantics of a language, called Orc, which is described in a companion paper[3] in this volume. Orc is designed to orchestrate executions of independent entities, called *sites*. Orchestration means creating multiple threads of execution where each thread calls a sequence of sites. Orc permits creating new threads, passing data among threads and selectively pruning threads. A detailed discussion of Orc and its application in a variety of problem areas appears in the accompanying paper. The semantics of Orc proposed here represents all possible threads of execution by a labelled tree. The tree is completely defined by the syntactic structure of the program, independent of its actual execution. Two programs are considered equivalent if their representative trees are equal. We show how to use the tree as a basis for reasoning about program executions.

A semantics for a programming language defines the intended meaning of all syntactically correct programs in the language. It serves as a contractual

*Work of this author is partially supported by the National Science Foundation grant CCR–0204323.

interface between a language's implementers and its users. Each implementer must guarantee that every execution of every program satisfies the definition, and each programmer may rely on this guarantee when writing programs.

There are many styles of presentation of programming language semantics. The operational styles give more concrete guidance to the implementer on how a program should be executed. The abstract styles are more helpful in proving the correctness of particular programs. The style adopted in this paper is neutral between implementer and programmer. Its main achievement is to permit simple proofs of familiar algebraic identities that hold between programs with different syntactic forms. The algebraic laws may help the implementer in designing automatic optimization strategies, and the programmer in developing efficient algorithms, while preserving correctness by construction.

The presentation of our semantics proceeds in two stages. First, the text of the program is translated into an abstract tree form. A standard collection of algebraic laws is proved simply by graph isomorphism. This is done in section 3. A particular execution of the program can then be recorded by annotating the nodes of the tree with information about the values of the program variables and the times at which they are assigned. Because a program is non-deterministic, there are many ways to do this: we define the complete set of valid annotations by healthiness conditions, which can be checked independently at every node of the tree. This is done in section 4. The paper concludes with a summary of further research that may prove useful for both the implementation and use of the language.

2. Syntax and Semantics

We describe the syntax and operational semantics of Orc in this section. This material is condensed from the companion paper[3] in this volume. We include it here for completeness. We encourage the reader to consult the companion paper for more details and examples.

2.1 Syntax

A computation is started from a host language program by executing an *Orc statement*

$z :\in f$([*actual-parameter*])

where z is a variable of the host program, called the *goal* variable, f is the name of a (defined) expression, called the *goal* expression, and [*actual-parameter*] is a (possibly empty) comma-separated list of actual parameters. The syntax of expression definition is

exprDefinition ::= *exprName*([*formal-parameter*]) $\triangleq$ *expr*
formal-parameter ::= *variable*

Next, we define the syntactic unit *expr* which denotes an Orc expression. Below, f and g are Orc expressions, F is the name of an expression defined separately, and x is a variable.

expr	::=	*term* $\mid$ $f \gg g$ $\mid$ $f \mid g$ $\mid$ f **where** $x :\in g$ $\mid$ F([*actual-parameter*])
term	::=	$\mathbf{0}$ $\mid$ $\mathbf{1}$ $\mid$ *site*([*actual-parameter*])
actual-parameter	::=	*constant* $\mid$ *variable* $\mid$ θ

Binding powers of the operators. The binding powers of the operators in increasing order of precedence are: $\triangleq$, **where** , $:\in$, $\mid$, $\gg$.

Well-formed expressions. The *free* variables of an expression are defined as follows, where M is a site or an expression name and L is a list of its variable parameters.

$free(\mathbf{0}) = \{\}$, $free(\mathbf{1}) = \{\}$
$free(M(L)) = \{x \mid x \in L\}$
$free(f\ op\ g) = free(f) \cup free(g)$, where *op* is $\mid$ or $\gg$
$free(f$ **where** $x :\in g) = (free(f) - \{x\}) \cup free(g)$

Variable x is *bound* in f if it is named in f and is not free. The binding rule is: given $((f$ **where** $x :\in g)$, any free occurrence of x in f is bound to its binding occurrence, the x defined just after **where** . We rename all bound variables in **where** expressions so that all variable names in an expression are distinct.

Expression f is *well-formed* if all free variables of it are its formal parameters.

Tag elimination. The full syntax of Orc includes *tags*, which are variables used to pass values of one expression to another. For example, we write $M >x> N(x)$, where x is a tag, to assign a name to the value produced by M, and to pass this value to N. We do not consider tags in this paper because they can be eliminated using the following identity.

$$(f >x> g) = (f \gg \{g \textbf{ where } x :\in \mathbf{1}\})$$

2.2 Operational semantics

In this section, we describe the semantics of Orc in operational terms. Evaluation of an expression (for a certain set of global variable values) yields a stream of values, which may be empty, non-empty but finite, or infinite. Additionally, the evaluation may assign values to certain tags and local variables. We describe the evaluation procedure for expressions based on their syntactic structures.

2.2.1 Site call. The simplest expression is a term representing a site call. To evaluate the expression, call the site with the appropriate parameter values. If the site responds, its return value is the (only) value of the expression. In this paper we do not ascribe any semantics to a site call. Its behavior is taken as entirely arbitrary, and therefore consistent with any semantics that may be ascribed later.

2.2.2 Operator $\gg$ for sequential composition. Operator $\gg$ allows sequencing of site calls and passing values between them. To explain the sequencing mechanism, we consider first $M \gg N$ where both operands are site calls. Evaluation of $M \gg N$ first calls M, and on receiving the response from M calls N. The value of the expression is the value returned by N. The value returned by M is referred to as θ; so in $M \gg R(\theta)$, R is called with the value returned by M as its argument. Each application of sequencing reassigns the value of θ; so in $M \gg R(\theta) \gg S(\theta)$, the first occurrence of θ refers to the value produced by M and the latter to the value produced by R.

When an expression produces at most one value, $\gg$ has the same meaning as the sequencing operator in a conventional sequential language (like ";" in Java). For expression $f \gg g$, where f and g are general Orc expressions, f produces a stream of values, and each value causes a fresh evaluation of g. The values produced by all instances of g in time-order is the stream produced by $f \gg g$. Note that during the evaluation of $f \gg g$, threads for both f and several instances of g may be executing simultaneously. We elaborate on this in section 2.2.3.

2.2.3 Operator $|$ for symmetric parallel composition. Using the sequencing operator, we can only create single-threaded computations. We introduce $|$ to permit symmetric creations of multiple threads. Evaluation of $(M \mid N)$ creates two parallel threads (one for M and one for N), and produces a stream containing the values returned by both threads in the order in which they are computed.

In general, evaluation of $f \mid g$, where f and g are Orc expressions, creates two threads to compute f and g, which may, in turn, spawn more threads. The evaluation produces a series of site calls, which are merged in time order. The result from each thread is a stream of values. The result from $f \mid g$ is the merge of these two streams in time order. If both threads produce values simultaneously, their merge order is arbitrary. Treatment of this is postponed to section 4.

It is instructive to consider the expression $(M \mid N) \gg R$. The evaluation starts by creating two threads to evaluate M and N. Suppose M returns a value first. Then R is called. If N returns a value next, R is called again. That is, each value from $(M \mid N)$ spawns a thread for evaluating the remaining part

of the expression. In $(M \mid N) \gg R(\theta)$, the value that spawns the thread for computing $R(\theta)$ is referenced as θ.

Expressions $M \mid M$ and M are different; the former makes two parallel calls to M, and the latter makes just one. Therefore, M produces at most one value, whereas $M \mid M$ may produce two (possibly identical) values. The expression $M \gg (N \mid R)$ is different from $M \gg N \mid M \gg R$. In the first case, exactly one call is made to M, and N and R are called after M responds. In the second case two parallel calls are made to M, and N and R are called only after the corresponding calls respond. The difference is significant where M returns different values on each call, and N and R use those values. The two computations are depicted pictorially in figure 1.

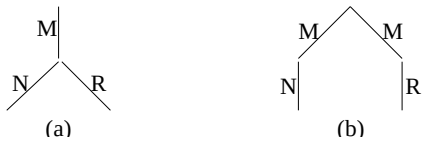

Figure 1. (a) $M \gg (N \mid R)$ and (b) $M \gg N \mid M \gg R$

2.2.4 Operator where for asymmetric parallel composition. An expression with a **where** clause (henceforth called a **where** expression), has the form $\{f$ **where** $x :\in g\}$. Expression f may name x as a parameter in some of its site calls. Evaluation of the **where** expression proceeds as follows. Evaluate f and g in parallel. When g returns its first result, assign the result to x and terminate evaluation of g. During evaluation of f, any site call which does not name x as a parameter may proceed, but site calls in which x is a parameter are deferred until x acquires a value. The stream of values produced by f under this evaluation strategy is the stream produced by $\{f$ **where** $x :\in g\}$.

2.2.5 Expression call. An expression call is like a function call; the body of the expression is substituted at the point of the call after assigning the appropriate values to the formal parameters. Unlike a function call, an expression returns a stream of values, not just one value. The values returned are non-deterministic, because the sites it calls may be non-deterministic.

2.2.6 Constant terms. There are two constant terms in Orc: **0** and **1**. Treat each as a site. Site **0** never responds and **1** responds immediately with the value of θ.

2.2.7 Defining Orc expressions. In Orc, an expression is defined by its name, a list of parameters which serve as its global variables, and an expression which serves as its body. For example,

$$\begin{array}{ll} BM(0) & \triangleq\ \mathbf{0} \\ BM(n+1) & \triangleq\ S \mid R \gg BM(n) \end{array}$$

defines the name BM, and specifies its formal parameter and body. Calling $BM(2)$, for instance, starts evaluation of a new instance of BM with actual parameter 2, which produces a stream of values.

The definition of BM is well-grounded so that for every n, $n \geq 0$, $BM(n)$ calls a finite number of sites and returns a finite number of values. Orc has expression definitions which are not well-grounded to allow for infinite computations. For example, a call to E where

$$E \ \triangleq\ S \mid R \gg E$$

may cause an unbounded number of site calls (and produce an unbounded number of values). However, every call of E occurs inside some context $x :\in \ \ldots E \ldots$, which assigns to x only the first value produced by the expression. Further computations, i.e., all later site calls by E and the values it returns, can be truncated. Thus all computations of interest depend only on finite recursion depth and a finite tree.

2.2.8 Starting and ending a computation. A computation is started from a host language program by executing an *Orc statement*

$z :\in\ f$([*actual-parameter*])

where z is a variable of the host program and f is the name of an expression, followed by a list of actual parameters. All actual parameters have values before f's evaluation starts. To execute this statement, start the evaluation of f with actual parameters substituted for the formal ones, assign the first value produced to variable z, and then terminate the evaluation of f. If f produces no value, the execution of the statement does not terminate.

3. A Semantic Model

We develop a semantic model of Orc which is defined in a manner entirely independent of the behaviors of the sites and the meanings of the site calls. As a result, two expressions that are equal in this semantics behave exactly alike when executed in the same environment, whatever that may be.

The model is denotational. The denotation of an expression is a tree whose edges are labelled with site calls, and whose nodes are labelled with declarations of local variables (i.e., their names and the associated trees) and a natural

number called *size*. Paths denote threads of execution. A path ending at a node of size n produces the value associated with the node n times as results of expression evaluation.

Two expressions are equal if both are well-formed and their denotation trees are equal. Equal expressions are interchangeable in every context. In this section, we look at the equality problem; in the next section, we show how to depict executions using denotation trees.

Informal Description of the Equality Theory. It is customary to regard two expressions equal if one can be replaced by the other within any expression. This suggests that equal expressions f and g produce the same *external effect* (i.e., call the same sites in the same order) and the same *internal effect* (i.e., produce the same values). Therefore, $(M \mid N)$ and $(N \mid M)$ are equal. In evaluating these expressions, we make the same site calls in both cases and produce the same values, no matter which sites respond. If only M responds, say, both expressions will produce the same value, the response received from M.

We create the tree from an expression assuming that *every site responds*. Thus, two expressions are equal if they have the same tree. This notion of equality, properly refined, is appropriate even when some sites may not respond during an execution, because then both expressions will behave identically. For example, consider $((M \mid N) \gg R)$ and $(M \gg R \mid N \gg R)$. If N does not respond and M does during an evaluation, both expressions will produce the value from the sequence MR (provided R responds). Moreover, they would have made identical site calls, to M and N simultaneously and to R after M responds.

A value produced by an expression is derived from the response of a site call. So for the equality of f and g, we need only establish that f and g have identical tree structures where corresponding nodes have the same size. To see the need for the latter requirement, consider $(M \gg \mathbf{0})$ and M. They both make the same site call, to M, though only the latter produces a value. And in $M \gg (\mathbf{1} \mid \mathbf{1})$, the value received from M is produced twice as the result of expression evaluation, whereas in M the value is produced only once. The size of a node (in these cases, the respective terminal nodes) denotes the number of times the corresponding value is produced.

3.1 The denotation tree

3.1.1 Structure of the denotation tree. The denotation of an expression is a *tree*. The tree has at least one node (its root), and it may be infinite. Each edge of the tree is labelled with a site call of the form $M(L)$, where L is a list of parameters. Each node has a set of *declarations*, where a declaration consists of a variable name and a denotation tree. The set of declarations may

be empty; otherwise, the variable names in the declarations at a node are distinct. Each node has a *size*, a natural number[1]. Size n specifies that during an execution the value associated with this node (i.e., received from the site call ending at this node) appears n times as the result of expression evaluation.

Bound Variable Renaming. Declarations at a node correspond to introduction of local variables. The reference to variable x in an edge label, say $M(x)$, is bound to x which is declared at the closest ancestor of this edge.

We may rename variable x, which appears in a declaration at a node, by y provided y is not the name of any variable in that declaration. Then, we replace x by y for all occurrences of x bound to this declaration. Renaming does not change any property of the tree.

3.1.2 Operations on denotation trees. We define three operations on denotation trees: *join* ($\cup$), *graft* ($+\!\!+$) and *declare* ($\circ$).

To compute $P \cup Q$ for trees P and Q, create a tree where P and Q share the root. The declarations at the root is the union of the declarations at the roots of P and Q; ensure distinct names in the declarations by renaming variables in P or Q. The size at the root is the sum of the sizes of both roots.

To compute $P +\!\!+ Q$, at each node u of P which has size n, $n > 0$, attach n copies of Q, as follows. First, join n copies of Q as described above; call the result Q^n. Let q be the root of Q and q^n of Q^n. Node q^n has n distinctly named variables for each variable declared at q and its size is $m \times n$, where m is the size of q. Next: (1) set the declarations at u to the union of the declarations at u and q^n, (2) set the size of u to that of q^n (i.e., $m \times n$), and (3) make all children of q^n children of u. Note that a node of P whose size is 0 is unaffected by graft.

To compute $(x, Q) \circ P$, add the declaration (x, Q) to the declarations at the root of P; rename x if necessary to avoid name clash. In $(x, Q) \circ P$, tree Q is a *subordinate* of tree P.

Two trees are equal if they are identical in all respects as unordered trees after possible renamings. Specifically, equal trees have 1-1 correspondence between their nodes and their edges so that (1) corresponding nodes have the same declarations (i.e., same variables and equal associated trees) and same size, and (2) corresponding edges have the same label and they are incident on nodes which correspond.

Simple facts about join, graft and declare. In the following, P, Q and R are trees, and c and d are declarations.

[1] In general, the size is an ordinal; see section 3.3.

1 $\cup$ is commutative and associative.

2 $+\!\!+$ is associative.

3 $(P \cup Q) +\!\!+ R = (P +\!\!+ R) \cup (Q +\!\!+ R)$

4 $d \circ (P +\!\!+ Q) = (d \circ P) +\!\!+ Q$

5 $d \circ (P \cup Q) = (d \circ P) \cup Q$

6 $c \circ (d \circ P) = d \circ (c \circ P)$

The commutativity and associativity of $\cup$ follow from its definition. The associativity of $\gg$ is also easy to see pictorially, but we sketch a proof.

A copy of R in $(P +\!\!+ Q) +\!\!+ R$ is equal to any copy of R in $P +\!\!+ (Q +\!\!+ R)$, because there has been no graft on (i.e., attachments to the nodes of) R in either tree. Next, we show that any copy of Q in one tree is equal to any in the other. The copies are identical because in both cases R has been grafted to Q, and grafting R to identical trees results in identical trees. To complete the proof, first note that in $(P +\!\!+ Q) +\!\!+ R$ and $P +\!\!+ (Q +\!\!+ R)$ there is exactly one copy of tree P. We show that copies of P in both trees are identical; i.e., corresponding nodes u and v in both trees have: (1) equal numbers of copies of Q and R attached to them; so, their declarations are identical, (2) equal sizes, and (3) identical edges of P incident on them. The proof of part (3) is trivial, because the edges of P are unaffected by graft. To prove (1) and (2), let the size of u (and v) in P be m, and the sizes of the roots of Q and R be n and r, respectively. The number of copies of Q attached to u in $(P +\!\!+ Q) +\!\!+ R$ is m, because it is the same as in $(P +\!\!+ Q)$. The number of copies of Q attached to v in $P +\!\!+ (Q +\!\!+ R)$ is again m because the size of v in P is m. And the number of copies of R attached to either is $m \times n$. The sizes of both u and v, in $(P +\!\!+ Q) +\!\!+ R$ and $P +\!\!+ (Q +\!\!+ R)$ respectively, are $m \times n \times r$.

Proof of $(P \cup Q) +\!\!+ R = (P +\!\!+ R) \cup (Q +\!\!+ R)$ is direct from the tree construction. Note that $P +\!\!+ (Q \cup R) \neq (P +\!\!+ Q) \cup (P +\!\!+ R)$. To see this, let P, Q and R have single variable edges labelled M, N and R respectively. The tree for $P +\!\!+ (Q \cup R)$ is given in figure 1(a) and for $(P +\!\!+ Q) \cup (P +\!\!+ R)$ in figure 1(b) (in page 5); they are different.

The proof of $d \circ (P +\!\!+ Q) = (d \circ P) +\!\!+ Q$ follows from the tree structure; declaration d appears at the root of tree P in both cases. And $d \circ (P \cup Q) = (d \circ P) \cup Q$ because in both cases declarations at the root are equal; d is added to the set of declarations of $(P \cup Q)$. We have $c \circ (d \circ P) = d \circ (c \circ P)$ because the declarations form a set and "$\circ$" adds a declaration to the set.

3.1.3 Denotations of expressions. Write $tree(f)$ for the denotation tree of expression f. For **0**, the tree has a single node (the root) whose size is 0. For **1**, the tree has a single node (the root) whose size is 1. For a site call

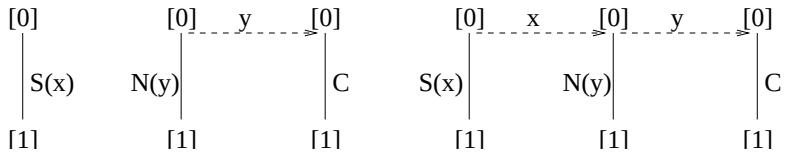

Figure 2. Trees for $S(x)$, $\{N(y)$ **where** $y :\in\ c\}$, $(S(x)$ **where** $x :\in\ \{N(y)$ **where** $y :\in\ c\})$

of the form $M(L)$, the tree consists of a single edge labelled with the term, the root has size 0 and the terminal node size 1. There are no declaration in any of these cases. The rest of the tree-construction rules follow.

- $tree(f \gg g)\ =\ tree(f) +\!\!+ tree(g)$
- $tree(f \mid g)\ =\ tree(f) \cup tree(g)$
- $tree(f$ **where** $x :\in\ g)\ =\ (x, tree(g)) \circ tree(f)$
- For expression F where $F\ \underline{\Delta}\ f$:
  $tree(F)$ is the least fixed point of the equation $tree(F) = tree(f)$.

We have described the operations $+\!\!+$, $\cup$ and $\circ$ earlier. In the definition of $tree(f$ **where** $x :\in\ g)$, $tree(g)$ is a *subordinate* tree of $tree(f)$. We treat least fixed point in more detail in section 3.3.

Example. We construct the denotation of

$$\begin{array}{ll} & (S(x) \textbf{ where } x :\in\ \{N(y) \textbf{ where } y :\in\ c\}) \\ \gg & (\{(\mathbf{1} \mid N(x)) \gg R(y) \textbf{ where } x :\in\ A \gg \mathbf{0}\} \textbf{ where } y :\in\ B) \end{array}$$

The construction involves all three operations, $+\!\!+$, $\cup$ and $\circ$.

In the figures, the size of a node is enclosed within square brackets. We show a declaration by drawing a dashed edge to the root of the subordinate tree and labeling the edge with the variable name.

For $S(x)$, $\{N(y)$ **where** $y :\in\ c\}$ and $(S(x)$ **where** $x :\in\ \{N(y)$ **where** $y :\in\ c\})$ the trees are shown in figure 2.

The trees for $\mathbf{1}$, $N(x)$ and $(\mathbf{1} \mid N(x))$ are in figure 3.

The trees for $R(y)$ and $(\mathbf{1} \mid N(x)) \gg R(y)$ are in figure 4.

The tree for $\{(\mathbf{1} \mid N(x)) \gg R(y)$ **where** $x :\in\ A \gg \mathbf{0}\}$ is in figure 5.

The tree for $(\{(\mathbf{1} \mid N(x)) \gg R(y)$ **where** $x :\in\ A \gg \mathbf{0}\}$ **where** $y :\in\ B)$ is in figure 6.

The tree for the whole expression is in figure 7.

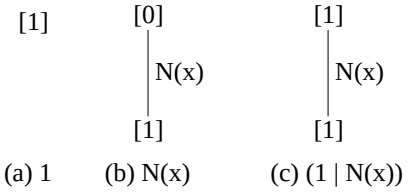

Figure 3. Trees for $\mathbf{1}$, $N(x)$ and $(\mathbf{1} \mid N(x))$

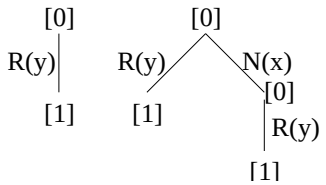

Figure 4. Trees for $R(y)$ and $(\mathbf{1} \mid N(x)) \gg R(y)$

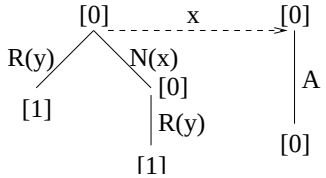

Figure 5. Tree for $\{(\mathbf{1} \mid N(x)) \gg R(y)$ **where** $x :\in A \gg \mathbf{0}\}$

3.2 Laws obeyed by Orc expressions

Well-formed expressions f and g are equal if their trees are identical. (See section 2.1 for definition of well-formed expressions.)

$$f = g \text{ iff } tree(f) = tree(g) \ .$$

Equal expressions are interchangeable in any context.

We list a number of laws about Orc expressions. The laws in section 3.2.1 are also valid for regular expressions of language theory (which is a Kleene algebra[1]). Orc expressions without **where** clauses can be regarded as regular expressions. An Orc term corresponds to a symbol in a regular expression: $\mathbf{0}$ and $\mathbf{1}$ correspond to the empty set and the set that contains the empty string, and $\mid$ and $\gg$ correspond to alternation and concatenation. There is no operator in Orc corresponding to $*$ of regular expressions, which we simulate using recursion. The **where** operator of Orc has no counterpart in language theory.

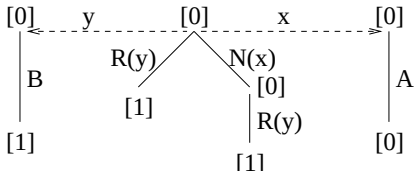

Figure 6. Tree for $(\{(\mathbf{1} \mid N(x)) \gg R(y)$ **where** $x :\in A \gg \mathbf{0}\}$ **where** $y :\in B)$

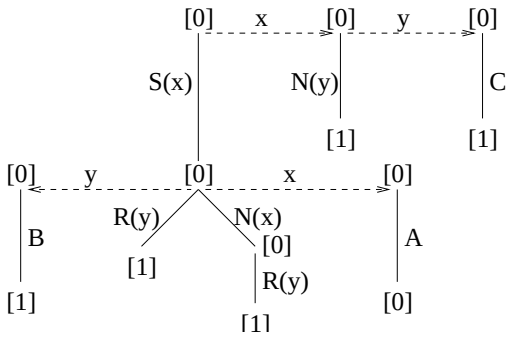

Figure 7. The tree for the whole expression

3.2.1 Kleene laws. All Orc expressions, including **where** expressions, obey the laws given in this section. Below f, g and h are Orc expressions. In all the identities, one side is well-formed iff the other side is.

(Zero and $\mid$)	$f \mid \mathbf{0} = f$
(Commutativity of $\mid$)	$f \mid g = g \mid f$
(Associativity of $\mid$)	$(f \mid g) \mid h = f \mid (g \mid h)$
(Left zero of $\gg$)	$\mathbf{0} \gg f = \mathbf{0}$
(Left unit of $\gg$)	$\mathbf{1} \gg f = f$
(Right unit of $\gg$)	$f \gg \mathbf{1} = f$
(Associativity of $\gg$)	$(f \gg g) \gg h = f \gg (g \gg h)$
(Right Distributivity of $\gg$ over $\mid$)	$(f \mid g) \gg h = (f \gg h \mid g \gg h)$

(Zero and $\mid$) $(f \mid \mathbf{0}) = f$: Join with $\mathbf{0}$ does not affect the size or the declarations at the root of f.

(Commutativity of $\mid$) $f \mid g = g \mid f$: From the commutativity of join.

(Associativity of $\mid$) $(f \mid g) \mid h = f \mid (g \mid h)$: From the associativity of join.

(Left zero of $\gg$) $\mathbf{0} \gg f = \mathbf{0}$: The tree for $\mathbf{0}$ has only a node of size zero; so, grafting has no effect.

(Left unit of $\gg$) $\mathbf{1} \gg f \;=\; f$: The tree for $\mathbf{1}$ has only a root node of size one; so, grafting f produces f.

(Right unit of $\gg$) $f \gg \mathbf{1} \;=\; f$: Similar arguments as above.

(Associativity of $\gg$) $(f \gg g) \gg h \;=\; f \gg (g \gg h)$: Operation graft is associative.

(Right Distributivity of $\gg$ over $\mid$) $(f \mid g) \gg h \;=\; (f \gg h \mid g \gg h)$: From the right distributivity of graft over join.

Some of the axioms of Kleene algebra do not hold in Orc. First is the *idempotence* of $\mid$, $f \mid f \;=\; f$. Expressions $M \mid M$ and M are different because the corresponding trees have different sizes at the terminal nodes. In Kleene algebra, $\mathbf{0}$ is both a right and a left zero. In Orc, it is only a left zero; that is, $f \gg \mathbf{0} \;=\; \mathbf{0}$ does not hold: expression $(M \gg \mathbf{0})$ differs from $\mathbf{0}$, because the corresponding trees are different. Another axiom of Kleene algebra is the left distributivity of $\gg$ over $\mid$: $f \gg (g \mid h) \;=\; (f \gg g) \mid (f \gg h)$. This does not hold in Orc because, as we have shown, the $+\!\!+$ does not left distribute over $\cup$.

3.2.2 Laws for `where` expressions. The following laws for **where** expressions have no counterpart in Kleene algebra.

(Distributivity over $\gg$)
$\{f \gg g \textbf{ where } x :\in h\} \;=\; \{f \textbf{ where } x :\in h\} \gg g$

(Distributivity over $\mid$)
$\{f \mid g \textbf{ where } x :\in h\} \;=\; \{f \textbf{ where } x :\in h\} \mid g$

(Distributivity over **where**)
$\{\{f \textbf{ where } x :\in g\} \textbf{ where } y :\in h\} \;=\; \{\{f \textbf{ where } y :\in h\} \textbf{ where } x :\in g\}$

On the need for well-formedness. For the laws given above, both sides of an identity must be checked syntactically for well-formedness. Unlike the laws in section 3.2.1, both sides may not be well-formed if only one side is. Consider

$p = (M \mid N(x) \textbf{ where } x :\in g)$

We show below that $tree(p)$ is identical to both $tree(q)$ and $tree(r)$, where

$q = (M \textbf{ where } x :\in g) \mid N(x)$
$r = (N(x) \textbf{ where } x :\in g) \mid M$

Expression r is well-formed though q is not, because x in the term $N(x)$ is not bound to any variable. So, $p \neq q$ though $p = r$.

Proofs. Use the following abbreviations.

$$\begin{array}{lll} P = tree(f) & Q = tree(g) & R = tree(h) \\ Q' = (x, tree(g)) & R' = (x, tree(h)) & R'' = (y, tree(h)) \end{array}$$

The required proofs are

$$\begin{array}{rcl} R' \circ (P +\!\!+ Q) & = & (R' \circ P) +\!\!+ Q \\ R' \circ (P \cup Q) & = & (R' \circ P) \cup Q \\ R'' \circ (R' \circ P) & = & R' \circ (R'' \circ P) \end{array}$$

These results follow directly from the properties of declaration ($\circ$); see section 3.1.2 (page 8).

3.3 Least Fixed point

To construct the tree for an expression call we need to solve an equation. To handle expression calls with parameters, say, $F(s)$ where $F \triangleq (\lambda r.f)$, find the least fixed point of $tree(F(s)) = tree(F \triangleq (\lambda r.f)s)$. To do this, replace all formal parameters by the actual parameter values and then construct the least denotation tree. As an example, consider the definition

$$\begin{array}{ll} BM(0) & \triangleq \mathbf{0} \\ BM(n+1) & \triangleq S \mid R \gg BM(n) \end{array}$$

To construct $tree(BM(2))$, say, we solve the following equations, which results in the tree shown in figure 8.

$$\begin{array}{rcl} tree(BM(2)) & = & tree(S \mid R \gg BM(1)) \\ tree(BM(1)) & = & tree(S \mid R \gg BM(0)) \\ tree(BM(0)) & = & tree(\mathbf{0}) \end{array}$$

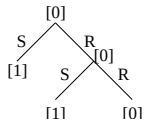

Figure 8. Solution of the equations for $BM(2)$

The solution is more involved for the following definition where the number of equations is infinite.

$$E \triangleq S \mid R \gg E$$

The resulting tree is the least fixed point of this equation. It is obtained by a chain of approximations for E: start with the approximation $\mathbf{0}$ for E, and substitute each approximation for E into the equation to obtain a better approximation. We claim (though we do not prove in this paper) that the limit of this chain of approximations is the least fixed point of the given equation. The first few approximations are shown in figure 9. The sequence is same as $BM(0)$, $BM(1)\ldots$, shown above.

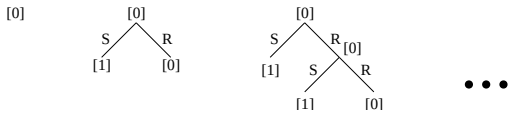

Figure 9. Approximations for the least fixed point of $E \triangleq S \mid R \gg E$

The reader may show that both $(E = E)$ and $(E = E \gg f)$ have $\mathbf{0}$ as their least fixed points. The least fixed point of $(E = M \gg E)$ is the tree which is a single infinite chain of edges labelled M in which every node has size zero. And for $(E = \mathbf{1} \mid E \gg M)$, the least fixed point is a denotation of the infinite expression $\mathbf{1} \mid M \mid M \gg M \mid M \gg M \gg M \mid \ldots$ (which is not a valid Orc expression) whose terminals have size one.

The need for ordinal as size. Theoretically, we need ordinals to represent sizes in denotation trees. Consider the equation $(E = \mathbf{1} \mid E)$. Its tree has a single node (root) with size ω. And, $E \gg E$ has a tree whose root has size ω^2. Similarly, $E = (\{M(x)$ **where** $x :\in g\} \mid E)$ has an infinite number of declarations at its root. Such expressions are rare in practice, and they are unimplementable because their executions have to create infinite number of threads simultaneously.

The least fixed point of $(E = M \mid E)$ is $(M \mid M \mid \cdots)$ which is $(!M)$ of Pi-calculus [2]. In this case, the tree has infinite degree at the root, but each terminal node has size one.

4. Healthiness conditions for executions

In this section, we augment the denotation tree with additional information to record the steps of an execution. An execution is a history of site calls (the actual parameter values passed to the sites and the times of the calls), the responses received from the sites (the values received and the times of receipt), and the assignments of values to the local variables. We record these steps by attaching a state to each node of the tree, as we explain below.

A node u in the tree has an associated state $u.state$. A *state* is an assignment of values to variables; we write $u.x$ for the value of x in $u.state$, provided x is

defined in that state. A value is a tuple $(magnitude, time)$, where $magnitude$ holds the actual value of the variable and $time$ denotes the time at which the magnitude is computed; write $u.x.time$ for the time component of x in $u.x$. The times associated with different variables in a state may be different, as we explain in section 4.4. The time at which node u is reached in a computation is $u.\theta.time$.

Notation. For node u

$u.state$:	the state of u
$u.def$:	the set of variables (including θ) defined in $u.state$
$u.decl$:	the set of variables declared at u
$u.null$:	is true iff $u.def = \{\}$
$u.x$:	the value of x in $u.state$
$u.x.time$:	the time component of x in $u.x$

For any execution, the states associated with the nodes satisfy certain healthiness conditions, which we specify in this section. Conversely, any set of states which satisfy the healthiness conditions is a possible execution. We give the healthiness conditions in three parts: (1) edge conditions, which specify for each edge (u, v) the relationship between $u.state$ and $v.state$, (2) root conditions, which specify the states at the roots of the goal tree and all subordinate trees, and (3) the conditions for subordinate computations, which give the semantics of **where** expressions.

Convention. Assume that a tree and all its subordinates have been renamed so that a variable is declared in at most one node.

4.1 Edge conditions

For edge (u, v), whose label is $M(L)$, $v.state$ specifies the effect of site call $M(L)$ in $u.state$.

$$
\begin{aligned}
\neg v.null \;\Rightarrow\quad & \neg u.null \\
& \wedge\ \theta \in v.def \\
& \wedge\ u.def = v.def - v.decl \\
& \wedge\ (\forall x : x \in u.def \wedge x \neq \theta : u.x = v.x) \\
& \wedge\ (\forall y : y \in L : y \in u.def \wedge v.\theta.time \geq u.y.time) \\
& \wedge\ v.\theta.time \geq u.\theta.time \qquad \textbf{(edge condition)}
\end{aligned}
$$

We study the conjuncts in turn. The state of v is non-null only if u's state is non-null (so a null state propagates down the tree). In every non-null state θ is defined (we will require in section 4.2 that θ be defined at each root). All variables defined in $u.state$ are also defined in $v.state$; the additional variables

defined in $v.state$ are declared at v. Values of the variables in $u.state$ (other than θ) are the same in $v.state$. The site call is made only if all parameters of the call are defined in $u.state$. Next, we justify the conditions on $v.\theta.time$ in the last two conjuncts.

The site call is made no earlier than $u.y.time$ for any parameter y, for $y \in L$, because the value of y is not available before $u.y.time$. And the call is made no earlier than $u.\theta.time$. The time of the corresponding response is $v.\theta.time$ which is at least the time of the call.

4.2 Root conditions

We specify the conditions on $r.state$ where r is a root node of the goal or a subordinate tree. The only condition for the goal tree is

$$r.def = \{\theta\} \text{ and } r.\theta.time \geq 0 \qquad \textbf{(goal root condition)}$$

The first part says that when an expression is called from a main (host language) program, all its formal parameters are replaced by actual parameters values; so they are not part of the root state. Only θ is defined and its associated time is non-negative.

For subordinate trees, consider node u of tree P which has a declaration (x, Q), so Q is subordinate to P. For root q of Q

$$q.state = u.state \qquad \textbf{(subordinate root condition)}$$

That is, the computations of q and u start simultaneously in the same state.

The condition is surprising and, apparently, circular. This is because x may be defined in $u.state$, but it is certainly not available to any node in Q; the purpose of Q is to compute the value of x. Here, we exploit the fact that Q represents the denotation of a well-formed expression. Therefore, no edge in Q accesses $q.x$. The presence or absence of x in $q.state$ (and in the states of all its descendants) is immaterial. Similar remarks apply for multiple declarations at node u; all variables in $u.def$ appear in $q.def$, only some of which would be accessed by a well-formed expression.

4.3 Subordinate tree conditions

Let P be a denotation tree in which node u has a declaration (x, Q); therefore, Q is subordinate to P. The first healthiness condition states that if x is defined (at u), then there is a node of Q of positive size whose θ value is same as that of x. Conversely, x is defined only if there is some such node of Q.

Subordinate assignment condition.

$$(x \in u.def) \;\equiv\; (\exists q : q \in Q,\ \text{size of } q \text{ is positive},\ \neg q.null : q.\theta = u.x)$$

The next condition states that all computations in Q and its subordinates cease once x is assigned value. Define predicate $cease(R, t)$, where R is a tree and t a time, which holds iff (1) $r.\theta.time$ at any node r of R is at most t, and (2) the condition applies recursively to all subordinate trees of R.

Subordinate termination condition.

$$cease(R,t) \;\triangleq\; (\forall r : r \in R : r.\theta.time \leq t) \\ \wedge\; (\forall S : S \text{ subordinate of } R : cease(S,t))$$

$$(x \in u.def) \;\Rightarrow\; cease(Q, u.x.time)$$

The subordinate tree conditions apply to the goal tree as well; its computation may assign the θ value of a node of positive size to the goal variable, and then cease.

A small identity. Call expression g *silent* if all nodes in its tree have size zero. That is, $g = g \gg \mathbf{0}$. We use the healthiness conditions to argue that $(f$ **where** $x :\in\ g)$ has the same execution as $(f \mid g)$ if g is silent (note that $f \mid g$ may not be well-formed if f references x).

In $(f$ **where** $x :\in\ g)$ variable x is declared at the root u of $tree(f)$. From the first condition for subordinate tree, $x \notin u.def$. Therefore, x plays no role in the computation. The condition for ceasing the computation holds vacuously for $tree(g)$. So, the execution of $(f$ **where** $x :\in\ g)$ is identical to starting f and g simultaneously without any constraints on termination (i.e, as $(f \mid g)$).

4.4 Discussion

We show that variables defined in a state may have different associated times, which may also be different from the time associated with θ. To see this, consider $(f$ **where** $x :\in\ g)$. In the operational semantics, the computations of f and g start in the same state, say s. If g returns a value, x is assigned the value and portions of f which are waiting for x may be resumed. This operational notion is captured within our semantics by creating a state s', which is s augmented with the $(magnitude, time)$ of x, and starting *both* f and g in s'. We consider the ramifications of this rule for both f and g.

We have explained in section 4.2 (under *subordinate root condition*), that the apparent circularity of using s' in place of s causes no semantic difficulty for g. A well-formed g does not access x; therefore, state assignments to its nodes are the same with s and s' except that in the latter case each state includes x as a defined variable.

We argue that s' is the appropriate state for starting the computation of f, as well. Consider, for example, $f = (M \mid N(x))$. The evaluation of M proceeds without waiting for x, whereas $N(x)$ has to wait for x. Therefore, M starts in

state s and $N(x)$ in state s'. So, it may seem that the root of $tree(f)$ should have two associated states, s and s'. Fortunately, it is sufficient to associate just s' with the root, because since M does not access x its execution is identical in both s and s', much like the way we argued for g, in the previous paragraph. This argument applies for arbitrary f, as well.

5. Conclusion

The most serious omission from the semantics presented here has been a treatment of site calls and their interactions with the Orc program. A semantics for sites should ideally enable each independent site to be treated separately so that its combination with an Orc program yields the semantics of a more restricted Orc program: interactions with the site are entirely hidden, so that the result can be simply interfaced with the remaining sites.

A second extension to the semantics presented could give a more step-by-step guidance to the implementer on how to execute programs without risking deadlock or making unnecessary site calls.

A third extension would provide more guidance to the programmer on how to establish correctness of programs and of sites. It is particularly important to help the designer of a site to discharge responsibility for nullifying the effect of all site calls except those involved in computing the final result delivered by the Orc program.

Related work. The proposed semantics is greatly influenced by denotations of regular expressions (i.e., Kleene algebra[1]). A regular expression is denoted by a tree labelled by symbols on its edges. Each path in the tree represents a string (the sequence of symbols along its edges) and two trees are identical if they have the same set of paths. Therefore, a regular expression can be denoted by a set of strings only. Orc expressions are also denoted by trees though there are several differences: (1) left distributivity of $\gg$ over $\mid$, i.e., $f \gg (g \mid h) = (f \gg g) \mid (f \gg h)$ does not hold in Orc; so, a tree can not be represented by the labels on its paths only (see figure 1 in page 5), (2) the **where** clause has no counterpart in Kleene algebra; its introduction requires us to attach subordinate trees to the nodes of a tree, and (3) lack of idempotence in Orc forces us to distinguish between $\mathbf{1}$ and $(\mathbf{1} \mid \mathbf{1})$, say, by associating a size with each node.

References

[1] Dexter Kozen. On Kleene algebras and closed semirings. In *Proceedings, Math. Found. of Comput. Sci.*, volume 452 of *Lecture Notes in Computer Science*, pages 26–47. Springer-Verlag, 1990.

[2] Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, May 1999.

[3] Jayadev Misra. Computation orchestration: A basis for wide-area computing. In Manfred Broy, editor, *Proc. of the NATO Advanced Study Institute, Engineering Theories of Software Intensive Systems*, NATO ASI Series, Marktoberdorf, Germany, 2004.