# Implementation Outline for Orc

William Cook, Jayadev Misra

September 27, 2005

## Contents

## 1 Implementation

This section describes an implementation of Orc which is operational. We are now integrating Orc programs with Java to determine if threads can be largely eliminated from Java, relying on Orc to provide all necessary concurrency. This will validate our hypothesis that concurrency can be separated cleanly from computation, and Java programs need only implement the computation aspects, mostly data and object manipulation.

One way to integrate Orc with Java is by using a statement of the form

$z {:\in}\ E(P)$

where $z$ is a variable of the host program, $E$ is the name of an Orc expression and $P$ is a list of actual parameters. All variable parameters in $P$ are variables of the host language program. To execute this statement, start evaluation of a fresh instance of $E$ with actual parameter values substituted for the formal ones, assign the *first* published value to $z$, and then terminate the evaluation of $E$. If evaluation of $E$ produces no value, the statement execution does not terminate.

In the rest of this section, we describe our implementation of Orc which is coded in Java. Inspired by the resemblance of Orc to regular expressions, we represent an Orc expression by a directed acyclic graph (*dag*), much like a finite state machine. The graph is acyclic because, unlike regular expressions, Orc expressions do not have a Kleene-star operator. The nodes of the graph have *instructions*. In order to evaluate an expression, the corresponding graph is traversed by placing *tokens* at its nodes, and executing the associated instructions. Multiple threads of Orc are represented by multiple tokens. The implementation does not use any Java thread facility.

We describe the dag construction in Section 2 and the evaluation procedure using tokens in Section 3.

## 2 Compiler

For each defined expression the compiler builds a dag with one *root* node and one *sink* node. For $x :\in f$ in the main program, the compiler also creates a dag for $f$, called the *goal* dag.

Each node in a dag has an associated instruction. The instructions are: $\mathbf{0}$, $\tau$, *return*, $M(L)$ (where $M$ is a site name and $L$ its parameters), $E(L)$ (where $E$ is a defined expression and $L$ its parameters), *assign*$(x)$, *remove*$(x)$, *where*$(x)$, where $x$ is a variable, and *choke*.

The dag construction is recursive, as shown in Figure 1. To simplify the figure, we show site $M$ and expression $E$ without parameters, though parameters will be associated with these instructions in a dag. Observe that the construction preserves a single root and sink for each dag.

### Dag Finalization

Observe from the construction that each sink node has $\tau$ as its associated instruction. To complete the construction, change the instruction in the sink from $\tau$ to *return* in all but the goal dag; in the goal dag, change it to *choke*. It will be shown in Section 3 that this construction will enable every dag, other than the goal dag, to return each published value to its caller. And the goal dag returns the first value it publishes to the main program, and then terminates the expression evaluation.

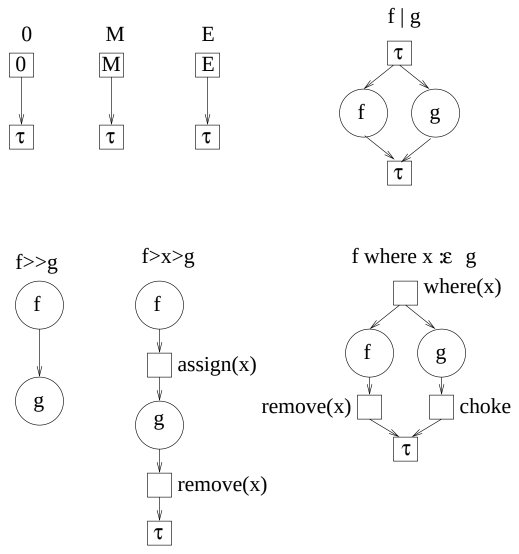


Figure 1: Dag construction

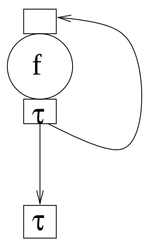
## Dag Optimization ($\tau$ elimination)

Any non-root $\tau$-node $p$ may be eliminated from a dag as follows. Draw an edge from every predecessor of $p$ to every successor of $p$ (every non-root node has a predecessor and every $\tau$-node has a successor after the dag finalization step) and eliminate $p$ and its incident edges.

We may have to retain some $\tau$-nodes because the evaluation procedure of Section 4 assumes that a node whose instruction is (1) $where(x)$ has a left and a right successor, and (2) site or expression call has exactly one successor.

## Compiling Iteration

The dag corresponding to $f^+$ is derived from that of $f$, as shown below. The sink of $f$, which is a $\tau$-node, now has two outgoing edges, one to the root of $f$ and the other to a $\tau$-node.



## 3 Dag Traversal

### 3.1 Overview of Traversal

A *token* resides at some node. A token has several associated fields: *position* identifies the node at which it resides, *context* includes the variable value bindings that can be accessed from this token, *val* is the value computed for this token, *caller* is the id of another token to which val is to be returned at the end of an expression evaluation, and *state*, which determines how the token is to be processed. There is another field, *cell*, which we explain later in this section.

If a token state is *ready*, the corresponding instruction should be executed; if *pending*, the token is waiting to receive a value from an expression or its instruction is a site call for which some parameter does not yet have a value; and if *suspended*, it is awaiting response from a deferred site.

The evaluation starts by placing a *ready* token at the root node of the goal dag, which carries the parameter values in its context.

A ready token $t$ causes execution of the instruction at its node, and then the token is deleted. We give a brief overview of execution, which depends on the kind of instruction.

**0**: do nothing, thus blocking the associated computation

$\tau$: place a copy of $t$ at all successors (from dag construction and finalization, every $\tau$-node has a successor)

*return*: the corresponding node is the sink of a non-goal dag; return $t.val$ to $t$'s caller.

site call: create a copy $u$ of $t$. For a parameter which is a cell, and whose value is undefined, wait-list $u$ at that cell and set $u.state$ to *pending*. When the cell gets defined, $u$ becomes ready. If all parameters have values, the corresponding site is called and token $u$ is moved to its successor node to receive the value. If the call is to an immediate site, $u$ receives a value (which is guaranteed) and is ready. If the call is to a deferred site, $u$ is suspended awaiting a response.

expression call: place token $v$, a copy of $t$, on $t$'s successor node (this node has exactly one successor, by dag construction) which will be used to receive the value from expression evaluation. Token $v$ is pending. To receive a value from the expression, a copy of $v$, $v'$, is created; it receives the value and becomes ready; $v$ remains pending forever to keep receiving values. Also, place a ready token $u$ at the root of $E$'s dag; context of $u$ is derived from the parameters of the call and $u$'s caller is set to $v$.

*assign*$(x)$: place token $u$, a copy of $t$, on the successor node (this node has exactly one successor, by dag construction), and add the pair $(x, t.val)$ to $u$'s context stack.

*remove*$(x)$: place token $u$, a copy of $t$, on the successor node (this node has exactly one successor, by dag construction), and remove the first pair that binds $x$ from $u$'s context stack.

**Processing** *where* **and** *choke***:** In processing $f$ **where** $x :\in g$, some tokens in $f$ may remain pending as long as $x$ does not acquire a value. And, once $x$ has a value, all tokens in $g$ have to be deleted, to terminate further computation. In order to do this, we assign a *cell* to $x$ where the value of $x$ will be stored, and each token carries as a field the name of the cell for which it was created (i.e., for evaluating $g$ and computing the value of $x$). Name of a cell is a proxy for a variable like $x$. The very first token, which is placed at the root of the goal dag, belongs to *RootCell*, a special cell which is created to return a single value to the main program.

A ready token $t$ is processed at a *where*$(x)$ node as follows. Create a new cell, $c$. Place a copy of $t$ at the left successor (corresponding to the root of $f$ in $f$ **where** $x :\in g$), and add $(x, c)$ to its context, indicating that the value of $x$ is to be retrieved indirectly through $c$; the value part of $c$, $c.val$, is initially undefined. Place a copy of $t$ on the right successor (corresponding to the root of $g$ in $f$ **where** $x :\in g$), and set $c$ as its cell.

A ready token $t$ is processed at a *choke* node as follows. Let $c$ be $t.cell$. Set $c.val$ to $t.val$, and set $u.state$ to *ready* for all tokens $u$ in *c.list*. Next, all tokens whose cell is $c$ is deleted. Additionally, any token created out of a token whose cell is $c$ is also deleted, because it is part of the computation of $g$. Note that the cells form a tree; the root of the tree is *RootCell* and if cell $c$ is created by token $t$ (by executing a *where* instruction), then $c$ is a child of $t.cell$. The choke instruction deletes all tokens whose cell is $c$ or a descendant of $c$.

## 4 Evaluation Procedure

### 4.1 Run time structures

The two main data structures are token and cell.

- A cell has the following fields:
  1. *parent*: the parent of the cell; parent of *RootCell* is $\perp$.
  2. *val*: the value of (the variable corresponding to) the cell; $\perp$ if undefined.
  3. *list*: the set of tokens which are waiting for this cell to have value.
- A token has the following fields:
  1. *position*: the node at which the token resides.
  2. *context*: bindings of variables to values or pointers to cells.
  3. *val*: value computed for this token.

4. *caller*: token to which any published value is to be sent.
5. *state*: *ready*, *pending* or *suspended*.
6. *cell*: this token's cell.

## 4.2 Initialization

1. Initialize the cell tree: create *RootCell*.
2. Create a ready token $s$ and place it on the root node of the goal dag.

| | |
|---|---|
| *s.position*: | root node of the goal dag |
| *s.context*: | the bindings of the global variables and values |
| *s.cell*: | *RootCell* |
| *s.state*: | *ready* |

## 4.3 The Main Loop of the Evaluation Procedure

```
$RootCell.val \neq \bot$      $\longrightarrow$  return $RootCell.val$ to main; terminate.
$t.ready$                     $\longrightarrow$  process $t$ using algorithm of Figure 2; delete $t$
$t.suspended \wedge \neg(\exists s :: s.ready)$
                              $\longrightarrow$  on receiving response $r$: $t.val := r$; $t.state := ready$
else                          $\longrightarrow$  “No value will be published”
```

**else clause** The else clause in the main loop is equivalent to $(\forall t :: t.pending)$. This can arise in processing an expression like $M(x)$ **where** $x :\in \mathbf{0}$, or just $\mathbf{0}$. A token is pending if it is waiting for a value from an expression, or its instruction is a site call which has a parameter without value. We can assert that a pending token stays pending until some non-pending token is processed. Therefore, $(\forall t :: t.pending)$ is stable. If $RootCell.val = \bot$, it will remain $\bot$ forever.

This situation should be contrasted with non-termination. Then there is a suspended token. The token may receive a value (from a deferred site); so, a computation can not be terminated as long as there is a suspended token and $RootCell.val = \bot$.

**Round-based Execution**

## 4.4 The invariant for the site call loop

We repeat a portion of code which deals with site calls.

```
$M(L)$
    $u := copy(t)$; $i := 0$ {the parameters in $L$ are $L_1 \cdots L_n$, $n$ possibly 0}
    while $u.state = ready \wedge i < n$ do {checked $i$ parameters already}
```

```
        $i := i + 1$
        **if** $L_i$ corrsponds to cell $c$ **then**
            **if** $c.val = \perp$ **then** add $u$ to $c.list$; $u.state := pending$ **fi**
        **fi**

    **od**
```

While it is easy to see the correctness of this code, we propose a loop invariant in anticipation of the proof required for concurrency.

$0 \leq i \leq n$,
$(u.state = ready \vee u.state = pending)$,
$u.state = ready \;\Rightarrow$
  values of the first $i$ parameters are available and $u \notin c.list$, for any $c$,
$u.state = pending \;\Rightarrow$
  $i > 0$ and
  values of the first $i - 1$ parameters are available and
  $u \in c.list$, where $L_i$ corresponds to $c$.

For sequential execution, on termination of the loop $u.state = ready \;\Rightarrow i = n$. And, from the invariant, values of all parameters are then available. Therefore, the condition $u.state = ready \;\Rightarrow i = n$ may be written as $u.state = ready$. The more elaborate condition is included in anticipation of concurrent execution.

## 5 Optimization

### 5.1 Saving contexts

It would too expensive for each token to carry its context. Since the contexts of related tokens differ minimally, we devise a scheme to store all contexts in a table and have a token point to the relevant portions of this table.

### 5.2 Effect of token processing

Each node of the cell tree has a group of tokens.Processing a token has the following effect.

1. create tokens at its cell: Note that $t.caller.cell = t.cell$. Therefore, a reurn instruction always creates a token of the same cell.
2. create child cells and tokens there: at a where instruction.
3. Modify fields of other tokens: at a choke node, where $u$ is a descendant of the parent of this cell.

```
  0  skip {No token is placed on the successor of a 0 node.}

  $\tau$  place copies of $t$ on all successor nodes

$return$  {Publish the value}
    $u := copy(t.caller)$; $u.val := t.val$; $u.state := ready$

$M(L)$
    $u := copy(t)$; $i := 0$ {the parameters in $L$ are $L_1 \cdots L_n$, $n$ possibly 0}
    while  $u.state = ready \land i < n$  do {checked $i$ parameters already}
        $i := i + 1$
        if  $L_i$ corrsponds to cell $c$  then
            if  $c.val = \bot$  then  add $u$ to $c.list$; $u.state := pending$  fi
        fi
    od
    if  $u.state = ready \ \land \ i = n$ {all parameters have values}  then
        {move $u$ to the next position} $u.position := u.position.next$; call $M$;
        if  $M$ is an immediate site  then
            receive response $r$; $u.val := r$ {$u.state = ready$}
        else  $u.state := suspended$
        fi
    fi

$E(L)$
    $v := copy(t)$; $v.position := t.position.next$; $v.state := pending$
    $u := newtoken()$
        $u.position :=$ root node of $E$
        { create context from $t$'s context and the call parameters }
        $u.context[E.formals[i]] := t.context(L[i])$;
        $u.caller = v$; $u.cell = t.cell$; $u.val = \bot$; $u.state := ready$

$assign(x)$
    $u := copy(t)$; $u.position := t.position.next$; $u.context := t.context +$
    $(x, t.val)$

$remove(x)$
    $u := copy(t)$; $u.position := t.position.next$; $u.context := t.context - (x, -)$

$where(x)$
    $c := newcell()$; $c.val := \bot$; $c.parent := t.cell$; $c.list := \{\}$
    { create $u$ on the left edge }
    $u := copy(t)$; $u.position := t.position.left$; $u.context := t.context + (x, c)$
    { create $v$ on the right edge }
    $v := copy(t)$; $v.position := t.position.right$; $v.cell := c$

$choke$  {No token is placed on any successor of a choke node.}
    $c := t.cell$;
    delete every token $v$, $v \neq t$, whose cell is a descendant of $c$ (including $c$);

    {$c.val = \bot$} $c.val := t.val$; {Henceforth, $c.list$ does not grow}
    for  every token $u$ in $c.list$  do  $u.state := ready$  od
```

Figure 2: Processing $t.ready$

### 5.3 Deleting tokens

We avoid explicitly deleting tokens at a *choke* node as follows. We associate a field *live* with a cell. A cell is *live* if the computation corresponding to that cell is ongoing, i.e., all its ancestors (including itself) have undefined values.

$$t.live \;\underline{\Delta}\; t.cell.live$$
$$c.live \;\underline{\Delta}\; (\forall d :\; d \text{ an ancestor of } c :\; d.val = \bot)$$

Therefore,
$$c.live \;\equiv\; c.val = \bot \;\wedge\; (c = RootCell \vee c.parent.live)$$

A token is *live* iff its corresponding cell is live. A dead (i.e., non-live) token is never processed, and should be garbage-collected. Only live tokens are processed.

Initially, *RootCell.live* holds, because $RootCell.val = \bot$. Whenever a new cell $c$ is created in a where instruction, set $c.live = true$ and $c.val = \bot$.

# 6 Concurrent and Distributed Execution

## 6.1 Concurrent Execution

Ready tokens may be processed concurrently.

1. Only live tokens are processed. A token's liveness is checked by looking at *val* fields of all ancestors, not by storing a bit in *c.live*. Tokens are not explicitly deleted.

2. Garbage collection: Delete all tokens which are dead. The cells need to be kept for their values.

3. Main Loop:

```
$RootCell.val \neq \bot$             $\longrightarrow$  return $RootCell.val$ to main; terminate.
$t.live \wedge t.ready$              $\longrightarrow$  process $t$ using algorithm of Figure 2; delete $t$
$t.live \wedge t.suspended$  $\wedge\; \neg(\exists s ::\; s.live \wedge s.ready)$
                                     $\longrightarrow$  on receiving response $r$: $t.val := r$; $t.state := ready$
**else**                             $\longrightarrow$  "No value will be published"
```

4. Changes in token Processing:

   (a) In processing a site call, the following code

   **if** $c.val = \bot$ **then** add $u$ to $c.list$; $u.state := pending$ **fi**

   is executed by first read-locking cell $c$. Therefore, other tokens can execute the same piece of code simultaneously. We assume that add $u$ to $c.list$ is an atomic operation.

(b) In processing a choke instruction, $c.val := t.val$ is executed by write-locking cell $c$. Therefore, no other token can execute a site call to check $c.val$ or add the token to $c.list$ simultaneously.

We have the earlier invariant:

$0 \leq i \leq n$,
$(u.state = ready \vee u.state = pending)$,
$u.state = ready \;\Rightarrow$
  values of the first $i$ parameters are available and $u \notin c.list$, for any $c$,
$u.state = pending \;\Rightarrow$
  $i > 0$ and
  values of the first $i-1$ parameters are available and
  $u \in c.list$, where $L_i$ corresponds to $c$.

It is now possible, unlike the sequential case, for $u.state = ready \wedge i < n$ to hold following completion of the loop in the site call (because, a concurrent choke instruction may set $u.state = ready$ when $i < n$).

The correctness argument is based on the observation that sharing occurs only in updating $c.val$, $c.list$ and $u.state$ at a site call and choke. And, the interference is eliminated by locking. Adding token $u$ to $c.list$ or setting $u.state$ to pending, are executed while $c.val$ remains $\perp$. Formalyy, we have to show that the given invariant holds during concurrent execution.

No token is added to $c.list$ once $c.val$ is assigned a value. Also, a token $u$ is in $c.list$ only if its state is pending. Therefore, there is no interference in setting the state of a token.

5. Deadlock Avoidance: deadlock is possible only if a token holds a lock forever. This is impossible. For both read-lock and write-lock the corresponding unlock instruction is guaranteed to beexecuted (because the code does not ask for another lock or wait for any response).

## 6.2 Distributed Execution

A concurrent execution in which an unpunctual subexpression is run concurrently is a distributed execution. The implementation for the main expression is a *master* and the dag for the unpunctual subexpression is implemented at a *slave*. The dag at the slave has a *return* instruction at its bottom. The interaction between the master and the slave is as follows.

1. The master sends a token to be placed at the root of the slave dag.

2. In executing a *return* instruction, the slave sends a value to the master to be put in the val field of some pending token.

3. A $where(x)$ and the corresponding *choke* instruction both belong to the master or the slave.

```
$\mathbf{0}$ skip {No token is placed on the successor of a $\mathbf{0}$ node.}

$\tau$ place copies of $t$ on all successor nodes

$return$ {Publish the value}
    $u := copy(t.caller)$; $u.val := t.val$; $u.state := ready$

$M(L)$
    $u := copy(t)$; $i := 0$ {the parameters in $L$ are $L_1 \cdots L_n$, $n$ possibly 0}
    while $u.state = ready \wedge i < n$ do
    $i := i + 1$
        if $L_i$ corrsponds to cell $c$ then
            $readLock(c)$
            if $c.val = \bot$ then add $u$ to $c.list$; $u.state := pending$ fi
            $readUnlock(c)$
        fi
    od
    if $u.state = ready \wedge i = n$ {all parameters have values} then
        {move $u$ to the next position} $u.position := u.position.next$; call $M$;
        if $M$ is an immediate site then
            receive response $r$; $u.val := r$ {$u.state = ready$}
        else $u.state := suspended$
        fi
    fi

$E(L)$
    $v := copy(t)$; $v.position := t.position.next$; $v.state := pending$
    $u := newtoken()$
        $u.position :=$ root node of $E$
        { create context from $t$'s context and the call parameters }
        $u.context[E.formals[i]] := t.context(L[i])$;
        $u.caller = v$; $u.cell = t.cell$; $u.val = \bot$; $u.state := ready$

$assign(x)$
    $u := copy(t)$; $u.position := t.position.next$; $u.context := t.context +$
    $(x, t.val)$

$remove(x)$
    $u := copy(t)$; $u.position := t.position.next$; $u.context := t.context - (x, -)$

$where(x)$
    $c := newcell()$; $c.val := \bot$; $c.parent := t.cell$; $c.list := \{\}$
    { create $u$ on the left edge }
    $u := copy(t)$; $u.position := t.position.left$; $u.context := t.context + (x, c)$
    { create $v$ on the right edge }
    $v := copy(t)$; $v.position := t.position.right$; $v.cell := c$

$choke$ {No token is placed on any successor of a choke node.}
    $c := t.cell$; {$c.val = \bot$} $writeLock(c)$; $c.val := t.val$; $writeUnlock(c)$;
    for every token $u$ in $c.list$ do $u.state := ready$ od
```

Figure 3: Concurrent Execution

4. The slave need not request $read-lock(c)$ from the master even if $c$ belongs to the master. The slave maintains the portion of $c.list$ for its tokens.

5. A *choke* instruction at the slave affects only a cell at the slave; it entails no communication with the master.

6. A *choke* instruction at the master is executed as follows. The master does not explicitly acquire the write-lock. It sends $c.val$ to slave (provided the slave needs $c.val$) and wait for an ack.

   The slave deletes all tokens from its portion of $c.list$, and caches $c.val$ for future use.

7. A token at the slave chcks its liveness only up to its root color (there could be a race condition here).

8. It is possible that lock are not needed at all. Start from the sequential version and go to the didtributed version.