Implementation of Orc

William Cook and Jayadev Misra

Department of Computer Science
University of Texas at Austin

Email: {cook, misra}@cs.utexas.edu
web: http://www.cs.utexas.edu/users/psp
Status of Implementation

- Implementation coded in Java.
- An Orc program can call Java programs as sites.
- A Java program can call an Orc program.

Another implementation by Galen Menzel using Concurrent Haskell.
Java Calling Orc

Include in the main (Java) program

\[ z \in E(L) \]

where

- \( z \) is a variable of the main program,
- \( E \) an Orc expression,
- \( L \) a list of actual parameters,
  - constants and variables of the main program.

The effect is: assign to \( z \) the first value published by \( E \) and terminate.
Implementation Using the Semantic Rules

\[
\begin{align*}
\text{(SYM1)} & \quad f \overset{l}{\leftrightarrow} f' \\
& \quad f \mid g \overset{l}{\leftrightarrow} f' \mid g \\
\text{(SEQ1V)} & \quad f \overset{\text{tc}}{\leftrightarrow} f' \\
& \quad f \gg g \overset{\tau}{\leftrightarrow} (f' \gg g) \mid [c/x]g
\end{align*}
\]

The expression structure has to change.

\[
\begin{align*}
\text{(DEF)} & \quad [[E(q) \blacktriangle f]] \in D \\
& \quad E(p) \overset{\tau}{\leftrightarrow} [p/q]f
\end{align*}
\]

Each expression has to be instantiated whenever it is called.
A simpler strategy

- Compile a fixed structure for each expression.
  - Compile each expression to a directed acyclic graph (dag).
  - Each node of the dag has an instruction.

- Runtime Dag Traversal: Place tokens at dag nodes.
  In each step,
  - Pick an appropriate token.
  - Execute the corresponding instruction, which may
    * make site/expression call
    * create new tokens
    * publish a value
For each defined expression and the goal expression build a dag.

Each dag has a root and a sink node.

Each node has an instruction:

- 0 for expression 0
- \( \tau \) for silent transition
- return to publish a value
- \( M(L) \) site call
- \( E(L) \) expression call
- assign(\( x \)) assign to variable \( x \)
- where(\( x \)) for starting a where expression
- choke for ending a where expression
Recursive Construction of Dag

\[
\begin{align*}
0 & \quad M & \quad E \\
\tau & \quad \tau & \quad \tau \\
\tau & \quad \tau & \quad \tau \\
\end{align*}
\]

\[
\begin{align*}
\text{choke} & \quad \text{assign}(x) \\
\end{align*}
\]

\[
\begin{align*}
f > \tau > g \\
f & \quad g \\
\end{align*}
\]

\[
\begin{align*}
f \mid g \\
f & \quad g \\
\tau \\
\end{align*}
\]
Notes on Dag construction

- There is a unique root and sink for each dag.
- The instruction at each sink is $\tau$. 
Dag Finalization

Change the instruction at each sink, from $\tau$ to:

- \textit{choke}, if this is the goal dag (i.e., for expression in the main program)
- \textit{return}, for all other dags.

Hence,
a sink does not have a $\tau$ instruction, i.e.,
Every $\tau$-node has a successor.
Construction of Example Dag

Figure 1: \((M \mid N(x) \text{ where } x : \in S \geq 0) > y > G(y)\)
Dag Optimization (τ-node Elimination)

Eliminate any non-root τ-node:

\[ A \rightarrow \tau \rightarrow B \]
\[ C \rightarrow \tau \rightarrow D \]

Restriction

- A where node has a left and a right successor.
- A site/expression call node has exactly one successor.
Reconstruction of Example Dag

Figure 2: \( (M \mid N(x) \text{ where } x \in S \gg 0) > y > G(y) \)
Dag Traversal: $((M \mid N) > x > (R \mid S(x)))$
\( \tau \bigcirc \) : ready token \quad \bigcirc \ : suspended token

1. \( M \quad \tau \bigcirc \) \\
   \( \quad \bigcirc \) assign(x) \\
   \( R \) \\
   \( \bigcirc \) S(x) \\
   \( \bigcirc \) : suspended token

2. \( M \quad \bigcirc \) assign(x) \\
   \( \tau \) \\
   \( R \) \\
   \( \bigcirc \) S(x)

3. \( M \) \\
   \( \tau \) \\
   \( R \) \\
   \( \bigcirc \) S(x) \\
   \( \bigcirc \) : suspended token

4. \( \bigcirc \) assign(x) \\
   \( \tau \) \\
   \( R \) \\
   \( \bigcirc \) S(x) \\
   \( \bigcirc \) : suspended token

5. \( \bigcirc \) assign(x) \\
   \( \tau \) \\
   \( R \) \\
   \( \bigcirc \) S(x) \\
   \( \bigcirc \) : suspended token

6. \( M \) \\
   \( \tau \) \\
   \( R \) \\
   \( \bigcirc \) S(x) \\
   \( \bigcirc \) : suspended token
Fields of a token

- **position**: the node in the dag.
- **context**: values of variables (such as $x$ in the example)
- **val**: token’s value, which may be returned to the caller.
- **state**: ready, pending, or suspended.
Initialization

Given \( z := E(p, 3) \) in the Main program and goal dag \( E(x, y) \)

- Create token \( t \) where \( t.context = \{(x, p’s \ value), (y, 3)\} \).
- \( t.val = \bot \).
- \( t.state = ready \).
- Put \( t \) on the root node of the goal dag.
Process token \( t \)’s instruction

- **0**: skip.
- **\( \tau \)**: put copies of \( t \) at all successor nodes.
- **Site call \( M(x) \)**:
  - call \( M \) with parameter value \( x \) from \( t.context \).
  - put suspended copy of \( t \) at the successor node.
- **\( assign(x) \)**:
  - add \( (x, t.val) \) to \( t.context \).
  - put copies of \( t \) at all successor nodes.
- **\( choke \)**:
  - return \( t.val \) to the caller. (This will be generalized.)
  - Terminate this computation.

Delete \( t \) after processing.
Processing expression call: caller’s dag

Initially
Expression Call E(p)
On receiving value

- For ready token $t$ at $E(p)$:
  - put $v$, a pending copy of $t$, at $M$.
  - Delete $t$.

- On receiving value from $E$:
  - create ready copy $v'$ of $v$ to get the value.
  - $v$ remains pending, to receive more values.
Processing expression call $E(x)$: callee’s dag

- when token $t$ at caller dag calls $E(p)$: put token $u$ at $E$’s root.
  - $u$ is ready.
  - $u.context$ has $(x, p$’s value from $t.context)$.
  - $u.caller := v$ — $caller$ is a field of a token.

- To process token $t$ at the sink of the dag, with instruction $return$:
  - $v := t.caller$
  - $v.val := t.val$; $v.state := ready$
  - delete $t$
Example: *Metronome*

\[
\tau
\]

Signal \quad Rtimer(1)

\[
\text{Metronome}
\]

\[
\text{return}
\]

Figure 3: *Metronome \ △ Signal \ | \ Rtimer(1) \ → \ Metronome*
0': on the left, always ready  
0'': on the right, ready  
i'': on the right, suspended
Token structure in Metronome

- $0'$ and $0''$ return signals to Main.
- $i'$ and $i''$, $i > 0$, return signals to $(i - 1)''$.
- $i'''$ is permanently pending; copy $i''$ created when it receives a signal.
Summary (so far)

- Compile dag for each expression.

- A token has: position, context, val, caller, state.

- Put a ready token with parameter values as context at the root of goal dag.

- Process any ready token, with instruction:
  
  $0, \tau, \text{Site/Expr call, assign}(x), \text{return}, \text{choke}$
where expression

Figure 4: $f \text{ where } x: \in g$

- Compute $f$ as far as possible. Call to $M(x)$ may wait.
- Compute $g$: At $\text{choke}$
  - assign value to $x$.
  - terminate $g$
For ready token $t$ at $\text{where}(x)$

- Create ready tokens $u$ and $v$ at left and right successors.
- Create cell $c$ where the value of $x$ will be stored. $c.val := \bot$
- Add $(x, c)$ to $u.context$. 
Site/Expr call in the left subgraph

• For ready token $t$ at site call $M(x)$ where $(x, c)$ is in $t.context$:
  
  if $c.val \neq \bot$ then call $M(c.val)$; put copy $u$ of $t$ at successor.

  – immediate site: receive response $r$; $u.val := r$; $u.state := ready$
  – deferred site: $u.state := suspended$

  if $c.val = \bot$ then $t$ is pending waiting for $c$.

• For token $t$ at expr call $E(x)$: proceed as before

delete $t$. 
For token $t$ at $where(x)$, contd.

Given that cell $c$ is created at $where(x)$:

- All tokens in $g$ are killed at $choke$.
- Identify all such tokens by cell $c$; kill all tokens of of cell $c$.
- $v.cell := c$ — $cell$ is a new field of a token.
Processing \textit{choke}

For token $t$ at \textit{choke}:

$c := t.cell$
$c.val := t.val$

now no token waits for $c$;
any token waiting only for $c$ is made ready;
delete all tokens of cell $c$
Cell within cell:  \[ h \text{ where } y : \in (f \text{ where } x : \in g) \]

- Suppose \( y \) is assigned at \( \text{choke2} \) before \( x \) is assigned.
- Need to kill \( f \text{ where } x : \in g \)'s computation, i.e., both \( f \) and \( g \).
- Tokens in \( f \) and \( g \) have different cells.
- In processing token \( t \) at \( \text{where}(x) \):
  create cell \( c \);
  \( c\.parent := t\.cell \)

- In processing token \( s \) at \( \text{choke} \):
  kill all tokens of \( s\.cell \) and descendant cells.
RootCell

- The initial token has cell \textit{RootCell}.
- The cells form a tree with root \textit{RootCell}.
Summary of Runtime structure

- Fields of a token: position, context, val, caller, state, cell.

- Fields of a cell: val, parent, waitList.
Overall algorithm

\[\text{RootCell.val} \neq \bot \quad \rightarrow \quad \text{return RootCell.val to main; terminate.}\]
\[t.\text{ready} \quad \rightarrow \quad \text{process instruction at } t; \text{ delete } t\]
\[t.\text{suspended} \land \neg (\exists s :: s.\text{ready}) \quad \rightarrow \quad \text{on receiving response } r:\]
\[\quad t.\text{val} := r; t.\text{state} := \text{ready}\]
\[\text{else} \quad \rightarrow \quad \text{“No value will be published”}\]

- **Round-based Execution**: Response from a deferred site is processed only if there is no ready token.
- **else** is same as \((\forall t :: t.\text{pending})\). There may be no token at all (for 0).
- **else** is executed for \(M(x) \text{ where } x \in 0\)
- **The algorithm may not terminate**: there are suspended tokens, but no deferred site responds.