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

$$\frac{f \xhookrightarrow{l} f'}{f \mid g \xhookrightarrow{l} f' \mid g} \qquad \text{(SYM1)}$$

$$\frac{f \xhookrightarrow{\dagger c} f'}{f \;>x>\; g \xhookrightarrow{\tau} (f' \;>x>\; g) \mid [c/x]g} \qquad \text{(SEQ1V)}$$

The expression structure has to change.

$$\frac{[\![\, E(q) \;\underline{\Delta}\; f \,]\!] \in D}{E(p) \xhookrightarrow{\tau} [p/q]f} \qquad \text{(DEF)}$$

Each expression has to be instantiated whenever it is called.

A simpler strategy

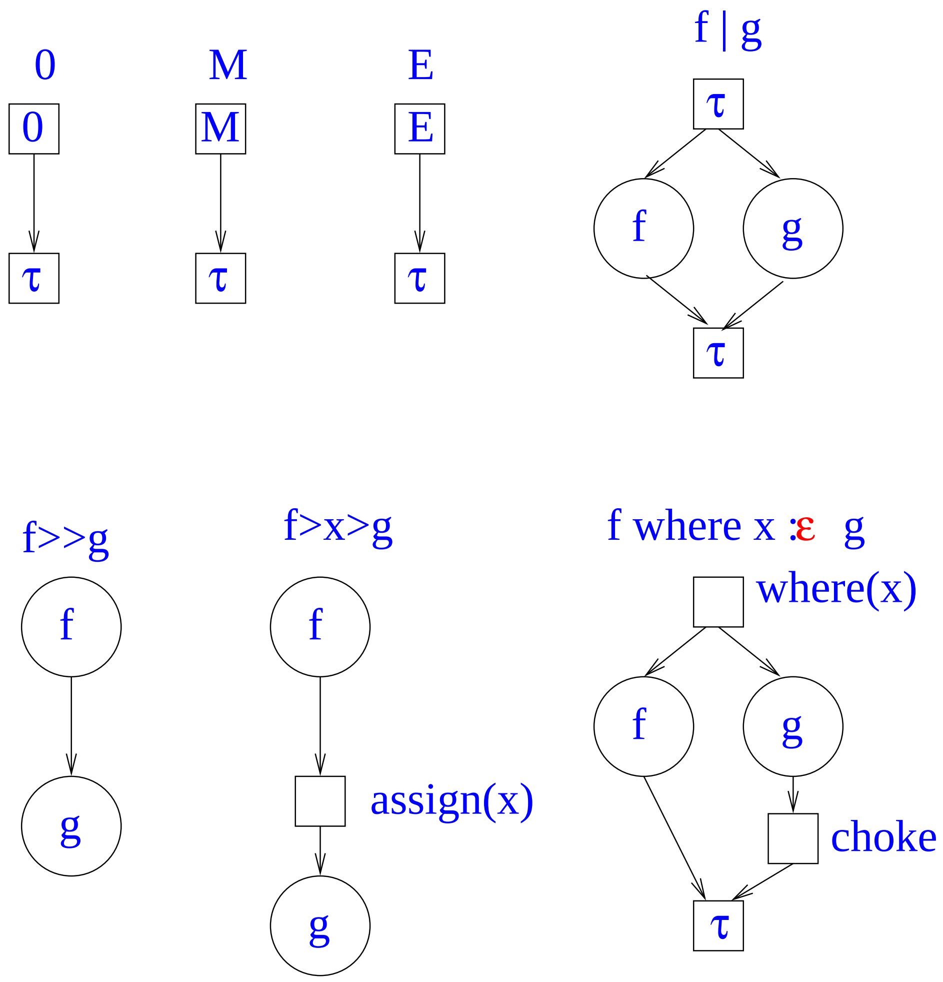
- Compile a fixed structure for each expression.
  - Compile each expression to a directed acyclic graph (dag).
  - Each node of the dag has an instruction.
- Runtime Dag Traversal: Place tokens at dag nodes.
  In each step,
  - Pick an appropriate token.
  - Execute the corresponding instruction, which may
    - make site/expression call
    - create new tokens
    - publish a value

Compiler

- 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
τ	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



Notes on Dag construction

- There is a unique root and sink for each dag.
- The instruction at each sink is τ.

Dag Finalization

Change the instruction at each sink, from τ to:

$choke$, if this is the goal dag (i.e., for expression in the main program)

$return$, for all other dags.

Hence,
a sink does not have a τ instruction, i.e.,
Every τ-node has a successor.

Construction of Example Dag

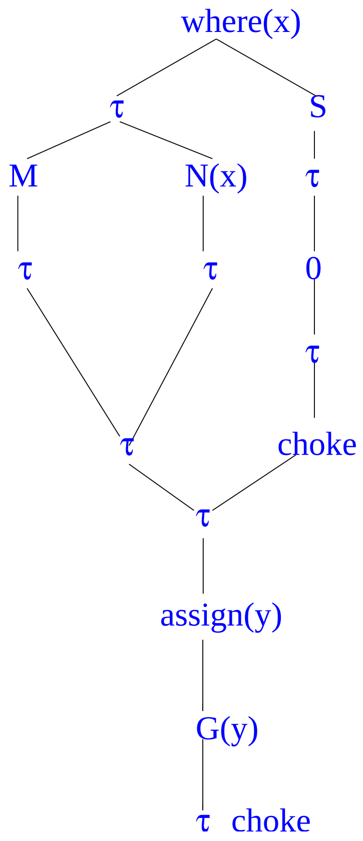
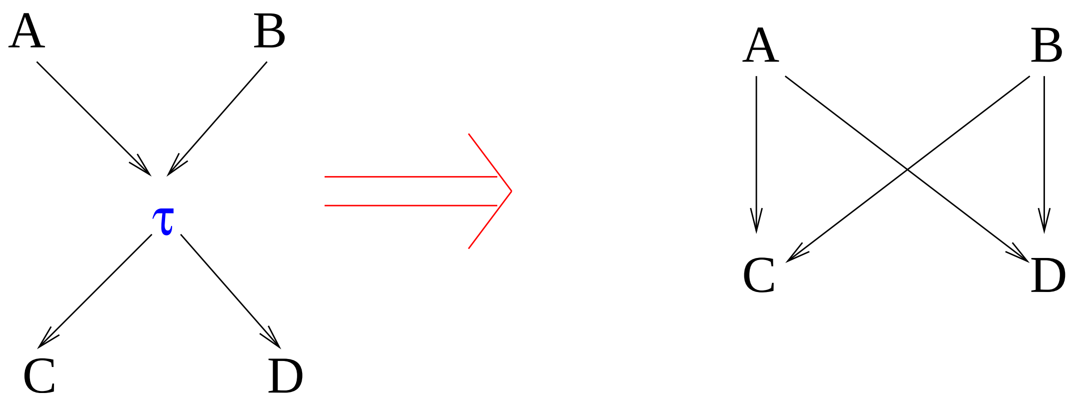


Figure 1: $(M \mid N(x)\ \text{where}\ x{:}\in S \gg 0)\ >y>\ G(y)$

Dag Optimization (τ -node Elimination)

Eliminate any non-root τ -node:



Restriction

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

Reconstruction of Example Dag

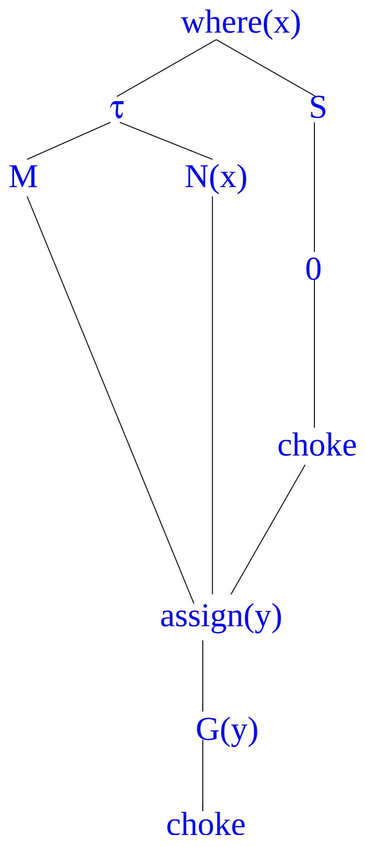
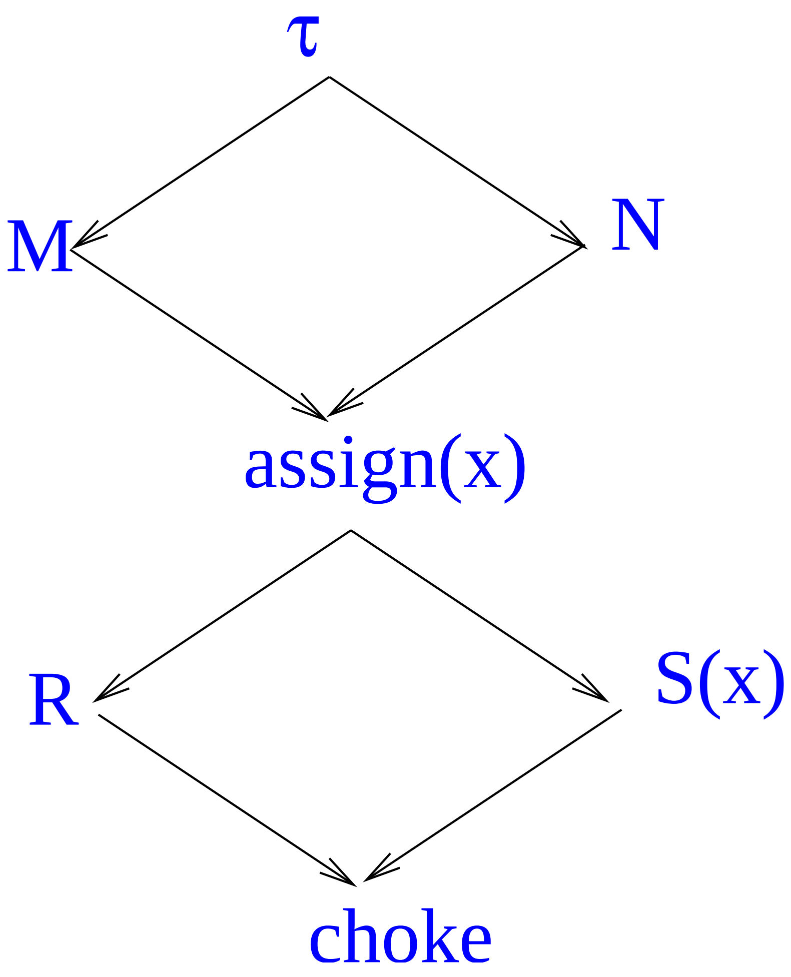
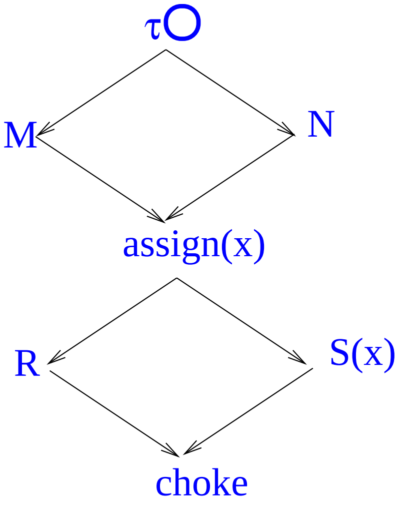


Figure 2: $(M \mid N(x) \text{ where } x :\in S \gg 0) \; > y > \; G(y)$

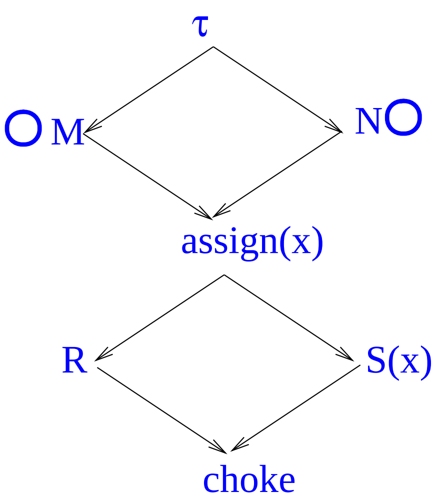
Dag Traversal: $(M \mid N) \;{>}x{>}\; (R \mid S(x))$



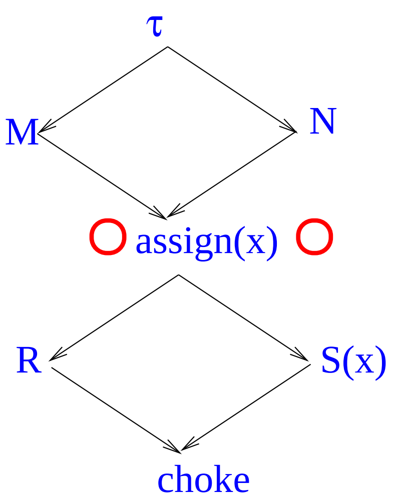
○ : ready token ○ : suspended token



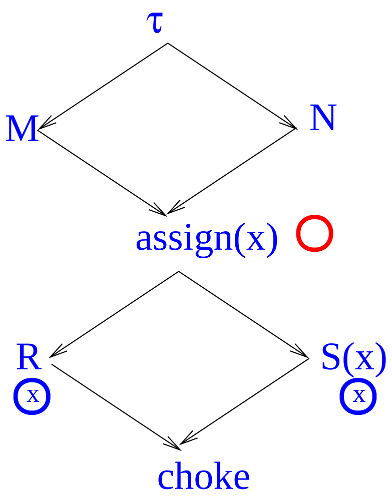
(1)



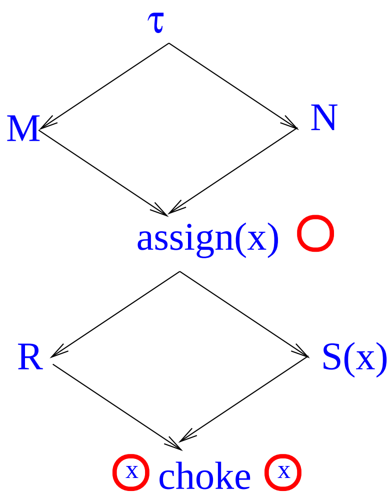
(2)



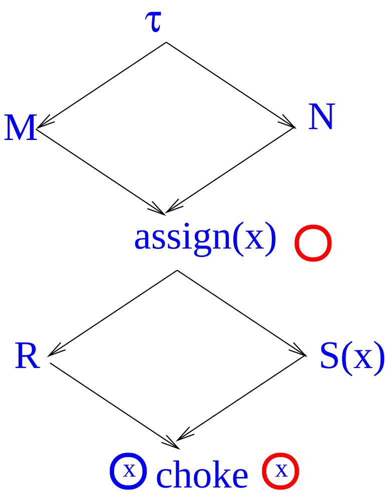
(3)



(4)



(5)



(6)

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 $E(x, y)$

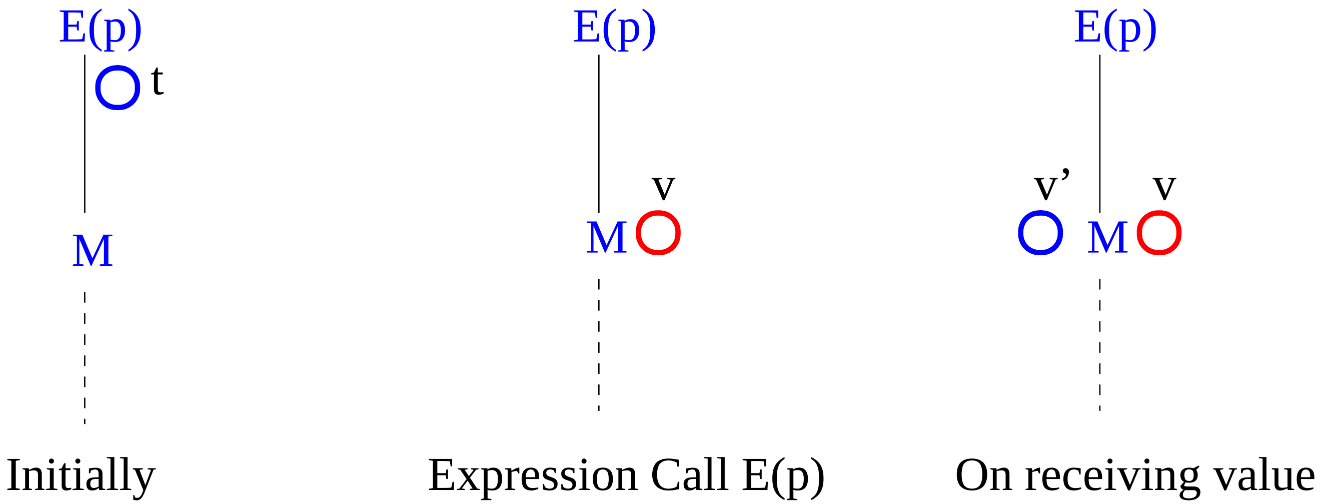
- Create token t where $t.context = \{(x, p\text{'s value}), (y, 3)\}$.
- $t.val = \perp$.
- $t.state = ready$.
- Put t on the root node of the goal dag.

Process token t's instruction

- 0: skip.
- τ: 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



- 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*

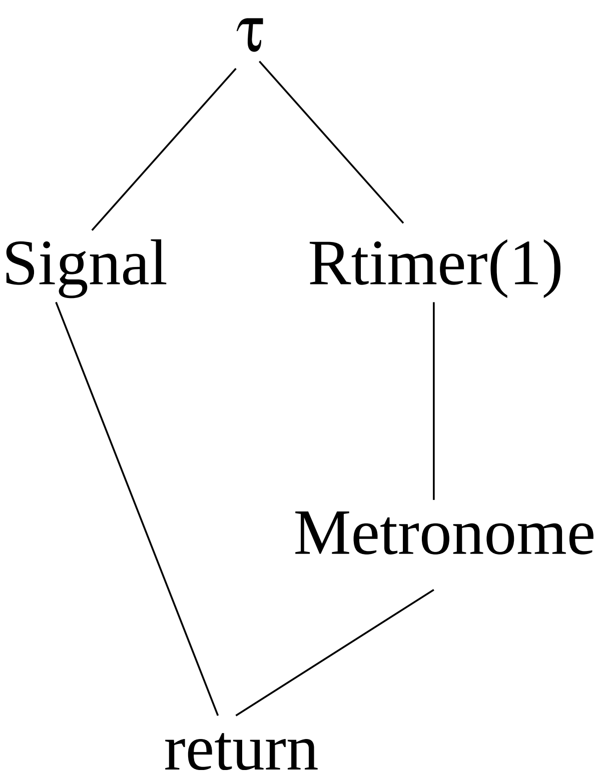
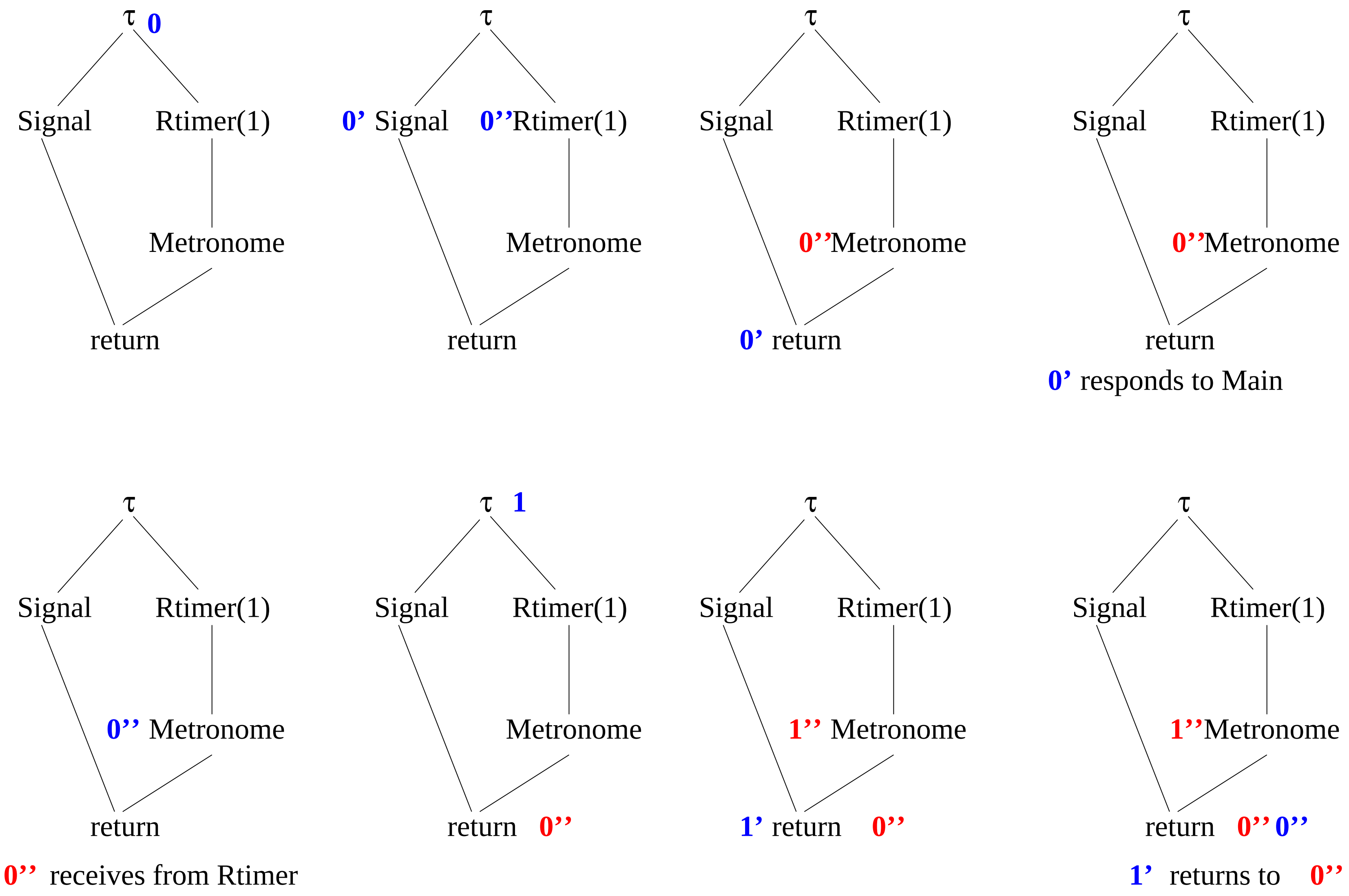


Figure 3: $Metronome \;\underline{\Delta}\; Signal \mid Rtimer(1) \gg Metronome$



i': on the left, always ready i'': 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, τ, Site/Expr call, $assign(x)$, $return$, $choke$

where expression

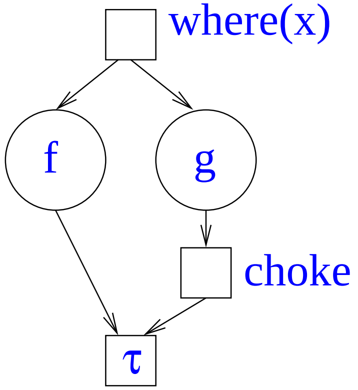
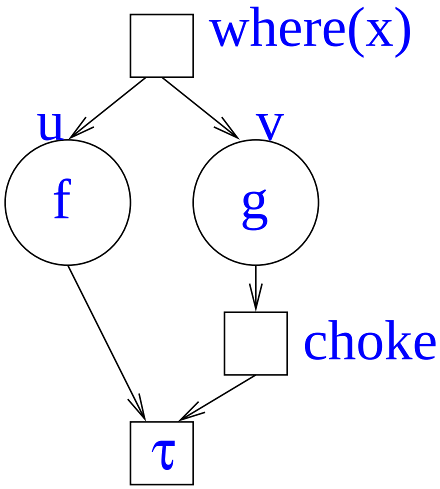


Figure 4: f where $x{:}{\in}\ g$

- Compute f as far as possible. Call to $M(x)$ may wait.
- Compute g: At $choke$
  - assign value to x.
  - terminate g

For ready token t at $where(x)$



- Create ready tokens u and v at left and right successors.
- Create cell c where the value of x will be stored. $c.val := \perp$
- 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.

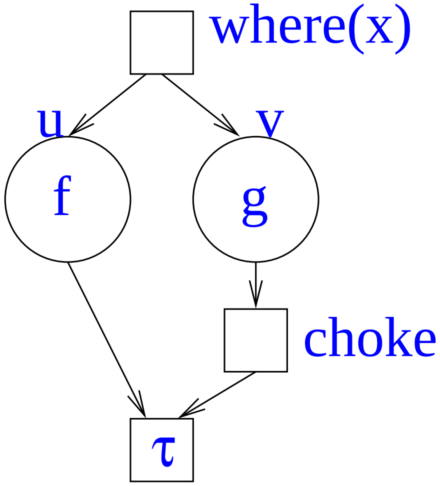
  - immediate site: receive response r; $u.val := r$; $u.state := ready$
  - deferred site: $u.state := suspended$

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

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

delete t.

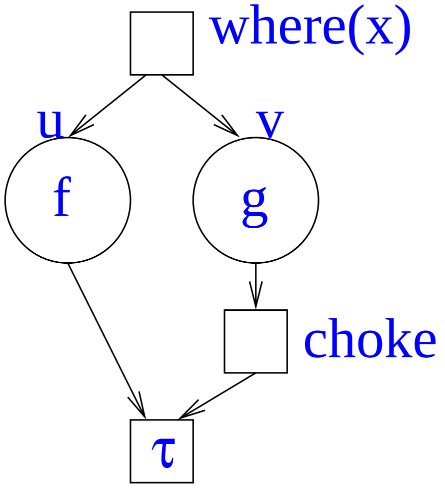
For token t at $where(x)$, contd.



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.

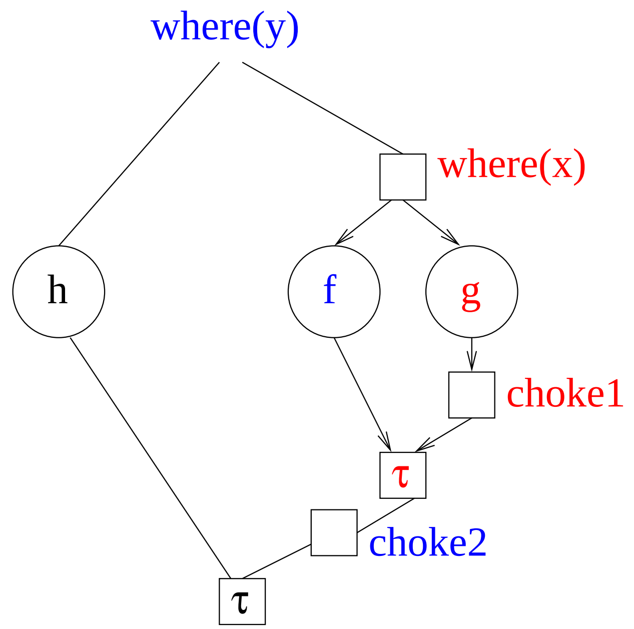
Processing *choke*



For token t at $choke$:

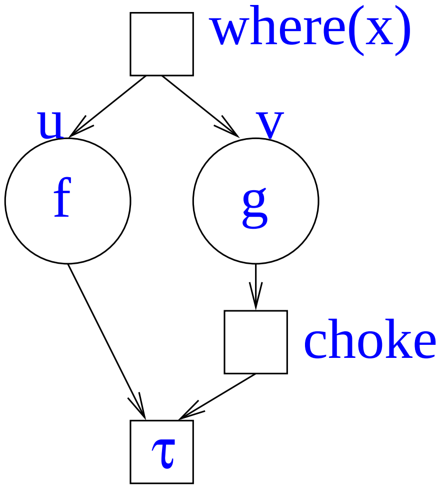
$c := t.cell$
$c.val := t.val$
now no token waits for c;
any token waiting only for c is made ready;
delete all tokens of cell c

Cell within cell: h where $y{:}{\in}$ $(f$ where $x{:}{\in}$ $g)$



- Suppose y is assigned at $choke2$ before x is assigned.
- Need to kill f where $x{:}{\in}$ g's computation, i.e, both f and g.
- Tokens in f and g have different cells.

Cell Tree



- In processing token t at $where(x)$:
  create cell c;
  $c.parent := t.cell$

- In processing token s at $choke$:
  kill all tokens of $s.cell$ and descendant cells.

RootCell

- The initial token has cell $RootCell$.
- The cells form a tree with root $RootCell$.

Summary of Runtime structure

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

Overall algorithm

```
$RootCell.val \neq \bot$              $\rightarrow$ return $RootCell.val$ to main; terminate.
$t.ready$                             $\rightarrow$ process instruction at $t$; delete $t$
$t.suspended \wedge \neg(\exists s :: s.ready)$
                                      $\rightarrow$ on receiving response $r$:
                                            $t.val := r;\ t.state := ready$
else                                  $\rightarrow$ “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 $(\forall t :: t.pending)$. There may be no token at all (for 0).
- else is executed for $M(x)$ where $x :\in 0$
- The algorithm may not terminate: there are suspended tokens, but no deferred site responds.