Dynamic Clone Creation

Jayadev Misra
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712
(512) 471-9547
misra@cs.utexas.edu

December 1, 1997

1 Rules

- Boxes can have parameters. The parameters could be local variables and clones.

  If a box has a clone as a parameter, then the box can be instantiated before or after the parameter has been instantiated. In the former case, the parameter is set to *nil* at the box instantiation.

- Created clones may have actions as well as methods. We see the need for both in practice.

- We don't, as yet, allow methods to have clones aas parameters.

2 Examples

2.1 Simple Sieve

The structure of clones is a linear list starting at the *source* and ending at the *Creator*. *source* outputs the sequence of naturals starting at 2 along its outgoing channel, r. The intermediate nodes are of type $PrimeNumber$. Each $PrimeNumber$ contains a prime number p, and accesses an incoming channel inc and an outgoing channel $outc$. The sole outgoing channel of *source* is r and the sole incoming channel of $Creator$ is $last$. Initially, there is no intermediate node; hence, $r = last$. A number that is sieved by all the intermediate nodes is a prime, and it appears as an item in $last$. The $Creator$ reads an item from $last$, creates a new intermediate node and links it to the last intermediate node and to itself.

clone r : *channel*
clone *source*
  nat v=2
  total action:: $r.put(v)$; $v := v + 1$
end {*source*}

box *PrimeNumber*(p: nat, *inc*: channel, *outc*: channel)
  partial action::
    ; $inc.get(v) \rightarrow$ if $\neg(p \ div\ v)$ then $outc.put(v)$ fi
end {*PrimeNumber*}

clone *Creator*

clone *last*: channel **init** r
clone *next*: channel **init** *nil*
clone *pn*: PrimeNumber

total action::
; $last.get(v) \rightarrow$
print v; $new(next)$; $new(pn(v, last, next))$; $last := next$
end {*Creator*}

2.2 Complex Sieve with method call

The strategy is as before except for the following differences. Suppose a number v is sieved by all primes below p and $p^2 > v$; then, v is a prime. In this case, the clone holding p calls the method *make* of *Creator* to instantiate a clone. the *Creator* is not linked to the last *PrimeNumber* clone, as was the case in the last example. Also, the source calls *Creator* to make the first clone (with prime 2) and it outputs all naturals starting from 3.

clone r : *channel*
clone *source*
nat v=3
total action:: $r.put(v)$; $v := v + 1$
end {*source*} **init** $Creator.make(2)$

box *PrimeNumber*(p: nat, *inc*: channel, *outc*: channel)
partial action::
; $inc.get(v) \rightarrow$
if (p *div* v) then skip
elseif $\{\neg(p\ div\ v)\wedge\}$ $p^2 \leq v$ then $outc.put(v)$
else $\{\neg(p\ div\ v) \wedge\ p^2 > v\}$ $Creator.make(v)$
fi
end {*PrimeNumber*}

clone *Creator*
clone *last*: channel **init** r
clone *next*: channel **init** *nil*
clone *pn*: PrimeNumber

total method $make(v$: nat)::
print v; $new(next)$; $new(pn(v, last, next))$; $last := next$
end {*Creator*}

2.3 Complex Sieve with Channels only

The strategy is as before, except that *Creator* does not include a method that can be called by the other clones; each clone deposits the number v in the channel s, and the *Creator* reads from s.

clone r, s : *channel*
clone *source*

```
    nat v=3
    total action:: r.put(v); v := v + 1
end {source} init s.put(2)

box PrimeNumber(p: nat, inc: channel, outc: channel)
    partial action::
        ; inc.get(v) →
            if (p div v) then skip
            elseif {¬(p div v)∧} p² ≤ v then outc.put(v)
            else {¬(p div v) ∧ p² > v} s.put(v)
            fi
end {PrimeNumber}

clone Creator
    clone last: channel init r
    clone next: channel init nil
    clone pn: PrimeNumber

    total action::
        ; s.get(v) →
            print v; new(next); new(pn(v, last, next)); last := next
end {Creator}
```

2.4 Complex Sieve with Channels only: A minor Optimization

Observe in the last solution that once a clone succeeds in the test $p^2 \leq v$, all future values, v , read from its input channel will also satisfy this test, because the numbers along the channel are increasing and the number p is fixed for a clone. Therefore, the test $p^2 \leq v$ may be eliminated in the future.

In the following solution, we have two box types $Prime1$ and $Prime2$. $Prime1$ is initially used for clone creation. Whenever a clone (that holds p and receives v) detects that $p^2 \leq v$ holds it creates a clone of type $Prime2$ in which the test $p^2 \leq v$ has been eliminated. Then the clone deactivates itself.

If the guard of the partial action in $Prime1$ is declared to be **antistable** then the optimizer can delete this clone and reclaim its space once its guard is found to be *false*.

```
clone r, s : channel
clone source
    nat v=3
    total action:: r.put(v); v := v + 1
end {source} init s.put(2)

box Prime2(p: nat, inc: channel, outc: channel)
    partial action::
        ; inc.get(v) →
            if (p div v) then skip
            else {¬(p div v)} outc.put(v)
            fi
end {Prime2}

box Prime1(p: nat, inc: channel, outc: channel)
```

```
   var b: boolean = false
   clone dup: Prime2 init nil
   partial action::
     ¬b; inc.get(v) →
       if (p div v) then skip
       elseif {¬(p div v)∧} p² ≤ v then outc.put(v); b := true; new(dup(p, inc, outc))
       else {¬(p div v) ∧ p² > v} s.put(v)
       fi
end {Prime2}

clone Creator
   clone last: channel init r
   clone next: channel init nil
   clone pn: PrimeNumber

   total action::
     ; s.get(v) →
       print v; new(next); new(pn(v, last, next)); last := next
end {Creator}
```

2.5 Sieve with Method calls only

```
clone source
   clone next: PrimeNumber init nil
   clone last: PrimeNumber init Two
   var v: nat = 3

   total action generate::
     if Two.test(v) then
       output v;
       new(next(v)); last.addSuccessor(next); last := next
     fi;
     v := v + 1
end {source}

clone Two: PrimeNumber(2)

box PrimeNumber(p: nat)
   clone next: PrimeNumber init nil
   total method addSuccessor(q : PrimeNumber)::
     next := q

   total function test(v)::
     if ¬(p div v) ∧ p² ≤ v {next ≠ nil} then
       test := next.test(v)
     else test := ¬(p div v)
     fi
end {PrimeNumber}
```

3 Fair Scheduling with Dynamic Clones

Here is an abstract representation of a fair schedule when actions are being added. Consider elements from some universe. We have three kinds of operations: add, remove and schedule, each taking an element as a parameter. A *history* is an infinite sequence of these operations, that statisfy the following constraints.

First, define two auxiliary variables S and P which are sets of elements; initially both are ϕ.

- $\{S = S_0 \wedge P = P_0 \wedge e \notin P\}$ $add(e)$ $\{S = S_0 \cup \{e\} \wedge P = P_0 \cup \{e\}\}$
- $\{S = S_0 \wedge e \in S\}$ $remove(e)$ $\{S = S_0 - \{e\}\}$
- $\{e \in S\}$ $schedule(e)$ $\{\}$
- Every element in S is eventually either removed or scheduled:
  $e \in S \mapsto$[remove(e) $\vee$ schedule(e)]

4 Questions

- No need to consider deletion of clones. Handled through Garbage collection.
- Treat static clone structure as a special case of dynamic clone structure?
- What kinds of parameters are allowed in box definitions?