STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume L, Number 1, 2005

CONTINUATIONS FOR REMOTE OBJECTS CONTROL

ENEIA TODORAN[1], FLORIAN MIRCEA BOIAN[2], CORNELIA MELENTI[1], AND NIKOLAOS PAPASPYROU[3]

Abstract. We have recently introduced the "continuation semantics for concurrency" (CSC) technique in an attempt to exploit the benefits of using continuations in concurrent systems development. CSC is a general technique for denotational semantics which provides excellent flexibility in the compositional modeling of concurrent control concepts. In this paper, we present a denotational semantics designed with CSC for a distributed languages incorporating two control concepts which have not been modeled denotationally before: remote object (process) destruction, and cloning.

1. Introduction

The CSC technique was recently introduced by us [13, 14] in an attempt to exploit the benefits of using continuations in concurrent languages development. It is a general technique for denotational semantic design, which can be used to model both sequential and parallel composition in interleaving semantics, as well as various mechanisms for synchronous and asynchronous communication [13]. Intuitively, it is a semantic formalization of a process scheduler simulated on a sequential machine. In the CSC approach, a *continuation* is an application-dependent configuration (structure) of computations, where by *computation* we mean a partially evaluated denotation (meaning function). Every moment there is only one *active* computation, which remains active only until it performs an elementary action. Subsequently, another computation taken from the continuation is planned for execution. In this way it is possible to obtain the desired interleaving behavior for parallel composition.

A continuation is a representation of what remains to be computed after taking an elementary step from the (currently) active computation. This corresponds to the original definition of continuations[1], but in the CSC approach continuations are structured entities and each computation contained in a continuation can be accessed and manipulated separately. Synchronization and communication information can also be encoded in continuations. What one gets with CSC is a

Received by the editors: January 10, 2005.

[1]A continuation is a representation of the rest of the computation, according to [11].

"pure" continuation based approach to communication and concurrency in which all control concepts are modeled as operations manipulating continuations.

As shown in [15], by using the CSC technique, denotational semantics can be used not only for formal specification and design, but also as a general method for building compositional interpreters for concurrent programming languages. In this approach, a denotational (compositional) mapping can use continuations for concurrency to produce incrementally a single stream of observables, i.e. a single execution trace, rather than an element of a power domain construction. By using a random number generator an arbitrary execution trace is chosen, thus simulating the non-deterministic behavior of a "real" concurrent system. Following [15], we will call such a compositional mapping a *denotational prototype*.

In this paper, we employ the CSC technique in designing a denotational semantics and a corresponding denotational prototype for a simple distributed language providing operations for remote object (process) control. In the sequel, an *object* is a thread (sequence) of computations with a local state. Distributed states are essential for defining the semantics of concurrent languages used in distributed computing. The language that we study in this paper provides a mechanism for synchronous communication taken from CSP [4]. It incorporates a notion of remote processes as *named objects* and allow object-to-object communication, as well as remote object destruction and cloning.

The last two operations can be encountered at operating system level, in some coordination languages [5], or in distributed object oriented and multi-agent systems such as Obliq and IBM Java Aglets [3, 7, 6]. The former operation kills a parallel running object and is similar to the "`kill -9`" system call in Unix. The latter operation creates an identical copy of a (parallel) running object. In this paper, we provide an accurate denotational semantic model for remote object destruction and cloning. To the best of our knowledge, such operations for remote object control have not been modeled denotationally until now, and all our attempts solve the problem by using only classic compositional techniques have failed.

Instead of using mathematical notation for the definition of the denotational models, we use the (lazy) functional programming language Haskell [9]. In this way, as in [14], we avoid unnecessary complexities accompanying the use of domain theory or the theory of metric spaces, which could have been adopted alternatively. At the same time, we allow our denotational models to be directly implementable, in the form of interpreters for the language under study, and thus to be easily tested and evaluated. The denotational semantics will only be tested on trivial (non-recursive) example programs. However, the corresponding denotational prototype will be tested on "real-life" examples. For example, we present a simple concurrent generator of prime numbers based on the sieve of Erathostenes.

2. Syntax and informal explanation

We consider a simple distributed language called L_{obj}. The syntax of L_{obj} is given below in BNF. The basic components are a set $(v \in)V$[2] of *data variables*, a set $(n \in)N$ of *numerical expressions*, a set $(b \in)B$ of *boolean expressions*, and a set $(y \in)Y$ of *procedure variables*. The language also uses a class $(w \in)W$ of *object variables*; while a data variable holds a data value (in our case an integer value) an object variable holds an object reference. L_{obj} comprises a simple language of expressions, supporting basic operators on numerical values and boolean values. In the grammar that follows, z denotes an integer constant, and v denotes a (numerical) variable.

$$n ::= z \mid v \mid n + n \mid n - n \mid n \,\%\, n \mid \ldots$$
$$b ::= n == n \mid n < n \mid \ldots$$

L_{obj} provides assignment ($v := n$), a primitive for writing the value of a numerical value (i.e. for producing an intermediate observable) at the standard output file (**write** n)[3], a null command (**skip**), recursion, a conditional command (**if** b **then** x **else** x), sequential composition ($x\,;\,x$), guarded nondeterministic choice (**ned** $[(\gamma \rightarrow x)^*]$), together with constructs for object (process) creation (**new** w **is** x), destruction (**kill** w), and cloning (**clone** w **is** w). The syntax of L_{obj} is formally defined as follows:

$$\begin{aligned} x ::= {} & \textbf{skip} \mid v := n \mid \textbf{write}\ n \mid \textbf{call}\ y \mid \textbf{letrec}\ y\ \textbf{be}\ x\ \textbf{in}\ x \\ & \mid \textbf{if}\ b\ \textbf{then}\ x\ \textbf{else}\ x \mid \textbf{ned}\ [(\gamma \rightarrow x)^*] \mid x\,;\,x \\ & \mid \textbf{new}\ w\ \textbf{is}\ x \mid \textbf{kill}\ w \mid \textbf{clone}\ w\ \textbf{is}\ w \end{aligned}$$

where

$$\gamma ::= w\,!\,n \mid ?\,v$$

The guards γ of a non-deterministic choice are constructs for *object-to-object* synchronous communication. In L_{obj}, an *object* is a thread (sequence) of computations acting on a *local state*. There is no shared memory area. Parallel objects can only communicate by exchanging messages. The communication mechanism is taken from CSP [4]. A communication can take place by the synchronous execution of two actions $w\,!\,n$ and $?\,v$, occurring in parallel objects. The primitive $w\,!\,n$ evaluates the expression n and sends the value to the object referred by w. An object executing the $?\,v$ statement is willing to communicate with an arbitrary partner object, as long as that partner explicitly mentions the name of the object in which the $?\,v$ primitive occurs; upon synchronization, the primitive $?\,v$ assigns the received value to the variable v. The expression n is evaluated in

[2] In this paper, the notation $(x, y, \ldots)X$ introduces the set X with typical variables x, y,...

[3] Expressions of boolean type can not be assigned or output; they can only be used as conditions.

the memory area of the sending object and the result is assigned to the variable v in the memory area of the receiving object. In order to communicate, two parallel objects synchronize (the first one that is ready to communicate waits for the other) and then they exchange a single value.

The **new** w **is** x statement can be used to create a new object (with a new private state) which evaluates x. A reference to the newly created object is assigned to variable w. Therefore, this statement not only creates a new object but also a new communication connection to this object, which can be used by a $w\,!\,n$ primitive. Moreover, the new reference can be used by the two constructs for remote object control: **kill** w and **clone** w_1 **is** w . The former destroys the parallel running object to which variable w refers. The latter clones the parallel running object referred by w and assigns the clone's identifier to the object variable w_1.

In Haskell, we implement the syntax of L_{obj} as follows:

```
type V = String
type W = String
type Y = String
data N = Z Int | V V | Plus N N | Minus N N | Mod N N
data B = Eq N N | Lt N N
data C = Snd W N | Rcv V
data X = Skip | Assign V N | Write N | Call Y  | LetRec Y X X
       | If B X X | Ned [(C,X)] | Seq X X
       | New W X | Kill W | Clone W W
```

3. Denotational semantics

L_{obj} is a language with distributed objects. Objects can be referred and controlled by using *object identifiers* (or *references*). For simplicity, we represent object references by integer numbers.

```
type O = Int
```

Each object in L_{obj} has a *local state*, which can be accessed or modified in an imperative manner. A state is usually represented as a mapping from variables to values. In L_{obj} a state has two components: one for data values and the other one for object references. We implement states by the type `S`. The operations `getv`, `setv`, `getw` and `setw` provide the basic functionality of a state.

```
type S = (W -> O,V -> Int)

getv :: V -> S -> Int
getv v (sw,sv) = sv v
```

```
setv :: S -> V -> Int -> S
setv (sw,sv) v i = (sw,subs sv v i)

getw :: W -> S -> O
getw w (sw,sv) = sw w

setw :: S -> W -> O -> S
setw (sw,sv) w o' = (subs sw w o',sv)
```

The mapping `subs` is defined as follows:

```
subs :: (Eq a) => (a -> b) -> a -> b -> (a -> b)
subs f x y = \x' -> (if (x==x') then y else f x')
```

We can already define simple valuations `evN` and `evB`, for numerical and boolean expressions. In general, the meaning of a (boolean) expression depends on the current state.

```
evN :: N -> S -> Int
evN (Z n) s = n
evN (V v) s = getv v s
evN (Plus n1 n2) s = (evN n1 s) + (evN n2 s)
evN (Minus n1 n2) s = (evN n1 s) - (evN n2 s)
evN (Mod n1 n2) s = (evN n1 s) `mod` (evN n2 s)

evB :: B -> S -> Bool
evB (Eq n1 n2) s = (evN n1 s) == (evN n2 s)
evB (Lt n1 n2) s = (evN n1 s) < (evN n2 s)
```

In L_{obj} it is possible for a program to block, if all parallel objects are waiting at nondeterministic constructs that do not have matching communication primitives. Such a deadlock is fundamentally different from non-termination (e.g. a procedure that repeatedly calls itself) and we expect it to be detected by the denotational semantics. We use the type `Q` to represent streams (lists) of observables. In the definition given below, `Epsilon` denotes *normal termination* and `Deadlock` denotes *deadlock*.

```
data Q = Epsilon | Deadlock | Q Int Q
```

We use the following `Show` instance to visualize the yields of our denotational models.

```
instance Show Q where
   show Epsilon = " "
   show Deadlock = " deadlock "
   show (Q n q) = " " ++ (show n) ++ (show q)
```

The denotational semantics for L_{obj} maps each statement to a *computation* (a partially evaluated denotation), which is an element of type `D`. We will use continuation semantics for concurrency, therefore it is reasonable to assume that a computation is a function that depends on the current continuation. In the definition below, `Cont` is the semantic class of continuations. The semantics of a program also depends on the current state.

```
type D = Cont -> S -> Final
```

`Final` is the final yield of the denotational mapping. In section 4, `Final` will implement a domain for random execution traces [15]. In this section, `Final` implements a classical power domain construction [10], and the denotational semantics produces the collection of all possible traces for any given program; the Haskell definition will be given later.

Following the CSC technique [13, 14], a continuation is a configuration of computations that can be executed in parallel. The CSC technique is very general. It does not impose any restriction on the structure of continuations. For our language with object creation it is convenient to define continuations to be multisets of objects. Objects are elements of type `Obj`. An object is a triple, consisting of an object identifier, a thread (sequence) of computations, and a local state. We use two basic notions to model the flow control: the *stack* to model sequential composition, and the *multiset* to model parallel composition. A stack models a single thread (or sequence) of computations. We implement both stacks and multisets as Haskell's lists. The type `PC` implements a multiset. The type `SC` implements a stack. An element of a `SC` stack is either a computation or a non-deterministic choice consisting of a list of guarded alternatives, where each alternative consists of a (synchronous) communication attempt and a computation. Haskell definitions are as follows:

```
type Cont = PC
type PC = [Obj]
type Obj = (O,SC,S)
type SC = [Comp]
data Comp = D D | S [(SemC,D)]
```

We use some auxiliary mappings on objects.

```
idOf :: Obj -> O
idOf (o,sc,s) = o

threadOf :: Obj -> SC
threadOf (o,sc,s) = sc

stateOf :: Obj -> S
```

```
stateOf (o,sc,s) = s

updThread :: Obj -> SC -> Obj
updThread (o,sc,s) sc' = (o,sc',s)

updState :: Obj -> S -> Obj
updState (o,sc,s) s' = (o,sc,s')
```

The type `SemC` implements communication attempts. The function `semC` maps the (syntactic) communication primitives of L_{obj} to corresponding (semantic) communication attempts.

```
data SemC = SemSnd W (S -> Int) | SemRcv V

semC :: C -> SemC
semC (Snd w e) = SemSnd w (evN e)
semC (Rcv v) = SemRcv v
```

The function `k` implements *continuation completion*. It maps a continuation to the program answer that would result if the continuation alone was left to execute. It first normalizes the continuation by using the auxiliary mapping `re`. The execution terminates if the (normalized) continuation is empty. Otherwise, `k` calls the function `kc` which implements a scheduler, by using the auxiliary functions `schedc`, `comp`, `scheds`, and `send`.

```
k :: Cont -> Final
k c = case (re c) of {
           [] -> epsilon;
           c  -> kc c;
      }

kc :: Cont -> Final
kc c = case ((schedc c) ++ (scheds c [])) of {
            []  -> deadlock;
            scd -> bigned (map exe scd);
       }

schedc :: PC -> [Sched]
schedc pc =
    [ (Schedc d (obj:pc') (stateOf obj)) | (D d,obj:pc') <-
     comp pc [] ]

comp :: PC -> PC -> [(Comp,PC)]
```

```
comp [] pc' = []
comp (obj:pc) pc' =
    (let p:sc = threadOf obj in
     [(p,(updThread obj sc):(pc ++ pc'))])
 ++ (comp pc (obj:pc'))

scheds :: PC -> PC -> [Sched]
scheds [] pc' = []
scheds (obj:pc) pc' =
    (send [obj] (pc ++ pc')) ++ (scheds pc (obj:pc'))

send :: PC -> PC -> [Sched]
send pc1 pc2 =
    [ (Scheds ((addc (D d1) (obj1:pc1')) ++
         (addc (D d2) (updState obj2 (setv (stateOf obj2)
                         v (pe (stateOf obj1))):pc2')))) |
      (S snd,obj1:pc1') <- comp pc1 [],
      (S rcv,obj2:pc2') <- comp pc2 [],
      (SemSnd w pe,d1) <- snd, (SemRcv v,d2) <- rcv,
      (getw w (stateOf obj1)) == idOf obj2
    ]
```

Continuations are multisets of objects. The semantic operators are designed in such a way as to maintain the following *invariant* of the continuations: *the thread of each object in a continuation is always non-empty, with the possible exception of the leftmost one which conceptually contains at its head the* active *computation.* The normalization function `re` removes the leftmost object in a continuation in case its thread has remained empty after taking an elementary step from the active computation.

```
re :: Cont -> Cont
re ((o,[],s):pc) = pc
re pc = pc
```

Both ordinary computations and pairs of communicating processes are handled by the scheduler mapping `kc`. The function `schedc` handles ordinary computations. The function `scheds` handles pairs of communicating objects (processes). The scheduler computes all possible schedules for a given continuation. Deadlock is detected when there are no schedules. A schedule is an element of the type `Sched`.

```
data Sched = Schedc D Cont S | Scheds Cont
```

Schedules of the form `Schedc d c s` are produced by the function `schedc`. Schedules of the form `Scheds c` are produced by `scheds`. The mapping `exe` executes a single schedule.

```
exe :: Sched -> Final
exe (Schedc d c s) = d c s
exe (Scheds c) = k c
```

The scheduler also uses the mapping `bigned` to compute the meaning corresponding to all possible schedules. A possible definition of `bigned` for a classical power domain semantics is given later in this section. An alternative definition of `bigned`, suitable for computing a single arbitrary execution trace, is considered in section 4.

The denotational function for L_{obj} uses the following semantic operators for modeling the flow of control: `addc`, `new`, `kill` and `clone`. The operator `addc` adds a computation to the continuation for sequential composition. The operators `new`, `kill` and `clone` are used for object creation, destruction, and cloning respectively.

```
addc :: Comp -> Cont -> Cont
addc p (obj:pc) = (updThread obj (p:threadOf obj)):pc

new :: Comp -> Cont -> W -> S -> Cont
new p (obj:pc) w s = let on = newo (obj:pc)
      in  (updState obj (setw s w on)):(on,[p],s0):pc

kill :: Cont -> W -> S -> Cont
kill pc w s =
   aux pc (getw w s)
   where aux :: PC -> O -> PC
         aux [] ok = []
         aux (obj:pc) ok =
             if (idOf obj == ok) then pc else (obj:(aux pc ok))

clone :: Cont -> W -> W -> S -> Cont
clone (obj:pc) wn wo s =
   let on = newo (obj:pc)
   in  aux (updState obj (setw s wn on):pc) (getw wo s) on
       where aux :: PC -> O -> O -> PC
             aux [] oo on = error "clone: invalid object name"
             aux (obj:pc) oo on =
                 if (idOf obj == oo) then
                    case (threadOf obj) of {
```

```
                  [] -> obj:pc;
                  _  -> obj:(on,threadOf obj,stateOf obj):pc;
               }
          else obj:aux pc oo on
```

The mapping `newo` takes as parameter a continuation and creates a new fresh object identifier. It returns an identifier which is not already in use by some object in the given continuation.

```
newo :: Cont -> O
newo c = (maximum [ idOf obj | obj <- c ]) + 1
```

For handling recursion, we use semantic environments and a fixed-point operator. A semantic environment is a mapping from procedure variables to computations.

```
type Env = Y -> D

fix :: (a -> a) -> a
fix f = f (fix f)
```

We are finally prepared to present the denotational semantics for L_{obj}.

```
sem :: X -> Env -> D
sem Skip e c s = k c
sem (Assign v n) e (obj:pc) s  =
    k (updState obj (setv s v (evN n s)):pc)
sem (Write n) e c s = prefix (evN n s) (k c)
sem (Ned gx) e c s =
    k (addc (S [(semC c,sem x e) | (c,x) <- gx ]) c)
sem (Seq x1 x2) e c s = sem x1 e (addc  (D (sem x2 e)) c) s
sem (Call y) e c s = e y c s
sem (LetRec y x1 x2) e c s =
    sem x2 (subs e y (fix (\d -> (sem x1 (subs e y d))))) c s
sem (If b x1 x2) e c s =
    if (evB b s) then (sem x1 e c s) else (sem x2 e c s)
sem (New w x) e c s = k (new (D (sem x e)) c w s)
sem (Kill w) e c s = k (kill c w s)
sem (Clone wn wo) e c s = k (clone c wn wo s)
```

When the CSC technique is employed in semantic design one can use a *linear-time* domain (see [1]) as final yield of a denotational model for synchronous communication [13]. Intuitively, an element of type `Final` is a set of `Q` sequences of observables. The constants `epsilon` and `deadlock` are of the type `Final`. The former models normal termination and the latter models deadlock detection in the `Final` domain. The `bigned` operator computes the union of a list of elements of

type `Final`. To this end, it is convenient to make `Q` an instance of `Eq`. The `prefix` operator implements the prefixing of an observable to a final program answer.

```
type Final = [Q]

instance Eq Q where
   Epsilon == Epsilon = True
   Deadlock == Deadlock = True
   (Q n1 q1) == (Q n2 q2) = (n1 == n2) && (q1 == q2)
   _ == _ = False

epsilon, deadlock :: Final
epsilon = [Epsilon]
deadlock = [Deadlock]

prefix :: Int -> Final -> Final
prefix n p = [ (Q n q) | q <- p ]

bigned :: [Final] -> Final
bigned [] = []
bigned (q:p) = q `union` (bigned p)

union :: (Eq a) => [a] -> [a] -> [a]
union [] ys = ys
union (x:xs) ys =
   if (x `elem` ys) then (xs `union` ys) else x:(xs `union` ys)
```

In order to test our denotational semantics we define *initial* values for the semantic environment, continuation, and state.

```
e0 :: Env;    e0 y c s = epsilon;
c0 :: Cont;   c0 = [(o0,[],s0)];
o0 :: O;      o0 = 0;
s0 :: S;      s0 = (\w -> o0,\v -> 0);
```

For experiments, we consider the following example programs in L_{obj}.

```
x1 = Seq (New "w1" (Write (Z 1))) (Seq (New "w2" (Write (Z 2)))
 (Write (Z 3)))
x2 = Seq (New "w1" (Ned [(Rcv "v",Write (V "v"))]))
         (Ned [(Snd "w1" (Z 1),Ned [(Rcv "v",Skip)]),
               (Snd "w1" (Z 2),Write (Z 2))])
x3 = Seq (New "w1" (Ned [(Rcv "v",
  Seq (Write (Z 1))
```

```
                              (Seq (Write (Z 2))
                                   (Ned [(Rcv "v",Skip)])))]))
          (Ned [(Snd "w1" (Z 0),
           Seq (Clone "w2" "w1")
                     (Seq (Clone "w3" "w1")
                          (Seq (Kill "w1")
                          (Seq (Kill "w2") (Kill "w3"))))])
```

One can perform the following tests[4]:

```
Main> sem x1 e0 c0 s0
[ 3 2 1 , 3 1 2 , 2 1 3 , 2 3 1 , 1 3 2 , 1 2 3 ]
Main> sem x2 e0 c0 s0
[ 1 deadlock , 2 2 ]
Main> sem x3 e0 c0 s0
[ , 1 2 2 1 2 , 1 2 2 1 , 1 2 1 2 2 , 1 1 2 2 2 , 1 1 2 1 , 1 1 2
1 2 , 1 1 2 1 2 2 , 1 1 2 2 1 2 , 1 1 2 2 1 , 1 1 1 , 1 1 1 2 , 1
1 1 2 2 , 1 1 1 2 2 2 , 1 1 , 1 1 2 2 , 1 1 2 , 1 2 1 1 2 2 , 1 2
1 1 , 1 2 1 1 2 , 1 2 1 2 1 , 1 2 1 2 1 2 , 1 2 1 2 , 1 2 1 , 1 ,
1 2 2 2 , 1 2 , 1 2 2 ]
```

4. Denotational prototype

In [15] we have introduced the notion of a *denotational prototype*. A denotational prototype is a compositional mapping that produces a single execution trace for a given concurrent program rather than the collection of all possible traces. By using a random number generator, an arbitrary execution trace is chosen, thus simulating the non-deterministic behavior of a "real" concurrent system. A denotational prototype is a compositional interpreter for the concurrent language under study, which can be used without difficulty to test non-trivial concurrent algorithms.

It is very easy to modify the denotational semantics given in section 3 to get a denotational prototype for L_{obj}. We change the definition of the type `Final` to reflect the fact the the final yield of the denotational prototype is a sequence of observables (a single execution trace) of type `Q`. The denotational prototype simulates the selection of an arbitrary execution trace by using a random number generator, which is an element of type `RNG`.

```
type Final = RNG -> Q
```

The new definitions for `epsilon` and `deadlock` are as follows:

[4]We accomplished the experiments by using the Hugs interpreter available from `http://www.haskell.org`.

```
epsilon, deadlock :: Final
epsilon = \rng -> Epsilon
deadlock = \rng -> Deadlock
```

Random numbers are natural numbers. A random number generator is a pair consisting of a random number and a mapping that produces a new random number from a given one. `rng0` is a poor man's random number generator that will be used to test our denotational prototype for L_{obj}.

```
type R = Int

type RNG = (R,R -> R)

rng0 :: RNG
rng0 = (17489,\r -> ((25173*r+13849) `mod` 65536))
```

All that remains to be done is to adapt the definitions of `bigned` and `prefix` to deal with single arbitrary execution traces. The new definitions are given below.

```
bigned :: [Final] -> Final
bigned fs = \(r,next) -> (nth fs (r `mod` (length fs))
       (next r,next))

nth :: [a] -> Int -> a
nth (z:xs) 0 = z
nth (z:xs) n = nth xs (n-1)

prefix :: Int -> Final -> Final
prefix n f = \rng -> (Q n (f rng))
```

The function `(test x m)` defined below calls `m` times our semantic interpreter to execute the program `x`. Each time, the random number generator is initialized with a new (pseudo-)random value. As a consequence, different results (execution traces) are produced at consecutive executions of the same program, thus simulating the non-deterministic behavior of a "real" concurrent system.

```
test :: X -> Int -> IO ()
test x m = aux x m rng0
     where aux x 0 rng = return ()
           aux x m (r,next) =
               do { putStr (show (sem x e0 c0 s0 (r,next))
                ++ "\n\n");
                    aux x (m-1) (next r,next);
                  }
```

We test the denotational prototype for L_{obj} on "real life" programs. The first one is a concurrent generator of prime numbers based on the sieve of Erathostenes. It creates a new object for each prime number. Therefore, in this case the performance degrades continuously. The second example program demonstrates remote object destruction and cloning. A counting object is created and left alone to do its job for a little while. Then, two clones of this object are created and there are three counters working in parallel. After some time, the three objects are killed and the program terminates.

```
x4 = LetRec "drive"
        (Ned [(Snd "c" (V "i"),Seq (Assign "i" (Plus (V "i") (Z 2)))
                                   (Call "drive"))])
     (LetRec "run"
        (Seq (Seq (Ned [(Rcv "i",Skip)])
                  (If (Eq (Z 0) (Mod (V "i") (V "p")))
                      Skip
                      (Ned [(Snd "cout" (V "i"),Skip)])))
             (Call "run"))
     (LetRec "sieve"
      (Seq (Ned [(Rcv "p",Skip)])
      (Seq (Write (V "p"))
      (Seq (New "cout" (Call "sieve")) (Call "run"))))
      (Seq (Assign "i" (Z 3)) (Seq (New "c" (Call "sieve"))
           (Call "drive"))))
x5 = (LetRec "y"
         (Seq (Write (V "v"))
              (Seq (Assign "v" (Plus (V "v") (Z 1))) (Call "y")))
     (LetRec "sleep"
         (If (Lt (Z 0) (V "v1"))
             (Seq (Assign "v1" (Minus (V "v1") (Z 1))) (Call "sleep"))
             Skip)
         (Seq (New "w" (Seq (Assign "v" (Z 10)) (Call "y")))
              (Seq (Seq (Seq (Assign "v1" (Z 3)) (Call "sleep"))
                        (Seq (Clone "w1" "w") (Clone "w2" "w")))
                   (Seq (Seq (Assign "v1" (Z 7)) (Call "sleep"))
                        (Seq (Kill "w") (Seq (Kill "w1")
                             (Kill "w2")))))))))
```

Now one can perform the following experiments:

```
Main> test x4 1
 3 5 7 11 13  17 19 23 29 31 37 41 43 47 53 59 61 67 71 73{Interrupted!}
```

```
Main> test x5 4
 10 11 12 13 14 13 14 15 15 14 15 16 16 17 16 17 18 18 17 18 19 19

 10 11 12 13 13 13 14 14 15 14 15 15 16 16 16 17 17 18 18 19

 10 11 12 13 13 13 14 14 15 14 15 15 16 16 16 17

 10 11 12 13 13 14 14 14 15 15 15 16 16 16 17 17 17 18 18 18 19 19
```

5. Conclusions and future research

The CSC technique provides a discipline for compositional development of concurrent programming languages based on the concept of a *continuation*. It provides the ability to encapsulate the concurrent behavior in continuations. The semantic model for remote object control given in this paper shows that, by using the CSC technique parallel computations can be manipulated as data in a strict denotational framework. Classic compositional technique do not seem to provide an adequate framework for handling such operations.

In the near future, fundamental research related to the CSC technique will be conducted in two main directions. First, in order to provide an abstract framework for handling context changes and locality in concurrent languages development, we plan to study the possibility of using the CSC technique in the possible world semantics, eventually by extending models given in [12, 2]. Second, in order to improve the flexibility, elegance and modularity of the denotational semantic descriptions, we also plan to study the possibility to define monads [8, 16] for the CSC technique.

References

[1] J.W. de Bakker and E.P. de Vink. *Control flow semantics*. MIT Press, 1996.

[2] S. Brookes. The essence of parallel Algol. *Information and Computation*, vol. 179(1), pages 118–149, 2002.

[3] L. Cardelli. A language with distributed scope. In *Proc. of the 22nd Annual ACM Symposium on Principles of Programming Languages*, pages 286–297, 1995.

[4] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[5] A.A. Holzbacher. A software environment for concurrent coordinated programming. In *Proc. of 1st Int. Conf. on Coordination Languages and Models*, pages 249–267, Springer, 1996.

[6] IBM Aglets website: http://www.trl.ibm.com/aglets.

[7] D. Lauge and M. Oshima. *Programming and deploying Java mobile agents with Aglets*. Addison Wesley, 1998.

[8] E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, University of Edinburgh, 1990.

[9] S. Peyton Jones and J. Hughes (editors). *Report on the programming language Haskell 98: a non-strict purely functional language*, 1999. Available from `http://www.haskell.org/`.

[10] G.D. Plotkin. A powerdomain construction. *SIAM Journal of Computing*, vol. 5, pages 522–587, 1976.

[11] C. Stratchey and C.P. Wadsworth. Continuations: a mathematical semantics for handling full jumps. Technical monograph PRG-11, Programming Research Group, Univ. Oxford, 1974.

[12] R.D. Tennent and J.K. Tobin. Continuations in possible world semantics. *Theoretical Computer Science*, vol. 85(2), pages 283–303, 1991.

[13] E. Todoran. Metric semantics for synchronous and asynchronous communication: a continuation-based approach. In *Proc. of FCT'99 Workshop on Distributed Systems*, *Electronic Notes in Theoretical Computer Science (ENTCS)*, vol. 28, pages 119–146, Elsevier, 2000.

[14] E. Todoran and N. Papaspyrou. Continuations for parallel logic programming, In *Proc. of 2nd International ACM-SIGPLAN Conference on Principles and practice of Declarative Programming (PPDP'00)*, pages 257–267, ACM Press, 2000.

[15] E. Todoran and N. Papaspyrou. Denotational prototype semantics for a simple concurrent language with synchronous communication. Technical report CDS-SW-TR-1-04, National Technical University of Athens, School of Electrical and Computer Engineering, Software Engineering Laboratory, 2004.

[16] P. Wadler. Monads for functional programming. In *Advanced Functional Programming*, Springer, LNCS 925, 1995.

[1] Technical University of Cluj-Napoca, Faculty of Automation and Computer Science, Department of Computer Science, Baritiu Str. 28, Cluj-Napoca, ROMANIA

E-mail address: {`Eneia.Todoran,Cornelia.Melenti`}`@cs.utcluj.ro`

[2] "Babes-Bolyai" University of Cluj-Napoca, Faculty of Mathematics and Computer Science, Department of Computer Science. M. Kogalniceanu Str. 1, Cluj-Napoca, ROMANIA

E-mail address: `florin@cs.ubbcluj.ro`

[3] National Technical University of Athens, Department of Electrical and Computer Engineering, Software Engineering Laboratory, Polytechnioupoli, 15780 Zografou, Athens, GREECE

E-mail address: `nickie@softlab.ntua.gr`