Object-Oriented Programming

Annalise Tarhan

July 8, 2020

1 Uninitialized Objects

1.1 Write a function to create a new object that is based on New but does not require an initial message.

```
fun {New2 Class}
    Nils={Map Class.attrs fun {$ _} nil end}
    Init={List.toRecord init Nils}
in
    {New Class Init}
end
```

2 Protected Methods in the Java Sense

2.1 Define a linguistic abstraction that allows annotating a method or attribute as protected in the Java sense.

```
functor
export
    setOfAllProtectedAttributes:[A]
    superClass:C
define
    class C
        attr pa:A
        meth A(X) skip end
    end
end
```

3 Method Wrapping

3.1 Rewrite TraceNew2 so that it uses a class with no external references.

```
fun {TraceNew2 Class Init}
    Obj={New Class Init}
    class Tracer
        meth uniqueInitMethod skip end
        meth otherwise(M)
            {Browse entering({Label M})}
            {Obj M}
            {Browse exiting({Label M})}
        end
    end
in
    {New Tracer uniqueInitMethod}
end
```

4 Implementing Inheritance and Static Binding

4.1 Generalize the implementation of the object system to handle static binding and to handle inheritance with any number of superclasses.

```
fun {New WClass InitialMethod}
    ...
    {Record.forAll State
        proc {$ A} {NewCell Class.attrs.A A} end}
    ...
end

fun {FromExtended C1 Supers}
    case Supers
    of nil then C1
    [] C2|nil
    then {From C1 C2 {Wrap c(methods:m() attrs:a())}}
    [] C2|C3|Rest
    then {FromExtended {From C1 C2 C3} Rest}
    end
end
```

5 Message Protocols with Active Objects

5.1 Redo the message protocols with active objects instead of port objects.

```
% Remote Method Invocation
class ServerProc
      meth init skip end
      meth calc(X Y)
            Y=X*X+2.0*X+2.0
      end
end
class ClientProc
      meth init skip end
      meth work(Y) Y1 Y2 in
            {Server calc(10.0 Y1)}
            {Wait Y1}
            {Server calc(20.0 Y2)}
            {Wait Y2}
            Y=Y1+Y2
      end
end
Server={NewActive ServerProc init}
Client={NewActive ClientProc init}
```

```
% Asynchronous RMI
% Same ServerProc as RMI
class ClientProc
      meth init skip end
      meth work(?Y) Y1 Y2 in
      {Server calc(10.0 Y1)}
      {Server calc(20.0 Y2)}
      Y=Y1+Y2
      end
end
```

```
% RMI with Callback (using thread)
class ServerProc
    meth init skip end
    meth calc(X ?Y Client) X1 D in
        {Client delta(D)}
        X1=X+D
        Y=X1*X1+2.0*X1+2.0
    end
end
class ClientProc
    meth init skip end
    meth work(?Z) Y in
        {Server calc(10.0 Y self)}
        thread Z=Y+100.0 end
    end
    meth delta(?D)
        D=1.0
    end
end
```

```
% RMI with Callback (using record continuation)
class ServerProc
    meth init skip end
    meth calc(X Client Cont) X1 D Y in
        {Client delta(D)}
        X1=X+D
        Y=X1*X1+2.0*X1+2.0
        {Client cont(Cont#Y)}
    end
end
class ClientProc
    meth init skip end
    meth work(?Z)
        {Server calc(10.0 Y self cont(Z))}
        thread Z=Y+100.0 end
    end
    meth cont(X)
        case X
        of cont(Z)#Y then Z=Y+100.0
        end
    end
    meth delta(?D) D=1.0 end
end
```

```
% RMI with Callback (using procedure continuation)
class ServerProc
    meth init skip end
    meth calc(X Client Cont)
        X1 D Y
    in
        {Client delta(D)}
        X1=X+D
        Y=X1*X1+2.0*X1+2.0
        {Client cont(Cont#Y)}
    end
end
class ClientProc
    meth init skip end
    meth work(?Z)
        C=proc {$ Y} Z=Y+100.0 end
    in
        {Server calc(10.0 self cont(C))}
    end
    meth cont(X)
        case X of cont(C)#Y then {C Y} end
    end
    meth delta(?D)
        D=1.0
    end
end
```

```
% Error reporting
class ServerProc
    meth init skip end
    meth sqrt(X Y E)
        try
            Y={Sqrt X}
            E=normal
        catch Exc then
            E=exception(Exc)
        end
    end
end
```

```
% Asynchronous RMI with callback
class ServerProc
    meth init skip end
    meth calc(X ?Y Client) then X1 D in
        {Client delta(D)}
        thread
            X1=X+D
            Y=X1*X1+2.0*X1+2.0
        end
    end
end
class ClientProc
    meth init skip end
    meth work(?Z) Y1 Y2 in
        {Server calc(10.0 Y1 self)}
        {Server calc(20.0 Y2 self)}
        thread Y=Y1+Y2 end
    end
    meth delta(?D) D=1.0 end
end
```

```
% Double callbacks
class ServerProc
    meth init skip end
    meth calc(X ?Y Client) then X1 D in
        {Client delta(D)}
        thread
            X1=X+D
            Y=X1*X1+2.0*X1+2.0
        end
    end
    meth serverdelta(?S) S=0.01 end
end
class ClientProc
    meth init skip end
    meth work(Z) Y in
        {Server calc(10.0 Y self)}
        thread Z=Y+100.0 end
    end
    meth delta(?D) S in
        {Server serverdelta(S)}
        thread D=1.0+S end
    end
end
```

6 The Flavius Josephus Problem

6.1 Use the sequential stateful model to solve the problem. Write two programs: one without short-circuiting and one with it.

```
fun {Josephus N K}
      Ring={NewArray 1 N true}
      Survivors={NewCell N}
      Index={NewCell 1}
      fun {IsAlive I} Ring.I end
      fun {NextIndex I} if I==N then 1 else I+1 end end
      fun {FindSurvivor I}
            if {IsAlive I} then I
            else {FindSurvivor {NextIndex I}} end
      end
      fun {SkipK I Skip}
            if {IsAlive I} then
                  if Skip==0 then I
                  else {SkipK {NextIndex I} Skip-1} end
            else {SkipK {NextIndex I} Skip} end
      end
      proc {While Expr Stmt}
            if {Expr} then {Stmt} {While Expr Stmt} end
      end
in
      {While
            fun {$} @Survivors>1 end
            proc {$}
            Index:={SkipK @Index K}
            {Array.put Ring @Index false}
            Survivors:=@Survivors-1
            end
      }
      {FindSurvivor 1}
end
```

```
class Victim
      attr index prev next
      meth init(I) index:=I prev:=I-1 next:=I+1 end
      meth setPrev(P) prev:=P end
      meth setNext(N) next:=N end
      meth getPrev(X) X=@prev end
      meth getNext(X) X=@next end
end

fun {Josephus N K}
      Ring={NewArray 1 N null}
      Survivors={NewCell N}
      Current={NewCell 1}
      proc {While Expr Stmt}
            if {Expr} then {Stmt} {While Expr Stmt} end
      end
      proc {KillCurrent}
            P N in
            {Ring.@Current getPrev(P)}
            {Ring.@Current getNext(N)}
            {Ring.P setNext(N)}
            {Ring.N setPrev(P)}
            Survivors:=@Survivors-1
            Current:=N
      end
in
      for I in 1..N do
            Ring.I:={New Victim init(I)}
      end
      {Ring.1 setPrev(N)}
      {Ring.N setNext(1)}
      {While
            fun {$} @Survivors>1 end
            proc {$}
                  for I in 1..K do
                        Current:={Ring.@Current getNext($)}
                  end
                  {KillCurrent}
            end
      }
      @Current
end
```