Portable Simula Revisited

Runtime Design

Working Draft

(as of 26 april 2018)

Background.

The 50th anniversary of Simula, the first object-oriented programming language, was held at the Department of Informatics, University of Oslo on 27th September, 2017. James Gosling, the inventor of Java, gave overwhelming credit to the legacy of Simula in his lecture. He mentioned that a number of programming languages nowadays compile to JVM. During his lecture, I began to wonder - why has no team made a simula implementation in java, written in java that compiles to java / java virtual machine(JVM).

Inspired by the lectures on the Simula Jubilee, I began to ask myself if I should try to make a Simula implementation using today's tools. It should certainly be a Recursive Descent parser based on the work of Niklaus Emil Wirth, something I've ever wanted since Borris Magnuson showed that it was an excellent technique.

In early November I started work on such a compiler. I was surprised that it took so short time to complete a rough version that compiled to Java code. This, first version, was finished just before Christmas. It was not in any way complete, but it generated some kind of Java code that could be compiled and executed. However, the original semantics of the Simula program were not preserved.

I presented these results in a meeting at Blindern, Ole-Johan Dahls Hus, just before Christmas in 2017. I was then encouraged by Stein Krogdahl and Dag Langmyhr to create a full-fledged Simula Implementation strictly after 'Simula Standard'.

From then on, I worked on a complete redesign of the Simula runtime system. The design is based on previous Simula implementations I have worked on, but this time some major changes have been made. Eg. a separate Garbage Collector is not necessary because the underlying Java system fixes that.

This document describes the runtime design of the new Portable Simula System.

Oslo, <dato>

Øystein Myhre Andersen

1. Innledning

Portable Simula Revisited

Runtime Organization

Innledning

Identifikasjon av program-elementer.

Med 'program-element' menes enhver størrelse som er deklarert eller spesifisert (parameter) i kildeteksten: variable, array, procedure, klasse, label og switch.

...

Entiteter eller Block Instances

Med entiteter mener vi program elementer som procedure, class, block, ..
Felles for disse er at di matrialiseres ved egne objekter ved utføringen av programmet.

Quantities

Felles for disse er at de brukes til å holde en 'verdi'

OM SIMULA's BLOCK STRUCTURE

- SubBlock
- Prfixed Block
- Class

Procedure

Hver av dem med kode layout ….

2. Simula's nested Block Structure

Each and every Simula Class, Prefixed Block, Procedure or Sub-Block is compiled into a Java Class. For example, consider the following Simula Class:

```
Class A(p1,p2); integer p1; real p2;
begin
  text Array TA(1:40);
  integer i;
  real r;
  i:=4;
  inner;
  r:=6;
end;
```

It will be compiled to a Java Class like:

```
public class A extends CLASS$ {
   // Declare parameters as attributes
   public int p1;
   public float p2;
   // Declare locals as attributes
   public $ARRAY<TXT$[]>TA=null;
   public int i=0;
   public float r=0.0f;

   // Normal Constructor
   public A(RTObject$ staticLink,int p1,float p2) {
      super(staticLink);
      // Parameter assignment to locals
      this.p1 = p1;
      this.p2 = p2;
      // Declaration Code
      int[] TA$LB={1}; int[] TA$UB={40};
      TA=new $ARRAY<TXT$[]>(new TXT$[TA$UB[0]-TA$LB[0]+1],TA$LB,TA$UB);

      // Create Class Body
      CODE$=new ClassBody(CODE$,this) {
         public void STM() {
            BBLK(); // Iff no prefix
            i=4;
            if(inner!=null) inner.STM();
            r=((float)(6));
            EBLK(); // Iff no prefix
      }};
   }

   // Class Statements
   public A STM() { return((A)CODE$.EXEC$()); }
   public A START() { START(this); return(this); }
}
```

Here we see that both parameters and local variables are declared as local members. The constructor consists of parameter transmission, Array declaration code and creation of concatenated statement code belonging to this Simula Class.

Two member methods to invoke the Statement Code is generated; one which START this Class in a new Thread and the other don't (STM). The Compiler decides which one to use dependent on the class being a 'component'.

Procedures however; follows a different scheme.

```
integer Procedure P(p1,p2)
  integer p1; real p2;
begin
  text Array TA(1:40);
  integer i;
  P:=p1;
  i:=6;
end;
```

No concatenation in this case but the procedure must be prepared to be called as a formal procedure without parameter specifications.

```
public class P extends RTObject$ {
   // Declare return value as attribute
   public int $result;
   public Object $result() { return($result); }
   // Declare parameters as attributes
   public int p1;
   public float p2;
   // Declare locals as attributes
   public $ARRAY<TXT$[]>TA=null;
   int i=0;

   // Parameter Transmission in case of Formal/Virtual Procedure Call
   private int $npar=0; // Number of actual parameters transmitted.
   public P setPar(Object param)
   { try { switch($npar++) {
          case 0: p1=intValue(param); break;
          case 1: p2=floatValue(param); break;
          default: throw new RuntimeException("Wrong number of parameters");
     } } catch(ClassCastException e) {
         throw new RuntimeException("Wrong type of parameter: "+param,e);}
     return(this);
   }

   // Constructor in case of Formal/Virtual Procedure Call
   public P(RTObject$ staticLink)
   { super(staticLink); }

   // Normal Constructor
   public P(RTObject$ staticLink,int p1,float p2) {
      super(staticLink);
      // Parameter assignment to locals
      this.p1 = p1;
      this.p2 = p2;
      // Declaration Code
      int[] TA$LB={1}; int[] TA$UB={40};
      TA=new $ARRAY<TXT$[]>(new TXT$[TA$UB[0]-TA$LB[0]+1],TA$LB,TA$UB);
      STM();
   }

   // Procedure Statements
   public P STM() {
      BBLK();
      $result=p1;
      i=6;
      EBLK();
      return(this);
   }
}
```

A normal procedure call *i:=P(4,3.14);* will generate code look like:

```
i = new P(PRG$,4,(float)3.14).$result;
```

A call on a formal Procedure; however; will look quite different. Suppose that F is a procedure quantity representing a formal parameter procedure. And P is the actual parameter. Then a call like *i:=P(4,3.14);* will generate code like this:

```
i = (int)F.CPF().setPar(new $NAME<Integer>() {
      public Integer get() {
            return (4);
      }
}).setPar(new $NAME<Double>() {
      public Double get() {
            return (3.14);
      }
}).STM().$result();
```

Block Instances at runtime.

During execution, program elements like procedure, class, block, etc.
are represented by objects of subclasses of RTObject.

The following attributes are defined:

`OperationalState STATE$`; One of { ***detached***,***resumed***,***attached***,***terminated*** }

boolean isQPSystemBlock()
This is a static property generated by the compiler.
It will return true for Block or Prefixed Block with local classes.

boolean `isDetachable()`
This is a static property generated by the compiler.
It will return true for Classes which can be Detached.

`RTObject$ SL$` This is a pointer to the object of the nearest textually enclosing block instance, also called 'static link'.

`RTObject$ DL$` If this block instance is attached this is a pointer to the object of the block instance to which the instance is attached (also called dynamic link), i.e. it points to the block instance which called this one.

`int` `BL$` Static block level.

`ClassBody CODE$` This is an object containing the concatenated statement code for Classes and Prefixed blocks.

`Thread THREAD$` This is a pointer to the Thread in which this block instance is running. If this block instance is detached it is used as the program part of the reactivation point.

The figure on next page shows the main structure of nested block instances at runtime.
Ole Johan dahl had a simular figure in a compendium in 1980.

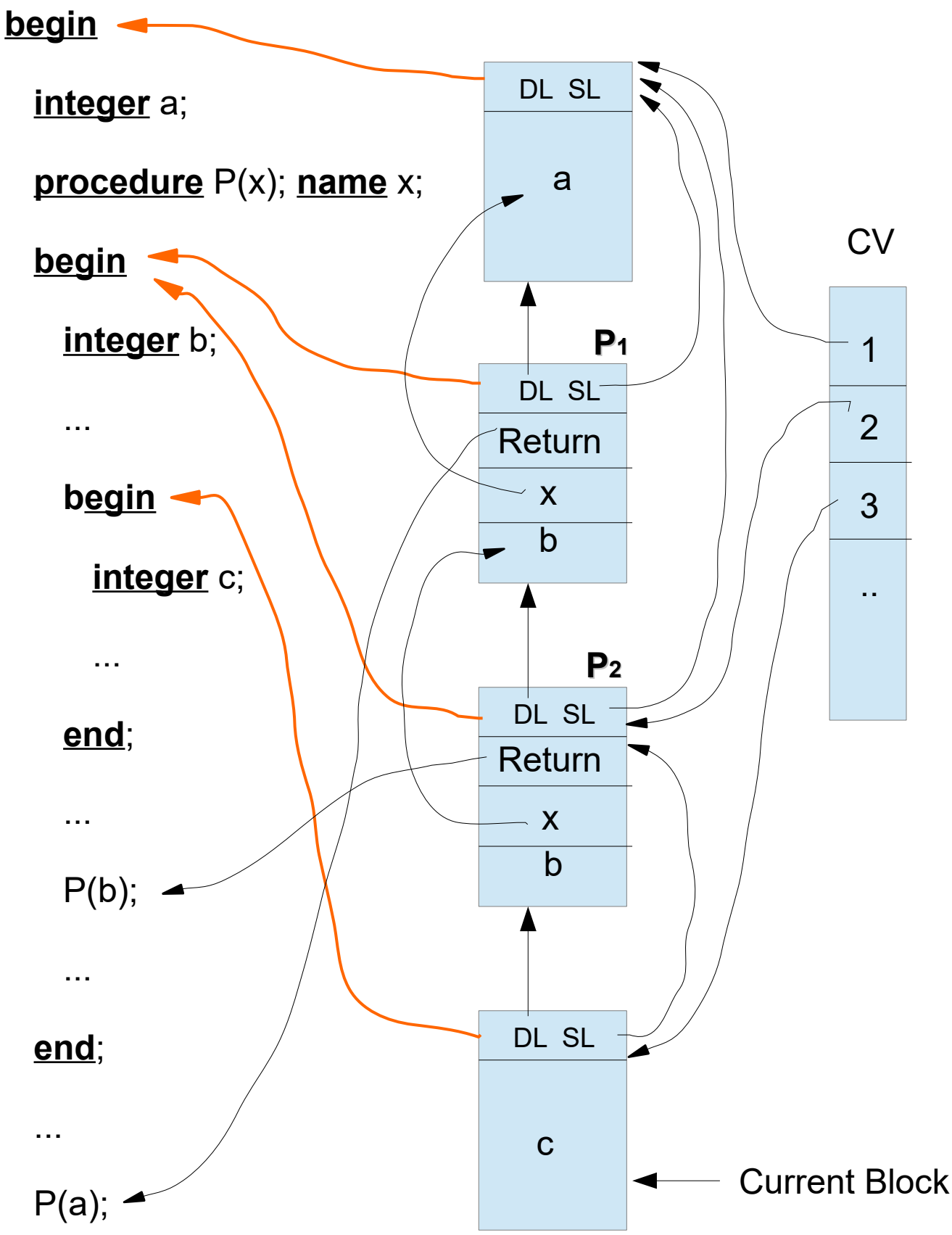


Fig. X Sample Block Structure

(due to Ole-Johan Dahl)

Håndtering av Procedyrer

- Parameterløs
- Procedure med parametre
- Formelle/Virtuelle Procedure-kall

3. Representation of Quantities.

Within an object, each quantity local to that object is represented. Local quantities are e.g. Declared variables in a block (sub-block, procedure body or prefixed block), or the attribute of a class. Quantities may be of value types, object reference type, text descriptors, array quantities, procedure quantities, label quantities or switch quantities..

3.1 Value type variables.

A variable of value type is represented in Java as follows:

Boolean	boolean
Character	char
Short integer	int *)
Integer	int
Real	float
Long real	double

*) Simula Standard 2.1.1 states; " An implementation may choose to implement short integer exactly as integer, i.e. ignoring the keyword short "

3. 2 Object Reference Types.

Object reference variables are represented in Java as such.

ref(ClassIdentifier)	ClassIdentifier
none	**null**

3. 3 Text Reference Types.

Text reference variables is a composite structure representated by a TXTREF Java Object.

Text	TXT$

where TXT$ has the following definition:

```
public class TXT$ extends RTObject {
   TEXTOBJ OBJ; // Reference to the text object.
   int START;   // Start index of OBJ.MAIN[], counting from zero.
                // Note this differ from Simula Definition.
   int LENGTH;
   int POS;     // Current index of OBJ.MAIN[], counting from zero.
                // Note this differ from Simula Definition.
   ...
}
```

The special text constant **notext** is represented by Java **null**.

The object TEXTOBJ is defined as:

```
public class TEXTOBJ extends RTObject {
   public int SIZE;        // Number of characters in the text object.
   public boolean CONST;   // True: Indicates a text constant
   char[] MAIN;            // The characters
   ...
}
```

3.4 Array Quantities.

A local array is represented by a reference to an array object:

```
public class $ARRAY<T>
{ public T Elt;
  public int[] LB;
  public int[] UB;
  public $ARRAY(T Elt,int[] LB,int[] UB)
  { this.Elt=Elt; this.LB=LB; this.UB=UB; }
  public $ARRAY<T> COPY() { … }
}
```

Example: A one-dimesional *Integer Array Table(6:56)* is declared by:

```
$ARRAY<int[]> Table;

int[] Table$LB={6};
int[] Table$UB={56};
Table=new $ARRAY<int[]>(new int[51],Table$LB,Table$UB);
```

And we acess elements like this:

```
int x=45; // Index
int i=Table.Elt[x-Table.LB[1]];  //  i := Table(x);
Table.Elt[x-Table.LB[1]]=i;      //  Table(x) := i;
```

And, two-dimesional *Text Array TA(6:56,8:88)* is coded like:

```
$ARRAY<TXT$[][]> TA;
int[] LB = {6,8};       NOTE: If expression – evaluation in wrong order
int[] UB = {56,88};     NOTE: If expression – evaluation in wrong order
TA=new $ARRAY<TXT$[][]>(new TXT$[51][81],LB,UB);

int x=45; // Index
int y=45; // Index
TXT$ t=TA.Elt[x-TA.LB[1]][y-TA.LB[2]];  //  t :- AI(x,y);
TA.Elt[x-TA.LB[1]][y-TA.LB[2]]=t;       //  AI(x,y) :- t;
```

And so on as many dimensions as you like.

A parameter Array is represented by the $ARRAY<?> quantity.

NOTE: Multi-dimensional Parameter Arrays are not yet Implemented. It is supposed that we can use reflection to Complete the implementation

3.5 Procedure Quantities.

A procedure is represented as a composite structure as shown below. Procedure quantities are used to represent formal procedure parameters. Local procedures, however, are not represented as quantities whitin an object.

```
public class $PRCQNT
{ RTObject$ staticLink;
  Class<?> procedure;

  // Constructor
  public $PRCQNT(RTObject$ staticLink,Class<?> procedure)
  { this.staticLink=staticLink; this.procedure=procedure; }

  public RTObject$ CPF()
  { try
    { Constructor<?> constr = procedure.getConstructor(new Class[]{RTObject$.class});
      Object obj=constr.newInstance(staticLink);
      return((RTObject$)obj);
    }
    catch(Throwable e) { throw new RuntimeException("Internal Error",e);     }
  }

}
```

The attribute 'staticLink' is a pointer to the block-instance in which the procedure is declared.

In traditional Simula implementatios, a prototype is generated to hold basic informations of block-instances, including procedures. In this implementation no Prototype is generated. Instead, we use Java's reflection mechanisms to create procedure object instances.

The atribute 'procedure' is a pointer to Java's 'prototype' i.e. a pointer to a Java Object containing all information of the class representing the Simula procedure.

The CPF method is used for generating call to formal or virtual procedures. Further details in a later section.

Suppose we have a Simula procedure 'P'. To create a procedure quantity at runtime we use a constructs like this:

```
new $PRCQNT(this,P.class);   // P is local
new $PRCQNT(CTX[n],P.class); // P is declared at block level n
```

3.6 Switch Quantities.

This implementation treats a Simula Switch as a $LABQNT Procedure.

3.7 Label Quantities. NOTE: Temporary Solytion – May be Reviced

Java does not support labels like Simula. The Java Virtual Machine (JVM), however, has labels. A JVM-label is simply a relative byte-address within the byte-code of a method.

This implementation will generate Java-code which is prepared for Byte Code Engineering. See chapter .....

Suppose a Simula program containing labels and goto's:

```
L: ...
 goto L;
```

This will be coded as:

```
L=new LABEL("L");
...
GOTO(L);
```

Where 'LABEL' is a Java-class:

```
public class LABEL
{ String ident;
  public int index; // I.e. ordinal number of the labeled statement.
  public LABEL(String ident) { this.ident=ident; }
}

public void GOTO(LABEL L) {} // Local GOTO - Needs ByteCode Engineering.
```

The identifier 'ident' is used to build a label-list which is used to create a table-switch at the beginning of methods representing Simula statements. The attribute 'index' is the index into this label-list.

A non-local label is represented as follows.

```
public class $LABQNT
{ RTObject SL; // Static link.
  LABEL L;     // Local Label.

  // Constructor
  public $LABQNT(RTObject staticLink,LABEL L)
  { this.SL=staticLink; this.L=L; }


  public void GOTO() // GOTO Non-Local Label
  { ... }


    }
```

The attribute 'SL' is the 'static link' i.e. a pointer to the block-instance in which the label is defined.

Suppose a Simula label defined in a block with block-level n. To create a label quantity at runtime we use a constructs like this:

```
new $LABQNT(this,L)                          // L is local
new $LABQNT(CTX[n],((BlockIdent)CTX[n]).L) // L defined at block level n
```

3.8 Parameter Transmission 'by Name'.

The basic principle is to establish an object within the calling scopet. This object will have two attributt methods; 'get' og 'put' to read the value of the actual parameter, or, if legal, to write into it. The following Java-class is used to perform such parameter transmissions:

```
abstract class $NAME<T> {
      abstract T get();
      void put(T x) { error("Illegal ..."); }
}
```

Note that we both use abstract Java classes and 'generics' i.e. the actual type is a parameter. Also note that the 'put' method has a default definition producing an error. This enables redefinition of the 'put' method to be droped for expression as actual parameters.

Suppose the Simula Procedure:

***procedure** P(k); **name** k; **integer** k; k:=k+1;*

It will be translated to something like this Java method:

```
void P($NAME<Integer> k) {
      k.put(k.get() + 1); // E.g: k=k+1
}
```

In the calling place, in practice in the actual parameter list, we create an object of a specific subclass of *$NAME<T>* by specifying the Integer type and defining the get and put methods. Eg. If the current parameter is a variable 'q', then the actual parameter will be coded as follows:

```
 new $NAME<Integer>() {
      Integer get() { return (q); }
      void put(Integer x) { q = (int) x;  }
}
```

However, if the actual parameter is an expression like (j + m * n) then it will be coded as follows:

```
new $NAME<Integer>() {
      Integer get() { return (j + m * n);       }
}
```

Here we see that the 'put' method is not redefined so that any attempt to assign a new value to this name parameter will result in an error message.

A classic example of name parameters is taken from Wikipedia's article about *Jensen's Device*. We change it slightly because Simula does not allow the control variable in a for-statement to be a formal parameter transferred by name.

```
long real procedure Sum(k, lower, upper, ak);
    value lower, upper; name k, ak;
integer k, lower, upper; long real ak;
        begin long real s;
                s := 0.0;
                k := lower;
                while k <= upper do
                begin
                        s := s + ak;
                        k := k + 1;
                end while;
                Sum := s
        end Sum;
```

Translated to Java, this will be:

```
public double Sum($NAME<Integer> k,
            int lower, int upper,
            $NAME<Double> ak) {
      public double s;
      s = ((double) (0.0));
      k.put(lower);
      while (k.get() <= upper) {
            s = s + ak.get();
            k.put(k.get() + 1);
      }
      return (s);
}
```

At procedure call like this:

```
integer i;
long real array A[0:99];
long real result;

resultat=Sum(i,10,60,A[i]);
```

Is translated to:

```
public int control;
public double[] A=new double[100];
public double result;


result = Sum(
  new $NAME<Integer>() {
      Integer get() {   return(i); }
      void put(Integer x) { i = (int)x; }
  },
  10,
  60,
  new $NAME<Double>() {
      Double get() { return (A[i]); }
      void put(Double x) { A[i] = (double) x; }
  });
```

Appendix 1: The complete Simula Code.

```
class JensensDevice;
begin
%  See: https://en.wikipedia.org/wiki/Jensen's_Device


long real procedure Sum(k, lower, upper, ak);
value lower, upper; name k,ak;
integer k, lower, upper; long real ak;
  begin
        long real s;
        s := 0.0;
        k := lower;
        while k <= upper do
        begin
              s := s + ak;
              k := k + 1;
        end while;
        Sum := s
  end Sum;

  integer i;
  long real array A[0:99];
  long real result;

  Sum(i,10,60,A[i]);

end JensensDevice;
```

Appendix 2: The generated Java Code

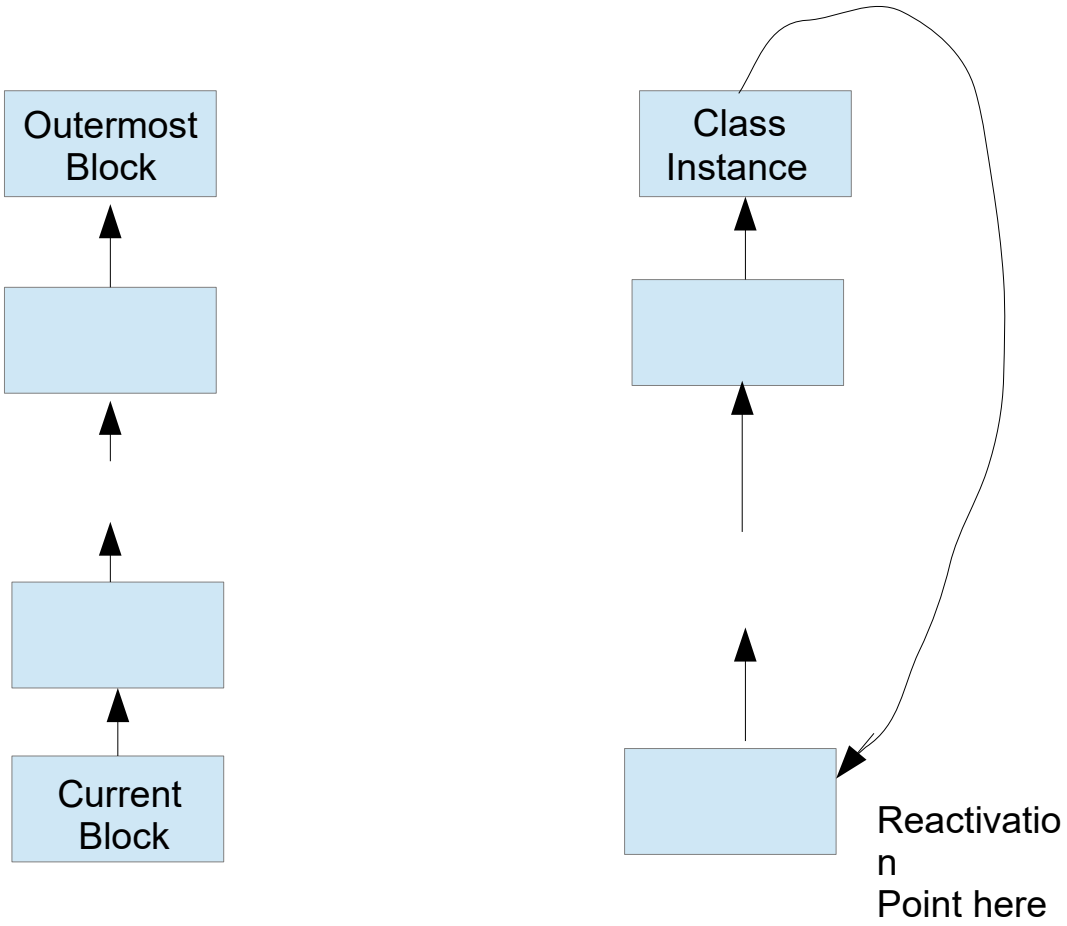
```
public class JensensDevice {
      public double Sum($NAME<Integer> k, int lower, int upper,
                  $NAME<Double> ak) {
                  double s;
                  s = ((double) (0.0));
                  k.put(lower);
                  while (k.get() <= upper) {
                        s = s + ak.get();
                        k.put(k.get() + 1);
                  }
                  return (s);
      }
      public int i;
      public double[] A[100];
      public double result;

      // Constructor
      public JensensDevice() {
            result = Sum(new $NAME<Integer>() {
                  Integer get() {   return(i); }
                  void put(Integer x) { i = (int)x; }
            }, 10, 60, new $NAME<Double>() {
                  Double get() { return (A[i]); }
                  void put(Double x) { A[i] = (double) x; }
            });
}     }
```

4. Quasiparalell Sequencing

Text fra Simula Standard

Kode og tegning av
- Detach
- Call
- Resume

Klassisk illustrering av Detach



Call Stack Chain

Detached Component

For å illustrere problematiken ved save/restore av operand-stakken under utføring QPS/Detach I JVM.

Anta følgende Simula Program:

```
System begin ref(C)  x;

    procedure P;
    begin  x:-new C();
              …
              call(x);
    end;

    class Component;
    begin
        procedure Q;
        begin ...  detach;  ...  end;
        ...
        Q;
        ...
    end Component;
    ...
    P;
    ...
end System;
```

De neste par sider kan blas raskt I fram og tilbake. Du får da en animering av effekten av Detach slik jeg først tenkte meg å Implementere det hele.

Legg merke til at Stack-Framen til Q blir frigjort mens framen til C blir hengende og sansynligvis bryter reglene for bruk av stacken.

Det fins løsning på dette ved at ikke bare STM-koden til Q termineres Men hele stacken tilbake til nærmeste block som er et komponent hode, I vårt tilfelle instansen av klassen Component.

Dette blir så pirkete og komplisert at jeg foreslår å vente og se hva 'Project Loom' kommer opp med.

Example Detach – **Call** – Detach

Denne gangen med separat Stack
Knyttet til Componentene

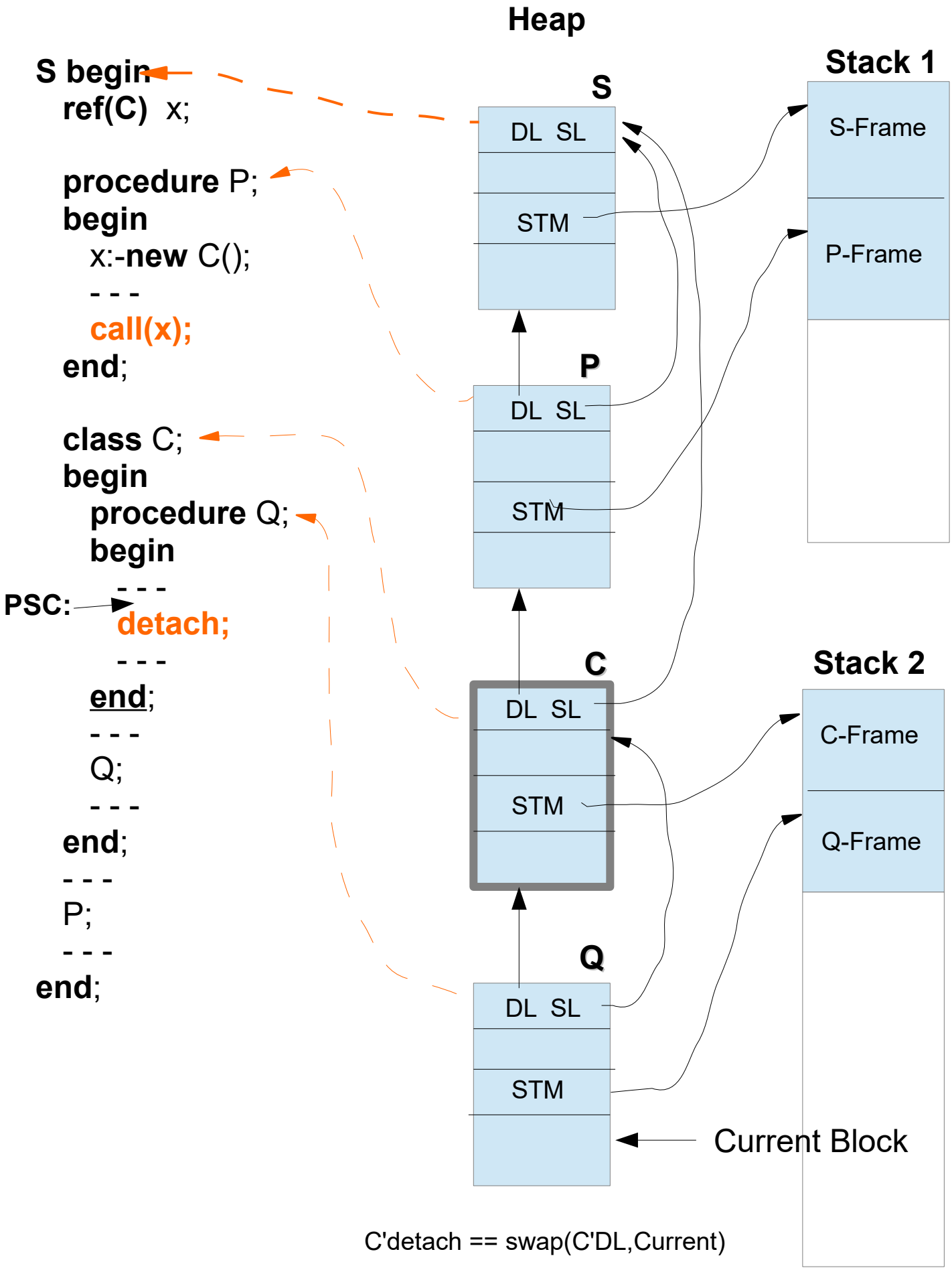


Fig. 5 Separat Stack: Just before Detach

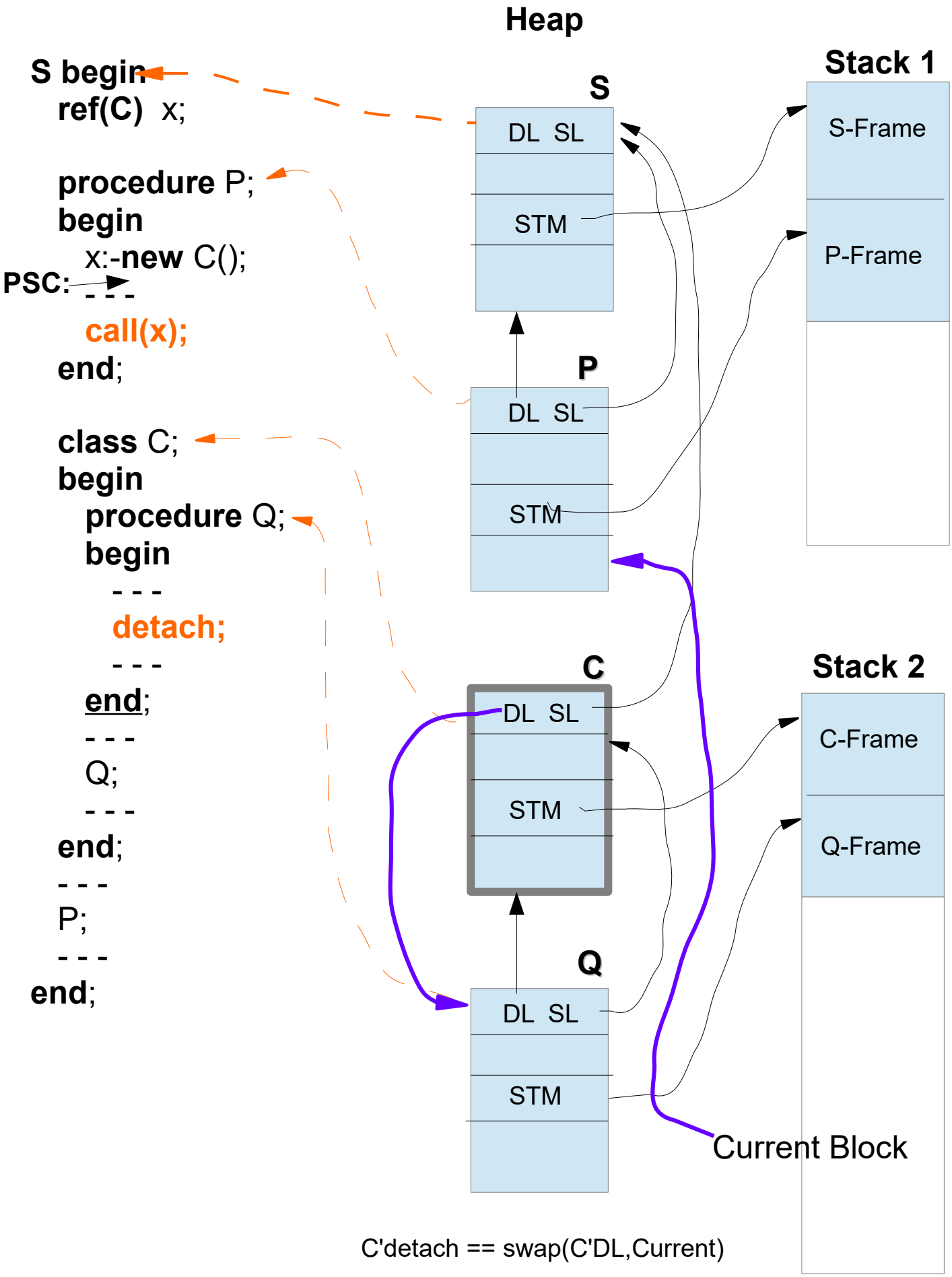


Fig. 6 Just after Detach

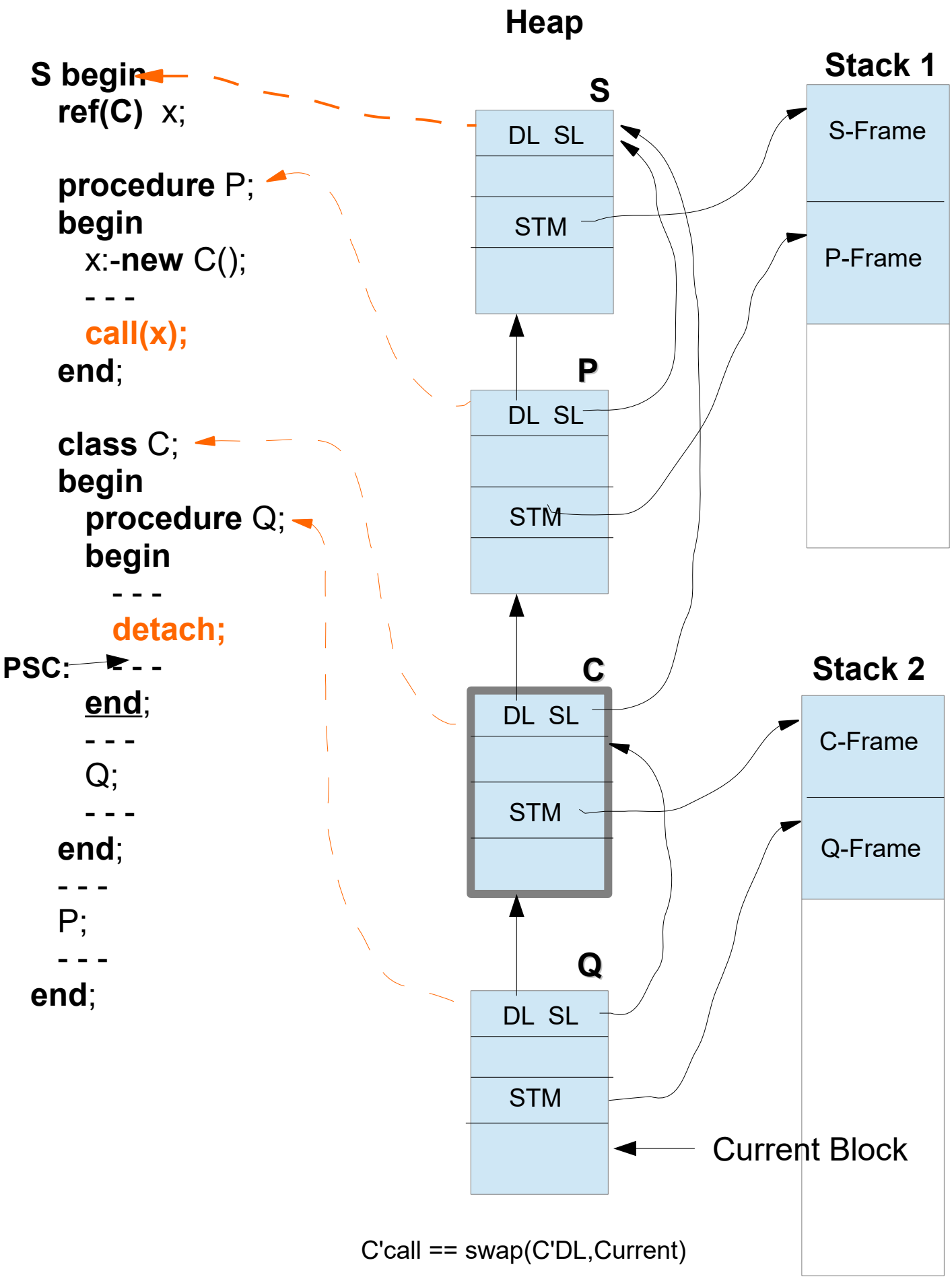


Fig. 7 Just after Call

Example Detach – **Resume** – Detach

Med separate Stack'er

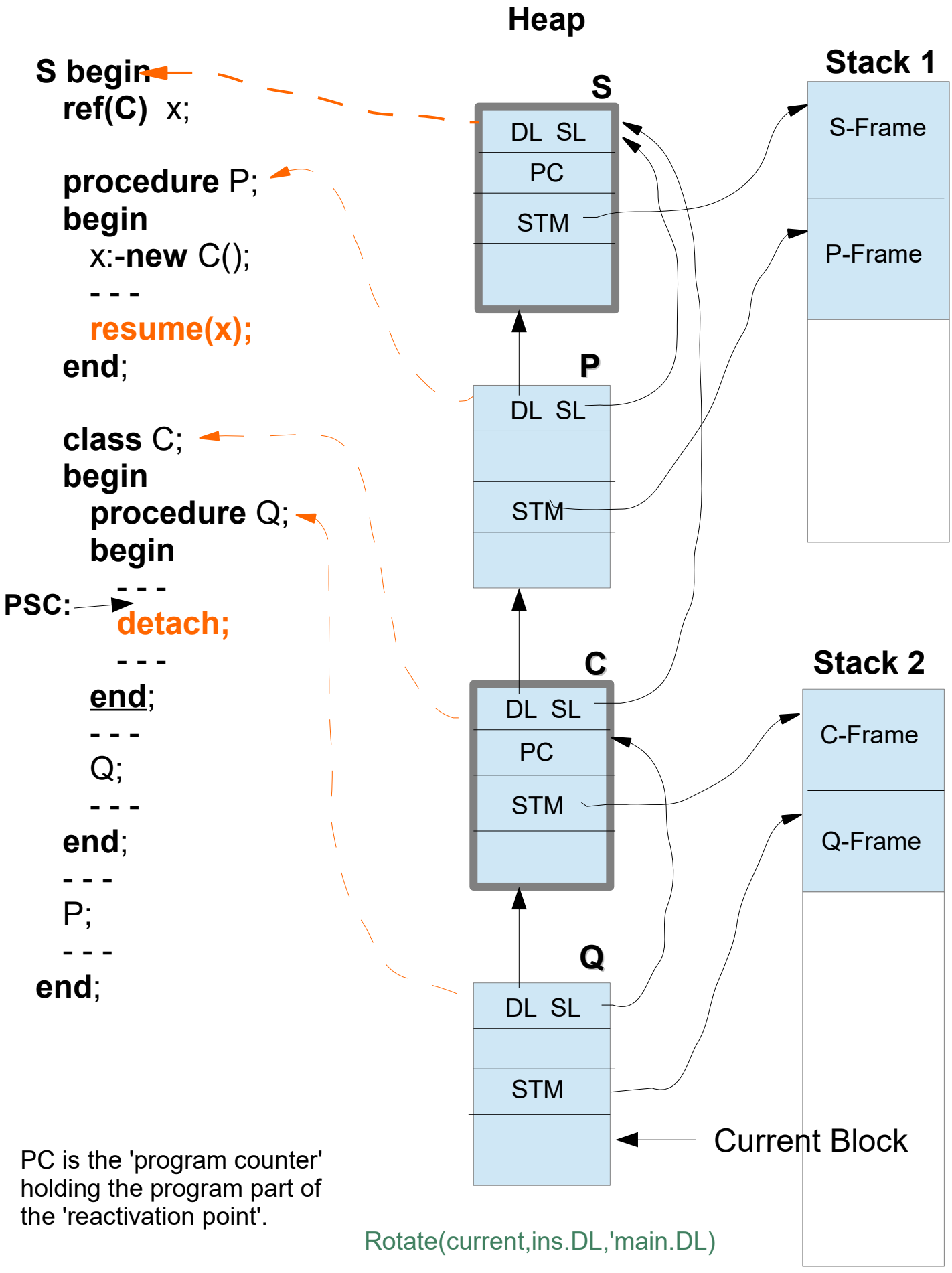


Fig. 8 Separat Stack: Just before 1.Detach

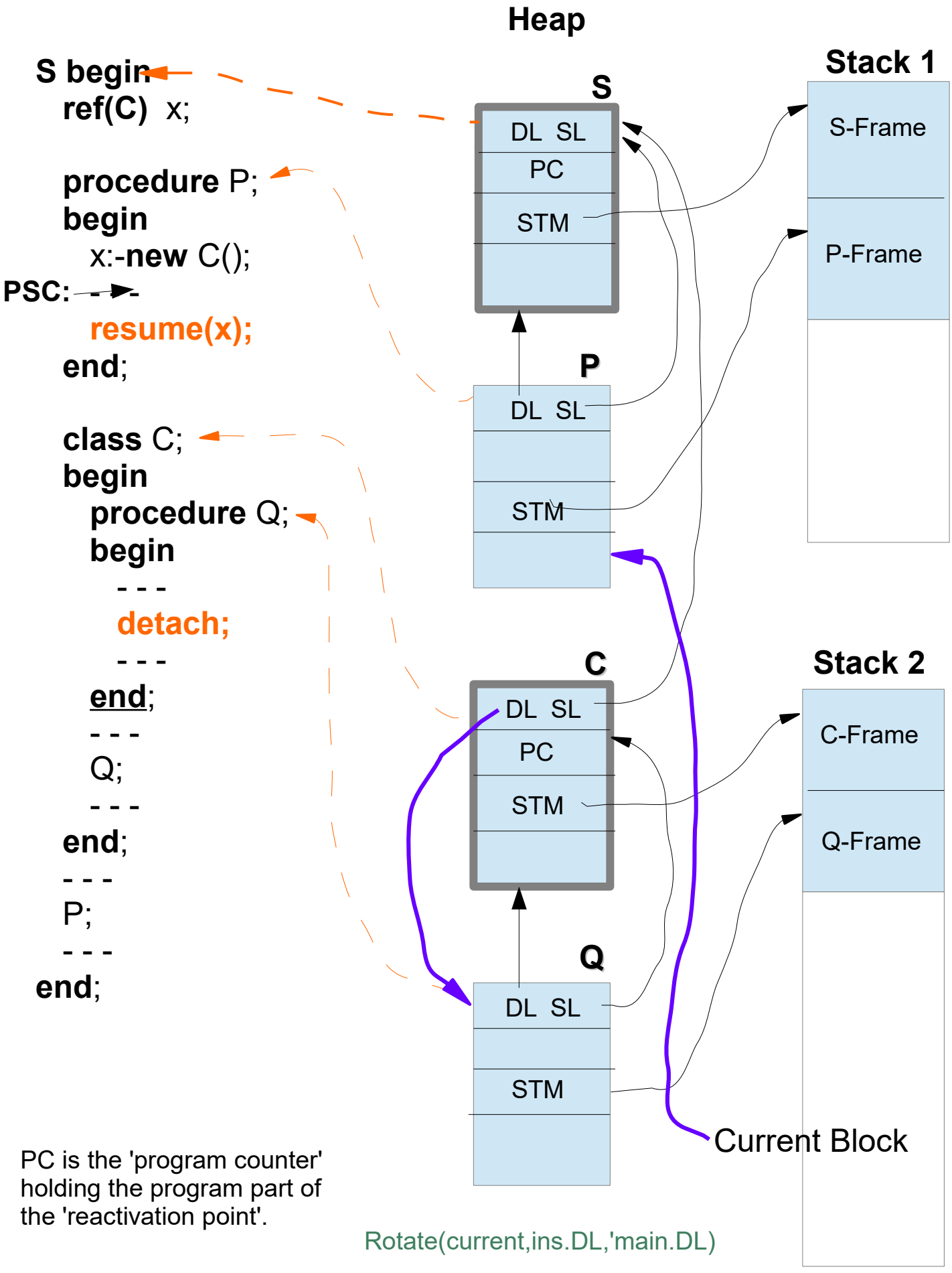


Fig. 9 Just after 1.Detach and before Resume

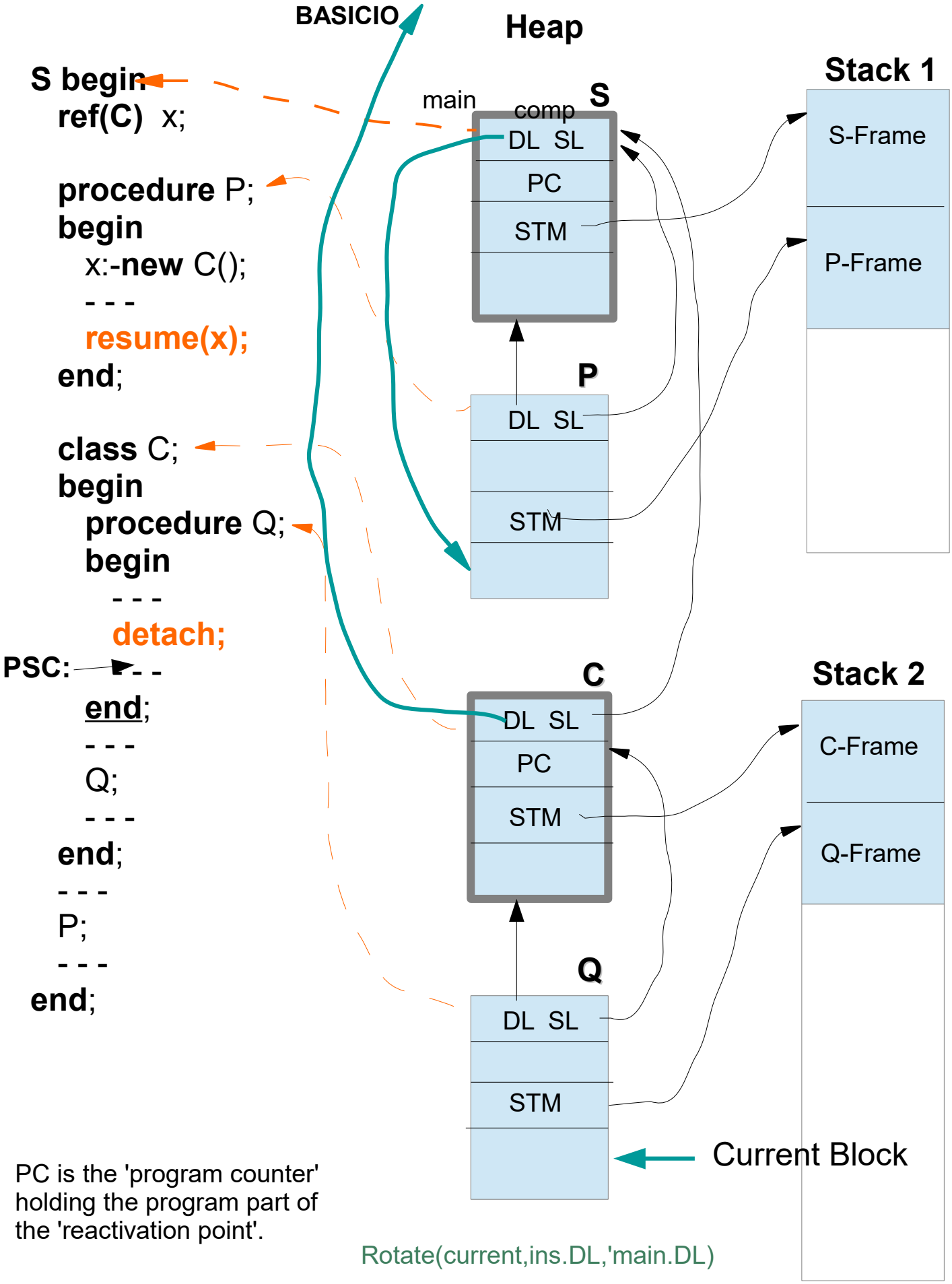


Fig. 10 Just after Resume

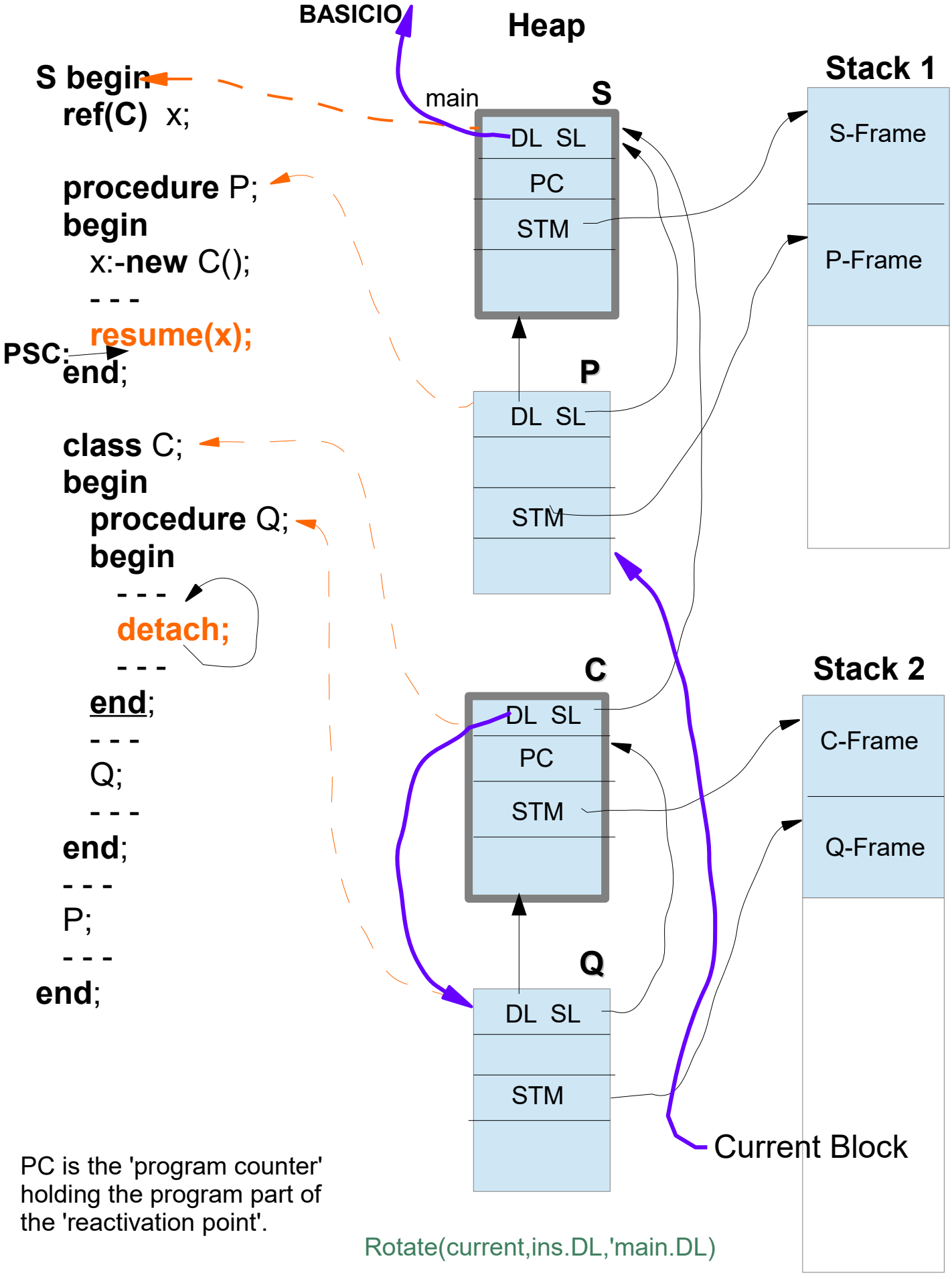


Fig. 11 Just after 2. Detach

```
public void detach()
{ RTObject$ ins;   //  Instance to be detached.
  RTObject$ dl;    //  Temporary reference to dynamical enclosure.
  RTObject$ main;  //  The head of the main component and also
                   //  the head of the quasi-parallel system.
  ins=this;

  //  Detach on a prefixed block is a no-operation.
  if(!CUR$.isQPSystemBlock())
  { // Make sure that the object is on the operating chain.
       // Note that a detached or terminated object cannot be on
       // the operating chain.
       dl=CUR$;
       while(dl!=ins) {
           dl=dl.DL$;
           if(dl==null) throw
             new RuntimeException("x.Detach: x is not on the operating chain.");
       }
       // Treat the case resumed and operating first,
       // because it is probably the dynamically most common.
       if(ins.STATE$==OperationalState.resumed) {
           ins.STATE$=OperationalState.detached;
           // Find main component for component to be detached. The main
           // component head must be the static enclosure of the object,
           // since the object has OperationalState.resumed.
           main=ins.SL$;
           // Rotate the contents of 'CUR$', 'ins.DL$' and 'main.DL$'.
           // <main.DL$,ins.DL$,CUR$>:=<ins.DL$,CUR$,main.DL$>
           dl=main.DL$; main.DL$=ins.DL$; ins.DL$=CUR$; CUR$=dl;
       } else {
           ins.STATE$=OperationalState.detached;
           // Swap the contents of object's 'DL$' and 'CUR$'.
           // <ins.DL$,CUR$>:=<CUR$,ins.DL$>
           dl=ins.DL$; ins.DL$=CUR$; CUR$=dl;
       }
       Util.ASSERT(CUR$.DL$!=CUR$,"Invariant");
       updateContextVector();
     swapThreads(ins.DL$);
  }
}
```

```
public void call(RTObject$ ins)
{ RTObject$ dl;     //  Temporary reference to dynamic enclosure.
  if(ins==null) throw new RuntimeException("Call(x): x is none.");
  if(ins.STATE$!=OperationalState.detached) throw
        new RuntimeException("Call(x): x is not in detached state.");
  // The object to be attached cannot be on the operating chain,
  // because then its state would have been resumed and not detached.

  // Swap the contents of 'CUR$' and object's 'dl'.
  //  <ins.DL$,CUR$>:=<CUR$,ins.DL$>;
  dl=ins.DL$; ins.DL$=CUR$; CUR$=dl;
  // From now on the object is in attached state.
  // It is no longer a component head.
  ins.STATE$=OperationalState.attached;
  updateContextVector();
  swapThreads(ins.DL$);
}
```

```
public void resume(RTObject$ ins)
{ RTObject$ comp;   //  Component head.
  RTObject$ mainSL; //  Static enclosure of main component head.
  RTObject$ main;   //  The head of the main component and also
                    //  the head of the quasi-parallel system.
  if(ins==null) throw new RuntimeException("Resume(x): x is none.");

  if(ins.STATE$!=OperationalState.resumed) // A no-operation?
  { // The object to be resumed must be local to a system head.
    main=ins.SL$;
    if(!main.isQPSystemBlock()) throw
           new RuntimeException("Resume(x): x is not local"
                                +" to sub-block or prefixed block.");

    if(ins.STATE$!=OperationalState.detached) throw
         new RuntimeException("Resume(x): x is not in detached state"
                              +" but "+ins.STATE$);
    // Find the operating component of the quasi-parallel system.
    comp=CUR$; mainSL=main.SL$;
    while(comp.DL$!=mainSL)
    { Util.ASSERT(comp.DL$!=comp,"Invariant");
      comp=comp.DL$;
    }
    if(comp.STATE$==OperationalState.resumed)
             comp.STATE$=OperationalState.detached;
    // Rotate the contents of 'ins.dl', 'comp.dl' and 'CUR$'.
    // Invariant:   comp.DL$ = mainSL
    //  <ins.DL$,comp.DL$,CUR$>=<comp.DL$,CUR$,ins.DL$>
    comp.DL$=CUR$; CUR$=ins.DL$; ins.DL$=mainSL;
    ins.STATE$=OperationalState.resumed;
      updateContextVector();
    swapThreads(comp.DL$);
  }
}
```

```
public void EBLK()
  { RTObject$ dl;   // A temporary to the instance dynamically
                    // enclosing the resumed object ('CUR$').
    RTObject$ main; // The head of the main component and also
                    // the head of the quasi-parallel system.

    // Treat the attached case first, it is probably most common.
    if(CUR$.STATE$==OperationalState.attached) {
       CUR$.STATE$=OperationalState.terminated;
      //  Make the dynamic enclosure the new current instance.
      CUR$=CUR$.DL$;
    } else if(CUR$.STATE$==OperationalState.resumed) {
      // Treat the case of a resumed and operating object.
      //  It is the head of an object component. The class
      //  object enters the terminated state, and the object component
      //  disappears from its system. The main component of that system
      //  takes its place as the operating component of the system.
      //  Invariant:  CUR$.STATE$ = resumed  and  CUR$.DL = main.SL
      CUR$.STATE$=OperationalState.terminated;
      //  Find main component (and system) head. It must be the static
      //  enclosure since the object has been RESUMEd.
      main=CUR$.SL$;
      //  The main component becomes the operating component.
      dl=CUR$.DL$; CUR$.DL$=null;
      CUR$=main.DL$; main.DL$=dl;
    }
    if(CUR$==null || CUR$==CTX$) {
       SYSOUT$.outimage();
      // PROGRAM PASSES THROUGH FINAL END
      System.exit(0);
    } else {
       updateContextVector();
      if(this.THREAD$!=CUR$.THREAD$)
      { if(QPS_TRACING) TRACE("Resume "+CUR$.THREAD$);
        CUR$.THREAD$.resume();
        if(QPS_TRACING) TRACE("Terminate "+this.THREAD$);
        this.THREAD$=null; // Leave it to the GarbageCollector
      }
    }
  }
```

5. Byte Code Engineering.

xxxxx

Goto Local Label – Bytekode Mekking (GROV SKISSE)

En samling lokale labler i et objekt fører til at 'statements' metoden endres slik at den starter med en hopp-tabell. Hver entry er et hopp til en lokal label:

```
0: getfield      #17  // Field lsc
3: tableswitch   {    // 0 to n
               0: 32
               1: 58
                  ...
               n: 174
         default: 274
       }
```

Variabelen 'lsc' er attributt i Objektet og settes til en passende Label-index forut for metode-kallet.

Et lokal goto-statement vil bli kodet til en slik Java-kode:

```
GOTO($M); // GOTO LOCAL LABEL
```

Som fører til denne byte-koden:

```
123:  getfield        $M
126:  invokevirtual   GOTO
```

Dette blir gjenkjent og erstattet med:

```
123:  goto  #58
```

En label-definition M: vil bli kodet til slik Java-kode:

```
$M=new LABEL("M");
```

Som fører til denne byte-koden:

```
61:  new          <RTObject$LABEL>
64:  dup
65:  aload_0
66:  ldc          "M"
68:  invokespecial  RTObject$LABEL.<init>
71:  putfield       $M
```

Som blir fjernet. Vi bare 'husker' at adressen '61' er label M.