From Functional to Concurrent Constraint Programming in 4 Easy Steps

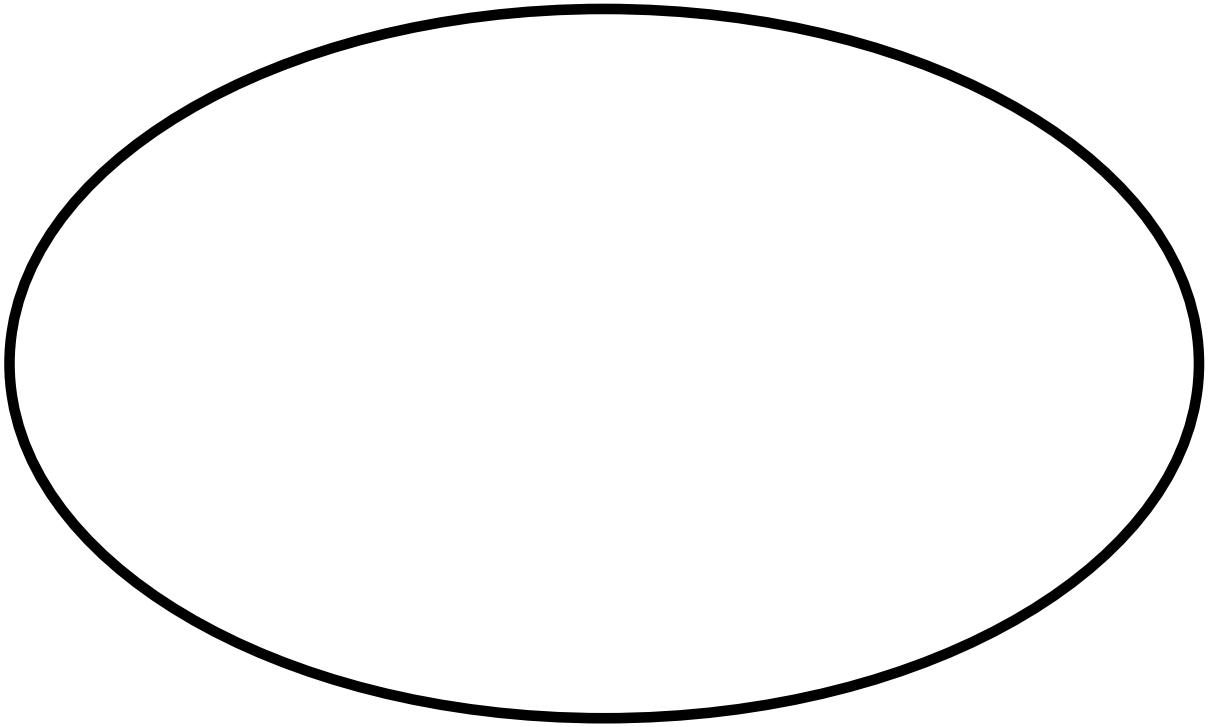
Martin Henz, DISCS

Eager Functional Programming

```
let val xs = 1 :: 2 :: 3 :: [];
    val ys = 4 :: 5 :: 6 :: [];
    val rec append = fn (vs,ws) =>
                        case vs
                        of [] => ws
                         | v :: vr => v :: append(v,ws)
in
    append(xs,ys)
end
```

```
let val xs = 1 :: 2 :: 3 :: [];
    val ys = 4 :: 5 :: 6 :: [];
    val rec append = fn (vs,ws) =>
                        case vs
                        of [] => ws
                         | v :: vr => v :: append(v,ws)
in
    append(xs,ys)
end
```

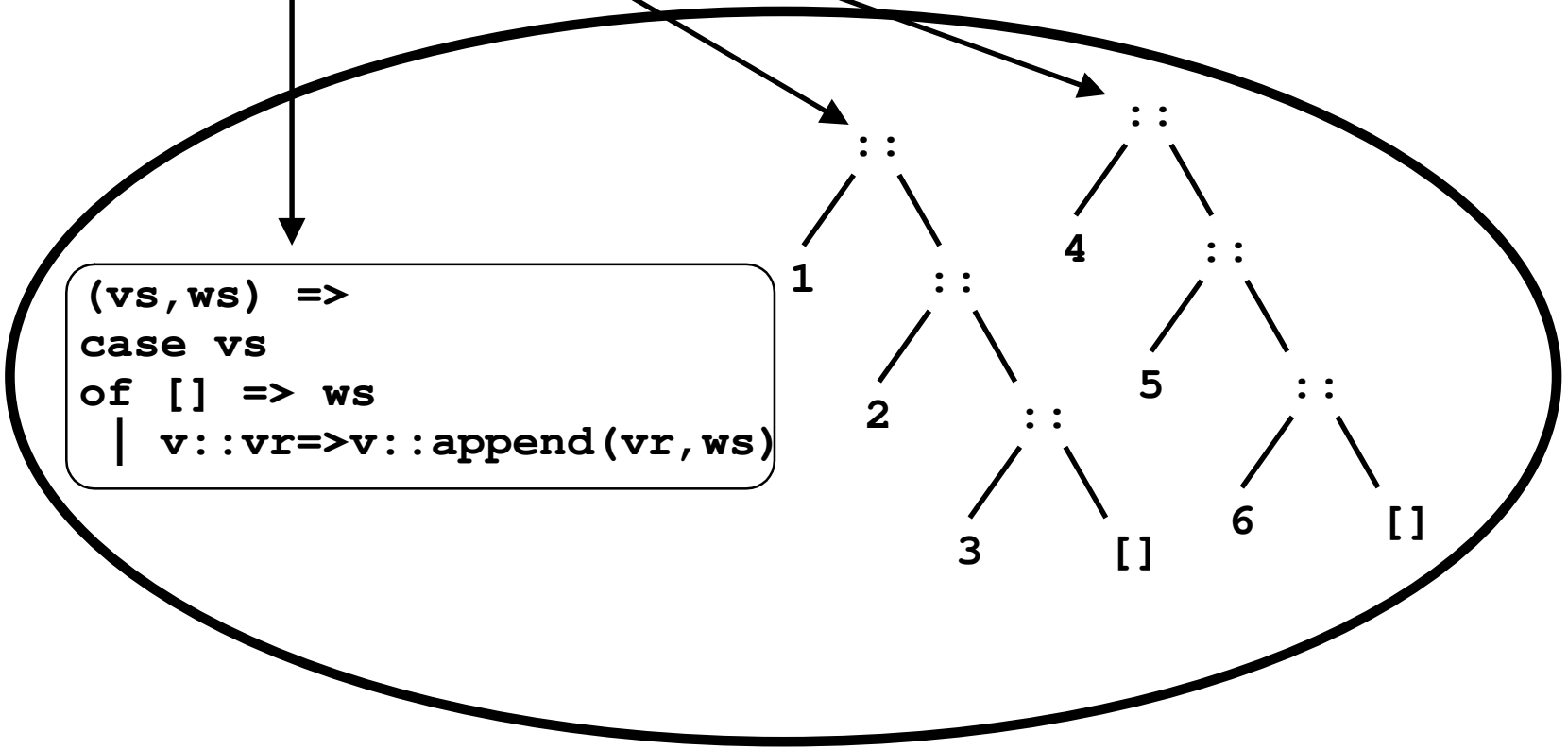
thread



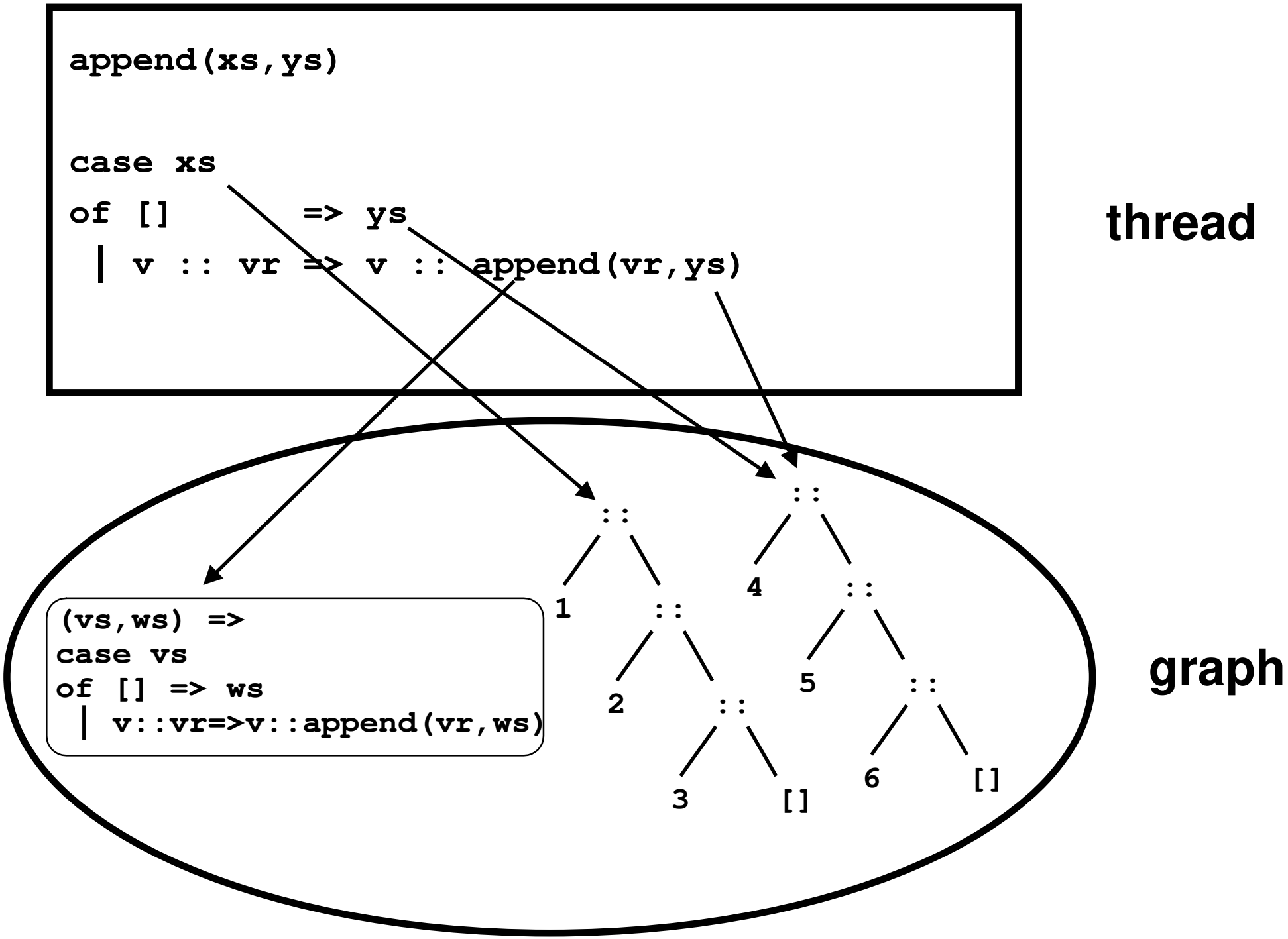
graph

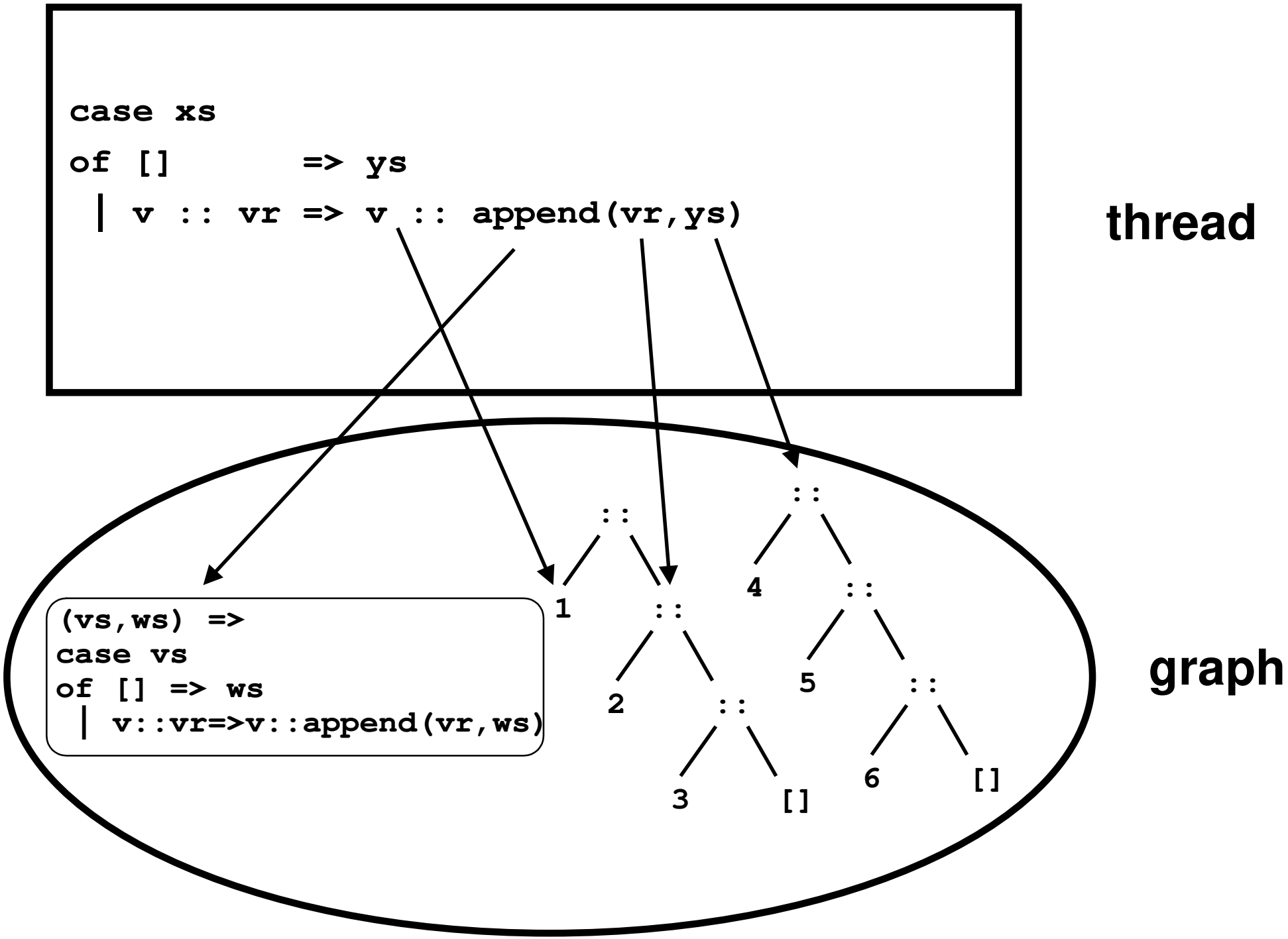
```
let val xs = 1 :: 2 :: 3 :: [];
    val ys = 4 :: 5 :: 6 :: [];
    val rec append = fn (vs,ws) =>
                        case vs
                        of [] => ws
                         | v :: vr => v :: append(v,ws)
in
    append(xs,ys)
end
```

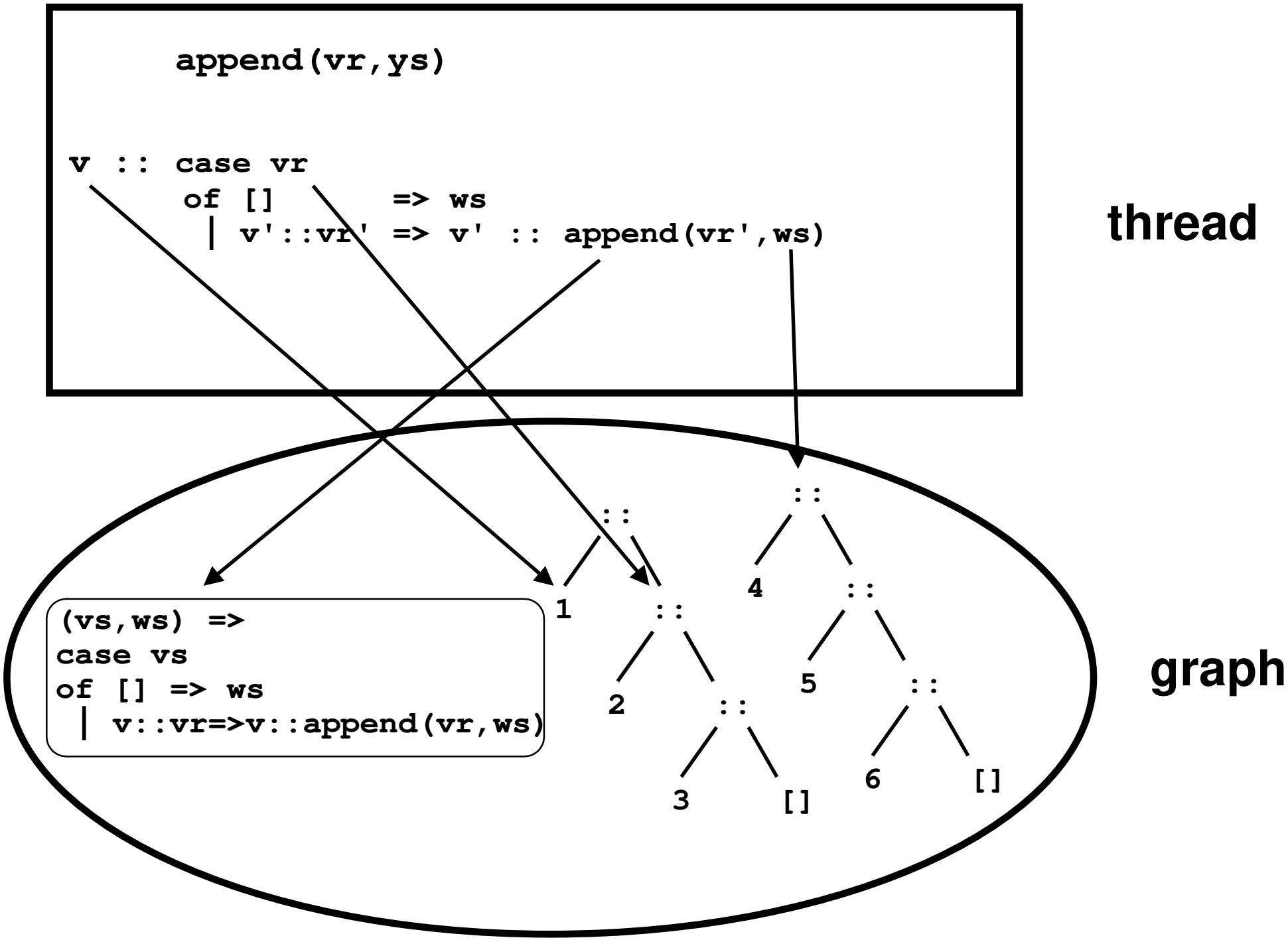
thread

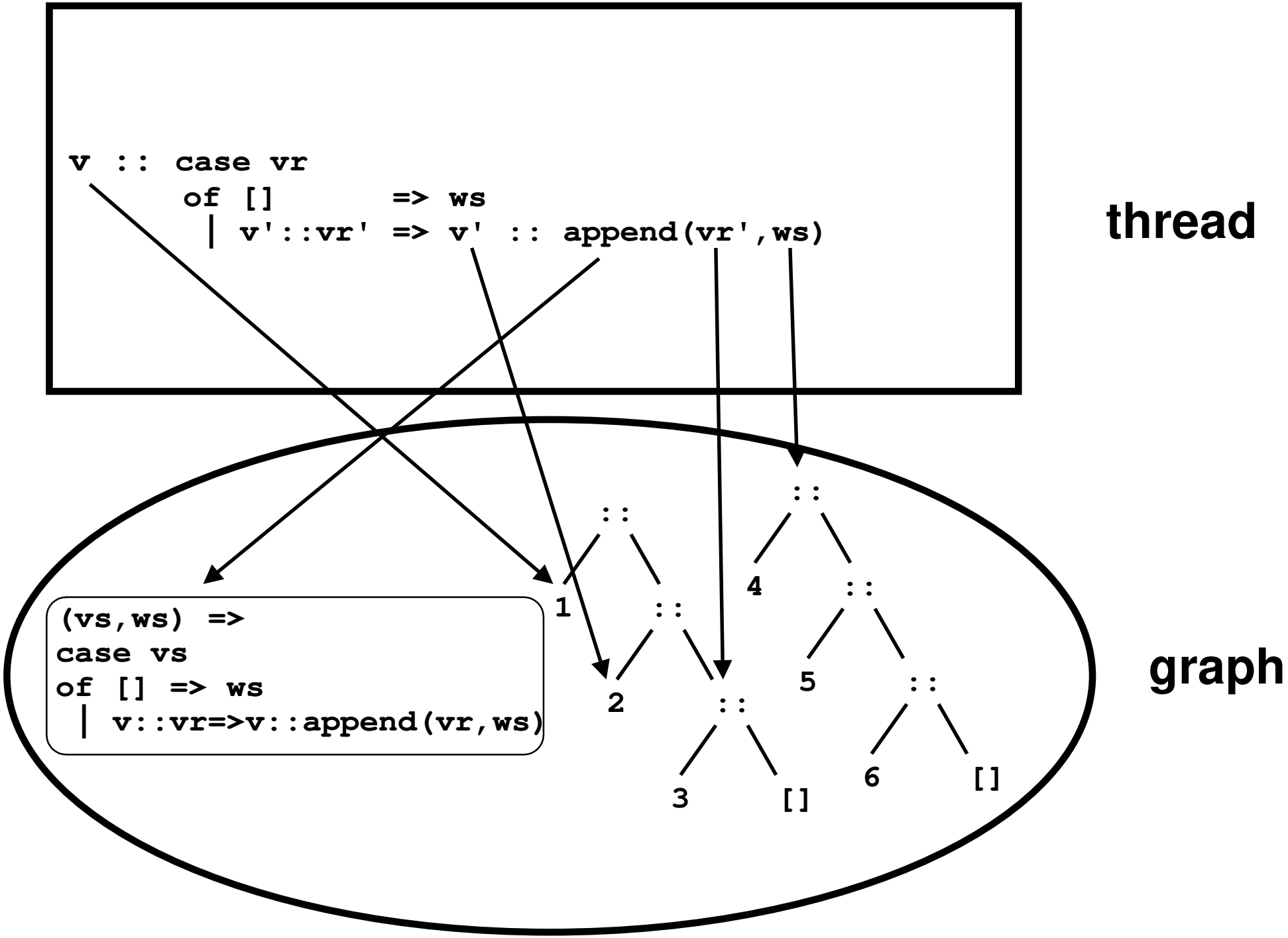


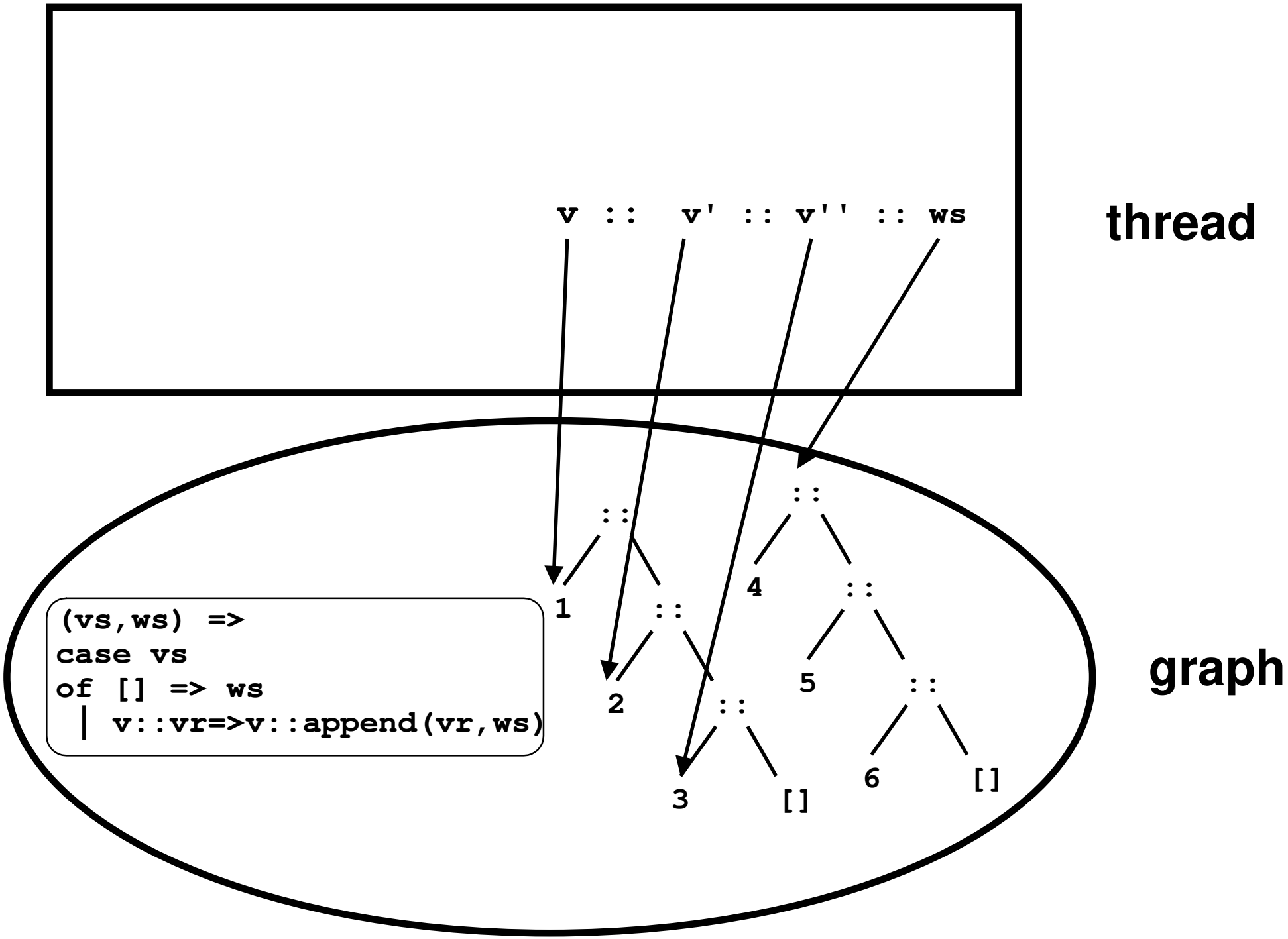
graph


append(xs,ys)
case xs
of []
=> ys
| v :: vr => v :: append(vr,ys)
thread
(vs,ws) =>
case vs
of [] => ws
| v::vr=>v::append(vr,ws)
::
::
::
1
4
::
::
2
5
::
::
3
[]
6
[]
graph

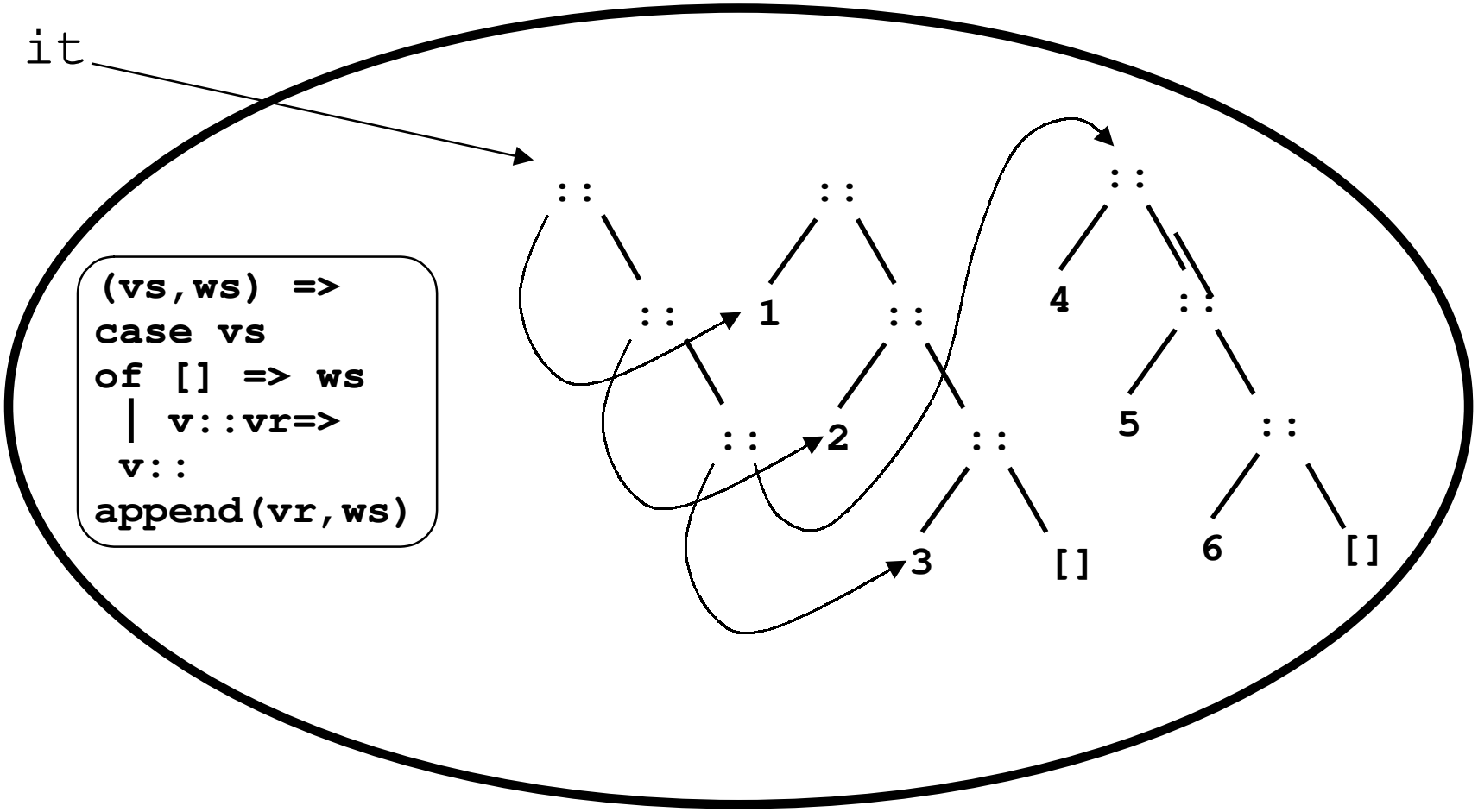

case xs
of []   => ys
 | v :: vr => v :: append(vr,ys)
thread
(vs,ws) =>
case vs
of [] => ws
 | v::vr=>v::append(vr,ws)
::
1
::
2
::
3
[]
::
4
::
5
::
6
[]
graph


append(vr,ys)
v :: case vr
of []
=> ws
| v'::vr' => v' :: append(vr',ws)
thread
(vs,ws) =>
case vs
of [] => ws
| v::vr=>v::append(vr,ws)
::
1
::
2
::
3
[]
::
4
::
5
::
6
[]
graph


v :: case vr
of []
=> ws
| v'::vr'
=> v' :: append(vr',ws)
thread
(vs,ws) =>
case vs
of [] => ws
| v::vr=>v::append(vr,ws)
graph
::
1
::
2
::
3
[]
::
4
::
5
::
6
[]


v :: v' :: v'' :: ws
thread
(vs,ws) =>
case vs
of [] => ws
| v::vr=>v::append(vr,ws)
::
1
::
2
::
3
[]
::
4
::
5
::
6
[]
graph

thread



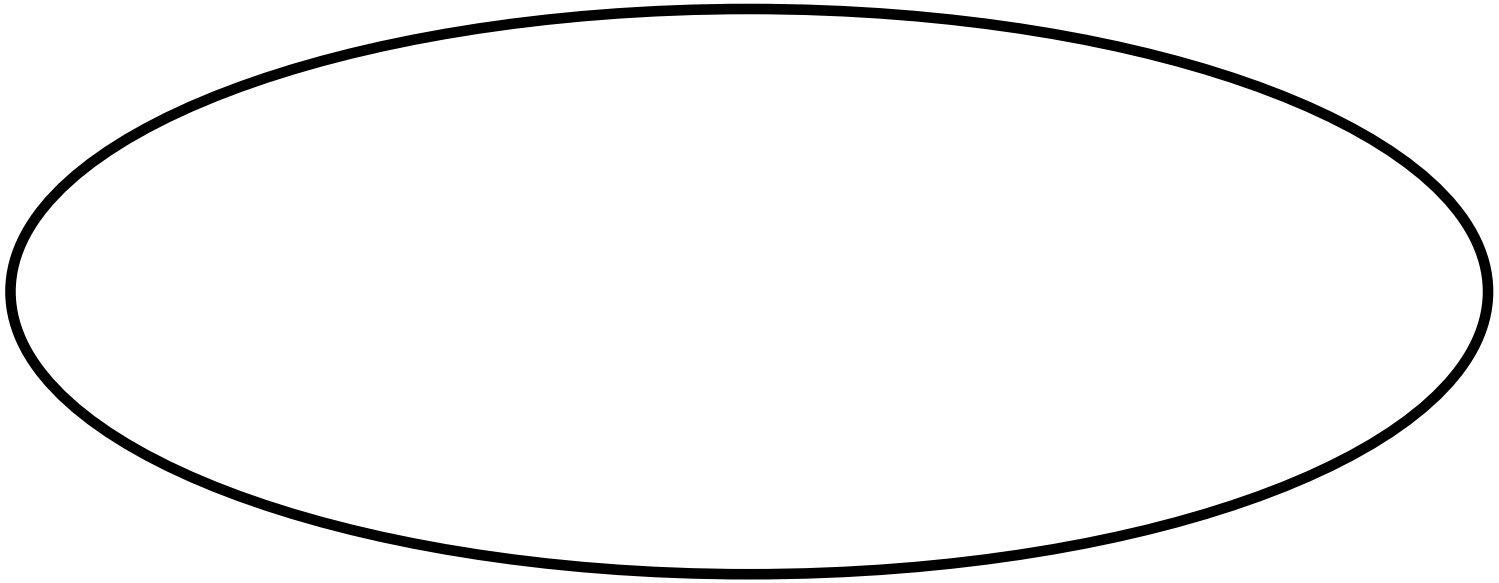
graph

Step 1: Explicit Return Arguments

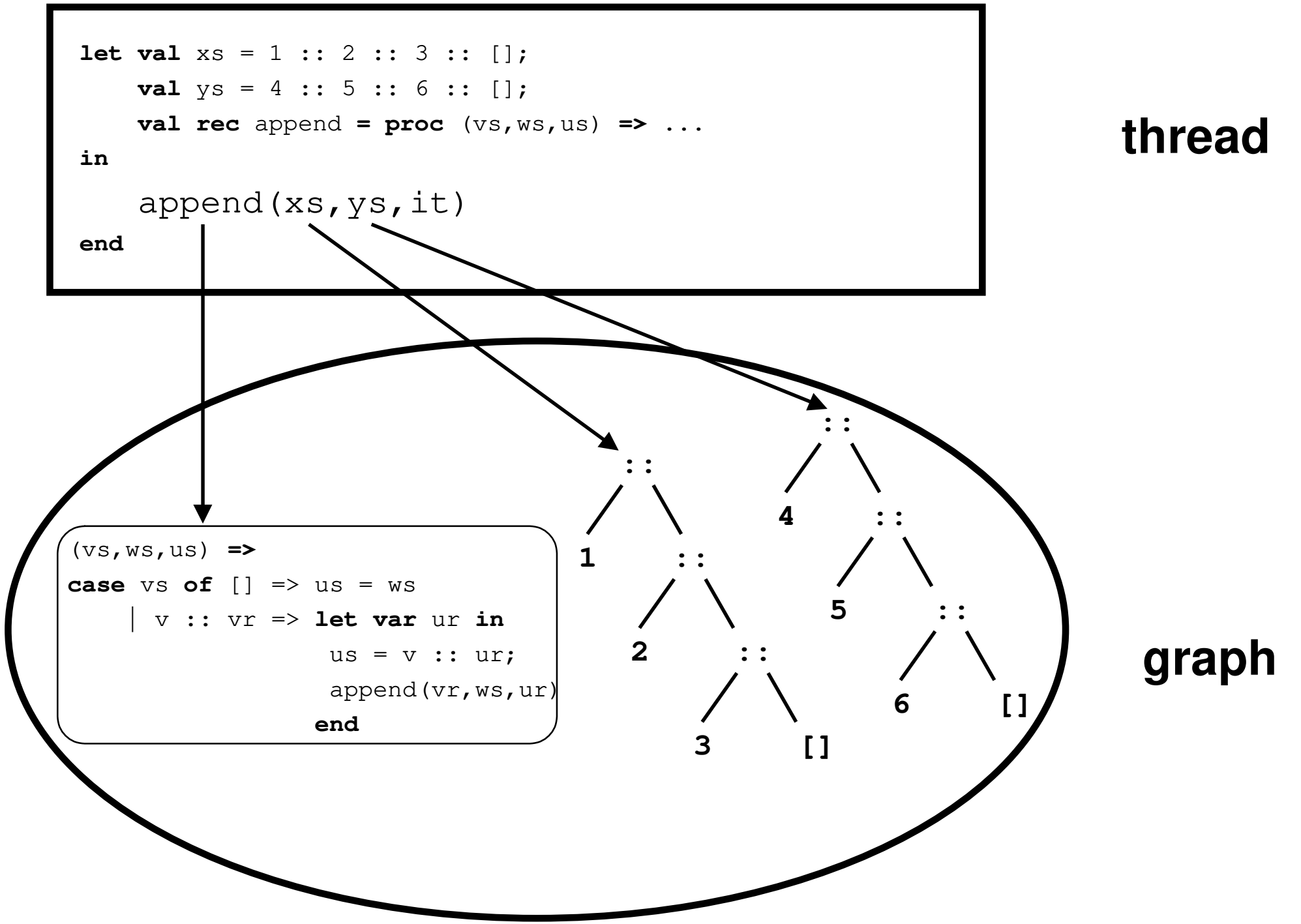
```
let val xs = 1 :: 2 :: 3 :: [];
    val ys = 4 :: 5 :: 6 :: [];
    val rec append = proc (vs,ws,us) =>
                         case vs
                         of []       => us = ws
                          | v :: vr => let var ur in
                                          us = v :: ur;
                                          append(vr,ws,ur)
                                       end
in
    append(xs,ys,it)
end
```

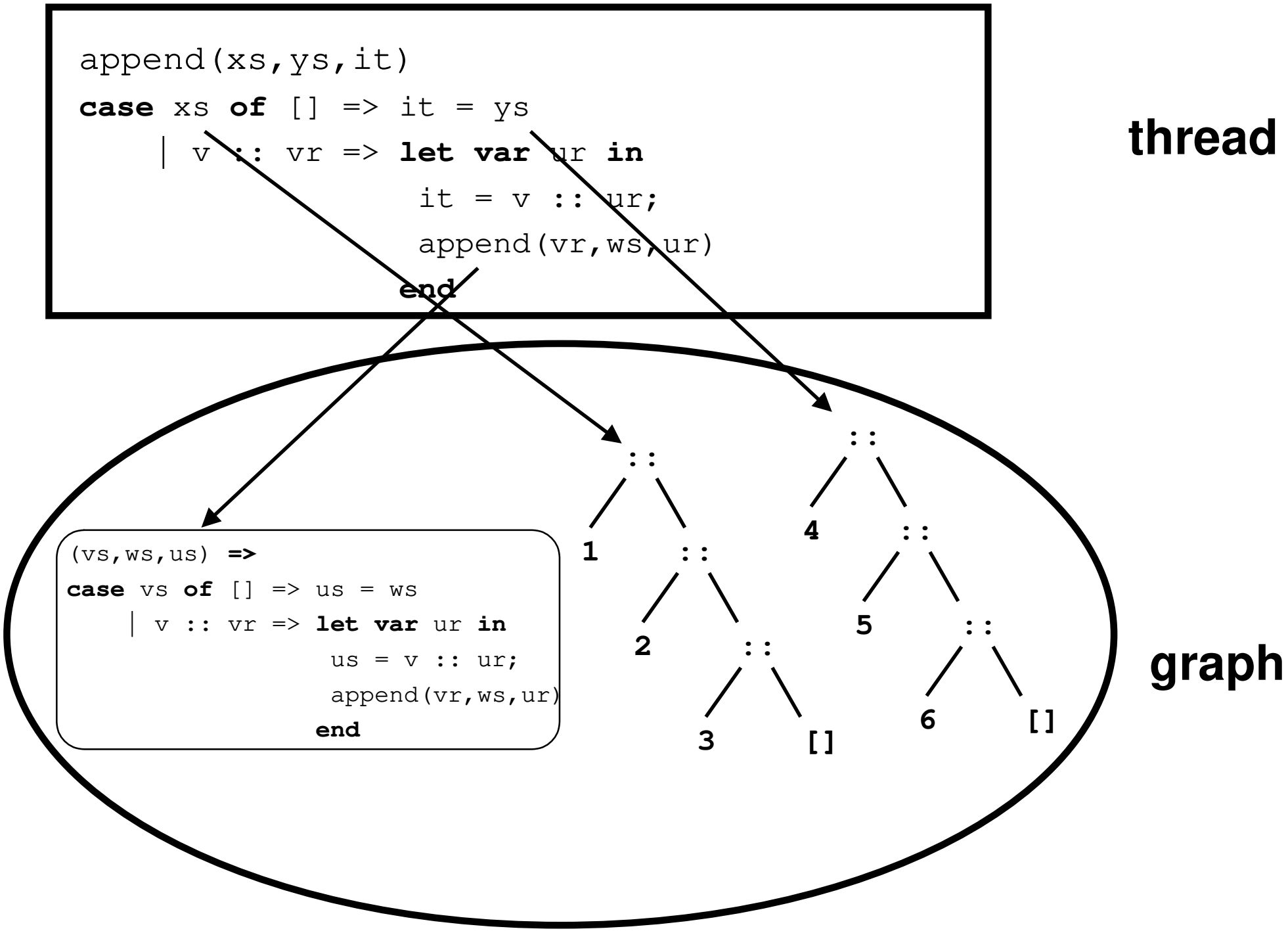
```
let val xs = 1 :: 2 :: 3 :: [];
    val ys = 4 :: 5 :: 6 :: [];
    val rec append =
         proc (vs,ws,us) =>
            case vs of []      => us = ws
                     | v :: vr => let var ur in
                                    us = v :: ur;
                                    append(vr,ws,ur)
                                  end
in  append(xs,ys,it)
end
```

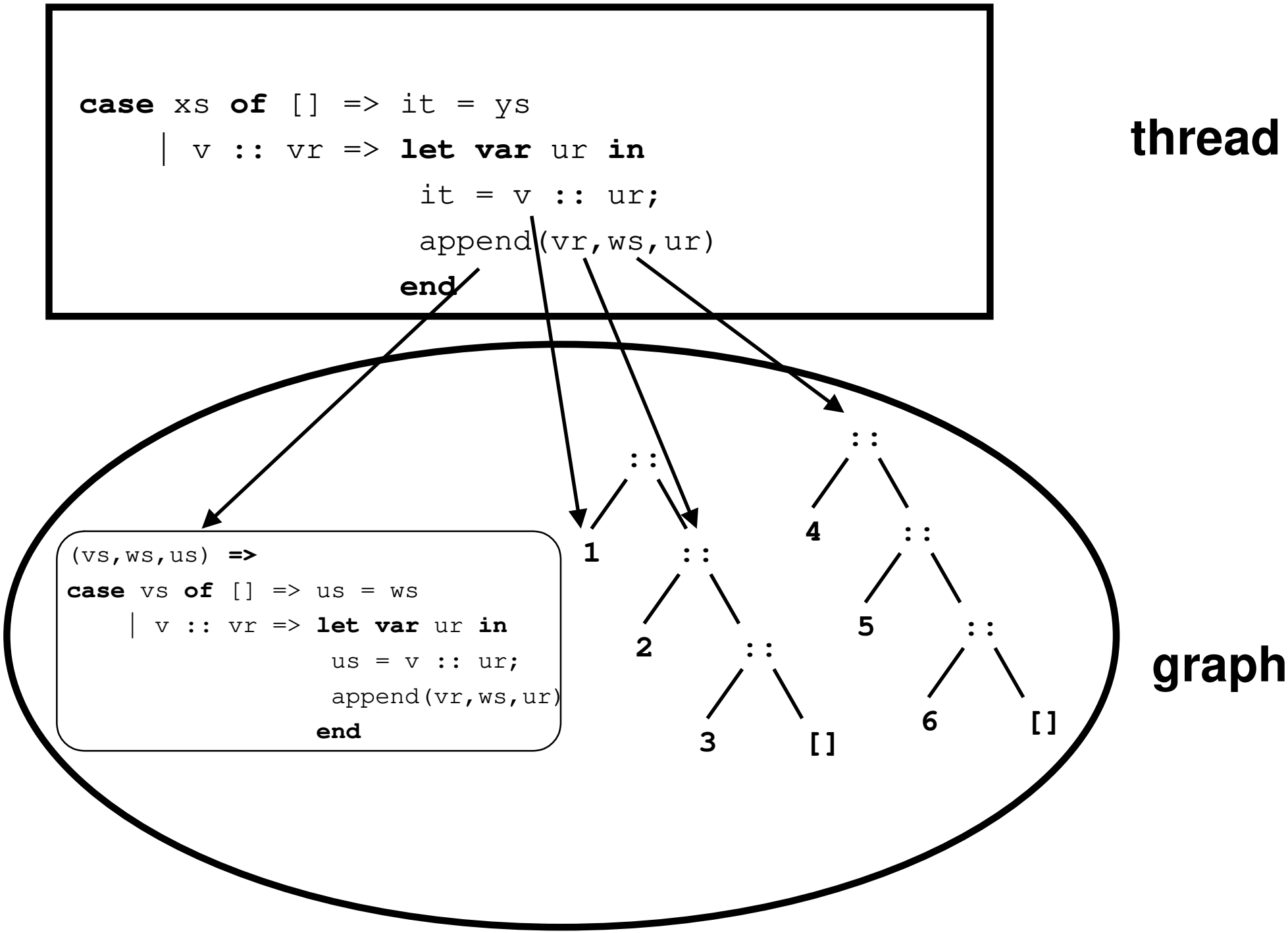
thread

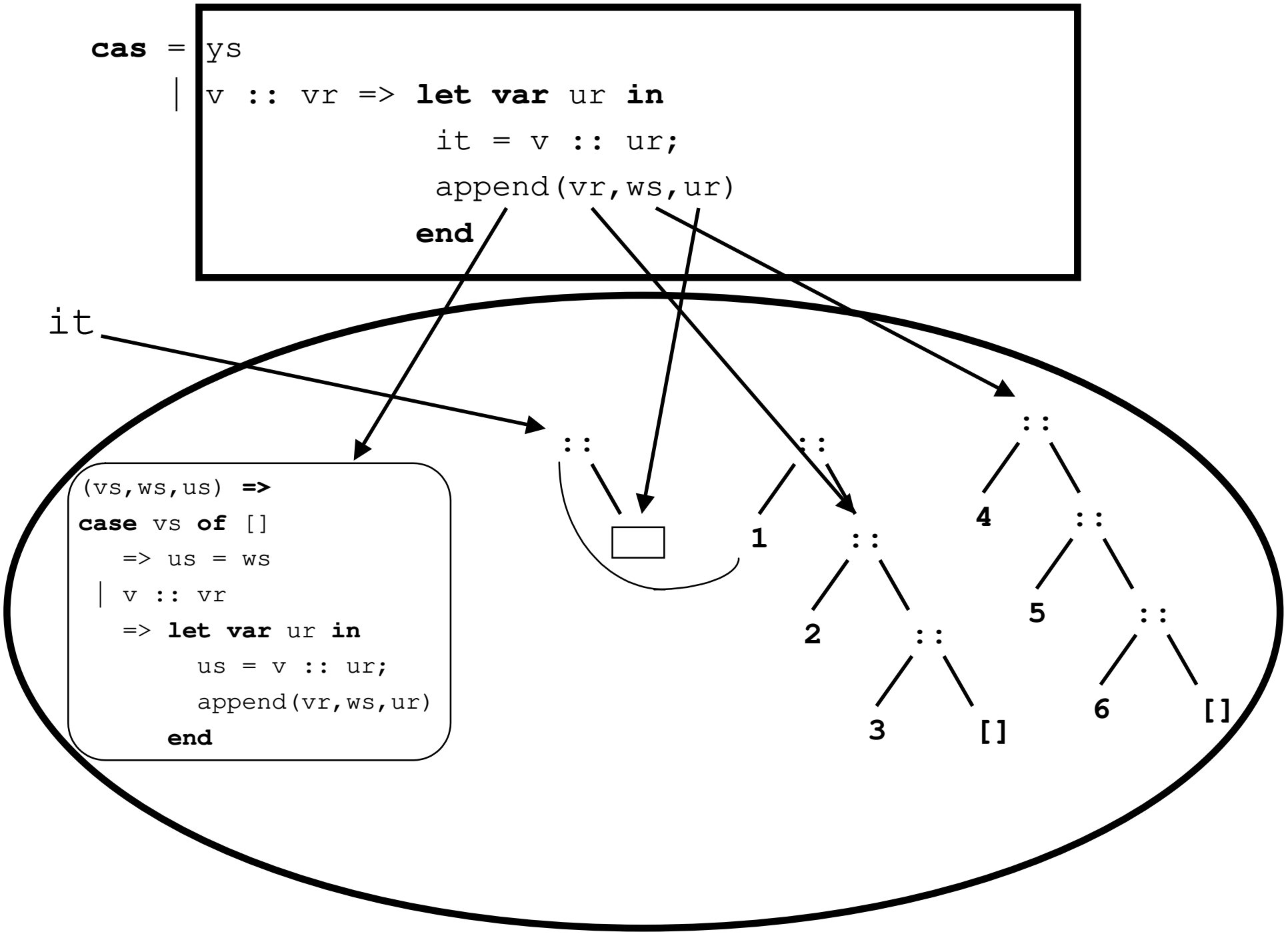


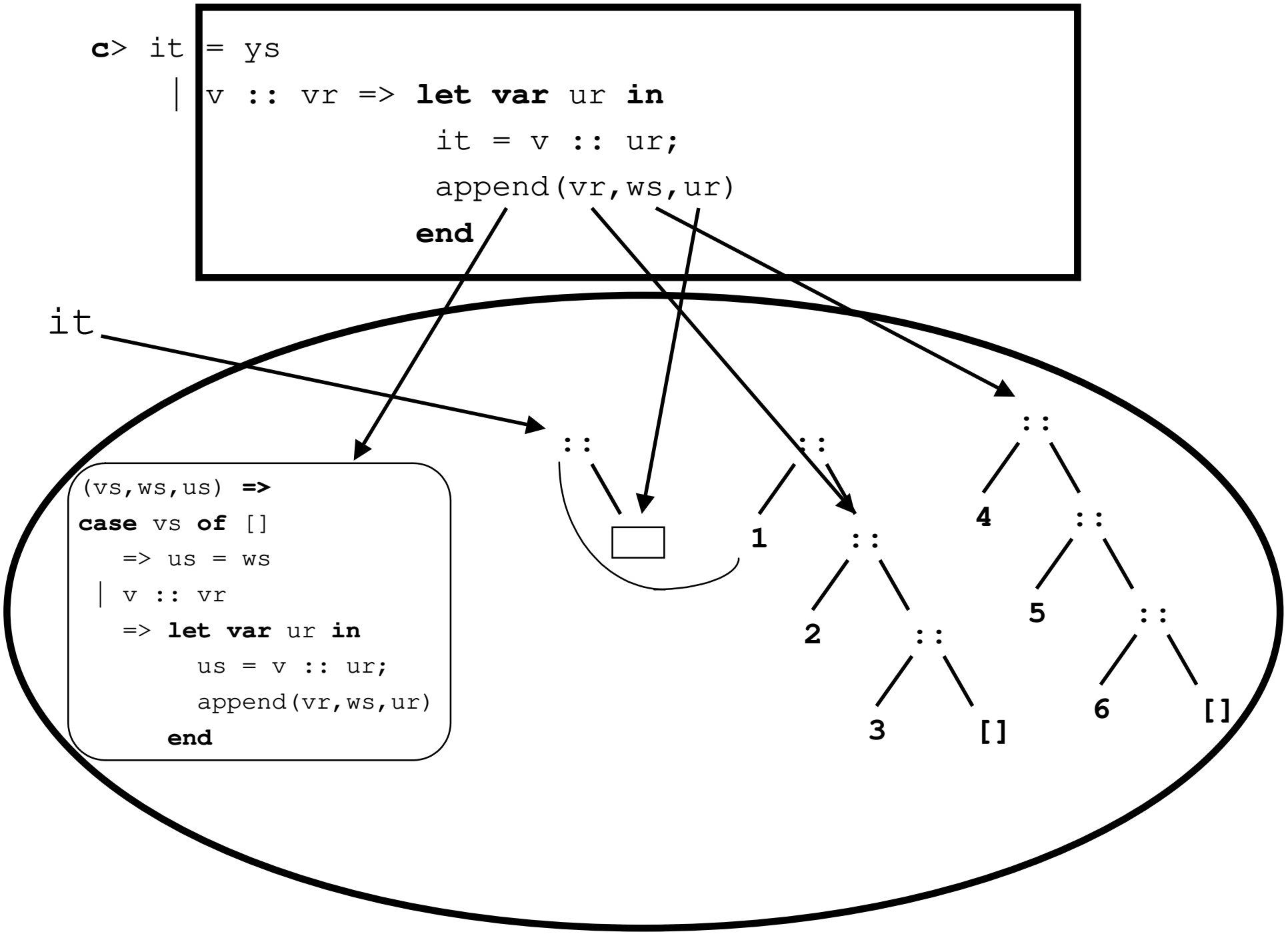
graph

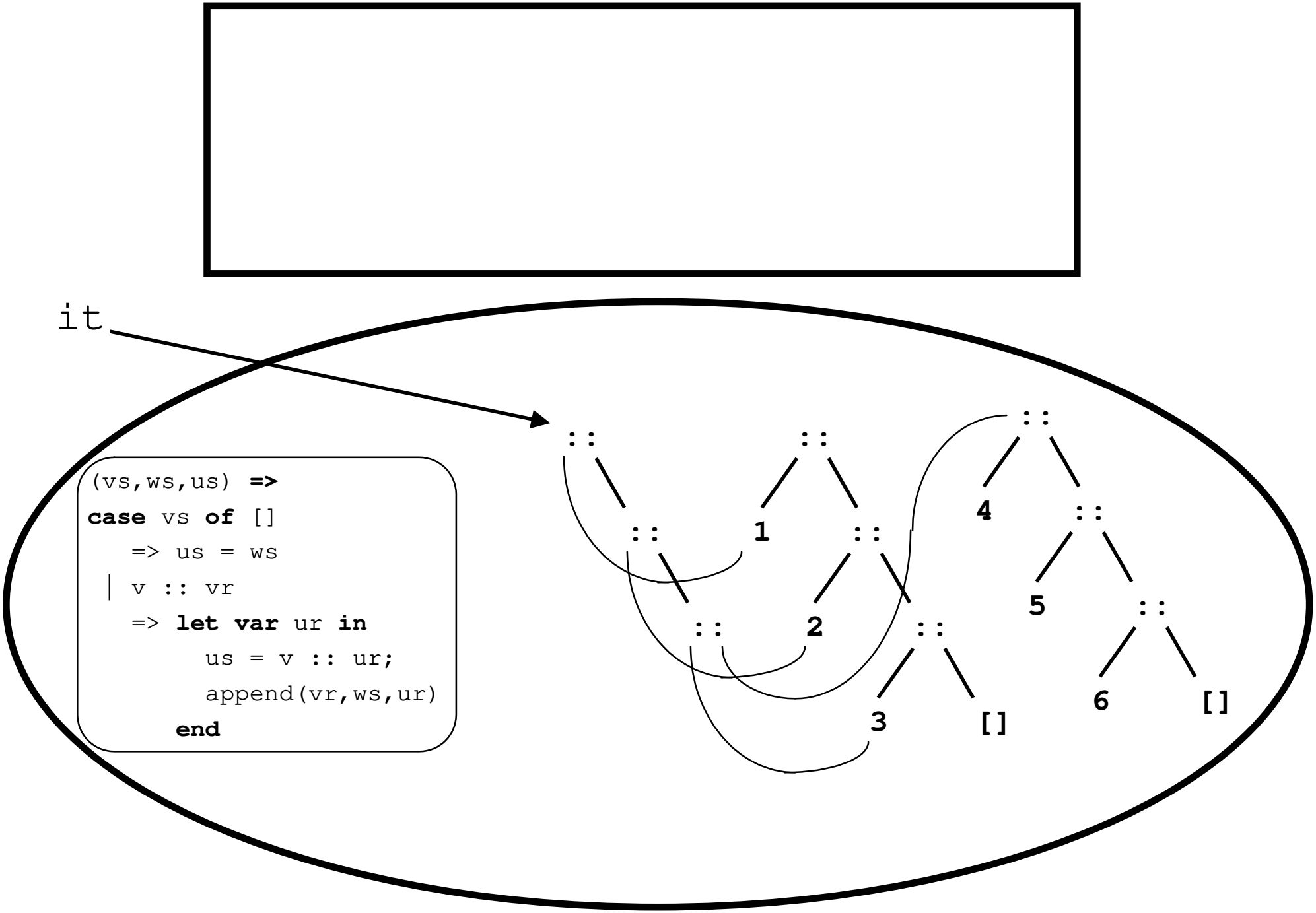

let val xs = 1 :: 2 :: 3 :: [];
val ys = 4 :: 5 :: 6 :: [];
val rec append = proc (vs,ws,us) => ...
in
append(xs,ys,it)
end
(vs,ws,us) =>
case vs of [] => us = ws
| v :: vr => let var ur in
us = v :: ur;
append(vr,ws,ur)
end
::
1
::
2
::
3
[]
::
4
::
5
::
6
[]
thread
graph


append(xs,ys,it)
case xs of [] => it = ys
| v :: vr => let var ur in
it = v :: ur;
append(vr,ws,ur)
end
thread
(vs,ws,us) =>
case vs of [] => us = ws
| v :: vr => let var ur in
us = v :: ur;
append(vr,ws,ur)
end
::
1
::
2
::
3
[]
::
4
::
5
::
6
[]
graph

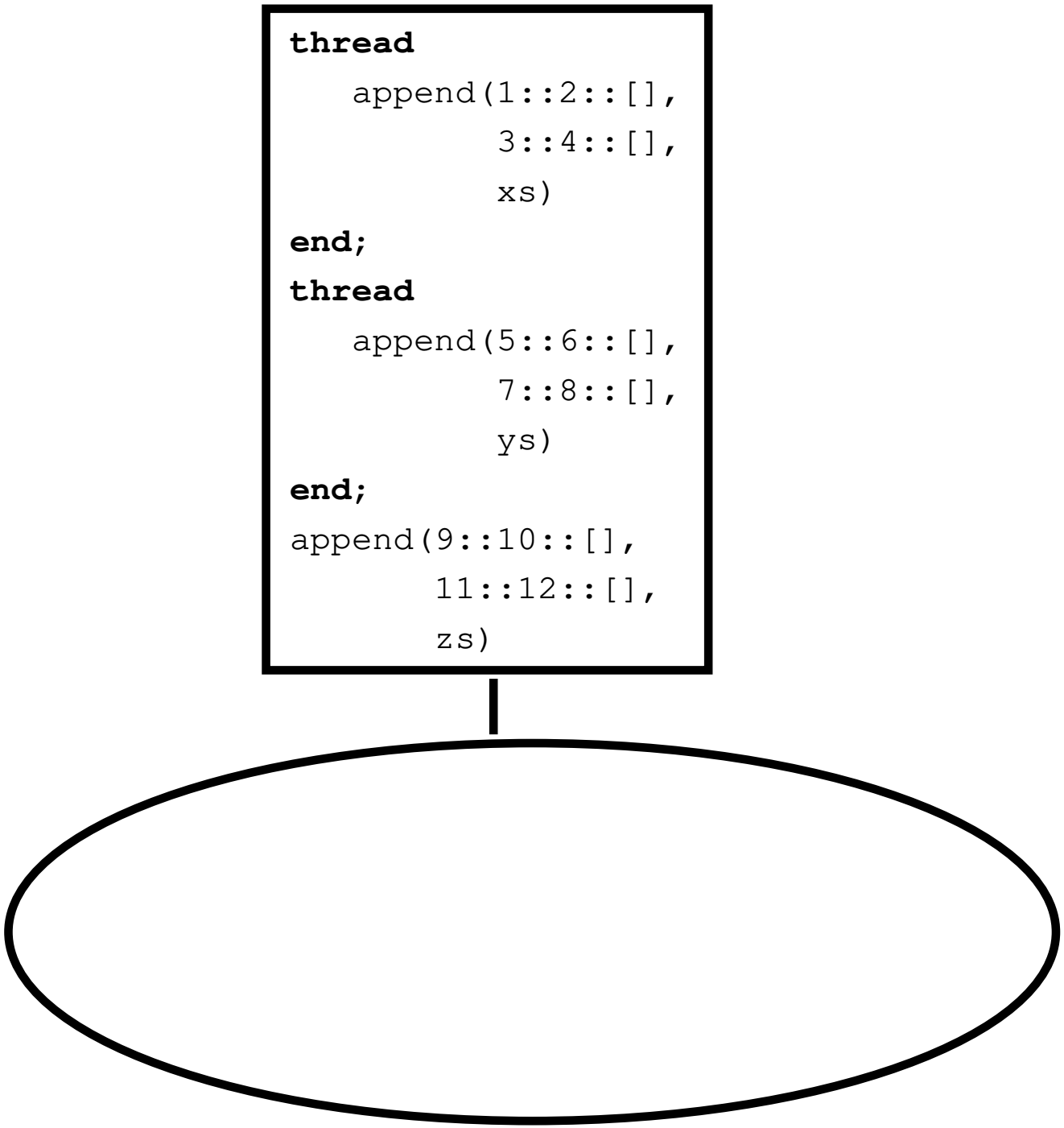

case xs of [] => it = ys
| v :: vr => let var ur in
it = v :: ur;
append(vr,ws,ur)
end
thread
(vs,ws,us) =>
case vs of [] => us = ws
| v :: vr => let var ur in
us = v :: ur;
append(vr,ws,ur)
end
::
1
::
2
::
3
[]
::
4
::
5
::
6
[]
graph


cas = ys
| v :: vr => let var ur in
it = v :: ur;
append(vr,ws,ur)
end
it
(vs,ws,us) =>
case vs of []
=> us = ws
| v :: vr
=> let var ur in
us = v :: ur;
append(vr,ws,ur)
end
::
::
::
::
::
::
1
2
3
[]
4
5
6
[]


c> it = ys
| v :: vr => let var ur in
it = v :: ur;
append(vr,ws,ur)
end
it
(vs,ws,us) =>
case vs of []
=> us = ws
| v :: vr
=> let var ur in
us = v :: ur;
append(vr,ws,ur)
end
::
::
::
::
::
::
1
2
3
[]
4
5
6
[]

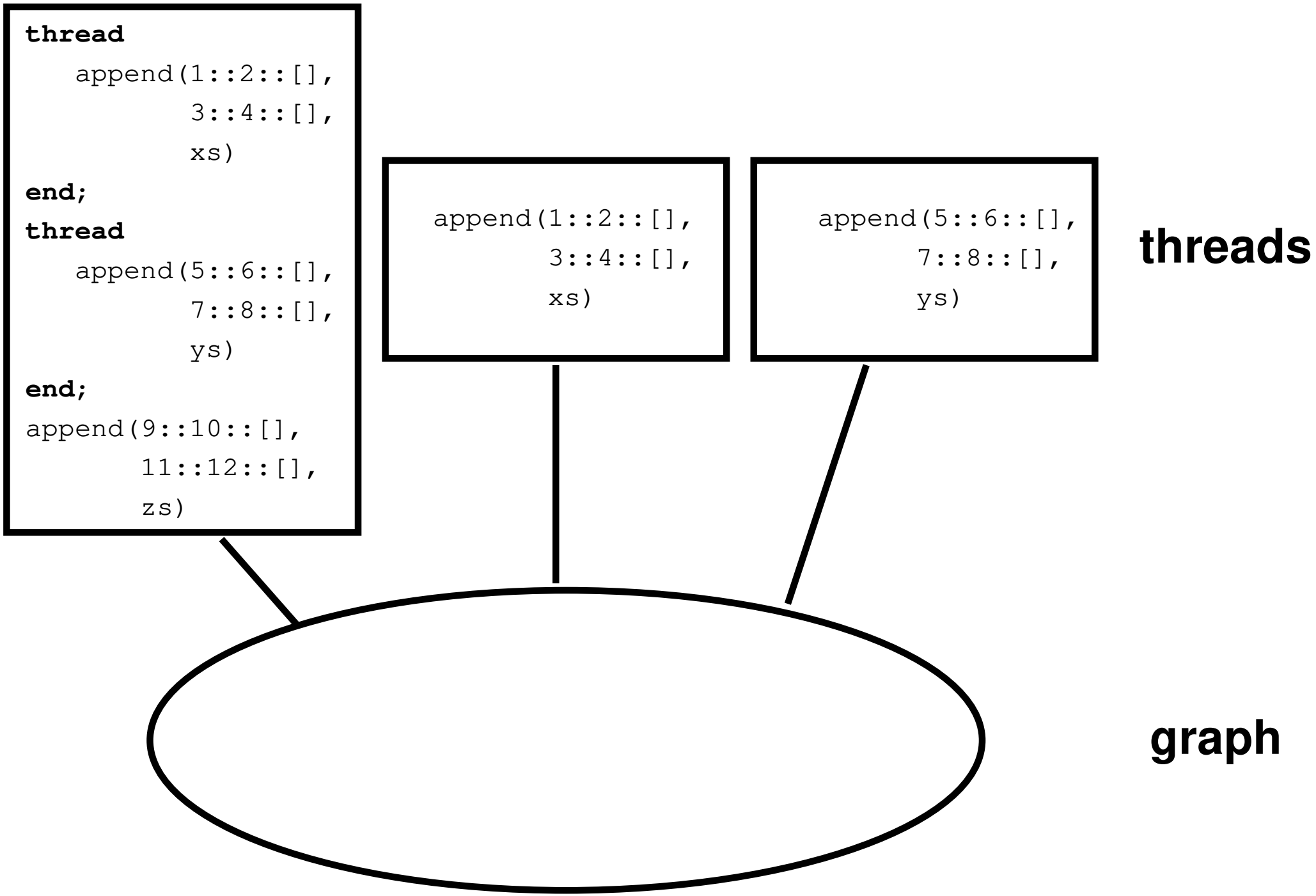

it
(vs,ws,us) =>
case vs of []
=> us = ws
| v :: vr
=> let var ur in
us = v :: ur;
append(vr,ws,ur)
end
::
::
::
::
::
::
1
2
3
[]
4
5
6
[]

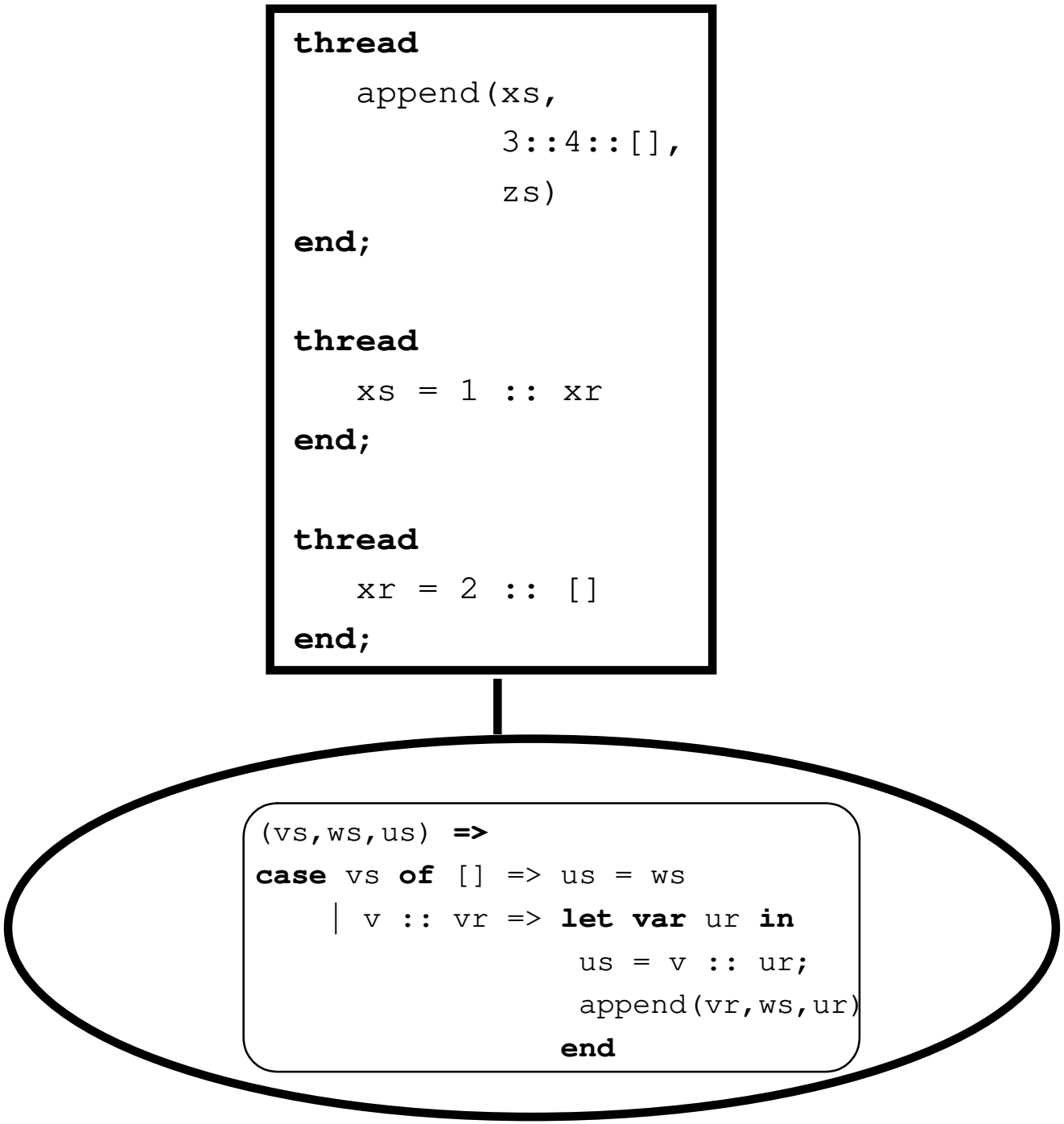
Step 2: Concurrency



thread

graph

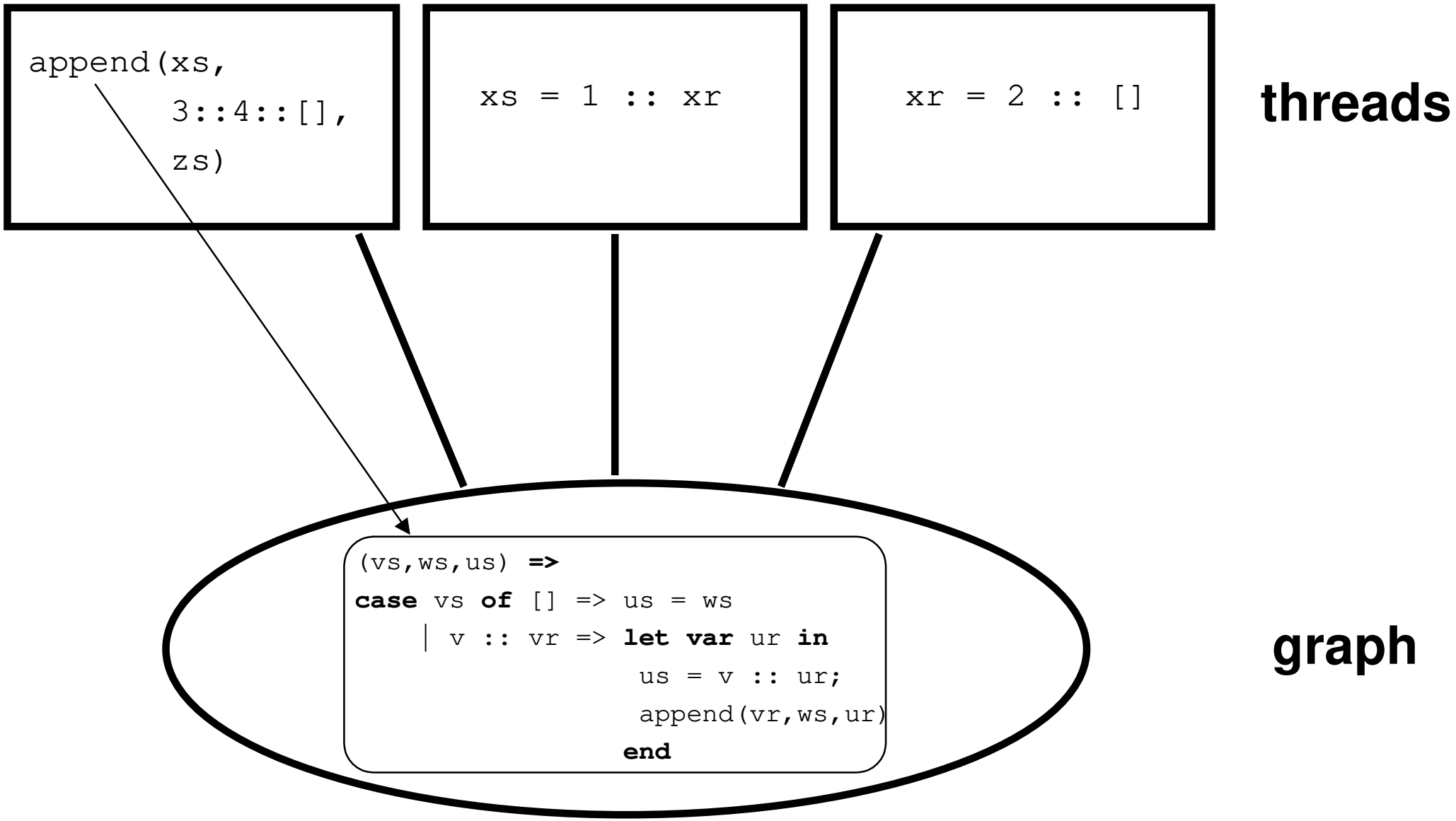

thread
append(1::2::[],
3::4::[],
xs)
end;
thread
append(5::6::[],
7::8::[],
ys)
end;
append(9::10::[],
11::12::[],
zs)
append(1::2::[],
3::4::[],
xs)
append(5::6::[],
7::8::[],
ys)
threads
graph



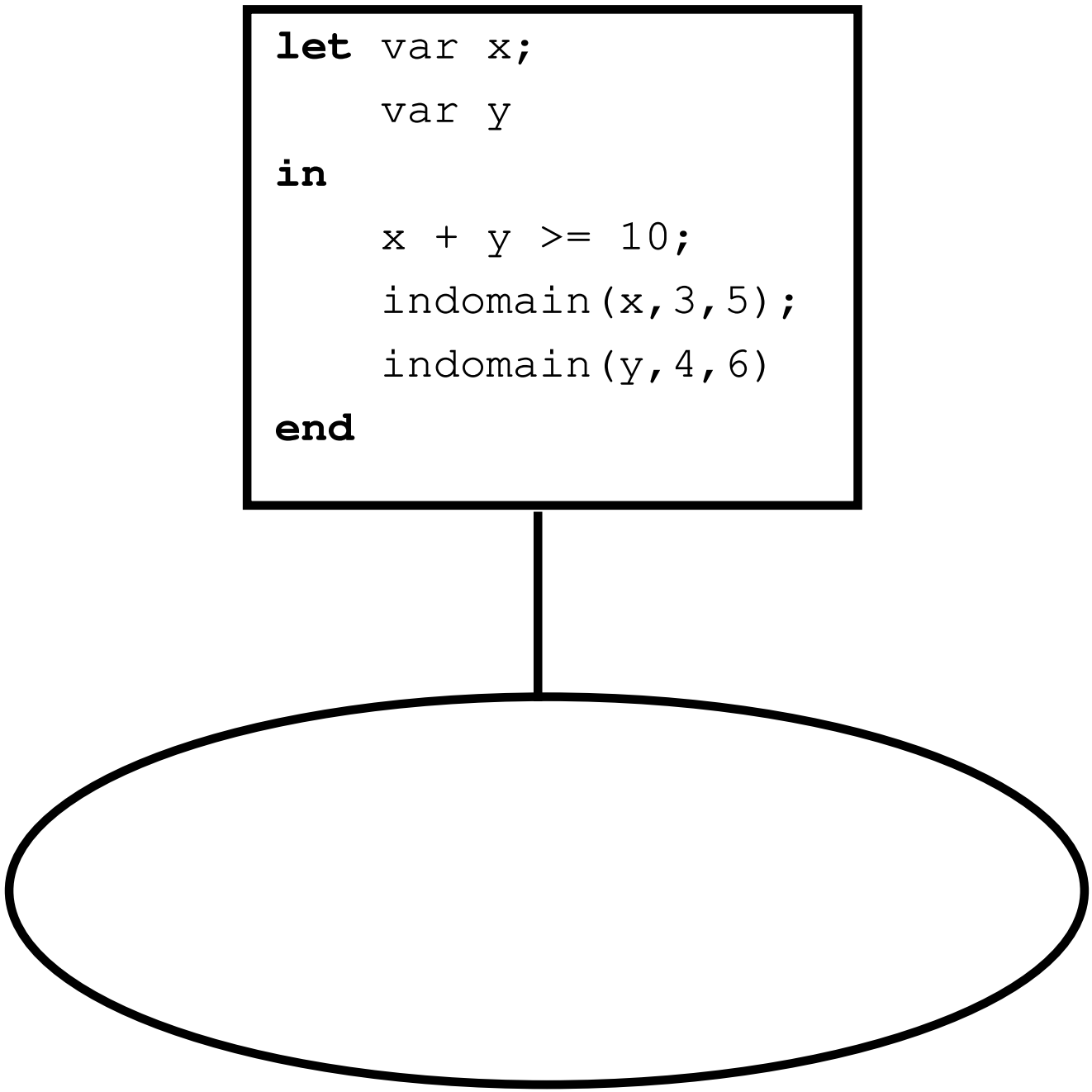
thread

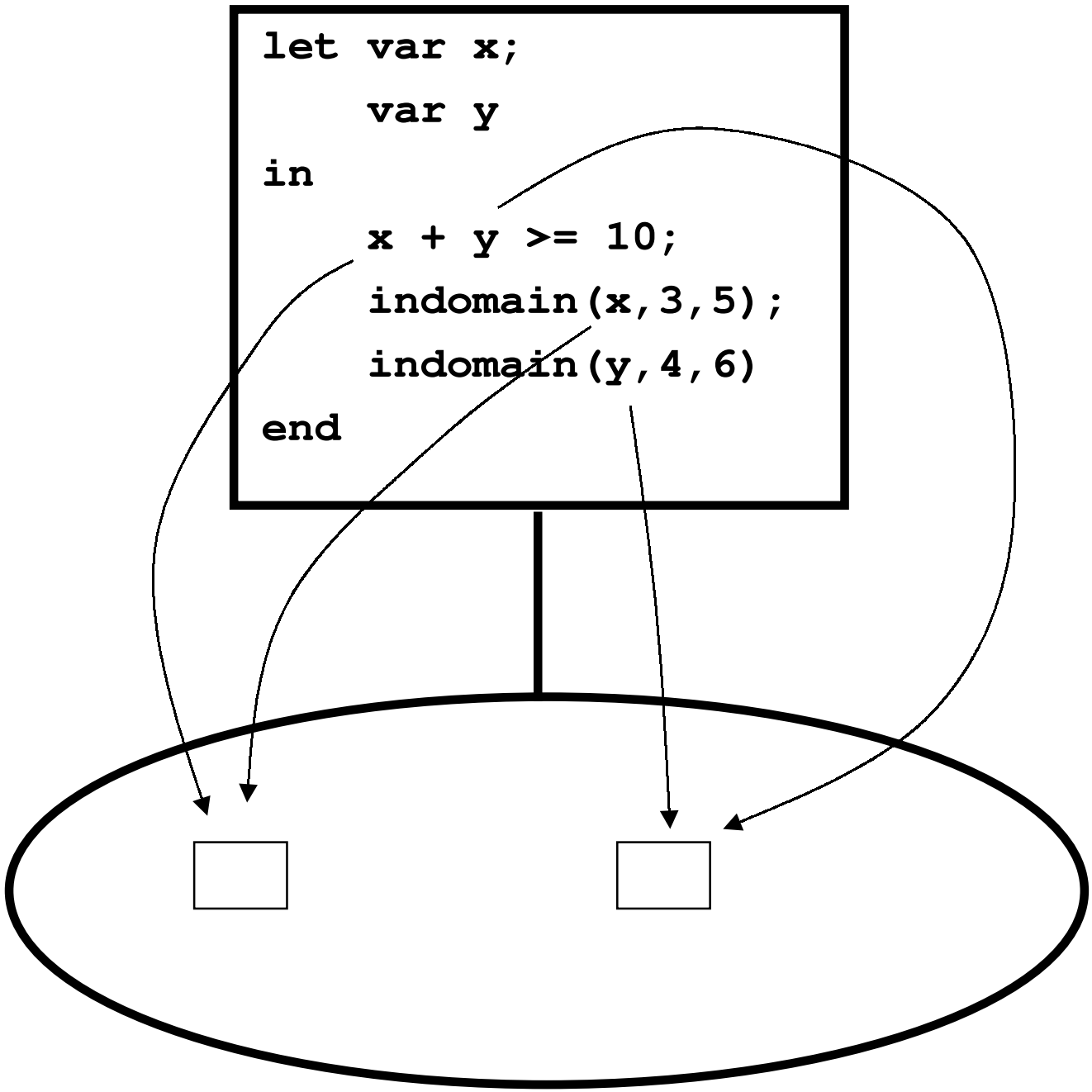
graph

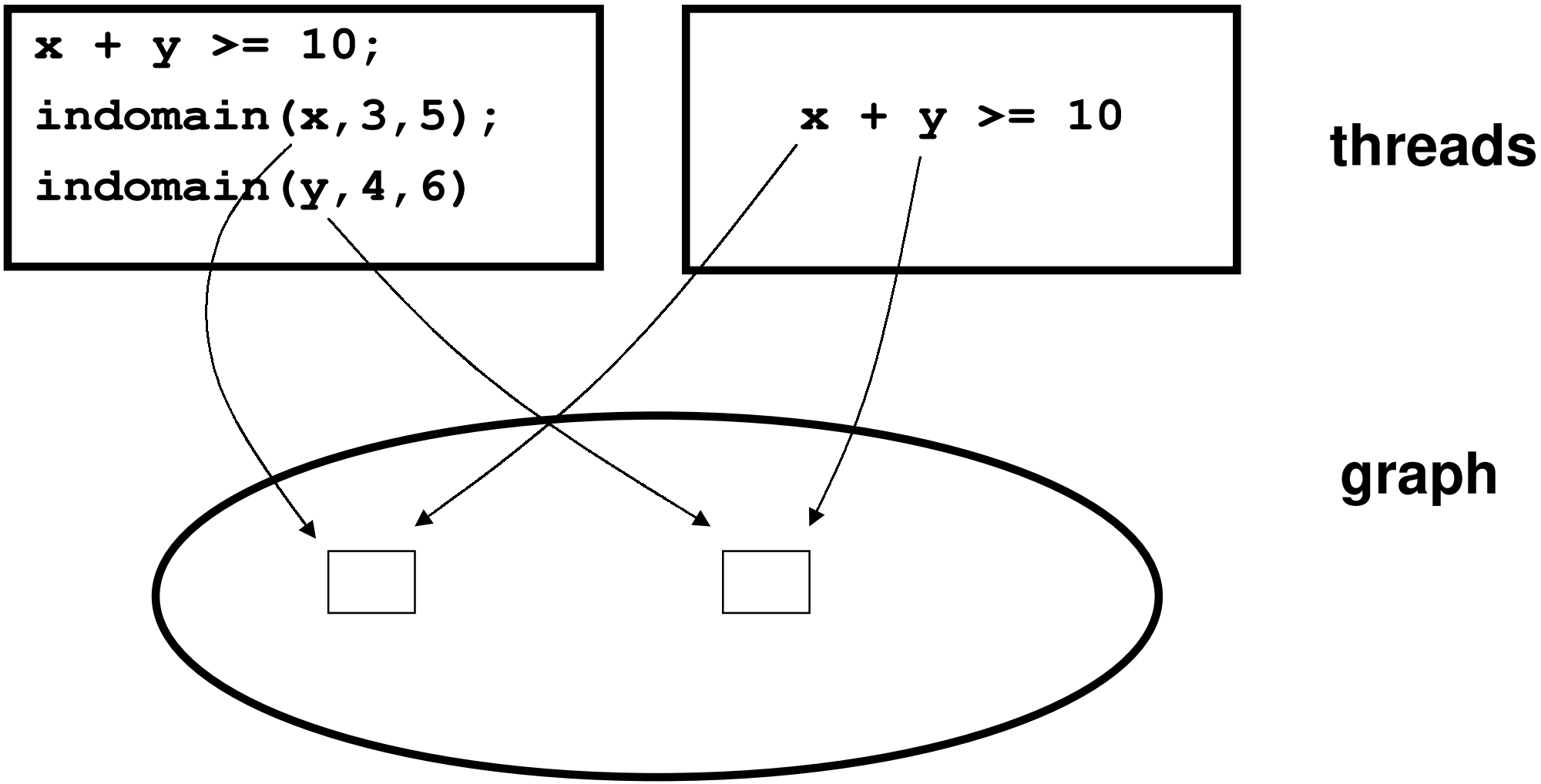
Data-driven Synchronization

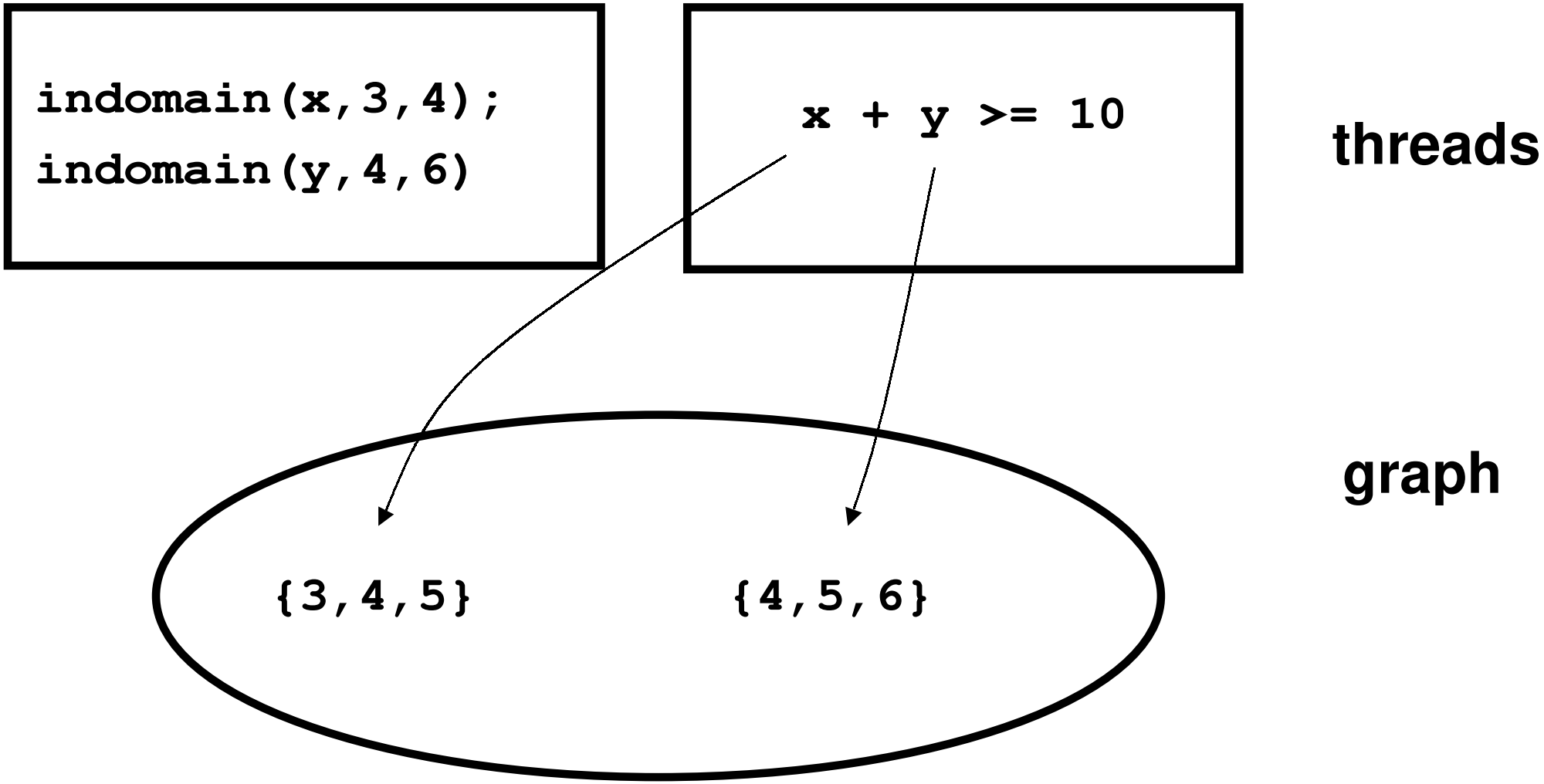


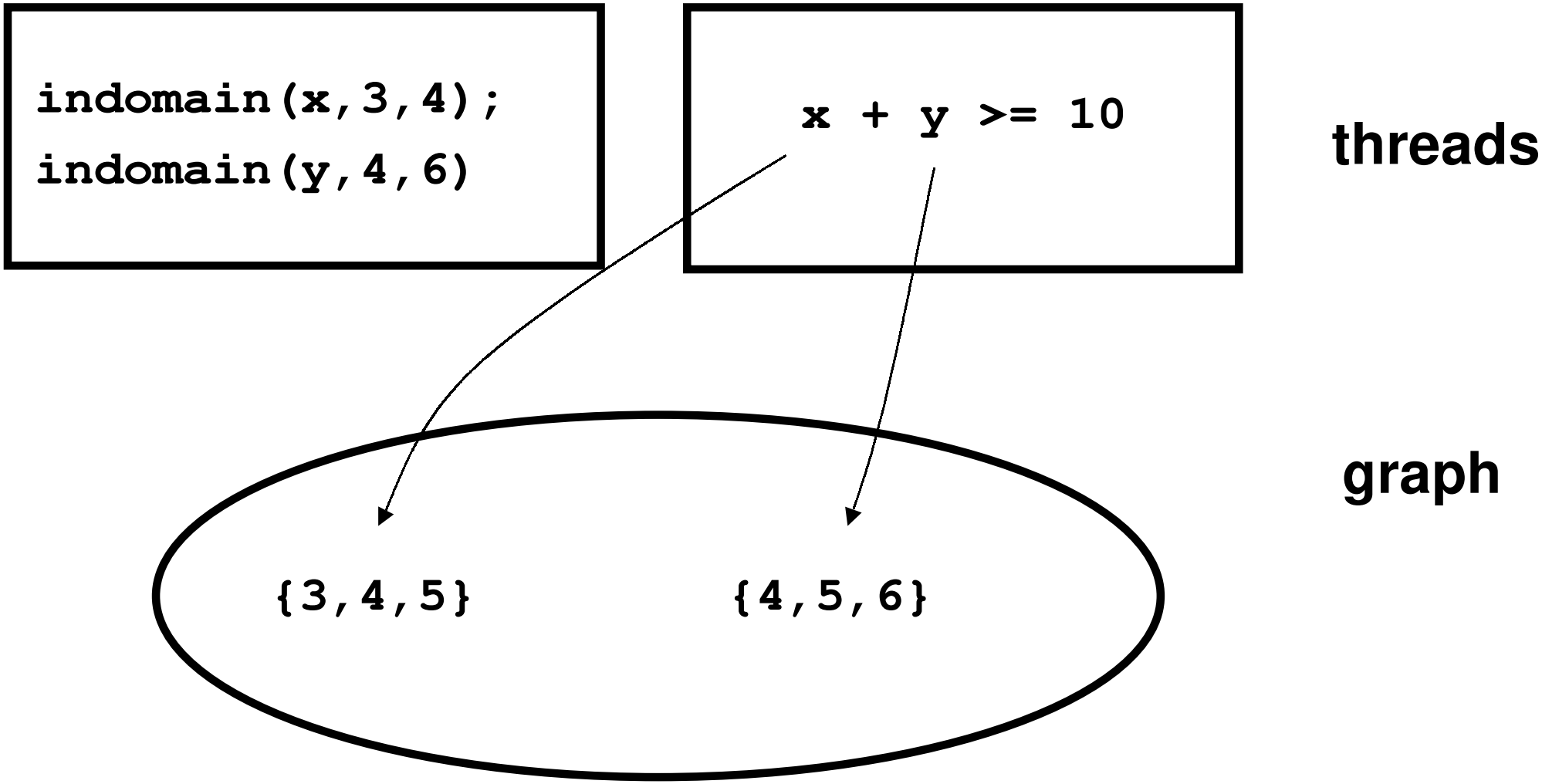
Step 3: Finite Domain Constraints


let var x;
var y
in
x + y >= 10;
indomain(x,3,5);
indomain(y,4,6)
end
thread
graph


let var x;
var y
in
x + y >= 10;
indomain(x,3,5);
indomain(y,4,6)
end
thread
graph

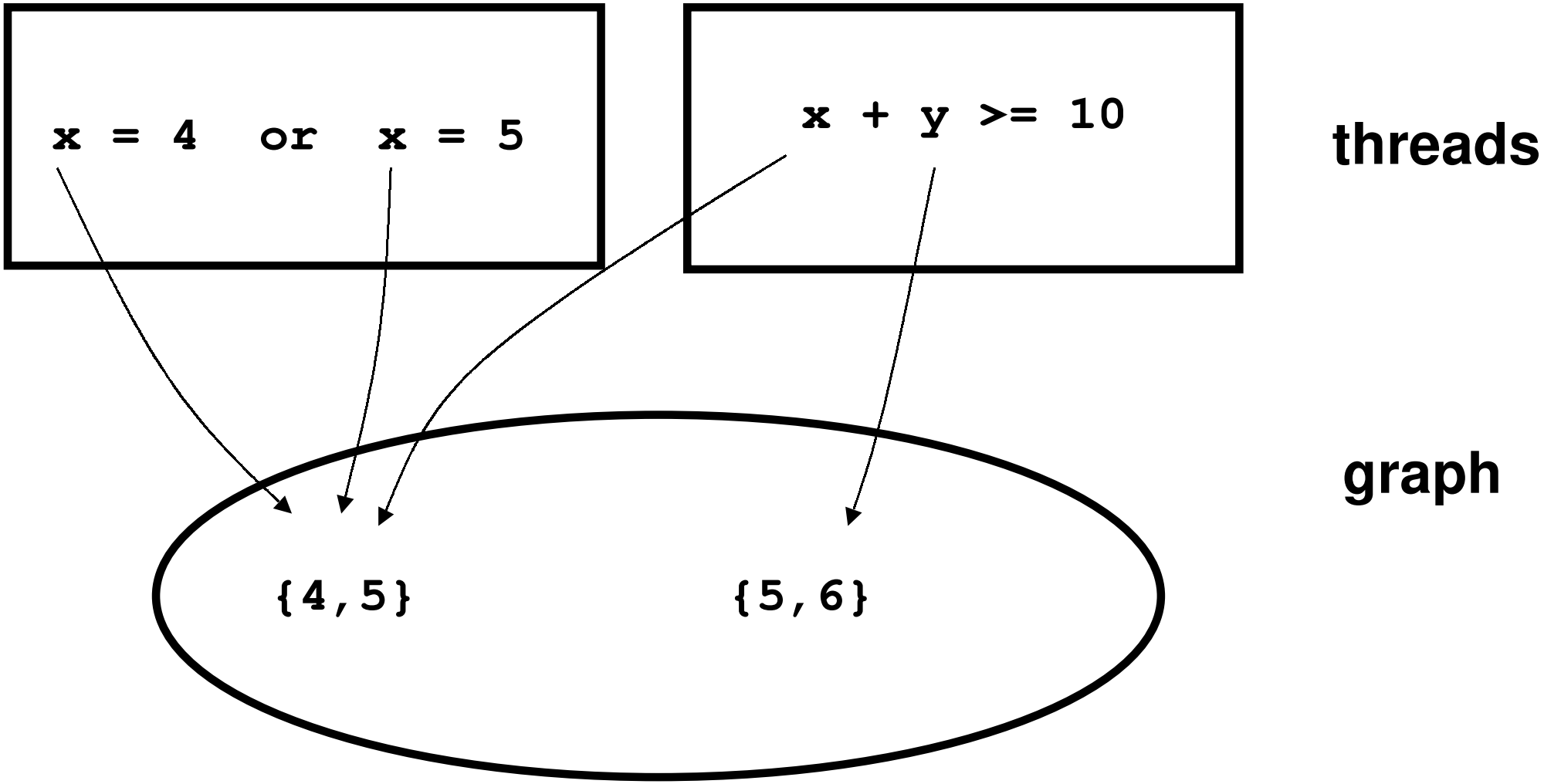

x + y >= 10;
indomain(x,3,5);
indomain(y,4,6)
x + y >= 10
threads
graph

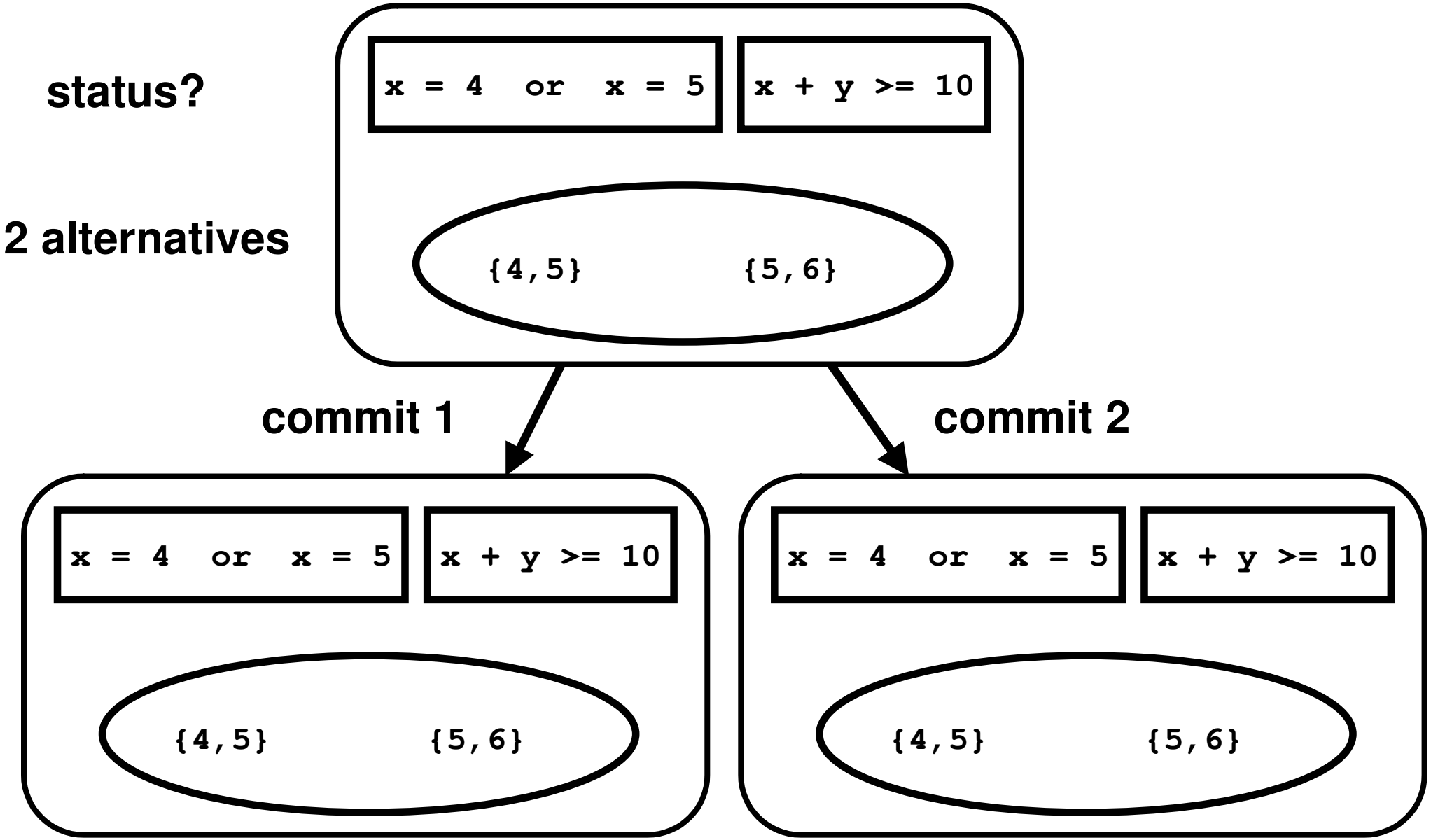

indomain(x,3,4);
indomain(y,4,6)
x + y >= 10
threads
graph
{3,4,5}
{4,5,6}

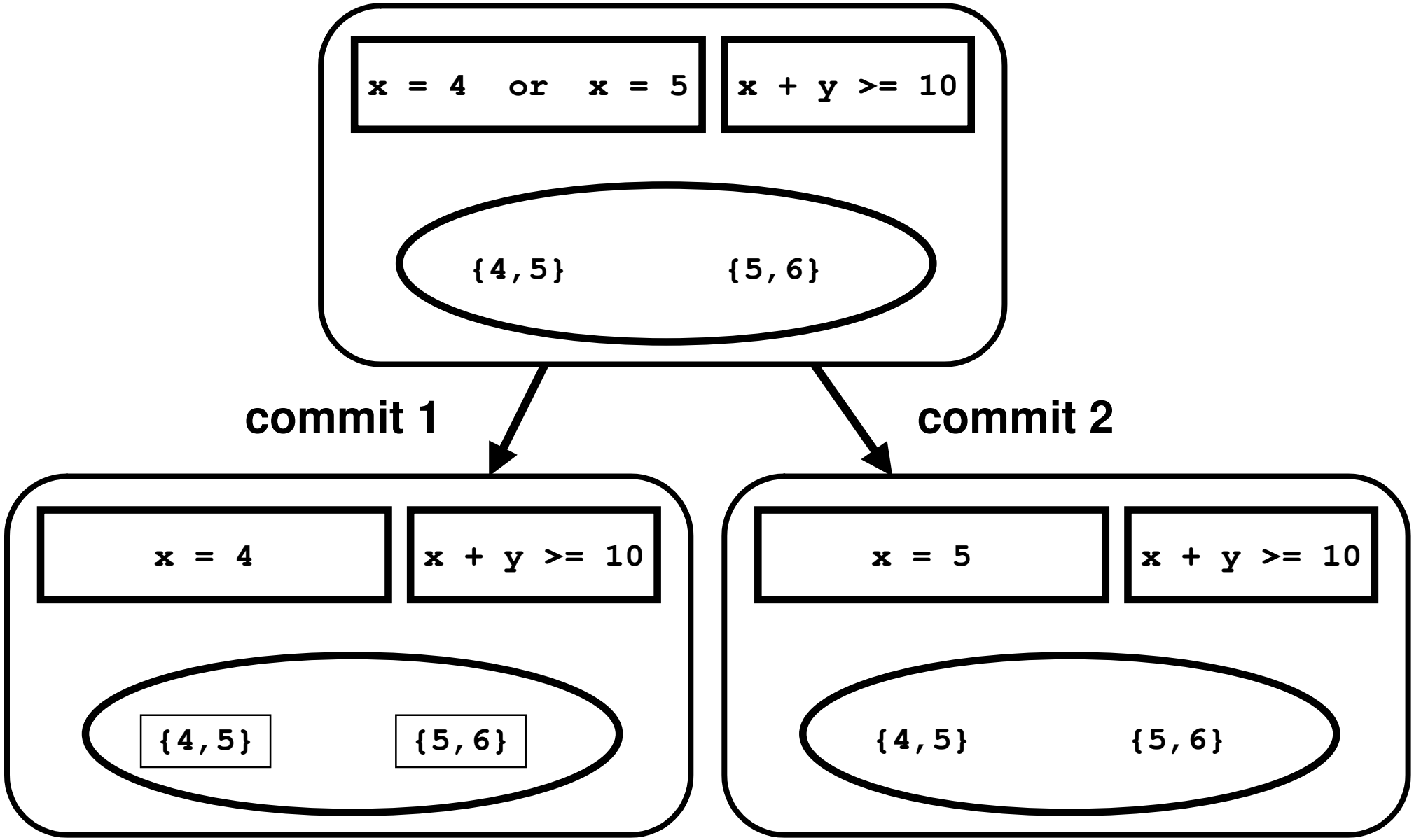

indomain(x,3,4);
indomain(y,4,6)
x + y >= 10
threads
graph
{3,4,5}
{4,5,6}

Step 4: Search

Choice Point




status?
2 alternatives
x = 4 or x = 5
x + y >= 10
{4,5}
{5,6}
commit 1
commit 2
x = 4 or x = 5
x + y >= 10
{4,5}
{5,6}
x = 4 or x = 5
x + y >= 10
{4,5}
{5,6}


x = 4 or x = 5
x + y >= 10
{4,5}
{5,6}
commit 1
commit 2
x = 4
x + y >= 10
{4,5}
{5,6}
x = 5
x + y >= 10
{4,5}
{5,6}

Oz

- concurrent constraint language
- functional syntax
- object-oriented programming
- feature and finite domain constraints

DFKI Oz 2.0

- emulator-based implementation
- programming interface
- constraint programming tools
- graphical user interface based on Tcl/Tk

- download from `http://www.ps.uni-sb.de/oz/`