Objects for Concurrent Constraint Programming

Martin Henz
School of Computing
National University of Singapore

Overview

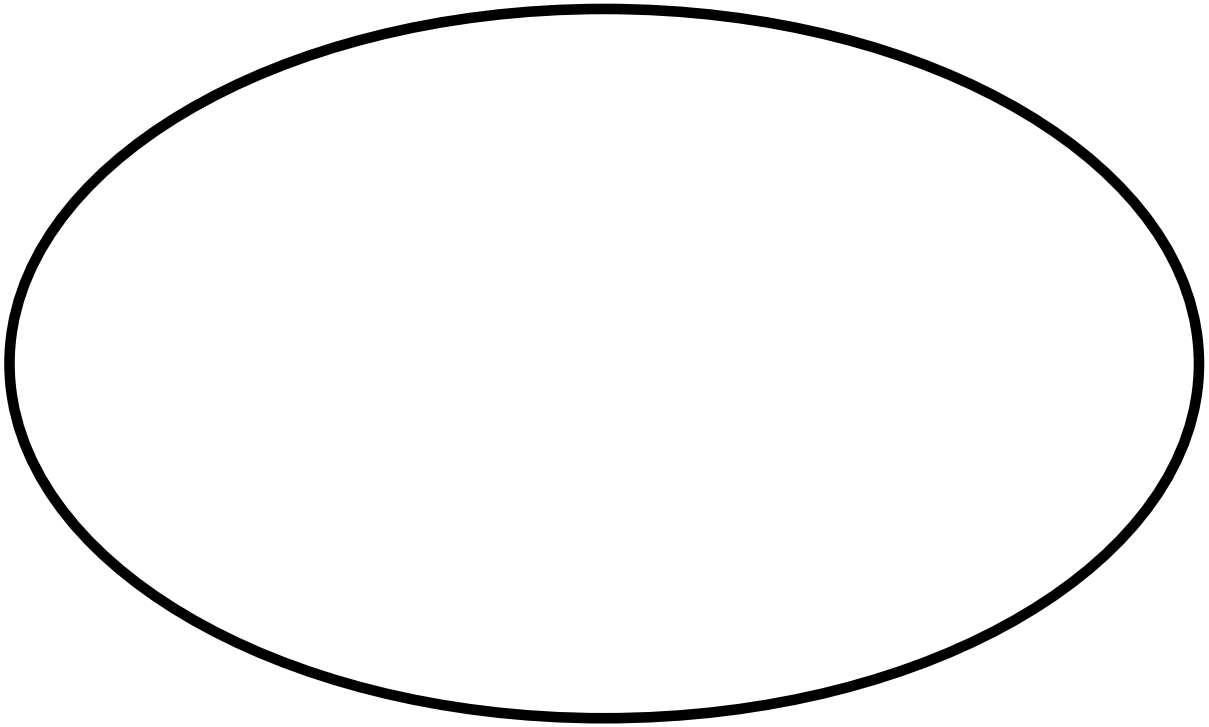
- From Functional to Concurrent Constraint Programming
- The Language Oz
- Sequential Object-oriented Programming in Oz
- Synchronization Techniques
- Implementation of Objects in Oz
- Demo

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
```

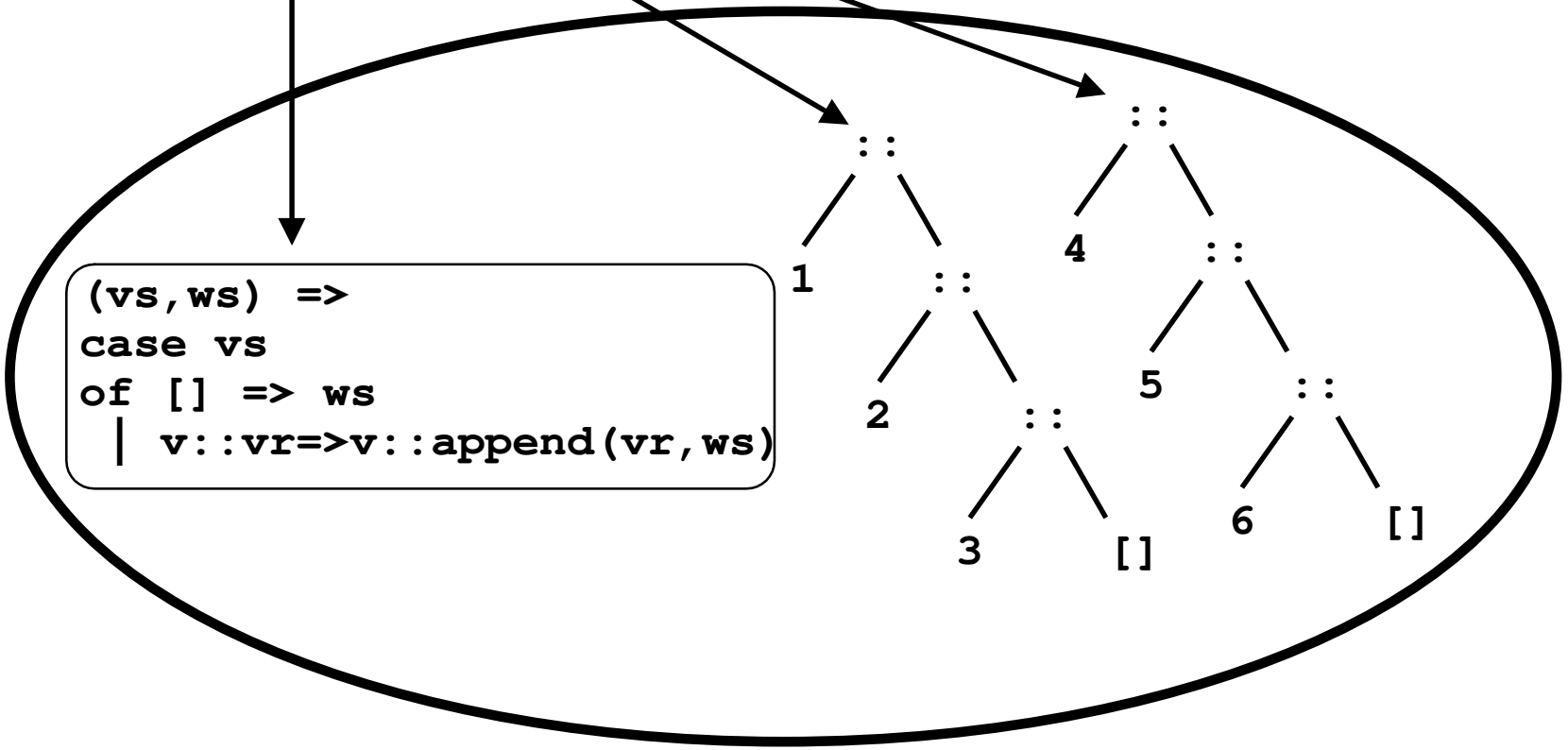
thread



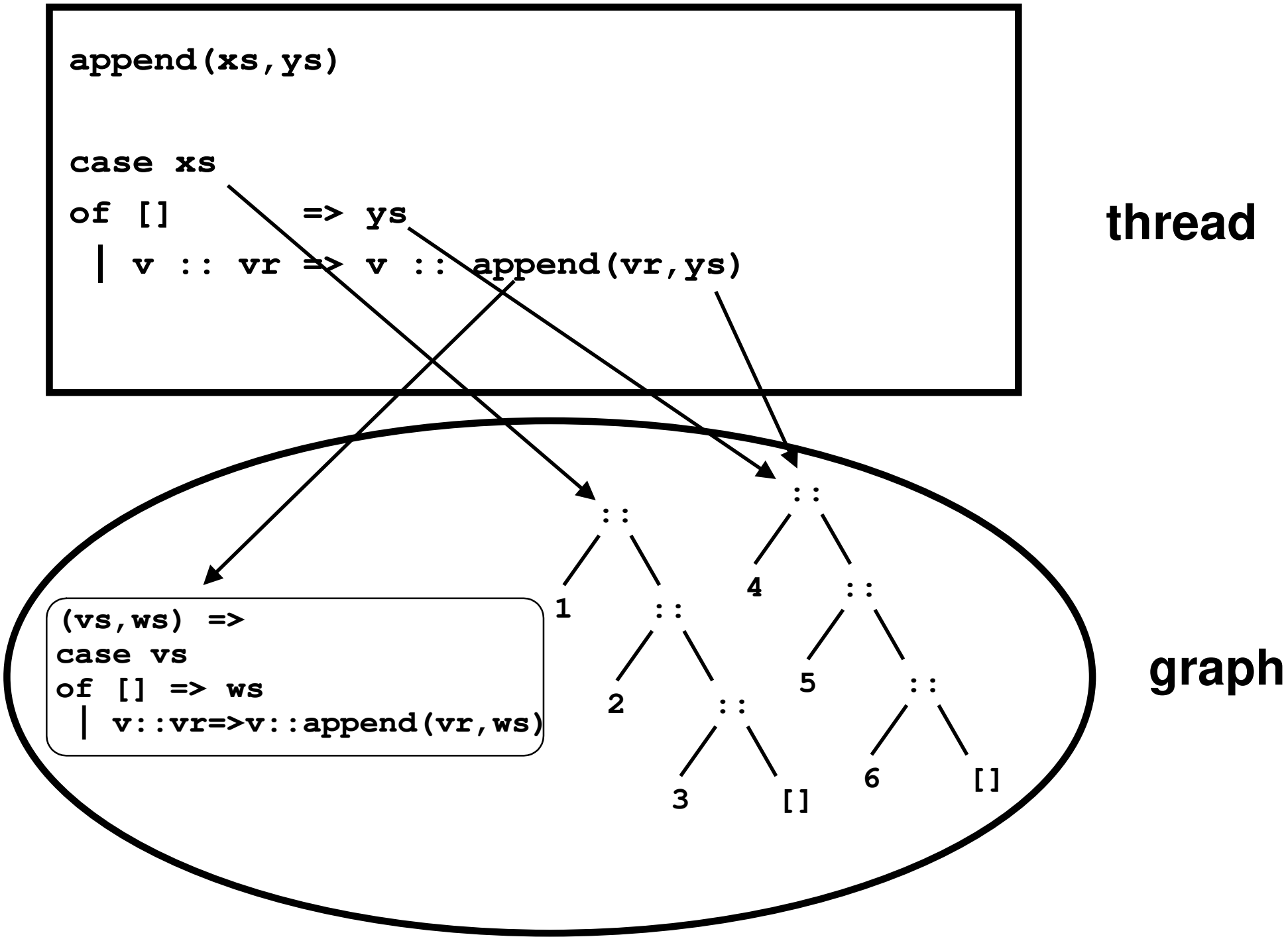
graph

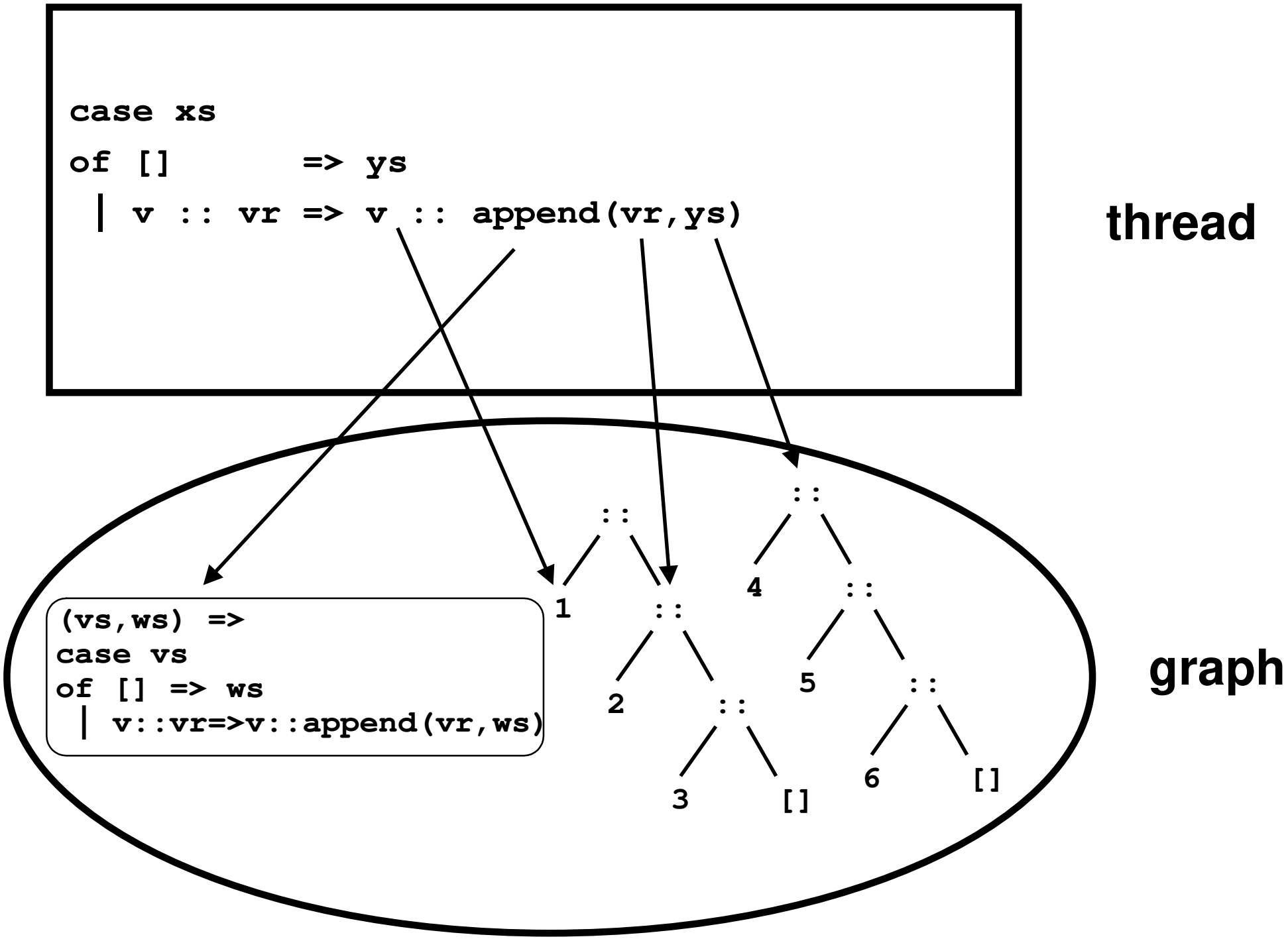
```
let val xs = 1 :: 2 :: 3 :: [];
    val ys = 4 :: 5 :: 6 :: [];
    val rec append = fn (vs,ws) =>
                        case vs
                        of [] => ws
                         | v :: vr => v :: append(v,ws)
in
    append(xs,ys)
end
```

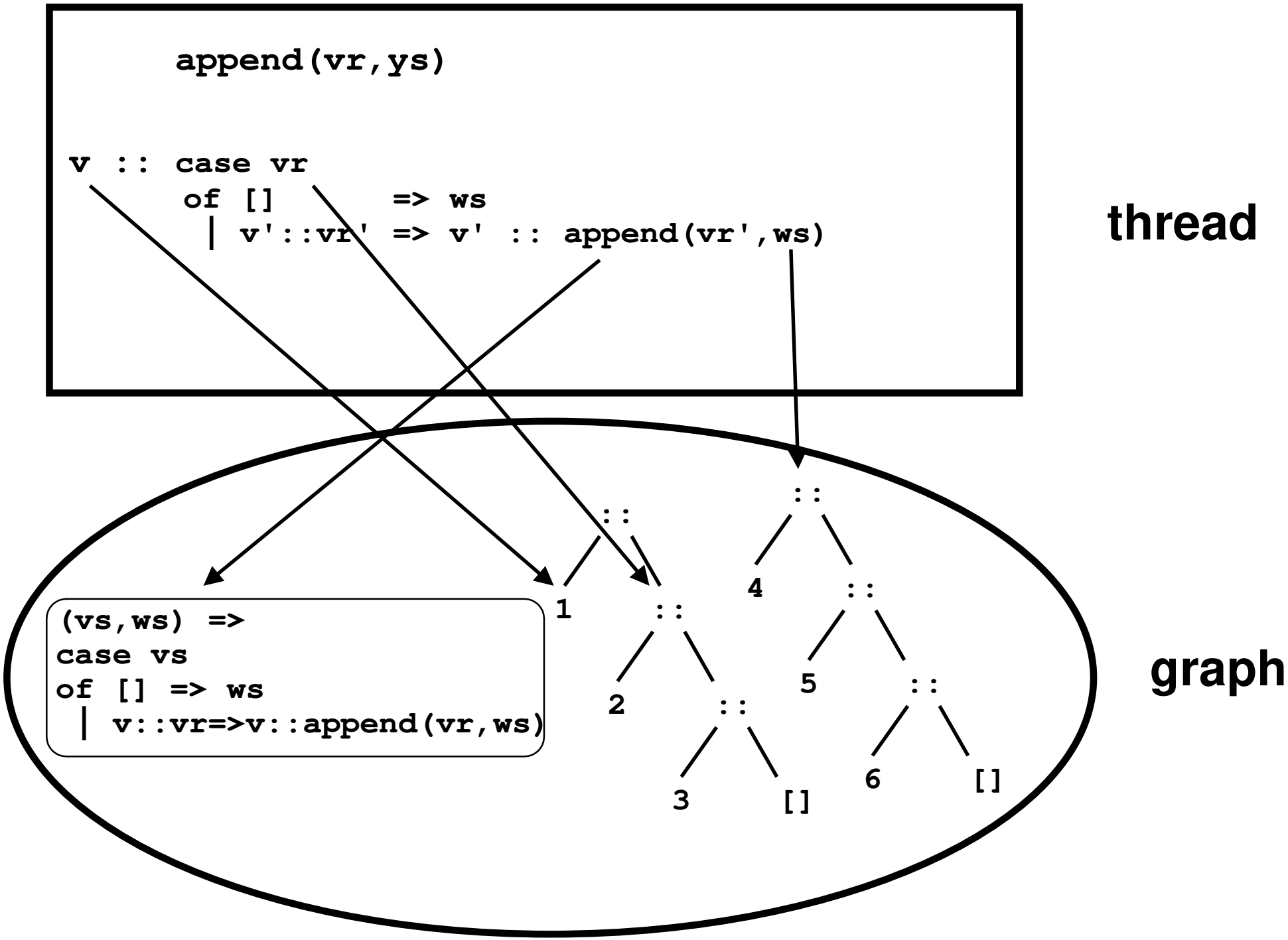
thread

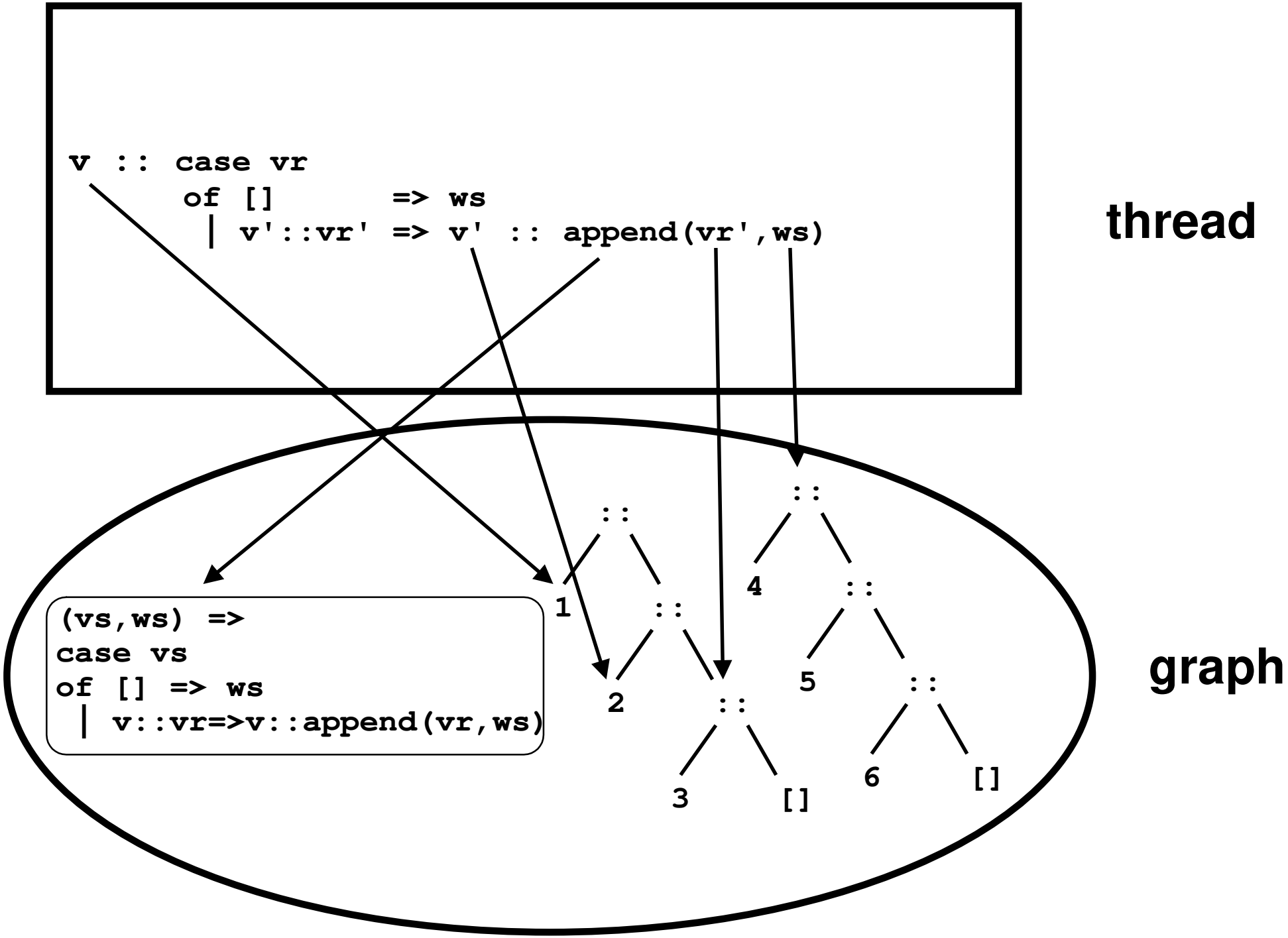


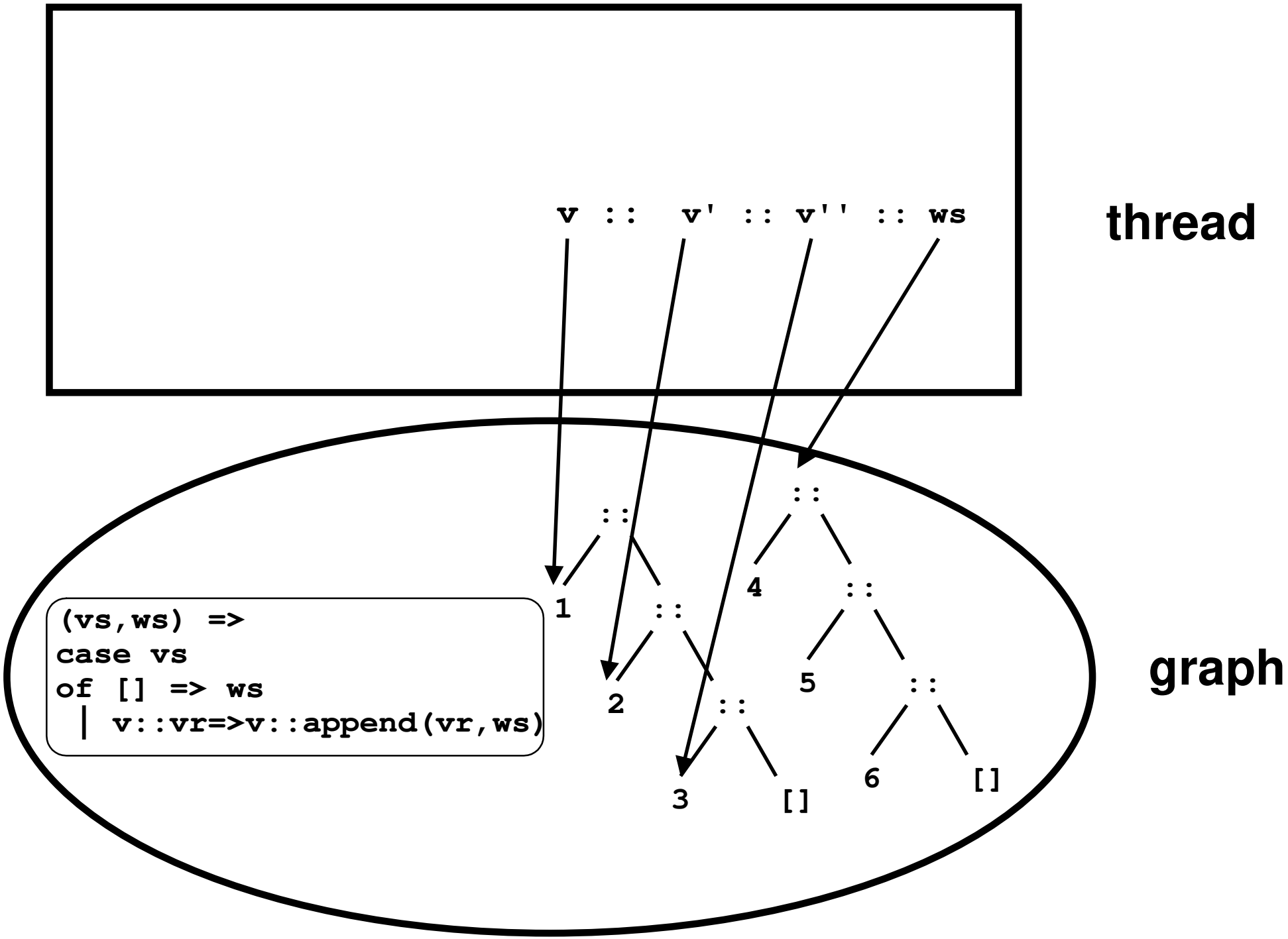
graph


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

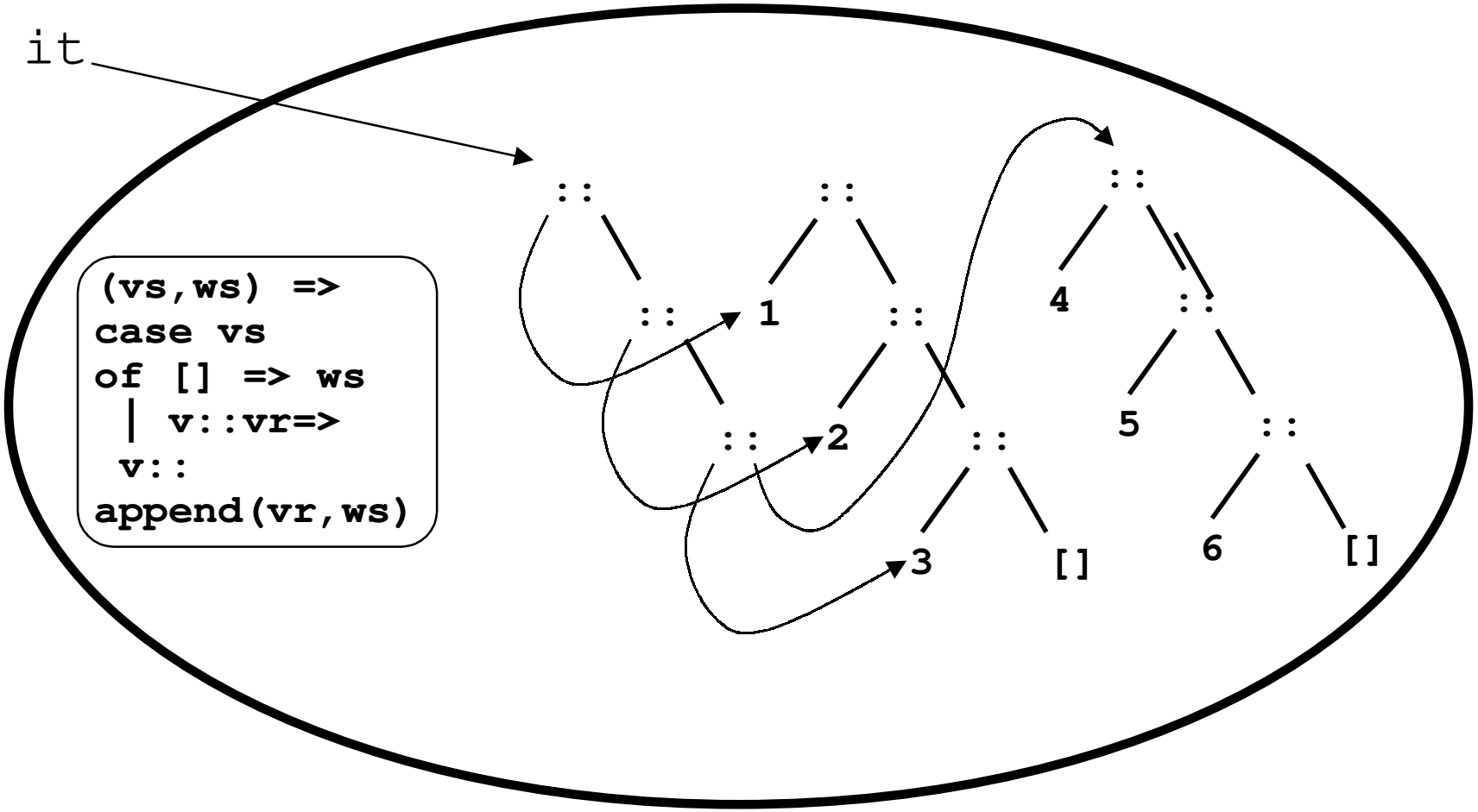

case xs
of []   => ys
 | v :: vr => v :: append(vr,ys)
thread
(vs,ws) =>
case vs
of [] => ws
 | v::vr=>v::append(vr,ws)
::
1
::
2
::
3
[]
::
4
::
5
::
6
[]
graph


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



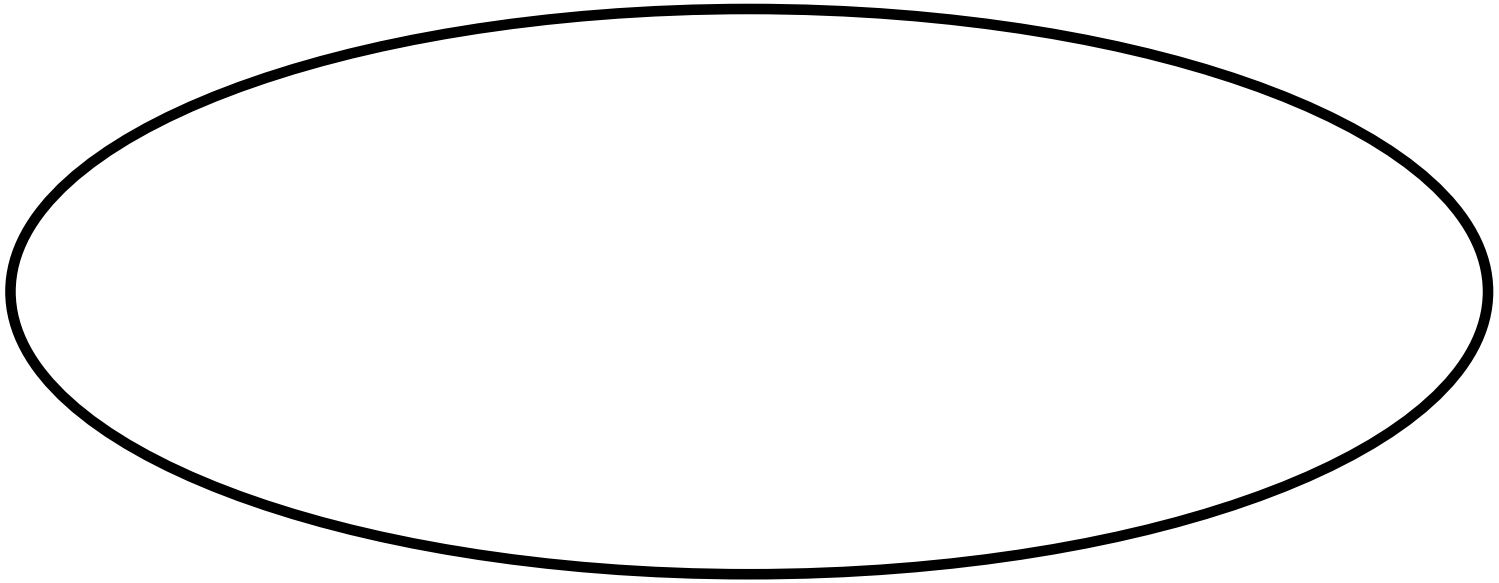
graph

Step 1: Explicit Return Arguments

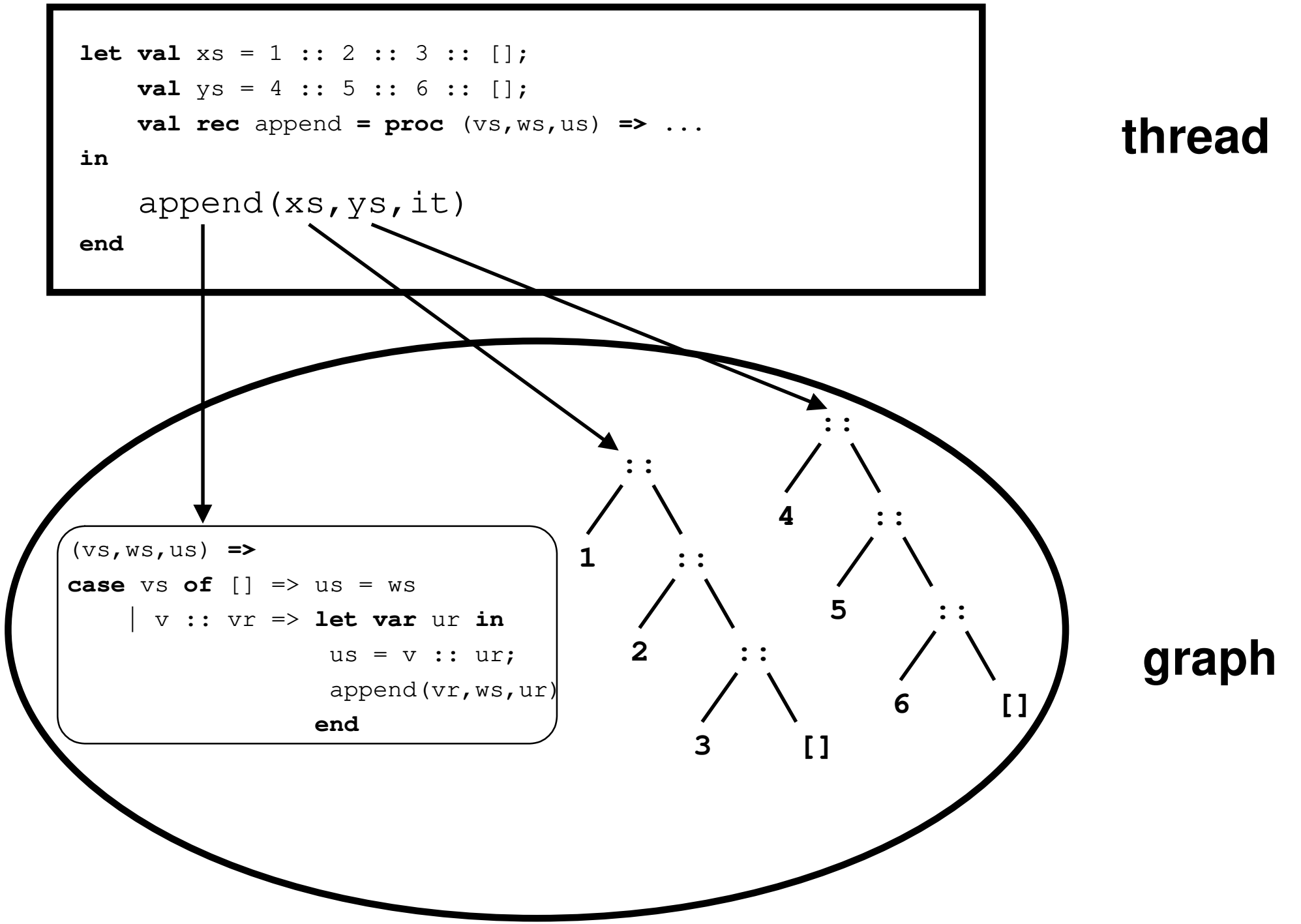
```
let val xs = 1 :: 2 :: 3 :: [];
    val ys = 4 :: 5 :: 6 :: [];
    val rec append = proc (vs,ws,us) =>
                         case vs
                         of []       => us = ws
                          | v :: vr => let var ur in
                                          us = v :: ur;
  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

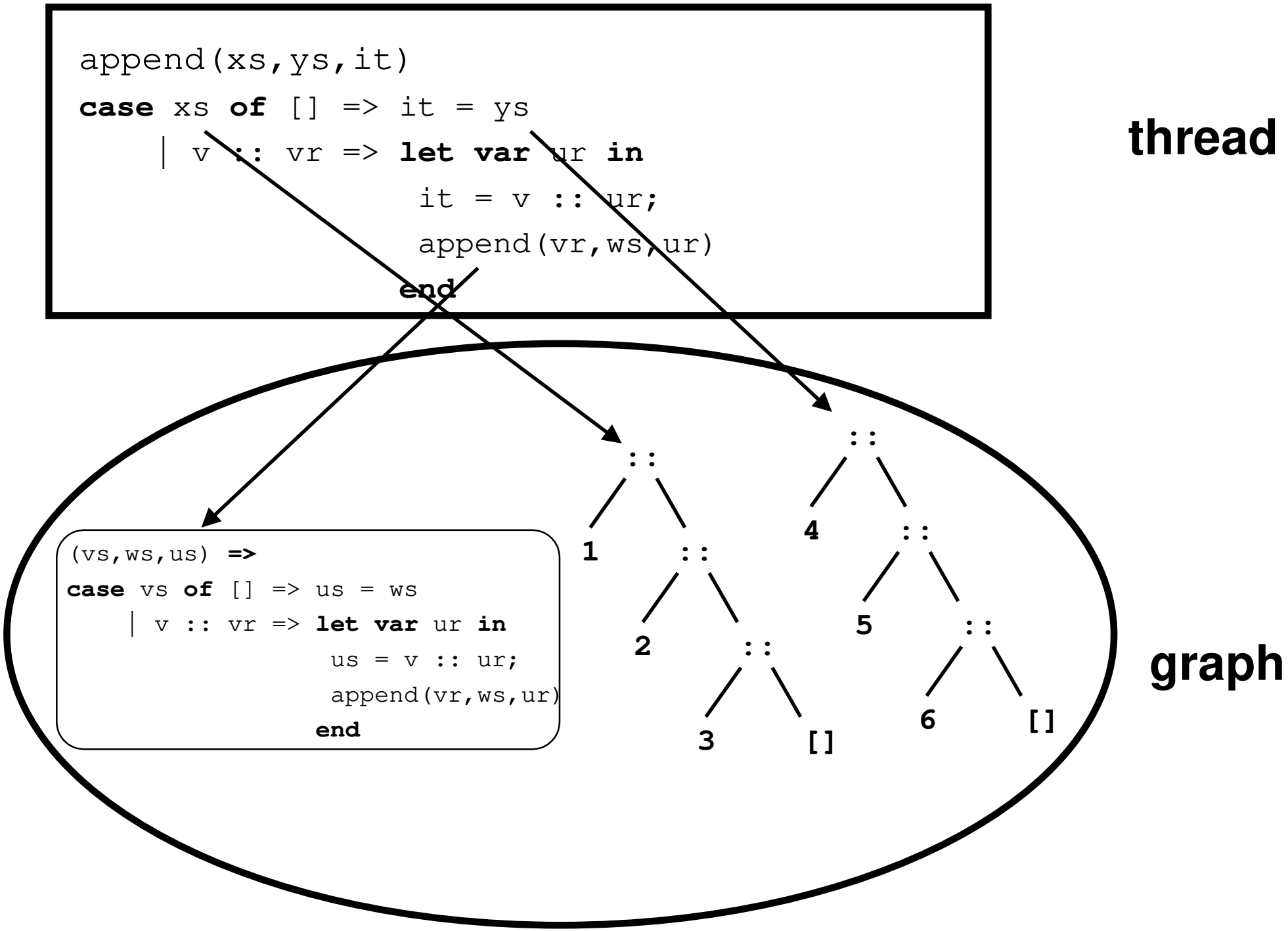
```
let val xs = 1 :: 2 :: 3 :: [];
    val ys = 4 :: 5 :: 6 :: [];
    val rec append =
         proc (vs,ws,us) =>
            case vs of []      => us = ws
                     | v :: vr => let var ur in
                                    us = v :: ur;
                                    append(vr,ws,ur)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

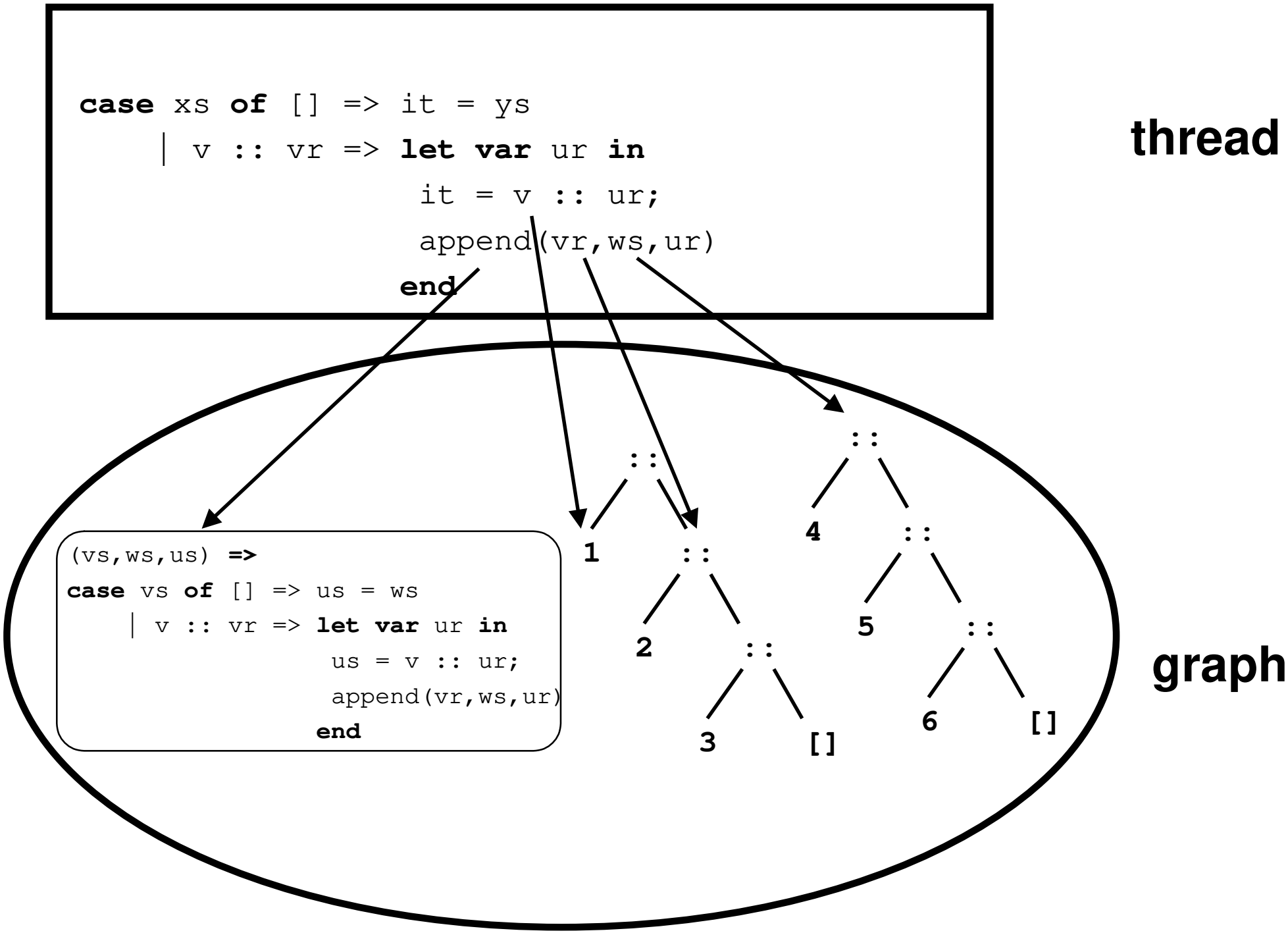
thread

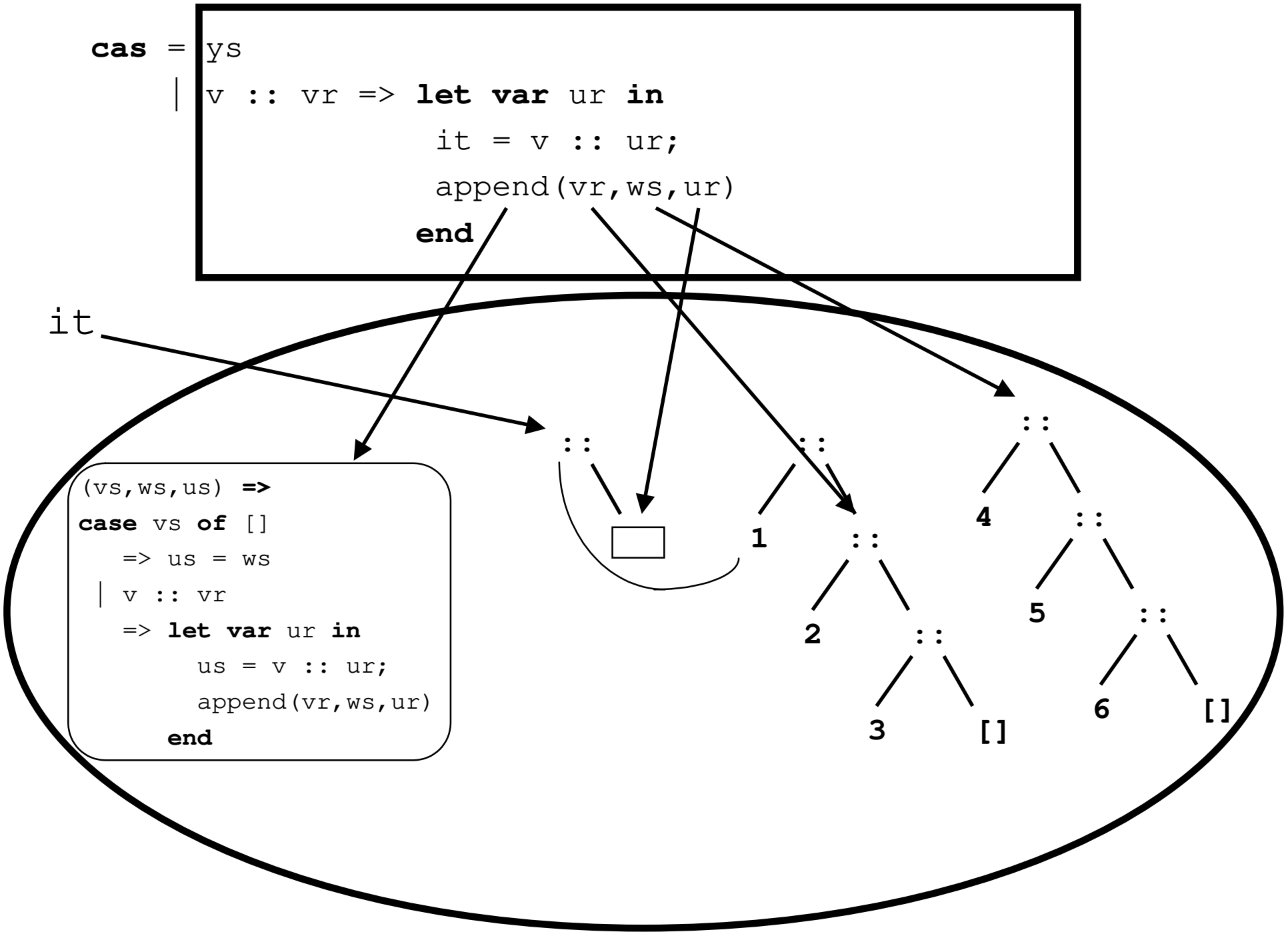


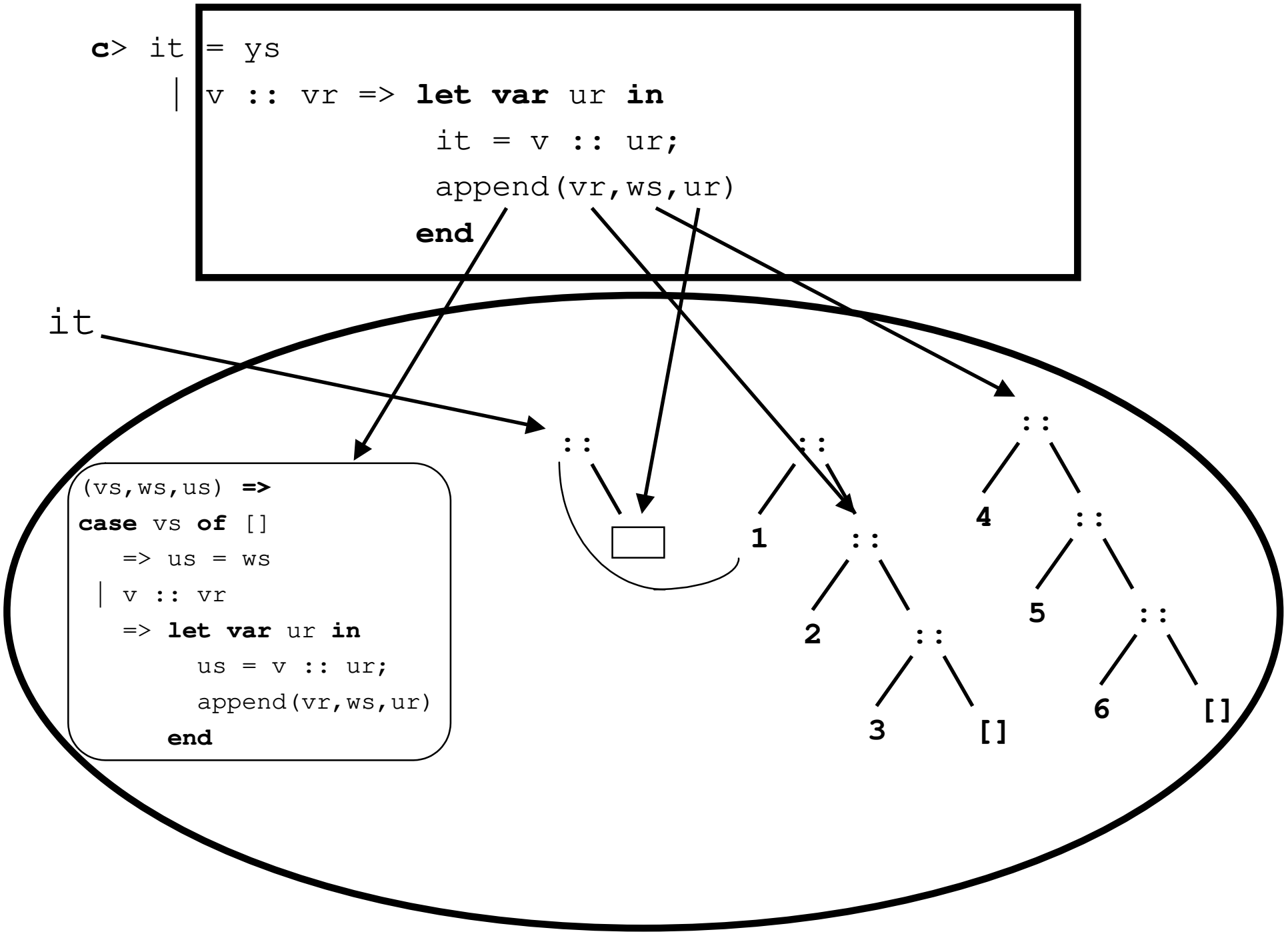
graph

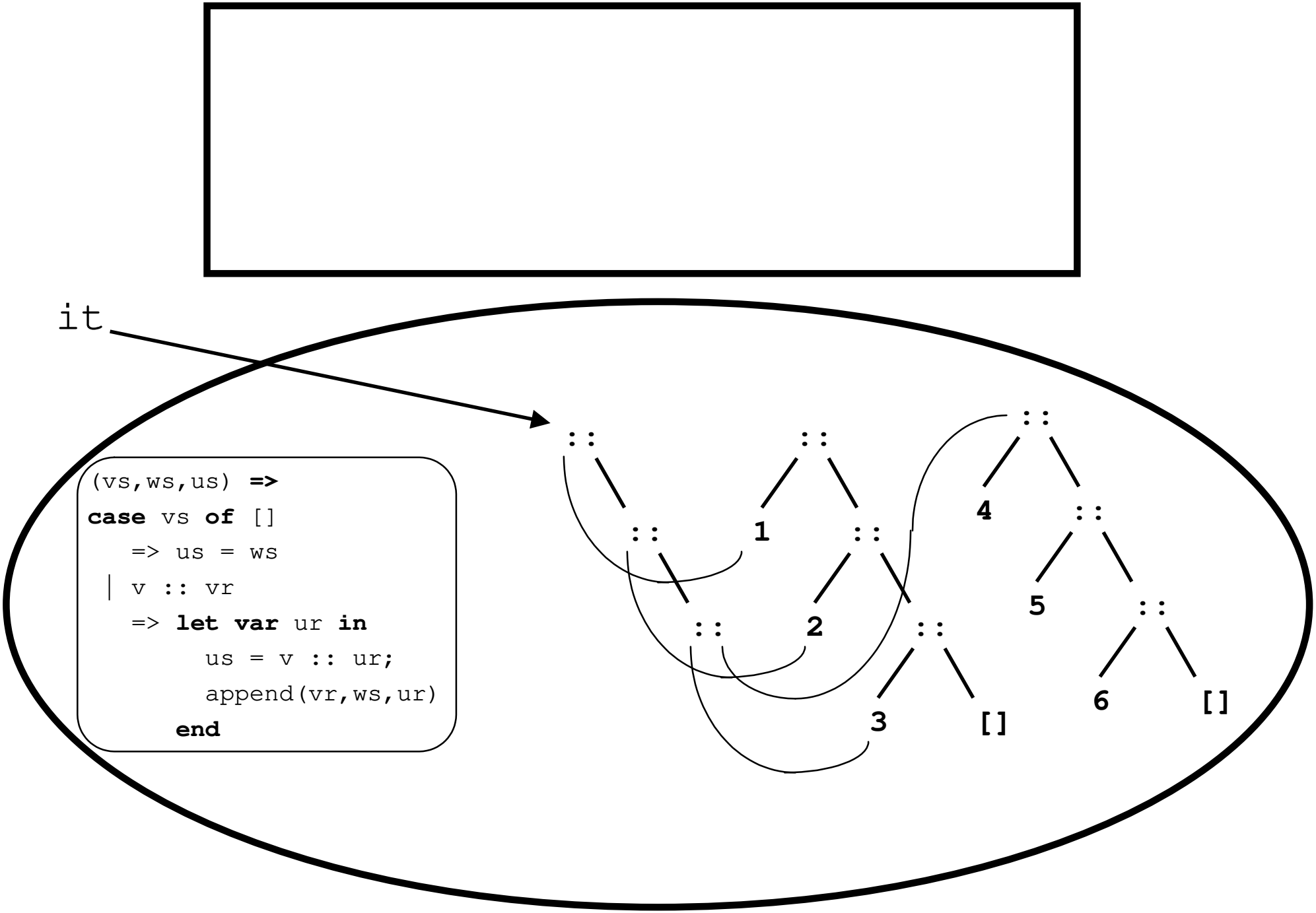

let val xs = 1 :: 2 :: 3 :: [];
val ys = 4 :: 5 :: 6 :: [];
val rec append = proc (vs,ws,us) => ...
in
append(xs,ys,it)
end
(vs,ws,us) =>
case vs of [] => us = ws
| v :: vr => let var ur in
us = v :: ur;
append(vr,ws,ur)
end
::
1
::
2
::
3
[]
::
4
::
5
::
6
[]
thread
graph


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

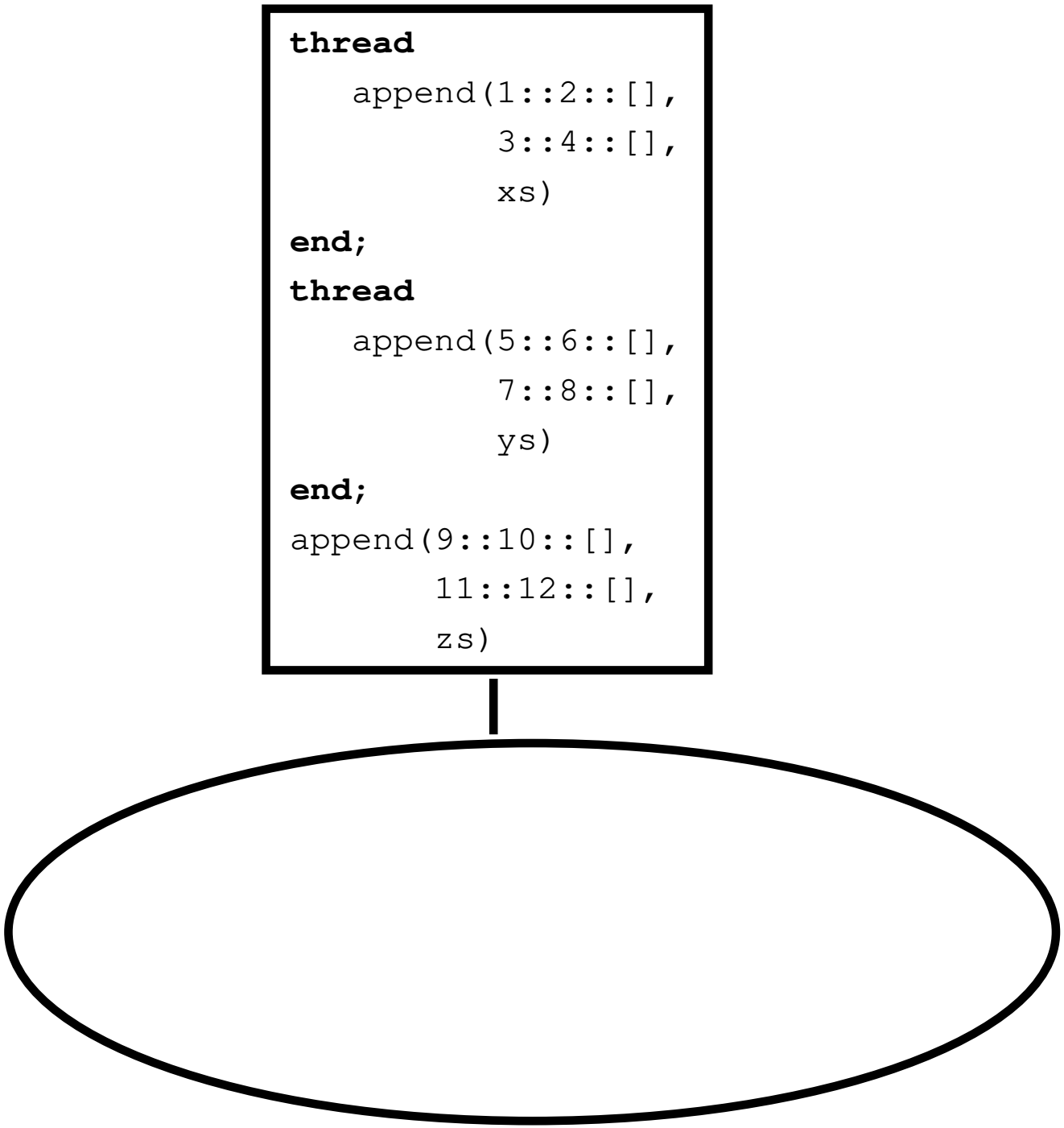

case xs of [] => it = ys
| v :: vr => let var ur in
it = v :: ur;
append(vr,ws,ur)
end
thread
(vs,ws,us) =>
case vs of [] => us = ws
| v :: vr => let var ur in
us = v :: ur;
append(vr,ws,ur)
end
::
1
::
2
::
3
[]
::
4
::
5
::
6
[]
graph


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
[]

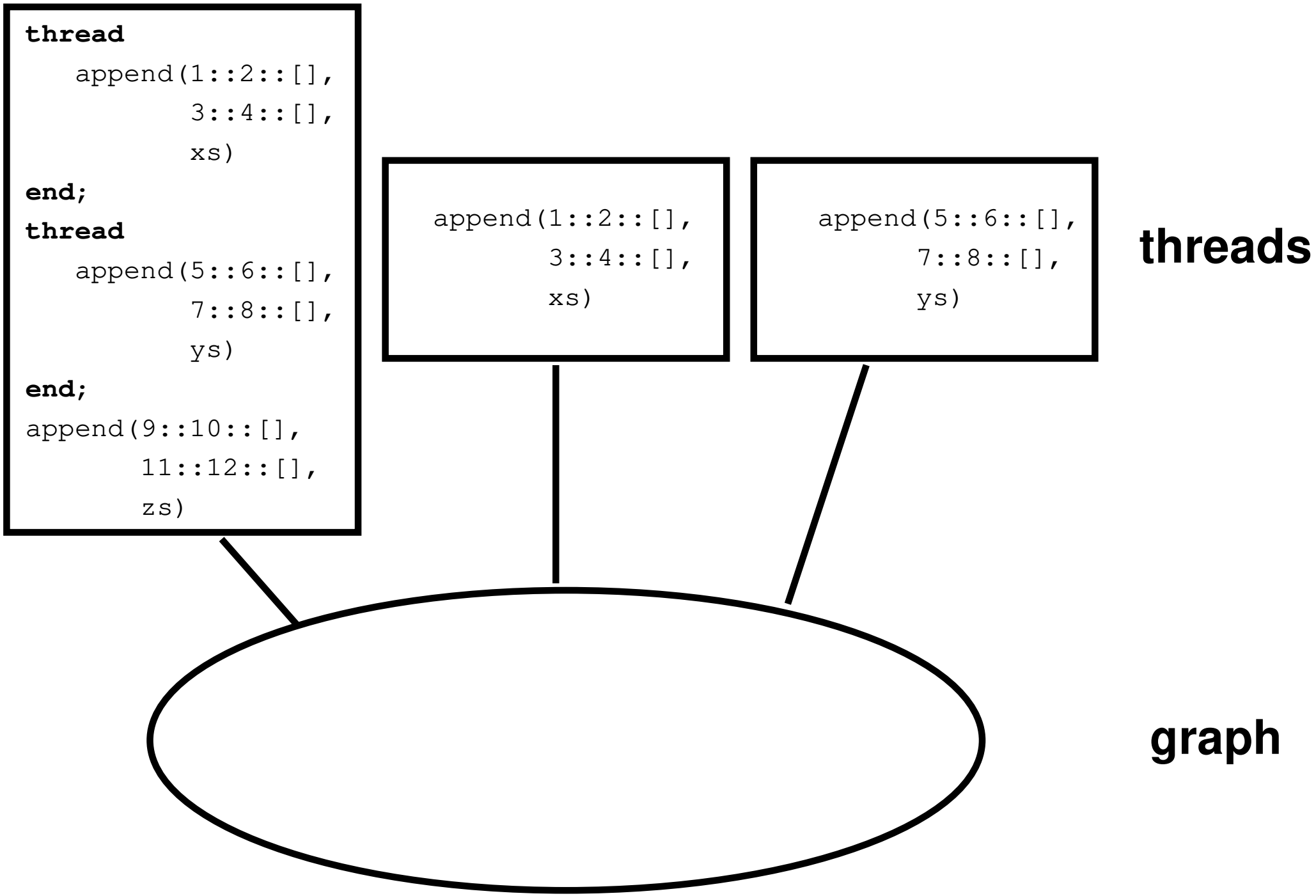

it
(vs,ws,us) =>
case vs of []
=> us = ws
| v :: vr
=> let var ur in
us = v :: ur;
append(vr,ws,ur)
end
::
::
::
::
::
::
1
2
3
[]
4
5
6
[]

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

Data-driven Synchronization

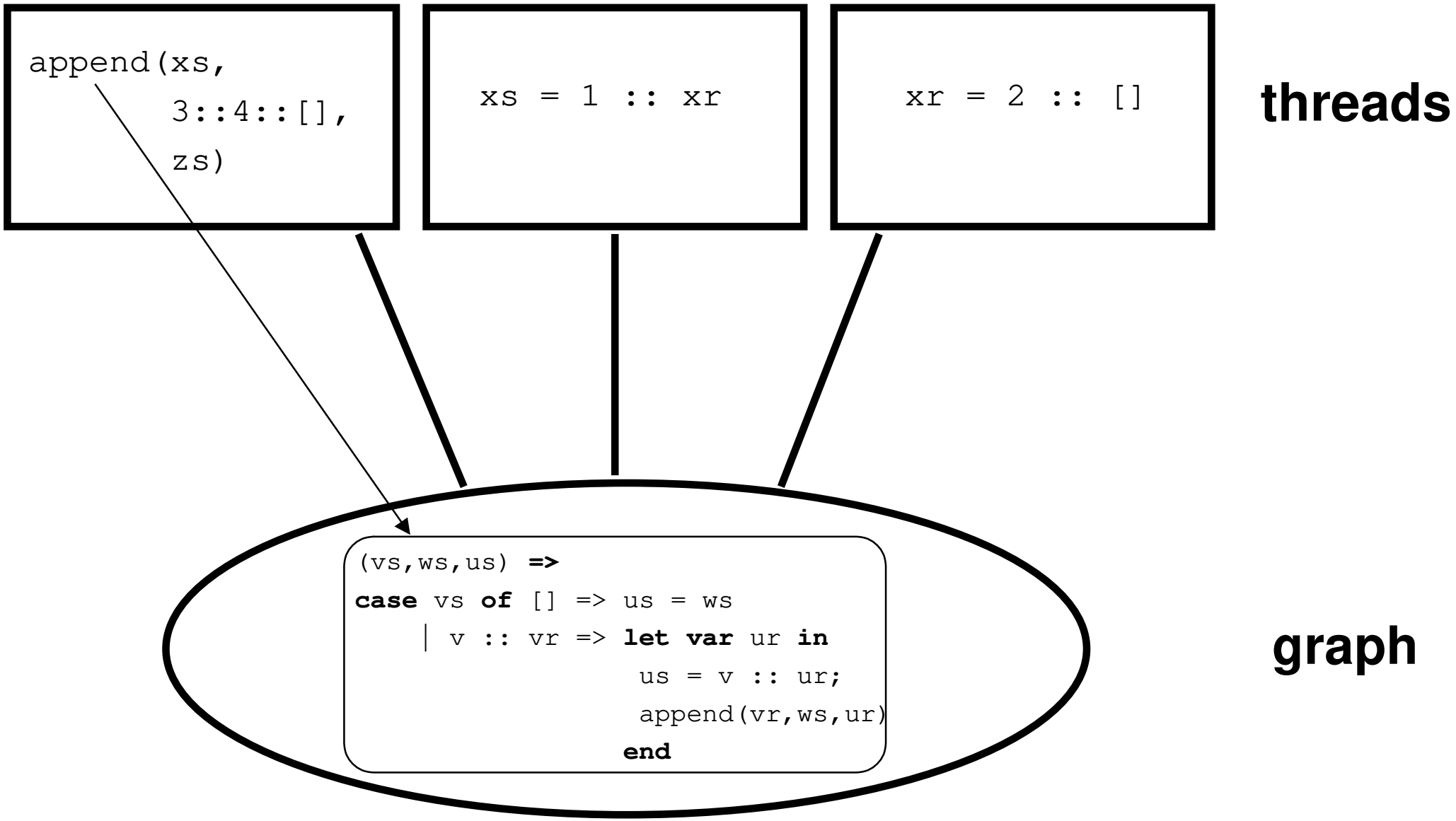
```
thread
   append(xs,
          3::4::[],
          zs)
end;

thread
   xs = 1 :: xr
end;

thread
   xr = 2 :: []
end;
```

thread

```
(vs,ws,us) =>
case vs of [] => us = ws
    | v :: vr => let var ur in
                   us = v :: ur;
                   append(vr,ws,ur)
                 end
```

graph

Data-driven Synchronization



Overview

- From Functional to Concurrent Constraint Programming
- The Language Oz
- Sequential Object-oriented Programming in Oz
- Synchronization Techniques
- Implementation of Objects in Oz
- Demo

The Language Oz

- **Lexically scoped Higher-order Programming**
- **Syntactic Support for Functional Programming**
- **Ubiquitous Data-flow Synchronization**
- **Explicit State**

Lexically Scoped Higher-order Programming

SML

```
let val makeAdder =
  fn (x) =>
    fn (y) =>
      x + y
in
  (makeAdder 1) 3
```

Oz

```
local
  fun {MakeAdder X}
    fun {$ Y}
      X + Y
    end
  end
in
  {{MakeAdder 1} 3}
end
```

Syntactic Support for Functional Programming

Full Oz

```
local
   fun {MakeAdder X}
      fun {$ Y}
         X + Y
      end
   end
in
   {{MakeAdder 1} 3}
end
```

Pure Oz

```
local MakeAdder in
   proc {MakeAdder X OutProc}
      proc {OutProc Y Z}
         Z = X + Y
      end
   end
   local P in
      {MakeAdder 1 P}
      {P 3 OutValue}
   end
end
```

Ubiquitous Dataflow Synchronization

- **Conditionals**

```
local Xs in
   thread {Compute Xs} end
   case Xs of X|Xr then {P X} end
end
```

- **Arithmetics**

```
local X in
   thread {Compute X} end
   ... X + Y = Z ...
end
```

- **Procedure Application**

```
local P in
   thread {Compute P} end
   ... {P X Y} ...
end
```

- **Wait**

```
local X in
   thread {Compute X} end
   ... {Wait X} ...
end
```

Explicit State

Oz

```
declare Counter =
local C = {NewCell 0}
in
  fun {$}
    OldValue={Access C}
    NewValue=OldValue + 1
  in
    {Assign C OldValue}
    NewValue
  end
end
{Show {Counter}}  % 1
{Show {Counter}}  % 2
```

Implicit State

Scheme

```
(define counter
(let ((c 0))
   (lambda ()
      (let ((oldValue c)
            (newValue
              (+ oldValue 1))
         (begin
            (set! C newValue)
            newValue))))

(counter)   ;; 1
(counter)   ;; 2
```

Atomic Read and Write

```
declare Counter =
local C = {NewCell 0}
in
   fun {$}
      OldValue NewValue
in
      {Exchange C OldValue NewValue}
      NewValue = OldValue + 1
      NewValue
end
{Show {Counter}}  % 1
{Show {Counter}}  % 2
```

Background of Oz

- **Originated in Programming Systems Lab (PSL), Saarbruecken, Germany, under Gert Smolka**
- **Start of development: 1992**
- **Programming System DFKI Oz 2.0 (public domain)**
- **Currently developed by PSL, SICS (Stockholm) under Seif Haridi and UCL (Louvain-la-Neuve, Belgium) under Peter Van Roy**
- **Development direction:**

  Distributed Programming (system Mozart)

DFKI Oz 2.0

- distribution: compiler, emulator, programming interface, libraries, documentation, binaries for Unix (Solaris, DEC Alpha, IRIX,...) and Windows95/NT

- performance of emulator: competitive with state-of-the-art byte-code emulators for Java, Scheme, Prolog, etc

- windows programming using Tcl/Tk

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

Tools for DFKI Oz

- Oz Browser: textual representation of rational feature trees
- Oz Explorer: Visualization and interactive exploration of search trees
- Gump: Flex/Bison for Oz

- Ozcar: Graphical source-level debugger (Mozart only)

Documentation: DFKI Oz User. s Manual,
Gump: A Frontend Generator for Oz

Some Application Projects Using Oz

- **LISA: Linguistically Driven Inference for Semantic Evaluation**
- **CHORUS: Constraint-based Higher-order Representations for Underspecified Semantics**

- **Data Bus Scheduling**
- **College Timetabling**
- **Friar Tuck: Sport Tournament Scheduling**

- **Dive-Oz: Distributed Virtual Reality in Oz**
- **CozmOz Saarbruecken: RoboCup**

- **Kimba: Automated Higher-order Theorem Proving**

More information: `http://www.ps.uni-sb.de/oz/projects`

Overview

- From Functional to Concurrent Constraint Programming
- The Language Oz
- Sequential Object-oriented Programming in Oz
- Synchronization Techniques
- Implementation of Objects in Oz
- Demo

Sequential Object-oriented Programming in Oz

- Functional programming with state can express sequential object-oriented programming
- Examples: Abelson/Sussman, CLOS, Objective CAML
- Objects in Oz similar to these approaches
- Syntactic support for classes

A Counter in Oz Syntax

```
declare
class Counter
   from BaseObject
   attr val: 0
   meth inc
      val <- @val + 1
   end
   meth printValue
      {Print @val}
   end
end
C = {New Counter printValue}
{C inc}
{C printValue}
```

A Counter using Atomic Read and Write

```
declare
class Counter
   from BaseObject
   attr val: 0
   meth inc
      Old New
      Old = val <- New
      New = Old + 1
   end
   meth printValue
      {Print @val}
   end
end
C = {New Counter printValue}
{C inc}
{C printValue}
```

Advanced Features

- Keyword parameters, optional parameters with defaults
- first-class attribute names
- first-class method names
- first-class classes
- first-class messages

Details in: Objects for Concurrent Constraint Programming
Kluwer Academic Publishers, 1998

Overview

- From Functional to Concurrent Constraint Programming
- The Language Oz
- Sequential Object-oriented Programming in Oz
- Synchronization Techniques
- Implementation of Objects in Oz
- Demo

Synchronization Techniques

- **Dataflow Synchronization**
- **Mutual Exclusion**
- **Semaphores**
- **Thread-reentrant Locking**
- **Active Objects**

Dataflow Synchronization

```
class Producer
   meth produceStream(Xs)
      X|Xr = Xs
    in
      {Produce X}
      {self produceStream(Xr)}
   end
end
```

```
class Consumer
   meth consumeStream(Xs)
      case Xs of X|Xr
      then
         {Consume X}
         {self consumeStream(Xr)}
      end
   end
end
```

```
declare Stream
thread {New Producer produceStream(Stream)} end
thread {New Consumer consumeStream(Stream)} end
```

Mutual Exclusion

```
class SafePassage
   from Passage
   attr token: 1
   meth pass
      {OpenFirst} {Enter} {CloseFirst}
      {OpenSecond} {Exit} {CloseSecond}
   end
   meth passSafely
      Old New in
      Old = token <- New
      {Wait Old}
      {self pass}
      New = Old
   end
end
```

General Locking Scheme

```
class Lock
   attr token: 1
   meth lock(Code)
      Old New in
      Old = token <- New
      {Wait Old}
      {Code}
      New = Old
   end
end
```

Thread-Reentrant Locking

```
class ReentrantLock
   attr token: 1
        lockingThread: 0
   meth lock(Code)
      This = {ThisThread}
   in
      case @lockingThread == This
      then {Code}
      else New Old in
         {Wait Old}
         lockingThread <- This
         {Code}
         lockingThread <- 0
         New = Old
      end
   end
end
```

Syntactic Support for Thread-Reentrant Locking

```
class Passage
   meth passSafely
      lock
         {OpenFirst} {Enter} {CloseFirst}
         {OpenSecond} {Exit} {CloseSecond}
      end
   end
end
```

Active Objects: Ports

```
fun {NewPort Ms}
   {NewCell Ms}
end

proc {Send Port M}
   Ms Mr in
   {Exchange Port Ms Mr}
   Ms = M | Mr
end

declare MessageStream P in
P = {NewPort MessageStream}
{Send P push(1)}
{Send P pop(X)}
{Send P push(2)}
```

Active Objects: Servers

```
fun {MakeServer Object}
   Stream
   proc {Serve Ms}
      case Ms of M|Mr
      then {Object M} {Serve Mr}
      end
   end
in
   thread {Serve Stream} end
   {NewPort Stream}
end

declare StackServer in
StackServer = {MakeServer MyStack}
{Send StackServer push(1)}
{Send StackServer pop(X)}
{Send StackServer push(2)}
```

Overview

- From Functional to Concurrent Constraint Programming
- The Language Oz
- Sequential Object-oriented Programming in Oz
- Synchronization Techniques
- Implementation of Objects in Oz
- Demo

General Implementation Techniques for Oz

- Virtual Machine interpreting byte code
- Stop-and-copy garbage collection
- transparent distribution implemented using TCP/IP

Implementation of Objects in Oz

Adaptation of implementation techniques for sequential object-oriented languages

- **Optimized memory layout**
- **keep self in register**
- **compile-time garbage collection for first-class messages**
- **method lookup caching and inline caching**

Performance similar to JVM (Sun JDK 1.0) and CLOS (Allegro CL 4.3, native code)

Overview

- From Functional to Concurrent Constraint Programming
- The Language Oz
- Sequential Object-oriented Programming in Oz
- Synchronization Techniques
- Implementation of Objects in Oz
- Demo

Summary

Oz combines the expressivity of modern functional and object-oriented languages with the power of logic variables.

Demo

- **Lift simulation**
- **Freight transportation**
- **Sport scheduling**