# Modules, States, Uniqueness, and Non-Determinism

Peter Achten, Sven-Bodo Scholz

Software Science
Radboud University Nijmegen

Radboud University

# Recap Tensor Comprehensions and With-Loops

```
res = { iv -> expr (iv) | iv < [3,4];
        iv -> def       | iv < [3,4] };
```

```
shape (res) == outer-shp ++ inner-shp
            == [3,4]     ++ shape (expr (iv))
            == [3,4]     ++ shape (def)
```

```
res = with {
        ([0,0] <= iv < [3,4]) : expr (iv);
      } : genarray( [3,4], def);
```

# Recap Shifting Between Inner and Outer Shape

```
res = { iv -> [0,0,0,0,0]               | iv < [3,4] };

res = { iv -> 0                         | iv < [3,4,5] };

res = { iv -> genarray ([4,5], 0)       | iv < [3] };

res = { iv -> genarray ([3,4,5], 0)| iv < [] };
```

```
res == reshape ([3,4,5], genarray ([60], 0))
```

# Recap: With-Loops and Concurrency

```
res = with {
        ([0,0] <= iv < [3,4]) : expr (iv);
      } : genarray( [3,4], 42);
```

lexical scoping => ***expr*** can only refer to variables defined before this definition!
side-effect-free => ***expr*** neither relies on some shared state nor does it change some shared state!
for each value of `iv` we compute ***expr*** `(iv)` exactly once!

⇒ Semantics guarantees that the order of evaluation does not affect the result
⇒ Parallelism is possible!

# Why do we Need Modules?

- Code Reuse
- Namespaces
- FFI (Foreign Function Interface)

# Challenges of Modules?

- Separate Compilation vs Whole-World Compilation
  - whole-world compilation => better optimisation possible
  - separate compilation => faster compilation
  - separate compilation => enables the use existing libraries
- Overloading
  - How do we deal with overloading across modules?
- FFI (Foreign Function Interface)
  - How can we get data from one language to another?
  - How can we work with stateful libraries?

# Modules and Namespaces

```
Module A;
export all;

int foo()
{...}

int bar()
{...}
```

Defines a *namespace* “A”

```
int main()
{
  x = A::foo();
  y = A::bar();
     ...
}
```

refers to “A”

```
int foo()
{...}

int main()
{
  x = A::foo();
  y = A::bar();
  z = foo();
     ...
}
```

lives in “MAIN”

# Using Modules...

```
Module A;
export all;

int foo()
{...}

int bar()
{...}
```

makes all symbols of "A" directly *usable* in "MAIN"

```
use A: all;

int main()
{
  x = foo();
  y = bar();
  ...
}
```

```
use A: all;

int foo()
{...}

int main()
{
  x = A::foo();
  y = bar();
  z = foo();
  ...
}
```

```
use A: all
  except {foo};

int foo()
{...}

int main()
{
  x = A::foo();
  y = bar();
  z = foo();
  ...
}
```

# Using Modules… no accidental overloading!

```
Module A;
export all;

int
foo(int[*] a)
{...}

int bar()
{...}
```

makes all symbols of "A" directly *usable* in "MAIN"

```
use A: all;

int main()
{
  x = foo(42);
  y = bar();
  ...
}
```

```
use A: all;

int foo(int a)
{...}

int main()
{
  x = A::foo(42);
  y = bar();
  z = foo(42);
  ...
}
```

```
use A: all
  except {foo};

int foo(int a)
{...}

int main()
{
  x = A::foo(42);
  y = bar();
  z = foo(42);
  ...
}
```

# Modules and Intended Overloading

```
Module A;
export all;

int foo (int[*] a)
{return 0;}
```

```
import A: all;

int foo (int a)                ?
{return 1;}

int main()
{
  x = foo (42);     1
  y = foo ([1]);    0
  ...
}
```

literally *imports* all definitions from “A”

# Modules and Overloading, Interesting Cases

```
Module A;
export all;

int foo (int[*] a)
{return 0;}
```

```
import A: all;

int foo (int a)
{return 1;}

int main ()
{
   x = foo (42);        1
   y = foo ([1]);       0
   y = A::foo (42);     0
   ...
}
```

# Modules and Overloading, Recursion

```
Module A;
use Array: all;
export all;

int[*] foo (int[*] a)
{return {[i]->foo(A[i])};}

int foo (int a)
{return 0;}
```

```
use Array: all;
import A: all;

int[.,.,.] foo (int[.,.,.] a)
{return {[i]->foo(a[i])+3};}

int[.] foo (int[.] a)
{return {[i]->foo(a[i])+1};}

int main ()
{
  x = foo ([42]);                    [1]
  y = A::foo ([42]);                 [0]
  xxx = foo ([[[42]]]);              [[[4]]]
  yyy = A::foo ([[[42]]]);           [[[0]]]
  ...
}
```

# Modules and Overloading

```
Module A;
export all;

int foo (int[*] a)
{return 0;}
```

```
import A: all;
use B: all except {foo};

int foo (int[.] a)
{return 1;}

int main ()
{
  ...
  x = foo (42);     0
  y = bar (42);     3
  z = B::foo (42);  2
  ...
}
```

inhibits imports but allows uses

```
Module B;
provide all;

int foo (int[*] a)
{return 2;}

int bar (int[*] a)
{return 3;}
```

# Example from the Stdlib:

Branch: **master** ▾ **Stdlib** / src / structures / **Structures.sac**

Find file | Copy path

**sbscholz** added Quaternion.sac in Structures in the extended-section of the Std... fd6a977 on 19 Sep 2017

2 contributors

21 lines (15 sloc) | 359 Bytes

Raw | Blame | History

```
module Structures;

import Array        : all;
import Char         : all;
import Bits         : all;
import String       : all;
#ifdef EXT_STDLIB
import Complex      : all;
import Quaternion   : all;
#ifndef SAC_BACKEND_MUTC
import List         : all;
import Color8       : all;
import Grey         : all;
#endif /* SAC_BACKEND_MUTC */
#endif

export all;



```

# SaC is stateless ????

- How can we allow for `print( a);` ????

- How do other functional languages do stateful computations?

  ➢ Monads in Haskell:
  type system enforces one linear chain of bind operations!

  ➢ Uniqueness Types in Clean:
  type system enforces a linear use of references!

# SaC's update in place

All these can be done in-place!

```
a = [1,2,3];
a = modarray( a, [0], 7);
a = modarray( a, [1], 42);
```

⇒ Observation I : each 'a' is used exactly once!

⇒ Observation II: none of the 'a' is "aliased"

⇒ Observation III: that is very close to ***uniqueness types***!

# Uniqueness Types without Uniqueness Types

**unq Terminal  print( unq Terminal terminal, int[*] a)**

terminal = print( terminal, a);
terminal = print( terminal, b);
terminal = print( terminal, c);

```
Class Terminal;

external classtype;
export all;

Terminal print (Terminal terminal, int[*] a)
{ . . . }
```

# More Hide-And-Seek: Reference Parameters

```
terminal  = print (terminal, a);
terminal2 = print (terminal, b);
terminal  = print (terminal, c);
```

```
print (terminal, a);
print (terminal, b);
print (terminal, c);
```

```
terminal = print (terminal, a);
terminal = print (terminal, b);
terminal = print (terminal, c);
```

```
Class Terminal;

external classtype;
export all;

void print (Terminal &terminal, int[*] a)
{ . . . }
```

# Hide-And-Seek for Pro's: Global Objects

print (terminal, a);
print (terminal, b);
print (terminal, c);

**Can we hide the terminal completely?**

print (a);
print (b);
print (c);

terminal = print (terminal, a);
terminal = print (terminal, b);
terminal = print (terminal, c);

```
Class Terminal;
external classtype;
export all;

objdef Terminal TheTerminal = createTheTerminal ();

void print (int[*] a)
{   print (TheTerminal, a); }
void print (Terminal &terminal, int[*] a)
{ . . . }
```

# Dealing with Global Objects

```
int[n:s] id ( int[n:s] a)
{
    print(a);
    return (a);
}

int[n:s] inc (int[n:s] a)
{
    b = id (a+1);
    return b;
}

int main ()
{
    a = inc ([1,2,3,4]);
    return a[0];
}
```

```
Term, int[n:s] id (Term T, int[n:s] a)
{
    T = print(T, a);
    return (T, a);
}

Term, int[n:s] inc (Term T, int[n:s] a)
{
    T, b = id (T, a+1);
    return (T, b);
}

int main ()
{
    T = createTheTerminal ();
    T, a = inc (T, [1,2,3,4]);
    return a[0];
}
```

# States in SaC

```
Class stack;
classtype int[100];
export all;

stack createStack()
{
   return to_stack (genarray ([100], 0));
}

stack push( stack s, int val)
{
    mys = from_stack (s);
    tos = mys[0] + 1;
    mys[0] = tos;
    mys[tos] = val;
    return to_stack (mys);
}
```

introduces new unique type “stack”

defines the representation of the type “stack”

make SaC object unique!

create SaC object from a unique one!

# States in SaC

```
Class stack;
classtype int[100];
export all;

stack createStack()
{...}
stack push( stack s, int val)
{ ...}
```

```
use stack: all;

int main()
{
  S = createStack();
  S = push( S, 10);
  S = push( S, 42);
  ...
}
```

```
use stack: all;

int main()
{
  S = createStack();
  S1 = push( S, 10);
  S2 = push( S, 42);
  ...
}
```

can be done destructively!

# Reference Parameters

```
Class stack;
classtype int[100];
export all;

stack createStack()
{...}
void push( stack &s, int val)
{ ...}
```

declares that a modified version of s is returned

```
use stack: all;

int main()
{
  S = createStack();
  push( S, 10);
  push( S, 42);
  ...
}
```

internally transformed

```
use stack: all;

int main()
{
  S = createStack();
  S = push( S, 10);
  S = push( S, 42);
  ...
}
```

# Global Objects!!

```
Class stack;
classtype int[100];
export all;

objdef stack myS= createStack();
stack createStack()
{...}
void push(int val)
{ ...myS...}
```

```
use stack: all;

int main()
{
  push( 10);
  push( 42);
  ...
}
```

internally transformed

```
use stack: all;

int main()
{
  myS = createStack();
  myS = push( myS, 10);
  myS = push( myS, 42);
  ...
}
```

# Example from the Stdlib:

Branch: master ▾ Stdlib / src / system / Terminal.sac

sbscholz partially adapted to the new object syntax.

1 contributor

23 lines (15 sloc) | 612 Bytes

```
class Terminal;

external classtype;

export all except { create_TheTerminal};


objdef Terminal TheTerminal = create_TheTerminal( );

/*
 *  The global object TheTerminal of class Terminal serv
 *  for a terminal screen. It is derived from the global
 *  order to represent this part or sub-world of the exe
 *  It is also used to synchronise the standard I/O stre
 *  and stderr.
 */

external Terminal create_TheTerminal( );
    #pragma effect World::TheWorld
    #pragma linksign[0]
    #pragma linkobj "src/Terminal/terminal.o"

```

Branch: master ▾ Stdlib / src / stdio / TermFile.sac

sbscholz created wrapper functions for TermFile.sac

2 contributors

207 lines (163 sloc) | 6 KB

```
class TermFile;

external classtype;

use String : {string};
use Terminal : { TheTerminal };

export all except { createStdIn, createStdOut, createStdErr};

objdef TermFile stdin = createStdIn();

objdef TermFile stdout = createStdOut();

```

```
external void  printf(string FORMAT, ...);
    #pragma effect TheTerminal, stdout
    #pragma linkname "SACprintf_TF"
    #pragma linkobj "src/TermFile/printf.o"
    /*
     * Print formatted output to STREAM which must be open for writing.
     * The syntax of format strings is identical to that known from C.
     * This function may be used to print values of types
     * char, string, int, float, and double.
     */

```

```

external TermFile createStdOut();
    #pragma effect TheTerminal
    #pragma linkname "SAC_create_stdout"
    #pragma linkobj "src/TermFile/stdstreams.o"
    #pragma linksign [0]
```

# Reference Counting and Uniqueness Types ala SaC vs Ownership in Rust

SaC: int[.] myMod( int[.] a) …

Rust: fn myMod(a: &mut int) …

*=> may need explicit copy!*

stack push( stack s, int val) …

fn push( stack: &mut int,  val: int)
    -> &mut int {…}

int inspect( &stack s) …

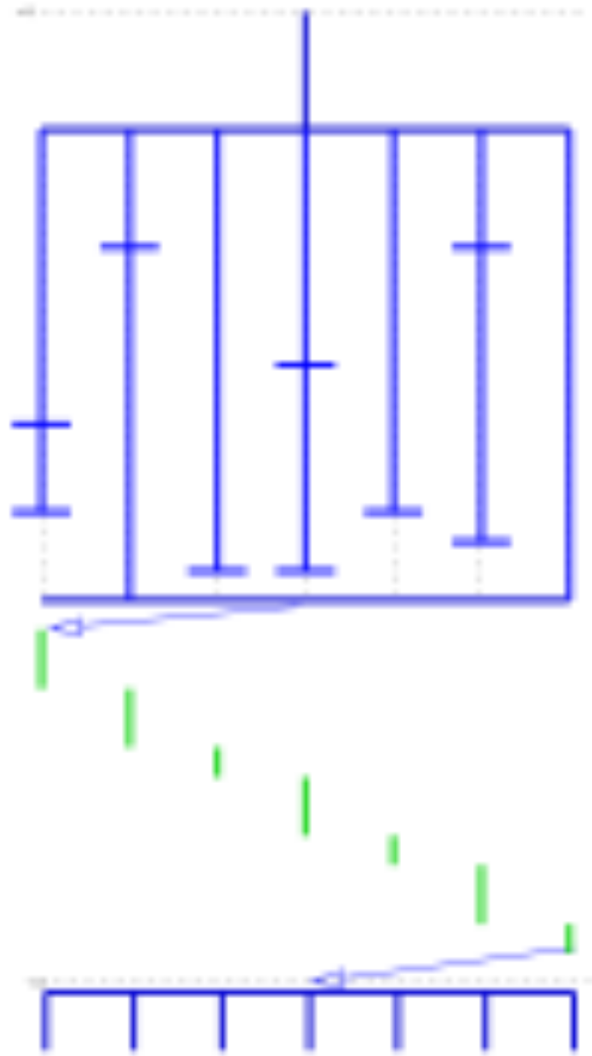
fn inspect( stack: & int) -> int {…}

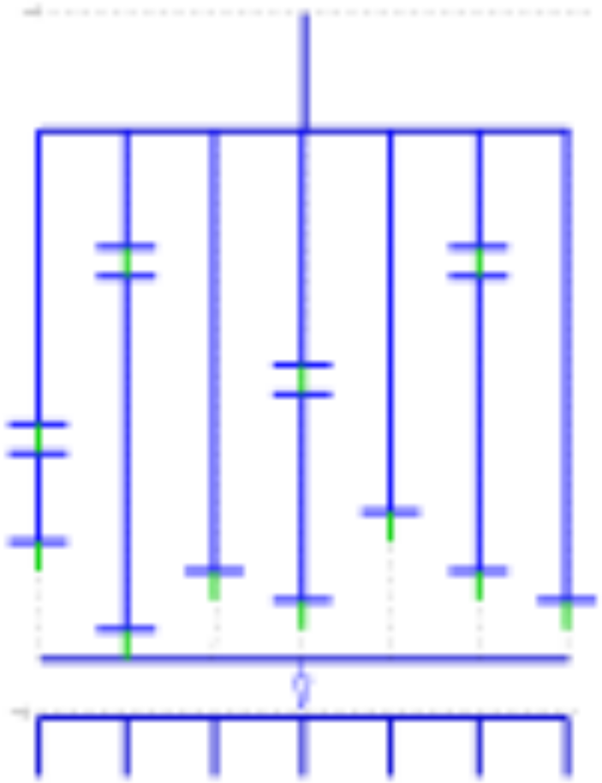
# Uniqueness / Mutable references and Parallelism

```
int[4]  myNewFun (int[2] iv)
{
    //   complex code
    print( res);  // just for debugging !!
    return res;
}

int main()
{
   …
   a = with {
          ([0,0] <= iv < [100,100]) : myNewFun (iv);
        } : genarray( [100,100], def);
   …
}
```

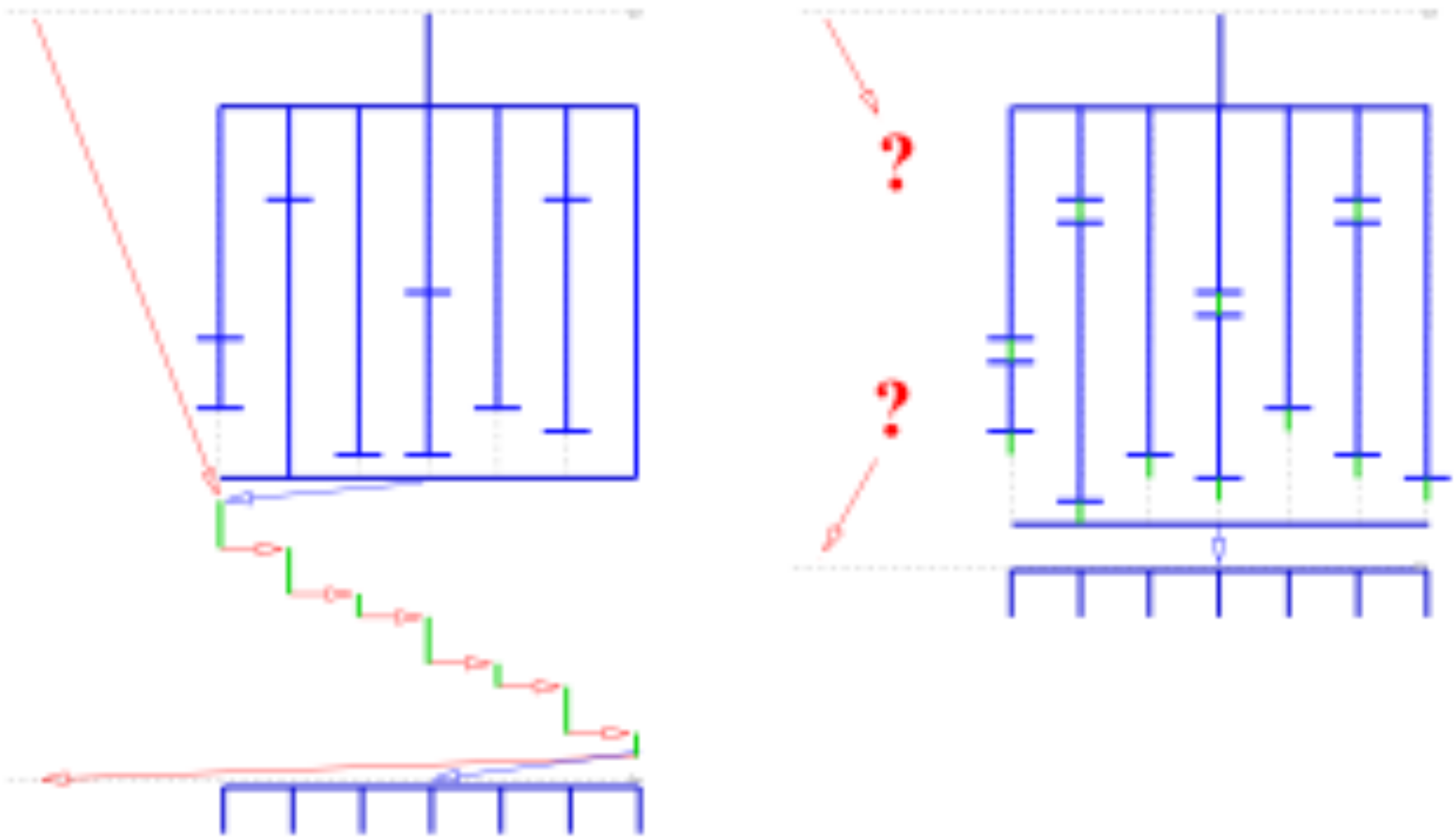
# Asynchronous I/O?



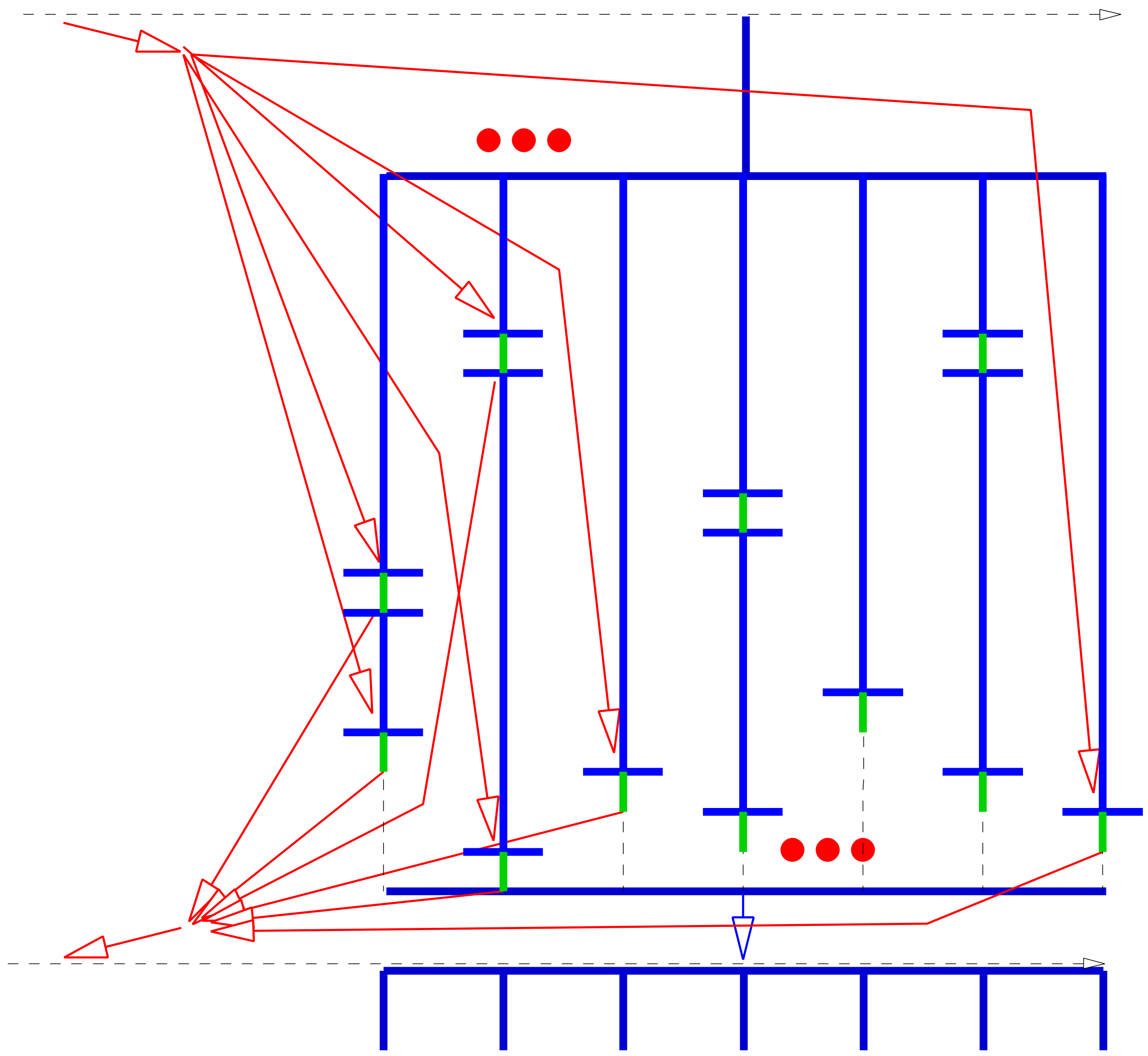


- scales with #cores
- improves debugging
- enables visualisation

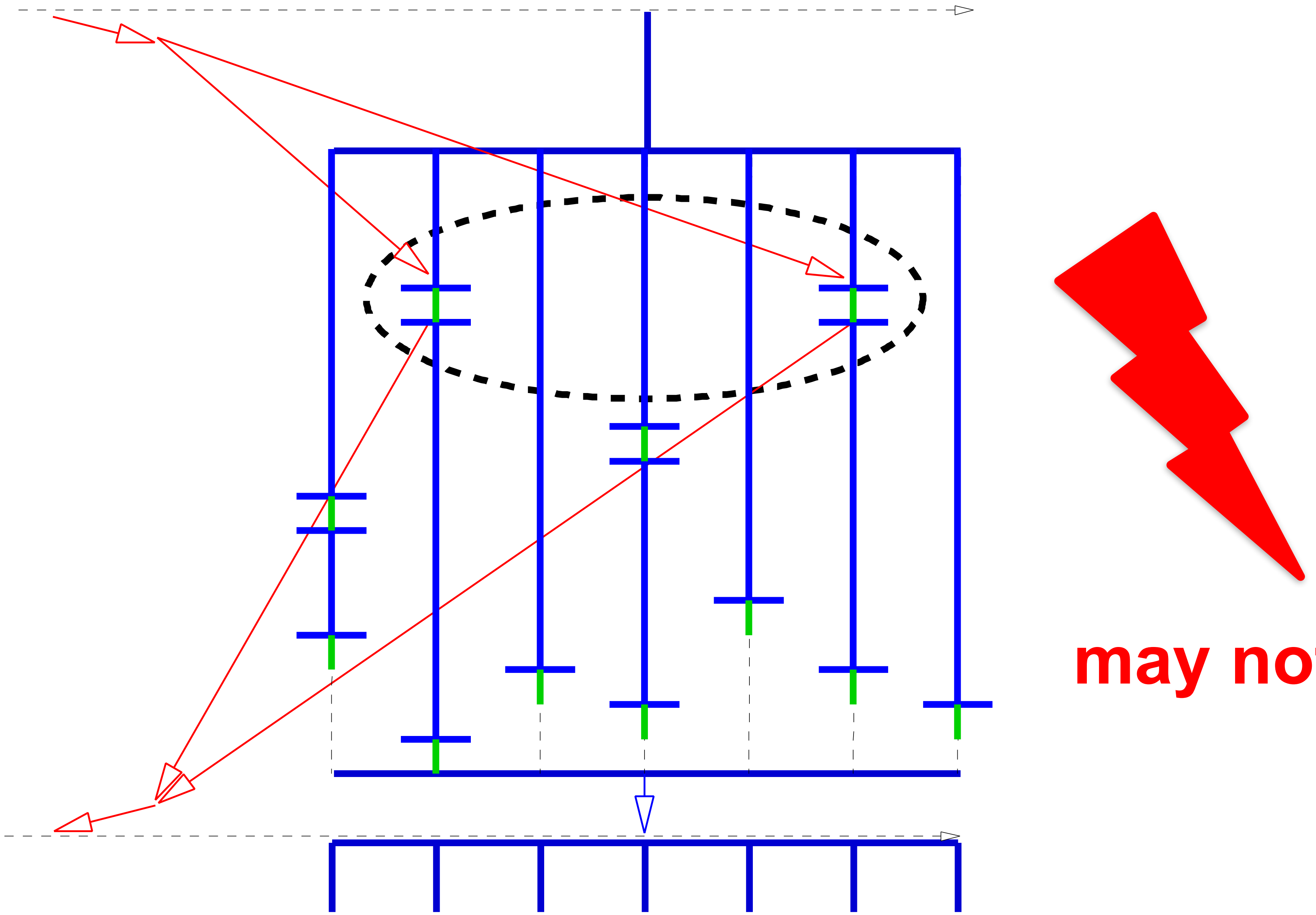
# But how to model the data dependencies?



# Attempt #1: Split State

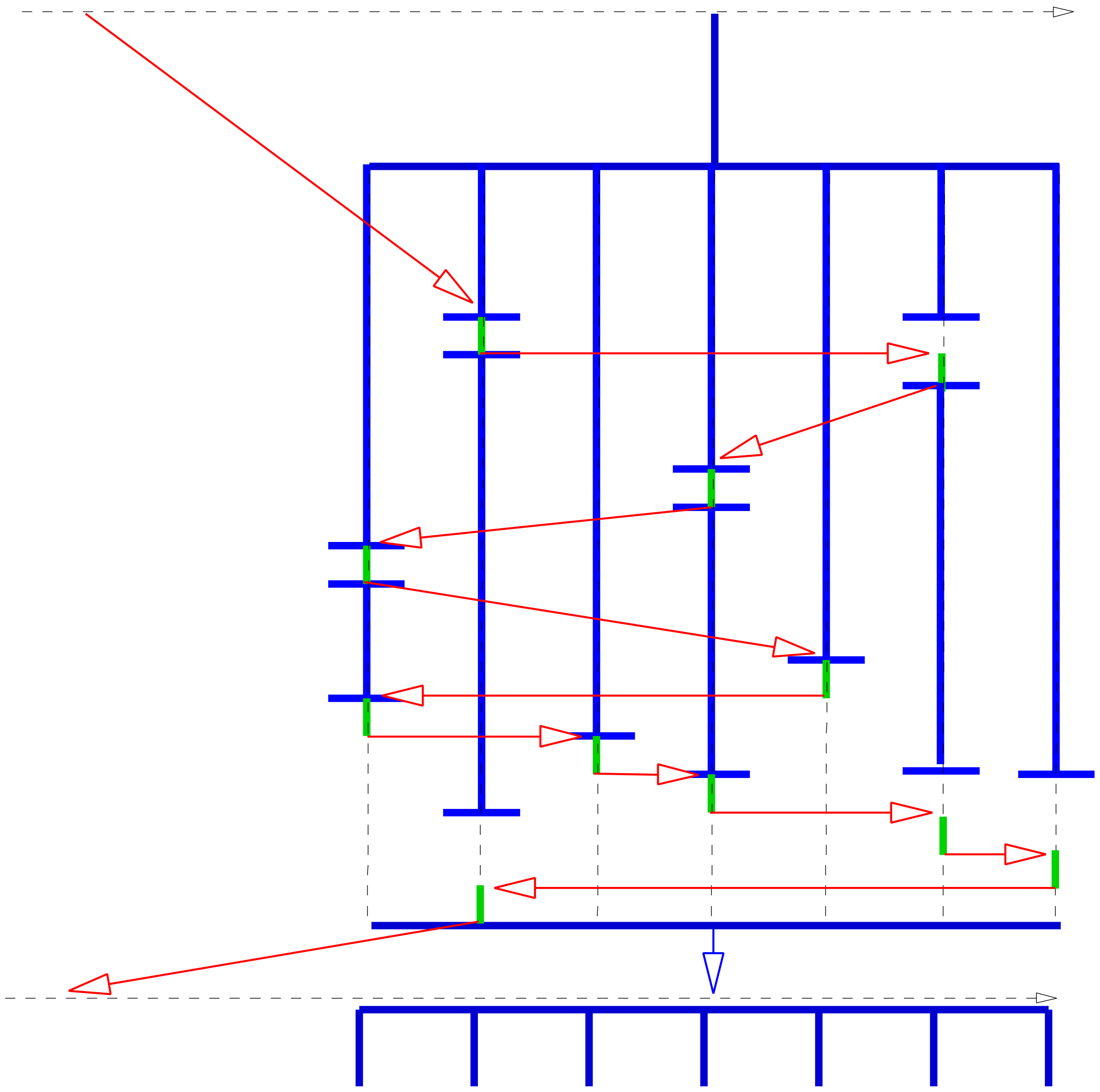


# Attempt #1: Split State



**may not be possible!**

# Solution: Non-Deterministic Order!



**[HerSchoGrel09]**
**Controlling chaos — on safe side-effects in data-parallel operations**

S. Herhut, S.-B. Scholz, C. Grelck (2009).
In *4th Workshop on Declarative Aspects of Multicore Programming (DAMP'09), Savannah, USA*. pp. 59–67. ACM Press.

## Practical Consequence

```
with {
   ([0] <= [i] <[10]) {
      printf( “Hi, I am # %d\n”, i);
   }
} : void
```

is legal SaC!!

# Practical Consequence

```
with {
   ([0] <= [i] <[10]) {
      printf( "Hi, I am # %d\n", i);
   }
} : void ;
```

translates into

```
stdout = with {
            ([0] <= [i] <[10]) {
              stdout = printf( stdout,
                               "Hi, I am # %d\n", i);
            } : stdout;
          } : propagate( stdout);
```

# Controlled Side-Effects

```
double[n] findSolutions (int[m] moves)
{
  if (isSolution (moves)) {
    res = [computeValue (moves)];
  } else {
    possible_moves = findMoves (moves);
    res = with {
             ([0] <= [i] < shape (possible_moves)) {
               incNumTries ();
             } : findSolutions (moves++possible_moves[i]);
          } : fold (++, []);
  }
  return res;
}
```

## Non-Determinism

```
double[n] findSolutions (int[m] moves)
{
  if (isSolution (moves)) {
    res = [ computeValue (moves)];
    setSolFound ();
  } else {
    possible_moves = findMoves (moves);
    res = with {
             ([0] <= [i] < shape (possible_moves)) {
               done = solFound();
               if (!done) incNumTries ();
             } : done? [] : findSolutions (moves++possible_moves[i]);
          } : fold (++, []);
  }
  return res;
}
```