Thesis Proposal:

Lattice-based Data Structures for Deterministic Parallel and Distributed Programming

Lindsey Kuper

December 6, 2013

Outline for this talk

- The problem and existing approaches
- Our approach: LVars
- Quasi-determinism with LVars
- The LVish library
- Joining forces: LVars and CRDTs
- Research plan

Outline for this talk

- The problem and existing approaches
- Our approach: LVars
- Quasi-determinism with LVars
- The LVish library

} *already done*

- Joining forces: LVars and CRDTs
- Research plan

Outline for this talk

- The problem and existing approaches
- Our approach: LVars
- Quasi-determinism with LVars
- The LVish library
- Joining forces: LVars and CRDTs
- Research plan

already done

still to do

Outline for this talk

- The problem and existing approaches
- Our approach: LVars
- Quasi-determinism with LVars
- The LVish library
- Joining forces: LVars and CRDTs
- Research plan

Deterministic parallel programming

Deterministic parallel programming

- *Parallel programming*: writing programs such that they can run on parallel hardware and thence faster

Deterministic parallel programming

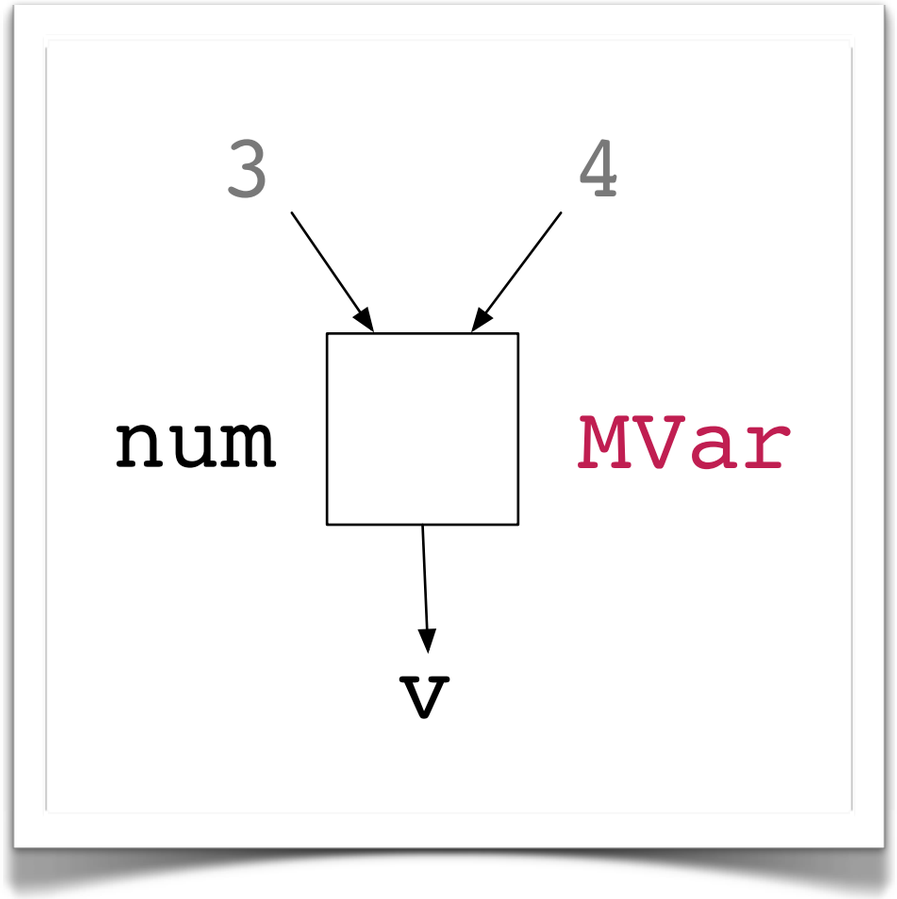
- *Parallel programming*: writing programs such that they can run on parallel hardware and thence faster
- Parallel tasks interact unpredictably, exposing *schedule nondeterminism*

Deterministic parallel programming

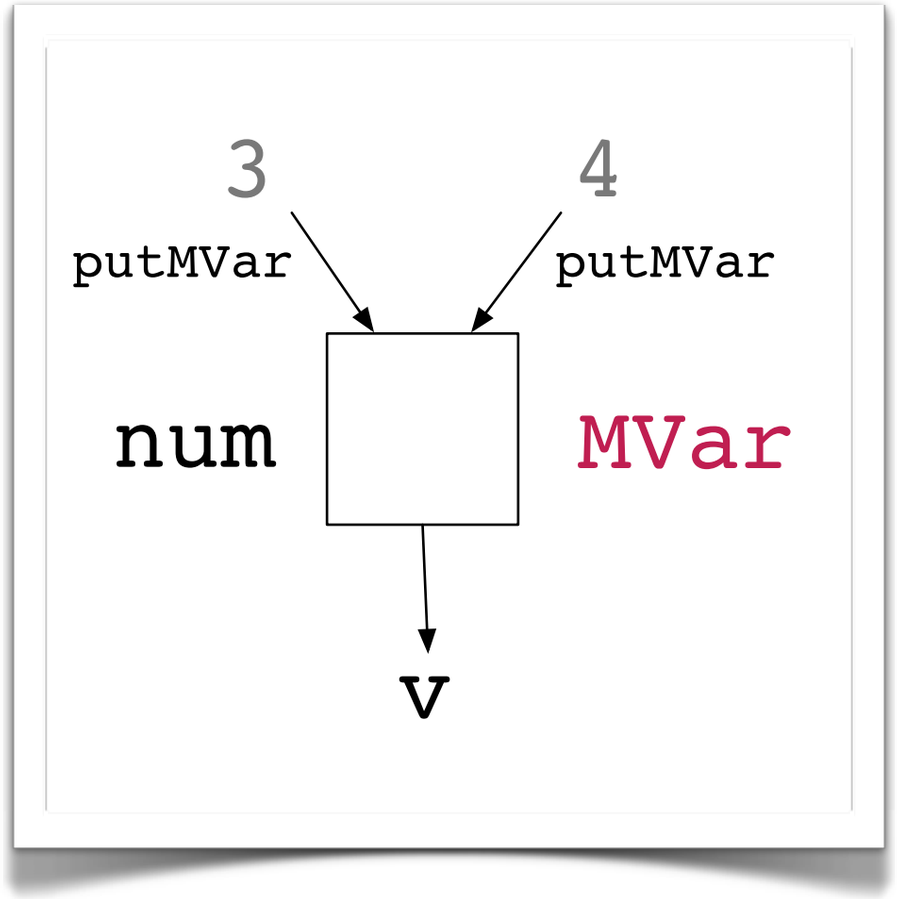
- *Parallel programming*: writing programs such that they can run on parallel hardware and thence faster
- Parallel tasks interact unpredictably, exposing *schedule nondeterminism*
- *Deterministic parallel programming* models ensure that the *observable results* of programs are the same on every run

What does this program evaluate to?

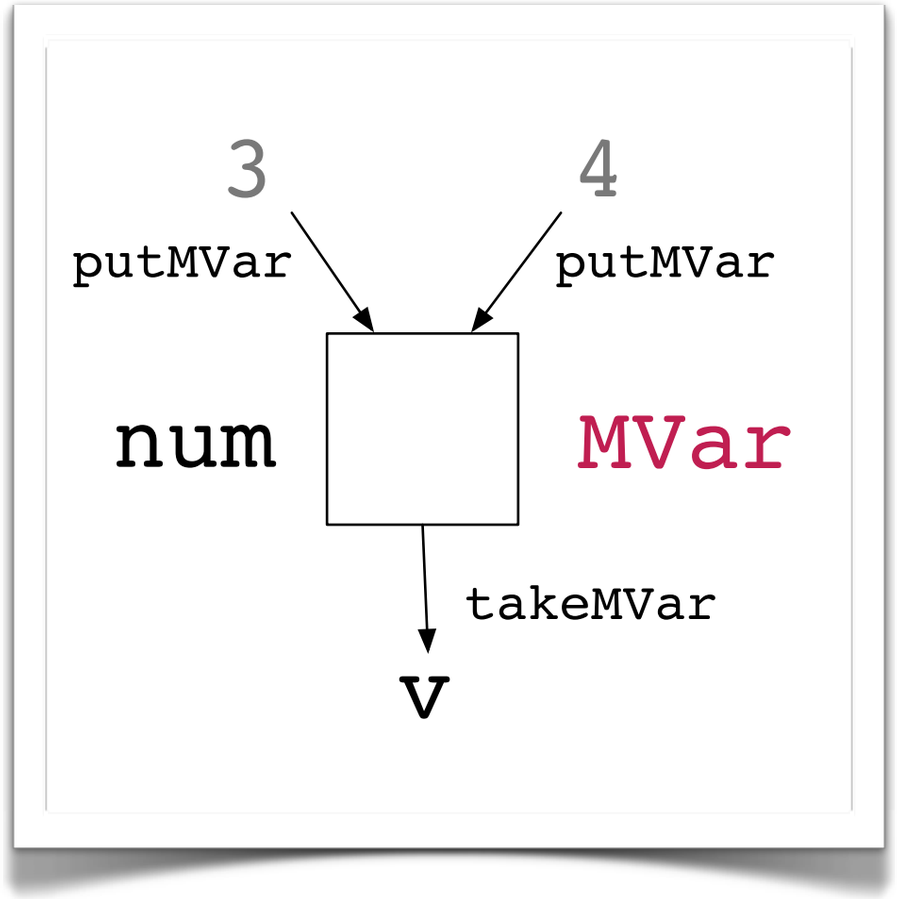
What does this program evaluate to?



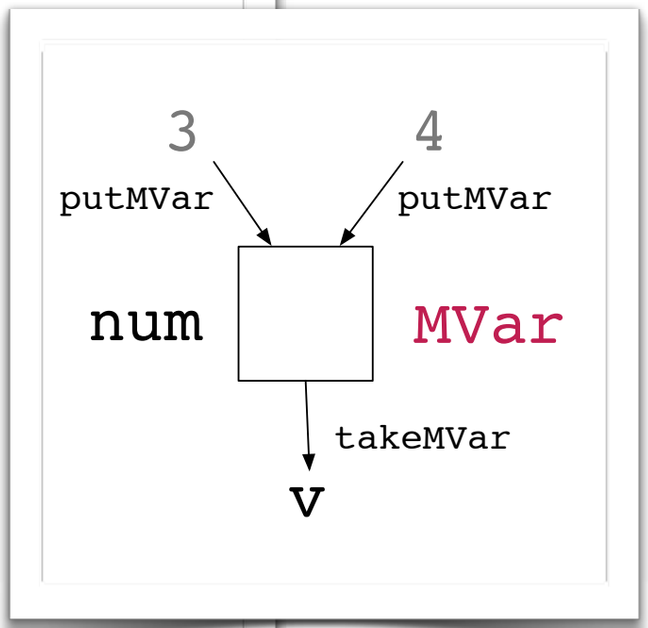
What does this program evaluate to?



What does this program evaluate to?

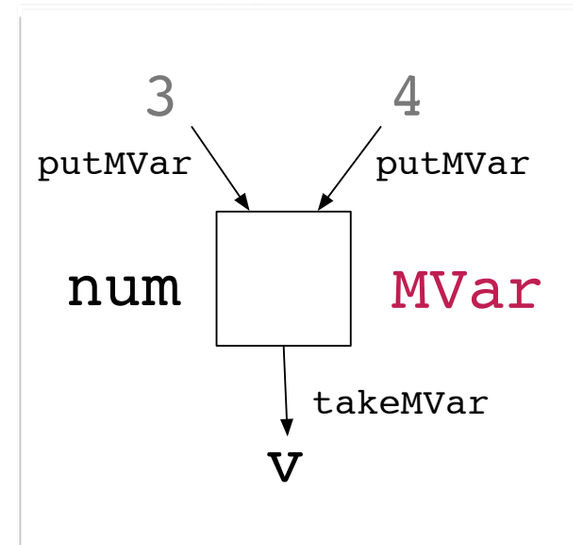


What does this program evaluate to?



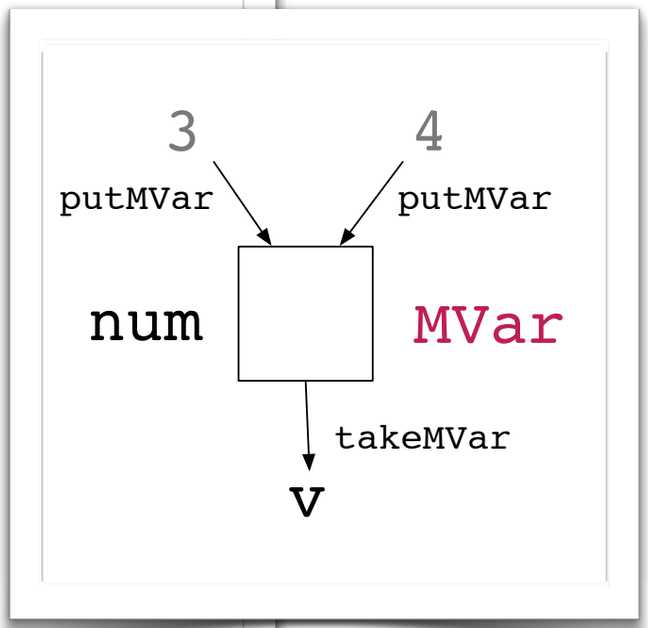
What does this program evaluate to?

```
p = do
```



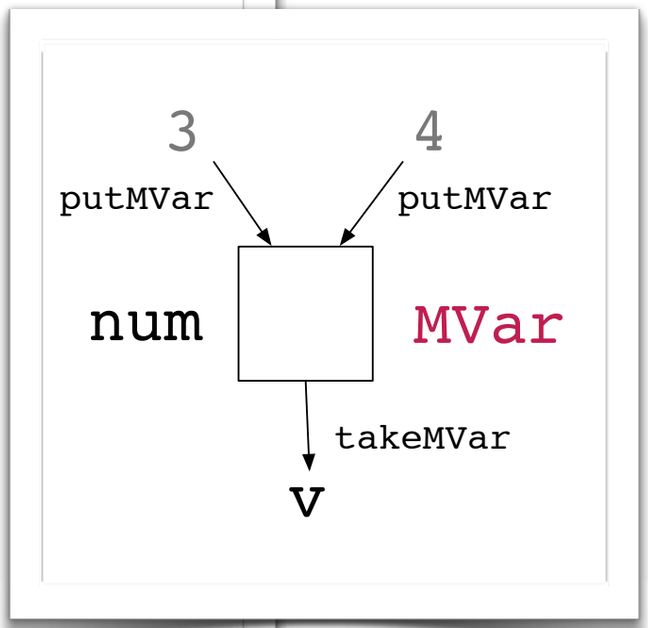
What does this program evaluate to?

```
p = do
  num <- newEmptyMVar
```



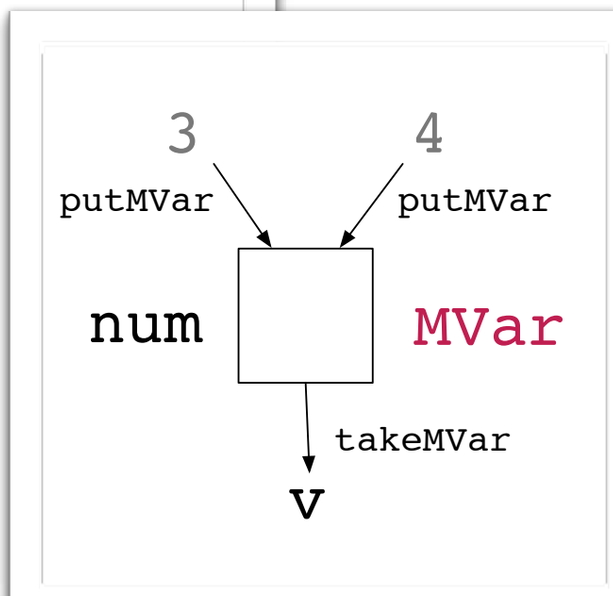
What does this program evaluate to?

```
p = do
  num <- newEmptyMVar
  forkIO (putMVar num 3)
```



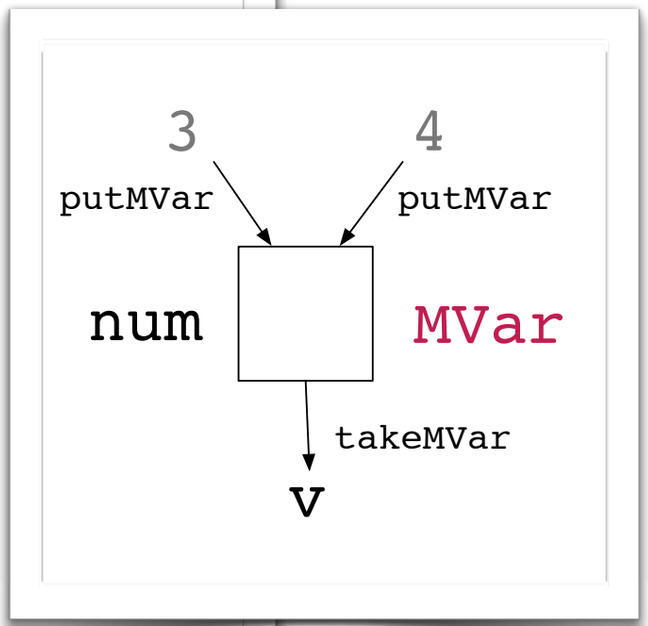
What does this program evaluate to?

```
p = do
  num <- newEmptyMVar
  forkIO (putMVar num 3)
  forkIO (putMVar num 4)
```



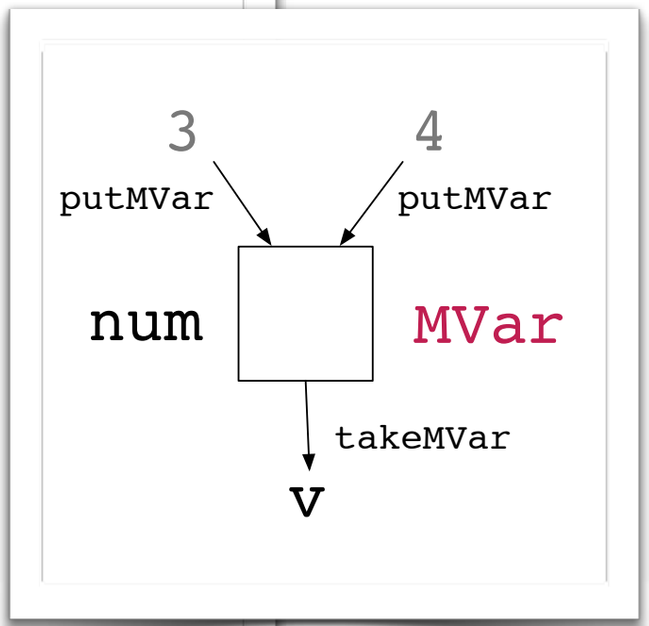
What does this program evaluate to?

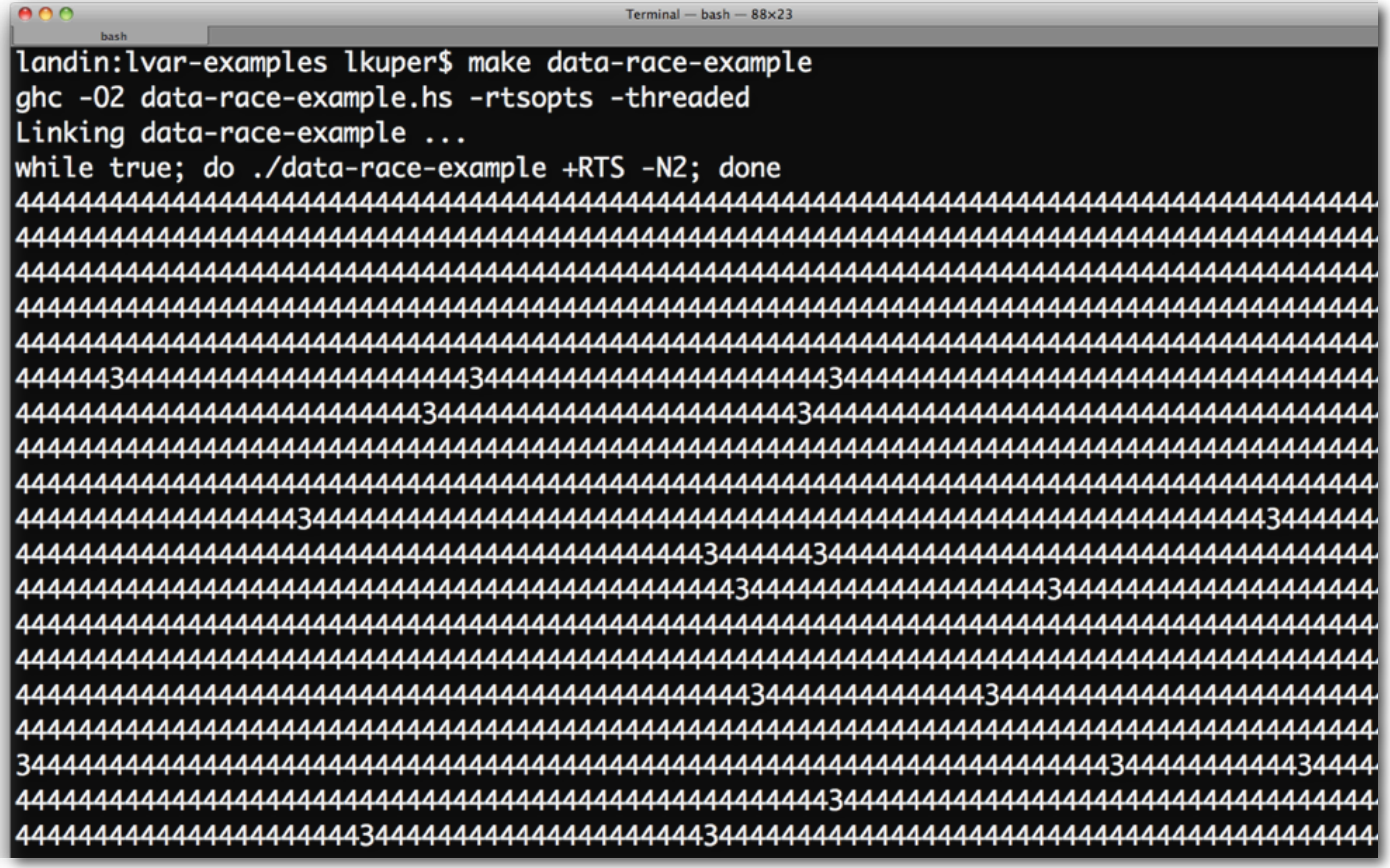
```
p = do
  num <- newEmptyMVar
  forkIO (putMVar num 3)
  forkIO (putMVar num 4)
  v <- takeMVar num
```



What does this program evaluate to?

```
p = do
  num <- newEmptyMVar
  forkIO (putMVar num 3)
  forkIO (putMVar num 4)
  v <- takeMVar num
  return v
```





```
landin:lvar-examples lkuper$ make data-race-example
ghc -O2 data-race-example.hs -rtsopts -threaded
Linking data-race-example ...
while true; do ./data-race-example +RTS -N2; done
4444444444444444444444444444444444444444444444444444444444444444444444444444444444444444
4444444444444444444444444444444444444444444444444444444444444444444444444444444444444444
4444444444444444444444444444444444444444444444444444444444444444444444444444444444444444
4444444444444444444444444444444444444444444444444444444444444444444444444444444444444444
4444444444444444444444444444444444444444444444444444444444444444444444444444444444444444
4444443444444444444444444444344444444444444444444443444444444444444444444444444444444444
4444444444444444444444444434444444444444444444444443444444444444444444444444444444444444
4444444444444444444444444444444444444444444444444444444444444444444444444444444444444444
4444444444444444444444444444444444444444444444444444444444444444444444444444444444444444
4444444444444444443444444444444444444444444444444444444444444444444444444444444344444444
4444444444444444444444444444444444444444444434444443444444444444444444444444444444444444
4444444444444444444444444444444444444444444444344444444444444444443444444444444444444444
4444444444444444444444444444444444444444444444444444444444444444444444444444444444444444
4444444444444444444444444444444444444444444444444444444444444444444444444444444444444444
4444444444444444444444444444444444444444444444434444444444444434444444444444444444444444
4444444444444444444444444444444444444444444444444444444444444444444444444444444444444444
3444444444444444444444444444444444444444444444444444444444444444444444344444444444344444
4444444444444444444444444444444444444444444444444444344444444444444444444444444444444444
4444444444444444444444344444444444444444444443444444444444444444444444444444444444444444
```

Disallow multiple writes?

```
p = do
  num <- newEmptyMVar
  forkIO (putMVar num 3)
  forkIO (putMVar num 4)
  v <- takeMVar num
  return v
```

Disallow multiple writes?

```haskell
p = do
  num <- newEmptyMVar
  forkIO (putMVar num 3)
  forkIO (putMVar num 4)
  v <- takeMVar num
  return v
```

Disallow multiple writes?

```
p = do
  num <- newEmptyMVar
  forkIO (putMVar num 3)
  forkIO (putMVar num 4)
  v <- takeMVar num
  return v
```

Tesler and Enea, 1968

Arvind *et al.*, 1989

IVars

Disallow multiple writes?

```
p :: Par Int
p = do
  num <- new
  fork    (put      num 3)
  fork    (put      num 4)
  v <- get        num
  return v
```

Tesler and Enea, 1968

Arvind *et al.*, 1989

IVars

Disallow multiple writes?

```
p :: Par Int
p = do
  num <- new
  fork    (put      num 3)
  fork    (put      num 4)
  v <- get          num
  return v
```

Tesler and Enea, 1968

Arvind *et al.*, 1989

IVars

```
./ivar-example +RTS -N2
ivar-example: multiple put
```

Deterministic programs that single-assignment forbids

```
p :: Par Int
p = do
  num <- new
  fork   (put      num 3)
  fork   (put      num 4)
  v <- get         num
  return v
```

Deterministic programs that single-assignment forbids

```haskell
p :: Par Int
p = do
  num <- new
  fork    (put      num 4)
  fork    (put      num 4)
  v <- get        num
  return v
```

Deterministic programs that single-assignment forbids

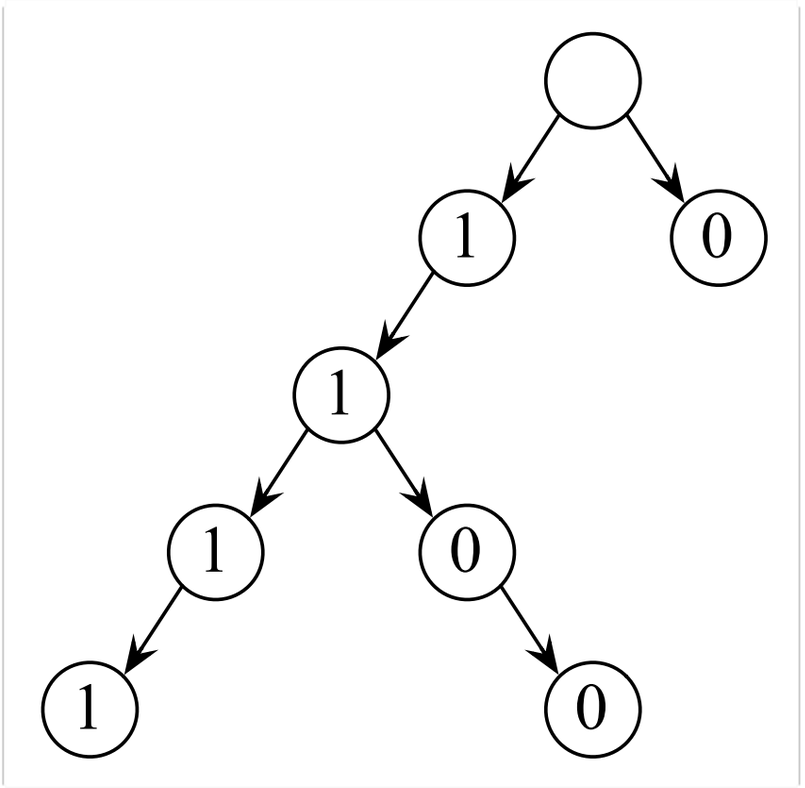
```haskell
p :: Par Int
p = do
  num <- new
  fork    (put      num 4)
  fork    (put      num 4)
  v <- get        num
  return v
```

```
./repeated-4-ivar +RTS -N2
repeated-4-ivar: multiple put
```

Deterministic programs that single-assignment forbids

```haskell
p :: Par Int
p = do
  num <- new
  fork (put num 4)
  fork (put num 4)
  v <- get num
  return v
```

```
./repeated-4-ivar +RTS -N2
repeated-4-ivar: multiple put
```

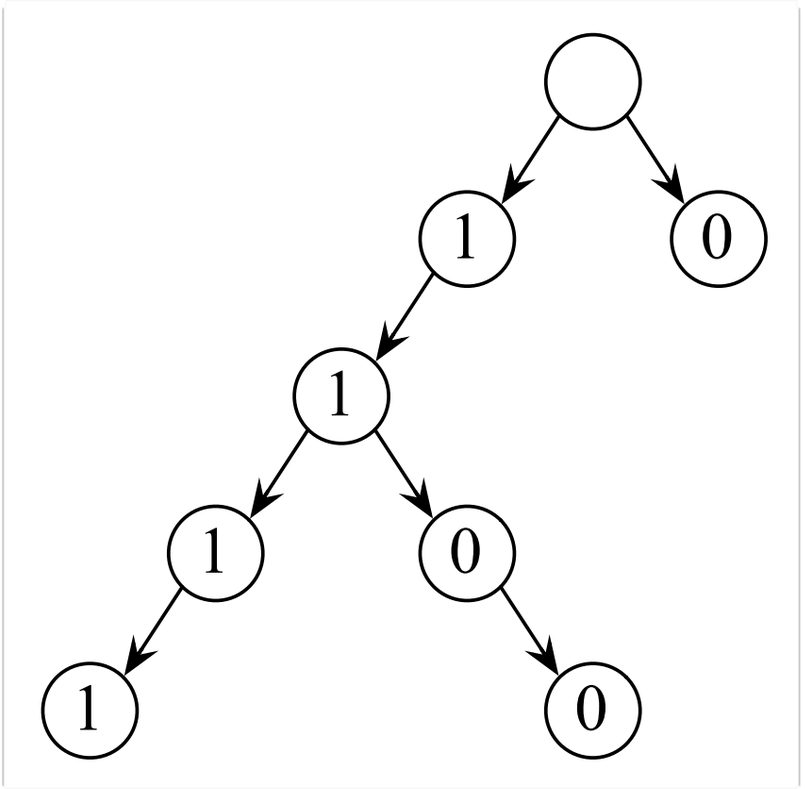


Deterministic programs that single-assignment forbids

```haskell
p :: Par Int
p = do
  num <- new
  fork (put num 4)
  fork (put num 4)
  v <- get num
  return v
```

```
./repeated-4-ivar +RTS -N2
repeated-4-ivar: multiple put
```

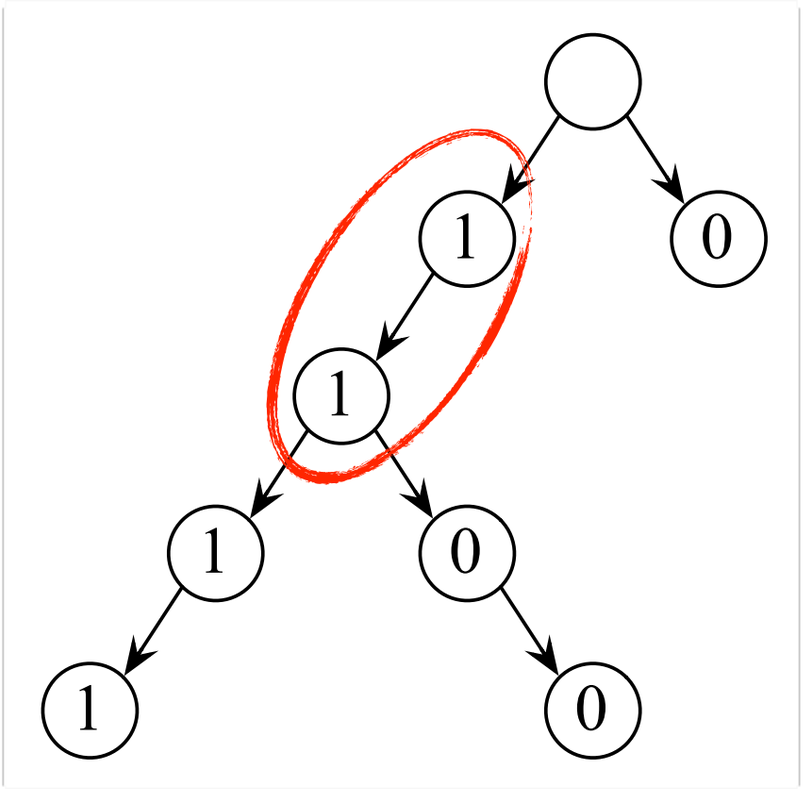
```haskell
do
  fork (insert t "0")
  fork (insert t "1100")
  fork (insert t "1111")
  v <- get t
  return v
```



Deterministic programs that single-assignment forbids

```haskell
p :: Par Int
p = do
  num <- new
  fork (put num 4)
  fork (put num 4)
  v <- get num
  return v
```

```
./repeated-4-ivar +RTS -N2
repeated-4-ivar: multiple put
```

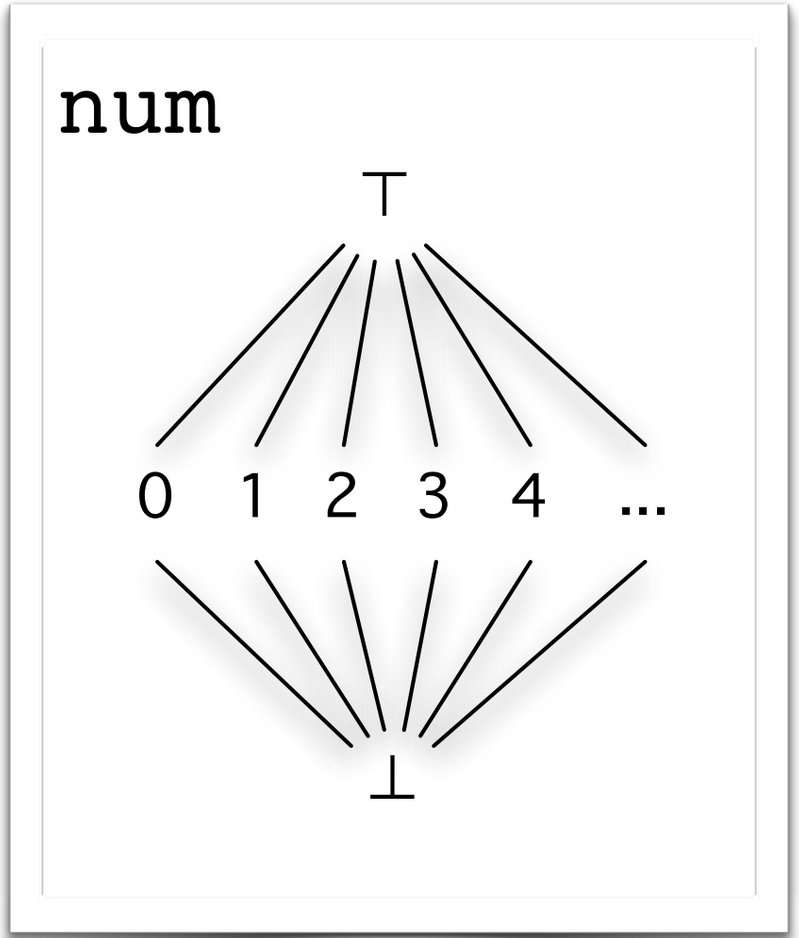



```haskell
do
  fork (insert t "0")
  fork (insert t "1100")
  fork (insert t "1111")
  v <- get t
  return v
```

Outline

- The problem and existing approaches
- Our approach: LVars
- Quasi-determinism with LVars
- The LVish library
- Joining forces: LVars and CRDTs
- Research plan

LVars: Multiple *monotonic* writes



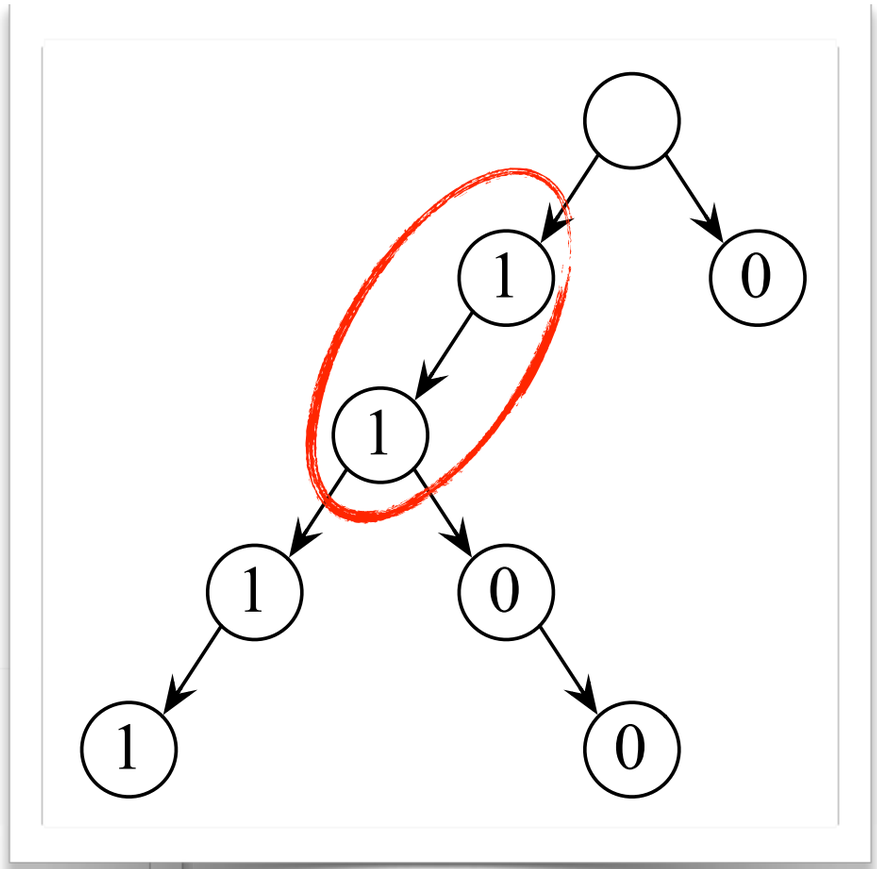
Raises an error, since $3 \sqcup 4 = \top$

```
do
   fork (put num 3)
   fork (put num 4)
```

Works fine, since $4 \sqcup 4 = 4$

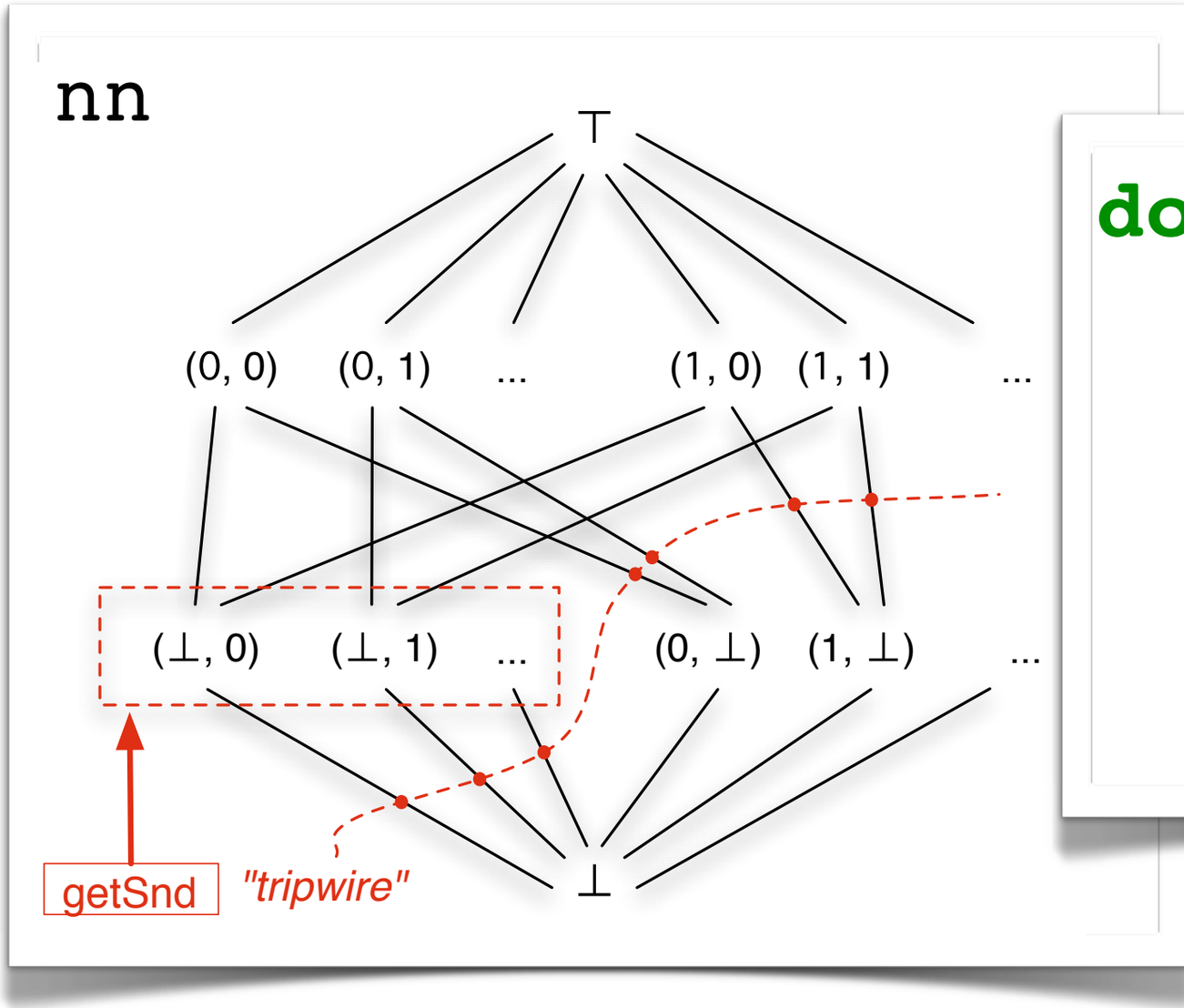
```
do
   fork (put num 4)
   fork (put num 4)
```

Overlapping writes are no problem



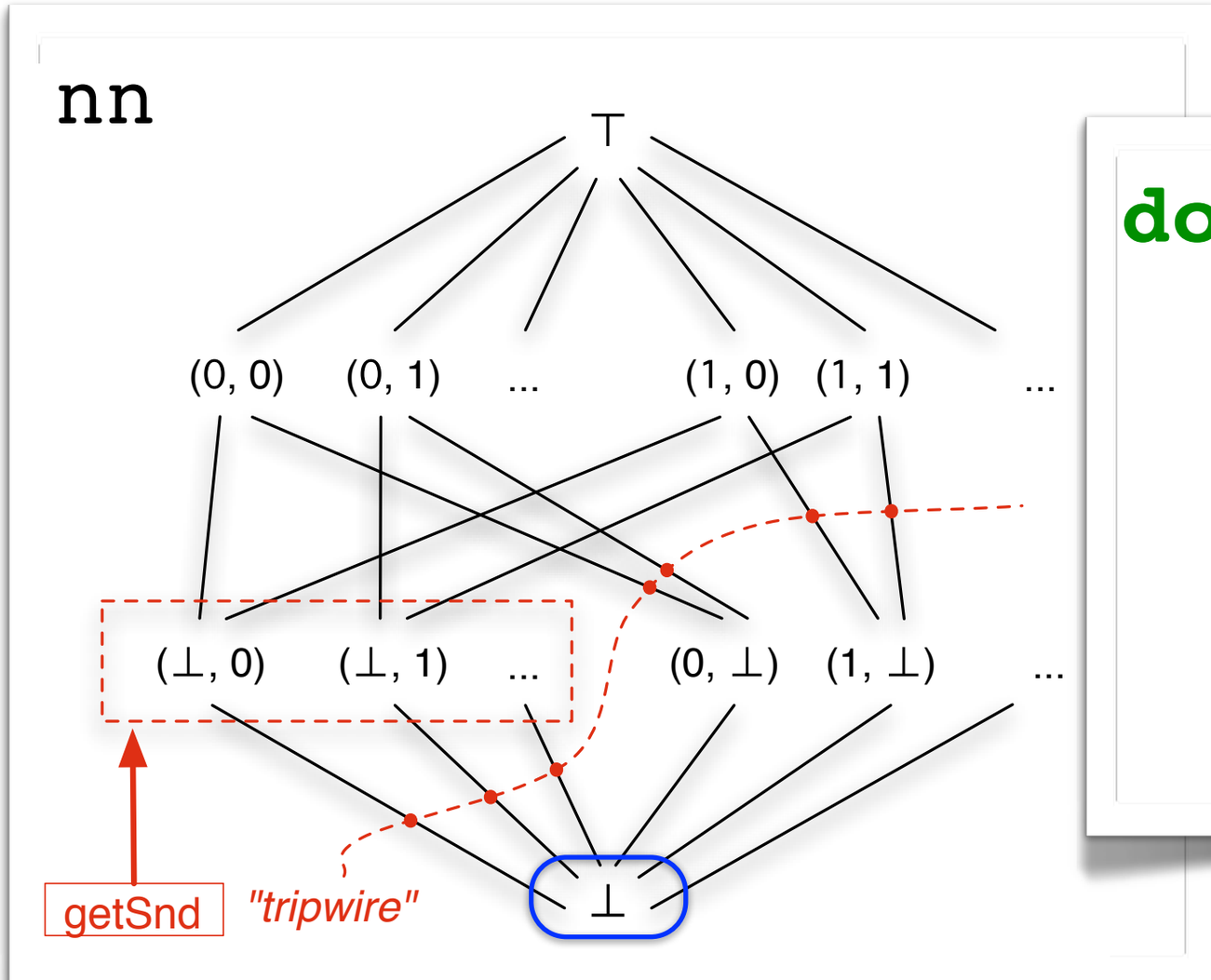

```
do
  fork (insert t "0")
  fork (insert t "1100")
  fork (insert t "1111")
  v <- get t
  return v
```

LVars: Threshold reads



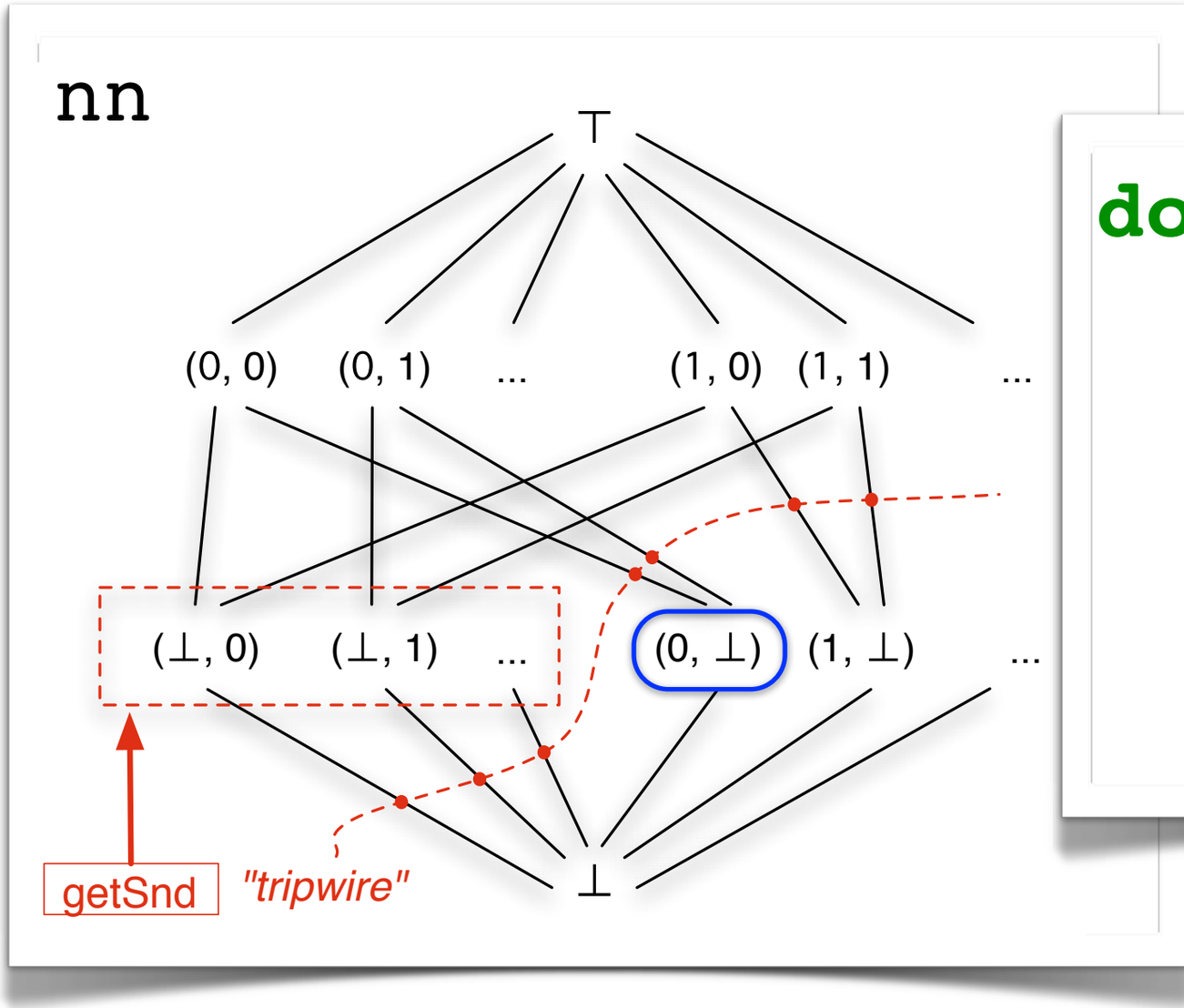

```
do
  nn <- newPair
  fork (putFst nn 0)
  fork (putSnd nn 1)
  v <- getSnd nn
  return v -- returns 1
```

LVars: Threshold reads



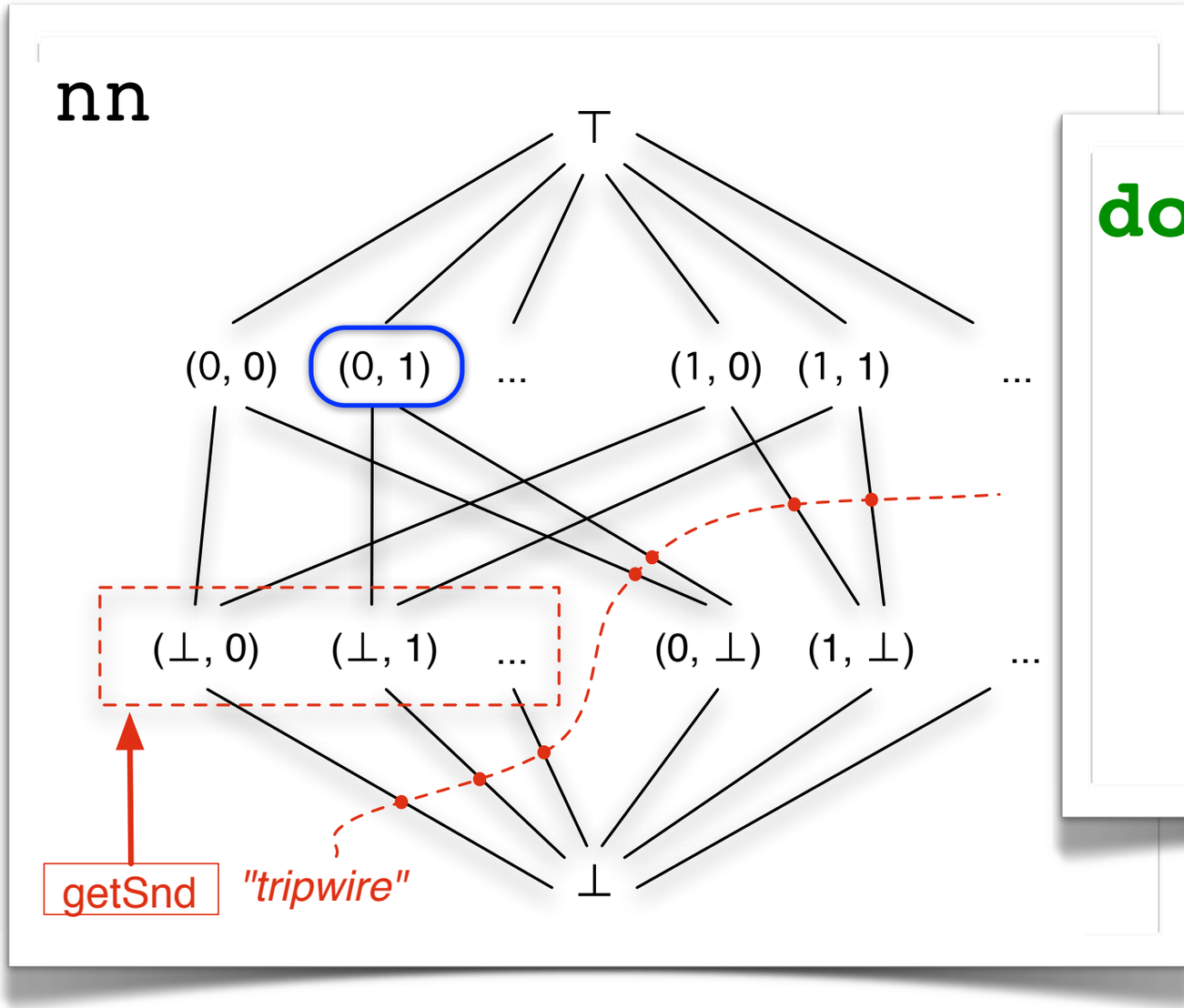

```
do
  nn <- newPair
  fork (putFst nn 0)
  fork (putSnd nn 1)
  v <- getSnd nn
  return v -- returns 1
```

LVars: Threshold reads



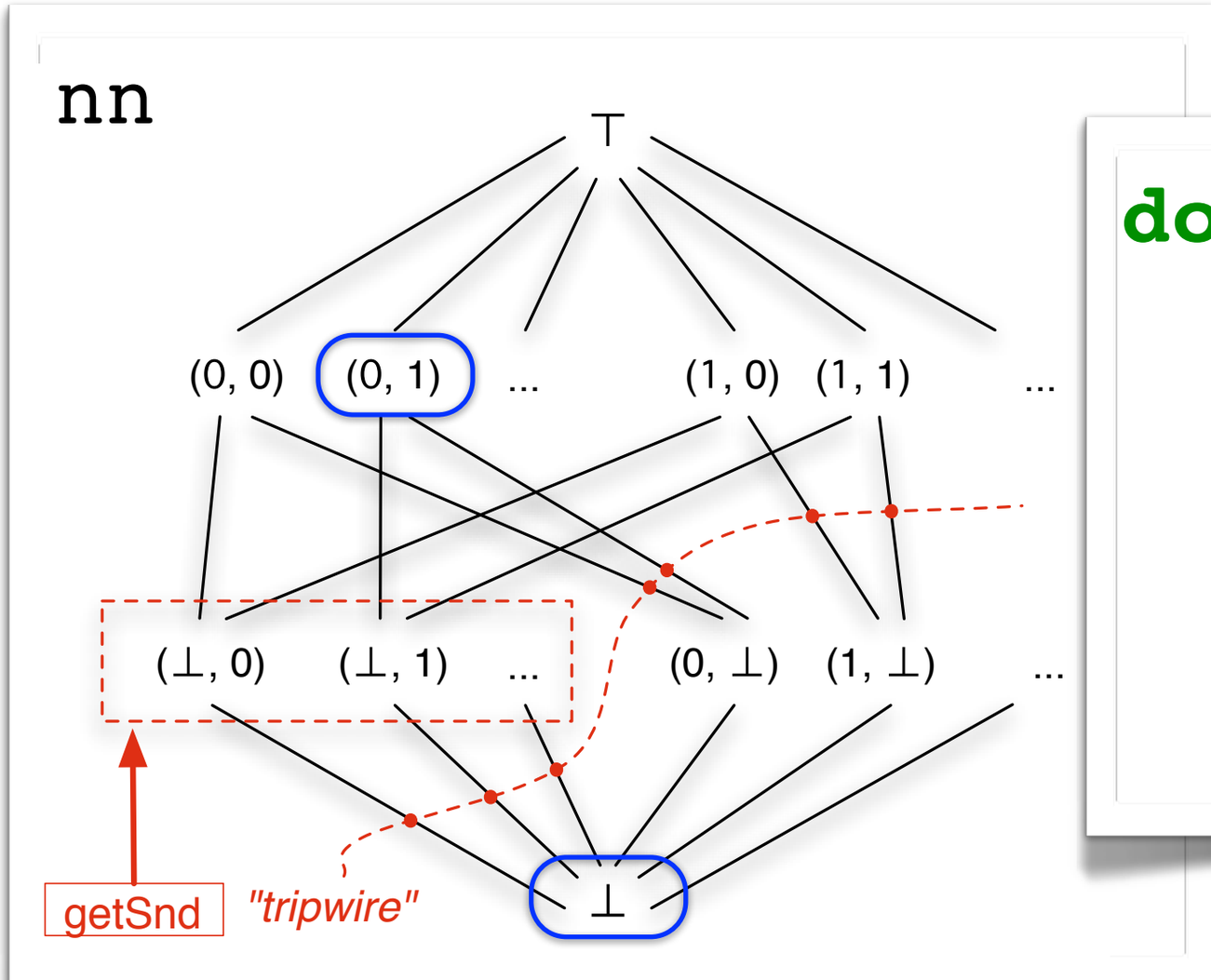

```
do
  nn <- newPair
  fork (putFst nn 0)
  fork (putSnd nn 1)
  v <- getSnd nn
  return v -- returns 1
```

LVars: Threshold reads



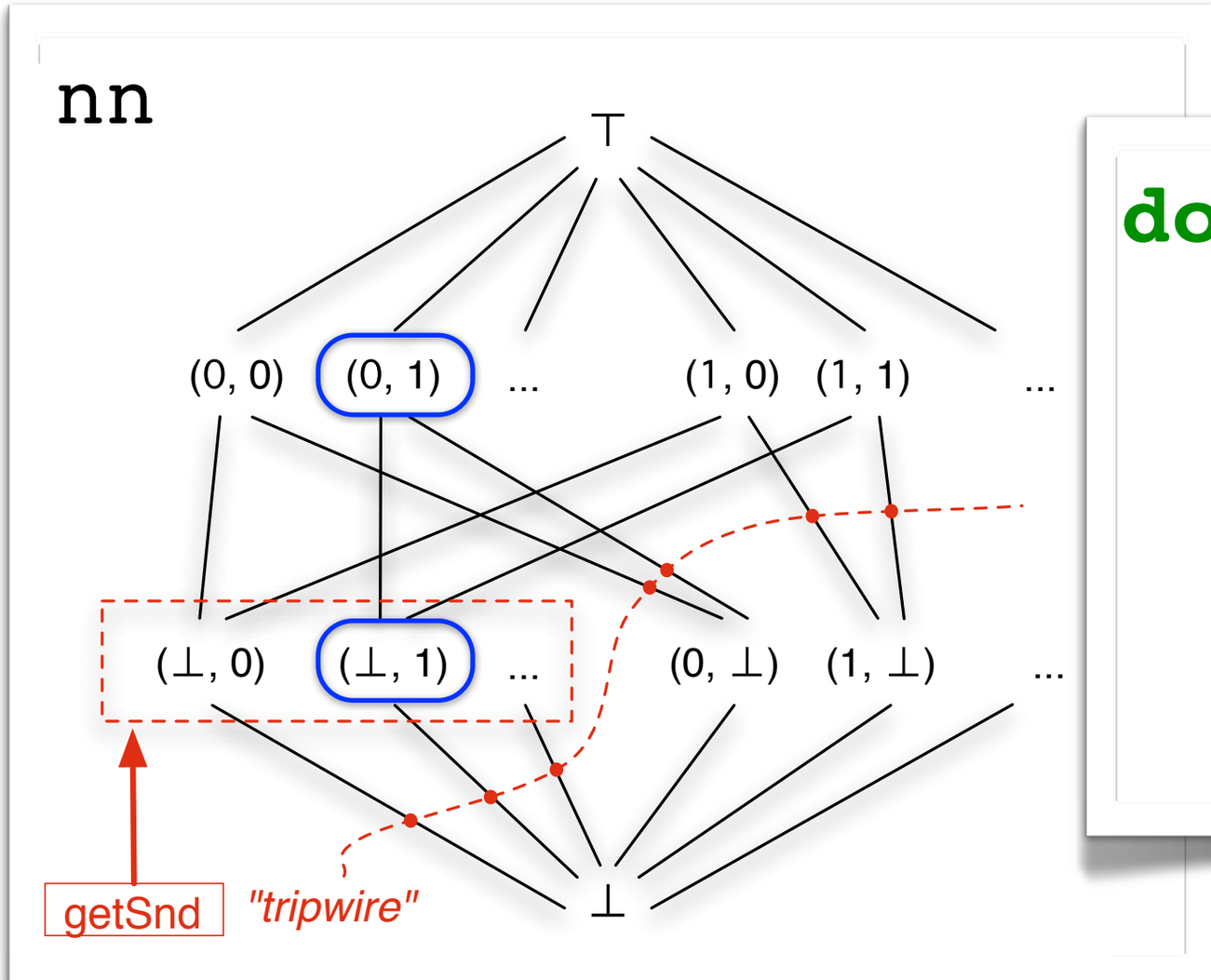

```
do
  nn <- newPair
  fork (putFst nn 0)
  fork (putSnd nn 1)
  v <- getSnd nn
  return v -- returns 1
```

LVars: Threshold reads



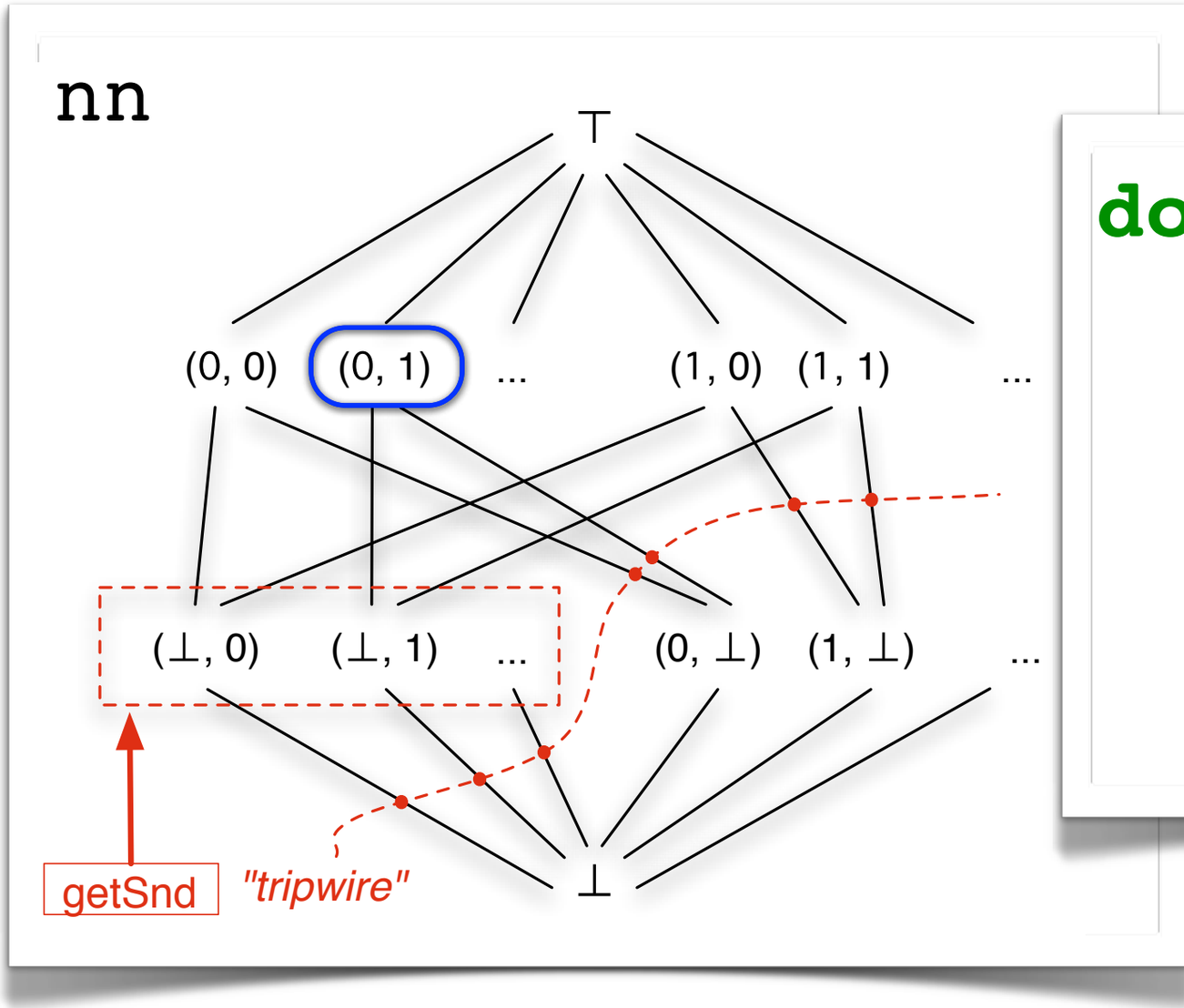

```
do
  nn <- newPair
  fork (putFst nn 0)
  fork (putSnd nn 1)
  v <- getSnd nn
  return v -- returns 1
```

LVars: Threshold reads



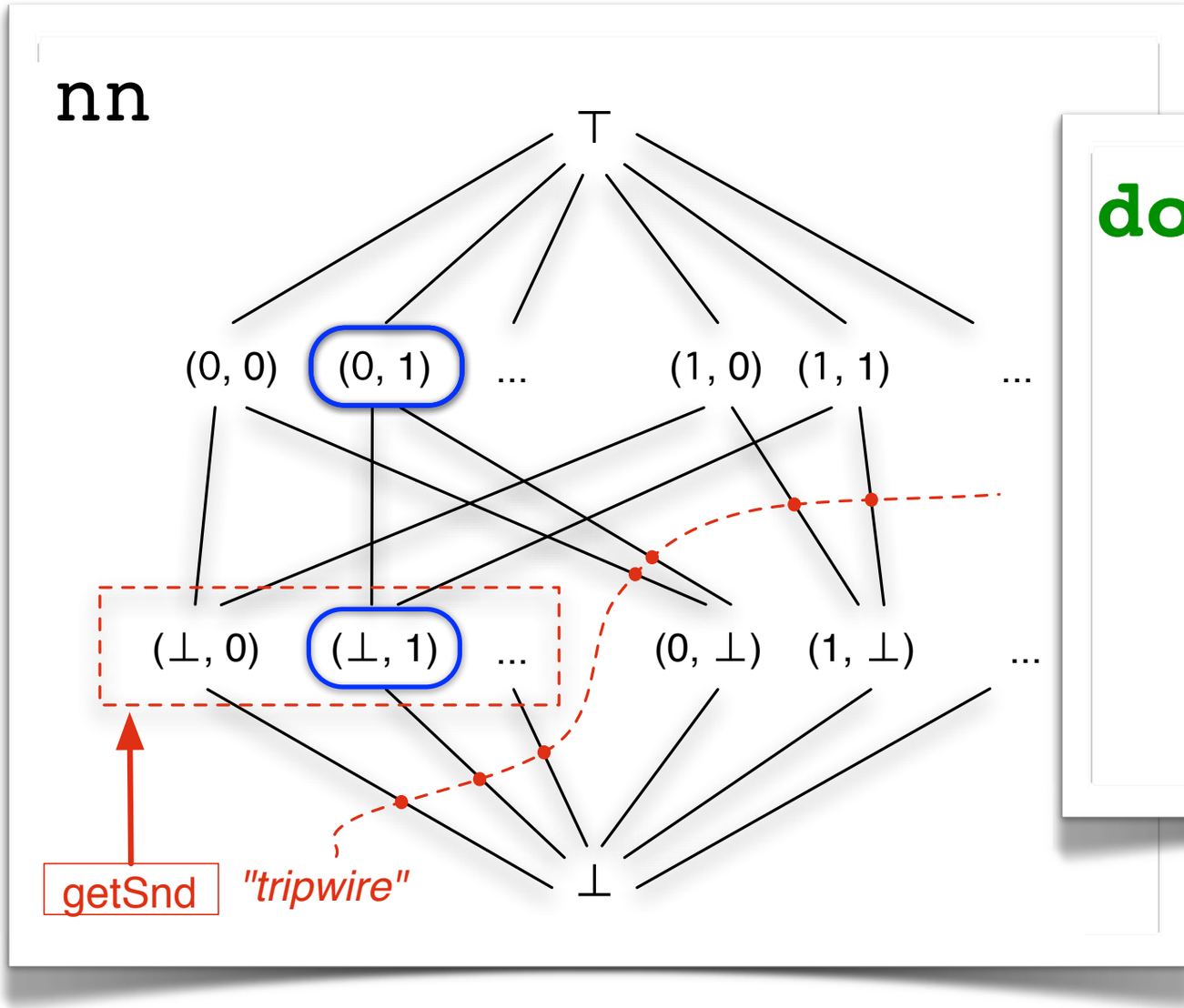

```
do
  nn <- newPair
  fork (putFst nn 0)
  fork (putSnd nn 1)
  v <- getSnd nn
  return v -- returns 1
```

LVars: Threshold reads



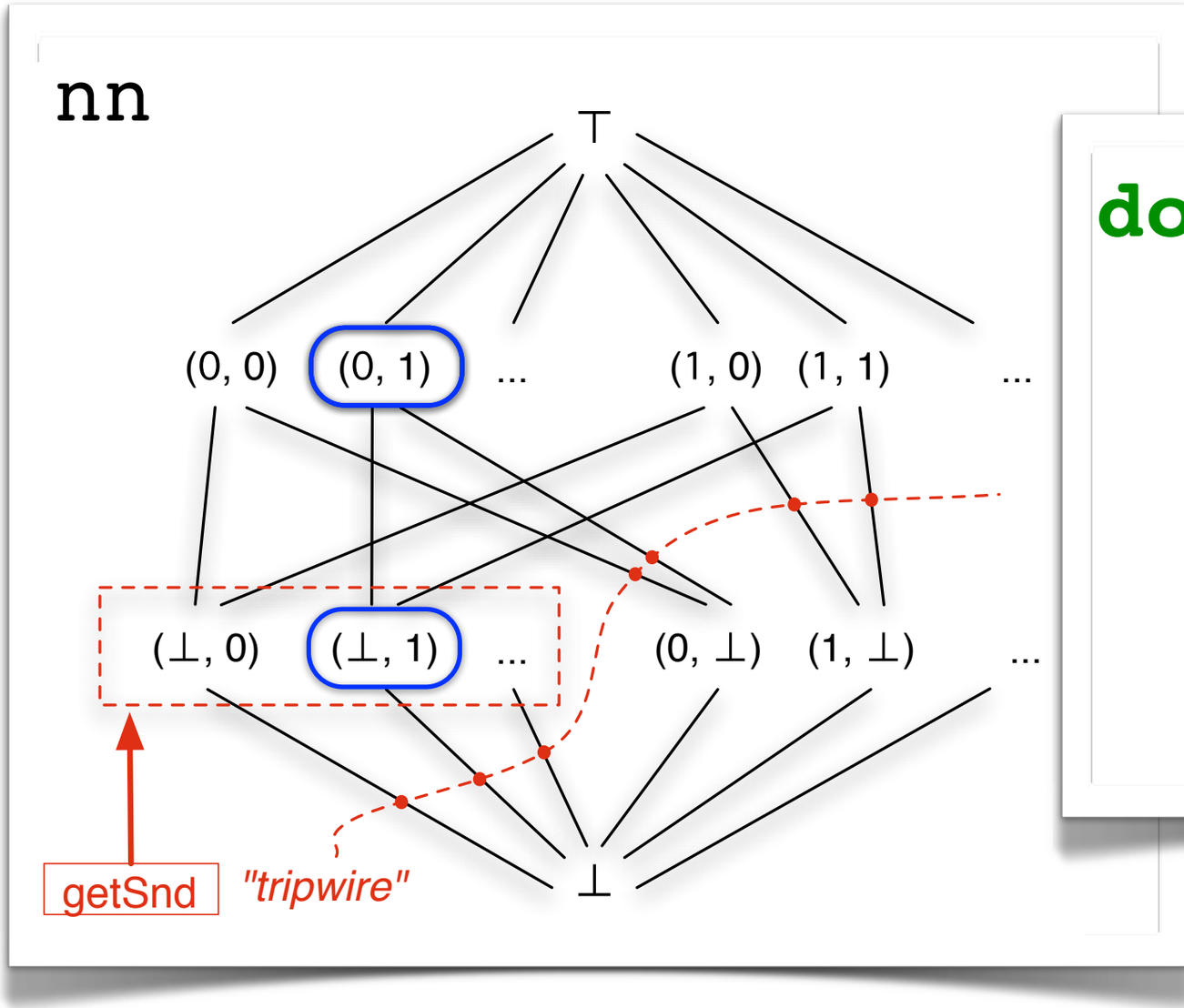

```
do
  nn <- newPair
  fork (putFst nn 0)
  fork (putSnd nn 1)
  v <- getSnd nn
  return v -- returns 1
```

LVars: Threshold reads




```
do
  nn <- newPair
  fork (putFst nn 0)
  fork (putSnd nn 1)
  v <- getSnd nn
  return v -- returns 1
```

LVars: Threshold reads




```
do
  nn <- newPair
  fork (putFst nn 0)
  fork (putSnd nn 1)
  v <- getSnd nn
  return v -- returns 1
```

The threshold set must be *pairwise incompatible*

Outline

- The problem and existing approaches
- Our approach: LVars
- Quasi-determinism with LVars
- The LVish library
- Joining forces: LVars and CRDTs
- Research plan

Problem: threshold reads can’t say “no”

Problem: threshold reads can't say "no"

- Since threshold reads are blocking, the answer to "has a given write occurred?" is always "yes"

Problem: threshold reads can't say "no"

- Since threshold reads are blocking, the answer to "has a given write occurred?" is always "yes"
- Example: find a connected graph component
  - Set of seen nodes grows monotonically…

Problem: threshold reads can't say "no"

- Since threshold reads are blocking, the answer to "has a given write occurred?" is always "yes"
- Example: find a connected graph component
  - Set of seen nodes grows monotonically…
- Algorithm relies on being able to find out *negative information* about a monotonically growing data structure

Problem: threshold reads can't say "no"

- Since threshold reads are blocking, the answer to "has a given write occurred?" is always "yes"
- Example: find a connected graph component
  - Set of seen nodes grows monotonically…
- Algorithm relies on being able to find out *negative information* about a monotonically growing data structure
- We cannot express this with threshold reads, *even though the result is deterministic*

Solution: LVar operations beyond `put` and `get`

Solution: LVar operations beyond `put` and `get`

- `freeze`: exact non-blocking read
  - introduces *quasi-determinism*: programs either produce the same result or raise an exception

Solution: LVar operations beyond `put` and `get`

- `freeze`: exact non-blocking read
  - introduces *quasi-determinism*: programs either produce the same result or raise an exception
- *Event handler*: function registered with an LVar, called whenever the LVar is updated

Solution: LVar operations beyond `put` and `get`

- `freeze`: exact non-blocking read
  - introduces *quasi-determinism*: programs either produce the same result or raise an exception
- *Event handler*: function registered with an LVar, called whenever the LVar is updated

```
traverse :: Graph -> Node -> Par (Set Node)
traverse g startNode = do
  seen <- newEmptySet
  putInSet seen startNode
  let f node = parMapM (putInSet seen) (nbrs g node)
  freezeSetAfter seen f
```

Outline

- The problem and existing approaches
- Our approach: LVars
- Quasi-determinism with LVars
- The LVish library
- Joining forces: LVars and CRDTs
- Research plan

The LVish library

The LVish library

- `Par` encapsulates LVar computations

The LVish library

- `Par` encapsulates LVar computations
- `Par` computations indexed by *effect level* for fine-grained effect tracking
  - Deterministic computations can't use `freeze`
  - Read-only computations are *cancelable*

The LVish library

- `Par` encapsulates LVar computations
- `Par` computations indexed by *effect level* for fine-grained effect tracking
  - Deterministic computations can't use `freeze`
  - Read-only computations are *cancelable*
- `runParThenFreeze` captures the deterministic "freeze-last" idiom

The LVish library

- `Par` encapsulates LVar computations
- `Par` computations indexed by *effect level* for fine-grained effect tracking
  - Deterministic computations can't use `freeze`
  - Read-only computations are *cancelable*
- `runParThenFreeze` captures the deterministic "freeze-last" idiom
- Data structures: `Data.LVar.Set`, *etc.*

The LVish library

- `Par` encapsulates LVar computations
- `Par` computations indexed by *effect level* for fine-grained effect tracking
  - Deterministic computations can't use `freeze`
  - Read-only computations are *cancelable*
- `runParThenFreeze` captures the deterministic "freeze-last" idiom
- Data structures: `Data.LVar.Set`, *etc.*
- Case studies: graph traversal, *k*-CFA, PhyBin
  - Non-idempotent `bump` operations

Outline

- The problem and existing approaches
- Our approach: LVars
- Quasi-determinism with LVars
- The LVish library
- Joining forces: LVars and CRDTs
- Research plan

Replication and eventual consistency

Replication and eventual consistency

- Replication is important and ubiquitous
  - Trade-off among **c**onsistency, **a**vailability, and **p**artition tolerance

Replication and eventual consistency

- Replication is important and ubiquitous
  - Trade-off among **c**onsistency, **a**vailability, and **p**artition tolerance
- *Eventually consistent* systems maximize availability

Replication and eventual consistency

- Replication is important and ubiquitous
  - Trade-off among **c**onsistency, **a**vailability, and **p**artition tolerance
- *Eventually consistent* systems maximize availability
- For conflict resolution, “last write wins” doesn't always make sense

Replication and eventual consistency

- Replication is important and ubiquitous
  - Trade-off among **c**onsistency, **a**vailability, and **p**artition tolerance
- *Eventually consistent* systems maximize availability
- For conflict resolution, “last write wins” doesn't always make sense
- *Strongly eventually consistent* (SEC) objects: correct replicas to which the same updates have been delivered agree

CvRDTs sound familiar

CvRDTs sound familiar

- *Conflict-free replicated data types* (CRDTs) satisfy sufficient conditions for SEC

CvRDTs sound familiar

- *Conflict-free replicated data types* (CRDTs) satisfy sufficient conditions for SEC
- Two styles of CRDT specifications
  - *State-based* or *convergent* (CvRDT)
  - *Op-based* or *commutative* (CmRDT)

CvRDTs sound familiar

- *Conflict-free replicated data types* (CRDTs) satisfy sufficient conditions for SEC
- Two styles of CRDT specifications
  - *State-based* or *convergent* (CvRDT)
  - *Op-based* or *commutative* (CmRDT)
- CvRDTs come equipped with a partial order $\leqslant$:
  - states form a join-semilattice ordered by $\leqslant$
  - merging replicas computes the lub of their states
  - state is *inflationary* across updates: $u(s) \geqslant s$

CvRDTs sound familiar

- *Conflict-free replicated data types* (CRDTs) satisfy sufficient conditions for SEC
- Two styles of CRDT specifications
  - *State-based* or *convergent* (CvRDT)
  - *Op-based* or *commutative* (CmRDT)
- CvRDTs come equipped with a partial order $\leqslant$:
  - states form a join-semilattice ordered by $\leqslant$
  - merging replicas computes the lub of their states
  - state is *inflationary* across updates: $u(s) \geqslant s$
- CvRDTs are SEC (Shapiro et al. 2011)

Integrating CvRDTs and LVars

Integrating CvRDTs and LVars

- CvRDTs aren't quite LVars
  - No notion of threshold reads
  - Notion of "update" distinct from "merge"
  - Objects are replicated, not shared

Integrating CvRDTs and LVars

- CvRDTs aren't quite LVars
  - No notion of threshold reads
  - Notion of "update" distinct from "merge"
  - Objects are replicated, not shared
- Proposal: add threshold reads to CvRDTs
  - Prove a *query consistency* property

Integrating CvRDTs and LVars

- CvRDTs aren't quite LVars
  - No notion of threshold reads
  - Notion of "update" distinct from "merge"
  - Objects are replicated, not shared
- Proposal: add threshold reads to CvRDTs
  - Prove a *query consistency* property
- Proposal: extend LVars to allow inflationary non-lub updates
  - `bump` is already an example of this!

CmRDTs and non-monotonic updates

CmRDTs and non-monotonic updates

- A CmRDT is an op-based object with the property that *all concurrent updates* commute
  - $u(u'(s)) = u'(u(s))$

CmRDTs and non-monotonic updates

- A CmRDT is an op-based object with the property that *all concurrent updates* commute
  - $u(u'(s)) = u'(u(s))$
- CmRDTs are SEC (Shapiro *et al*. 2011)

CmRDTs and non-monotonic updates

- A CmRDT is an op-based object with the property that *all concurrent updates* commute
  - $u(u'(s)) = u'(u(s))$
- CmRDTs are SEC (Shapiro *et al*. 2011)
- CmRDTs and CvRDTs are equivalent!

CmRDTs and non-monotonic updates

- A CmRDT is an op-based object with the property that *all concurrent updates* commute
  - $u(u'(s)) = u'(u(s))$
- CmRDTs are SEC (Shapiro *et al*. 2011)
- CmRDTs and CvRDTs are equivalent!
- Suggests a strategy: *simulate* non-monotonic data structures with monotonic ones
  - 2P-Sets: grow-only sets track additions and removals
  - PN-Counters: same for increments and decrements

CmRDTs and non-monotonic updates

- A CmRDT is an op-based object with the property that *all concurrent updates* commute
  - $u(u'(s)) = u'(u(s))$
- CmRDTs are SEC (Shapiro *et al*. 2011)
- CmRDTs and CvRDTs are equivalent!
- Suggests a strategy: *simulate* non-monotonic data structures with monotonic ones
  - 2P-Sets: grow-only sets track additions and removals
  - PN-Counters: same for increments and decrements
- Proposal: add 2P-Sets and PN-Counters to LVish

Outline

- The problem and existing approaches
- Our approach: LVars
- Quasi-determinism with LVars
- The LVish library
- Joining forces: LVars and CRDTs
- Research plan

What’s already done

What's already done

- Basic LVars model (`put` and `get`) [FHPC '13]
  - PLT Redex model; determinism proof

What's already done

- Basic LVars model (`put` and `get`) [FHPC '13]
  - PLT Redex model; determinism proof
- LVars with freezing and handlers [POPL '14]
  - PLT Redex model; quasi-determinism proof

What's already done

- Basic LVars model (`put` and `get`) [FHPC '13]
  - PLT Redex model; determinism proof
- LVars with freezing and handlers [POPL '14]
  - PLT Redex model; quasi-determinism proof
- LVish library implementation [POPL '14]
  - Graph traversal and *k*-CFA case studies

What's already done

- Basic LVars model (`put` and `get`) [FHPC '13]
  - PLT Redex model; determinism proof
- LVars with freezing and handlers [POPL '14]
  - PLT Redex model; quasi-determinism proof
- LVish library implementation [POPL '14]
  - Graph traversal and *k*-CFA case studies
- Effect tracking and `bump` implementation [under submission]
  - PhyBin case study

Still to do

Still to do

- Add `bump` to λ_{LVish} (~1 month)
  - Update existing determinism proofs

Still to do

- Add `bump` to λ_{LVish} (~1 month)
  - Update existing determinism proofs
- Add threshold reads to CvRDTs (~2 months)
  - Prove a query consistency property

Still to do

- Add `bump` to λ_{LVish} (~1 month)
  - Update existing determinism proofs
- Add threshold reads to CvRDTs (~2 months)
  - Prove a query consistency property
- Add PN-Counters and 2P-Sets to LVish (~3 months)
  - Implement at least one application using them

Still to do

- Add `bump` to λ_{LVish} (~1 month)
  - Update existing determinism proofs
- Add threshold reads to CvRDTs (~2 months)
  - Prove a query consistency property
- Add PN-Counters and 2P-Sets to LVish (~3 months)
  - Implement at least one application using them
- Write (~3 months)
  - Extended journal version of POPL paper with `bump`
  - New paper on integrating CRDTs and LVars

Still to do

- Add `bump` to λ_{LVish} (~1 month)
  - Update existing determinism proofs
- Add threshold reads to CvRDTs (~2 months)
  - Prove a query consistency property
- Add PN-Counters and 2P-Sets to LVish (~3 months)
  - Implement at least one application using them
- Write (~3 months)
  - Extended journal version of POPL paper with `bump`
  - New paper on integrating CRDTs and LVars
- Defend in ~September 2014

Lattice-based data structures are a general and practical foundation for deterministic and quasi-deterministic parallel and distributed programming.