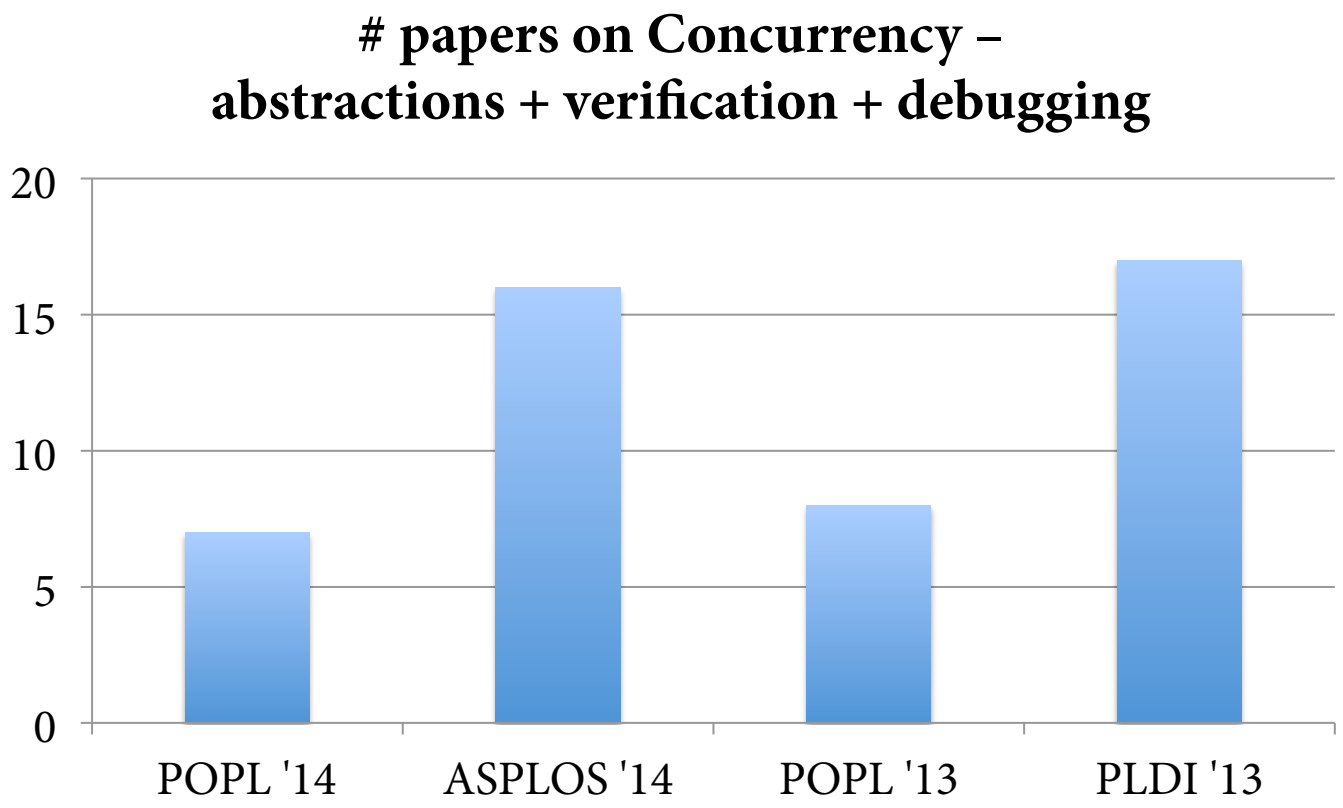
Functional Abstractions for Practical and Scalable Concurrent Programming

KC Sivaramakrishnan
Purdue University

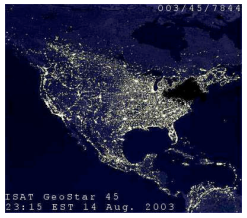
Concurrent programming is (still) hard!



Concurrency bugs can be disastrous

Concurrency bugs can be disastrous

2003 Northeast blackout[1]

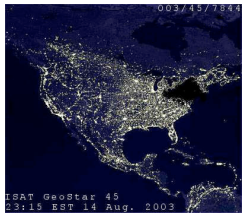


- Data race disables alarm system
- Effect
  - 256 power stations go offline
  - ~7 hr major blackout
  - 11 fatalities, $6 billion

[1] http://www.scientificamerican.com/article/2003-blackout-five-years-later/
[2] http://sunnyday.mit.edu/papers/therac.pdf
[3] http://www.cnbc.com/id/100587334

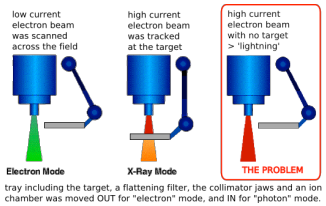
Concurrency bugs can be disastrous

2003 Northeast blackout[1]



- Data race disables alarm system
- Effect
  - 256 power stations go offline
  - ~7 hr major blackout
  - 11 fatalities, $6 billion

Therac-25 incidents[2]

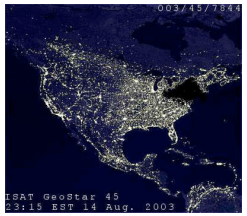


- Data race b/w UI and controller
- 6 fatalities

[1] http://www.scientificamerican.com/article/2003-blackout-five-years-later/
[2] http://sunnyday.mit.edu/papers/therac.pdf
[3] http://www.cnbc.com/id/100587334

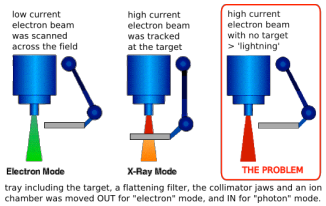
Concurrency bugs can be disastrous

2003 Northeast blackout[1]



- Data race disables alarm system
- Effect
  - 256 power stations go offline
  - ~7 hr major blackout
  - 11 fatalities, $6 billion

Therac-25 incidents[2]



- Data race b/w UI and controller
- 6 fatalities

Facebook IPO @ NASDAQ[3]

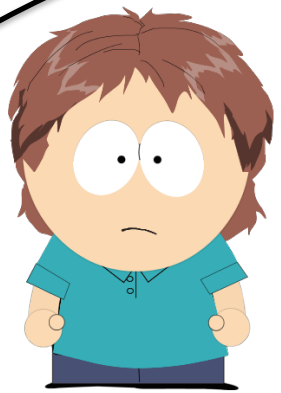
- Race between validation and new orders arrival
- NASDAQ compensation = $62 million

[1] http://www.scientificamerican.com/article/2003-blackout-five-years-later/
[2] http://sunnyday.mit.edu/papers/therac.pdf
[3] http://www.cnbc.com/id/100587334

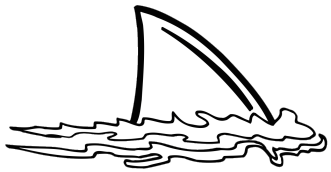
Safe and scalable
concurrent program

Safe and scalable
concurrent program

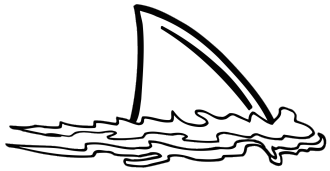
Fences, Locks,
Condition variables, etc.



Safe and scalable
concurrent program

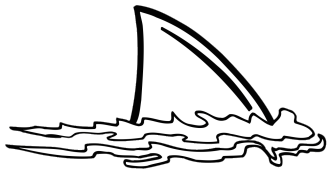


Data race

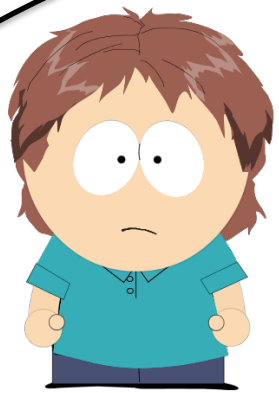


Deadlock

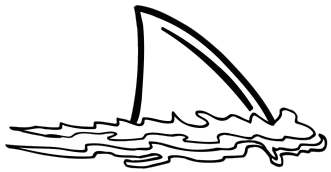
Fences, Locks,
Condition variables, etc.



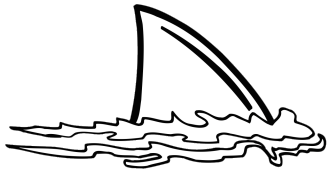
Atomicity
violations



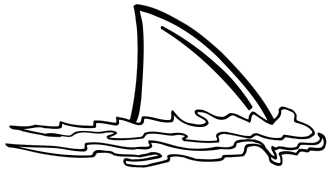
Safe and scalable
concurrent program



Data race

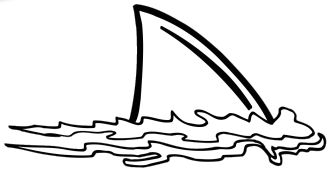


Weak memory
semantics

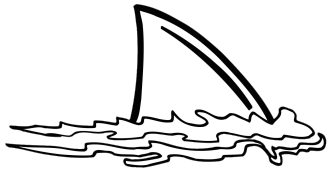


Deadlock

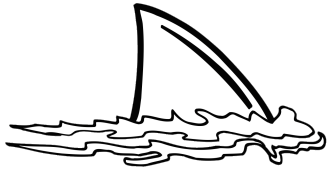
Fences, Locks,
Condition variables, etc.



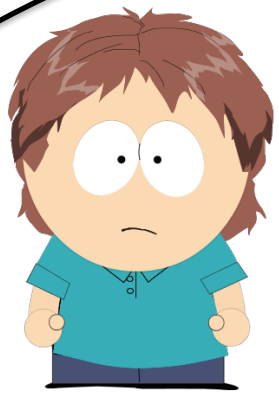
Eventual
consistency



No cache
coherence



Atomicity
violations



Stick to
Status Quo

Safe and scalable
concurrent program

Stick to Status Quo

Testing

Safe and scalable concurrent program

Stick to Status Quo

Testing

Verification

Safe and scalable concurrent program

Stick to Status Quo

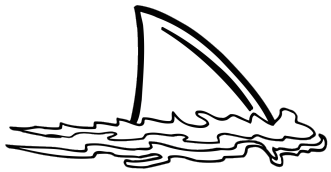
Testing

Verification

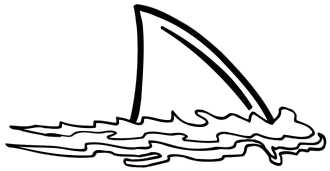
PL Support

Safe and scalable concurrent program

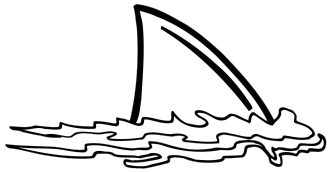
Safe and scalable concurrent program



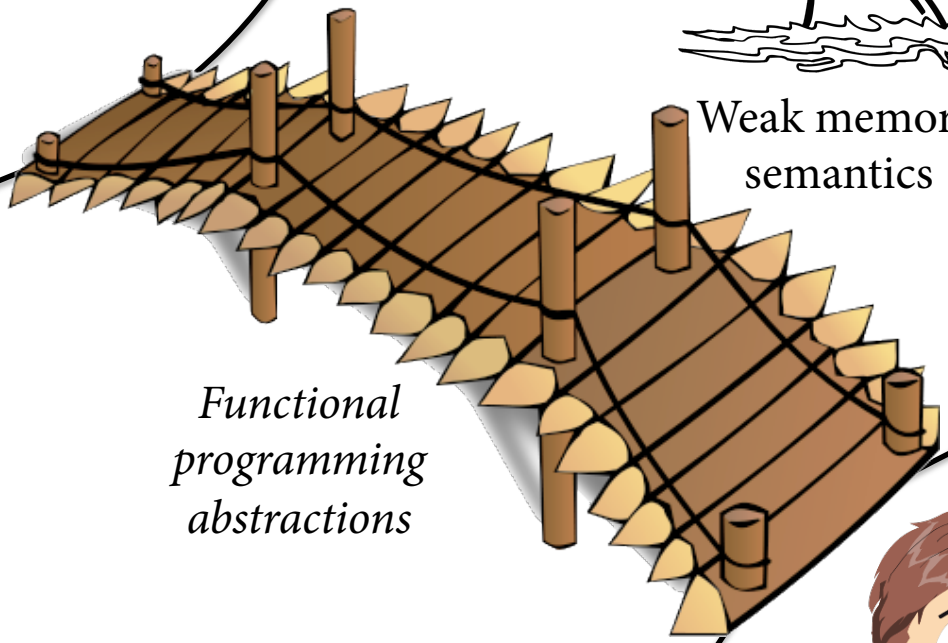
Data race



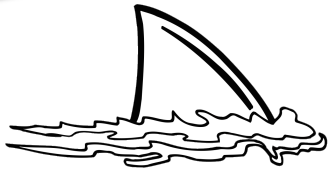
Deadlock



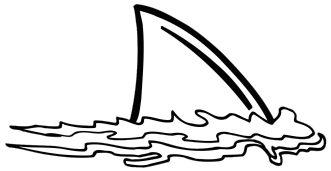
Weak memory semantics



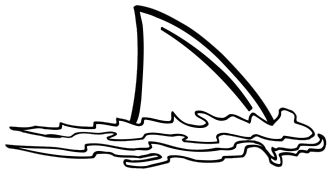
Functional programming abstractions



Eventual consistency



Non-cache coherence



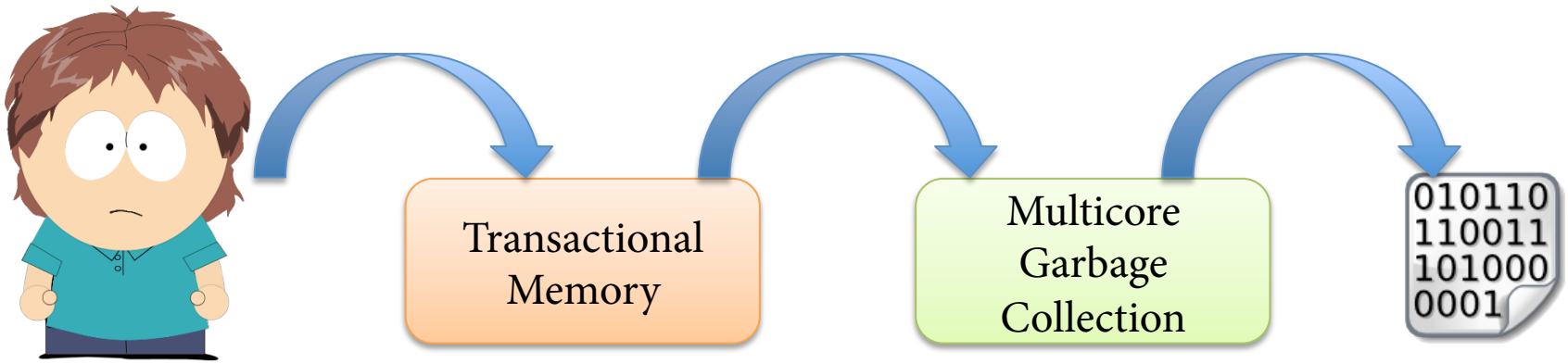
Atomicity violations

Programming abstractions *simplify* Concurrent Programming

Transactional Memory

Multicore Garbage Collection

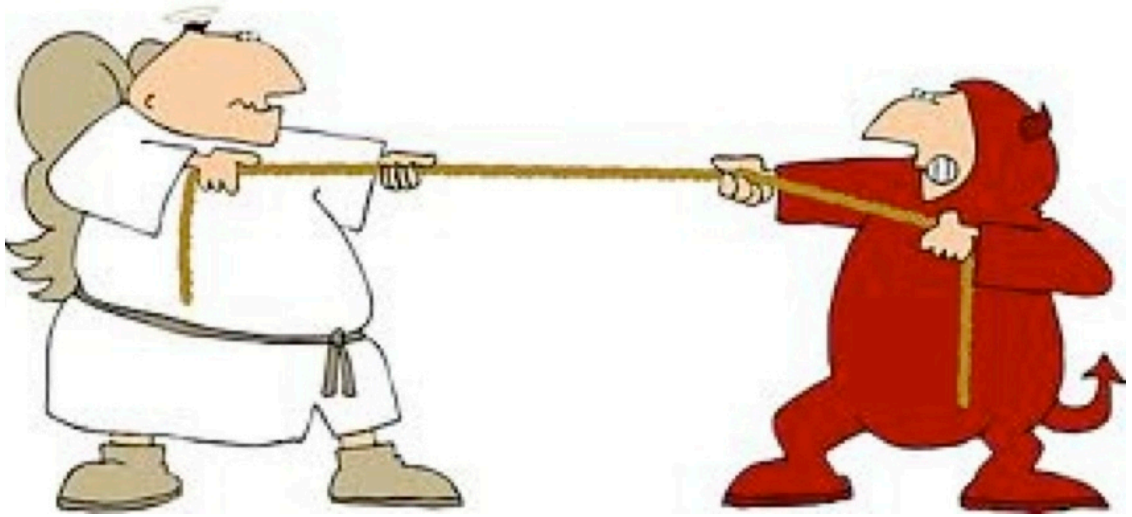
But, programming abstractions introduce a *level of indirection*



Slower than locks under high contention

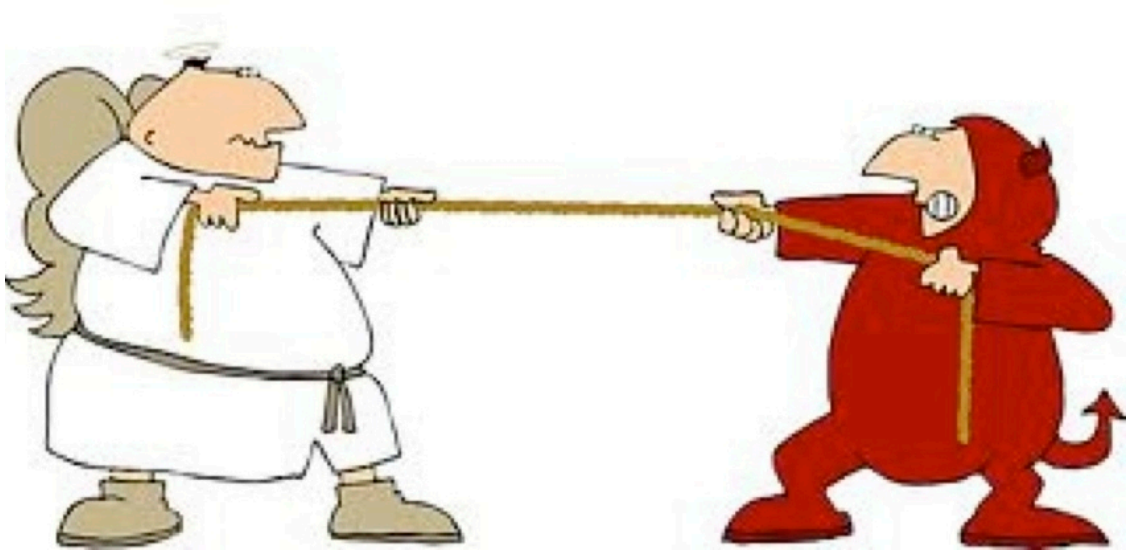
Stop-the-world GC hinders scalability

Safety
&
Simplicity



Performance
&
Functionality

Safety
&
Simplicity



Performance
&
Functionality

Always desirable to marry the two whenever possible!

MultiMLton

Intel SCC
48-core Non-cache-coherent

Azul Vega 3
864-core CC-UMA

Compute clouds

Windows Azure

amazon web services™

MultiMLton

Intel SCC
48-core Non-cache-coherent

Azul Vega 3
864-core CC-UMA

Compute clouds

Windows Azure

amazon web services™

Language Design

Asynchronous CML [PLDI '11]
Memoizing Communication [ICFP '09]
Rx-CML – optimistic CML [PADL '14]

Runtime Systems

Parasitic threads [DAMP '10, JFP '14]
Thread-local GC [ISMM '12, MARC '12, JFP '14]

MultiMLton

Intel SCC
48-core Non-cache-coherent

Azul Vega 3
864-core CC-UMA

Compute clouds

Windows Azure

amazon web services™

Language Design

Asynchronous CML [PLDI '11]
Memoizing Communication [ICFP '09]
Rx-CML – optimistic CML [PADL '14]

Runtime Systems

Parasitic threads [DAMP '10, JFP '14]
Thread-local GC [ISMM '12, MARC '12, JFP '14]

Scheduler activations for Haskell

Schedulers for Haskell threads as Haskell libraries
[In submission to OOPSLA '14]

Sting (Java)

Session type based protocol optimization
[Coordination '10, SCP '13]

MultiMLton

Language Design

Rx-CML – optimistic CML [PADL '14]

Scheduler activations for Haskell

Schedulers for Haskell threads as Haskell libraries
[In submission to OOPSLA '14]

Rx-CML : A Prescription for Safely Relaxing Synchrony

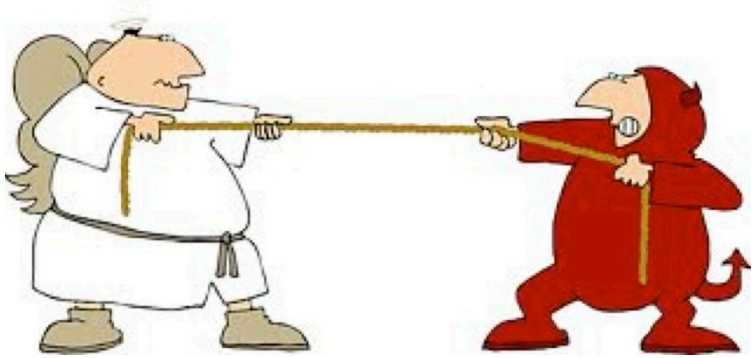
Abstraction	Synchronous communication	Transactional memory	Garbage collection
			
Domain	Asynchronous distributed system	Shared memory concurrency	Memory management

Abstraction	Synchronous communication	Transactional memory	Garbage collection
↓	↓	↓	↓
Domain	Asynchronous distributed system	Shared memory concurrency	Memory management

Synchronous communication =
atomic { data transfer +
synchronization }

Abstraction	Synchronous communication	Transactional memory	Garbage collection
↓	↓	↓	↓
Domain	Asynchronous distributed system	Shared memory concurrency	Memory management

Synchronous communication =
atomic { data transfer +
synchronization }



synchrony latency

Can we discharge synchronous communications asynchronously and ensure observable equivalence?

Can we discharge synchronous communications asynchronously and ensure observable equivalence?

Formalize:

$$[\![\, \texttt{send}\,(c, v)]\!]k \;\equiv\; [\![\, \texttt{asend}\,(c, v)]\!]k$$

Implement:

Distributed Concurrent ML in MultiMLton +
Speculative execution framework

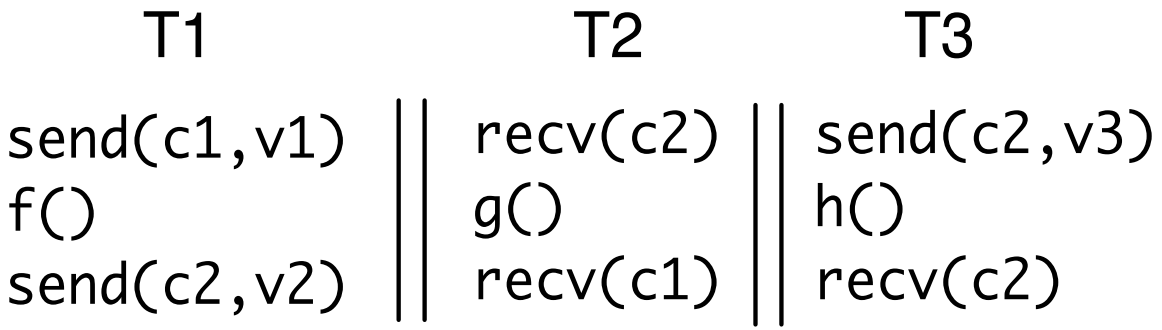
Concurrent ML

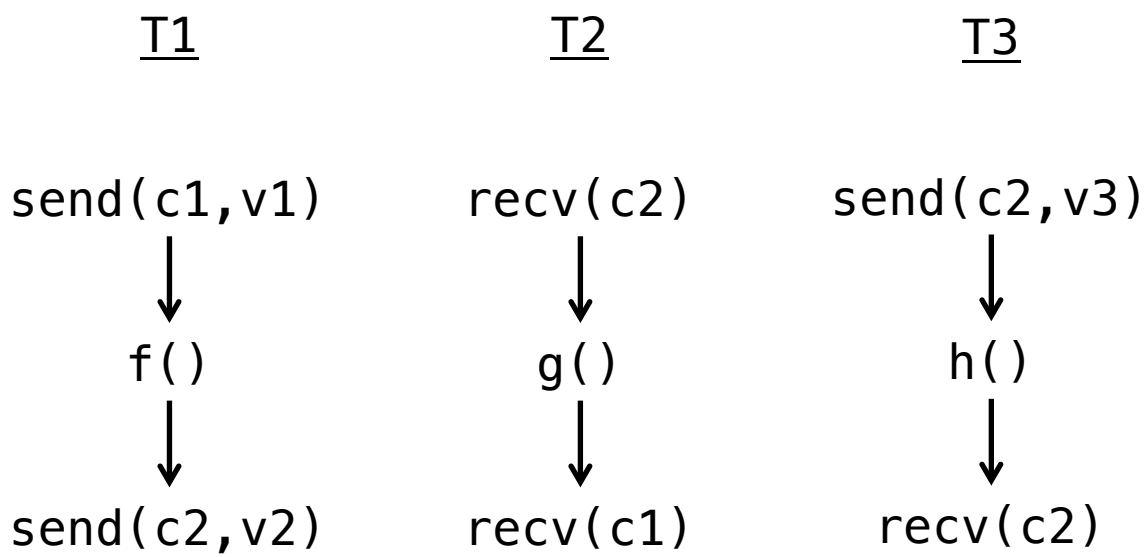
```
val spawn      : (unit -> unit) -> thread_id

val channel    : unit -> 'a chan
val send       : 'a chan * 'a -> unit
val recv       : 'a chan -> 'a

val sendEvt    : 'a chan * 'a -> unit event
val recvEvt    : 'a chan -> 'a event
val sync       : 'a event -> 'a
val never      : 'a event
val alwaysEvt  : 'a -> 'a event
val wrap       : 'a event -> ('a -> 'b) -> 'b event
val guard      : (unit -> 'a event) -> 'a event
val choose     : 'a event list -> 'a event
...
```

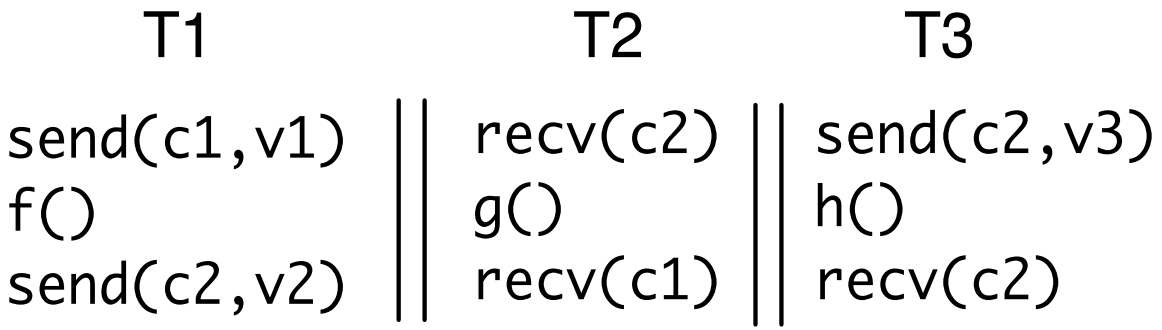
Thread creation

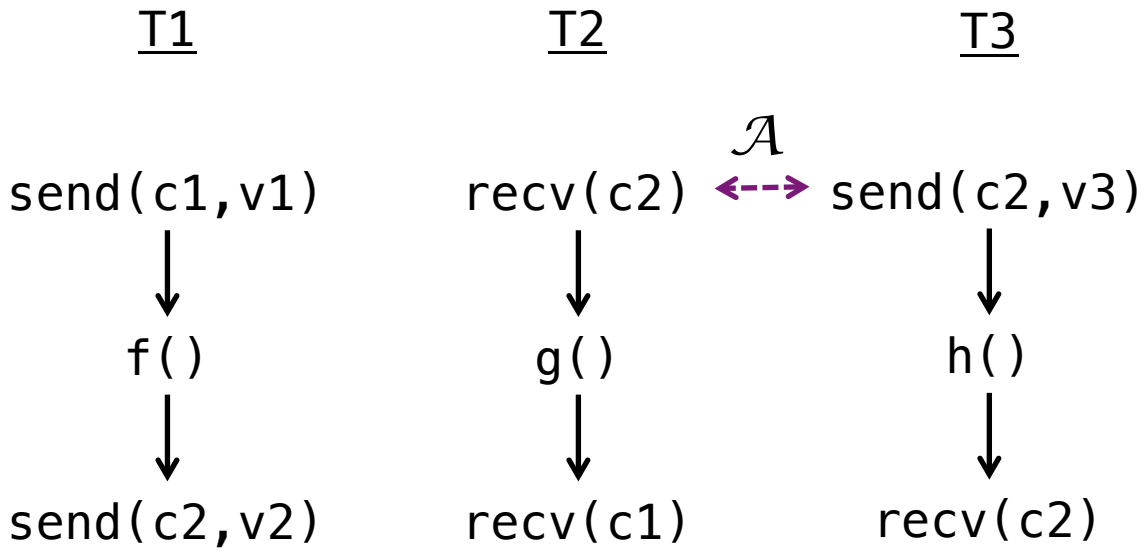
Synchronous message passing

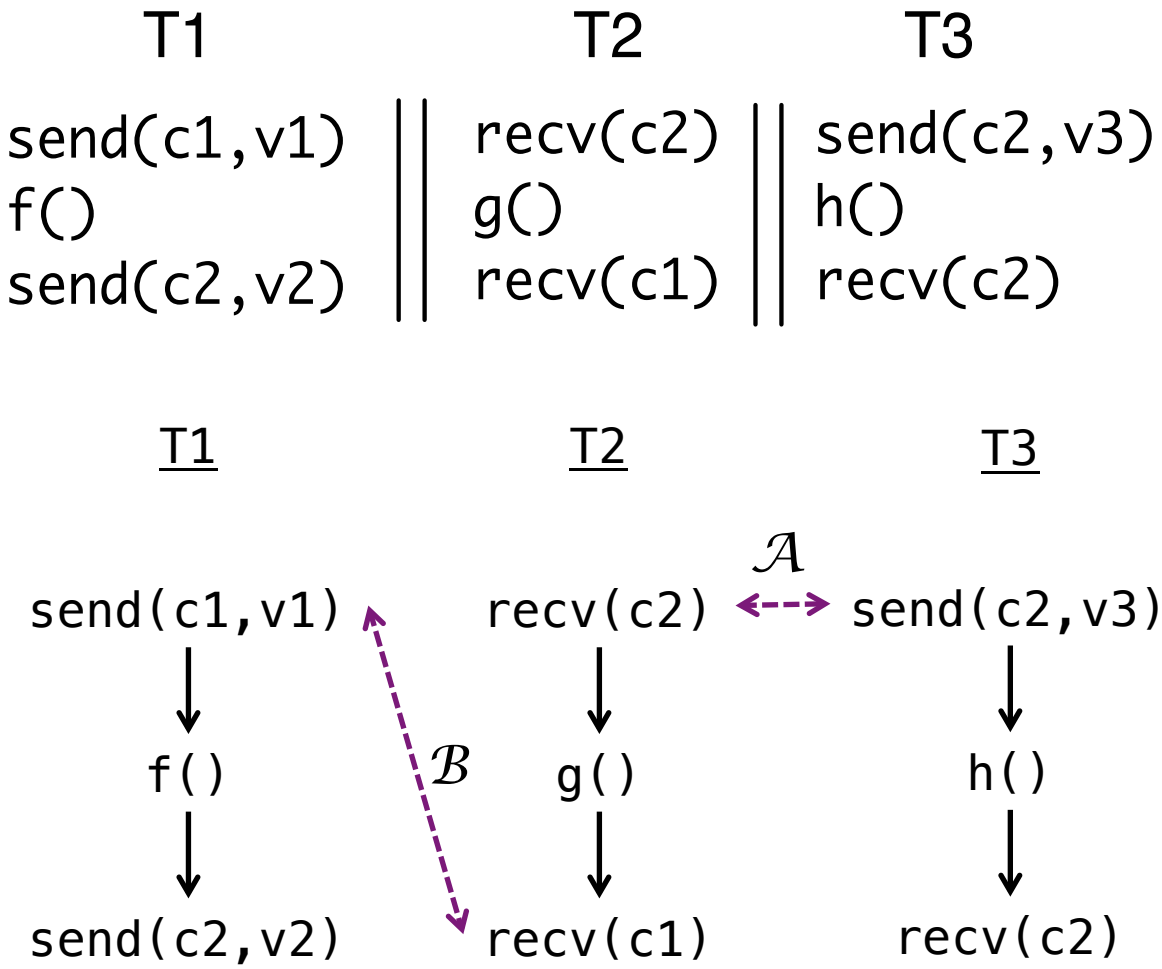
First-class events

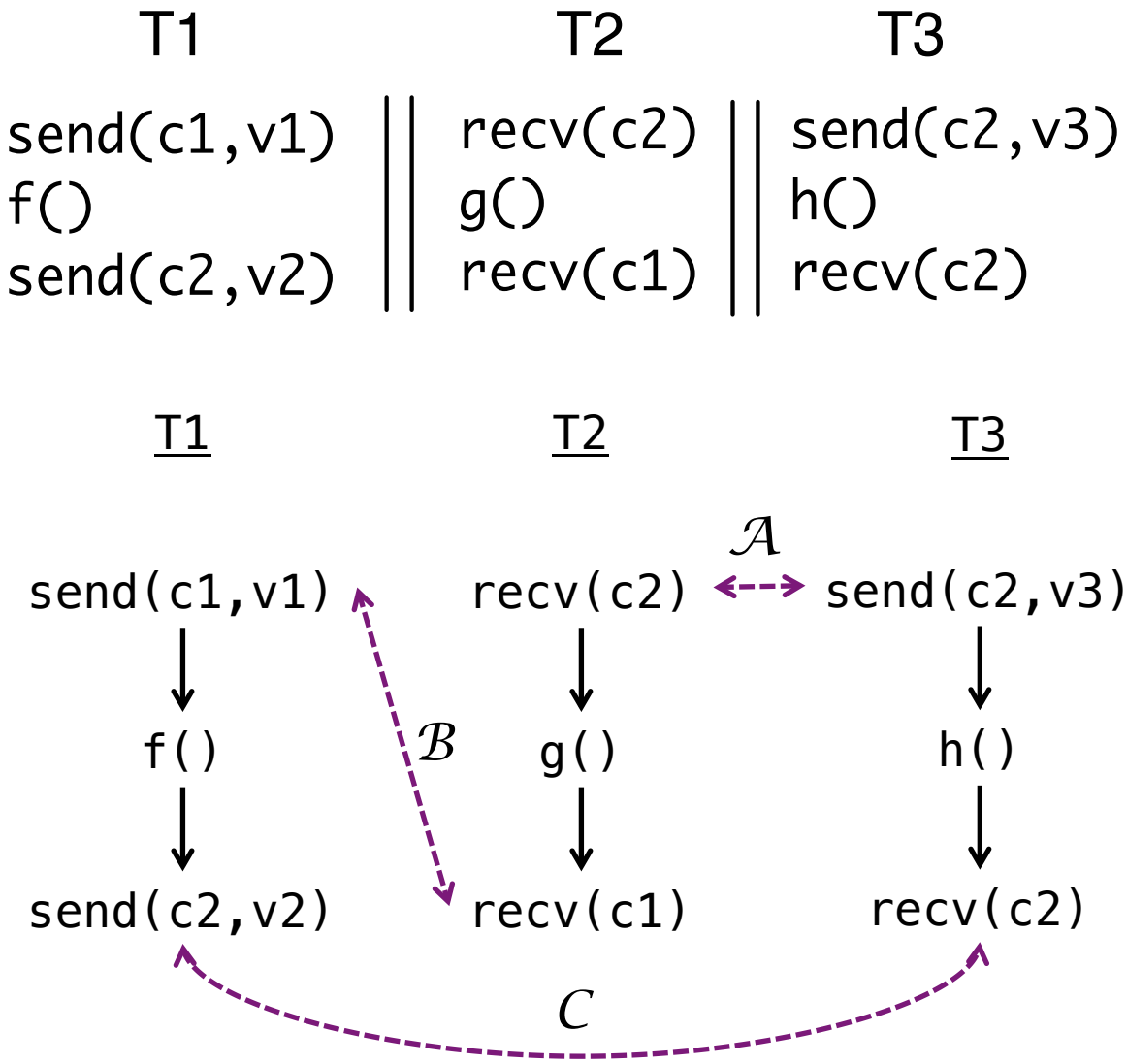

T1
T2
T3
send(c1,v1)
f()
send(c2,v2)
recv(c2)
g()
recv(c1)
send(c2,v3)
h()
recv(c2)

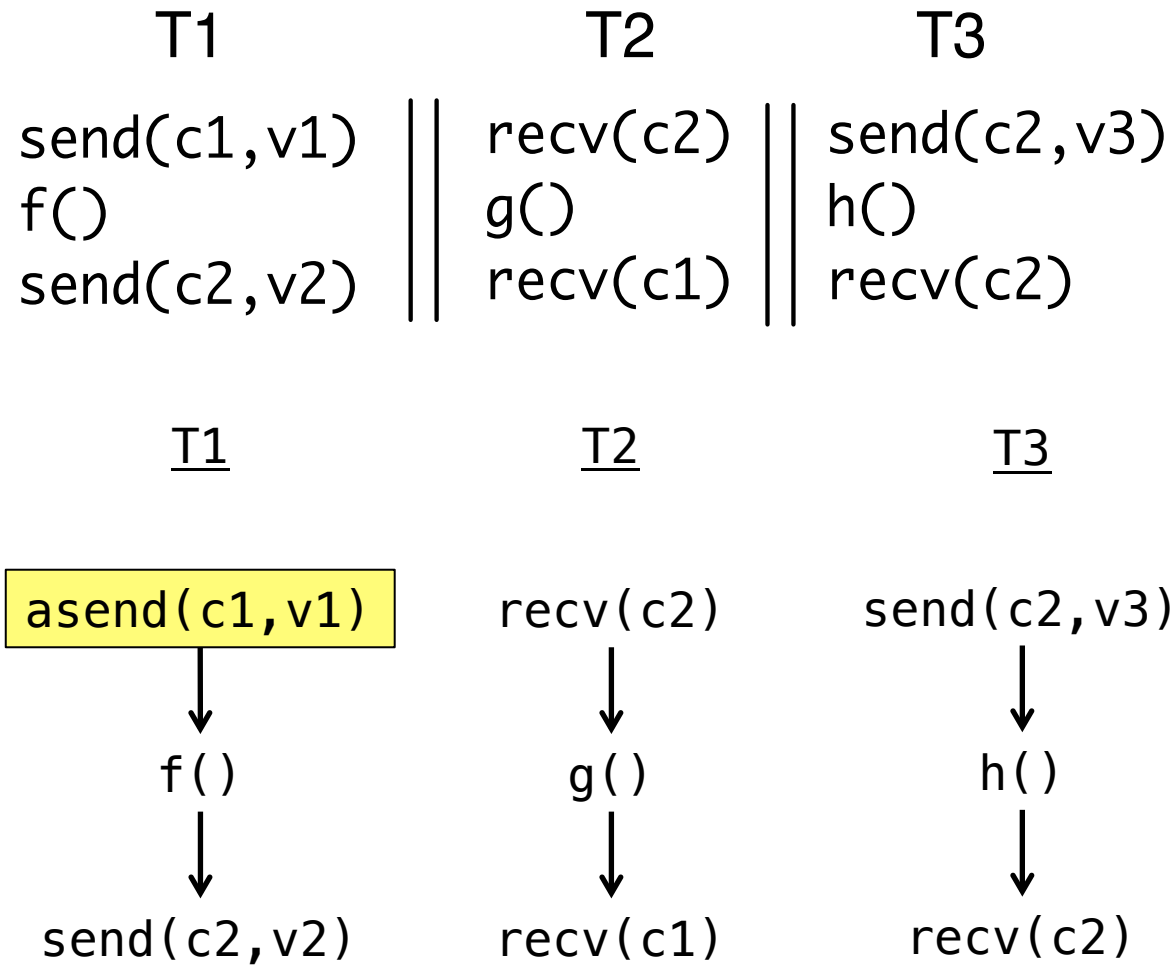

T1
T2
T3
send(c1,v1)
f()
send(c2,v2)
recv(c2)
g()
recv(c1)
send(c2,v3)
h()
recv(c2)

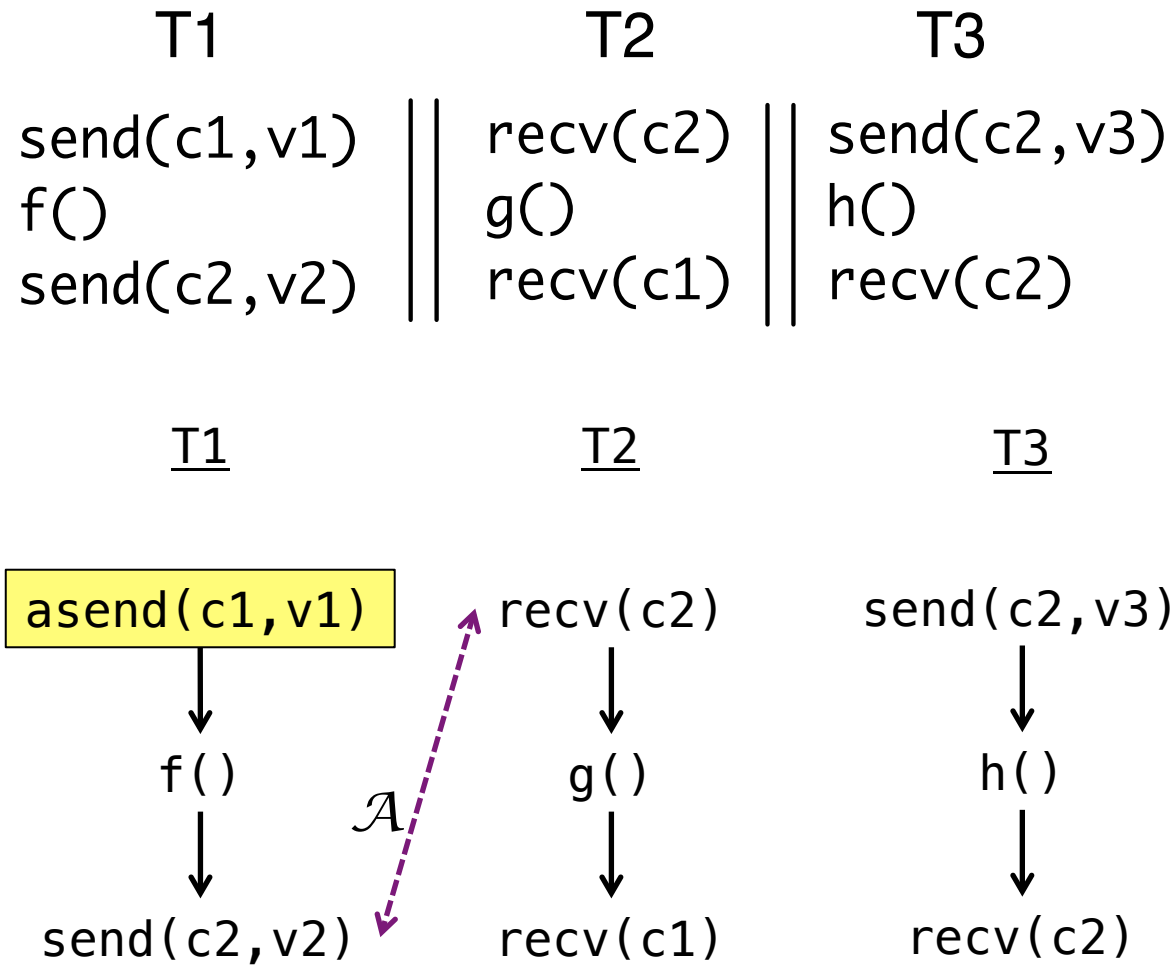

T1
T2
T3
send(c1,v1)
f()
send(c2,v2)
recv(c2)
g()
recv(c1)
send(c2,v3)
h()
recv(c2)

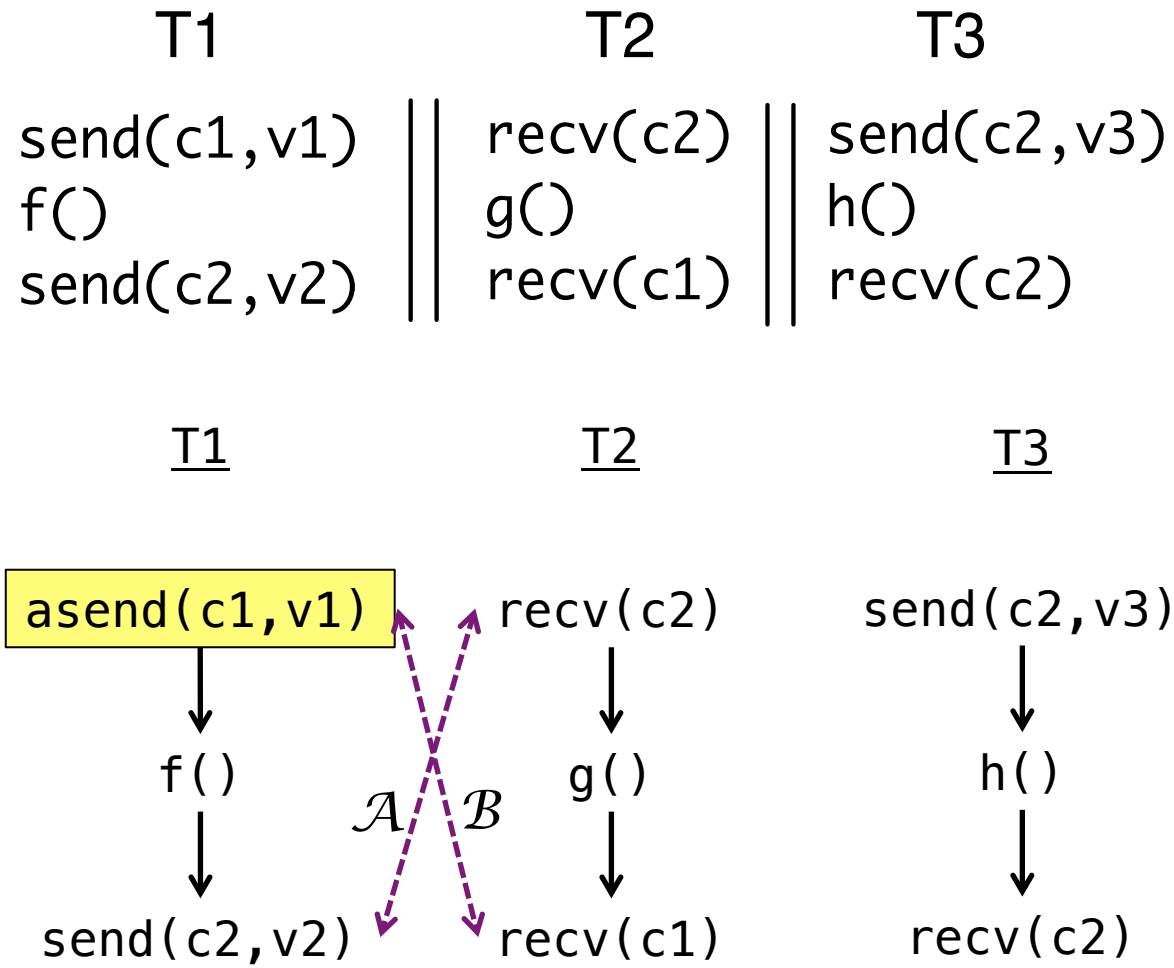

T1
T2
T3
𝒜
send(c1,v1)
f()
send(c2,v2)
recv(c2)
g()
recv(c1)
send(c2,v3)
h()
recv(c2)

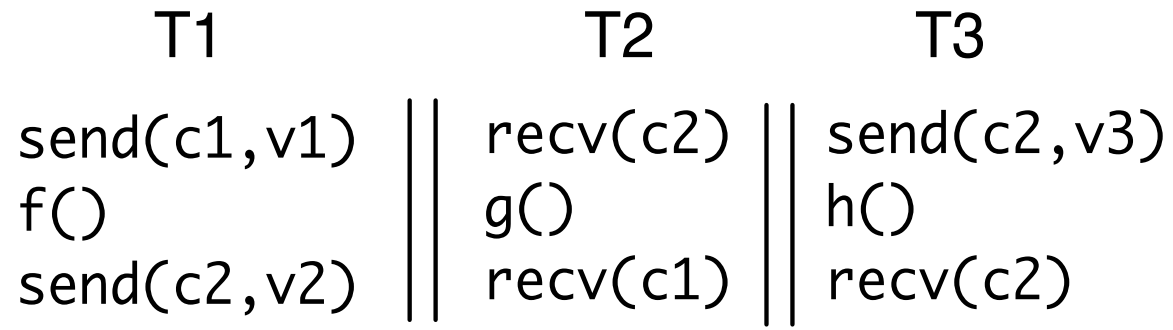

T1
T2
T3
send(c1,v1)
f()
send(c2,v2)
recv(c2)
g()
recv(c1)
send(c2,v3)
h()
recv(c2)
T1
T2
T3
send(c1,v1)
f()
send(c2,v2)
recv(c2)
g()
recv(c1)
send(c2,v3)
h()
recv(c2)
A
B

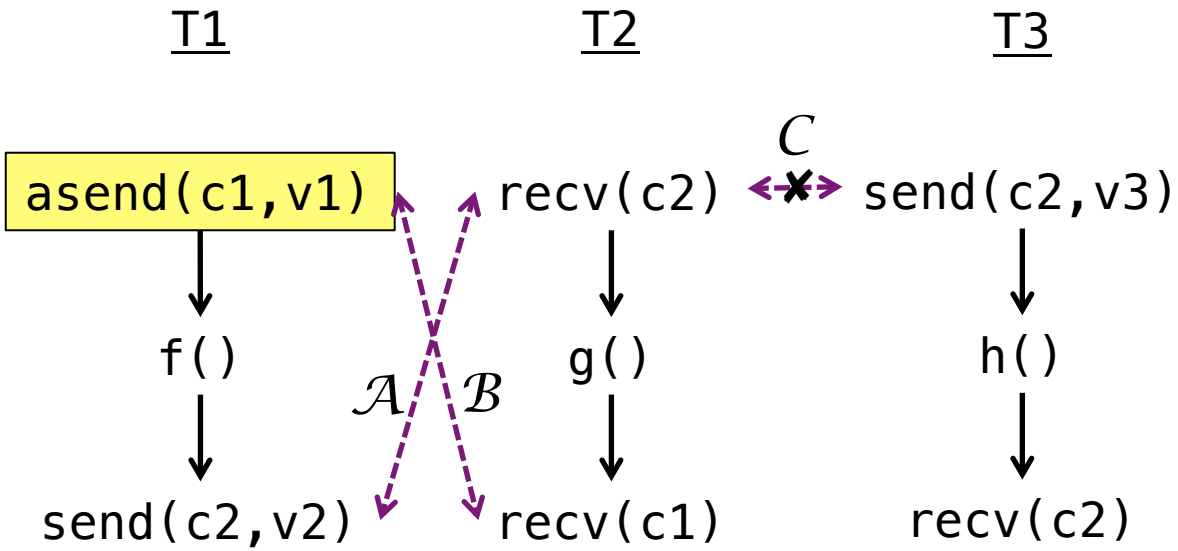

T1
T2
T3
send(c1,v1)
f()
send(c2,v2)
recv(c2)
g()
recv(c1)
send(c2,v3)
h()
recv(c2)
T1
T2
T3
send(c1,v1)
f()
send(c2,v2)
recv(c2)
g()
recv(c1)
send(c2,v3)
h()
recv(c2)
A
B
C

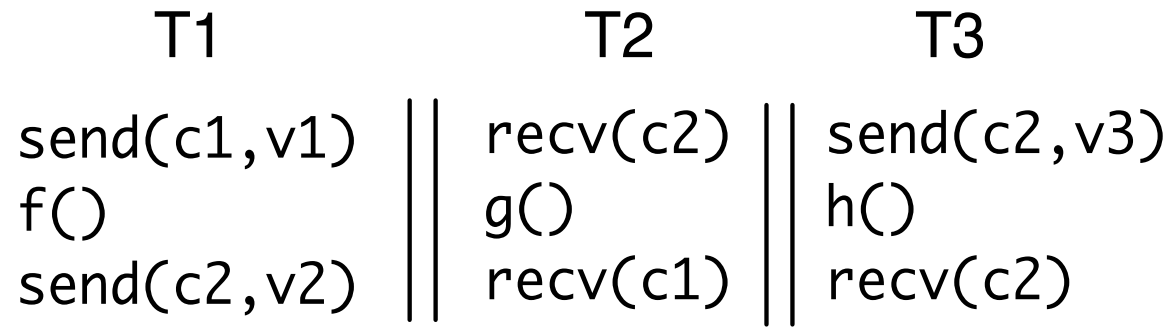

T1
T2
T3
send(c1,v1)
f()
send(c2,v2)
recv(c2)
g()
recv(c1)
send(c2,v3)
h()
recv(c2)
T1
T2
T3
asend(c1,v1)
f()
send(c2,v2)
recv(c2)
g()
recv(c1)
send(c2,v3)
h()
recv(c2)

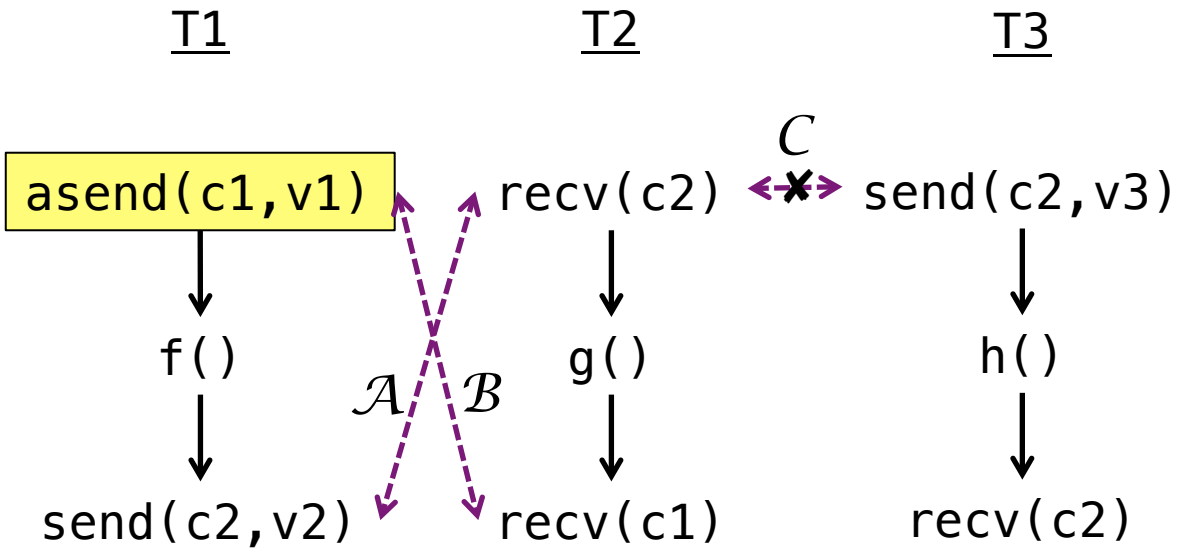

T1
T2
T3
send(c1,v1)
f()
send(c2,v2)
recv(c2)
g()
recv(c1)
send(c2,v3)
h()
recv(c2)
T1
T2
T3
asend(c1,v1)
f()
send(c2,v2)
recv(c2)
g()
recv(c1)
send(c2,v3)
h()
recv(c2)
A


T1
T2
T3
send(c1,v1)
f()
send(c2,v2)
recv(c2)
g()
recv(c1)
send(c2,v3)
h()
recv(c2)
T1
T2
T3
asend(c1,v1)
f()
send(c2,v2)
recv(c2)
g()
recv(c1)
send(c2,v3)
h()
recv(c2)
A
B


T1
T2
T3
send(c1,v1)
f()
send(c2,v2)
recv(c2)
g()
recv(c1)
send(c2,v3)
h()
recv(c2)


T1
T2
T3
C
asend(c1,v1)
recv(c2)
send(c2,v3)
f()
g()
h()
A
B
send(c2,v2)
recv(c1)
recv(c2)





Cyclic dependence ⇒ divergent behavior

Distributed group chat app

No central server & causal dependence → *causal broadcast*

Distributed group chat app

No central server & causal dependence → *causal broadcast*

```
fun bsend (BCHAN (vcList, acList), v: 'a, id: int) : unit =
let
  val _ = map (fn vc => if (vc = nth (vcList, id)) then () else send (vc, v))
              vcList (* phase 1 -- Value distribution *)
  val _ = map (fn ac => if (ac = nth (acList, id)) then () else recv ac)
              acList (* phase 2 -- Acknowledgments *)
in ()
end
```

synchronously send values

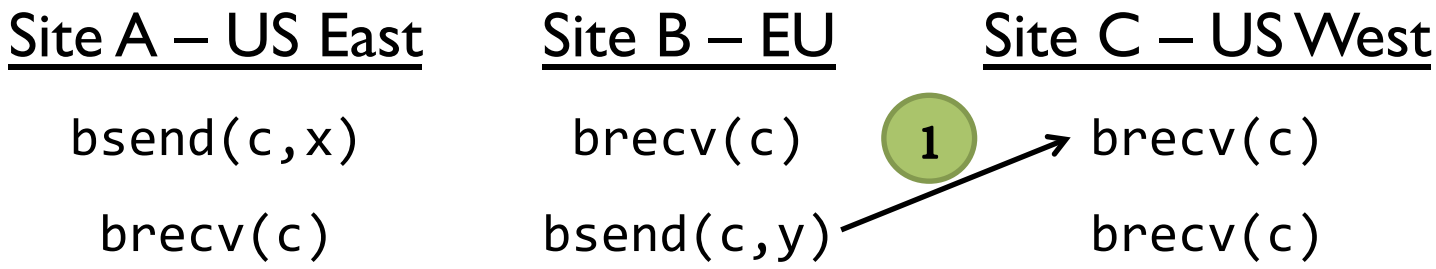
prevent receivers from proceeding until all members have received the value

Distributed group chat app: Performance

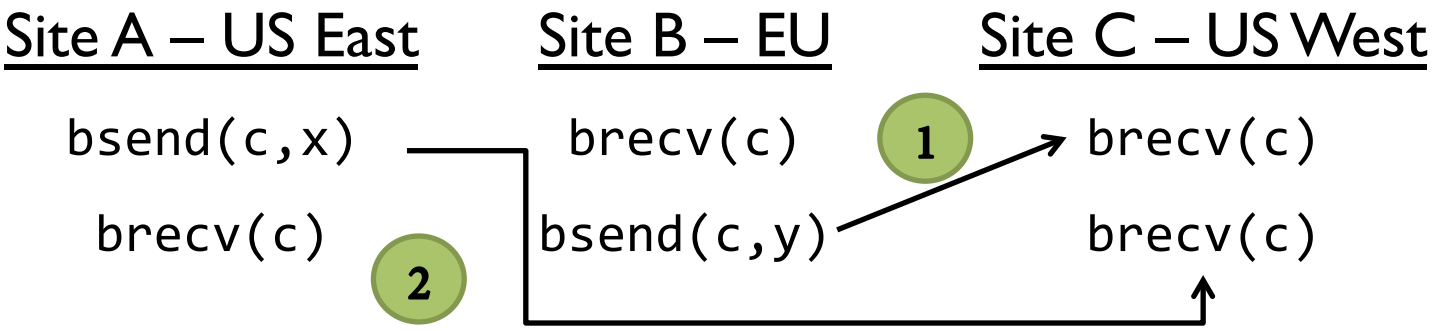
Distributed group chat app: Performance

Site A – US East	Site B – EU	Site C – US West
`bsend(c,x)`	`brecv(c)`	`brecv(c)`
`brecv(c)`	`bsend(c,y)`	`brecv(c)`

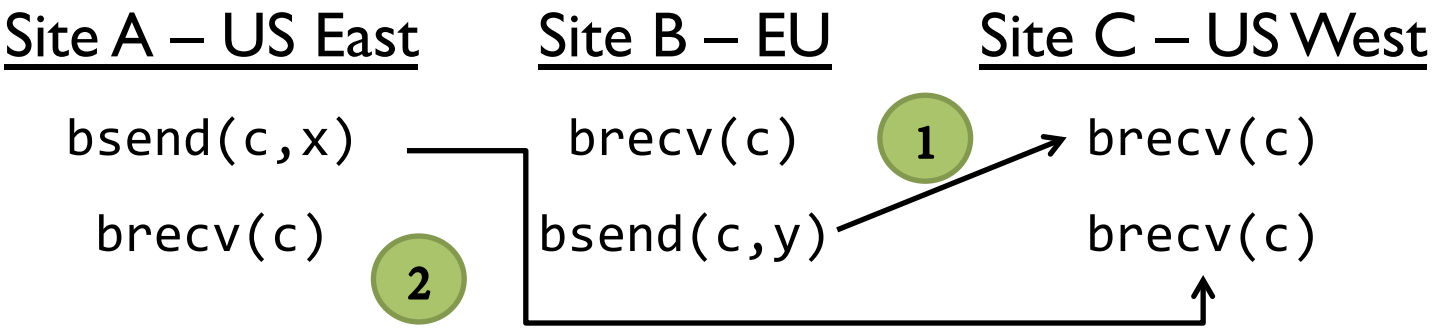
Distributed group chat app: Performance



Distributed group chat app: Performance



Distributed group chat app: Performance



Execution	*Avg.time (ms)*	*Errors*
Sync	1540	0
Unsafe Async	520	7
Safe Async ($\mathbb{R}^{\text{CML}}$)	533	0

Formalization

Reason axiomatically

$$\mathrm{E} := \langle \mathrm{P}, \mathrm{A}, \rightarrow_{po}, \rightarrow_{co} \rangle$$

Happens-before relation

$$\begin{aligned} \rightarrow_{hb} \quad = \quad & (\rightarrow_{po} \cup \rightarrow_{td} \cup \\ & \{(\alpha, \beta) \mid \alpha \rightarrow_{co} \alpha' \wedge \alpha' \rightarrow_{po} \beta\} \cup \\ & \{(\beta, \alpha) \mid \beta \rightarrow_{po} \alpha' \wedge \alpha' \rightarrow_{co} \alpha\})^{+} \end{aligned}$$

Well-formed execution

$$\text{Obs (WF\_Exec (P))} \in \{\text{Obs (Sync\_Exec (P))}\}$$

Formalization

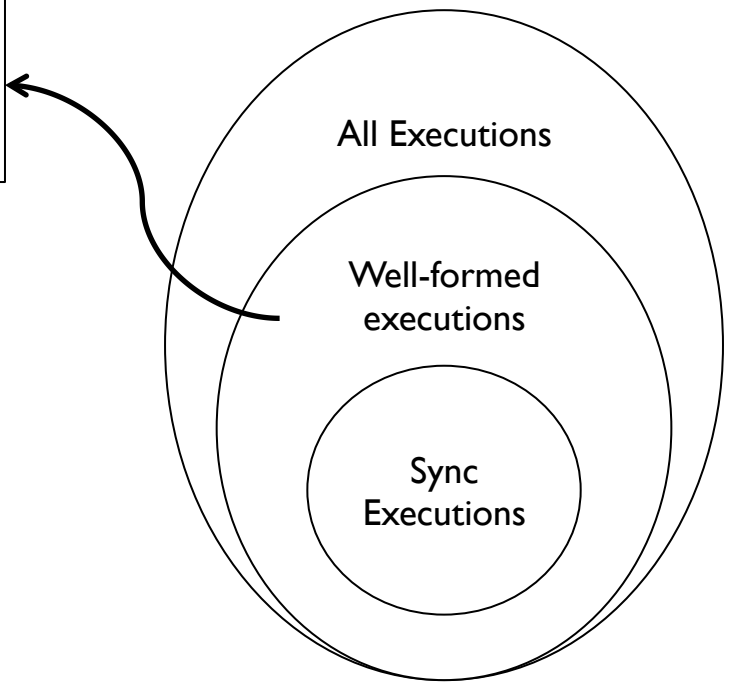
Reason axiomatically $\mathrm{E} := \langle \mathrm{P}, \mathrm{A}, \rightarrow_{po}, \rightarrow_{co} \rangle$

Happens-before relation

$$\rightarrow_{hb} \quad = \quad (\rightarrow_{po} \cup \rightarrow_{td} \cup \{(\alpha, \beta) \mid \alpha \rightarrow_{co} \alpha' \wedge \alpha' \rightarrow_{po} \beta\} \cup \{(\beta, \alpha) \mid \beta \rightarrow_{po} \alpha' \wedge \alpha' \rightarrow_{co} \alpha\})^{+}$$

Well-formed execution $\text{Obs (WF\_Exec (P))} \in \{\text{Obs (Sync\_Exec (P))}\}$

- No happens before cycle
- Sensible intra-thread semantics
- No outstanding speculative actions



Formalization

Reason axiomatically $\mathrm{E} := \langle \mathrm{P}, \mathrm{A}, \rightarrow_{po}, \rightarrow_{co} \rangle$

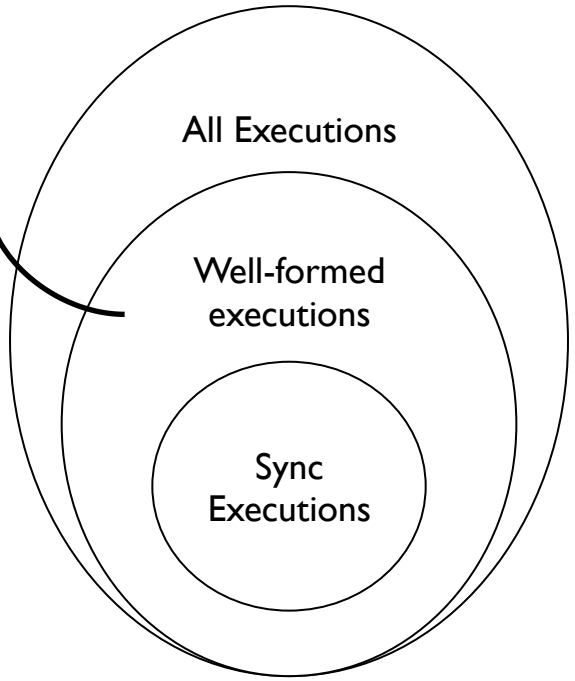
Happens-before relation

$$\rightarrow_{hb} = (\rightarrow_{po} \cup \rightarrow_{td} \cup \{(\alpha, \beta) \mid \alpha \rightarrow_{co} \alpha' \wedge \alpha' \rightarrow_{po} \beta\} \cup \{(\beta, \alpha) \mid \beta \rightarrow_{po} \alpha' \wedge \alpha' \rightarrow_{co} \alpha\})^{+}$$

Well-formed execution $\mathrm{Obs}(\mathrm{WF\_Exec}(\mathrm{P})) \in \{\mathrm{Obs}(\mathrm{Sync\_Exec}(\mathrm{P}))\}$

- No happens before cycle
- Sensible intra-thread semantics
- No outstanding speculative actions

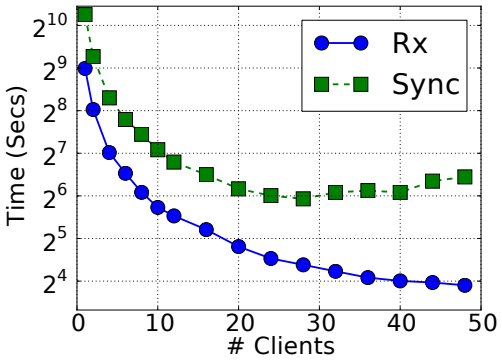
Recipe for implementation



Implementation & Results

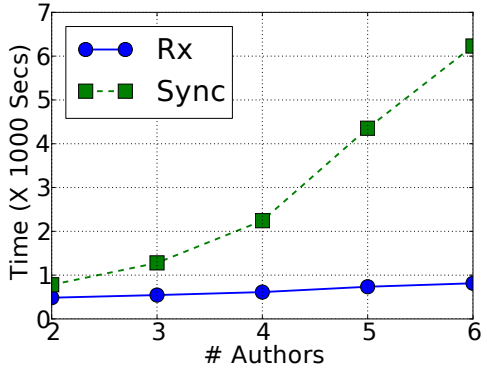
Implementation & Results

- Dependence graph ≡ Axiomatic execution
  - WF Check before observable actions
  - Ill-formed? Rollback and re-execute non-speculatively – Progress!
- WF Check, checkpoint, rollback are uncoordinated!
- Replicated channel consistency through speculative execution
- Benchmark: Optimistic OLTP & P2P Collaborative editing



OLTP

With 48 clients - 5.8X faster than sync
1.4X slower than async



Collaborative Editing

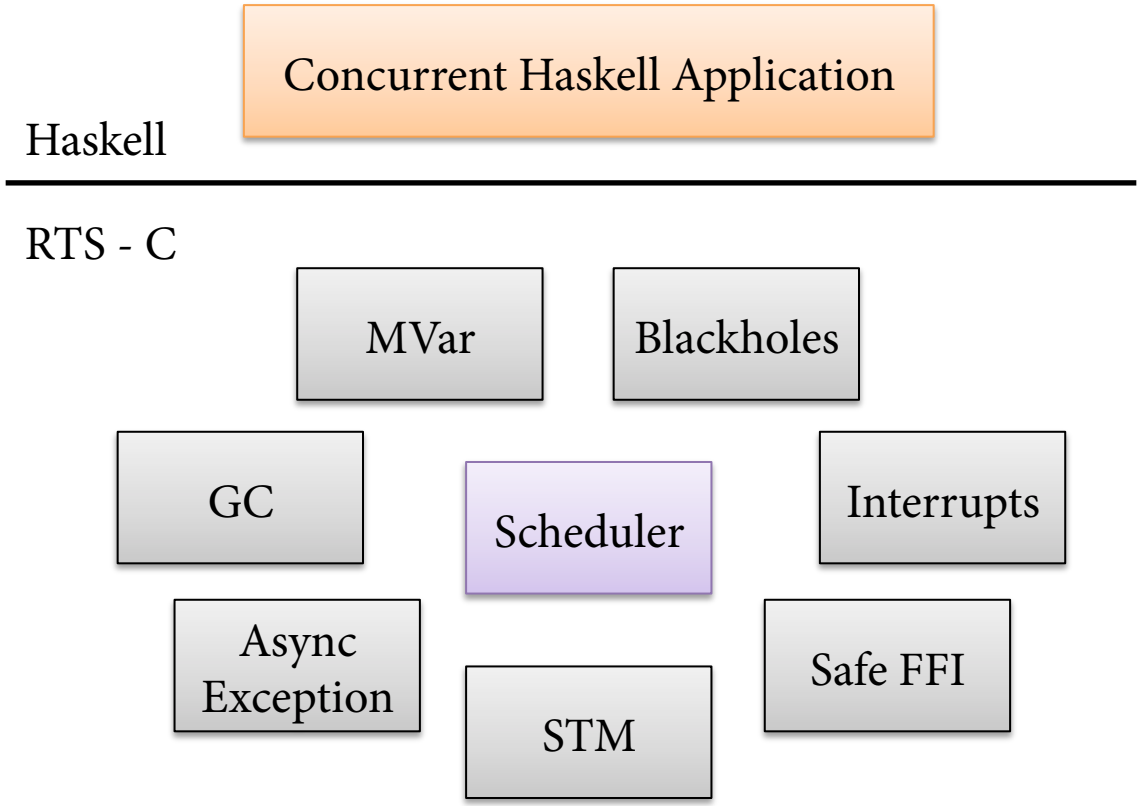
With 6 authors - 7.6X faster than sync
2.3X slower than async

Scheduler Activations in Haskell

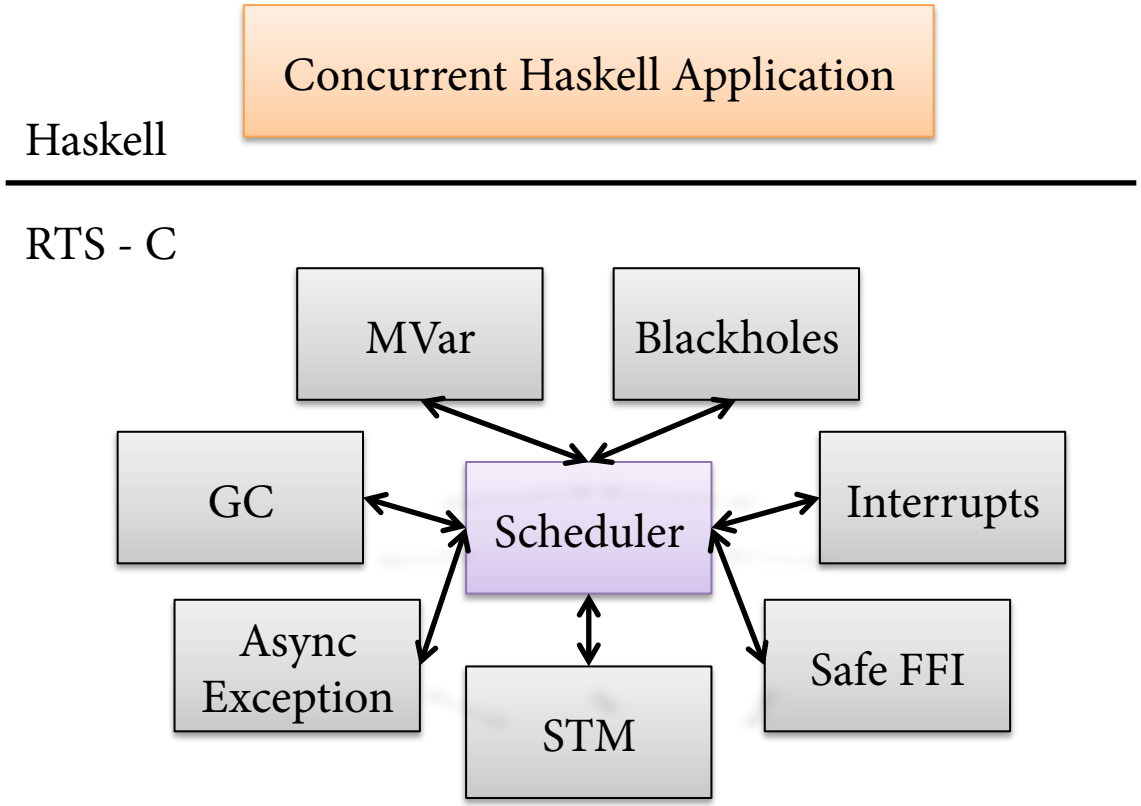
How to write schedulers for Haskell threads as Haskell libraries?

How to write schedulers for Haskell threads as Haskell libraries in GHC?

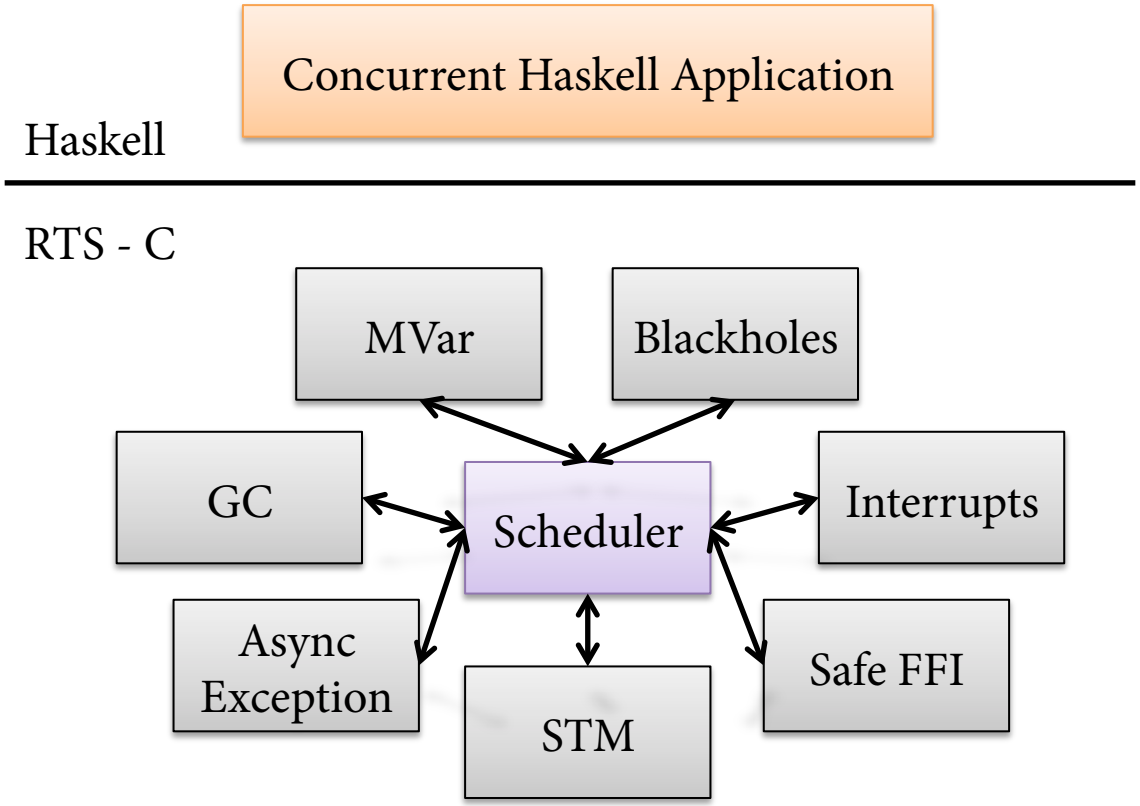
How to write schedulers for Haskell threads as Haskell libraries in GHC?

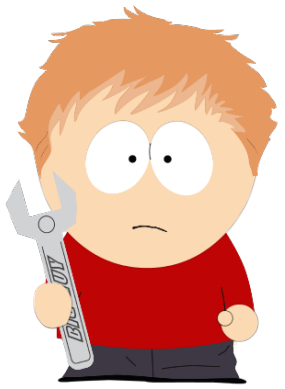


How to write schedulers for Haskell threads as Haskell libraries in GHC?



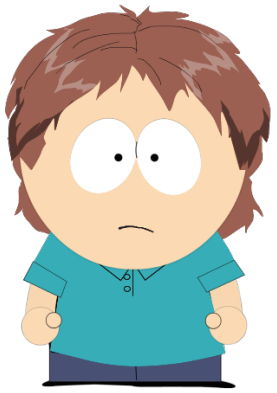
How to write schedulers for Haskell threads as Haskell libraries in GHC?



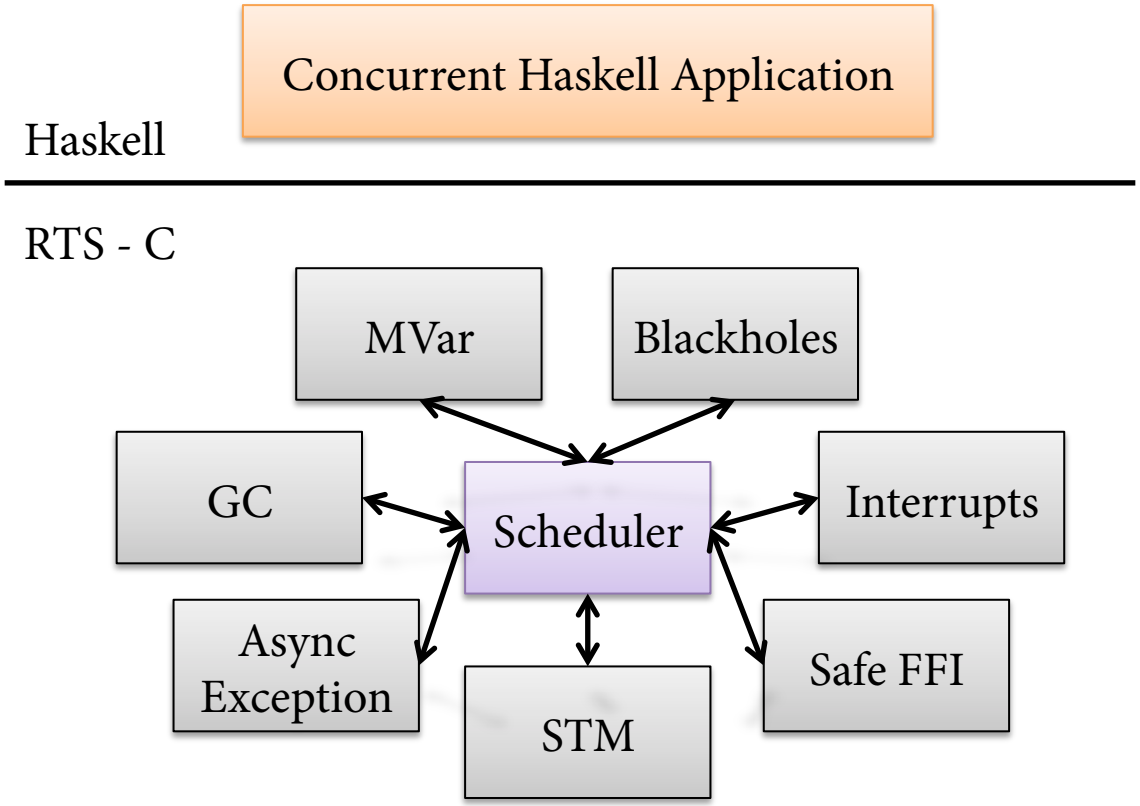


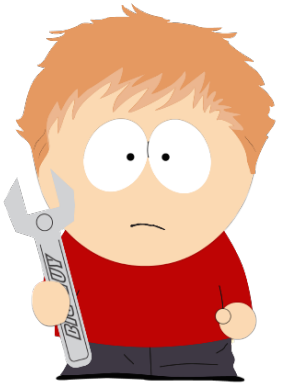
Lang Dev

How to write schedulers for Haskell threads as Haskell libraries in GHC?

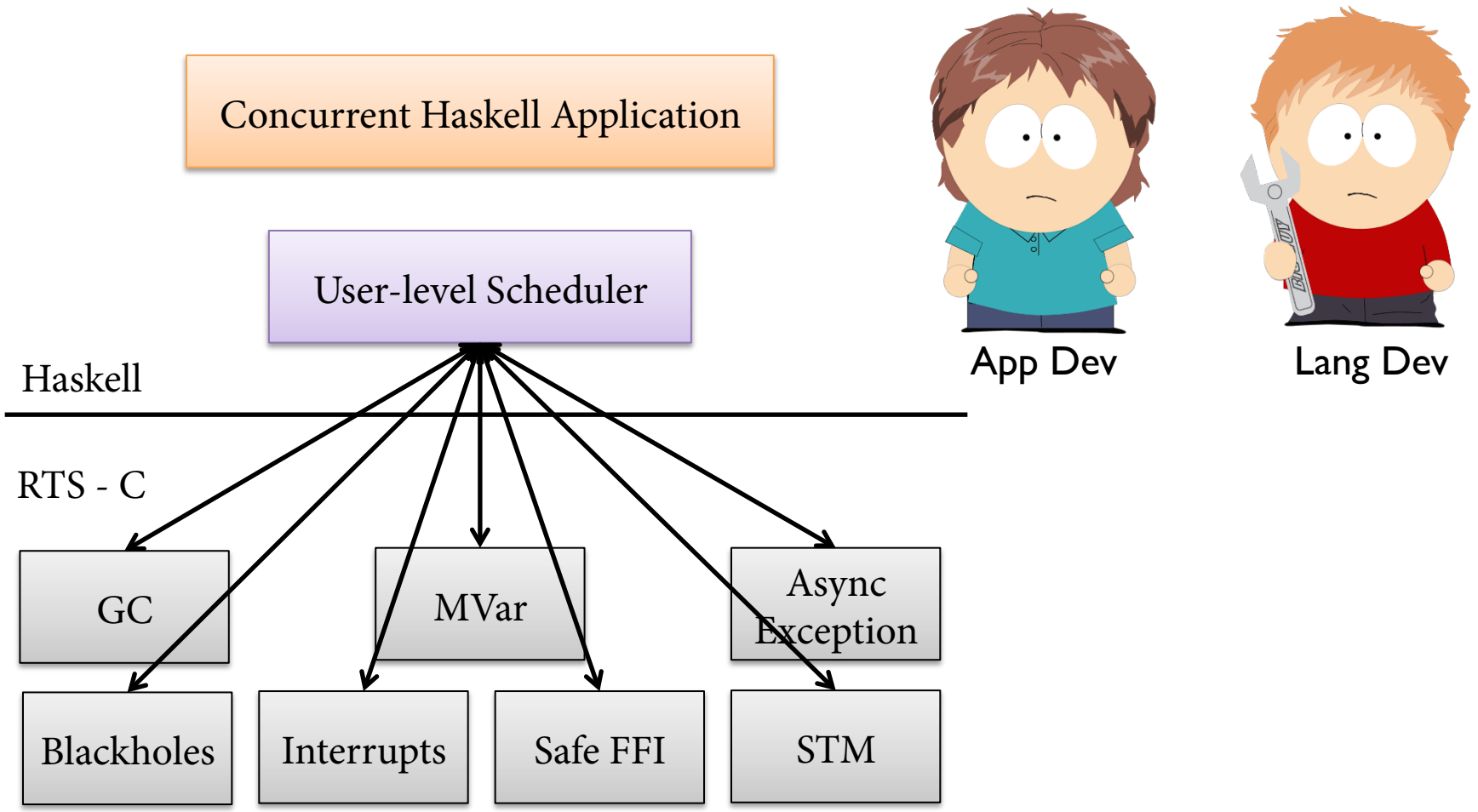


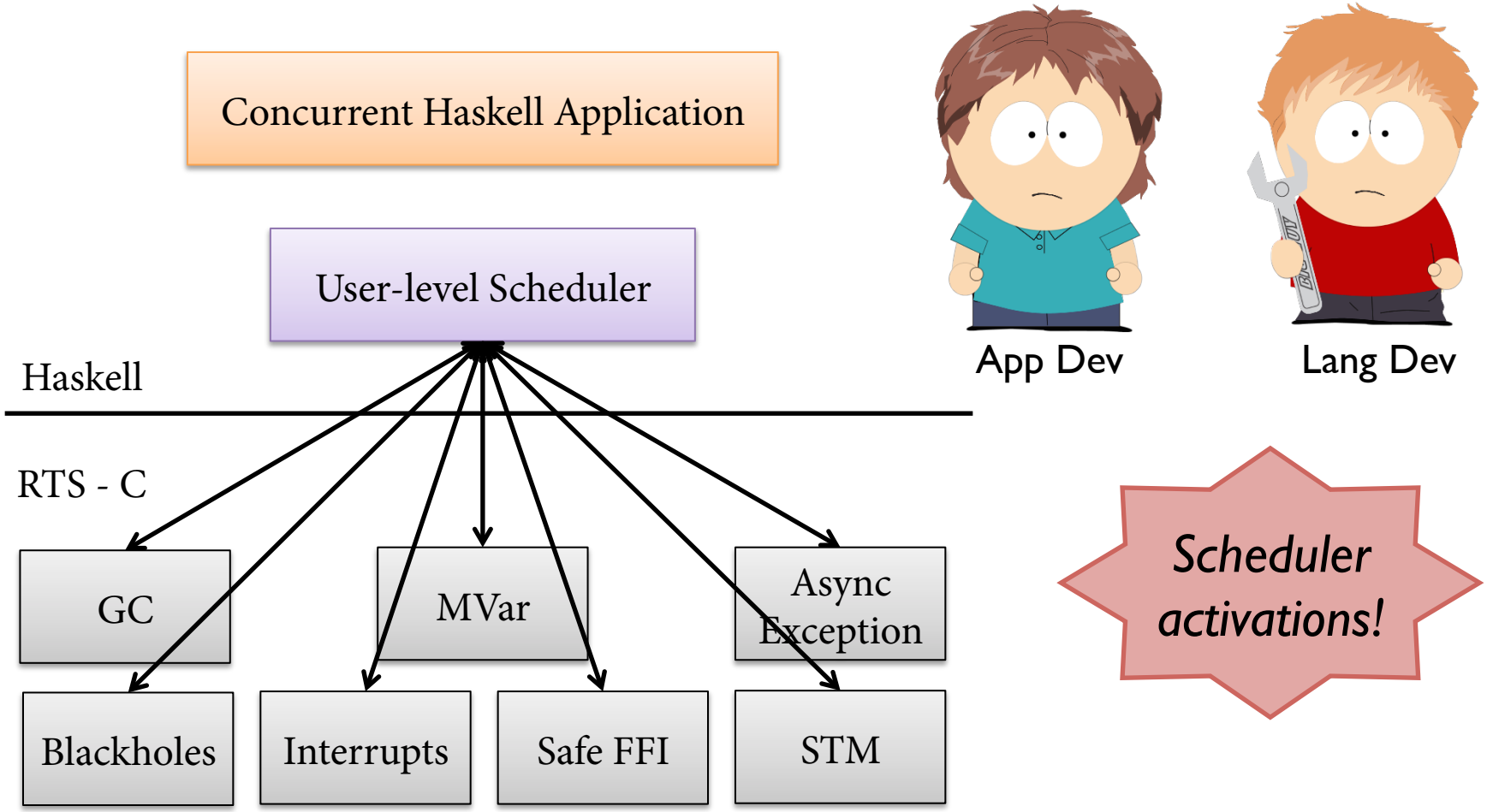
App Dev





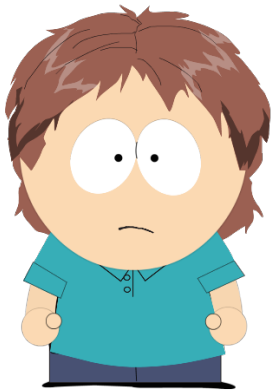
Lang Dev


Concurrent Haskell Application
User-level Scheduler
Haskell
RTS - C
GC
MVar
Async Exception
Blackholes
Interrupts
Safe FFI
STM
App Dev
Lang Dev

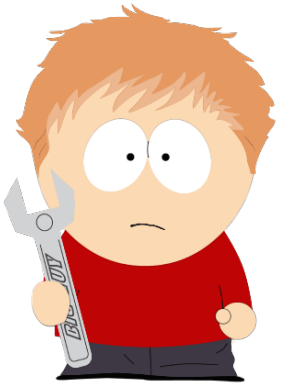

Concurrent Haskell Application
User-level Scheduler
Haskell
RTS - C
GC
MVar
Async Exception
Blackholes
Interrupts
Safe FFI
STM
App Dev
Lang Dev
Scheduler activations!

Concurrent Haskell Application

User-level Scheduler



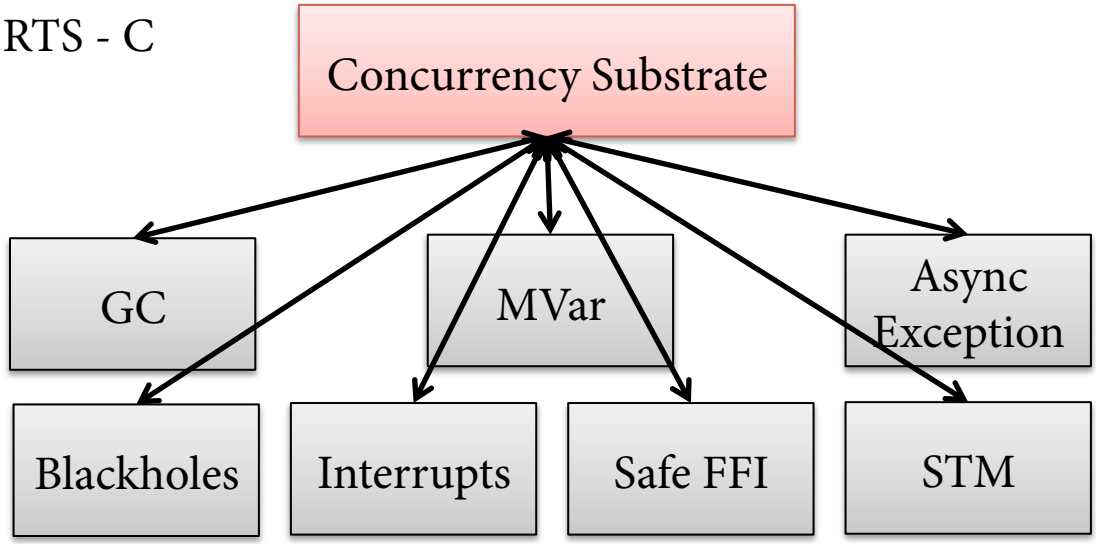
App Dev



Lang Dev

Haskell

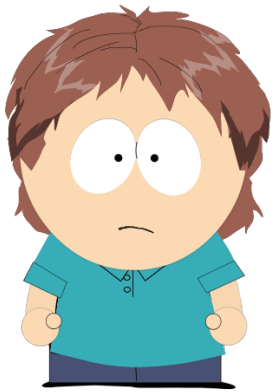
RTS - C



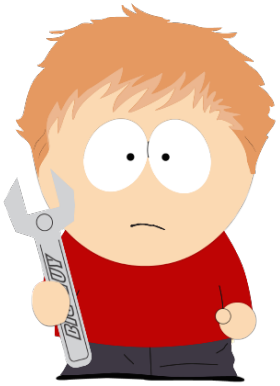
Concurrent Haskell Application

User-level Scheduler

EnQ and DeQ Activations



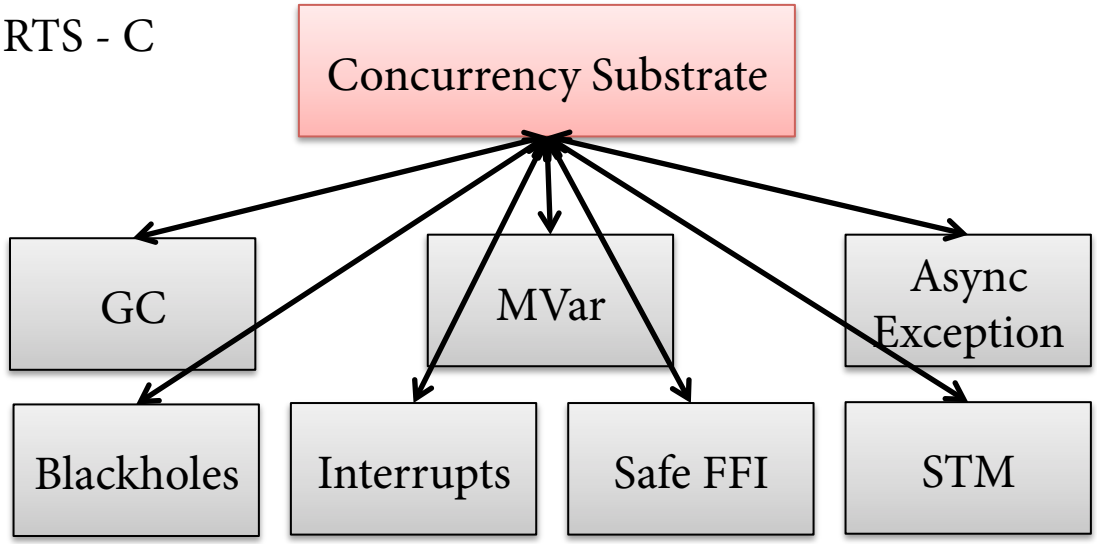
App Dev

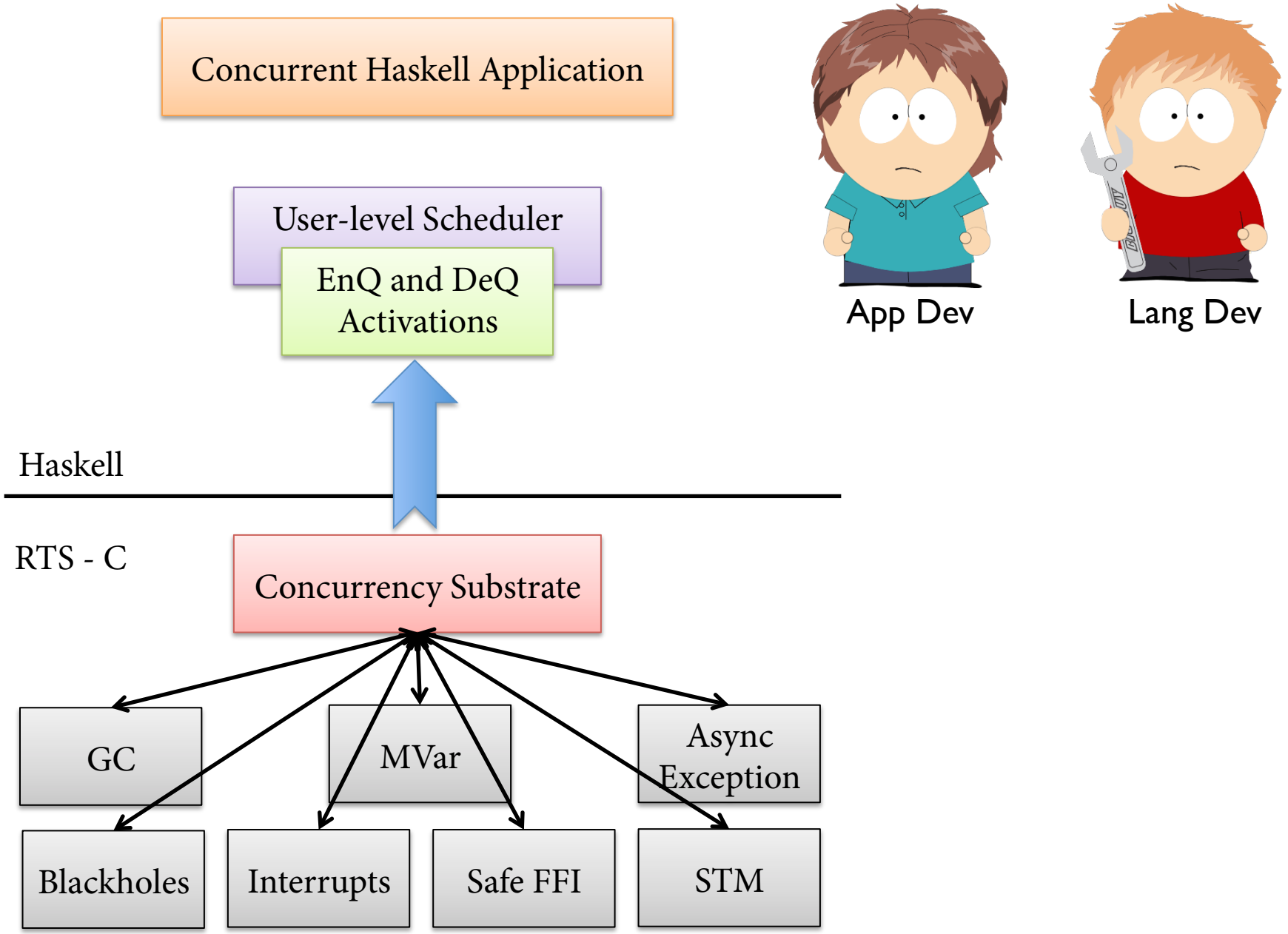


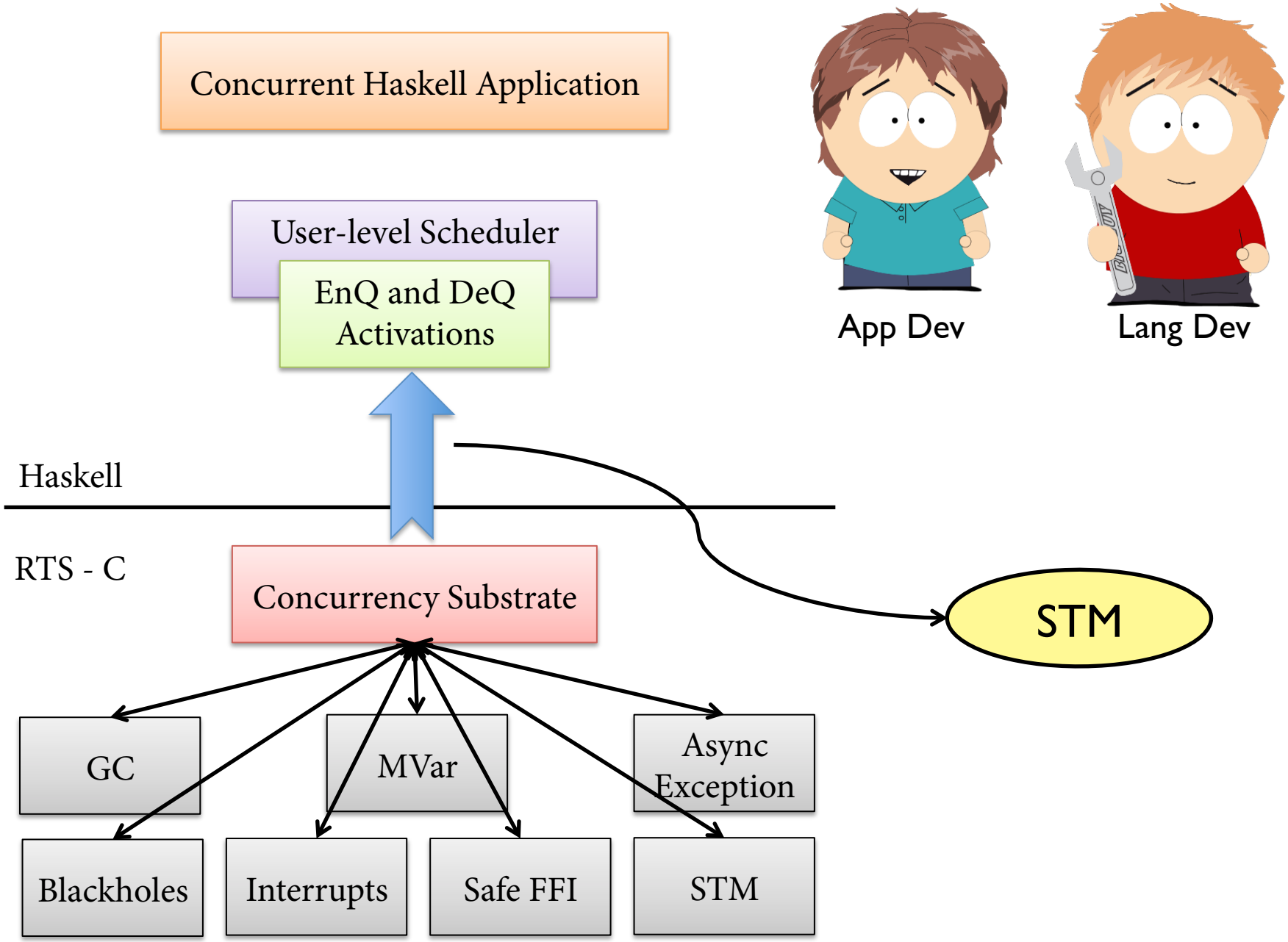
Lang Dev

Haskell

RTS - C




Concurrent Haskell Application
User-level Scheduler
EnQ and DeQ Activations
Haskell
RTS - C
Concurrency Substrate
GC
MVar
Async Exception
Blackholes
Interrupts
Safe FFI
STM
App Dev
Lang Dev


Concurrent Haskell Application
User-level Scheduler
EnQ and DeQ Activations
App Dev
Lang Dev
Haskell
RTS - C
Concurrency Substrate
STM
GC
MVar
Async Exception
Blackholes
Interrupts
Safe FFI
STM

Concurrency Substrate API

```
data SCont
newSCont      :: IO () -> IO SCont
switch        :: (SCont -> STM SCont) -> IO ()
runOnIdleHEC :: SCont -> IO ()
```

Concurrency Substrate API

```
data SCont
newSCont      :: IO () -> IO SCont
switch        :: (SCont -> STM SCont) -> IO ()
runOnIdleHEC  :: SCont -> IO ()

type DequeueAct = SCont -> STM SCont
type EnqueueAct = SCont -> STM ()

-- read activations
dequeueAct     :: DequeueAct
enqueueAct     :: EnqueueAct
-- update activations
setDequeueAct  :: DequeueAct -> IO ()
setEnqueueAct  :: EnqueueAct -> IO ()
```

Concurrency Substrate API

```
data SCont
newSCont      :: IO () -> IO SCont
switch        :: (SCont -> STM SCont) -> IO ()
runOnIdleHEC  :: SCont -> IO ()

type DequeueAct = SCont -> STM SCont
type EnqueueAct = SCont -> STM ()

-- read activations
dequeueAct     :: DequeueAct
enqueueAct     :: EnqueueAct
-- update activations
setDequeueAct  :: DequeueAct -> IO ()
setEnqueueAct  :: EnqueueAct -> IO ()
```

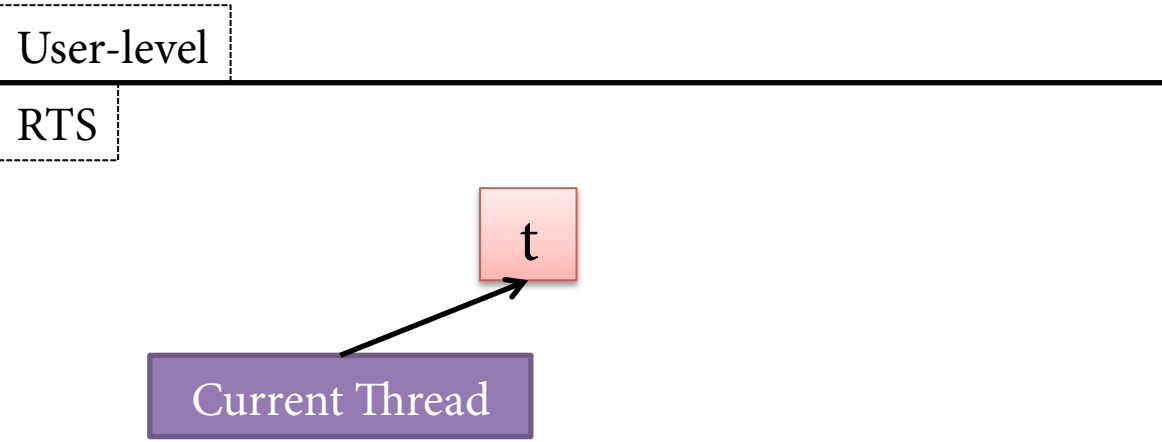
Scheduler access is under STM

Concurrency Substrate API

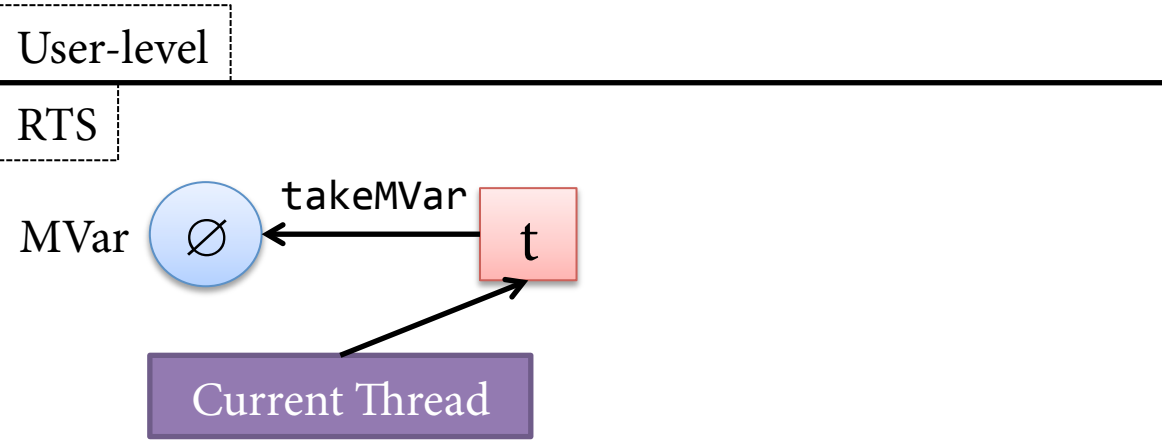
```
data SCont
newSCont      :: IO () -> IO SCont
switch        :: (SCont -> STM SCont) -> IO ()
runOnIdleHEC  :: SCont -> IO ()

type DequeueAct = SCont -> STM SCont
type EnqueueAct = SCont -> STM ()

-- read activations
dequeueAct     :: DequeueAct
enqueueAct     :: EnqueueAct
-- update activations
setDequeueAct  :: DequeueAct -> IO ()
setEnqueueAct  :: EnqueueAct -> IO ()

-- update activations
getAux         :: SCont -> STM Dynamic
setAux         :: SCont -> Dynamic -> STM ()
```

Scheduler access is under STM

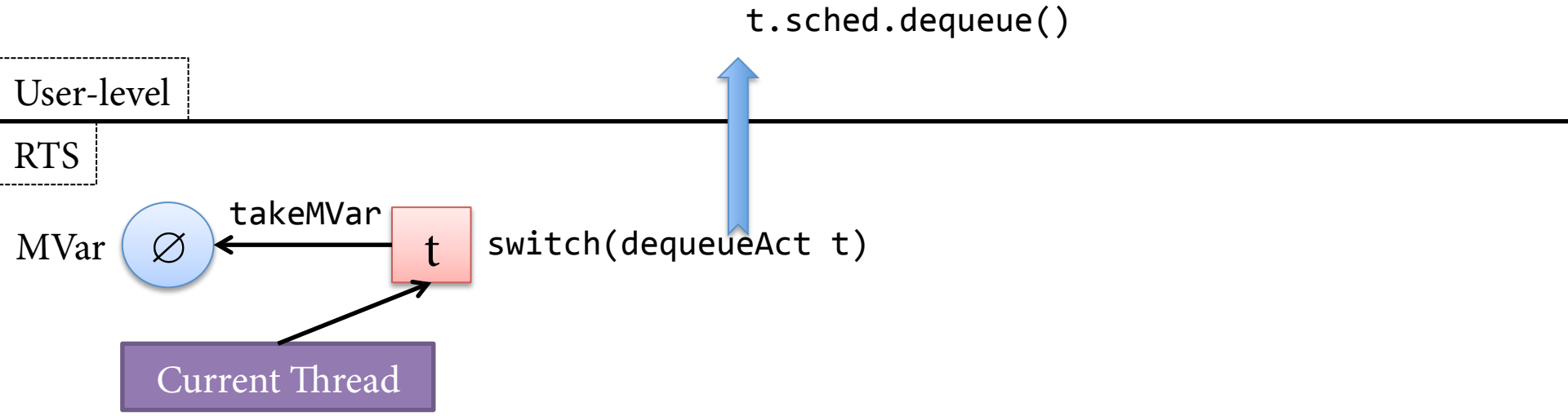
RTS Interaction - Blocking



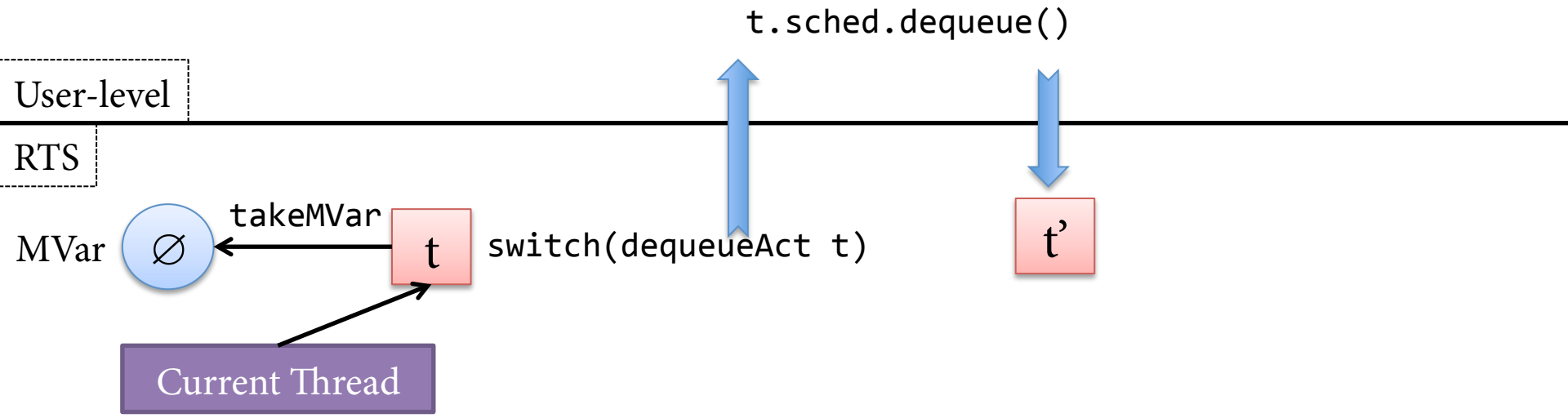
RTS Interaction - Blocking



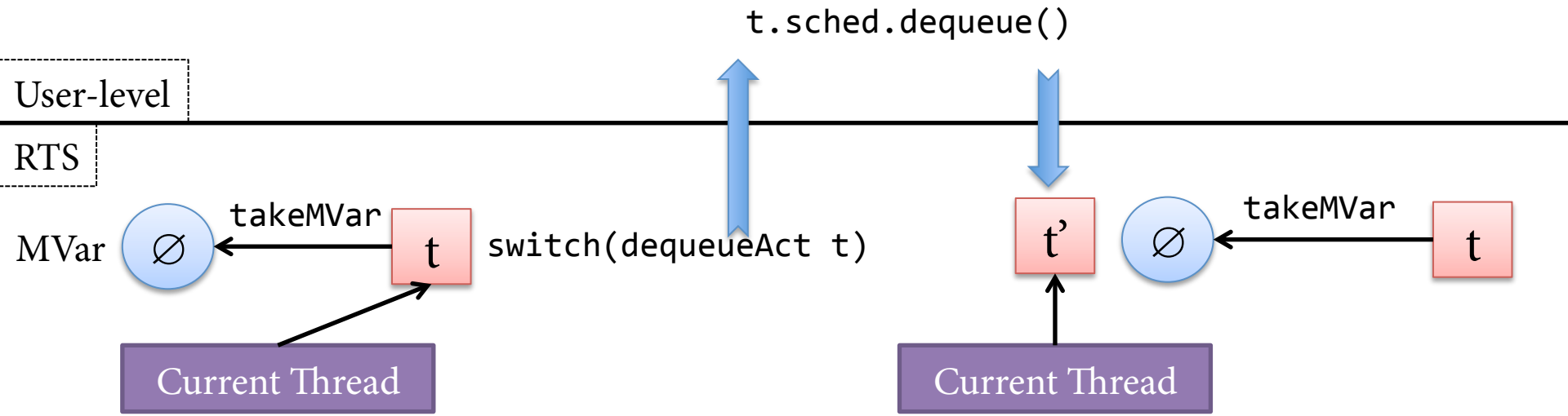
RTS Interaction - Blocking



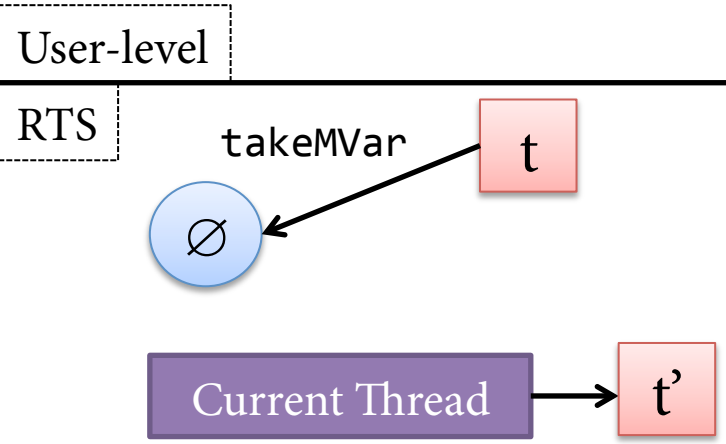
RTS Interaction - Blocking



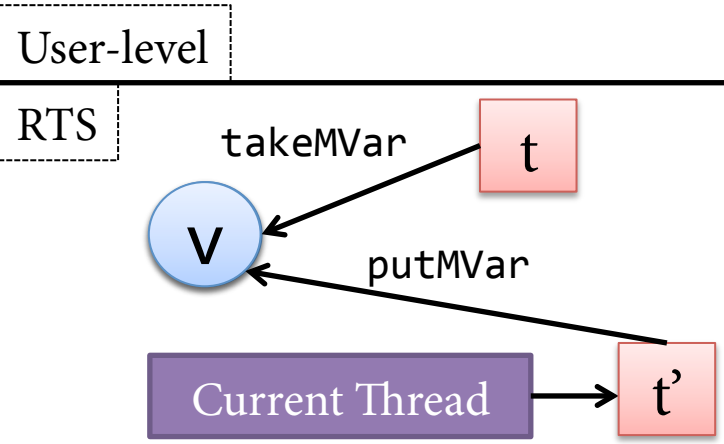
RTS Interaction - Blocking



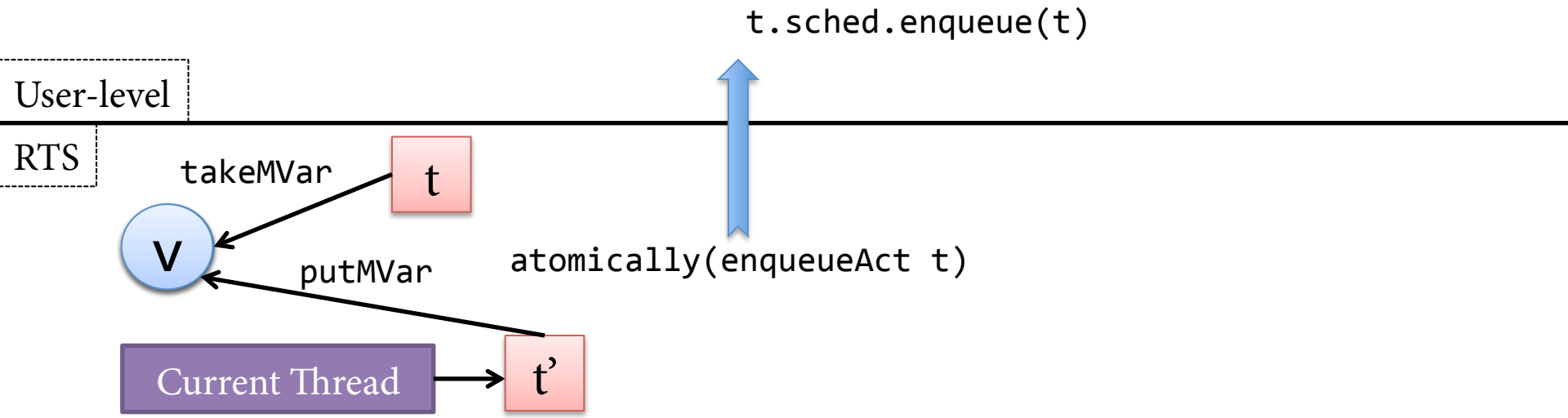
RTS Interaction - Unblocking



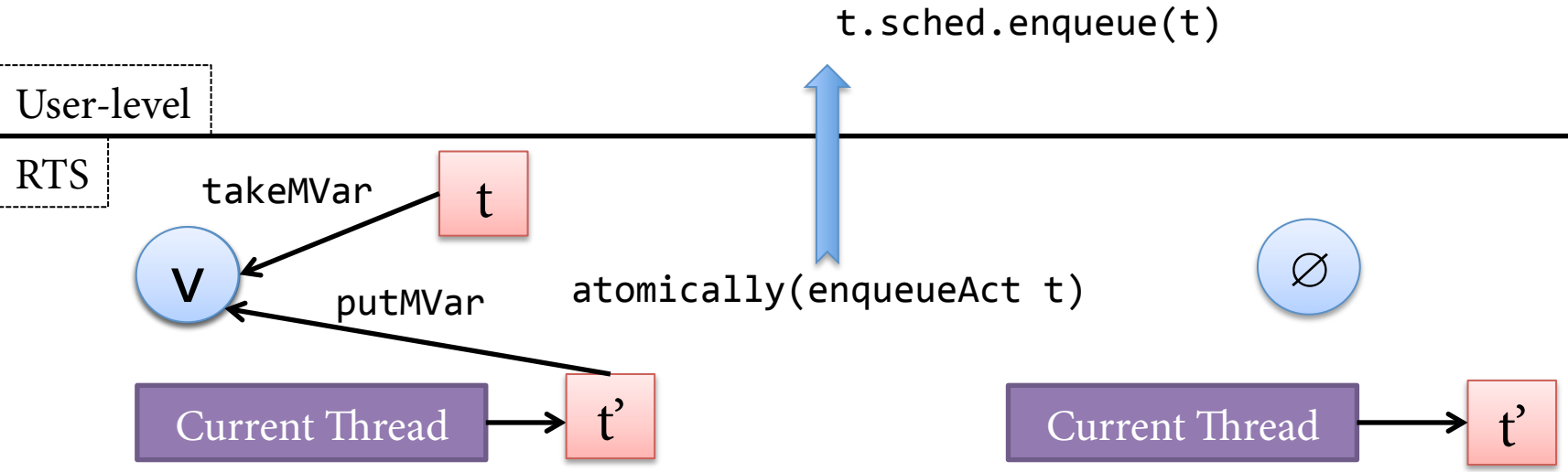
RTS Interaction - Unblocking



RTS Interaction - Unblocking



RTS Interaction - Unblocking



Multicore capable, preemptive, round-robin work-sharing scheduler

```
newtype Sched = Sched (Array Int (TVar[SCont]))

dequeueActivation:: Sched -> SCont -> STM SCont
dequeueActivation (Sched pa) _ = do
  cc <- getCurrentHEC -- get current HEC number
  l <- readTVar $ pa!cc
  case l of
    [] -> retry
    x:tl -> do
      writeTVar (pa!cc) tl
      return x

enqueueActivation:: Sched -> SCont -> STM ()
enqueueActivation (Sched pa) sc = do
  dyn <- getAux sc
  let (hec::Int, _::TVar Int) = fromJust $
       fromDynamic dyn
  l <- readTVar $ pa!hec
  writeTVar (pa!hec) $ l++[sc]
```

```
newScheduler :: IO ()
newScheduler = do
  -- Initialise Auxiliary state
  switch $ \s -> do
    counter <- newTVar (0::Int)
    setAux s $ toDyn $ (0::Int,counter)
    return s
  -- Allocate scheduler
  nc <- getNumHECs
  sched <- (Sched . listArray (0,nc-1)) <$>
           replicateM n (newTVar [])
  -- Initialise activations
  setDequeueAct s $ dequeueActivation sched
  setEnqueueAct s $ enqueueActivation sched

newHEC :: IO ()
newHEC = do
 -- Initial task
 s <- newSCont $ switch dequeueAct
 -- Run in parallel
 runOnIdleHEC s
```

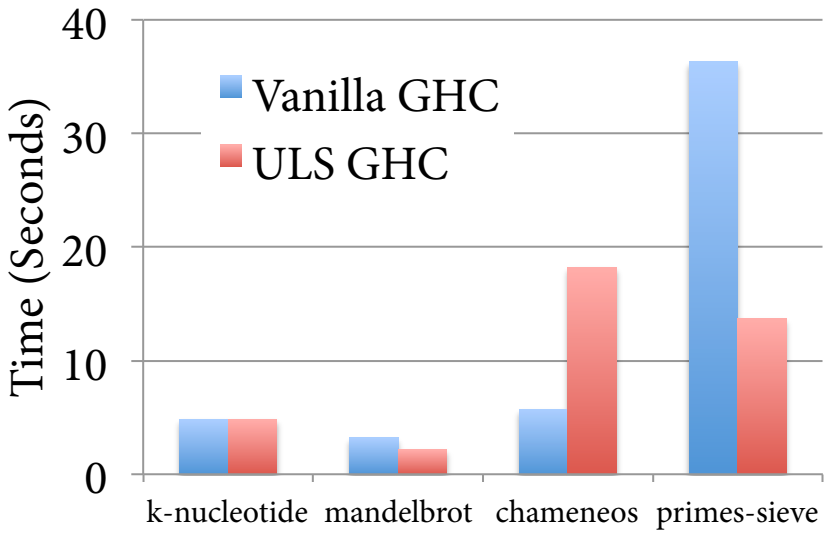
```
forkIO :: IO () -> IO SCont
forkIO task = do
  numHECs <- getNumHECs
  -- epilogue: Switch to next thread
  newSC <- newSCont (task >> switch dequeueAct)
  -- Create and initialise new Aux state
  switch $ \s -> do
    dyn <- getAux s
    let (_::Int, t::TVar Int) = fromJust $
         fromDynamic dyn
    nextHEC <- readTVar t
    writeTVar t $ (nextHEC + 1) `mod` numHECs
    setAux newSC $ toDyn (nextHEC, t)
    return s
  -- Add new thread to scheduler
  atomically $ enqueueAct newSC
  return newSC

yield :: IO ()
yield = switch (\s -> enqueueAct s >> dequeueAct s)
```

Multicore capable, preemptive, round-robin work-sharing scheduler

```
newtype Sched = Sched (Array Int (TVar[SCont]))

dequeueActivation:: Sched -> SCont -> STM SCont
dequeueActivation (Sched pa) _ = do
  cc <- getCurrentHEC -- get current HEC number
  l <- readTVar $ pa!cc
  case l of
    [] -> retry
    x:tl -> do
      writeTVar (pa!cc) tl
      return x

enqueueActivation:: Sched -> SCont -> STM ()
enqueueActivation (Sched pa) sc = do
  dyn <- getAux sc
  let (hec::Int, _::TVar Int) = fromJust $
       fromDynamic dyn
  l <- readTVar $ pa!hec
  writeTVar (pa!hec) $ l++[sc]
```

```
newScheduler :: IO ()
newScheduler = do
  -- Initialise Auxiliary state
  switch $ \s -> do
    counter <- newTVar (0::Int)
    setAux s $ toDyn $ (0::Int,counter)
    return s
  -- Allocate scheduler
  nc <- getNumHECs
  sched <- (Sched . listArray (0,nc-1)) <$>
           replicateM n (newTVar [])
  -- Initialise activations
  setDequeueAct s $ dequeueActivation sched
  setEnqueueAct s $ enqueueActivation sched

newHEC :: IO ()
newHEC = do
 -- Initial task
 s <- newSCont $ switch dequeueAct
 -- Run in parallel
 runOnIdleHEC s
```

```
forkIO :: IO () -> IO SCont
forkIO task = do
  numHECs <- getNumHECs
  -- epilogue: Switch to next thread
  newSC <- newSCont (task >> switch dequeueAct)
  -- Create and initialise new Aux state
  switch $ \s -> do
    dyn <- getAux s
    let (_::Int, t::TVar Int) = fromJust $
         fromDynamic dyn
    nextHEC <- readTVar t
    writeTVar t $ (nextHEC + 1) `mod` numHECs
    setAux newSC $ toDyn (nextHEC, t)
    return s
  -- Add new thread to scheduler
  atomically $ enqueueAct newSC
  return newSC

yield :: IO ()
yield = switch (\s -> enqueueAct s >> dequeueAct s)
```

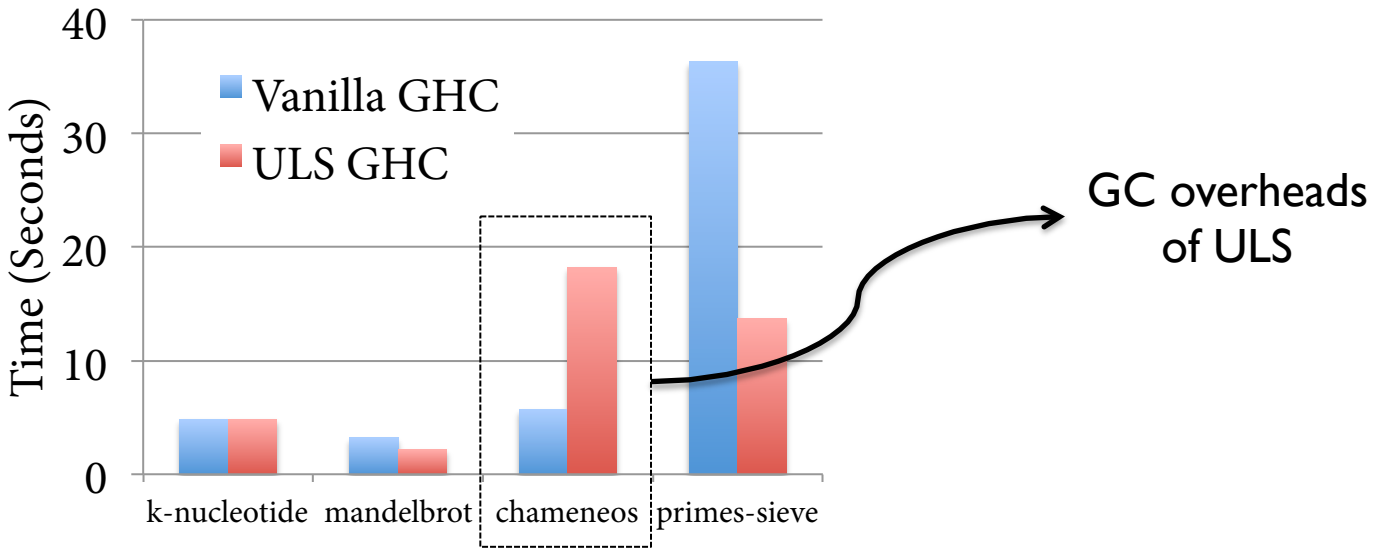


Multicore capable, preemptive, round-robin work-sharing scheduler

```
newtype Sched = Sched (Array Int (TVar[SCont]))

dequeueActivation:: Sched -> SCont -> STM SCont
dequeueActivation (Sched pa) _ = do
  cc <- getCurrentHEC -- get current HEC number
  l <- readTVar $ pa!cc
  case l of
    [] -> retry
    x:tl -> do
      writeTVar (pa!cc) tl
      return x

enqueueActivation:: Sched -> SCont -> STM ()
enqueueActivation (Sched pa) sc = do
  dyn <- getAux sc
  let (hec::Int, _::TVar Int) = fromJust $
       fromDynamic dyn
  l <- readTVar $ pa!hec
  writeTVar (pa!hec) $ l++[sc]
```

```
newScheduler :: IO ()
newScheduler = do
  -- Initialise Auxiliary state
  switch $ \s -> do
    counter <- newTVar (0::Int)
    setAux s $ toDyn $ (0::Int,counter)
    return s
  -- Allocate scheduler
  nc <- getNumHECs
  sched <- (Sched . listArray (0,nc-1)) <$>
           replicateM n (newTVar [])
  -- Initialise activations
  setDequeueAct s $ dequeueActivation sched
  setEnqueueAct s $ enqueueActivation sched

newHEC :: IO ()
newHEC = do
 -- Initial task
 s <- newSCont $ switch dequeueAct
 -- Run in parallel
 runOnIdleHEC s
```

```
forkIO :: IO () -> IO SCont
forkIO task = do
  numHECs <- getNumHECs
  -- epilogue: Switch to next thread
  newSC <- newSCont (task >> switch dequeueAct)
  -- Create and initialise new Aux state
  switch $ \s -> do
    dyn <- getAux s
    let (_::Int, t::TVar Int) = fromJust $
         fromDynamic dyn
    nextHEC <- readTVar t
    writeTVar t $ (nextHEC + 1) `mod` numHECs
    setAux newSC $ toDyn (nextHEC, t)
    return s
  -- Add new thread to scheduler
  atomically $ enqueueAct newSC
  return newSC

yield :: IO ()
yield = switch (\s -> enqueueAct s >> dequeueAct s)
```

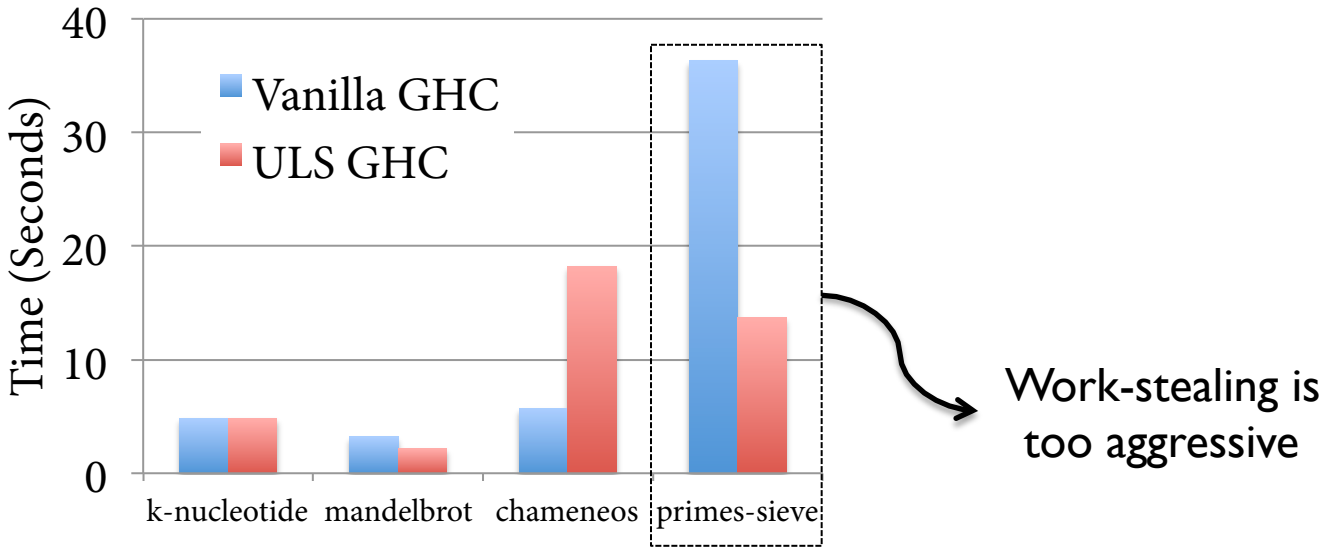


Multicore capable, preemptive, round-robin work-sharing scheduler

```
newtype Sched = Sched (Array Int (TVar[SCont]))

dequeueActivation:: Sched -> SCont -> STM SCont
dequeueActivation (Sched pa) _ = do
  cc <- getCurrentHEC -- get current HEC number
  l <- readTVar $ pa!cc
  case l of
    [] -> retry
    x:tl -> do
      writeTVar (pa!cc) tl
      return x

enqueueActivation:: Sched -> SCont -> STM ()
enqueueActivation (Sched pa) sc = do
  dyn <- getAux sc
  let (hec::Int, _::TVar Int) = fromJust $
       fromDynamic dyn
  l <- readTVar $ pa!hec
  writeTVar (pa!hec) $ l++[sc]
```

```
newScheduler :: IO ()
newScheduler = do
  -- Initialise Auxiliary state
  switch $ \s -> do
    counter <- newTVar (0::Int)
    setAux s $ toDyn $ (0::Int,counter)
    return s
  -- Allocate scheduler
  nc <- getNumHECs
  sched <- (Sched . listArray (0,nc-1)) <$>
           replicateM n (newTVar [])
  -- Initialise activations
  setDequeueAct s $ dequeueActivation sched
  setEnqueueAct s $ enqueueActivation sched

newHEC :: IO ()
newHEC = do
 -- Initial task
 s <- newSCont $ switch dequeueAct
 -- Run in parallel
 runOnIdleHEC s
```

```
forkIO :: IO () -> IO SCont
forkIO task = do
  numHECs <- getNumHECs
  -- epilogue: Switch to next thread
  newSC <- newSCont (task >> switch dequeueAct)
  -- Create and initialise new Aux state
  switch $ \s -> do
    dyn <- getAux s
    let (_::Int, t::TVar Int) = fromJust $
         fromDynamic dyn
    nextHEC <- readTVar t
    writeTVar t $ (nextHEC + 1) `mod` numHECs
    setAux newSC $ toDyn (nextHEC, t)
    return s
  -- Add new thread to scheduler
  atomically $ enqueueAct newSC
  return newSC

yield :: IO ()
yield = switch (\s -> enqueueAct s >> dequeueAct s)
```



Formalization

Concurrency Substrate

$$x, y \in Variable \quad r, s, \in Name$$

$$
\begin{array}{lcl}
Md & ::= & \mathsf{return}\ M \mid M \gg\!= N \\
Ex & ::= & \mathsf{throw}\ M \mid \mathsf{catch}\ M\ N \mid \mathsf{catchSTM}\ M\ N \\
Stm & ::= & \mathsf{newTVar}\ M \mid \mathsf{readTVar}\ r \mid \mathsf{writeTVar}\ r\ M \\
 & \mid & \mathsf{atomically}\ M \mid \mathsf{retry} \\
Sc & ::= & \mathsf{newSCont}\ M \mid \mathsf{switch}\ M \mid \mathsf{runOnIdleHEC}\ s \\
Sls & ::= & \mathsf{getAux}\ s \mid \mathsf{setAux}\ s\ M \\
Act & ::= & \mathsf{dequeueAct}\ s \mid \mathsf{enqueueAct}\ s \\
 & \mid & \mathsf{setDequeueAct}\ M \mid \mathsf{setEnqueueAct}\ M
\end{array}
$$

Term

$$
\begin{array}{lcl}
M, N & ::= & r \mid x \mid \lambda x \mathrel{->} M \mid M\ N \mid \ldots \\
 & \mid & Md \mid Ex \mid Stm \mid Sc \mid Sls \mid Act
\end{array}
$$

$$
\begin{array}{rcll}
\text{Program state} & P & ::= & S; \Theta \\
\text{HEC soup} & S & ::= & \emptyset \mid H \parallel S \\
\text{HEC} & H & ::= & \langle s,M,D \rangle \mid \langle s,M,D \rangle_{Sleeping} \\
 & & \mid & \langle s,M,D \rangle_{Outcall} \mid \mathtt{Idle} \\
\text{Heap} & \Theta & ::= & r \mapsto M \oplus s \mapsto (M, D) \\
\text{SLS Store} & D & ::= & (M, N, r) \\
\text{IO Context} & \mathbb{E} & ::= & \bullet \mid \mathbb{E} \gg\!= M \mid \mathsf{catch}\ \mathbb{E}\ M \\
\text{STM Context} & \mathbb{P} & ::= & \bullet \mid \mathbb{P} \gg\!= M
\end{array}
$$

Upcalls from the RTS

Dequeue upcall instantiation $\quad H; \Theta \overset{deq}{\hookrightarrow} H'; \Theta'$

(UpDequeue)

$$\frac{s'\ \mathsf{fresh} \quad r\ \mathsf{fresh} \quad D' = (\mathsf{deq}(D), \mathsf{enq}(D), r) \quad M' = \mathsf{switch}\ (\lambda x.\, \mathsf{deq}(D)\ s) \quad \Theta' = \Theta[s \mapsto (M, D)][r \mapsto \mathsf{toDyn}\ ()]}{\langle s, M, D \rangle; \Theta \overset{deq}{\hookrightarrow} \langle s', M', D' \rangle; \Theta'}$$

Enqueue upcall instantiation $\quad H; \Theta \overset{enq\ s}{\hookrightarrow} H'; \Theta'$

(UpEnqueueIdle)

$$\frac{s'\ \mathsf{fresh} \quad r\ \mathsf{fresh} \quad D' = (\mathsf{deq}(D), \mathsf{enq}(D), r) \quad M' = \mathsf{atomically}\ (\mathsf{enq}(D)\ s) \quad \Theta' = \Theta[s \mapsto (M, D)][r \mapsto \mathsf{toDyn}\ ()]}{\mathtt{Idle}; \Theta[s \mapsto (M, D)] \overset{enq\ s}{\hookrightarrow} \langle s', M', D' \rangle; \Theta'}$$

(UpEnqueueRunning)

$$\frac{M'' = \mathsf{atomically}\ (\mathsf{enq}(D)\ s) \gg M'}{\langle s', M', D' \rangle; \Theta[s \mapsto (M, D)] \overset{enq\ s}{\hookrightarrow} \langle s', M'', D' \rangle; \Theta[s \mapsto (M, D)]}$$

Formalization

Concurrency Substrate

$$x, y \in Variable \quad r, s, \in Name$$

$$
\begin{array}{lcl}
Md & ::= & \mathsf{return}\ M \mid M \gg\!= N \\
Ex & ::= & \mathsf{throw}\ M \mid \mathsf{catch}\ M\ N \mid \mathsf{catchSTM}\ M\ N \\
Stm & ::= & \mathsf{newTVar}\ M \mid \mathsf{readTVar}\ r \mid \mathsf{writeTVar}\ r\ M \\
 & \mid & \mathsf{atomically}\ M \mid \mathsf{retry} \\
Sc & ::= & \mathsf{newSCont}\ M \mid \mathsf{switch}\ M \mid \mathsf{runOnIdleHEC}\ s \\
Sls & ::= & \mathsf{getAux}\ s \mid \mathsf{setAux}\ s\ M \\
Act & ::= & \mathsf{dequeueAct}\ s \mid \mathsf{enqueueAct}\ s \\
 & \mid & \mathsf{setDequeueAct}\ M \mid \mathsf{setEnqueueAct}\ M
\end{array}
$$

Term

$$
\begin{array}{lcl}
M, N & ::= & r \mid x \mid \lambda x \mathrel{->} M \mid M\ N \mid \ldots \\
 & \mid & Md \mid Ex \mid Stm \mid Sc \mid Sls \mid Act
\end{array}
$$

$$
\begin{array}{rrcl}
\text{Program state} & P & ::= & S; \Theta \\
\text{HEC soup} & S & ::= & \emptyset \mid H \parallel S \\
\text{HEC} & H & ::= & \langle s,M,D \rangle \mid \langle s,M,D \rangle_{Sleeping} \\
 & & \mid & \langle s,M,D \rangle_{Outcall} \mid \texttt{Idle} \\
\text{Heap} & \Theta & ::= & r \mapsto M \oplus s \mapsto (M, D) \\
\text{SLS Store} & D & ::= & (M, N, r) \\
\text{IO Context} & \mathbb{E} & ::= & \bullet \mid \mathbb{E} \gg\!= M \mid \mathsf{catch}\ \mathbb{E}\ M \\
\text{STM Context} & \mathbb{P} & ::= & \bullet \mid \mathbb{P} \gg\!= M
\end{array}
$$

Upcalls from the RTS

Dequeue upcall instantiation $H; \Theta \overset{deq}{\hookrightarrow} H'; \Theta'$

(UpDequeue)

$$\frac{s' \text{ fresh} \quad r \text{ fresh} \quad D' = (\mathsf{deq}(D), \mathsf{enq}(D), r) \quad M' = \mathsf{switch}\ (\lambda x.\, \mathsf{deq}(D)\ s) \quad \Theta' = \Theta[s \mapsto (M, D)][r \mapsto \mathsf{toDyn}\ ()]}{\langle s, M, D \rangle; \Theta \overset{deq}{\hookrightarrow} \langle s', M', D' \rangle; \Theta'}$$

Enqueue upcall instantiation $H; \Theta \overset{enq\ s}{\hookrightarrow} H'; \Theta'$

(UpEnqueueIdle)

$$\frac{s' \text{ fresh} \quad r \text{ fresh} \quad D' = (\mathsf{deq}(D), \mathsf{enq}(D), r) \quad M' = \mathsf{atomically}\ (\mathsf{enq}(D)\ s) \quad \Theta' = \Theta[s \mapsto (M, D)][r \mapsto \mathsf{toDyn}\ ()]}{\texttt{Idle}; \Theta[s \mapsto (M, D)] \overset{enq\ s}{\hookrightarrow} \langle s', M', D' \rangle; \Theta'}$$

(UpEnqueueRunning)

$$\frac{M'' = \mathsf{atomically}\ (\mathsf{enq}(D)\ s) \gg M'}{\langle s', M', D' \rangle; \Theta[s \mapsto (M, D)] \overset{enq\ s}{\hookrightarrow} \langle s', M'', D' \rangle; \Theta[s \mapsto (M, D)]}$$

IO

E[M] tick **E[yield >> M]**

Formalization

Concurrency Substrate

$$x, y \in Variable \quad r, s, \in Name$$

$$
\begin{array}{lcl}
Md & ::= & \mathsf{return}\ M \mid M \gg\!= N \\
Ex & ::= & \mathsf{throw}\ M \mid \mathsf{catch}\ M\ N \mid \mathsf{catchSTM}\ M\ N \\
Stm & ::= & \mathsf{newTVar}\ M \mid \mathsf{readTVar}\ r \mid \mathsf{writeTVar}\ r\ M \\
 & \mid & \mathsf{atomically}\ M \mid \mathsf{retry} \\
Sc & ::= & \mathsf{newSCont}\ M \mid \mathsf{switch}\ M \mid \mathsf{runOnIdleHEC}\ s \\
Sls & ::= & \mathsf{getAux}\ s \mid \mathsf{setAux}\ s\ M \\
Act & ::= & \mathsf{dequeueAct}\ s \mid \mathsf{enqueueAct}\ s \\
 & \mid & \mathsf{setDequeueAct}\ M \mid \mathsf{setEnqueueAct}\ M
\end{array}
$$

Term

$$
\begin{array}{lcl}
M, N & ::= & r \mid x \mid \lambda x \text{ -> } M \mid M\ N \mid \ldots \\
 & \mid & Md \mid Ex \mid Stm \mid Sc \mid Sls \mid Act
\end{array}
$$

$$
\begin{array}{rrcl}
\text{Program state} & P & ::= & S; \Theta \\
\text{HEC soup} & S & ::= & \emptyset \mid H \parallel S \\
\text{HEC} & H & ::= & \langle s,M,D \rangle \mid \langle s,M,D \rangle_{Sleeping} \\
 & & \mid & \langle s,M,D \rangle_{Outcall} \mid \mathtt{Idle} \\
\text{Heap} & \Theta & ::= & r \mapsto M \oplus s \mapsto (M, D) \\
\text{SLS Store} & D & ::= & (M, N, r) \\
\text{IO Context} & \mathbb{E} & ::= & \bullet \mid \mathbb{E} \gg\!= M \mid \mathsf{catch}\ \mathbb{E}\ M \\
\text{STM Context} & \mathbb{P} & ::= & \bullet \mid \mathbb{P} \gg\!= M
\end{array}
$$

Upcalls from the RTS

Dequeue upcall instantiation $H; \Theta \overset{deq}{\hookrightarrow} H'; \Theta'$

(UpDequeue)

$$\frac{s' \text{ fresh} \quad r \text{ fresh} \quad D' = (\mathsf{deq}(D), \mathsf{enq}(D), r) \quad M' = \mathsf{switch}\ (\lambda x.\ \mathsf{deq}(D)\ s) \quad \Theta' = \Theta[s \mapsto (M, D)][r \mapsto \mathsf{toDyn}\ ()]}{\langle s, M, D \rangle; \Theta \overset{deq}{\hookrightarrow} \langle s', M', D' \rangle; \Theta'}$$

Enqueue upcall instantiation $H; \Theta \overset{enq\ s}{\hookrightarrow} H'; \Theta'$

(UpEnqueueIdle)

$$\frac{s' \text{ fresh} \quad r \text{ fresh} \quad D' = (\mathsf{deq}(D), \mathsf{enq}(D), r) \quad M' = \mathsf{atomically}\ (\mathsf{enq}(D)\ s) \quad \Theta' = \Theta[s \mapsto (M, D)][r \mapsto \mathsf{toDyn}\ ()]}{\mathtt{Idle}; \Theta[s \mapsto (M, D)] \overset{enq\ s}{\hookrightarrow} \langle s', M', D' \rangle; \Theta'}$$

(UpEnqueueRunning)

$$\frac{M'' = \mathsf{atomically}\ (\mathsf{enq}(D)\ s) \gg M'}{\langle s', M', D' \rangle; \Theta[s \mapsto (M, D)] \overset{enq\ s}{\hookrightarrow} \langle s', M'', D' \rangle; \Theta[s \mapsto (M, D)]}$$

IO

`E[M]` → tick → `E[yield >> M]`

atomically STM

`E[atomically(P[M])]` → tick → Suspend transaction and switch

Formalization

Concurrency Substrate

$$x, y \in Variable \quad r, s, \in Name$$

$$
\begin{array}{lcl}
Md & ::= & \mathsf{return}\ M \mid M \gg\!= N \\
Ex & ::= & \mathsf{throw}\ M \mid \mathsf{catch}\ M\ N \mid \mathsf{catchSTM}\ M\ N \\
Stm & ::= & \mathsf{newTVar}\ M \mid \mathsf{readTVar}\ r \mid \mathsf{writeTVar}\ r\ M \\
 & \mid & \mathsf{atomically}\ M \mid \mathsf{retry} \\
Sc & ::= & \mathsf{newSCont}\ M \mid \mathsf{switch}\ M \mid \mathsf{runOnIdleHEC}\ s \\
Sls & ::= & \mathsf{getAux}\ s \mid \mathsf{setAux}\ s\ M \\
Act & ::= & \mathsf{dequeueAct}\ s \mid \mathsf{enqueueAct}\ s \\
 & \mid & \mathsf{setDequeueAct}\ M \mid \mathsf{setEnqueueAct}\ M
\end{array}
$$

Term

$$
\begin{array}{lcl}
M, N & ::= & r \mid x \mid \lambda x \mathrel{->} M \mid M\ N \mid \ldots \\
 & \mid & Md \mid Ex \mid Stm \mid Sc \mid Sls \mid Act
\end{array}
$$

$$
\begin{array}{rrcl}
\text{Program state} & P & ::= & S; \Theta \\
\text{HEC soup} & S & ::= & \emptyset \mid H \parallel S \\
\text{HEC} & H & ::= & \langle s,M,D \rangle \mid \langle s,M,D \rangle_{Sleeping} \\
 & & \mid & \langle s,M,D \rangle_{Outcall} \mid \mathtt{Idle} \\
\text{Heap} & \Theta & ::= & r \mapsto M \oplus s \mapsto (M, D) \\
\text{SLS Store} & D & ::= & (M, N, r) \\
\text{IO Context} & \mathbb{E} & ::= & \bullet \mid \mathbb{E} \gg\!= M \mid \mathsf{catch}\ \mathbb{E}\ M \\
\text{STM Context} & \mathbb{P} & ::= & \bullet \mid \mathbb{P} \gg\!= M
\end{array}
$$

Upcalls from the RTS

Dequeue upcall instantiation $H; \Theta \overset{deq}{\hookrightarrow} H'; \Theta'$

(UpDequeue)

$$\frac{s' \text{ fresh} \quad r \text{ fresh} \quad D' = (\mathsf{deq}(D), \mathsf{enq}(D), r) \quad M' = \mathsf{switch}\ (\lambda x.\, \mathsf{deq}(D)\ s) \quad \Theta' = \Theta[s \mapsto (M, D)][r \mapsto \mathsf{toDyn}\ ()]}{\langle s, M, D \rangle; \Theta \overset{deq}{\hookrightarrow} \langle s', M', D' \rangle; \Theta'}$$

Enqueue upcall instantiation $H; \Theta \overset{enq\ s}{\hookrightarrow} H'; \Theta'$

(UpEnqueueIdle)

$$\frac{s' \text{ fresh} \quad r \text{ fresh} \quad D' = (\mathsf{deq}(D), \mathsf{enq}(D), r) \quad M' = \mathsf{atomically}\ (\mathsf{enq}(D)\ s) \quad \Theta' = \Theta[s \mapsto (M, D)][r \mapsto \mathsf{toDyn}\ ()]}{\mathtt{Idle}; \Theta[s \mapsto (M, D)] \overset{enq\ s}{\hookrightarrow} \langle s', M', D' \rangle; \Theta'}$$

(UpEnqueueRunning)

$$\frac{M'' = \mathsf{atomically}\ (\mathsf{enq}(D)\ s) \gg M'}{\langle s', M', D' \rangle; \Theta[s \mapsto (M, D)] \overset{enq\ s}{\hookrightarrow} \langle s', M'', D' \rangle; \Theta[s \mapsto (M, D)]}$$

IO

atomically STM

Switch STM

`E[M]` → tick → `E[yield >> M]`

`E[atomically(P[M])]` → tick → Suspend transaction and switch

`E[switch(P[M])]` → tick

Formalization

Concurrency Substrate

$$x, y \in Variable \quad r, s, \in Name$$

$$
\begin{array}{lcl}
Md & ::= & \mathsf{return}\ M \mid M \gg\!= N \\
Ex & ::= & \mathsf{throw}\ M \mid \mathsf{catch}\ M\ N \mid \mathsf{catchSTM}\ M\ N \\
Stm & ::= & \mathsf{newTVar}\ M \mid \mathsf{readTVar}\ r \mid \mathsf{writeTVar}\ r\ M \\
 & \mid & \mathsf{atomically}\ M \mid \mathsf{retry} \\
Sc & ::= & \mathsf{newSCont}\ M \mid \mathsf{switch}\ M \mid \mathsf{runOnIdleHEC}\ s \\
Sls & ::= & \mathsf{getAux}\ s \mid \mathsf{setAux}\ s\ M \\
Act & ::= & \mathsf{dequeueAct}\ s \mid \mathsf{enqueueAct}\ s \\
 & \mid & \mathsf{setDequeueAct}\ M \mid \mathsf{setEnqueueAct}\ M
\end{array}
$$

Term

$$
\begin{array}{lcl}
M, N & ::= & r \mid x \mid \lambda x \mathrel{->} M \mid M\ N \mid \ldots \\
 & \mid & Md \mid Ex \mid Stm \mid Sc \mid Sls \mid Act
\end{array}
$$

$$
\begin{array}{rrcl}
\text{Program state} & P & ::= & S; \Theta \\
\text{HEC soup} & S & ::= & \emptyset \mid H \parallel S \\
\text{HEC} & H & ::= & \langle s,M,D \rangle \mid \langle s,M,D \rangle_{Sleeping} \\
 & & \mid & \langle s,M,D \rangle_{Outcall} \mid \mathtt{Idle} \\
\text{Heap} & \Theta & ::= & r \mapsto M \oplus s \mapsto (M, D) \\
\text{SLS Store} & D & ::= & (M, N, r) \\
\text{IO Context} & \mathbb{E} & ::= & \bullet \mid \mathbb{E} \gg\!= M \mid \mathsf{catch}\ \mathbb{E}\ M \\
\text{STM Context} & \mathbb{P} & ::= & \bullet \mid \mathbb{P} \gg\!= M
\end{array}
$$

Upcalls from the RTS

Dequeue upcall instantiation $H; \Theta \overset{deq}{\hookrightarrow} H'; \Theta'$

(UpDequeue)

$$\frac{s' \text{ fresh} \quad r \text{ fresh} \quad D' = (\mathsf{deq}(D), \mathsf{enq}(D), r) \quad M' = \mathsf{switch}\ (\lambda x.\, \mathsf{deq}(D)\ s) \quad \Theta' = \Theta[s \mapsto (M, D)][r \mapsto \mathsf{toDyn}\ ()]}{\langle s, M, D \rangle; \Theta \overset{deq}{\hookrightarrow} \langle s', M', D' \rangle; \Theta'}$$

Enqueue upcall instantiation $H; \Theta \overset{enq\ s}{\hookrightarrow} H'; \Theta'$

(UpEnqueueIdle)

$$\frac{s' \text{ fresh} \quad r \text{ fresh} \quad D' = (\mathsf{deq}(D), \mathsf{enq}(D), r) \quad M' = \mathsf{atomically}\ (\mathsf{enq}(D)\ s) \quad \Theta' = \Theta[s \mapsto (M, D)][r \mapsto \mathsf{toDyn}\ ()]}{\mathtt{Idle}; \Theta[s \mapsto (M, D)] \overset{enq\ s}{\hookrightarrow} \langle s', M', D' \rangle; \Theta'}$$

(UpEnqueueRunning)

$$\frac{M'' = \mathsf{atomically}\ (\mathsf{enq}(D)\ s) \gg M'}{\langle s', M', D' \rangle; \Theta[s \mapsto (M, D)] \overset{enq\ s}{\hookrightarrow} \langle s', M'', D' \rangle; \Theta[s \mapsto (M, D)]}$$

IO

`E[M]` → tick → `E[yield >> M]`

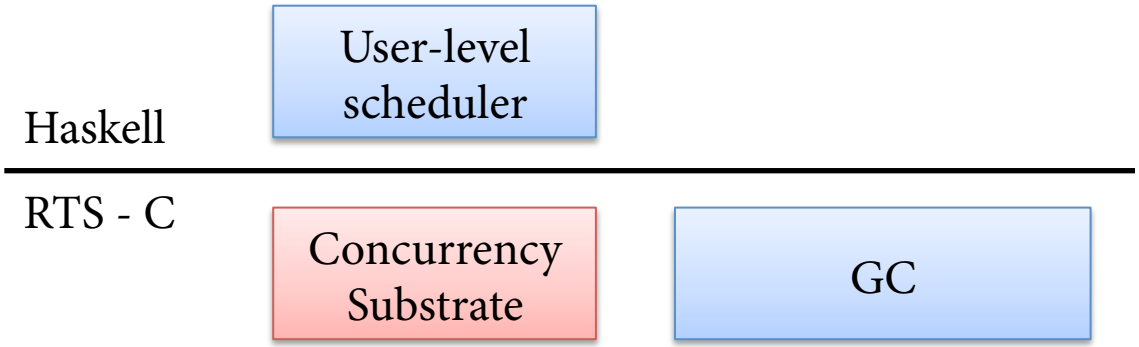
atomically STM

`E[atomically(P[M])]` → tick → Suspend transaction and switch

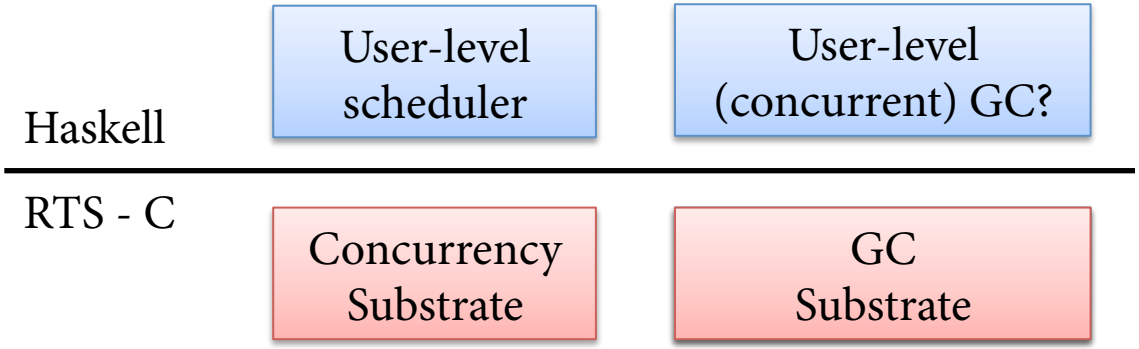
Switch STM

`E[switch(P[M])]` → tick → *Disable Interrupts!*

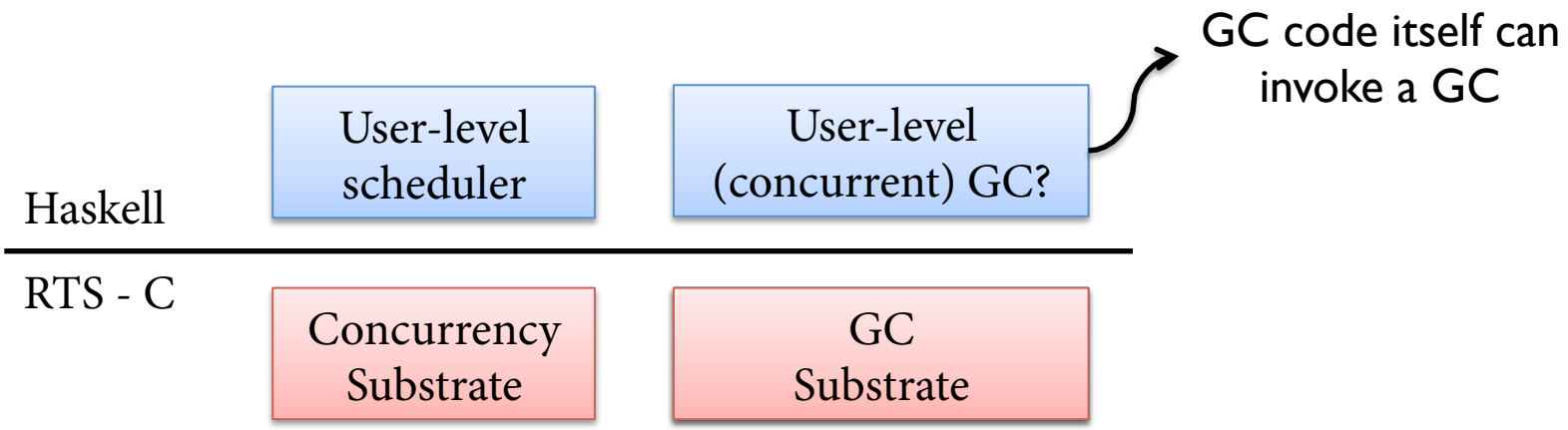
Future directions



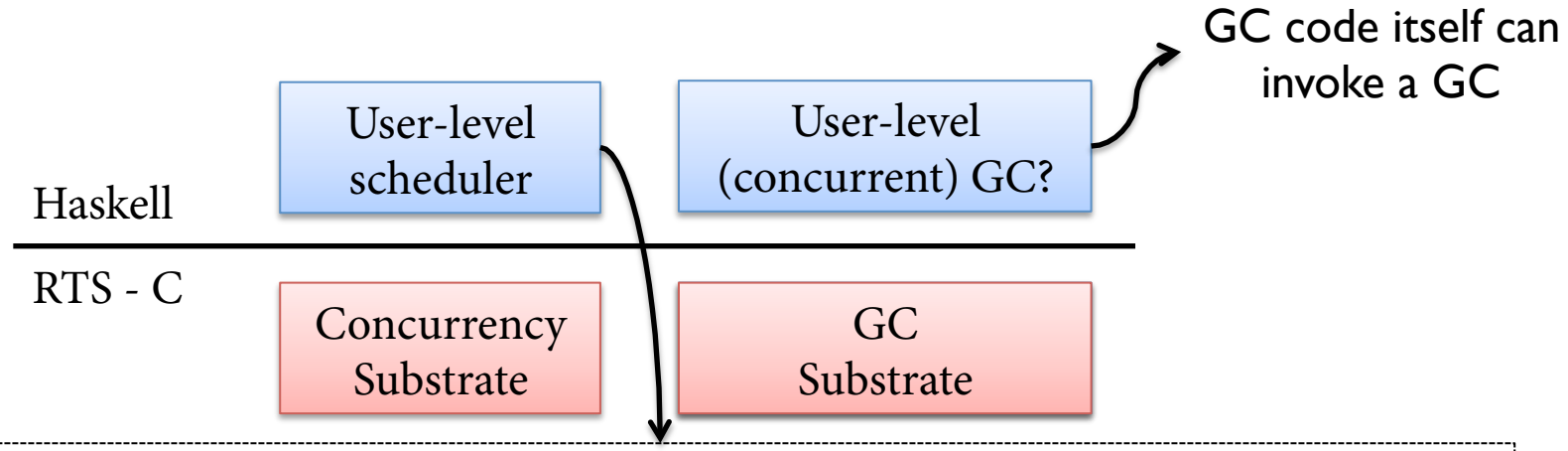
Future directions



Future directions

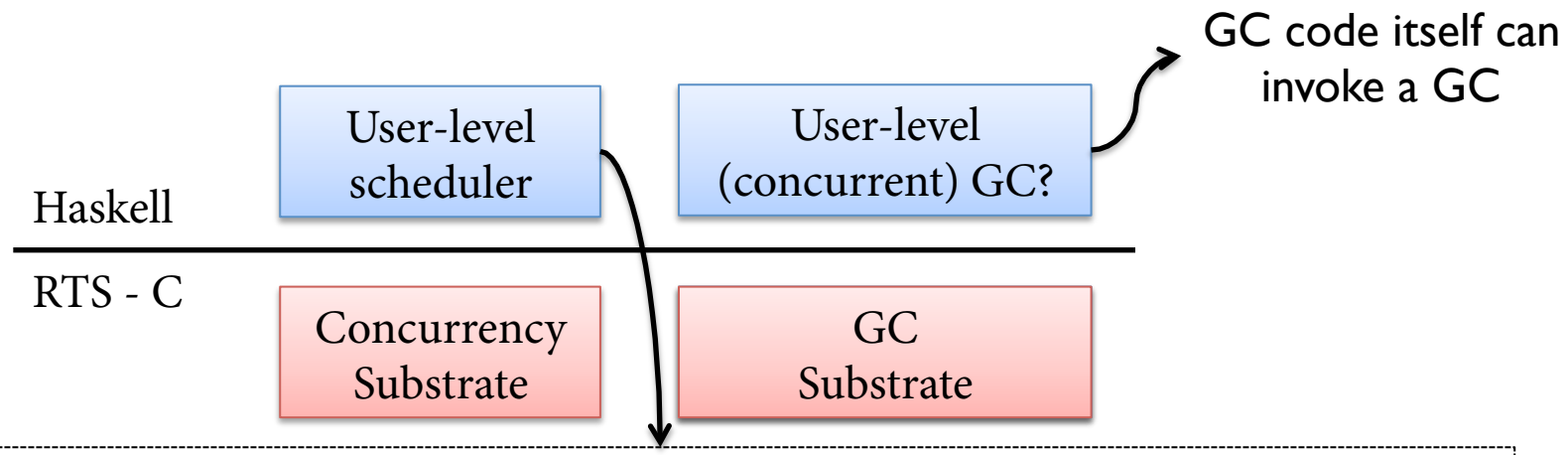


Future directions



- Verifying the functional correctness of schedulers
  - Correct Scheduler ⇒ Each thread runs as if given its own processor and register set
  - Scheduler access under STM ⇒ Treat scheduler as sequential process

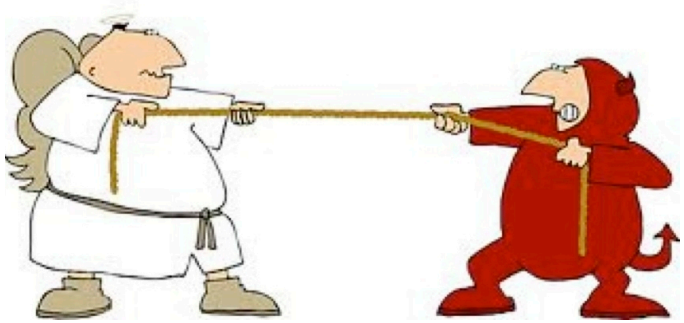
Future directions



- Verifying the functional correctness of schedulers
  - Correct Scheduler ⇒ Each thread runs as if given its own processor and register set
  - Scheduler access under STM ⇒ Treat scheduler as sequential process

- FP abstractions for eventually consistent systems
  - Operations on ECDTs described as pure functions over axiomatic executions
  - A relational specification language for specifying consistency assertions over axiomatic executions

Conclusion



- Functional programming abstractions simplify concurrent programming
  - Rx-CML: Synchronous communication over geo-distributed systems
  - Concurrency Substrate: Scheduler activation + STM for writing schedulers
- Abstractions introduce indirection resulting in overheads
  - Rx-CML slower than explicit async under contention
  - Conc. Subs: Scheduler allocations increase GC overheads