CompCertX:

Verified Separate Compilation for Compositional Verification of Layered Systems

Yale

Ronghui Gu
Jérémie Koenig
Zhong Shao
Newman Wu
Shu-Chun Weng

Reservoir Labs

Tahina Ramananandro

University of Science and Technology of China

Haozhong Zhang
Yu Guo

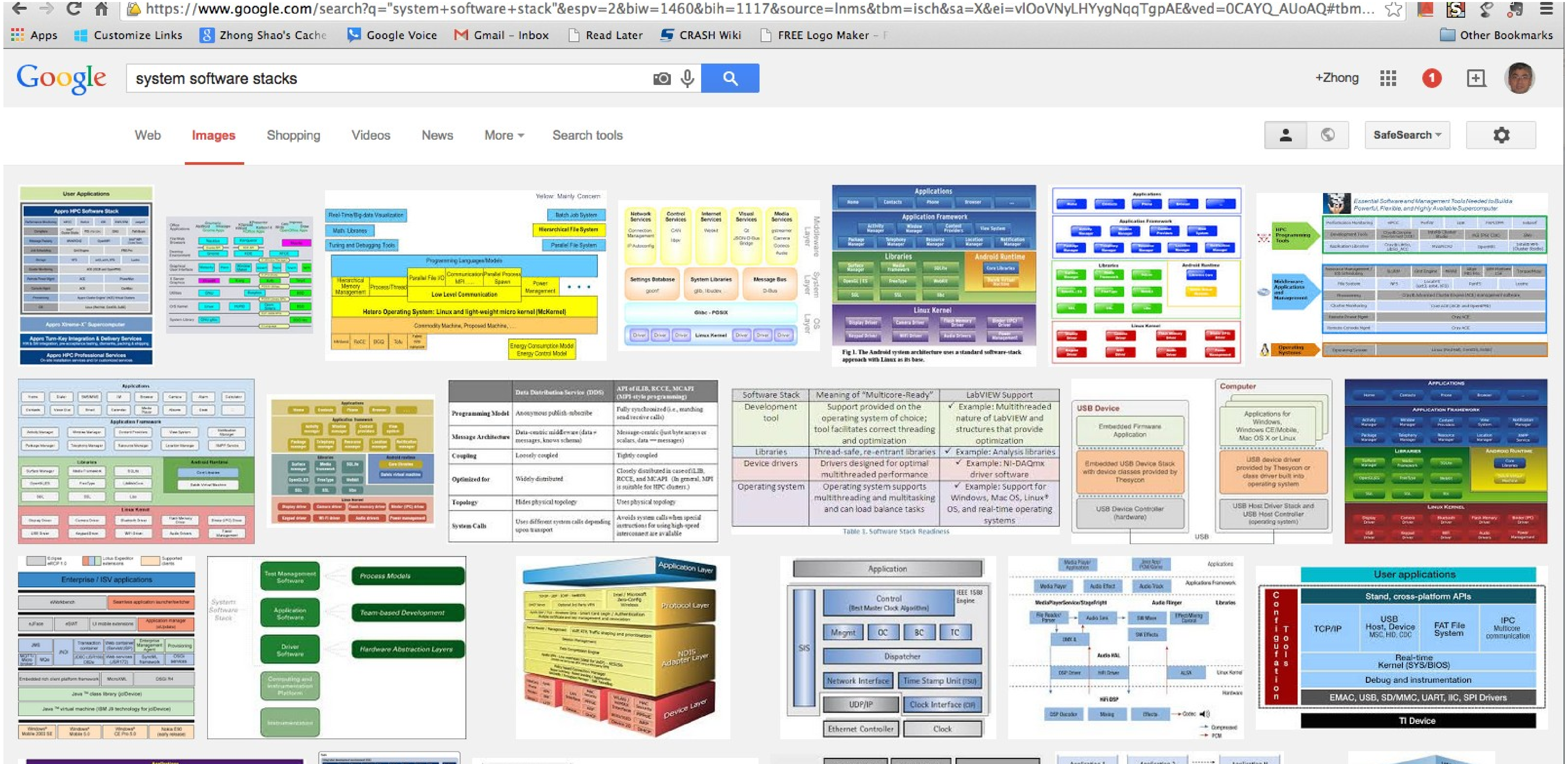
August 24, 2015

Motivation

How to build reliable & secure ***system software stacks****?*

Motivation

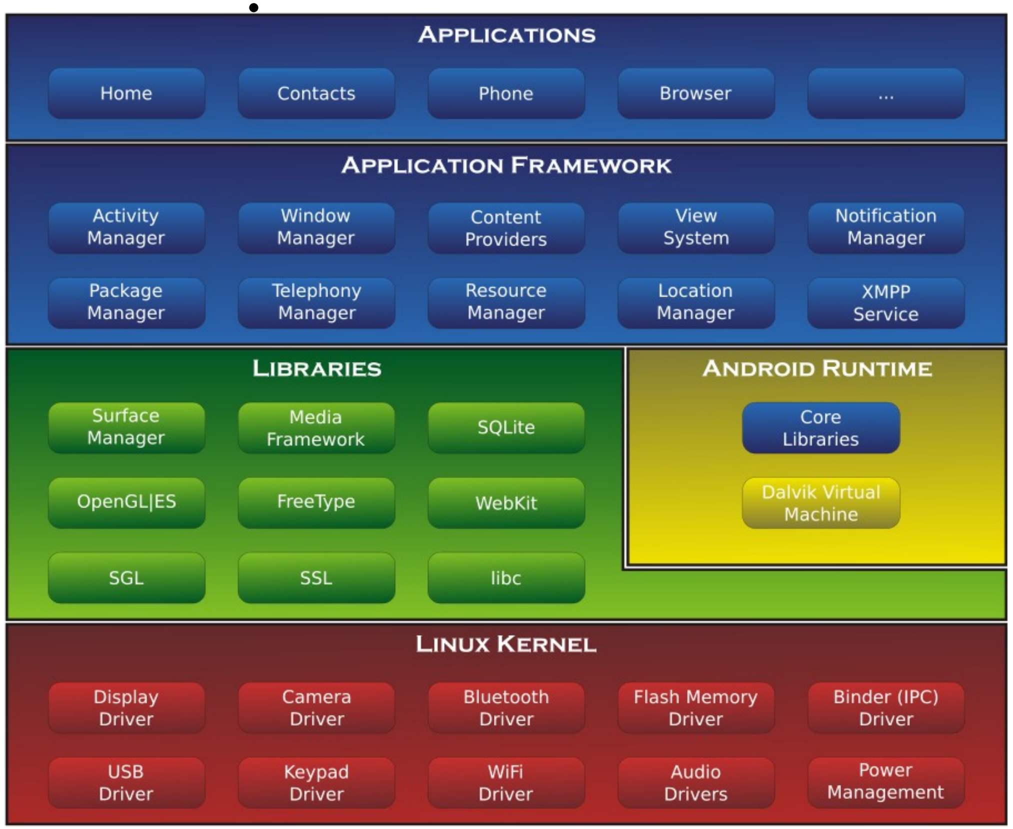
How to build reliable & secure ***system software stacks****?*

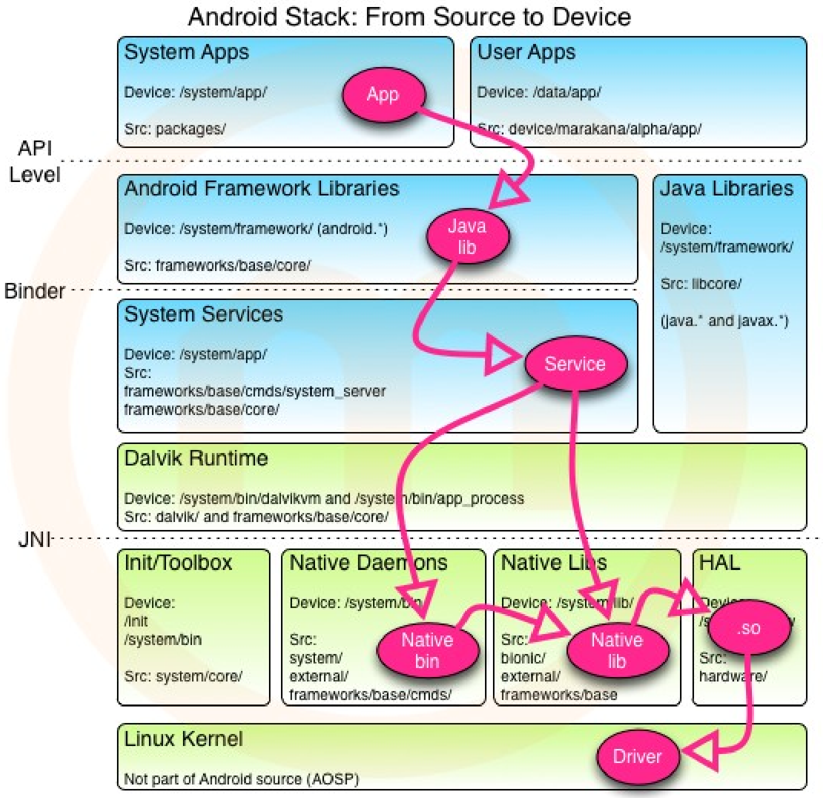


Motivation

Android architecture & system stack

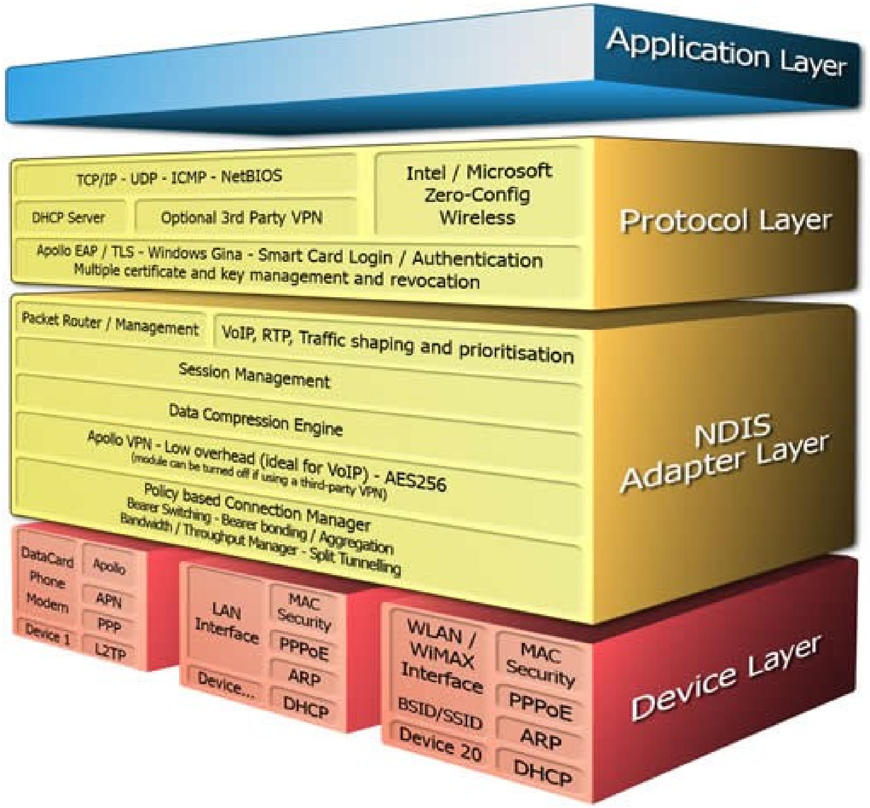
From https://thenewcircle.com/s/post/1031/android_stack_source_to_device &
http://en.wikipedia.org/wiki/Android_(operating_system)





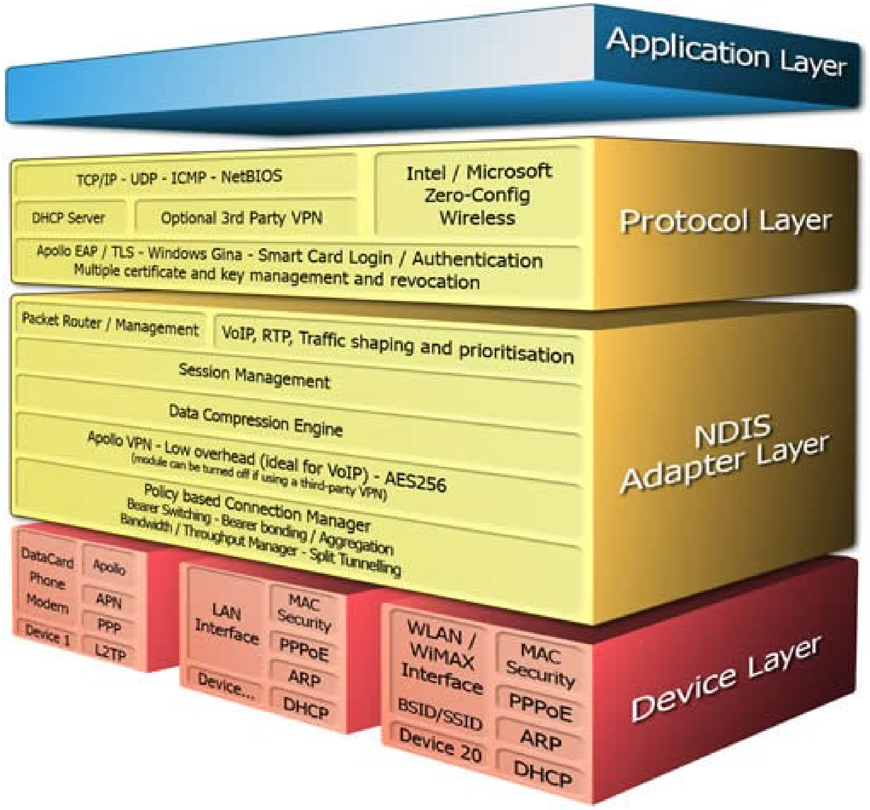
Motivation

Apollo Mobile Communication Stack
http://www.layer2connections.com/apollo_clients.html

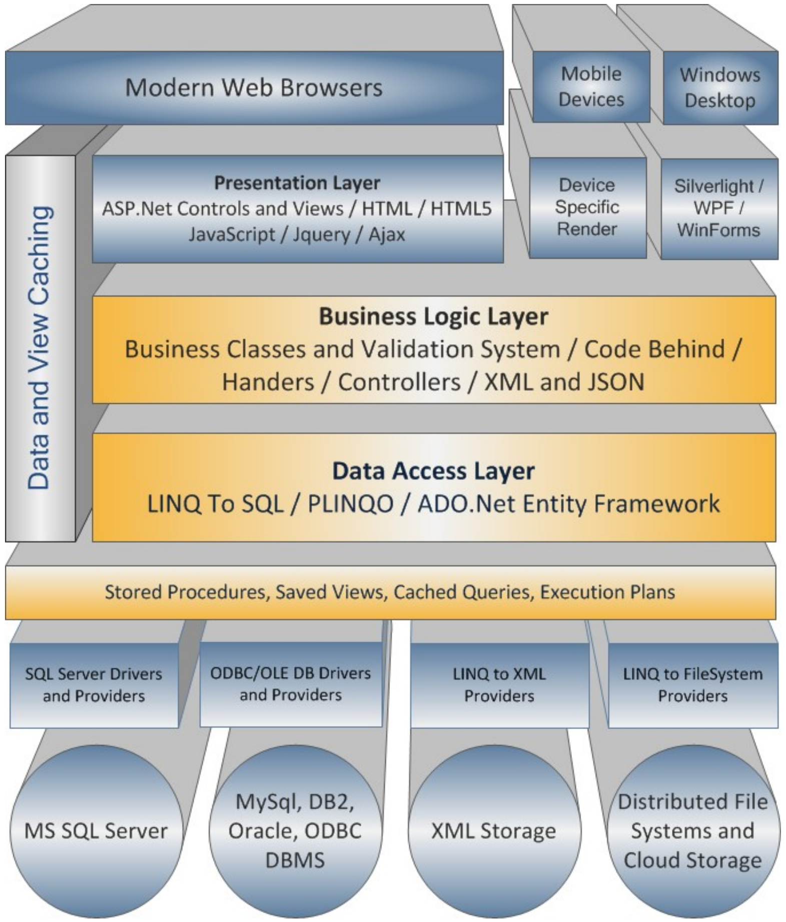


Motivation

Apollo Mobile Communication Stack
http://www.layer2connections.com/apollo_clients.html



Web Application Development Stack
http://www.brightware.co.uk/Technology.aspx



Motivation (cont’d)

- Common themes: all system stacks are built based on abstraction, modularity, and layering
- Abstraction layers are ubiquitous!

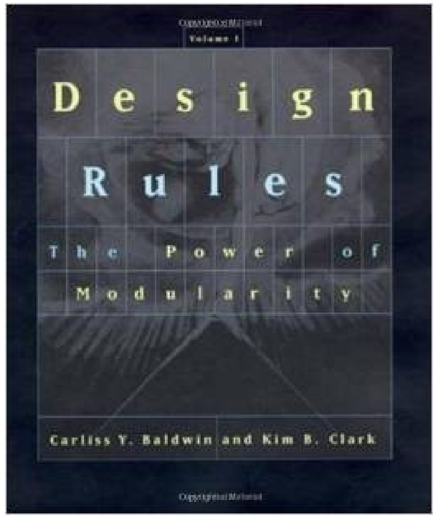
Motivation (cont'd)

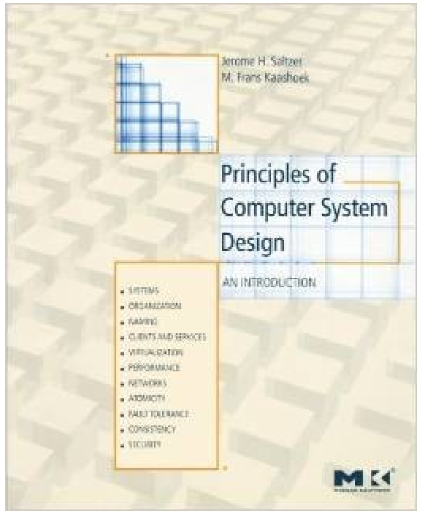
- Common themes: all system stacks are built based on abstraction, modularity, and layering

- Abstraction layers are ubiquitous!

Such use of abstraction, modularity, and layering is "**the key factor that drove the computer industry toward today's explosive levels of innovation and growth** because *complex products can be built from smaller subsystems that can be designed independently yet function together as a whole."*

Baldwin & Clark " Design Rules: Volume 1, The Power of Modularity", MIT Press, 2000





Problems

- What is an ***abstraction layer***?
- How to formally ***specify*** an abstraction layer?
- How to ***program***, ***verify***, and ***compile*** each layer?
- How to ***compose*** abstraction layers?
- How to apply ***certified abstraction layers*** to build *reliable* and *secure* system software?

Our Contributions

- We introduce deep specification and present a language-based formalization of certified abstraction layer

- We developed new languages & tools in Coq
  - A formal layer calculus for composing certified layers
  - ClightX for writing certified layers in a C-like language
  - AsmX for writing certified layers in assembly
  - CompCertX that compiles ClightX layers into AsmX layers

- We built multiple certified OS kernels in Coq
  - mCertiKOS-hyper consists of 37 layers, took less than one-person-year to develop, and can boot Linux as a guest

Our Contributions

- We introduce deep specification and present a language-based formalization of certified abstraction layer

- We developed new languages & tools in Coq
    - A formal layer calculus for composing certified layers
    - ClightX for writing certified layers in a C-like language
    - AsmX for writing certified layers in assembly
    - CompCertX that compiles ClightX layers into AsmX layers

- We built multiple certified OS kernels in Coq
    - mCertiKOS-hyper consists of 37 layers, took less than one-person-year to develop, and can boot Linux as a guest

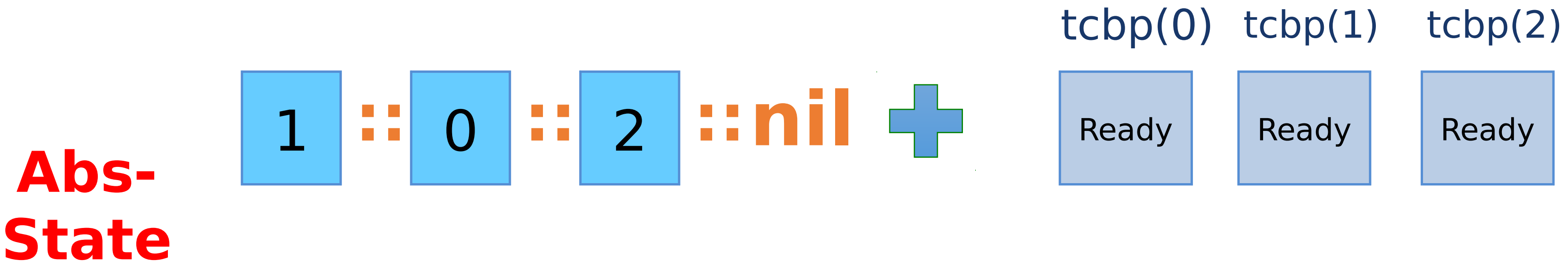
Example: Thread Queues

```
extern unsigned int dequeue(unsigned int);
extern void set_state(unsigned int, unsigned int);
extern void enqueue(unsigned int, unsigned int);

void thread_enqueue(unsigned int proc_index)
{
    set_state(proc_index, TSTATE_READY);
    enqueue(NUM_CHAN, proc_index);
}

void thread_wakeup(unsigned int chan_index)
{
    unsigned int proc_index;
    proc_index = dequeue(chan_index);
    if(proc_index != NUM_PROC)
        thread_enqueue(proc_index);
}
```

Example: Thread Queues

```
extern unsigned int dequeue(unsigned int);
extern void set_state(unsigned int, unsigned int);
extern void enqueue(unsigned int, unsigned int);

void thread_enqueue(unsigned int proc_index)
{
    set_state(proc_index, TSTATE_READY);
    enqueue(NUM_CHAN, proc_index);
}

void thread_wakeup(unsigned int chan_index)
{
    unsigned int proc_index;
    proc_index = dequeue(chan_index);
    if(proc_index != NUM_PROC)
        thread_enqueue(proc_index);
}
```

Layer primitives



Abs-State

1 :: 0 :: 2 ::nil +

tcbp(0)	tcbp(1)	tcbp(2)
Ready	Ready	Ready

Example: Thread Queues

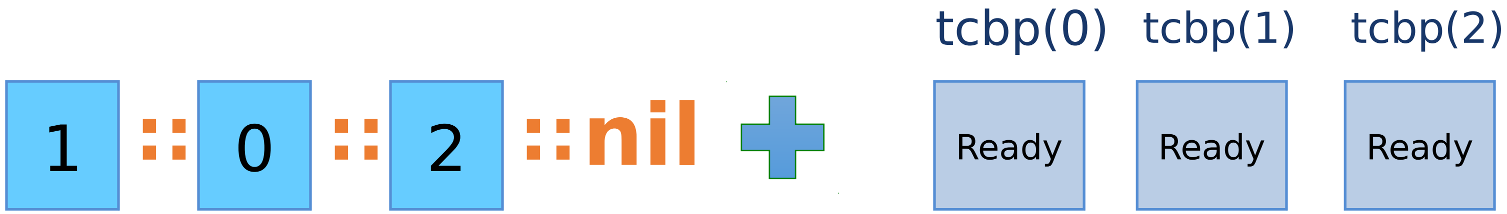
```
extern unsigned int dequeue(unsigned int);
extern void set_state(unsigned int, unsigned int);
extern void enqueue(unsigned int, unsigned int);
```

Layer primitives

```
void thread_enqueue(unsigned int proc_index)
{
    set_state(proc_index, TSTATE_READY);
    enqueue(NUM_CHAN, proc_index);
}

void thread_wakeup(unsigned int chan_index)
{
    unsigned int proc_index;
    proc_index = dequeue(chan_index);
    if(proc_index != NUM_PROC)
        thread_enqueue(proc_index);
}
```

Client code

Abs-State



1 :: 0 :: 2 ::nil +

tcbp(0) tcbp(1) tcbp(2)

Ready Ready Ready

Example: Thread Queues

```
extern unsigned int dequeue(unsigned int);
extern void set_state(unsigned int, unsigned int);
extern void enqueue(unsigned int, unsigned int);
```

Layer primitives

```
void thread_enqueue(unsigned int proc_index)
{
    set_state(proc_index, TSTATE_READY);
    enqueue(NUM_CHAN, proc_index);
}

void thread_wakeup(unsigned int chan_index)
{
    unsigned int proc_index;
    proc_index = dequeue(chan_index);
    if(proc_index != NUM_PROC)
        thread_enqueue(proc_index);
}
```

Client code

Abs-State

1 :: 0 :: 2 ::nil ✚

tcbp(0) tcbp(1) tcbp(2)

Ready Ready Ready

Example: Thread Queues

```
extern unsigned int dequeue(unsigned int);
extern void set_state(unsigned int, unsigned int);
extern void enqueue(unsigned int, unsigned int);
```

Layer primitives

```
void thread_enqueue(unsigned int proc_index)
{
    set_state(proc_index, TSTATE_READY);
    enqueue(NUM_CHAN, proc_index);
}

void thread_wakeup(unsigned int chan_index)
{
    unsigned int proc_index;
    proc_index = dequeue(chan_index);
    if(proc_index != NUM_PROC)
        thread_enqueue(proc_index);
}
```

Client code

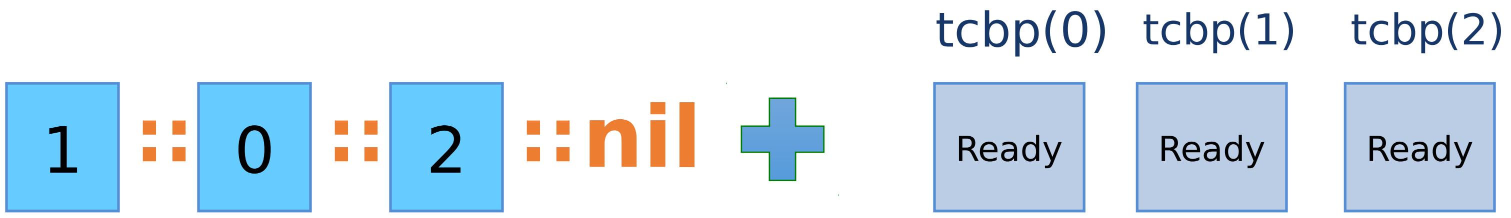
Abs-State

1 :: 0 :: 2 ::nil +

tcbp(0)	tcbp(1)	tcbp(2)
Ready	Ready	Ready

Example: Thread Queues

Abs-State



1 :: 0 :: 2 ::nil +

tcbp(0)	tcbp(1)	tcbp(2)
Ready	Ready	Ready

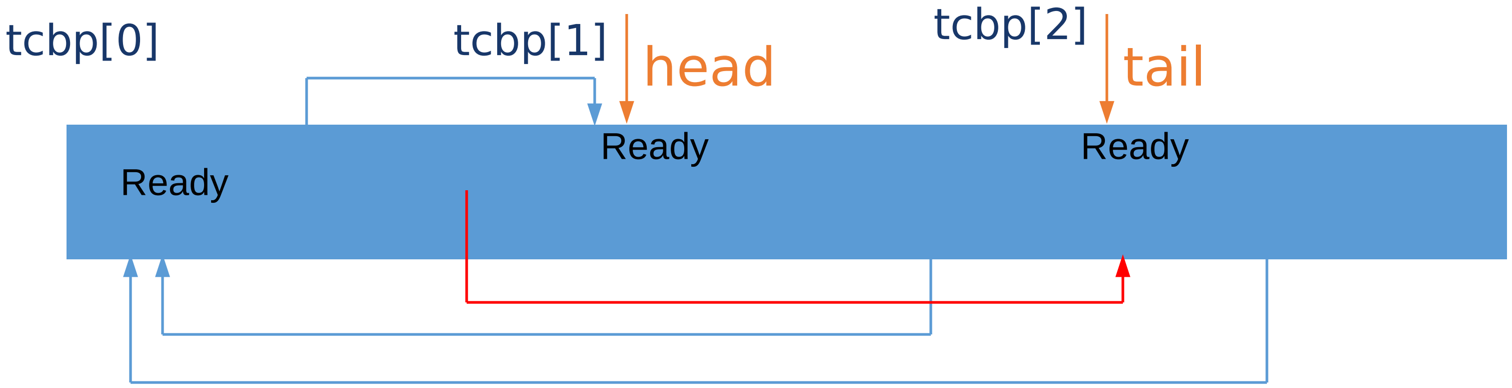
Example: Thread Queues

Abs-State

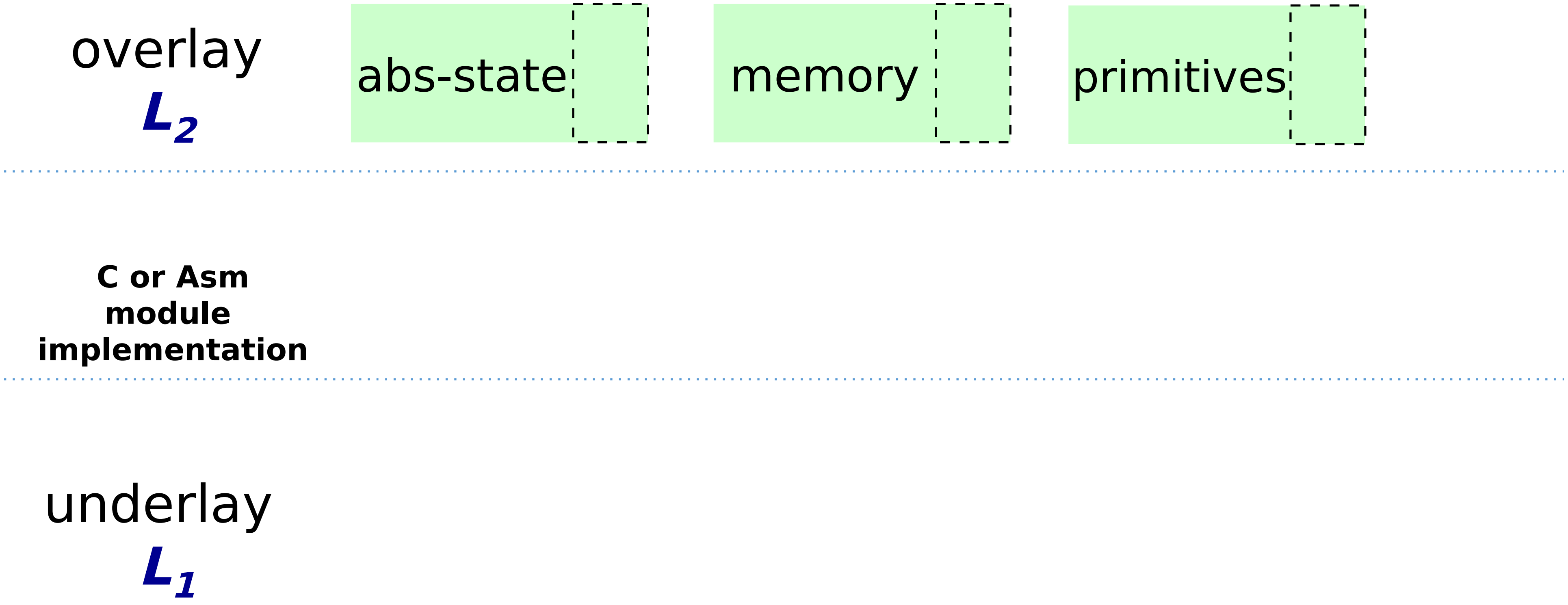
1 :: 0 :: 2 ::nil ✚

tcbp(0)	tcbp(1)	tcbp(2)
Ready	Ready	Ready

Concrete Memory



What is an Abstraction Layer?



overlay
L_2

abs-state

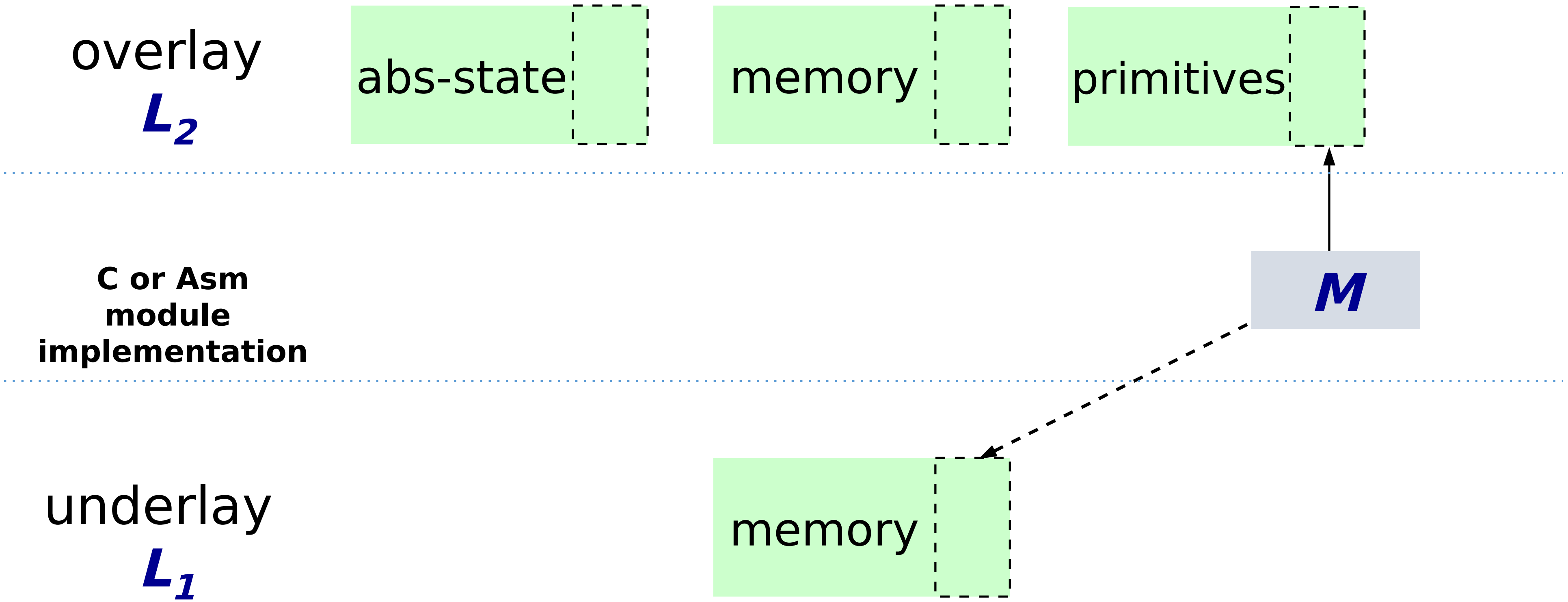
memory

primitives

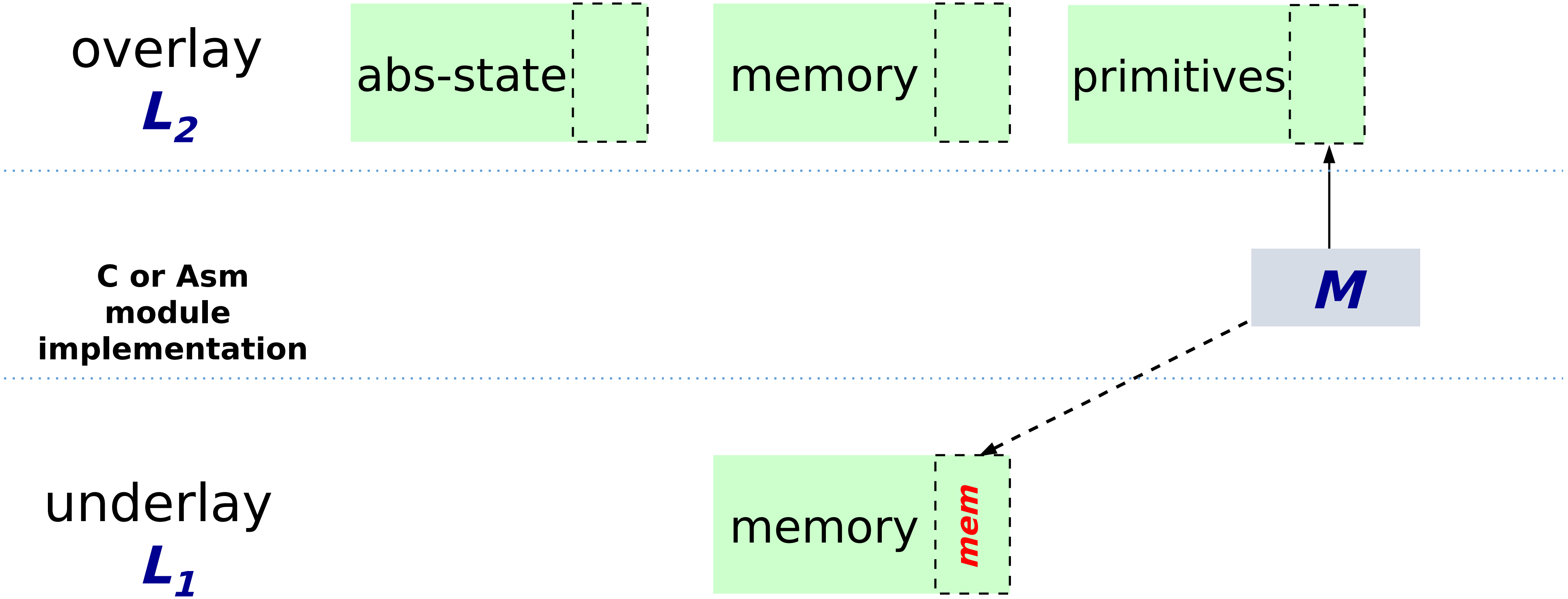
C or Asm module implementation

underlay
L_1

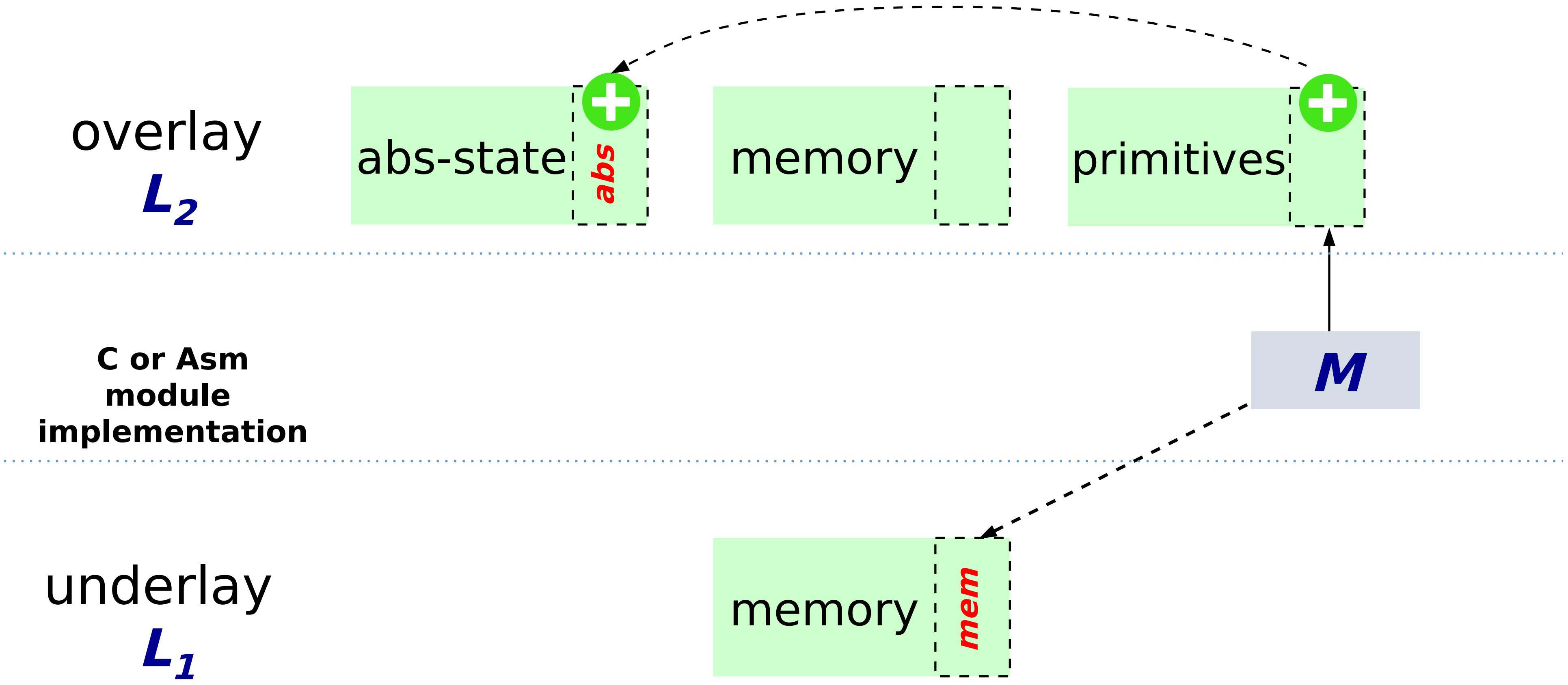
What is an Abstraction Layer?



What is an Abstraction Layer?



What is an Abstraction Layer?



overlay

L_2

abs-state

abs

memory

primitives

C or Asm module implementation

M

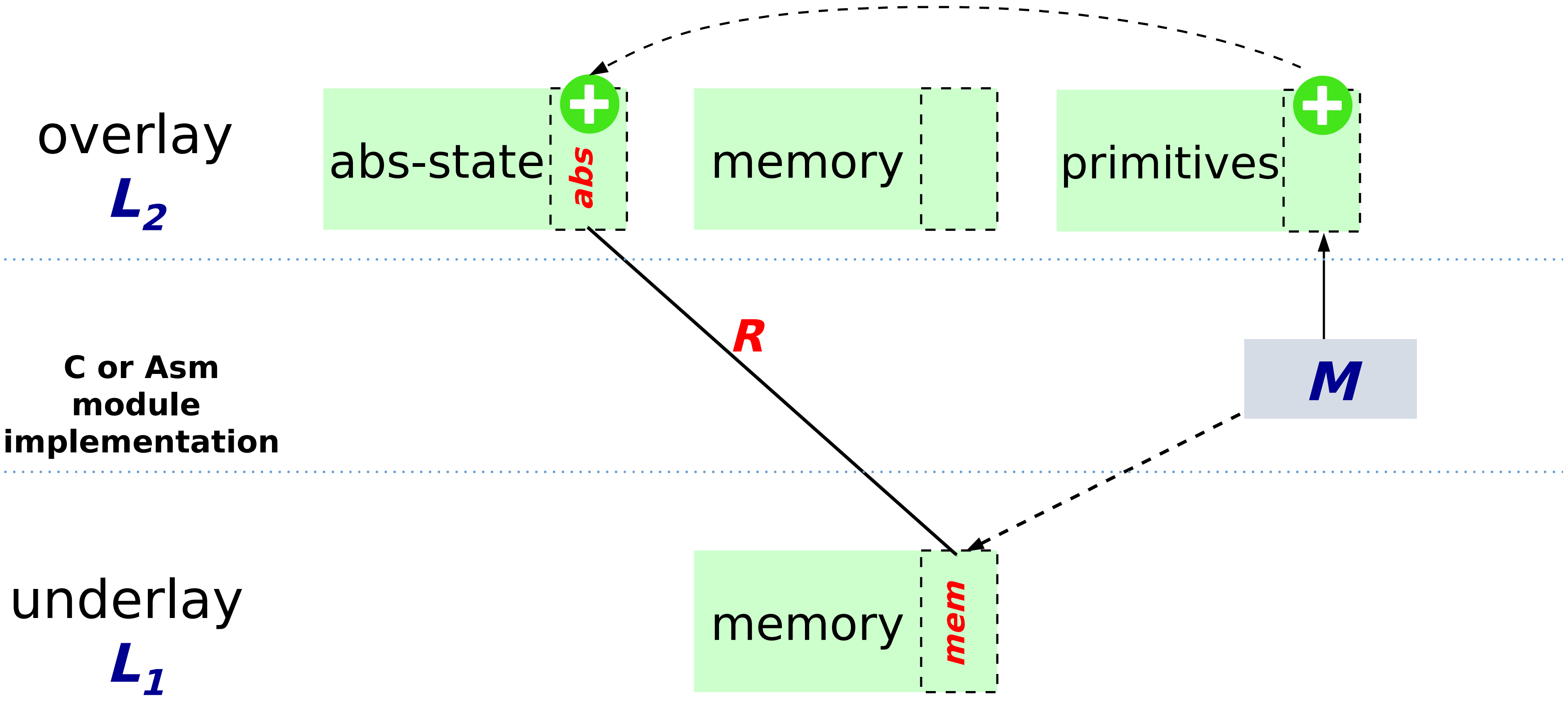
underlay

L_1

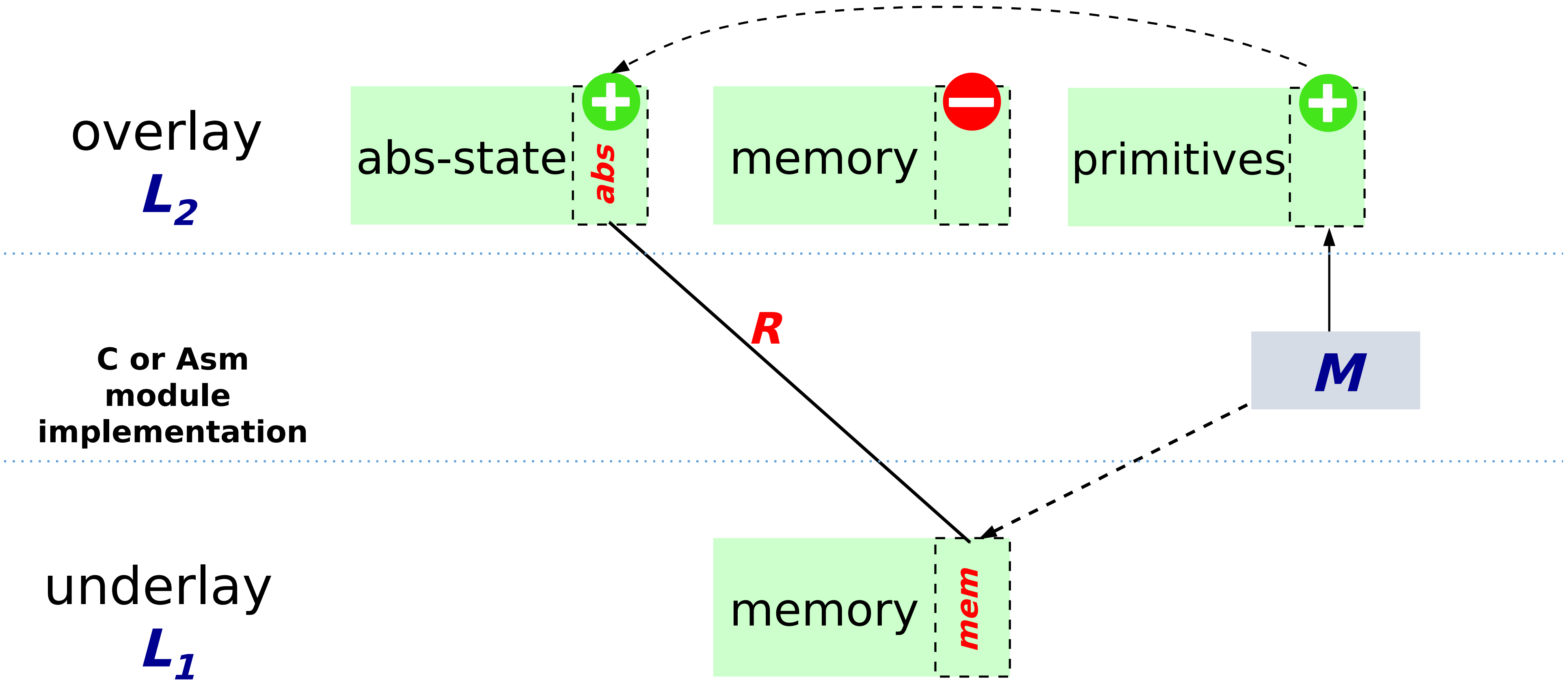
memory

mem

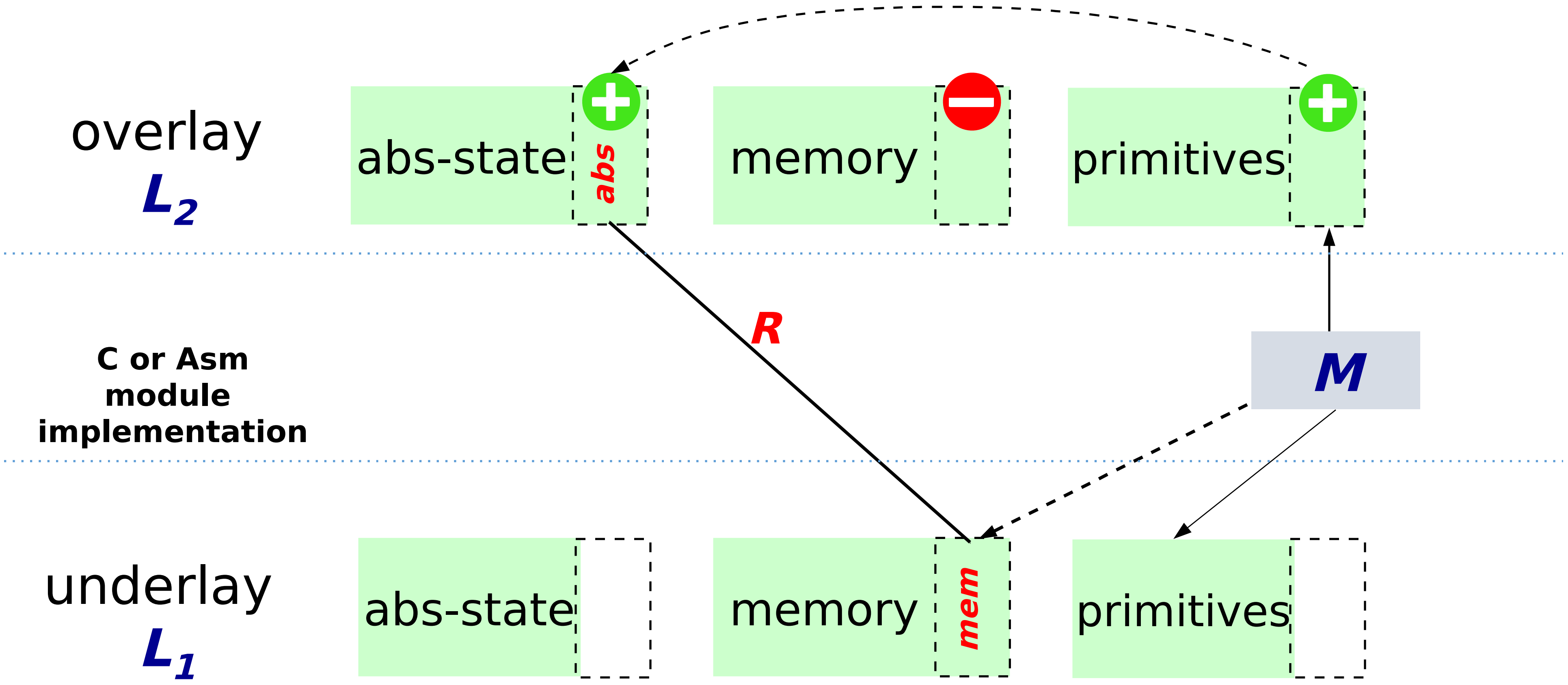
What is an Abstraction Layer?



What is an Abstraction Layer?



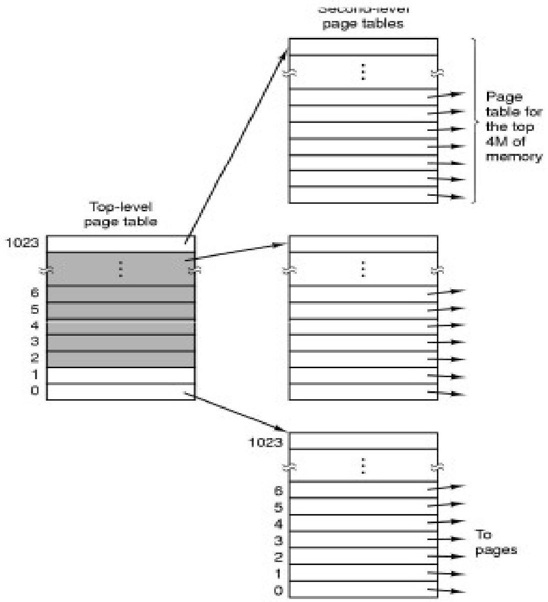
What is an Abstraction Layer?



Example: Page Tables

concrete C types

```
struct PMap {
    char * page_dir[1024];
    uint  page_table[1024][1024];
};
```



abstract Coq spec

```
Inductive PTPerm: Type :=
     | PTP
     | PTU
     | PTK.

Inductive PTEInfo:=
     | PTEValid (v : Z) (p :  PTPerm)
     | PTEUnPresent.

Definition PMap := ZMap.t PTEInfo.
```

Example: Page Tables

abstract layer spec

abstract state ➕

PMap := ZMap.t PTEInfo
(* *vaddr ⇀ (paddr, perm)* *)

Invariants: kernel page table is a direct map; user parts are isolated

abstract primitives ➕
(Coq functions)

```
Definition page_table_init := …
Definition page_table_insert :=…
Definition page_table_rmv := …
Definition page_table_read := …
```

concrete C implementation

memory ➖

```
char * page_dir[1024];

uint page_table[1024][1024];
```

C functions

```
int page_table_init() { … }
int page_table_insert { … }
int page_table_rmv() { … }
int page_table_read() { … }
```

Formalizing Abstraction Layers

What is a ***certified*** abstraction layer

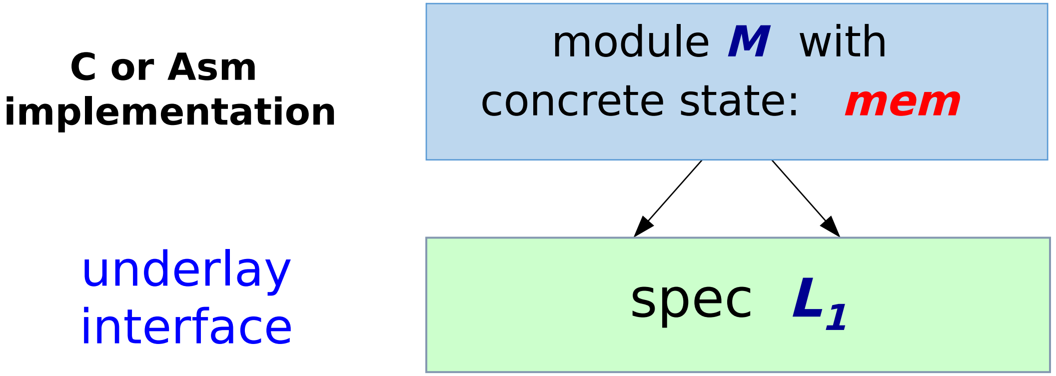
(L_1, M, L_2) ?

C or Asm
implementation

module ***M*** with
concrete state: ***mem***

Formalizing Abstraction Layers

What is a ***certified*** abstraction layer
(L_1, M, L_2) ?



C or Asm implementation

module ***M*** with
concrete state: ***mem***

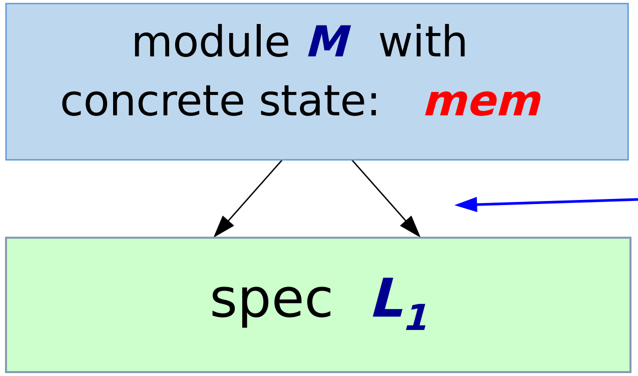
underlay interface

spec ***L_1***

Formalizing Abstraction Layers

What is a ***certified*** abstraction layer

(L_1, M, L_2) ?



C or Asm implementation

module ***M*** with

concrete state: ***mem***

underlay interface

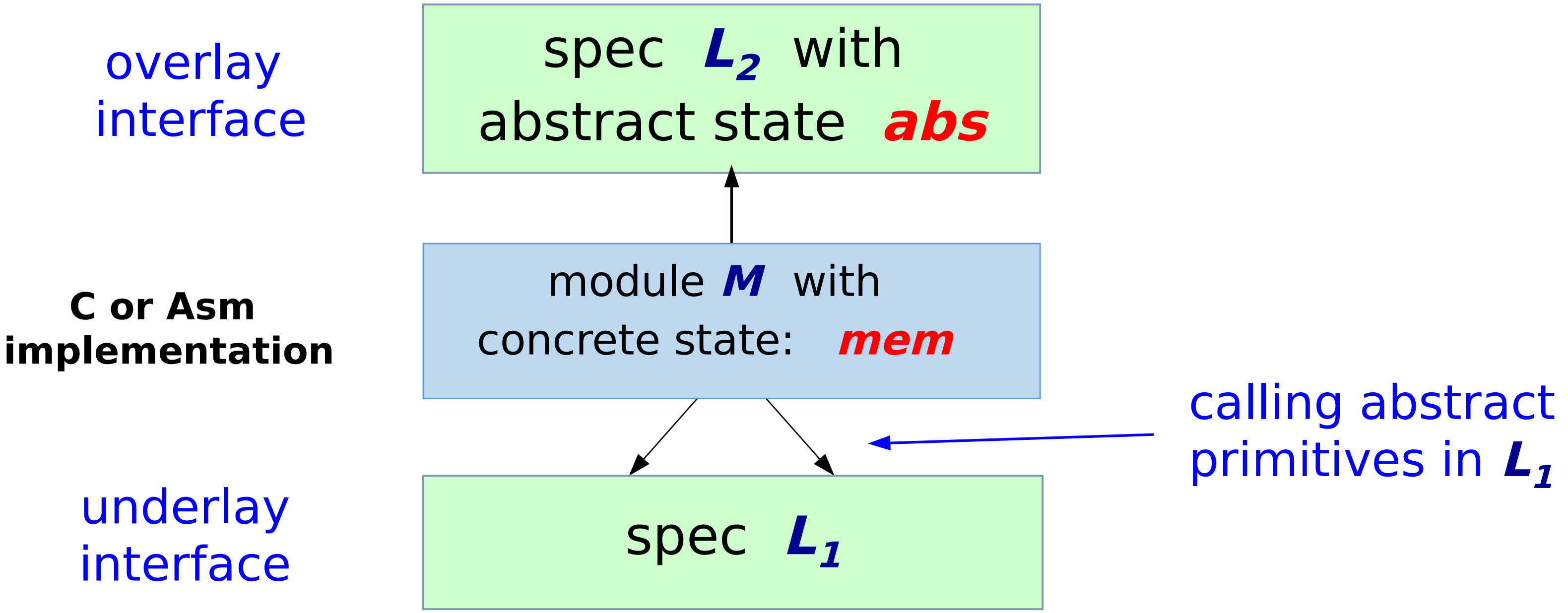
spec ***L_1***

calling abstract primitives in ***L_1***

Formalizing Abstraction Layers

What is a ***certified*** abstraction layer

(L_1, M, L_2) ?



overlay
interface

spec L_2 with
abstract state ***abs***

C or Asm
implementation

module ***M*** with
concrete state: ***mem***

calling abstract
primitives in L_1

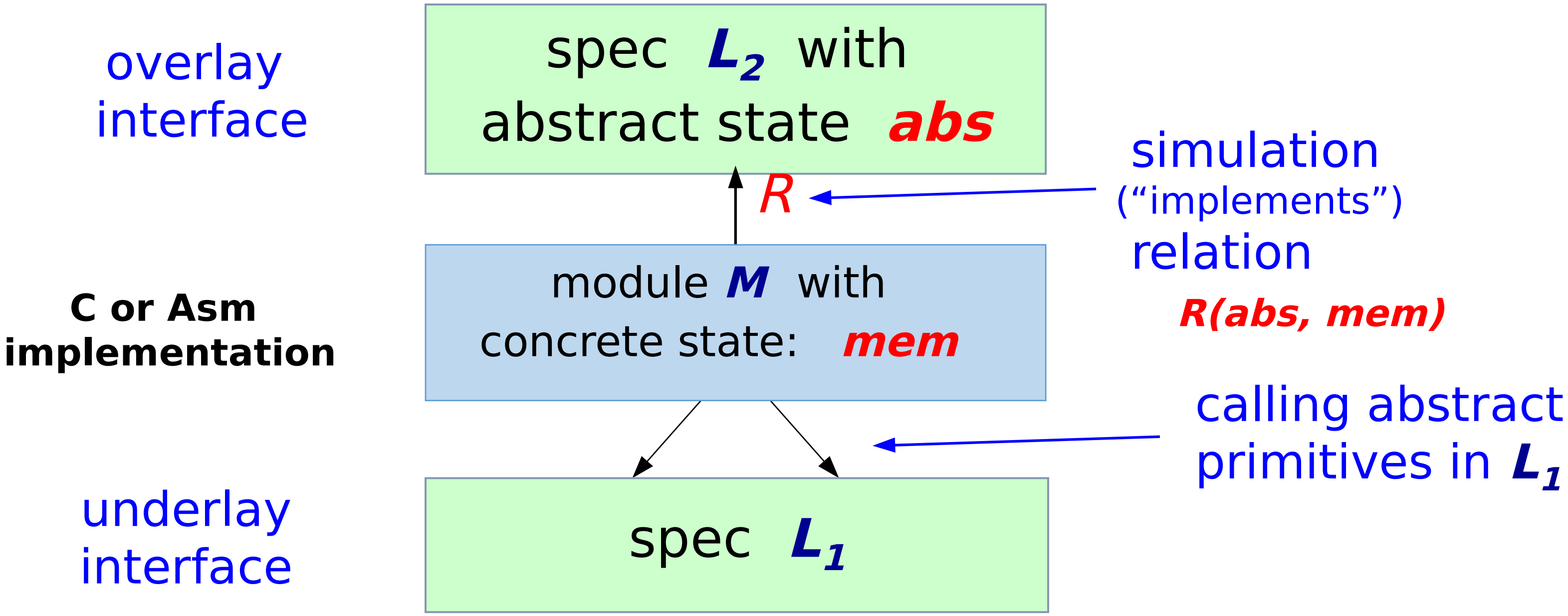
underlay
interface

spec L_1

Formalizing Abstraction Layers

What is a ***certified*** abstraction layer

$(L_1,\ M,\ L_2)$?



overlay interface

spec L_2 with abstract state ***abs***

R

simulation (“implements”) relation

R(abs, mem)

C or Asm implementation

module M with concrete state: ***mem***

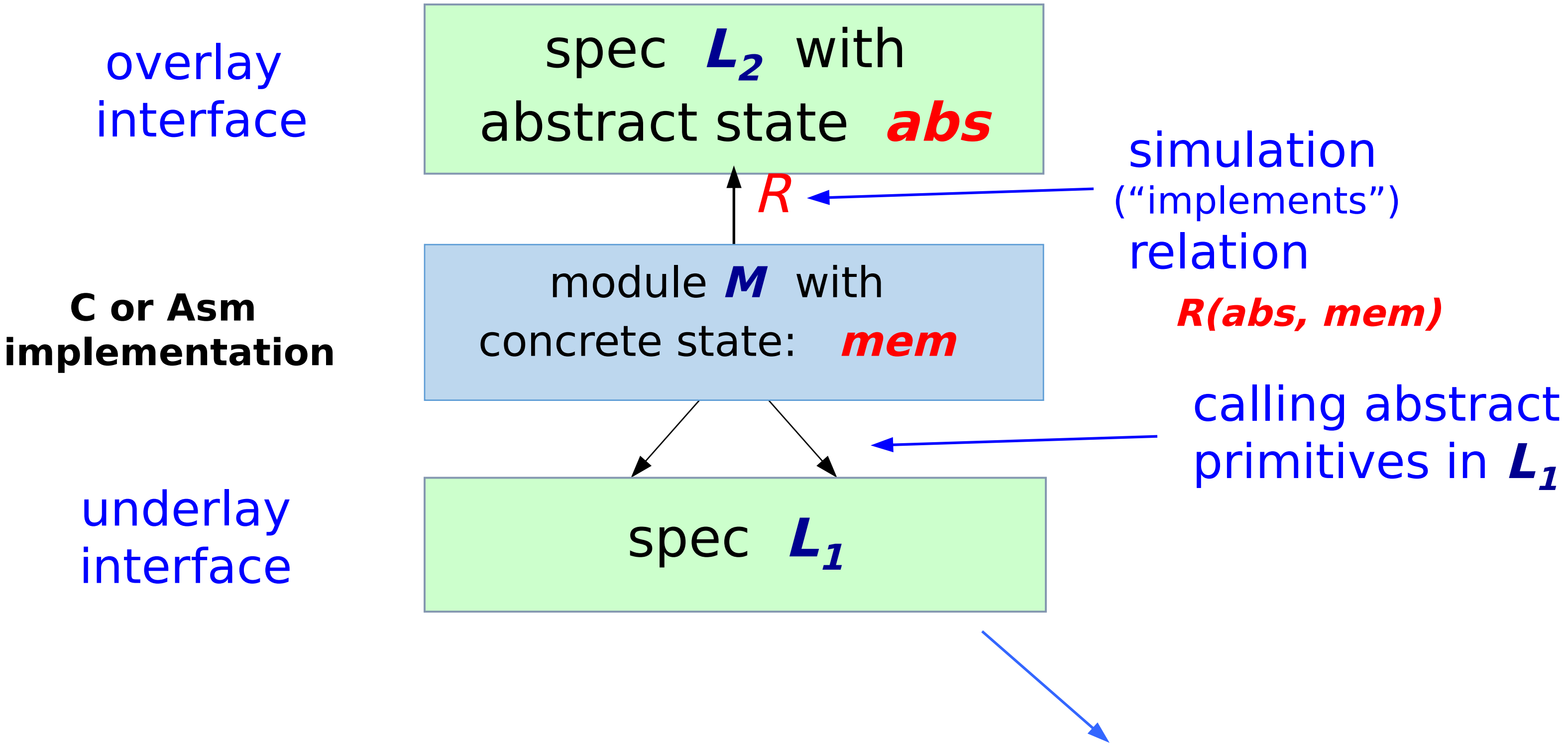
calling abstract primitives in L_1

underlay interface

spec L_1

Formalizing Abstraction Layers

What is a ***certified*** abstraction layer
(L_1, M, L_2) ?



overlay interface

spec L_2 with
abstract state ***abs***

R

simulation
("implements")
relation

R(abs, mem)

C or Asm implementation

module M with
concrete state: ***mem***

calling abstract primitives in L_1

underlay interface

spec L_1

Recorded as the *well-formed layer* judgment

$$\boxed{L_1 \vdash_R M : L_2}$$

Compositional Verification:
The "Implements" Simulation Relation

$L_1 \vdash_R M : L_2$ Defined as $L_2 \le_R [\![\; M \;]\!] \; L_1$

compositional
per-module
semantics
$[\![\, \bullet \,]\!]$

Compositional Verification: The “Implements” Simulation Relation

$L_1 \vdash_R M : L_2$ Defined as $L_2 \leq_R [\![\; M \;]\!] \; L_1$

compositional
per-module
semantics
$[\![\bullet]\!]$

for each function **f** in Dom(L_2)

$$\text{abs2} \xrightarrow{L_2(\mathbf{f})} \text{abs2}'$$

Compositional Verification:
The “Implements” Simulation Relation

$\boldsymbol{L_1} \vdash_{\boldsymbol{R}} \boldsymbol{M} : \boldsymbol{L_2}$ Defined as $\boldsymbol{L_2} \leq_R [\![\boldsymbol{M}]\!] \boldsymbol{L_1}$

compositional
per-module
semantics
$[\![\bullet]\!]$

for each function **f** in Dom($\boldsymbol{L_2}$)

$$\begin{tikzcd}
\text{abs2} \arrow[r, "\boldsymbol{L_2}\mathbf{(f)}"] \arrow[d, dash, "R"'] & \text{abs2'} \\
\text{mem1} &
\end{tikzcd}$$

Compositional Verification: The “Implements” Simulation Relation

$\boxed{L_1 \vdash_R M : L_2}$ Defined as $\boxed{L_2 \leq_R \llbracket M \rrbracket L_1}$

compositional
per-module
semantics
$\llbracket \bullet \rrbracket$

for each function **f** in Dom(L_2)

$$\begin{tikzcd}
\text{abs2} \arrow[r, "L_2(\mathbf{f})"] \arrow[d, dash, "R"'] & \text{abs2'} \\
\text{mem1} \arrow[r, dotted, "\llbracket M \rrbracket (L_1)(\mathbf{f})"'] & \text{mem1'}
\end{tikzcd}$$

Compositional Verification: The “Implements” Simulation Relation

$\boxed{L_1 \vdash_R M : L_2}$ Defined as $\boxed{L_2 \le_R [\![M]\!] L_1}$

compositional per-module semantics $[\![\bullet]\!]$

for each function **f** in Dom(L_2)

$$\begin{tikzcd}
\text{abs2} \arrow[r, "L_2(\mathbf{f})"] \arrow[d, dash, "R"'] & \text{abs2'} \arrow[d, dash, dotted, "R"] \\
\text{mem1} \arrow[r, dotted, "{[\![M]\!] (L_1)(\mathbf{f})}"'] & \text{mem1'}
\end{tikzcd}$$

Forward Downward Simulation:

- Whenever L_2(**f**) takes abs2 to abs2’ in one step, and R(abs2, mem1) holds,
- then there exists mem1’ such that $[\![M]\!]$ (L_1)(**f**) takes mem1 to mem1’ in zero or more steps , and R(abs2’, mem1’) also holds.

Compositional Verification: The “Implements” Simulation Relation

$\boxed{L_1 \vdash_R M : L_2}$ Defined as $\boxed{L_2 \le_R [\![M]\!] L_1}$

compositional per-module semantics $[\![\bullet]\!]$

for each function **f** in Dom(L_2)

$$\begin{tikzcd}
\text{abs2} \arrow[r, "L_2(\mathbf{f})"] \arrow[d, dash, "R"'] & \text{abs2'} \arrow[d, dash, dotted, "R"] \\
\text{mem1} \arrow[r, dotted, "{[\![M]\!] (L_1)(\mathbf{f})}"'] & \text{mem1'}
\end{tikzcd}$$

Forward Downward Simulation

Why is it enough?

Forward Downward Simulation:

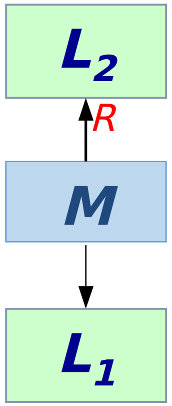
- Whenever L_2**(f)** takes abs2 to abs2’ in one step, and R(abs2, mem1) holds,
- then there exists mem1’ such that $[\![M]\!] (L_1)$**(f)** takes mem1 to mem1’ in zero or more steps , and R(abs2’, mem1’) also holds.

Deep Specification and Contextual Refinement

Our ultimate goal:

Making refinement "contextual" using the whole-program semantics $[\![\bullet]\!]$

$$\boxed{L_1 \models_R M : L_2}$$



L_2 is a **deep specification** of M over L_1

if under any valid program context P of L_2,

$[\![P \oplus M]\!] (L_1)$ and $[\![P]\!] (L_2)$ are

observationally equivalent

L_2 captures everything about running M over L_1

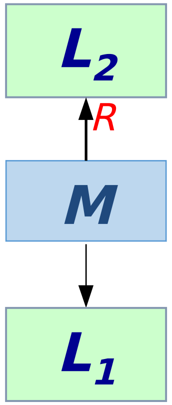
Soundness

Compositional Verification

$L_1 \vdash_R M : L_2$

Making refinement “contextual” using the whole-program semantics 【•】

$L_1 \models_R M : L_2$



L_2 is a **deep specification** of M over L_1

if under any valid program context P of L_2,

【 $P \oplus M$ 】 (L_1) and 【 P 】 (L_2) are

observationally equivalent

L_2 captures everything about running M over L_1

Reversing the Simulation Relation

$$\boxed{L_1 \vdash_R M : L_2} \longrightarrow \boxed{L_2 \leq_R [\![M]\!] L_1}$$

Reversing the Simulation Relation

$$L_1 \vdash_R M : L_2 \quad \longrightarrow \quad L_2 \leq_R \llbracket M \rrbracket \, L_1$$

$$\Big\downarrow \text{ Contextualize}$$

$\forall$ valid P:

$$\text{【} P \text{】}(L_2) \leq_R \text{【} P \oplus M \text{】}(L_1)$$

Reversing the Simulation Relation

$L_1 \vdash_R M : L_2$ → $L_2 \leq_R [\![M]\!] L_1$

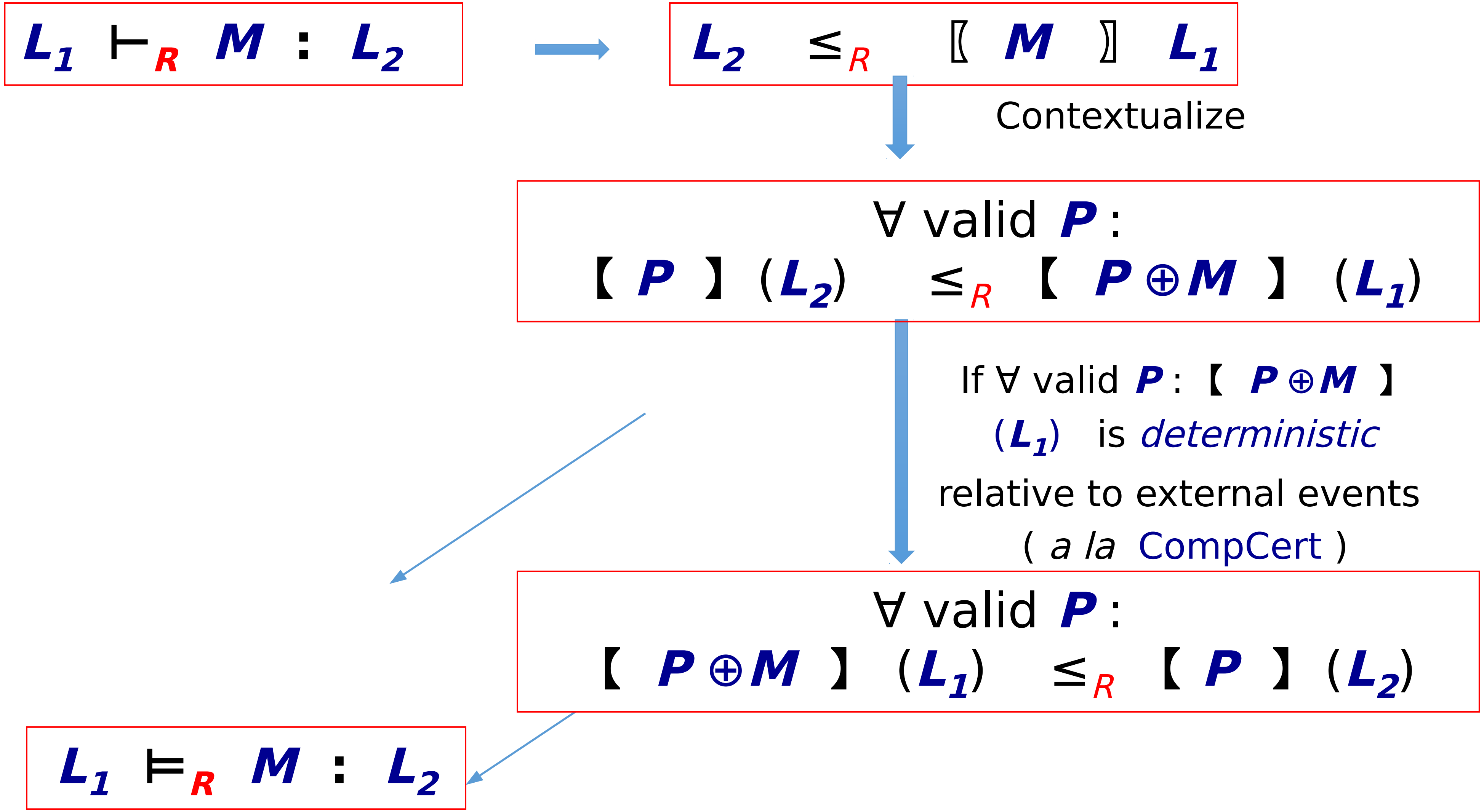
Contextualize

$$\forall \text{ valid } P : \\ 【P】(L_2) \leq_R 【P \oplus M】(L_1)$$

If ∀ valid P: $【P \oplus M】(L_1)$ is *deterministic* relative to external events (*a la* CompCert)

$$\forall \text{ valid } P : \\ 【P \oplus M】(L_1) \leq_R 【P】(L_2)$$

Reversing the Simulation Relation



$L_1 \vdash_R M : L_2$

$L_2 \leq_R [\![M]\!] \; L_1$

Contextualize

∀ valid P:

$[\![P]\!](L_2) \leq_R [\![P \oplus M]\!](L_1)$

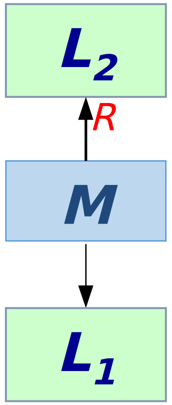
If ∀ valid P: $[\![P \oplus M]\!](L_1)$ is *deterministic* relative to external events (*a la* CompCert)

∀ valid P:

$[\![P \oplus M]\!](L_1) \leq_R [\![P]\!](L_2)$

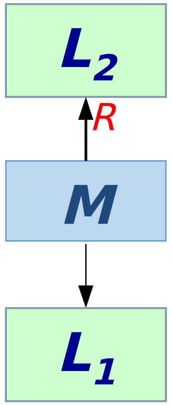
$L_1 \vDash_R M : L_2$

Why Deep Spec is Really Cool?



L_2 is a **deep specification** of M over L_1 if under any valid program context P of L_2, 【 $P \oplus M$ 】 (L_1) and 【 P 】 (L_2) are observationally equivalent

Why Deep Spec is Really Cool?



L_2 is a **deep specification** of M over L_1 if under any valid program context P of L_2, $\llbracket P \oplus M \rrbracket (L_1)$ and $\llbracket P \rrbracket (L_2)$ are observationally equivalent

Deep spec L_2 captures all we need to know about a layer M

- No need to ever look at M again!
- Any property about M can be proved using L_2 alone.

Implementation Independence : any two implementations of the same deep spec are *contextually equivalent*

POPL
2015
Mumbai, India
January 12-18

Our Contributions

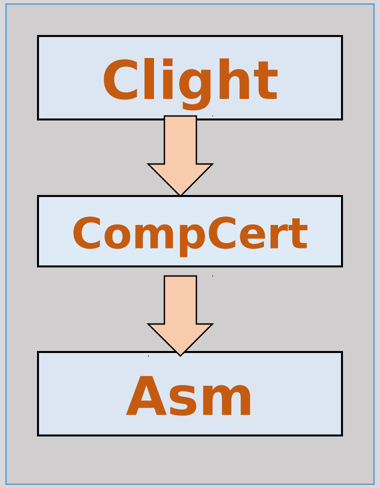
- We introduce deep specification and present a language-based formalization of certified abstraction layer

- We developed new languages & tools in Coq
  - A formal layer calculus for composing certified layers
  - ClightX for writing certified layers in a C-like language
  - AsmX for writing certified layers in assembly
  - CompCertX that compiles ClightX layers into AsmX layers

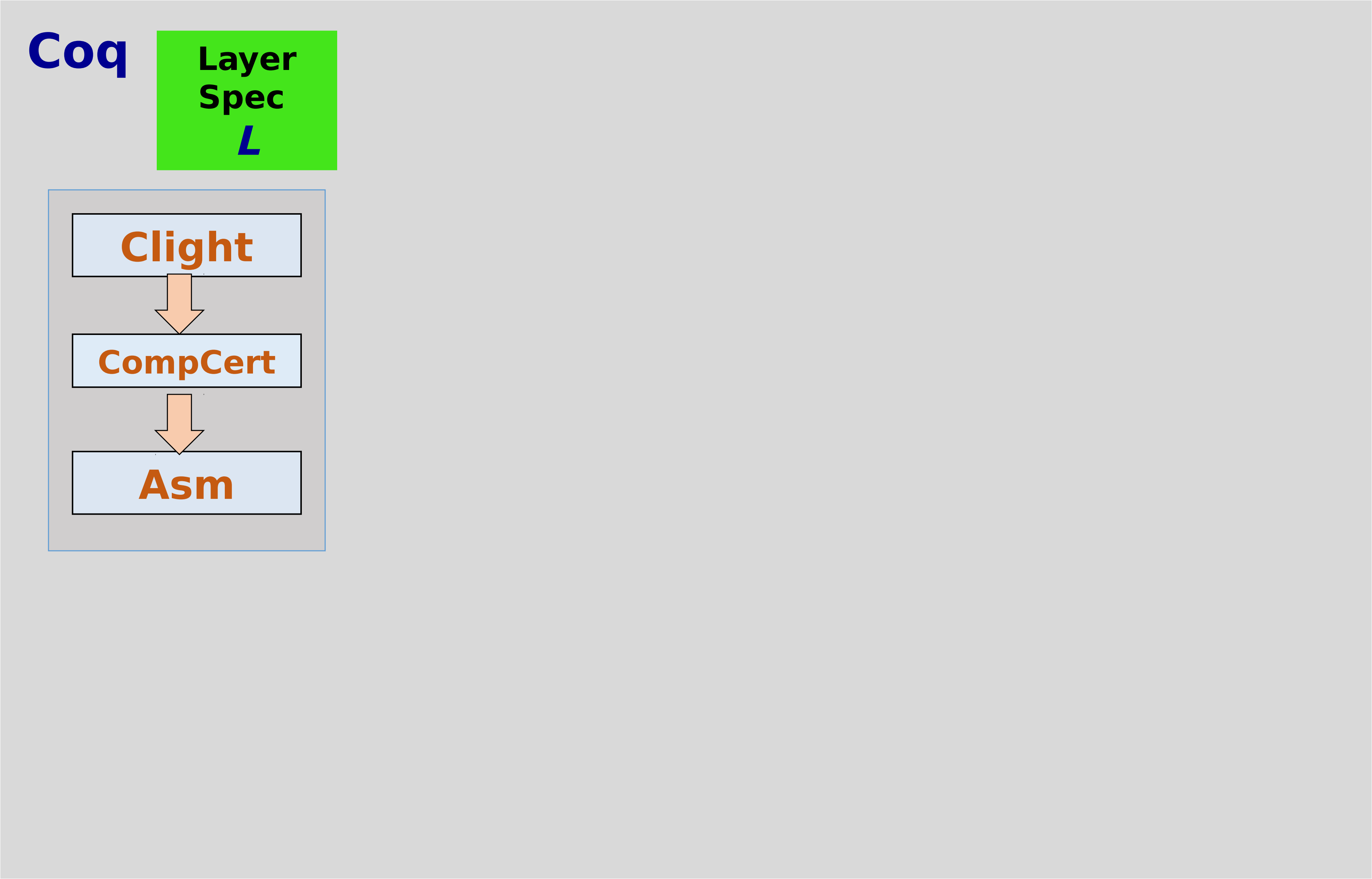
- We built multiple certified OS kernels in Coq
  - mCertiKOS-hyper consists of 37 layers, took less than one-person-year to develop, and can boot Linux as a guest

Coq

Coq



What We Have Done



Coq

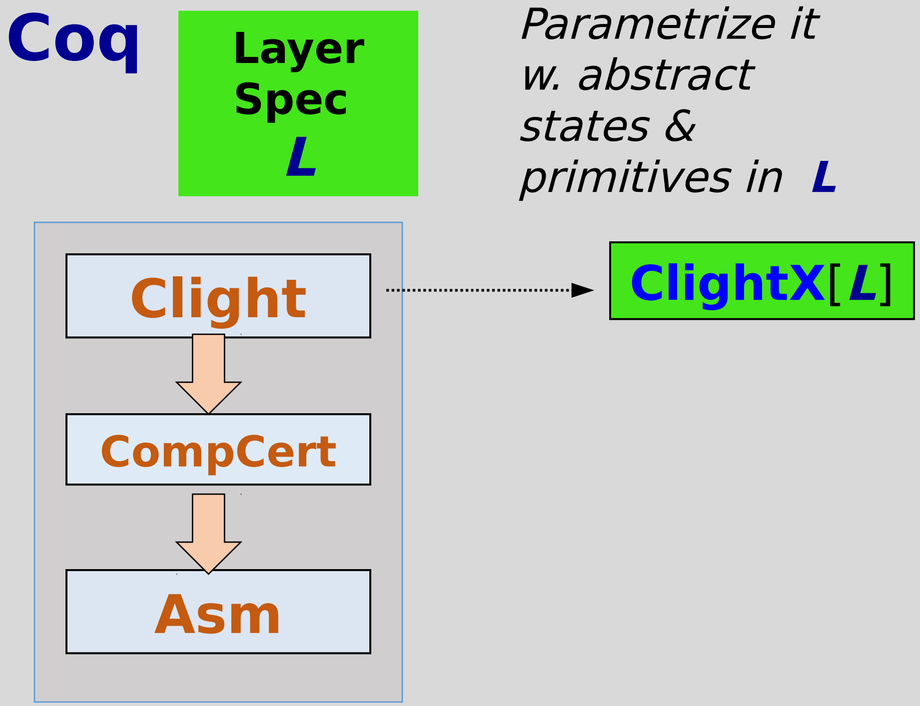
Layer Spec *L*

Clight

CompCert

Asm

What We Have Done



Coq

Layer Spec *L*

*Parametrize it w. abstract states & primitives in **L***

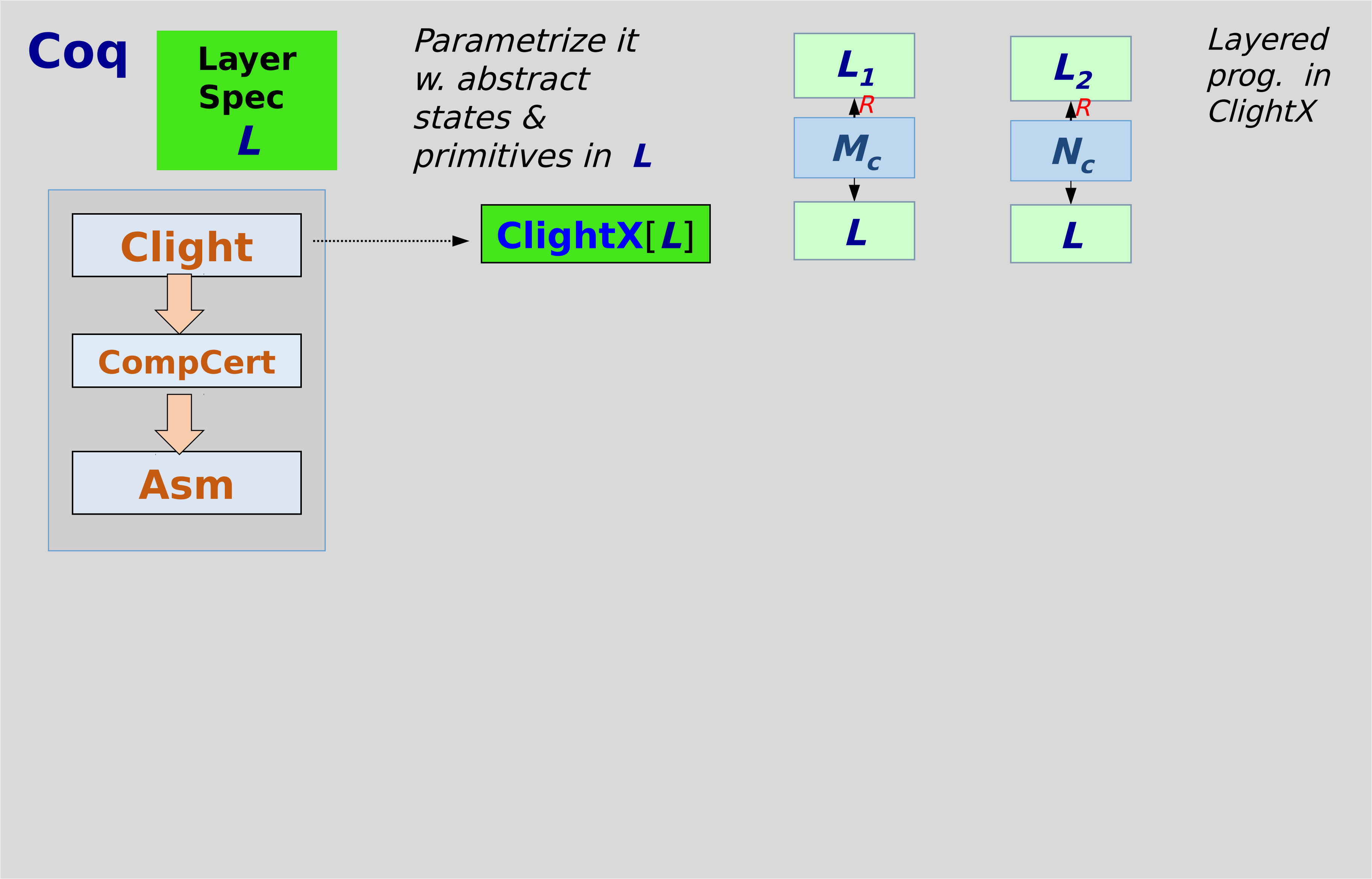
Clight

CompCert

Asm

ClightX[*L*]

What We Have Done



Coq

Layer Spec L

Parametrize it w. abstract states & primitives in L

Clight

CompCert

Asm

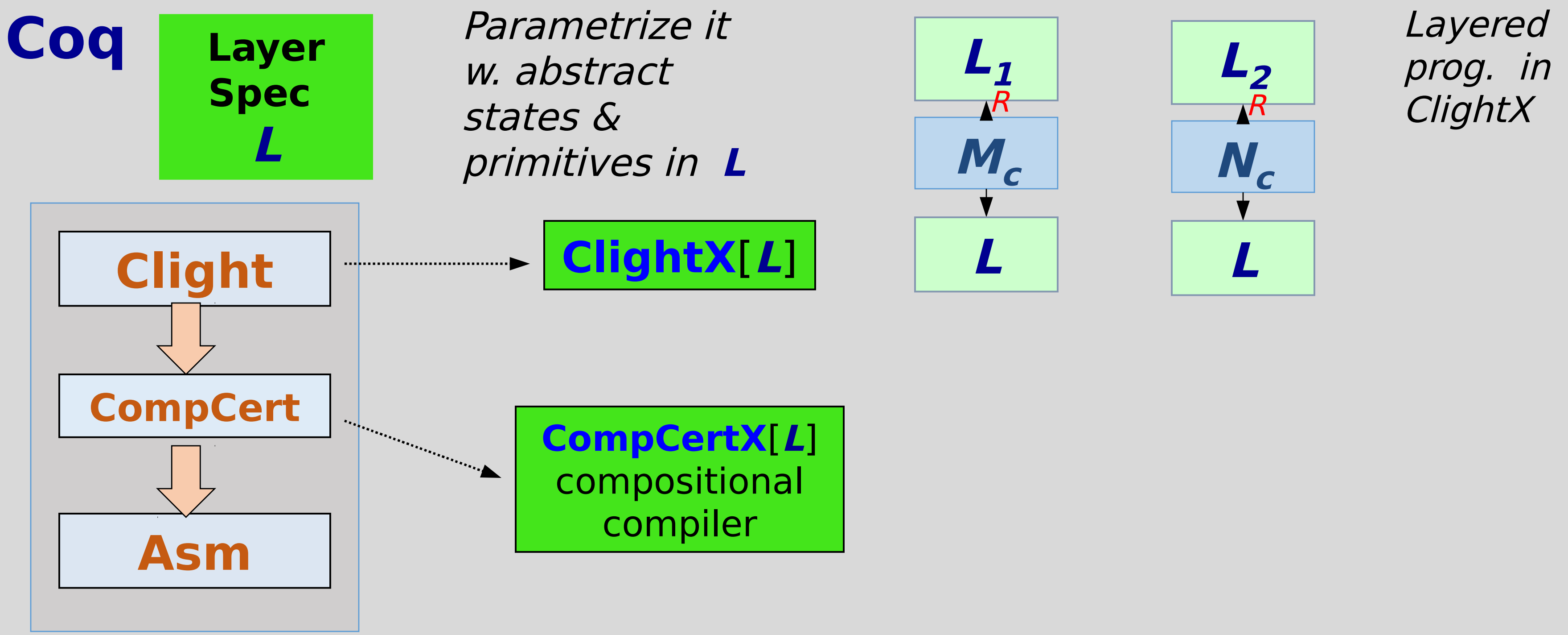
ClightX[L]

L_1
R
M_c
L

L_2
R
N_c
L

Layered prog. in ClightX

What We Have Done



Coq

Layer Spec
L

Parametrize it w. abstract states & primitives in ***L***

Clight

CompCert

Asm

ClightX[*L*]

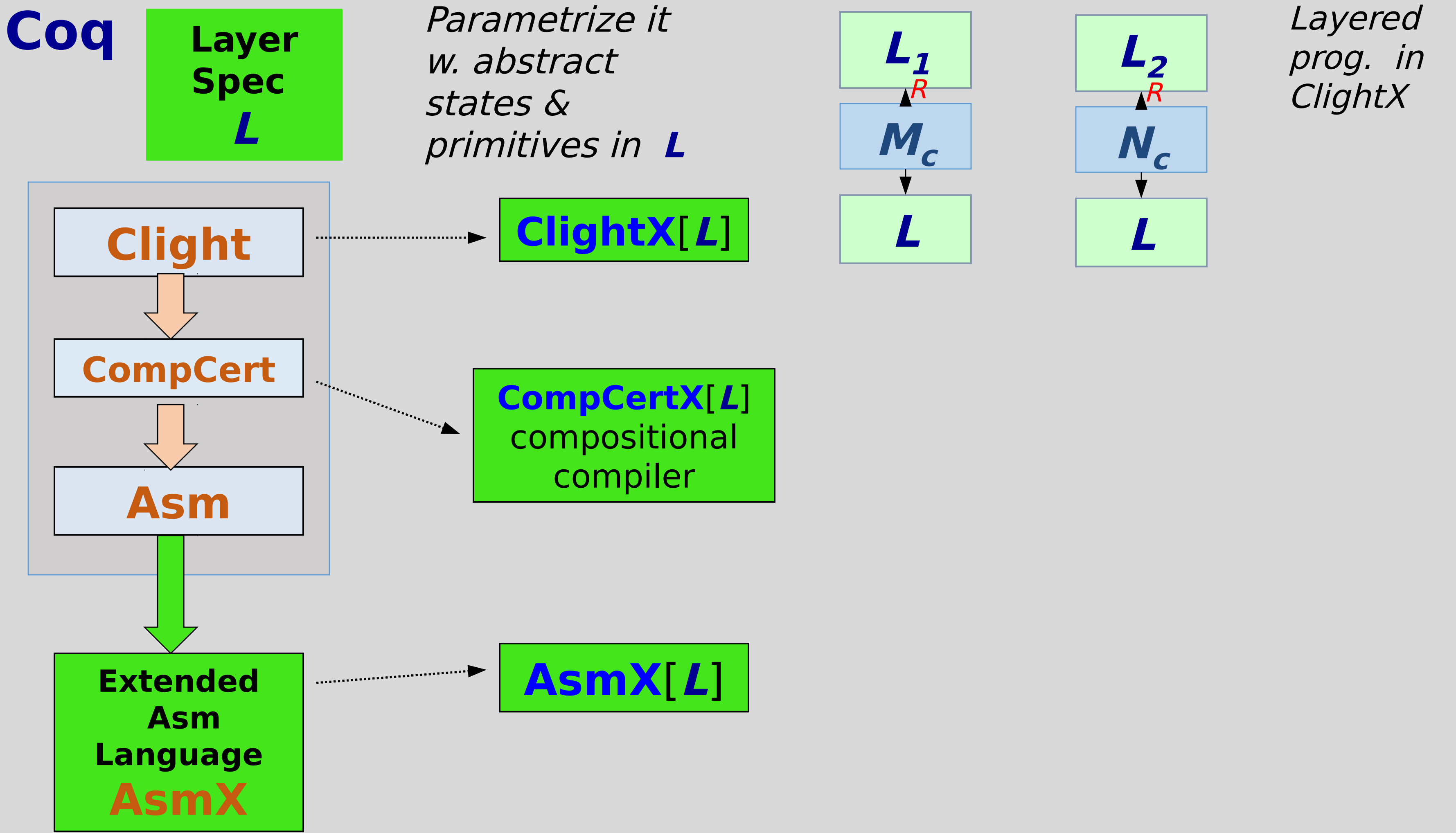
CompCertX[*L*]
compositional compiler

L_1
R
M_c
L

L_2
R
N_c
L

Layered prog. in ClightX

What We Have Done



Coq

Layer Spec
L

Parametrize it w. abstract states & primitives in ***L***

L_1 $\xleftarrow{R}$ M_c → L

L_2 $\xleftarrow{R}$ N_c → L

Layered prog. in ClightX

Clight → **ClightX[*L*]**

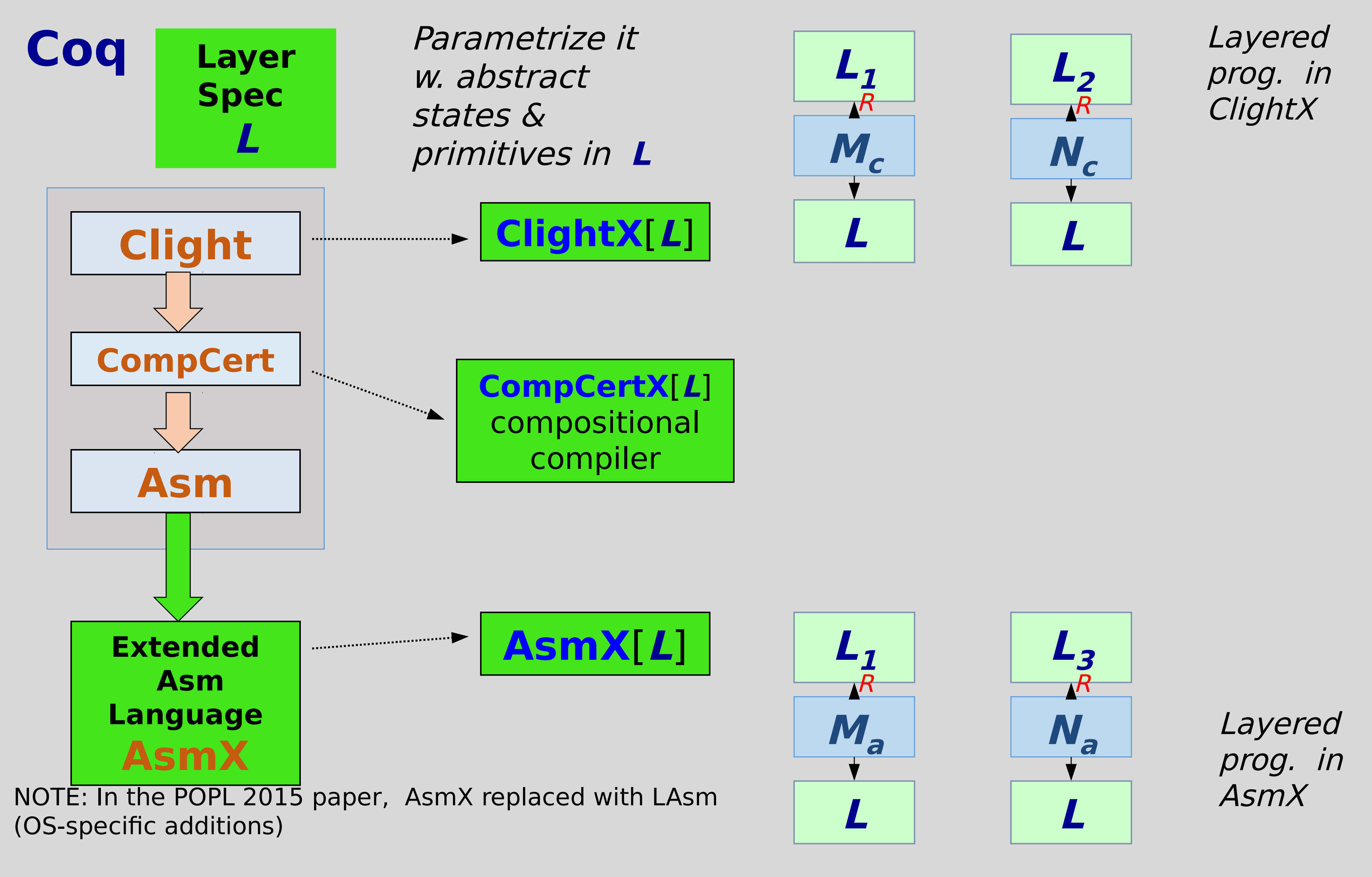
CompCert → **CompCertX[*L*]** compositional compiler

Asm

Extended Asm Language AsmX → **AsmX[*L*]**

NOTE: In the POPL 2015 paper, AsmX replaced with LAsm (OS-specific additions)

What We Have Done



Coq

Layer Spec *L*

*Parametrize it w. abstract states & primitives in **L***

Clight

CompCert

Asm

ClightX[*L*]

CompCertX[*L*] compositional compiler

Extended Asm Language AsmX

AsmX[*L*]

L_1 R M_c L

L_2 R N_c L

Layered prog. in ClightX

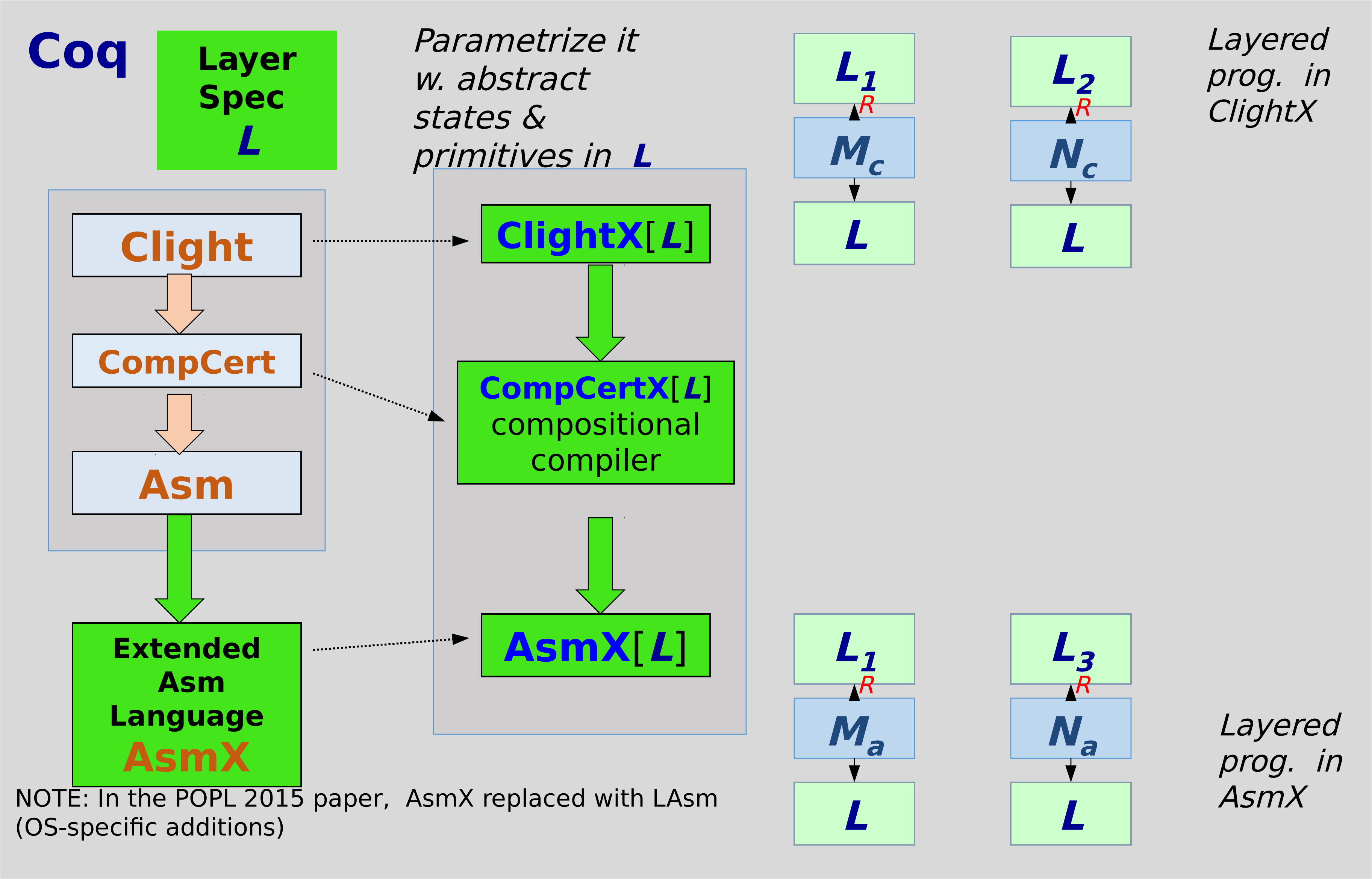
L_1 R M_a L

L_3 R N_a L

Layered prog. in AsmX

NOTE: In the POPL 2015 paper, AsmX replaced with LAsm (OS-specific additions)

What We Have Done



Coq

Layer Spec *L*

*Parametrize it w. abstract states & primitives in **L***

Clight

CompCert

Asm

Extended Asm Language AsmX

ClightX[*L*]

CompCertX[*L*] compositional compiler

AsmX[*L*]

L_1 $\xleftarrow{R}$ M_c → L

L_2 $\xleftarrow{R}$ N_c → L

Layered prog. in ClightX

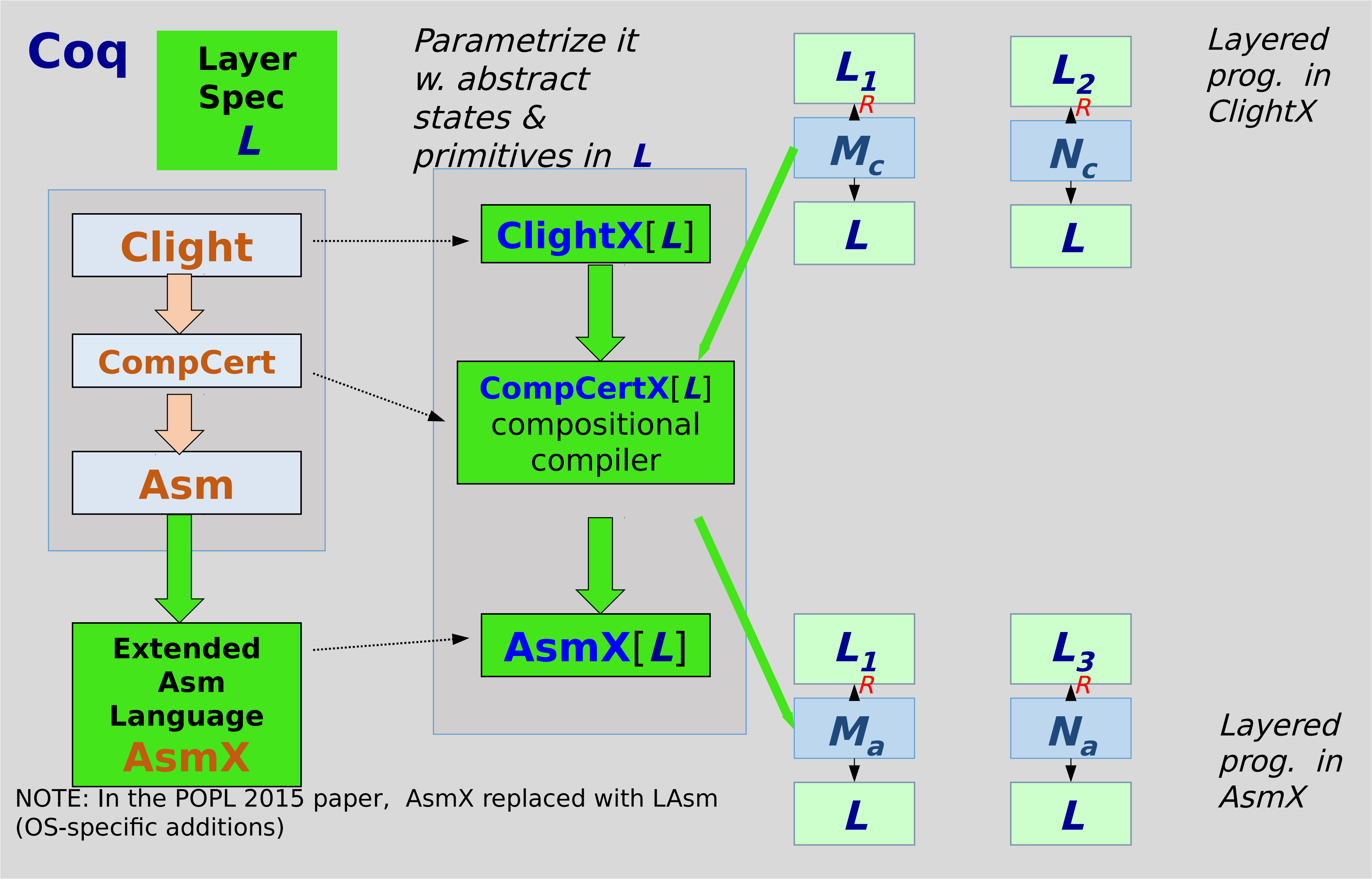
L_1 $\xleftarrow{R}$ M_a → L

L_3 $\xleftarrow{R}$ N_a → L

Layered prog. in AsmX

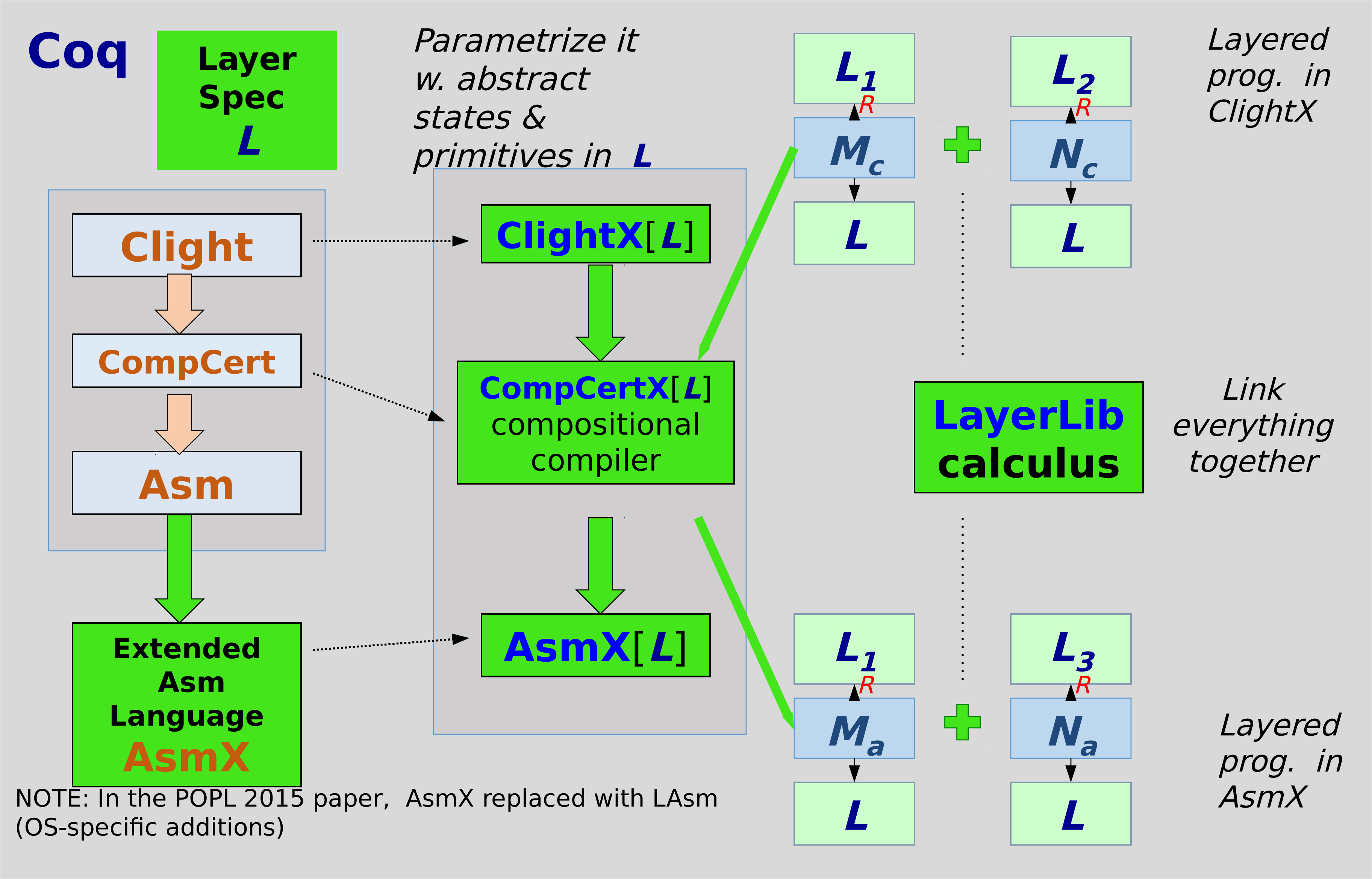
NOTE: In the POPL 2015 paper, AsmX replaced with LAsm (OS-specific additions)

What We Have Done



NOTE: In the POPL 2015 paper, AsmX replaced with LAsm (OS-specific additions)

What We Have Done



Coq

Layer Spec L

Parametrize it w. abstract states & primitives in L

Clight

CompCert

Asm

ClightX[L]

CompCertX[L] compositional compiler

AsmX[L]

Extended Asm Language AsmX

$L_1 \xleftarrow{R} M_c \rightarrow L$ + $L_2 \xleftarrow{R} N_c \rightarrow L$

Layered prog. in ClightX

LayerLib calculus

Link everything together

$L_1 \xleftarrow{R} M_a \rightarrow L$ + $L_3 \xleftarrow{R} N_a \rightarrow L$

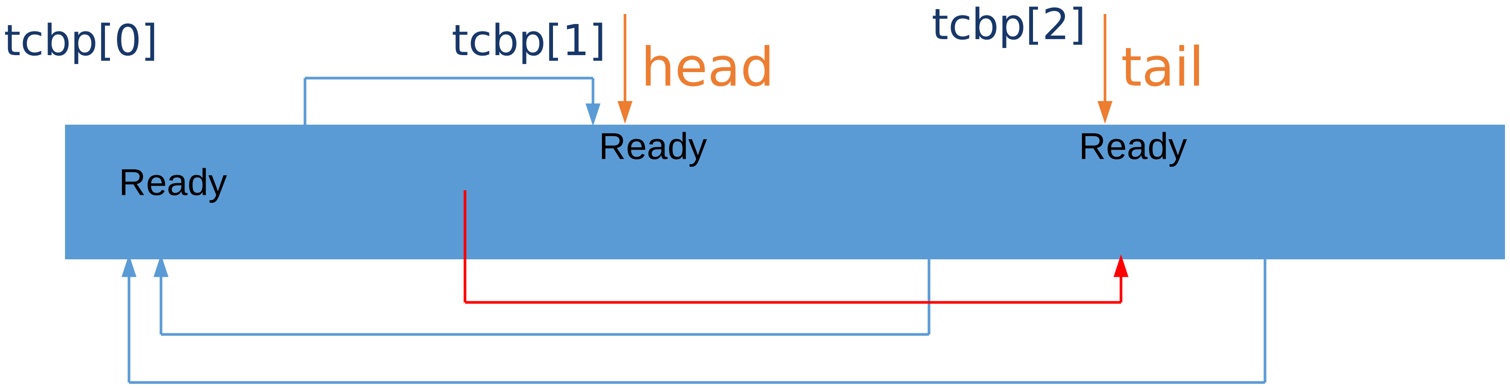
Layered prog. in AsmX

NOTE: In the POPL 2015 paper, AsmX replaced with LAsm (OS-specific additions)

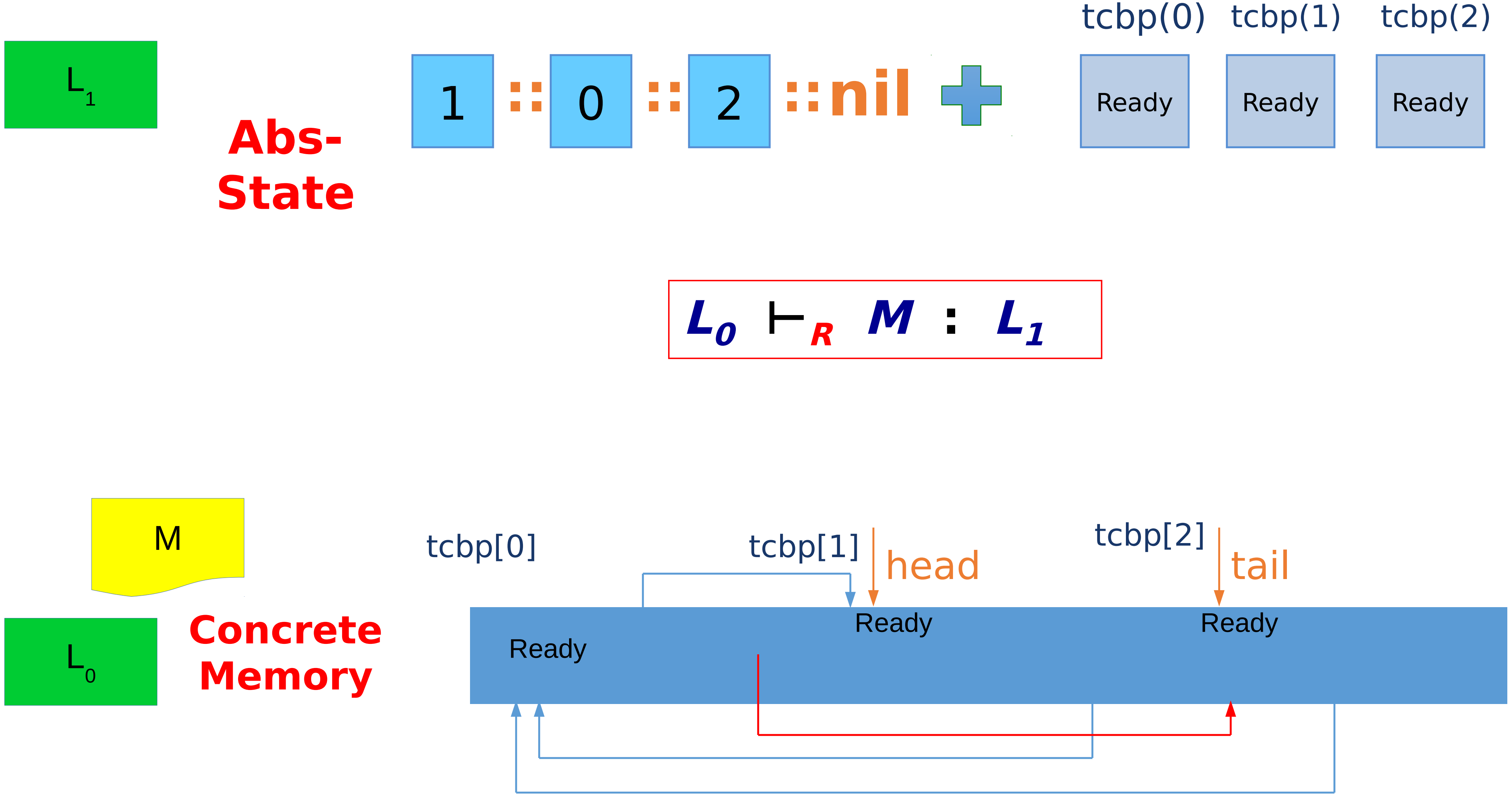
Example: Thread Queues

Abs-State

Concrete Memory



Example: Thread Queues



L_1

Abs-State

1 :: 0 :: 2 :: nil +

tcbp(0) tcbp(1) tcbp(2)

Ready Ready Ready

$$L_0 \vdash_R M : L_1$$

M

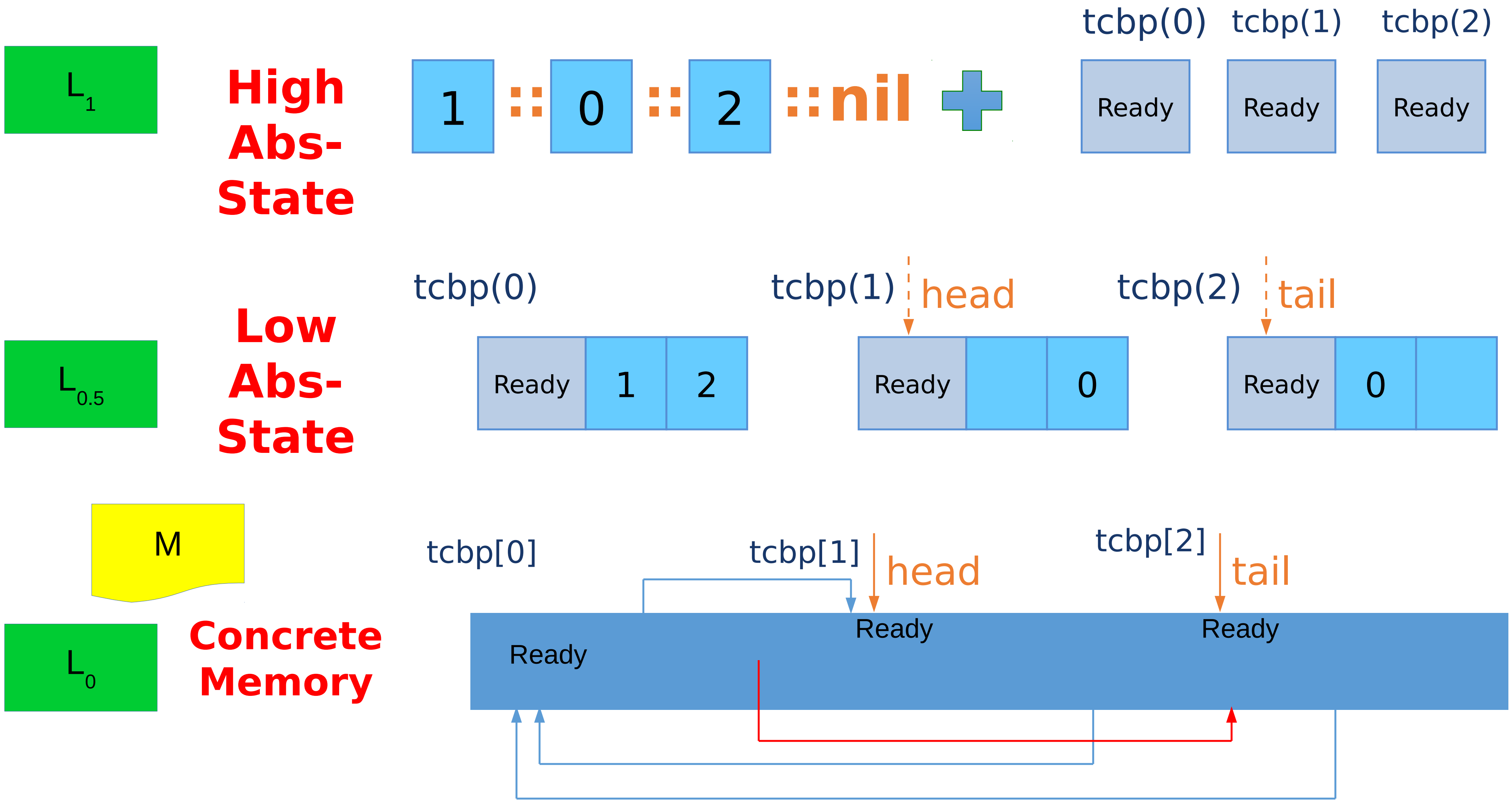
L_0

Concrete Memory

tcbp[0] tcbp[1] head tcbp[2] tail

Ready Ready Ready

Example: Thread Queues



L_1

High Abs-State

1 :: 0 :: 2 ::nil +

tcbp(0) Ready

tcbp(1) Ready

tcbp(2) Ready

$L_{0.5}$

Low Abs-State

tcbp(0) Ready 1 2

tcbp(1) head Ready 0

tcbp(2) tail Ready 0

M

L_0

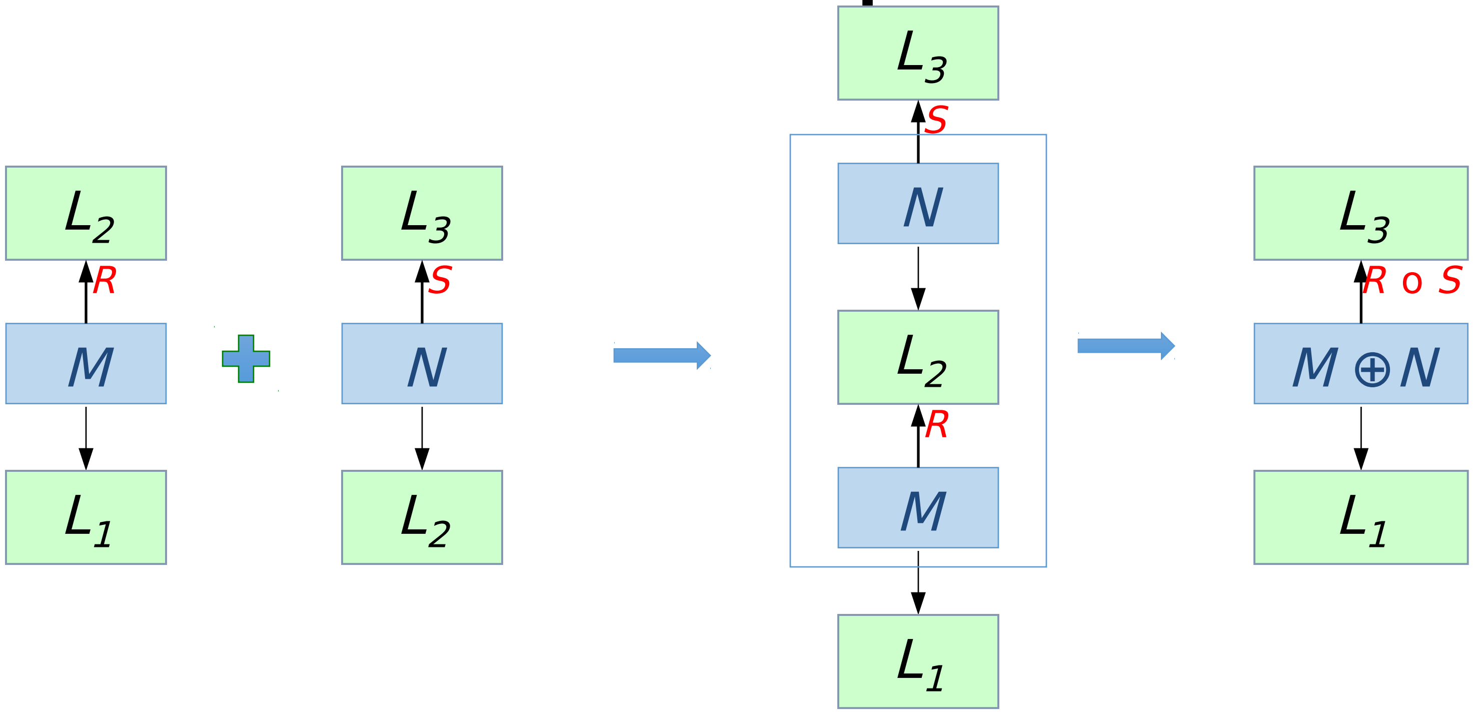
Concrete Memory

tcbp[0] Ready

tcbp[1] head Ready

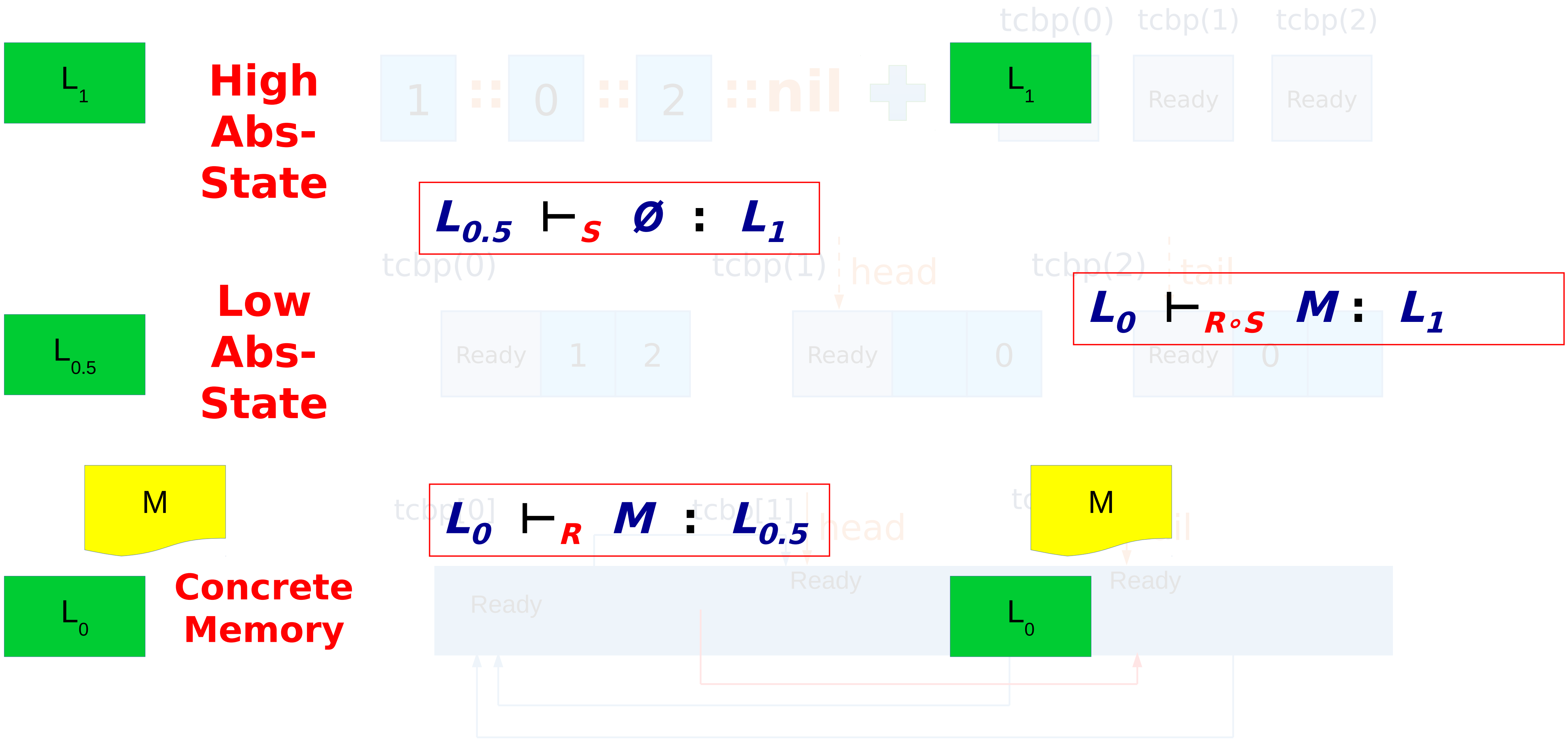
tcbp[2] tail Ready

LayerLib: Vertical Composition



$$\frac{L_1 \vdash_R M : L_2 \qquad L_2 \vdash_S N : L_3}{L_1 \vdash_{R \circ S} M \oplus N : L_3} \textsc{Vcomp}$$

Example: Thread Queues



High Abs-State

$L_{0.5} \vdash_{S} \emptyset : L_1$

Low Abs-State

$L_0 \vdash_{R \circ S} M : L_1$

$L_0 \vdash_{R} M : L_{0.5}$

Concrete Memory

Example: Thread Queues

C Implementation

Low Layer Spec in Coq

High Layer Spec in Coq

M

Example: Thread Queues

C Implementation

```
typedef enum {
   TD_READY, TD_RUN,
   TD_SLEEP, TD_DEAD
}  td_state;
```

Low Layer Spec in Coq

```
Inductive td_state :=
| TD_READY | TD_RUN
| TD_SLEEP | TD_DEAD.
```

High Layer Spec in Coq

```
Inductive td_state :=
| TD_READY | TD_RUN
| TD_SLEEP | TD_DEAD.
```

Example: Thread Queues

C Implementation

```
typedef enum {
   TD_READY, TD_RUN,
   TD_SLEEP, TD_DEAD
}  td_state;

struct tcb {
   td_state tds;
   struct tcb  *prev,  *next;
};
```

Low Layer Spec in Coq

```
Inductive td_state :=
| TD_READY | TD_RUN
| TD_SLEEP | TD_DEAD.


Record tcb := TCBV {
  tds : td_state;
  prev : Z; next : Z
}
```

High Layer Spec in Coq

```
Inductive td_state :=
| TD_READY | TD_RUN
| TD_SLEEP | TD_DEAD.

Definition tcb := td_state.
```

Example: Thread Queues

C Implementation

```
typedef enum {
   TD_READY, TD_RUN,
   TD_SLEEP, TD_DEAD
}  td_state;

struct tcb {
   td_state tds;
   struct tcb  *prev,  *next;
};

struct tdq {
  struct tcb  *head,  *tail;
};
```

Low Layer Spec in Coq

```
Inductive td_state :=
| TD_READY | TD_RUN
| TD_SLEEP | TD_DEAD.


Record tcb := TCBV {
  tds : td_state;
  prev : Z; next : Z
}

Record tdq := TDQV {
 head: Z; tail: Z
}
```

High Layer Spec in Coq

```
Inductive td_state :=
| TD_READY | TD_RUN
| TD_SLEEP | TD_DEAD.

Definition tcb := td_state.





Definition tdq := List Z.
```

Example: Thread Queues

C Implementation

```
typedef enum {
   TD_READY, TD_RUN,
   TD_SLEEP, TD_DEAD
}  td_state;

struct tcb {
   td_state tds;
   struct tcb  *prev,  *next;
};

struct tdq {
  struct tcb  *head,  *tail;
};

struct tcb tcbp[64];
struct tdq tdqp[64];
```

Low Layer Spec in Coq

```
Inductive td_state :=
| TD_READY | TD_RUN
| TD_SLEEP | TD_DEAD.


Record tcb := TCBV {
  tds : td_state;
  prev : Z; next : Z
}

Record tdq := TDQV {
 head: Z; tail: Z
}

Record abs:=ABS {
             tcbp :  ZMap.t tcb;
             tdqp : ZMap.t tdq
}
```

High Layer Spec in Coq

```
Inductive td_state :=
| TD_READY | TD_RUN
| TD_SLEEP | TD_DEAD.

Definition tcb := td_state.





Definition tdq := list Z.


Record abs := ABS {
             tcbp : ZMap.t tcb;
             tdqp : ZMap.t tdq
}
```

Example: Thread Queues

C Implementation

```
typedef enum {
   TD_READY, TD_RUN,
   TD_SLEEP, TD_DEAD
}  td_state;

struct tcb {
   td_state tds;
   struct tcb  *prev,  *next;
};

struct tdq {
  struct tcb  *head,  *tail;
};

struct tcb tcbp[64];
struct tdq tdqp[64];

struct tcb * dequeue
       (struct tdq *q)  {
               ……
}
```

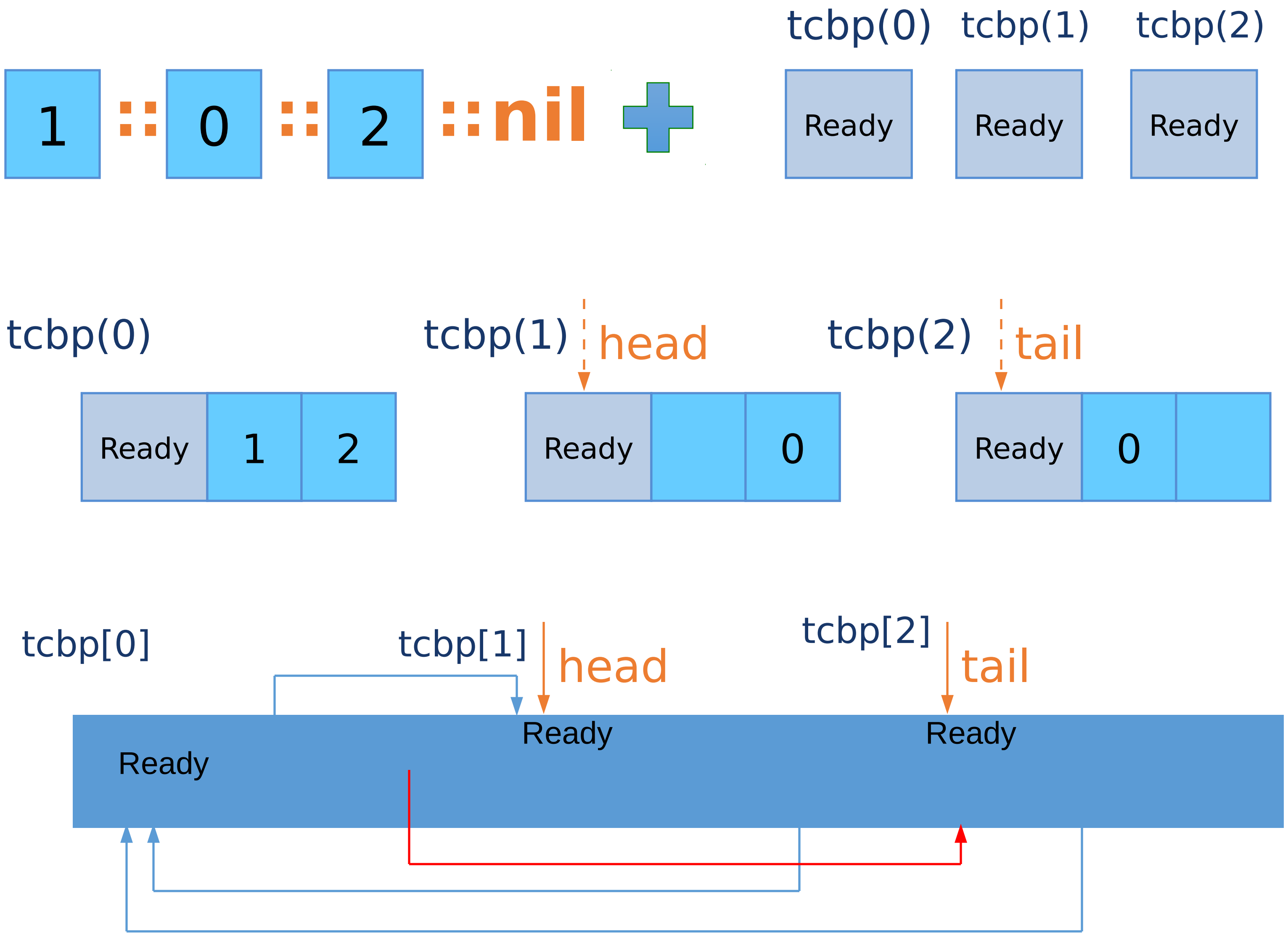
Low Layer Spec in Coq

```
Inductive td_state :=
| TD_READY | TD_RUN
| TD_SLEEP | TD_DEAD.


Record tcb := TCBV {
  tds : td_state;
  prev : Z; next : Z
}

Record tdq := TDQV {
 head: Z; tail: Z
}

Record abs:=ABS {
             tcbp :  ZMap.t tcb;
             tdqp : ZMap.t tdq
}

Definition dequeue
          (d : abs) (i : Z) :=
………………
```

High Layer Spec in Coq

```
Inductive td_state :=
| TD_READY | TD_RUN
| TD_SLEEP | TD_DEAD.

Definition tcb := td_state.




Definition tdq := list Z.


Record abs := ABS {
             tcbp : ZMap.t tcb;
             tdqp : ZMap.t tdq
}

Definition dequeue
       (d : abs) (i : Z) :=
match (d.tdqp i) with
   | h :: q' =>
         Some(set_tdq d i q', h)
   | nil => None
end
```

Example: Dequeue



High Abs-State

1 :: 0 :: 2 ::nil +

tcbp(0) tcbp(1) tcbp(2)

Ready Ready Ready

Low Abs-State

tcbp(0)

Ready 1 2

tcbp(1) head

Ready 0

tcbp(2) tail

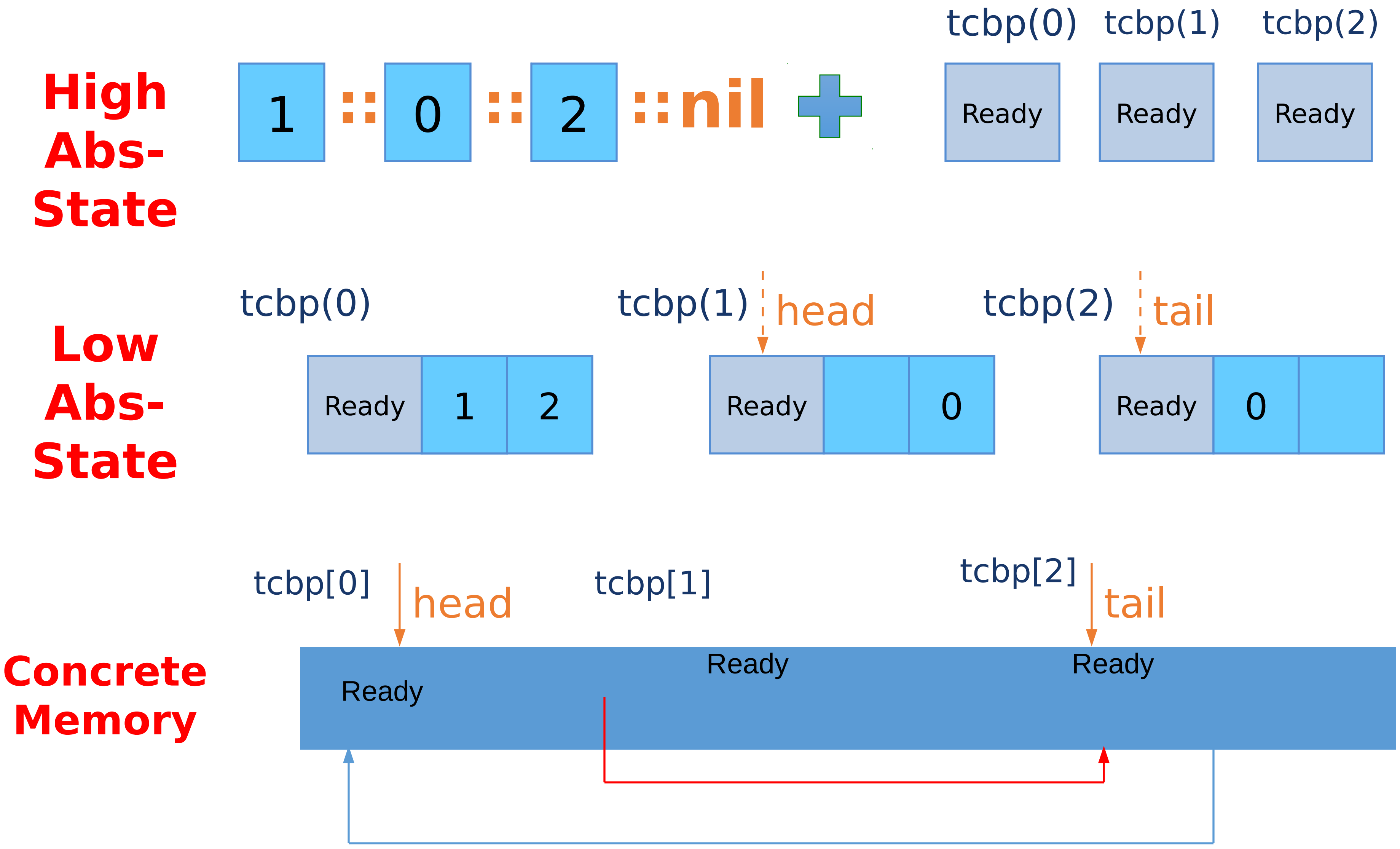
Ready 0

Concrete Memory

tcbp[0] tcbp[1] head tcbp[2] tail

Ready Ready Ready

Example: Dequeue



High Abs-State

1 :: 0 :: 2 ::nil +

tcbp(0) tcbp(1) tcbp(2)

Ready Ready Ready

Low Abs-State

tcbp(0)

Ready 1 2

tcbp(1) head

Ready 0

tcbp(2) tail

Ready 0

Concrete Memory

tcbp[0] head

tcbp[1]

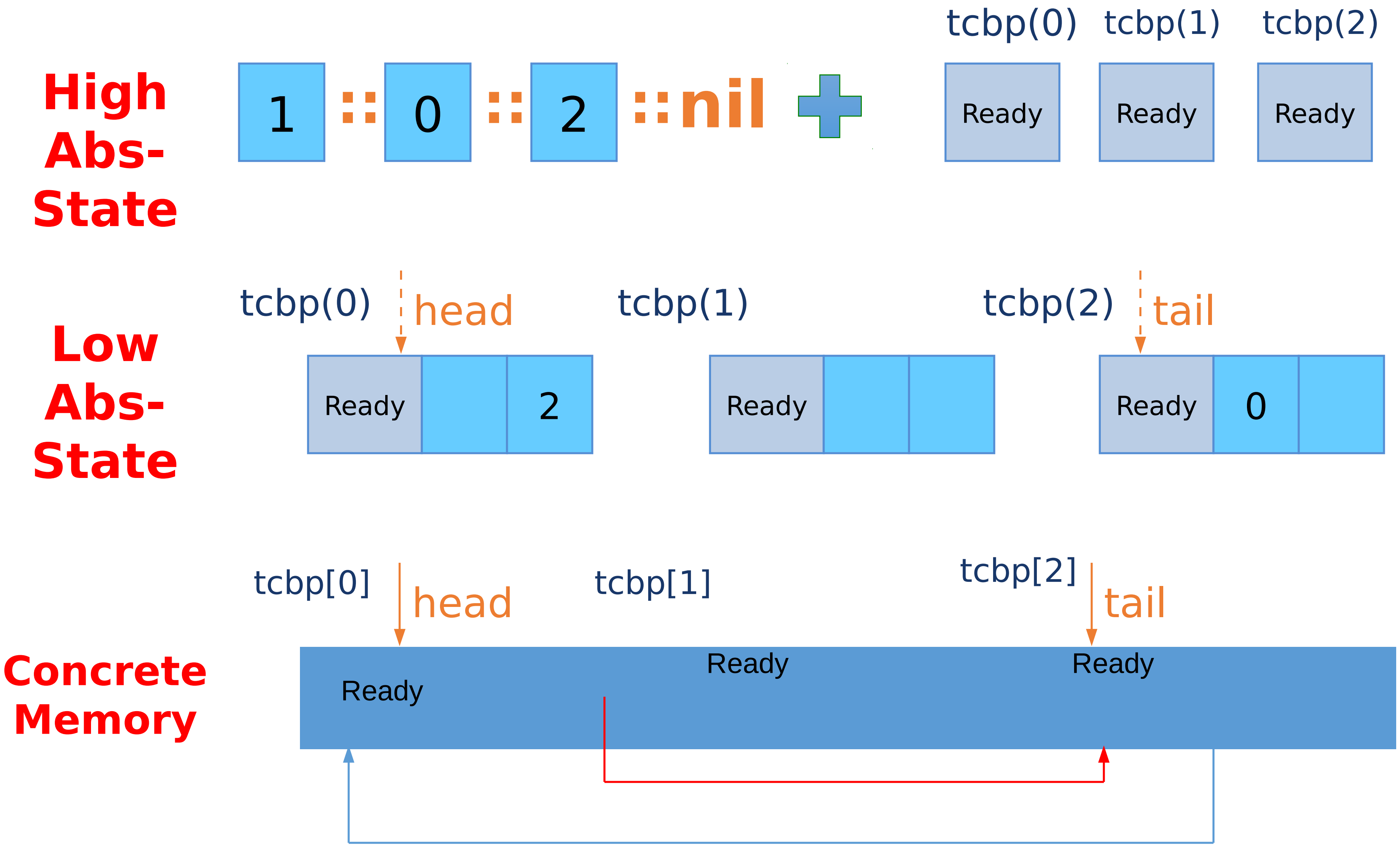
tcbp[2] tail

Ready

Ready

Ready

Example: Dequeue



High Abs-State

1 :: 0 :: 2 ::nil +

tcbp(0) Ready

tcbp(1) Ready

tcbp(2) Ready

Low Abs-State

tcbp(0) head: Ready | | 2

tcbp(1): Ready | |

tcbp(2) tail: Ready | 0 |

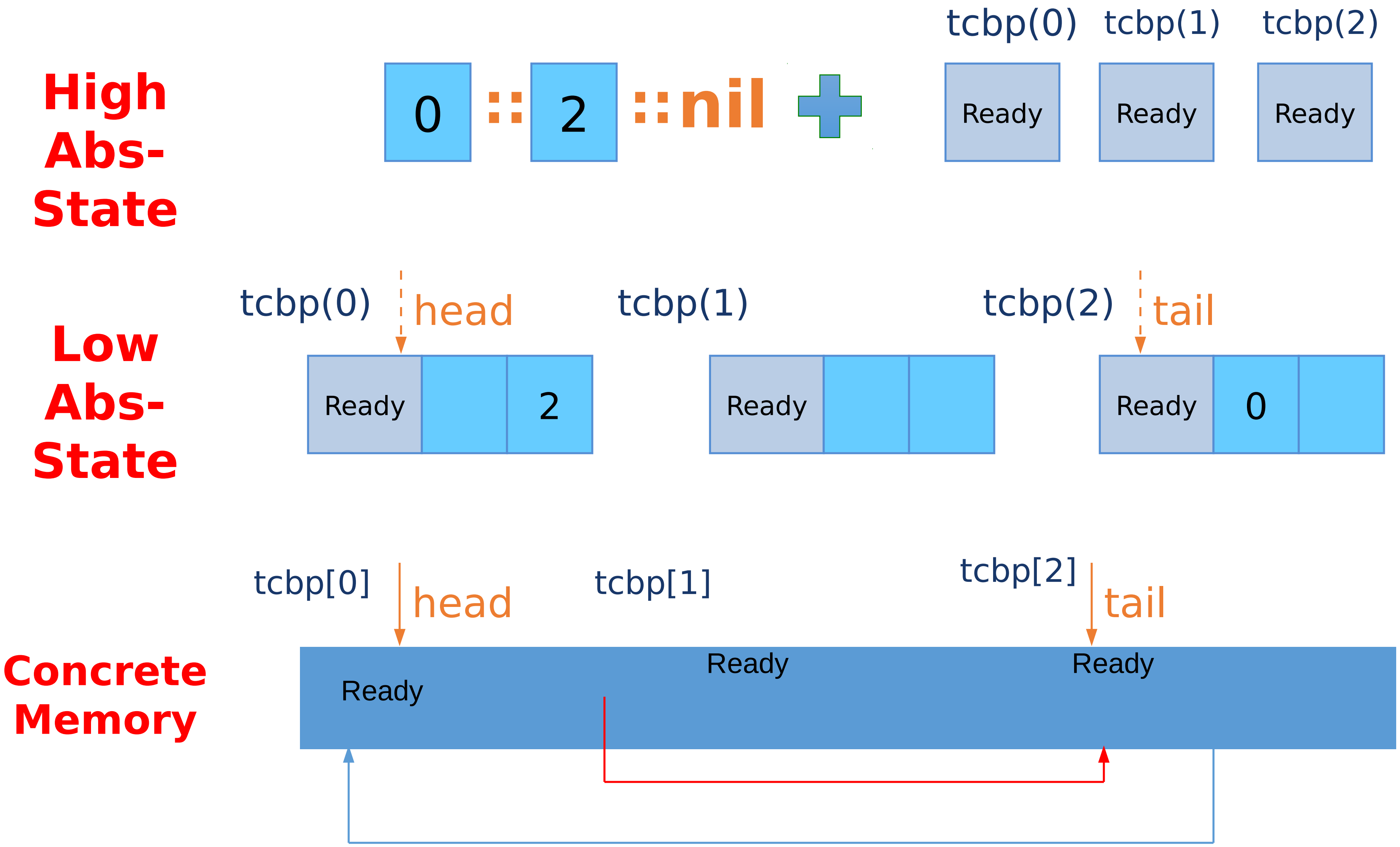
Concrete Memory

tcbp[0] head: Ready

tcbp[1]: Ready

tcbp[2] tail: Ready

Example: Dequeue



Our Contributions

- We introduce deep specification and present a language-based formalization of certified abstraction layer

- We developed new languages & tools in Coq
  - A formal layer calculus for composing certified layers
  - ClightX for writing certified layers in a C-like language
  - AsmX for writing certified layers in assembly
  - CompCertX that compiles ClightX layers into AsmX layers

- We built multiple certified OS kernels in Coq
  - mCertiKOS-hyper consists of 37 layers, took less than one-person-year to develop, and can boot Linux as a guest

From CompCert 2.3 to CompCertX

- Clight to ClightX, Asm to AsmX
  - Parameterize over layer specifications
  - Per-module semantics
- Compilation passes
  - Allow function arguments, etc.
- Summary of changes :
  - 420 lines changed (out of ~120k) due to parameterization
  - 7k new lines due to per-module semantics

Our Contributions

- We introduce deep specification and present a language-based formalization of certified abstraction layer
- We developed new languages & tools in Coq
  - A formal layer calculus for composing certified layers
  - ClightX for writing certified layers in a C-like language
  - AsmX for writing certified layers in assembly
  - CompCertX that compiles ClightX layers into AsmX layers
- We built multiple certified OS kernels in Coq
  - mCertiKOS-hyper consists of 37 layers, took less than one-person-year to develop, and can boot Linux as a guest

ClightX

- Same syntax as CompCert 2.3 Clight
- Semantics :
  - Parameterized over external function calls
  - Made per-function instead of whole-machine

CompCert Clight: external function calls

```
Inductive step: state → trace → state → Prop :=
| …
|  step_external_function: forall ef targs tres vargs k m vres t m',
      external_call ef ge vargs m t vres m' →
      step (Callstate (External ef targs tres) vargs k m)
         t (Returnstate vres k m')
| …
```

CompCert Clight: external function calls

```
Definition extcall_sem: Type  :=
   forall (F V: Type), Genv.t F V → list val → mem → trace → val → mem → Prop.
```

```
Definition external_call (ef: external_function): extcall_sem :=
  match ef with
  | EF_external name sg ⇒ external_functions_sem name sg
  | …
  end
```

```
Inductive step: state → trace → state → Prop :=
| …
|  step_external_function: forall ef targs tres vargs k m vres t m',
      external_call ef ge vargs m t vres m' →
      step (Callstate (External ef targs tres) vargs k m)
         t (Returnstate vres k m')
| …
```

CompCert Clight: external function calls

```
Definition extcall_sem: Type  :=
  forall (F V: Type), Genv.t F V → list val → mem → trace → val → mem → Prop.

Axiom external_functions_sem: ident → signature → extcall_sem.


Definition external_call (ef: external_function): extcall_sem :=
  match ef with
  | EF_external name sg ⇒ external_functions_sem name sg
  | …
  end



Inductive step: state → trace → state → Prop :=
| …
|  step_external_function: forall ef targs tres vargs k m vres t m',
      external_call ef ge vargs m t vres m' →
      step (Callstate (External ef targs tres) vargs k m)
         t (Returnstate vres k m')
| …
```

ClightX: parameterization over primitives

```
Definition extcall_sem: Type  :=
   forall (F V: Type), Genv.t F V → list val → mem → trace → val → mem → Prop.
```

```
Class ExtCallOps := {
  external_call (ef: external_function): extcall_sem
}.
```

```
Inductive step {ec_ops: ExtCallOps}: state → trace → state → Prop :=
| …
|  step_external_function: forall ef targs tres vargs k m vres t m',
      external_call ef ge vargs m t vres m' →
      step (Callstate (External ef targs tres) vargs k m)
         t (Returnstate vres k m')
| …
```

ClightX: parameterization

```
Definition extcall_sem: Type  :=
   forall (F V: Type), Genv.t F V → list val → mem → trace → val → mem → Prop.
```

```
Class ExtCallOps := {
  external_call (ef: external_function): extcall_sem
}.
```

Where is the abstract state ?

```
Inductive step {ec_ops: ExtCallOps}: state → trace → state → Prop :=
| …
|  step_external_function: forall ef targs tres vargs k m vres t m',
      external_call ef ge vargs m t vres m' →
      step (Callstate (External ef targs tres) vargs k m)
         t (Returnstate vres k m')
| …
```

ClightX: parameterization

```
Definition extcall_sem: Type  :=
   forall (F V: Type), Genv.t F V → list val → mem → trace → val → mem → Prop.
```

```
Class ExtCallOps := {
  external_call (ef: external_function): extcall_sem
}.
```

Here *fits* the abstract state !

```
Inductive step {ec_ops: ExtCallOps}: state → trace → state → Prop :=
| …
|  step_external_function: forall ef targs tres vargs k m vres t m',
      external_call ef ge vargs m t vres m' →
      step (Callstate (External ef targs tres) vargs k m)
         t (Returnstate vres k m')
| …
```

CompCertX: memory model

```
Class MemoryOps (mem: Type) := {
    load:    memory_chunk → mem → block → Z → option val;
    store:   memory_chunk → mem → block → Z → val → option mem;
    …
}
```

CompCertX: memory model

```
Class MemoryOps (mem: Type) := {
    load:    memory_chunk → mem → block → Z → option val;
    store:   memory_chunk → mem → block → Z → val → option mem;
    …
}
```

```
Instance compcert_mem: MemoryOps compcert.common.Memory.mem := …
```

CompCertX: memory model

```
Class MemoryOps (mem: Type) := {
    load:    memory_chunk → mem → block → Z → option val;
    store:   memory_chunk → mem → block → Z → val → option mem;
    …
}

Definition extcall_sem {mem: Type} {mem_ops: MemoryOps mem}: Type  :=
   forall (F V: Type), Genv.t F V → list val → mem → trace → val → mem → Prop.
```

CompCertX: memory model

```
Class MemoryOps (mem: Type) := {
    load:    memory_chunk → mem → block → Z → option val;
     store:   memory_chunk → mem → block → Z → val → option mem;
     …
}

Definition extcall_sem {mem: Type} {mem_ops: MemoryOps mem}: Type  :=
   forall (F V: Type), Genv.t F V → list val → mem → trace → val → mem → Prop.
```

And that's all !
CompCertX itself knows nothing
about the abstract state...

Memory model and abstract state

```
Class MemoryOps (mem: Type) := {
    load:    memory_chunk → mem → block → Z → option val;
    store:   memory_chunk → mem → block → Z → val → option mem;
    …
}

Class AbstractStateOps (data mwd: Type) := {
    get_abstract_data: mwd → data;
    put_abstract_data: data → mwd → data;
    …
}
```

Memory model and abstract state

```
Class MemoryOps (mem: Type) := {
    load:    memory_chunk → mem → block → Z → option val;
    store:   memory_chunk → mem → block → Z → val → option mem;
    …
}

Class AbstractStateOps (data mwd: Type) := {
    get_abstract_data: mwd → data;
    put_abstract_data: data → mwd → data;
    …
}

Global Instance mem_with_data_mem:
  forall (mem data: Type),
         MemoryOps mem → MemoryOps (mem * data)
:= …

Global Instance mem_with_data_abstract_state:
  forall (mem data: Type),
         AbstractStateOps data (mem * data)
:= …
```

Lenses

CompCert Clight initial and final states

- Whole-machine semantics
- Initial state:
  - Call "main"
  - No arguments
  - Empty memory (except initialized globals)
- Final state:
  - Return an integer
  - Memory is not relevant

```
Inductive initial_state (p: program):
state → Prop := initial_state_intro:
    forall b f m0,
    let ge := Genv.globalenv p in
    Genv.init_mem p = Some m0 →
    Genv.find_symbol ge p.(prog_main) = Some b →
    Genv.find_funct_ptr ge b = Some f →
    type_of_fundef f =
     Tfunction Tnil type_int32s cc_default →
    initial_state p (Callstate f nil Kstop m0).
```

```
Inductive final_state:
  state → int → Prop := final_state_intro:
    forall r m,
    final_state (Returnstate (Vint r) Kstop m) r.
```

Clight vs. ClightX initial and final states

- Whole-machine semantics
- Initial state:
  - Call "main"
  - No arguments
  - Empty memory (except initialized globals)
- Final state:
  - Return an integer
  - Memory is not relevant

- Per-module semantics for a function *i* with arguments *args* and memory *m*
- Initial state :
  - Call the function i
  - With arguments args
  - And the memory m
- Final state
  - Return a value
  - Memory is relevant

ClightX initial and final states

- Per-function semantics for a function *i* with arguments *args* and memory *m*
- Initial state :
  - Call the function i
  - With arguments args
  - And the memory m
- Final state
  - Return a value
  - Memory is relevant

```
Inductive initial_state `{MemoryOps} (p: program)
(i: ident) (m: mem) (sg: signature) (args: list val):

state → Prop := initial_state_intro:
    forall b f targs tres tcc,
    Genv.find_symbol ge i = Some b →
    Genv.find_funct_ptr ge b = Some f →
    type_of_fundef f =
     Tfunction targs tres tcc →
    sg = signature_of_type targs tres tcc →
    initial_state p i m sg args
      (Callstate f args Kstop m).
```

```
Inductive final_state (sg: signature):
  state → val * mem → Prop := final_state_intro:
    forall v m,
    final_state (Returnstate v Kstop m) (v, m).
```

Our Contributions

- We introduce deep specification and present a language-based formalization of certified abstraction layer
- We developed new languages & tools in Coq
  - A formal layer calculus for composing certified layers
  - ClightX for writing certified layers in a C-like language
  - AsmX for writing certified layers in assembly
  - CompCertX that compiles ClightX layers into AsmX layers
- We built multiple certified OS kernels in Coq
  - mCertiKOS-hyper consists of 37 layers, took less than one-person-year to develop, and can boot Linux as a guest

Assembly code

- Low-level stack management
  - Context switching
  - Process creation
- ~200 lines of CertiKOS (out of >5k)

```
kctxt_switch:
        leal    0(%eax,%eax,2), %eax
        leal    KCtxtPool_LOC(,%eax,8), %eax
        movl    %esp, 0(%eax)
        movl    %edi, 4(%eax)
        movl    %esi, 8(%eax)
        movl    %ebx, 12(%eax)
        movl    %ebp, 16(%eax)
        popl    %ecx
        movl    %ecx, 20(%eax)
        leal    0(%edx,%edx,2), %edx
        leal    KCtxtPool_LOC(,%edx,8), %edx
        movl    0(%edx), %esp
        movl    4(%edx), %edi
        movl    8(%edx), %esi
        movl    12(%edx), %ebx
        movl    16(%edx), %ebp
        movl    20(%edx), %ecx
        pushl   %ecx
        xorl    %eax, %eax
        ret
```

AsmX: Layer primitives

- Support for two kinds of primitives:
  - C-style primitives with arguments and calling convention
  - assembly primitives with full register set
    (a la CompCert built-ins)

CompCert x86 Asm external function calls

- External functions + calling convention

```
Inductive external_call'
      (ef: external_function) (ge: Genv.t)
      (vargs: list val) (m1: mem) (t: trace) (vres: list val) (m2: mem) : Prop :=
external_call'_intro: forall v,
    external_call ef ge (decode_longs (sig_args (ef_sig ef)) vargs) m1 t v m2 →
    vres = encode_long (sig_res (ef_sig ef)) v →
    external_call' ef ge vargs m1 t vres m2.
```

```
Inductive step (ge: genv): state -> trace -> state -> Prop :=
| …
|  exec_step_external:
      forall b ef args res rs m t rs' m',
      rs PC = Vptr b Int.zero →
      Genv.find_funct_ptr ge b = Some (External ef) →
      extcall_arguments rs m (ef_sig ef) args →
      external_call' ef ge args m t res m' →
      rs' = (set_regs (loc_external_result (ef_sig ef)) res (rs #PC ← (rs RA) #RA ← Vundef) →
      step ge (State rs m) t (State rs' m')
| …
```

AsmX external function calls

- C-style externals

```
Inductive step {eco: ExtCallOps}                          (ge: genv):
 state -> trace -> state -> Prop :=
| …
|  exec_step_external:
      forall b ef args res rs m t rs' m',
      rs PC = Vptr b Int.zero →
      Genv.find_funct_ptr ge b = Some (External ef) →
      extcall_arguments rs m (ef_sig ef) args →
      external_call' ef ge args m t res m' →
      rs' = (set_regs (loc_external_result (ef_sig ef)) res (rs #PC ← (rs RA) #RA ← Vundef) →
      step ge (State rs m) t (State rs' m')




| …
```

AsmX external function calls

- C-style externals + assembly-style primitives

```
Definition primcall_sem {mem: Type} {mem_ops: MemoryOps mem}: Type  :=
   Asm.genv → regset → mem → trace → regset → mem → Prop.

 Class PrimCallOps := {
   primitive_call (ef: external_function): primcall_sem
 }.

 Inductive step {eco: ExtCallOps} {pco: PrimCallOps}  (ge: genv):
  state -> trace -> state -> Prop :=
 | …
 |  exec_step_external:
      forall b ef args res rs m t rs' m',
      rs PC = Vptr b Int.zero →
      Genv.find_funct_ptr ge b = Some (External ef) →
      extcall_arguments rs m (ef_sig ef) args →
      external_call' ef ge args m t res m'  →
      rs' = (set_regs (loc_external_result (ef_sig ef)) res (rs #PC ← (rs RA) #RA ← Vundef) →
      step ge (State rs m) t (State rs' m')
 | exec_step_primitive:
      forall b ef rs m t rs' m',
          rs PC = Vptr b Int.zero →
          Genv.find_funct_ptr ge b = Some (External ef) →
          primitive_call ef ge rs m t rs' m' →
          step ge (State rs m) t (State rs' m')
 | …
```

AsmX external function calls

- C-style externals + assembly-style primitives

```
Definition primcall_sem {mem: Type} {mem_ops: MemoryOps mem}: Type  :=
   Asm.genv → regset → mem → trace → regset → mem → Prop.

 Class PrimCallOps := {
   primitive_call (ef: external_function): primcall_sem
 }.

 Inductive step {eco: ExtCallOps} {pco: PrimCallOps}  (ge: genv):
  state -> trace -> state -> Prop :=
 | …
 |  exec_step_external:
      forall b ef args res rs m t rs' m',
      rs PC = Vptr b Int.zero →
      Genv.find_funct_ptr ge b = Some (External ef) →
      extcall_arguments rs m (ef_sig ef) args →
      external_call' ef ge args m t res m' →
      rs' = (set_regs (loc_external_result (ef_sig ef)) res (rs #PC ← (rs RA) #RA ← Vundef) →
      step ge (State rs m) t (State rs' m')
 | exec_step_primitive:
      forall b ef rs m t rs' m',
          rs PC = Vptr b Int.zero →
          Genv.find_funct_ptr ge b = Some (External ef) →
          primitive_call ef ge rs m t rs' m' →
          step ge (State rs m) t (State rs' m')
 | …
```

Keep both rules
to avoid distinguishing between
C layers and assembly layers

CompCert Asm initial and final states

- CompCert x86 Asm Whole-Program Semantics
- Initial state :
  - Call main
  - Empty memory (except initialized globals)
  - Empty register set
- Final state :
  - Return an integer
  - Memory and register sets are not relevant

```
Inductive initial_state (p: program): state -> Prop :=
  | initial_state_intro: forall m0,
      Genv.init_mem p = Some m0 →
      let ge := Genv.globalenv p in
      let rs0 :=
        (Pregmap.init Vundef)
        # PC <- (symbol_offset ge p.(prog_main) Int.zero)
        # RA <- Vzero
        # ESP <- Vzero in
      initial_state p (State rs0 m0).

Inductive final_state: state -> int -> Prop :=
  | final_state_intro: forall rs m r,
      rs#PC = Vzero →
      rs#EAX = Vint r →
      final_state (State rs m) r.
```

Asm vs. AsmX initial and final states

- CompCert x86 Asm Whole-Program Semantics
- Initial state :
  - Call main
  - Empty memory (except initialized globals)
  - Empty register set
- Final state :
  - Return an integer
  - Memory and register sets are not relevant

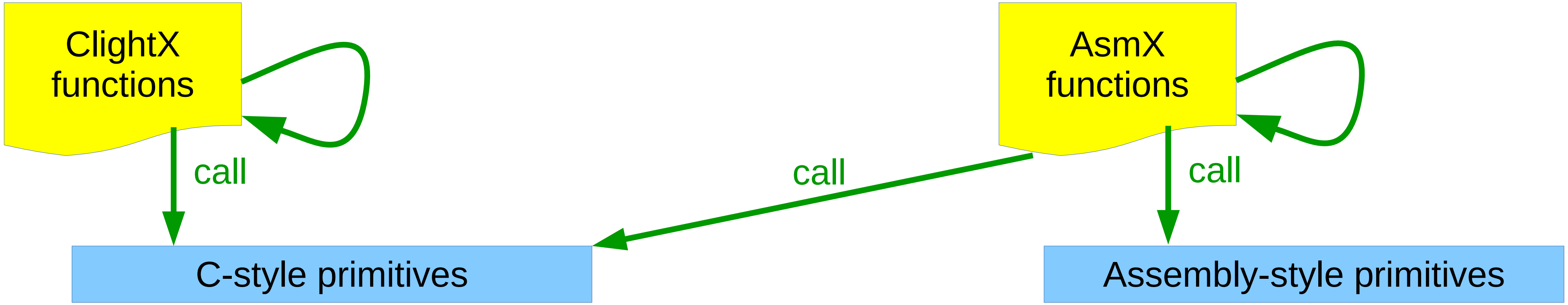
- AsmX per-module semantics for function *i* with register set *rs* and memory *m*
  - Initial state
    - Call function *i*
    - With memory *m*
    - And register set *rs*
  - Final state
    - Return a list of 32-bit values
    - Memory and register sets are relevant
- Function semantics can use :
  - C-style pattern with arguments and calling convention for CompCertX
  - Assembly-style pattern with no return values for the layer language
  - Whole memory and register set are relevant in both cases

AsmX initial and final states (C-style pattern)

- AsmX per-module semantics for function *i* with arguments *args*, register set *rs* and memory *m*
  - Initial state
    - Call function *i*
    - With arguments *args*
    - With memory *m*
    - And register set *rs*
  - Final state
    - Return a list of 32-bit values
    - Memory and register sets are relevant
  - Embed the C calling convention

```
Inductive initial_state (rs: regset) (p: Asm.program)
(i: ident) (sg: signature) (args: list val) (m: mem):
state -> Prop :=
| initial_state'_intro:  forall
    b
    (Hb: Genv.find_symbol (Genv.globalenv p) i = Some b)
    (Hpc: rs PC = Vptr b Int.zero)
    (Hargs: extcall_arguments rs m sg args),
  initial_state rs p i sg args m (State lm m).

Inductive final_state (rs0: regset) (sg: signature) : state ->
(list val * mem) -> Prop :=
  | final_state_intro:
      forall rs,
        (rs0 # RA) = (rs # PC) →
        (rs0 # ESP) = (rs # ESP) →
        forall v,
          v = List.map rs (loc_external_result sg) →
          forall
            (CALLEE_SAVE:
              forall r,
              ~ In r destroyed_at_call →
              Val.lessdef (rs0 (preg_of r)) (rs (preg_of r))),
          forall m_,
            final_state rs0 sg (State rs m_) (v, m_).
```

Functions and primitives



ClightX functions

AsmX functions

call

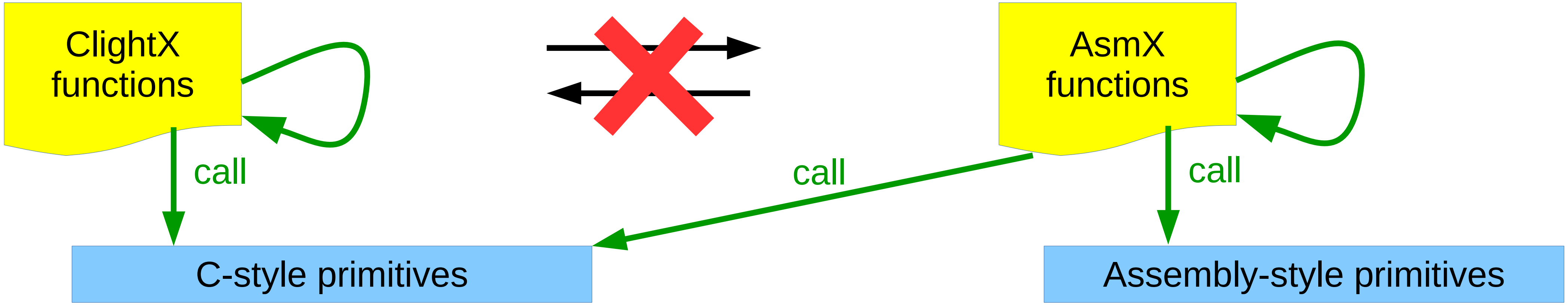
call

call

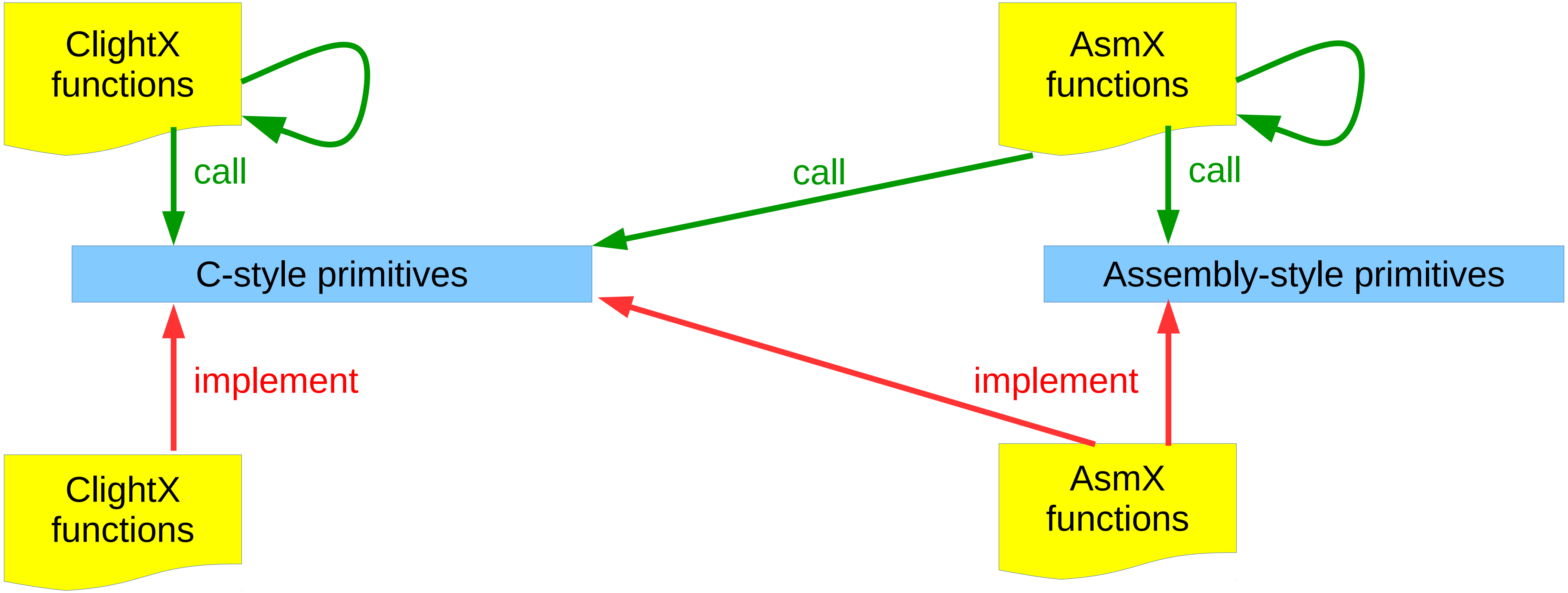
C-style primitives

Assembly-style primitives

Functions and primitives



Functions and primitives



ClightX functions

call

C-style primitives

implement

ClightX functions

AsmX functions

call

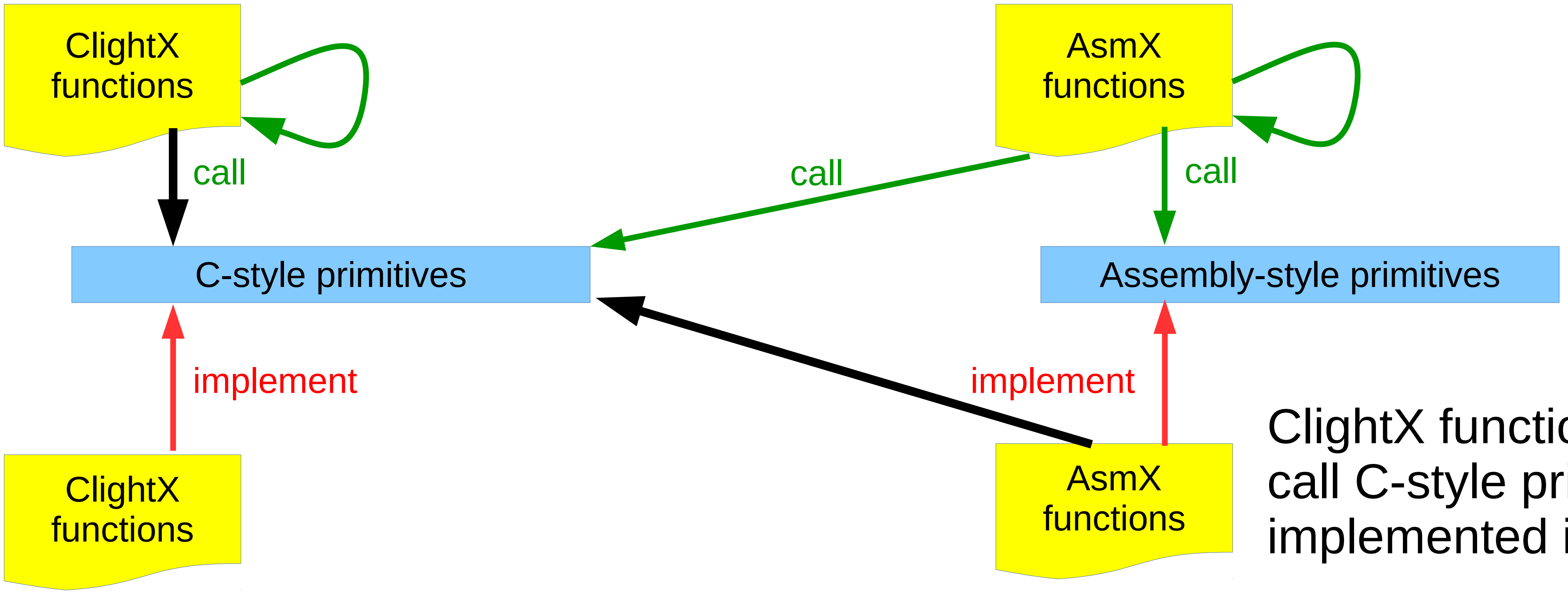
call

Assembly-style primitives

implement

AsmX functions

Functions and primitives



ClightX functions

call

AsmX functions

call

call

C-style primitives

Assembly-style primitives

implement

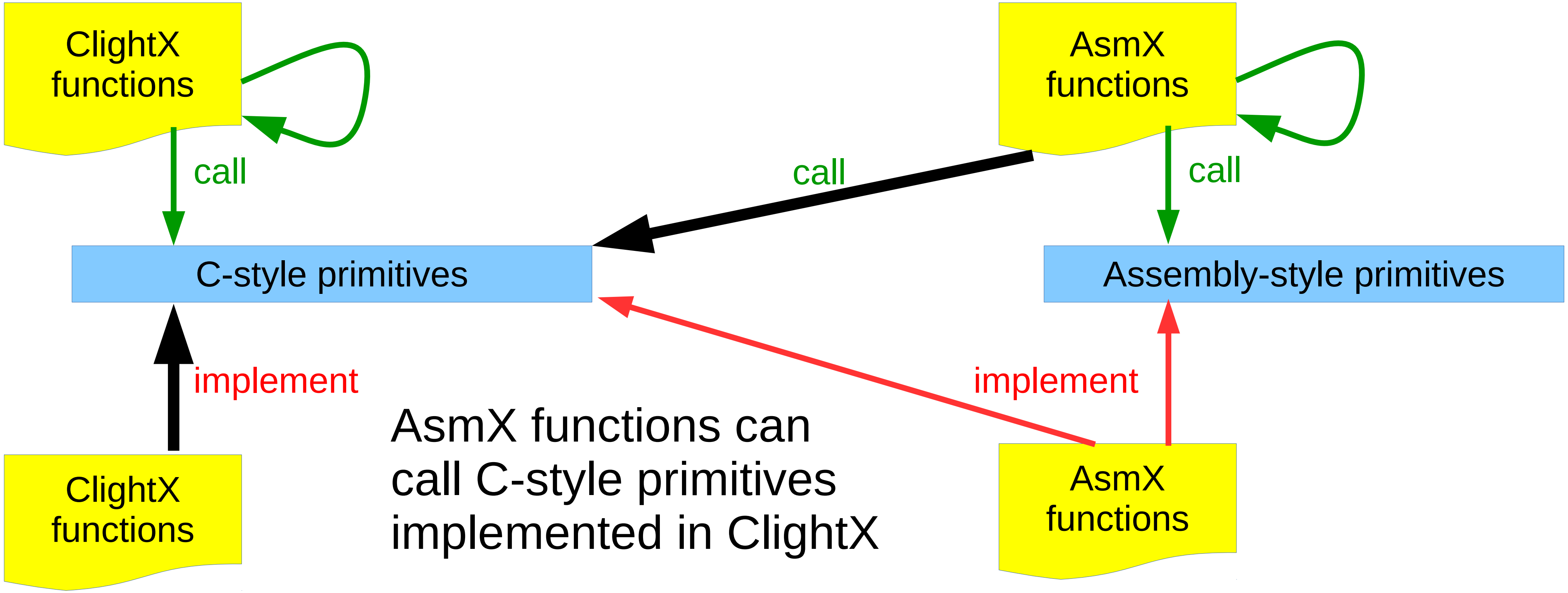
implement

ClightX functions

AsmX functions

ClightX functions can
call C-style primitives
implemented in AsmX

Functions and primitives



AsmX functions can
call C-style primitives
implemented in ClightX

POPL 2015 Mumbai, India January 12-18

Our Contributions

- We introduce deep specification and present a language-based formalization of certified abstraction layer
- We developed new languages & tools in Coq
  - A formal layer calculus for composing/linking certified layers
  - ClightX for writing certified layers in a C-like language
  - AsmX for writing certified layers in assembly
  - CompCertX that compiles ClightX layers into AsmX layers
- We built multiple certified OS kernels in Coq
  - mCertiKOS-hyper consists of 37 layers, took less than one-person-year to develop, and can boot Linux as a guest

Programming & Compiling Layers

ClightX

$$L \vdash_R M_c : L_1 \rightarrow L_1 \leq_R [\![M_c]\!]_{\text{ClightX}} (L)$$

Programming & Compiling Layers

ClightX

$$L \vdash_{R} M_c : L_1$$

$$L_1 \leq_{R} [\![\; M_c \;]\!]_{\text{ClightX}} (L)$$

CompCertX correctness theorem (where *minj* is a memory injection)

$$[\![\; M_c \;]\!]_{\text{ClightX}} (L) \leq_{minj} [\![\; \text{CompCertX}[L](M_c) \;]\!]_{\text{AsmX}} (L)$$

Programming & Compiling Layers

ClightX $L \vdash_{R} M_c : L_1$ → $L_1 \le_{R} [\![M_c]\!]_{\text{ClightX}}(L)$

↓

CompCertX correctness theorem (where *minj* is a memory injection)

$$[\![M_c]\!]_{\text{ClightX}}(L) \le_{minj} [\![\text{CompCertX[L]}(M_c)]\!]_{\text{AsmX}}(L)$$

↓

$$L_1 \le_{R \circ minj} [\![\text{CompCertX[L]}(M_c)]\!]_{\text{AsmX}}(L)$$

Programming & Compiling Layers

ClightX $L \vdash_R M_c : L_1$ → $L_1 \le_R \llbracket M_c \rrbracket_{\text{ClightX}}(L)$

↓

CompCertX correctness theorem (**where** ***minj*** **is a memory injection**)

$$\llbracket M_c \rrbracket_{\text{ClightX}}(L) \le_{minj} \llbracket \text{CompCertX[L]}(M_c) \rrbracket_{\text{AsmX}}(L)$$

↓

$$L_1 \le_{R \circ minj} \llbracket \text{CompCertX[L]}(M_c) \rrbracket_{\text{AsmX}}(L)$$

↓

R must absorb such memory injection: $R \circ minj = R$ *then we have:*

$$L_1 \le_R \llbracket \text{CompCertX[L]}(M_c) \rrbracket_{\text{AsmX}}(L)$$

Programming & Compiling Layers

ClightX $L \vdash_{R} M_c : L_1$ → $L_1 \leq_{R} \llbracket M_c \rrbracket_{\text{ClightX}}(L)$

CompCertX correctness theorem (where *minj* is a memory injection)

$$\llbracket M_c \rrbracket_{\text{ClightX}}(L) \leq_{minj} \llbracket \text{CompCertX[L]}(M_c) \rrbracket_{\text{AsmX}}(L)$$

$$L_1 \leq_{R \circ minj} \llbracket \text{CompCertX[L]}(M_c) \rrbracket_{\text{AsmX}}(L)$$

R must absorb such memory injection: $R \circ minj = R$ *then we have:*

$$L_1 \leq_{R} \llbracket \text{CompCertX[L]}(M_c) \rrbracket_{\text{AsmX}}(L)$$

Let $M_a = \text{CompCertX[L]}(M_c)$ then $L \vdash_{R} M_a : L_1$ **AsmX**

Programming & Compiling Layers

ClightX $L \vdash_R M_c : L_1$ → $L_1 \leq_R [\![\; M_c \;]\!]_{\text{ClightX}}(L)$

↓

CompCertX correctness theorem (where *minj* is a memory injection)

$$[\![\; M_c \;]\!]_{\text{ClightX}}(L) \leq_{minj} [\![\; \text{CompCertX[L]}(M_c) \;]\!]_{\text{AsmX}}(L)$$

↓

$$L_1 \leq_{R \circ minj} [\![\; \text{CompCertX[L]}(M_c) \;]\!]_{\text{AsmX}}(L)$$

↓

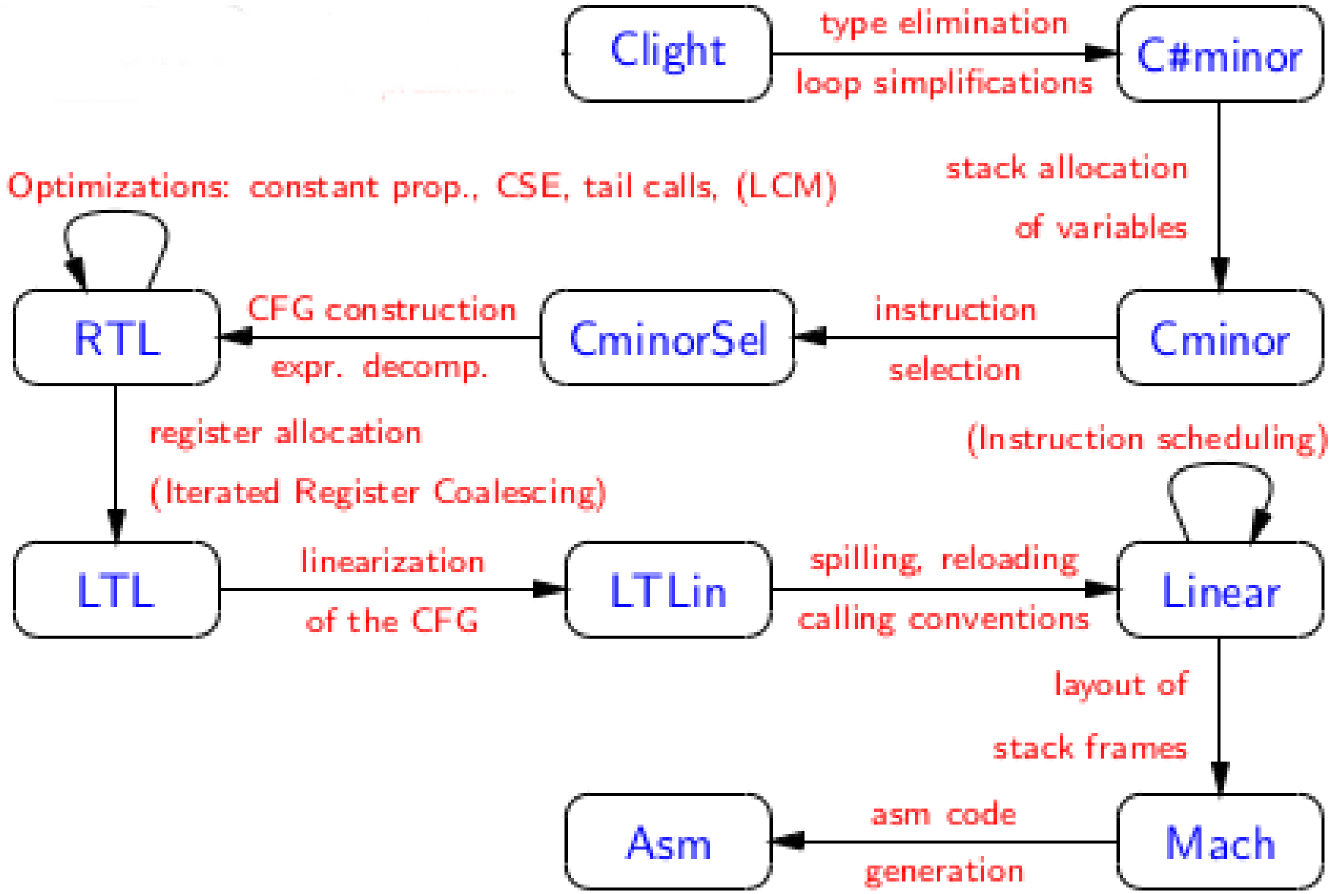
R must absorb such memory injection: $R \circ minj = R$ *then we have:*

$$L_1 \leq_R [\![\; \text{CompCertX[L]}(M_c) \;]\!]_{\text{AsmX}}(L)$$

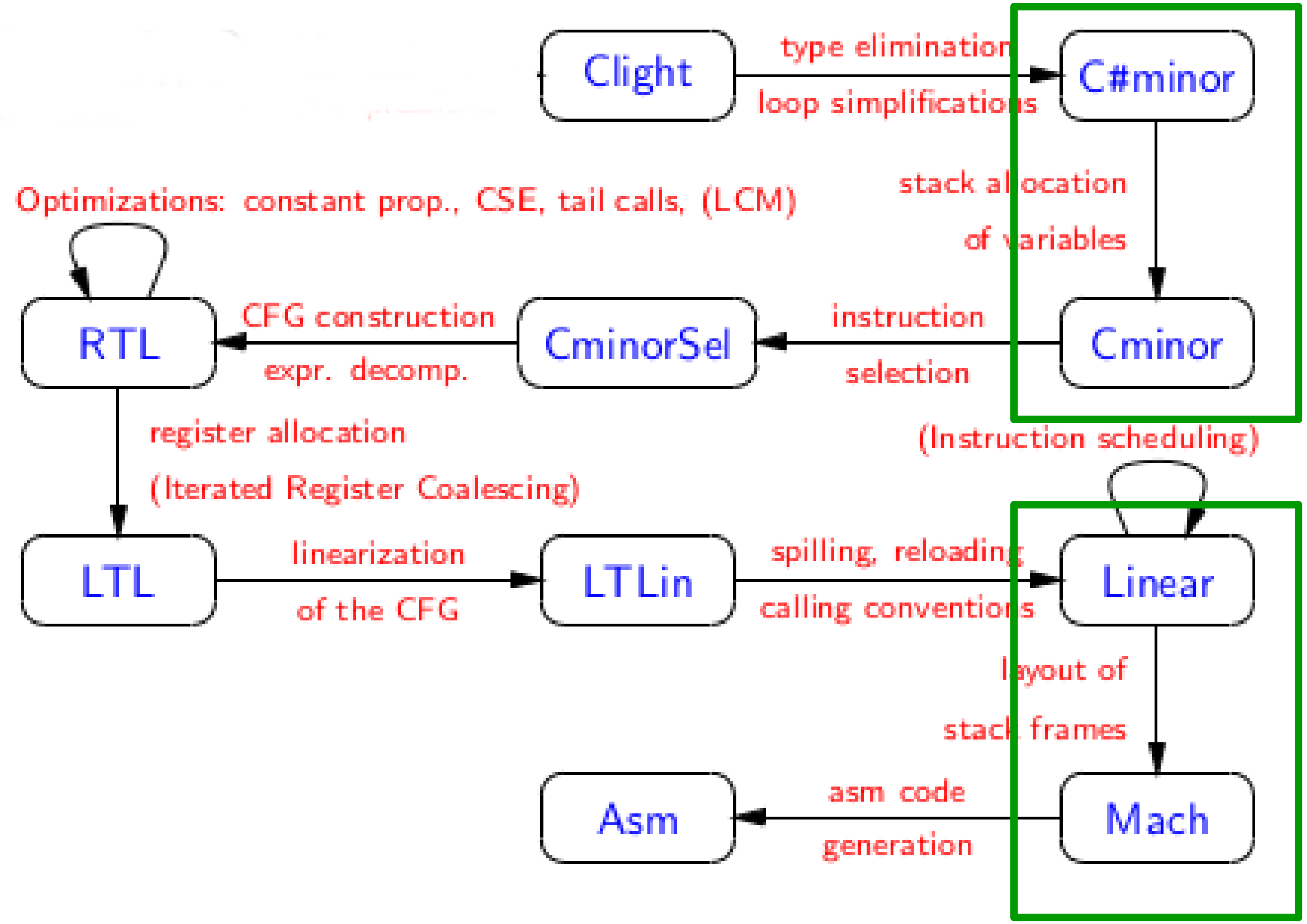
↓

Let $M_a = \text{CompCertX[L]}(M_c)$ then $L \vdash_R M_a : L_1$ **AsmX**

Reminder: CompCert compilation passes



Reminder: CompCert compilation passes



Memory injections

Assembly layer refinement proof

$$\boxed{L_1 \vdash_R M : L_2}$$

$$\begin{tikzcd}
\text{Assembly} & s2 \arrow[rr, "L_2(f)"] \arrow[d, dash, "R"'] & & s2' \arrow[d, dash, dotted, "R"] & \text{Assembly} \\
\text{Assembly} & s1 \arrow[rr, dotted, "\text{AsmX}", "⟦ M ⟧(L_1)(f)"'] & & s1' & \text{Assembly}
\end{tikzcd}$$

Layer refinement proof

$\boxed{L_1 \vdash_R M : L_2}$

$$\begin{tikzcd}
\text{Assembly} & s2 \arrow[r, "L_2(\mathbf{f})"] \arrow[d, dash, "R"'] & s2' \arrow[d, dash, dotted, "R"] & \text{Assembly} \\
\text{Assembly} & s1 \arrow[r, dotted, "{[\![M_c]\!] (L_1)(\mathbf{f})}"'] & s1'C &
\end{tikzcd}$$

ClightX ?

~~AsmX~~

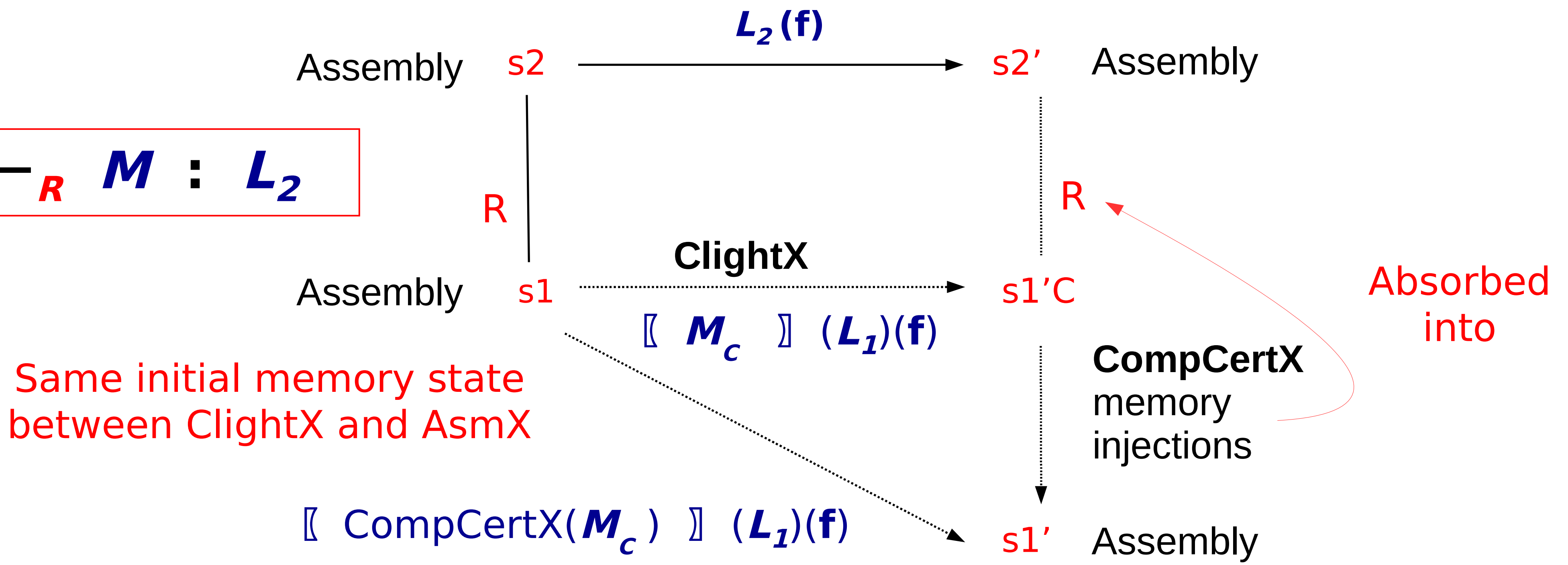
Layer refinement proof with CompCertX

$$\boxed{L_1 \vdash_R M : L_2}$$

$$\begin{tikzcd}
\text{Assembly} & s2 \arrow[rr, "L_2(\mathbf{f})"] \arrow[d, dash, "R"'] & & s2' \arrow[d, dotted, dash, "R"] & \text{Assembly} \\
? & s1' \arrow[rr, dotted, "\textbf{ClightX}", "\llbracket M_c \rrbracket (L_1)(\mathbf{f})"'] \arrow[d, dotted, "?"'] & & s1'C \arrow[d, dotted, "\textbf{CompCertX}\ \text{memory injections}"] & \\
\text{Assembly} & s1'? \arrow[rr, dotted, "\text{AsmX}", "\llbracket \text{CompCertX}(M_c) \rrbracket (L_1)(\mathbf{f})"'] & & s1' & \text{Assembly}
\end{tikzcd}$$

Layer refinement proof with CompCertX

$$\boxed{L_1 \vdash_R M : L_2}$$



$L_2(\mathbf{f})$

Assembly s2 → s2' Assembly

R

R

Assembly s1 → s1'C

ClightX

$⟦ M_c ⟧ (L_1)(\mathbf{f})$

CompCertX memory injections

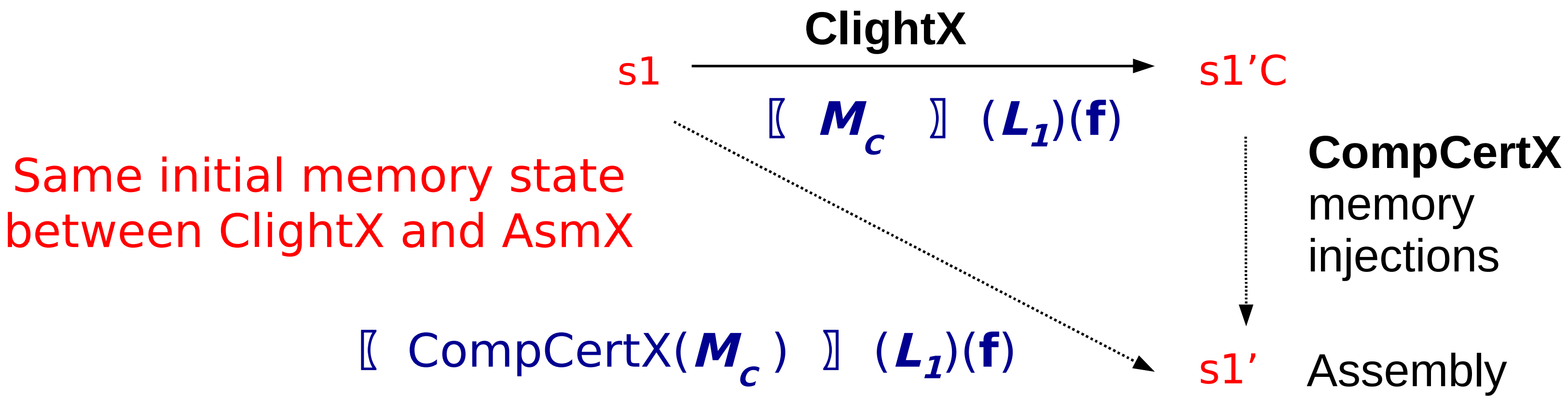
Absorbed into

Same initial memory state between ClightX and AsmX

$⟦ \text{CompCertX}(M_c) ⟧ (L_1)(\mathbf{f})$

s1' Assembly

CompCertX correctness statement



s1

ClightX

s1'C

$⟦ \boldsymbol{M_c} ⟧ (\boldsymbol{L_1})(\mathbf{f})$

Same initial memory state
between ClightX and AsmX

CompCertX
memory
injections

$⟦ \text{CompCertX}(\boldsymbol{M_c}) ⟧ (\boldsymbol{L_1})(\mathbf{f})$

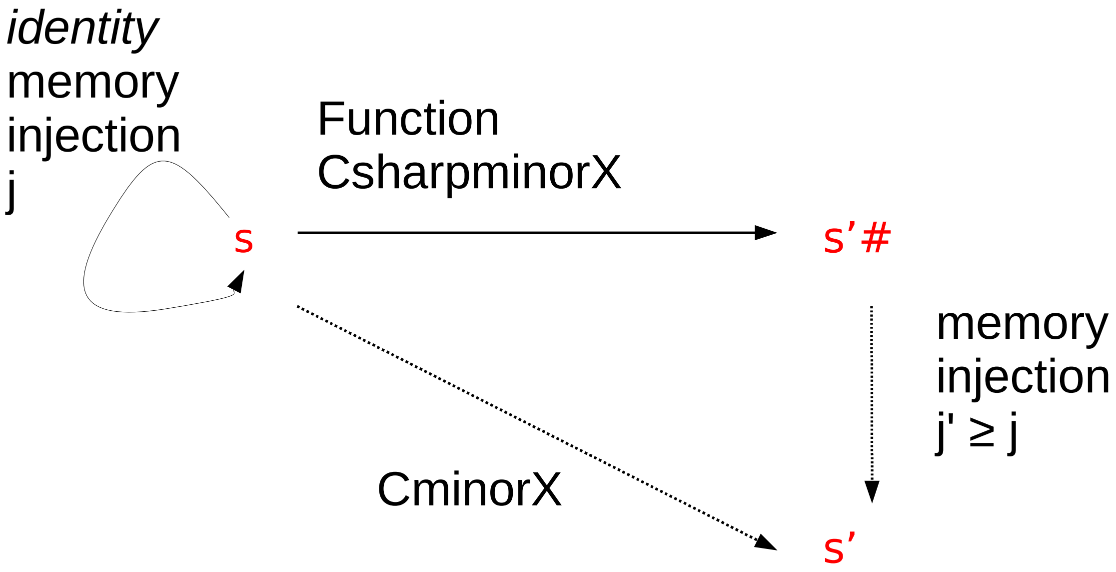
s1'

Assembly

Example: CsharpminorX to CminorX correctness proof

$$\begin{tikzcd}
s\# \arrow[r, "\text{One single step CsharpminorX}"] \arrow[d, "\text{memory injection } j"'] & s'\# \arrow[d, dotted, "\text{memory injection } j' \geq j"] \\
s \arrow[r, dotted, "\text{CminorX}"] & s'
\end{tikzcd}$$

Example: CsharpminorX to CminorX correctness proof



identity memory injection j

Function CsharpminorX

s

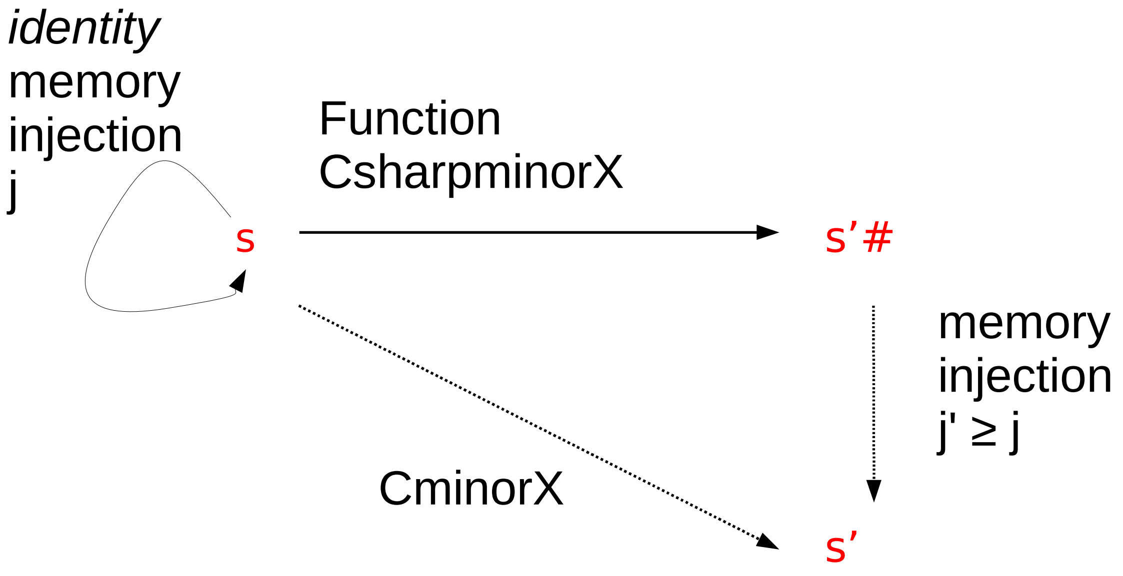
s'#

memory injection j' ≥ j

CminorX

s'

Example: CsharpminorX to CminorX correctness proof



identity
memory
injection
j

s

Function
CsharpminorX

s′#

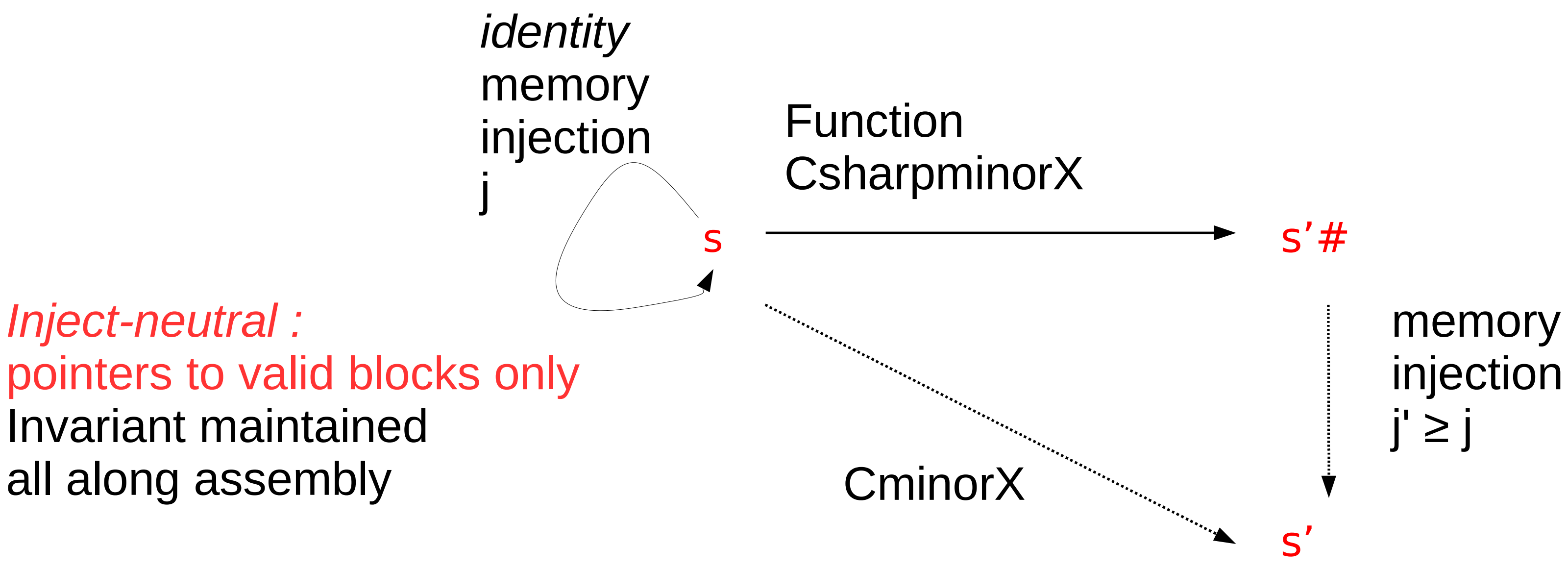
memory
injection
j' ≥ j

CminorX

s′

Wrap final state
into ∃ j' ≥ j

Example: CsharpminorX to CminorX correctness proof



identity
memory
injection
j

s

Function
CsharpminorX

s’#

Inject-neutral :
pointers to valid blocks only
Invariant maintained
all along assembly

memory
injection
j' ≥ j

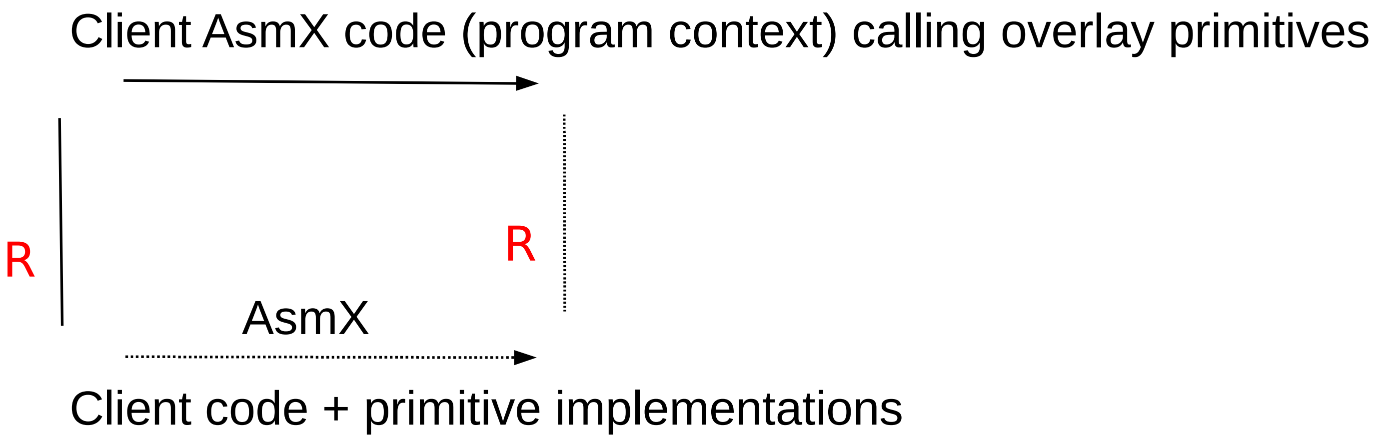
CminorX

s’

Requirements

- C-style layer primitives must abide by CompCert external function requirements :
  - Stability by memory injection, extension, etc.
  - Must preserve *inject-neutral* (no pointers to invalid blocks created)
- Also needed for assembly-style primitives
  - Layer refinement relation also includes a memory injection
  - Accumulates all memory injections due to calls to code compiled by CompCertX

Layer refinement and memory injections

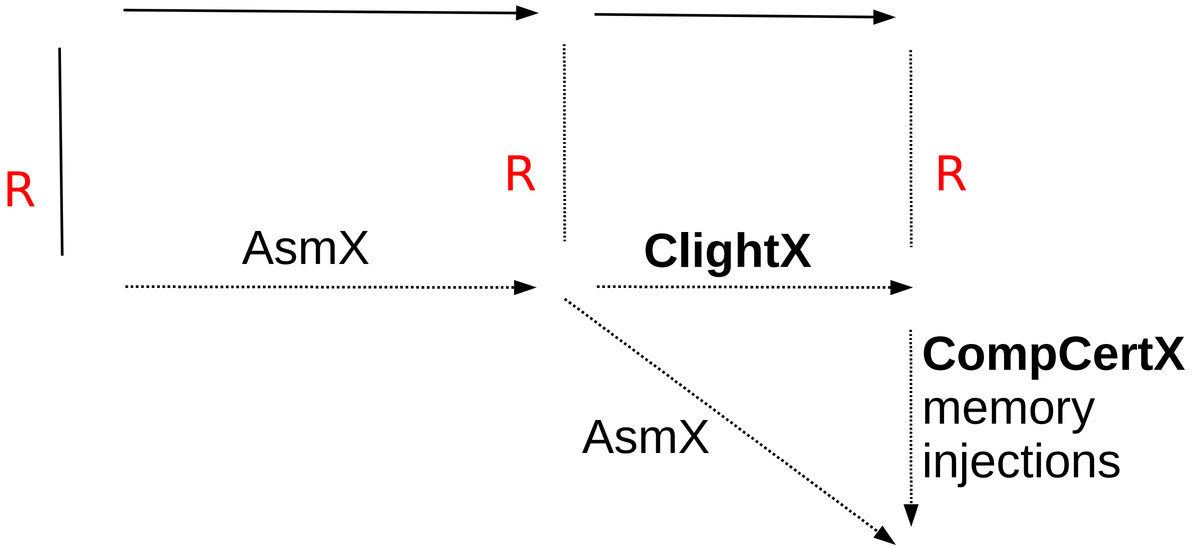


Client AsmX code (program context) calling overlay primitives

Client code + primitive implementations

Layer refinement and memory injections

Client AsmX code (program context) calling overlay primitives



R

R

R

AsmX

ClightX

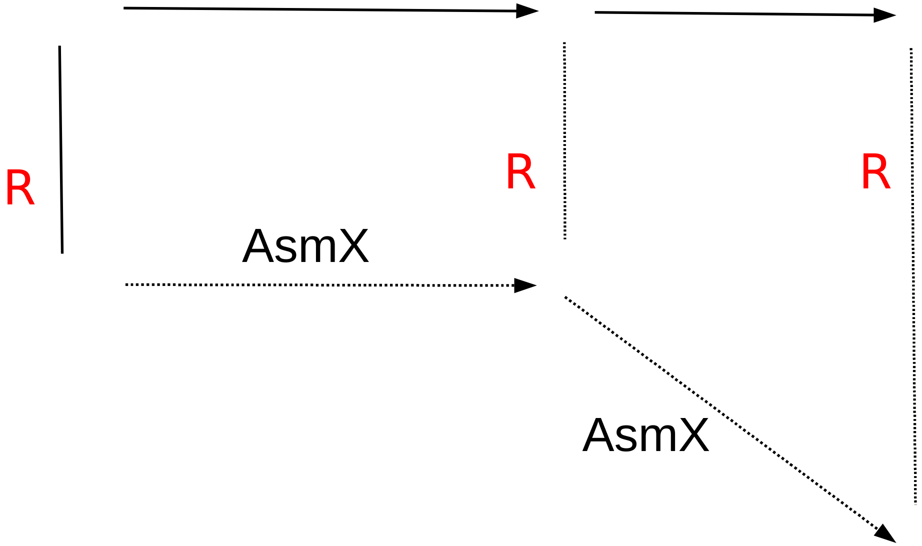
CompCertX
memory
injections

AsmX

Client code + primitive implementations

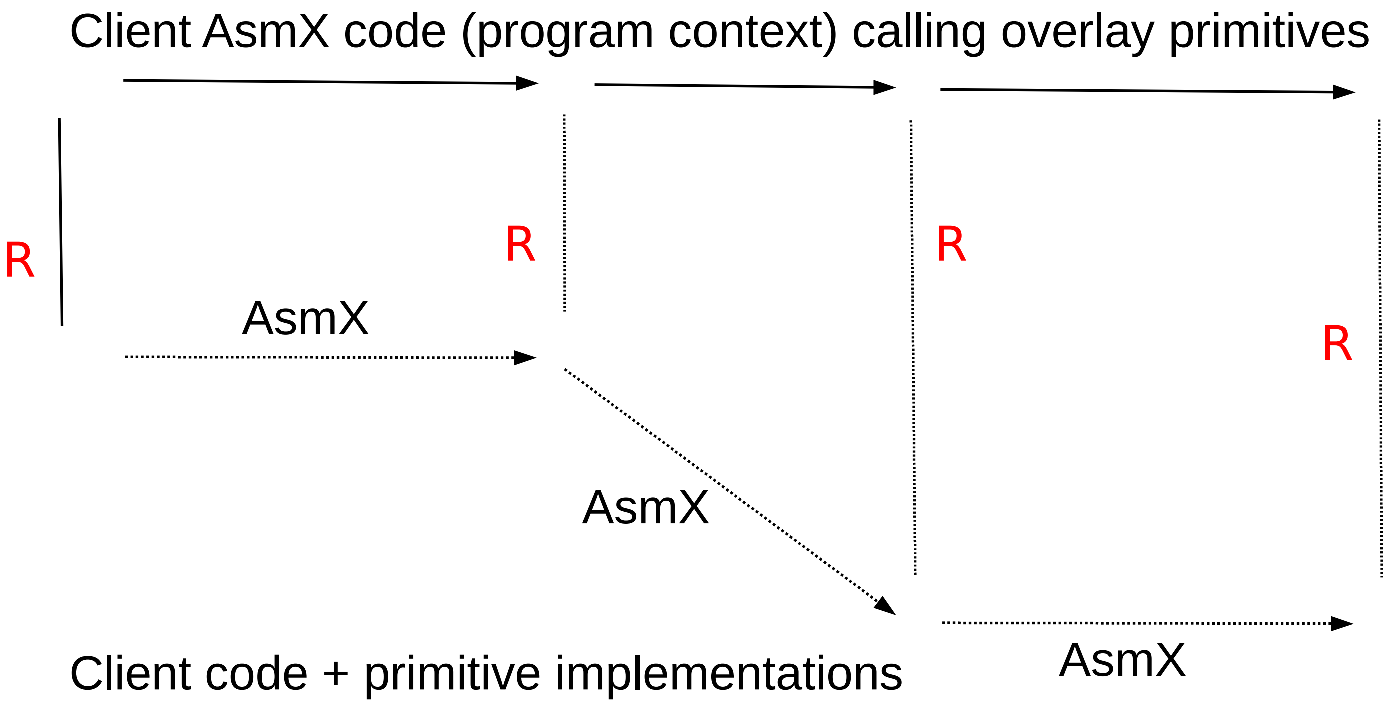
Layer refinement and memory injections

Client AsmX code (program context) calling overlay primitives



Client code + primitive implementations

Layer refinement and memory injections



Client AsmX code (program context) calling overlay primitives

R

R

R

AsmX

R

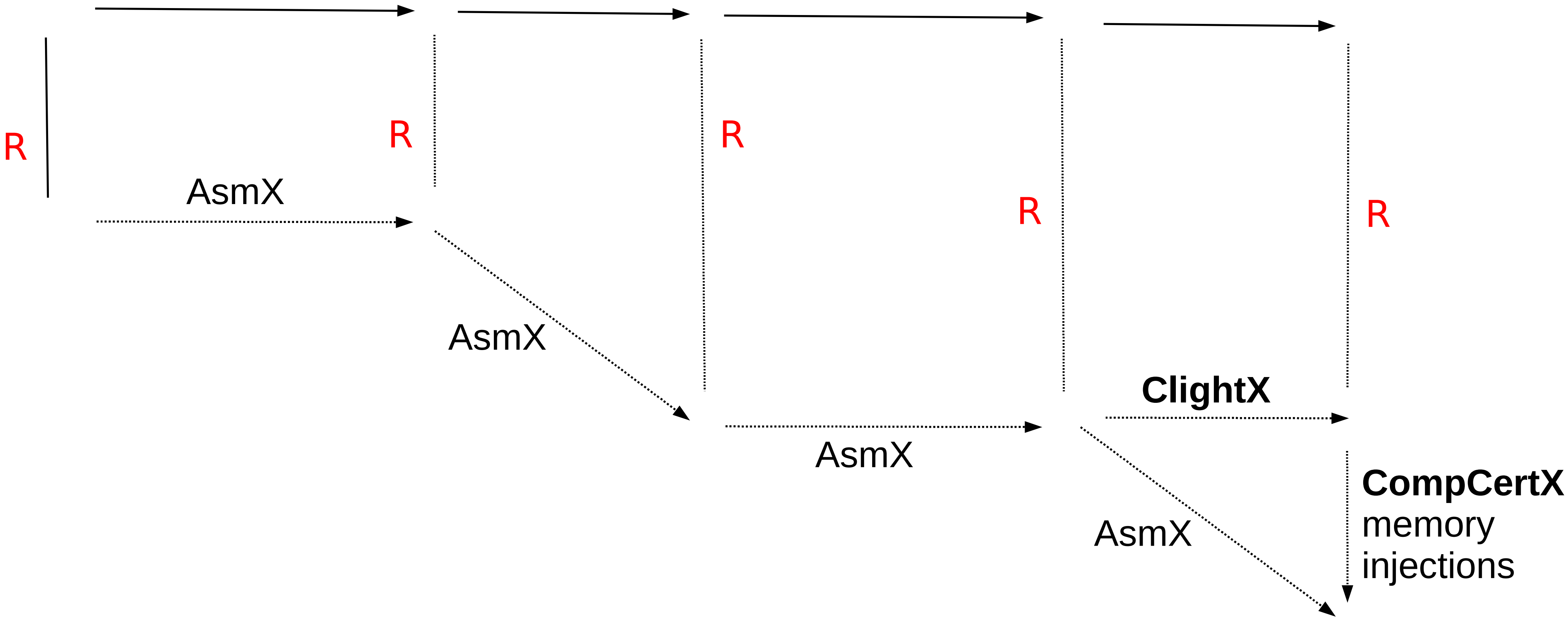
AsmX

Client code + primitive implementations

AsmX

Layer refinement and memory injections

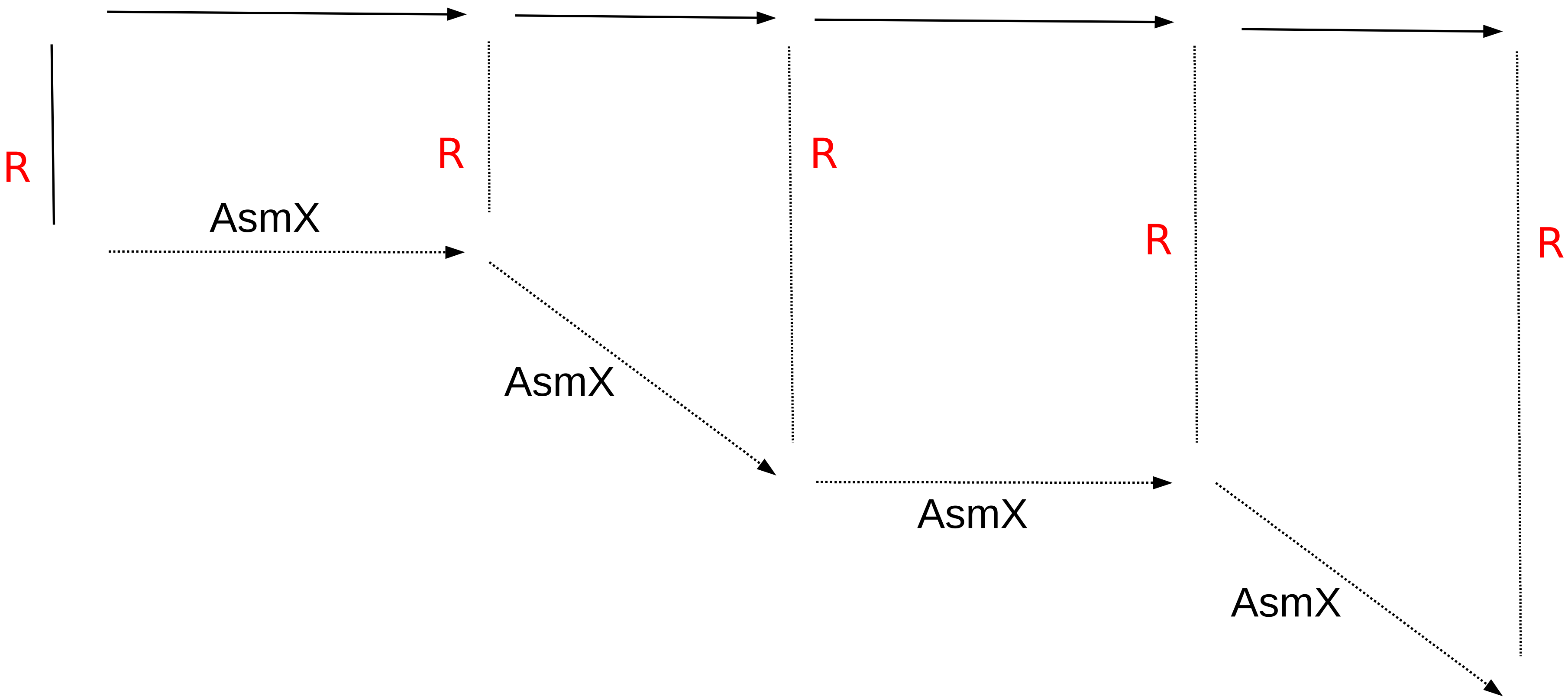
Client AsmX code (program context) calling overlay primitives



Client code + primitive implementations

Layer refinement and memory injections

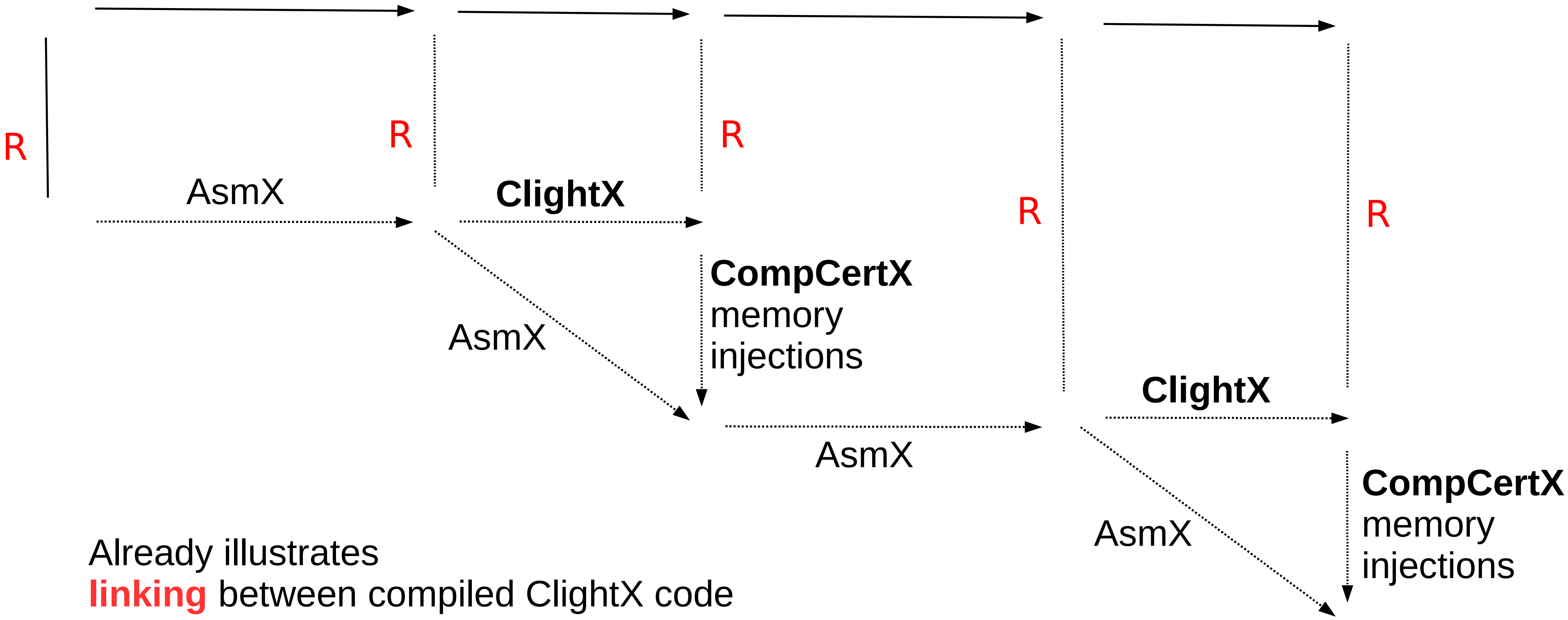
Client AsmX code (program context) calling overlay primitives



Client code + primitive implementations

Layer refinement and memory injections

Client AsmX code (program context) calling overlay primitives



R

R

AsmX

R

ClightX

CompCertX
memory
injections

AsmX

AsmX

R

R

ClightX

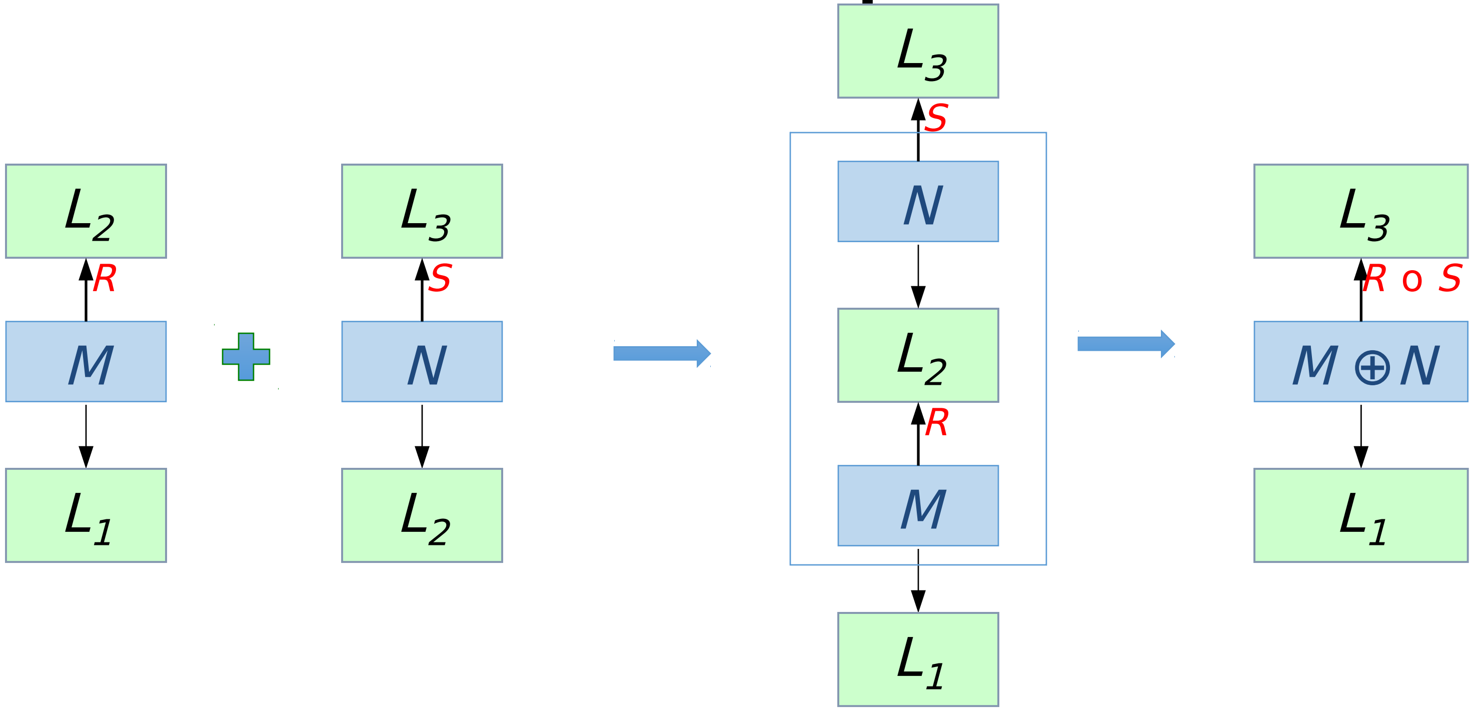
CompCertX
memory
injections

AsmX

Already illustrates
linking between compiled ClightX code
and AsmX program context

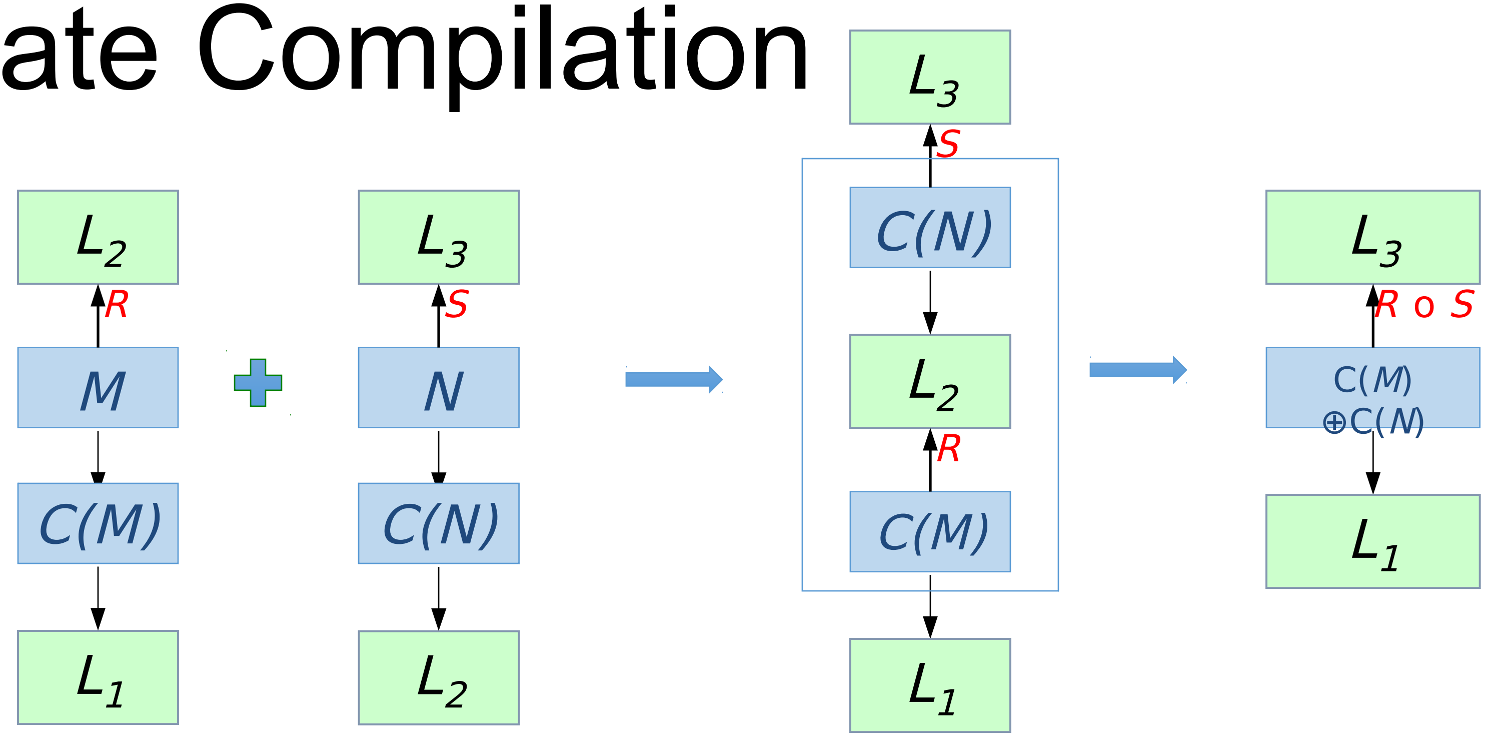
Client code + primitive implementations

LayerLib: Vertical Composition



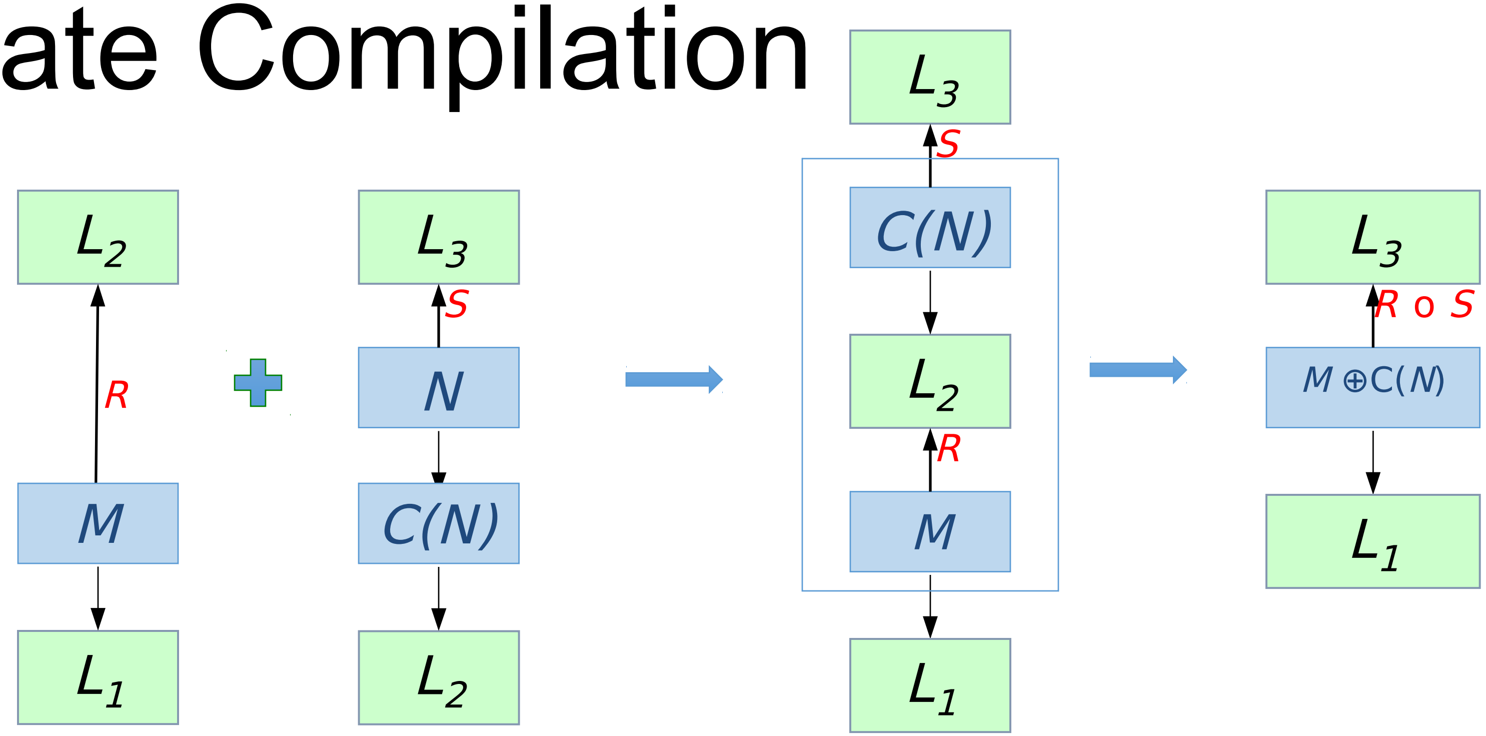
$$\frac{L_1 \vdash_R M : L_2 \qquad\qquad L_2 \vdash_S N : L_3}{L_1 \vdash_{R \circ S} M \oplus N : L_3}\ \textsc{Vcomp}$$

LayerLib: Vertical Composition and Separate Compilation



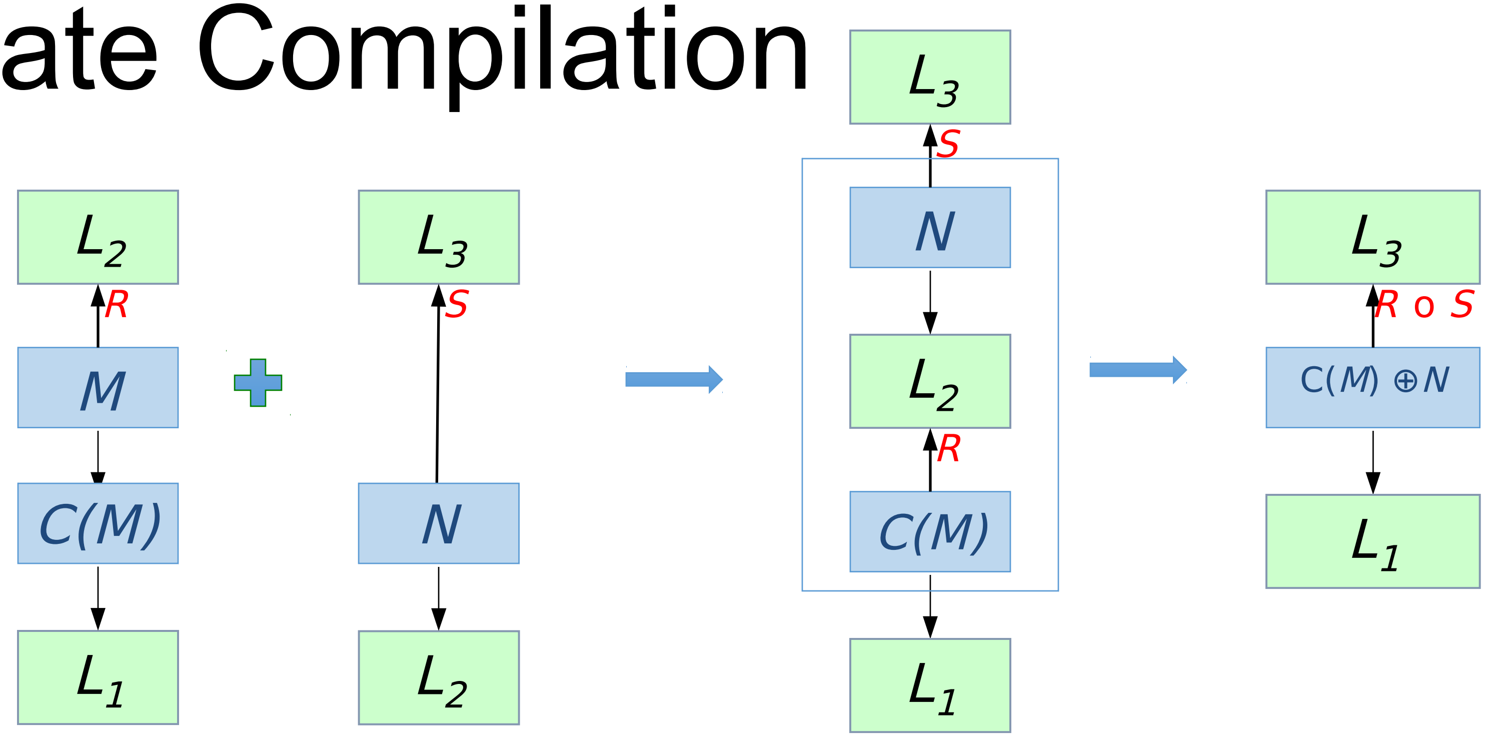
LayerLib: Vertical Composition and Separate Compilation

ClightX code N
« calls »
AsmX code M
through L2
primitives

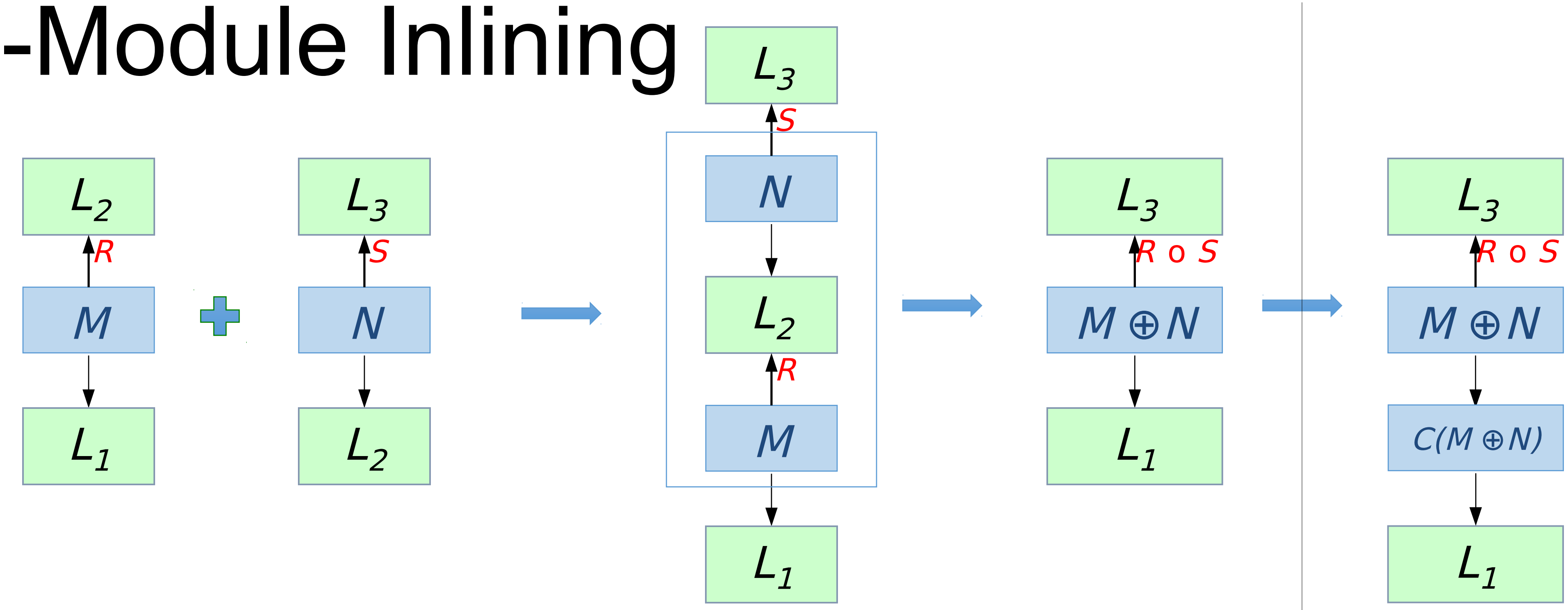


LayerLib: Vertical Composition and Separate Compilation

AsmX code N
« calls »
ClightX code M
through L2
primitives



LayerLib: Vertical Composition and Cross-Module Inlining

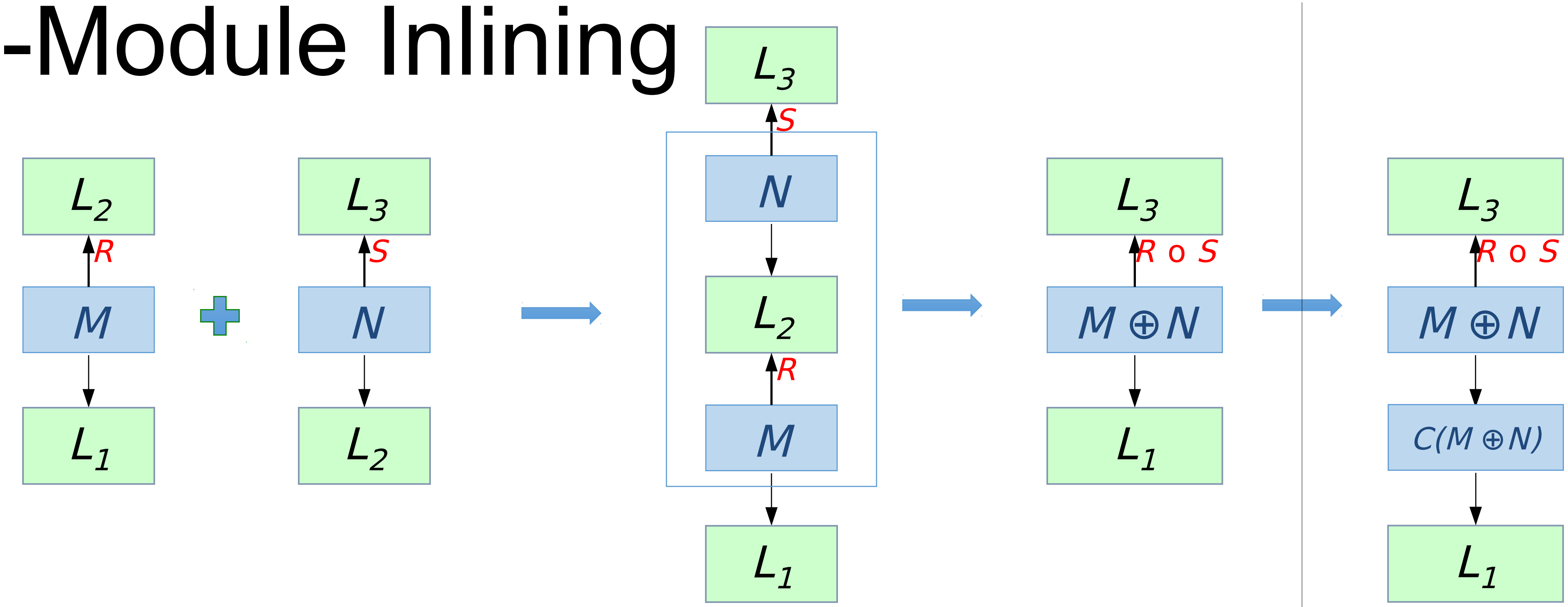


This is work in progress.

C layer refinement

Assembly layer refinement

LayerLib: Vertical Composition and Cross-Module Inlining



Useful if :
- N concretizes high-level data into **low-level data**
- M concretizes low-level data into memory

C layer refinement

Assembly layer refinement

Example : Page maps

MPTOP

idpde_init : Initialize the kernel page map to identity page map

idpde_init.c

Abstract state

MPTIntro

set_IDPTE : Access (get/set) the page map

---

set_IDPTE.c

MALT

Concrete in-memory representation

Example: Page Tables

MPTOP

idpde_init.c

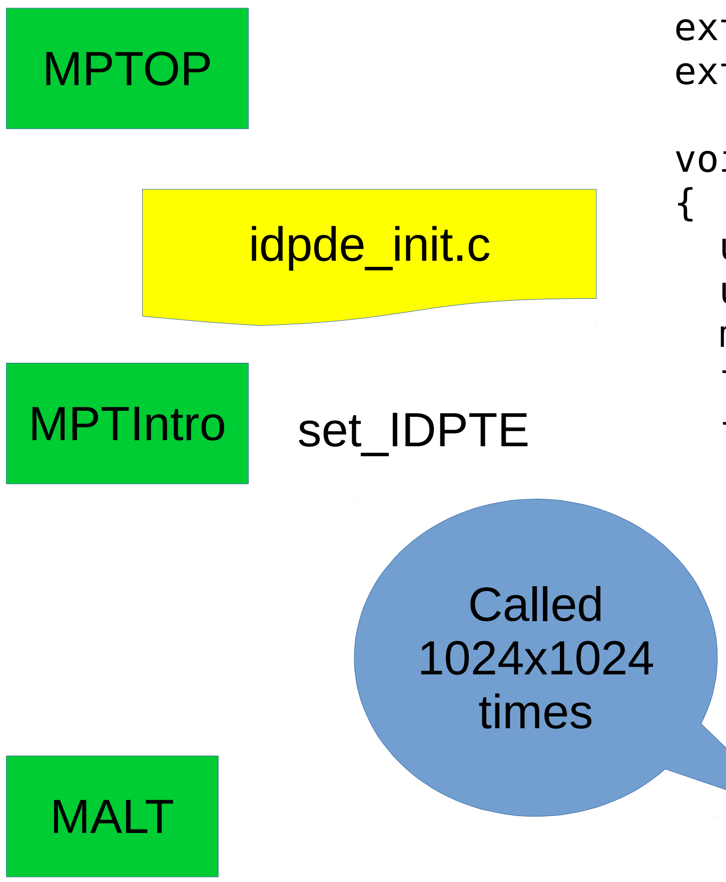
MPTIntro

set_IDPTE

MALT

```
#define PT_PERM_PTKF 3
#define PT_PERM_PTKT 259

extern void mem_init(unsigned int);
extern void set_IDPTE(unsigned int, unsigned int, unsigned int);

void idpde_init(unsigned int mbi_adr)
{
  unsigned int i, j;
  unsigned int perm;
  mem_init(mbi_adr);
  for(i = 0; i < 1024; i ++)
  {
    if (i < 256)
      perm = PT_PERM_PTKT;
    else if (i >= 960)
      perm = PT_PERM_PTKT;
    else
      perm = PT_PERM_PTKF;
    for(j = 0; j < 1024; j ++)
    {
      set_IDPTE(i, j, perm);
    }
  }
}
```

Example: Page Tables


MPTOP

idpde_init.c

MPTIntro

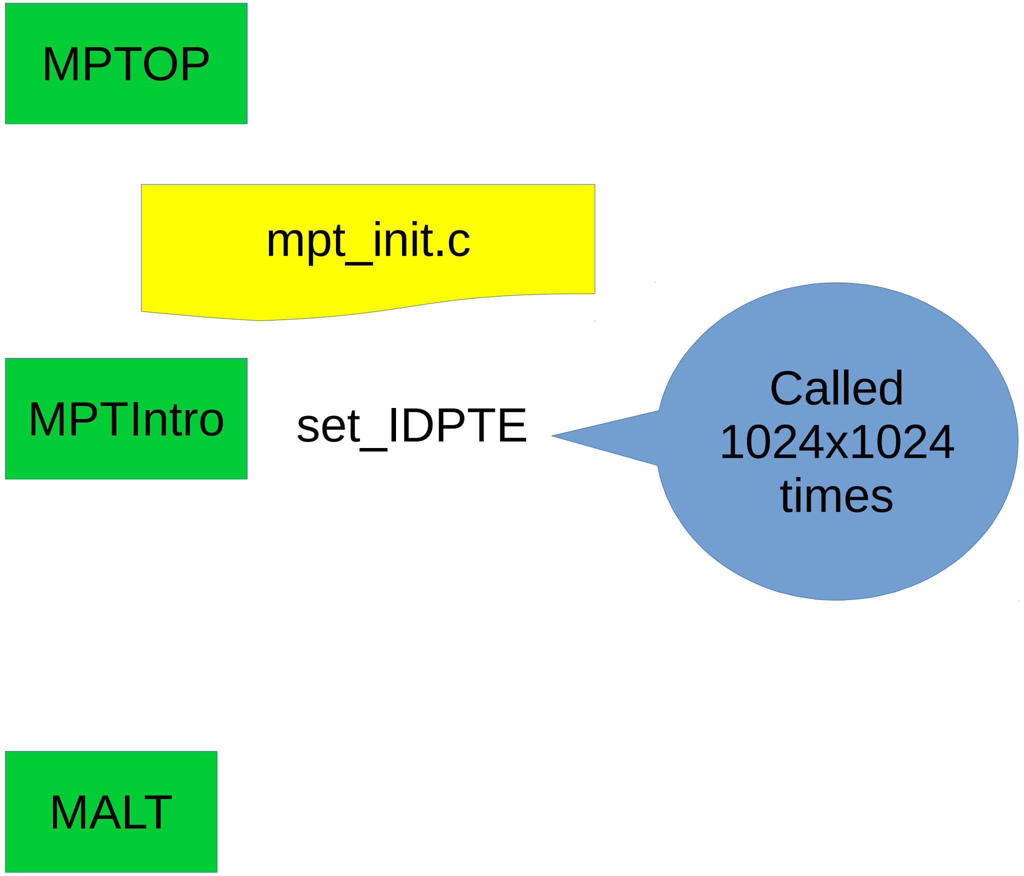
set_IDPTE

Called 1024x1024 times

MALT

```
#define PT_PERM_PTKF 3
#define PT_PERM_PTKT 259

extern void mem_init(unsigned int);
extern void set_IDPTE(unsigned int, unsigned int, unsigned int);

void idpde_init(unsigned int mbi_adr)
{
  unsigned int i, j;
  unsigned int perm;
  mem_init(mbi_adr);
  for(i = 0; i < 1024; i ++)
  {
    if (i < 256)
      perm = PT_PERM_PTKT;
    else if (i >= 960)
      perm = PT_PERM_PTKT;
    else
      perm = PT_PERM_PTKF;
    for(j = 0; j < 1024; j ++)
    {
      set_IDPTE(i, j, perm);
    }
  }
}
```

Example: Page Tables



MPTOP

mpt_init.c

MPTIntro

set_IDPTE

Called 1024x1024 times

MALT

Example: Page Tables

MPTOP

MPTIntro

set_IDPTE

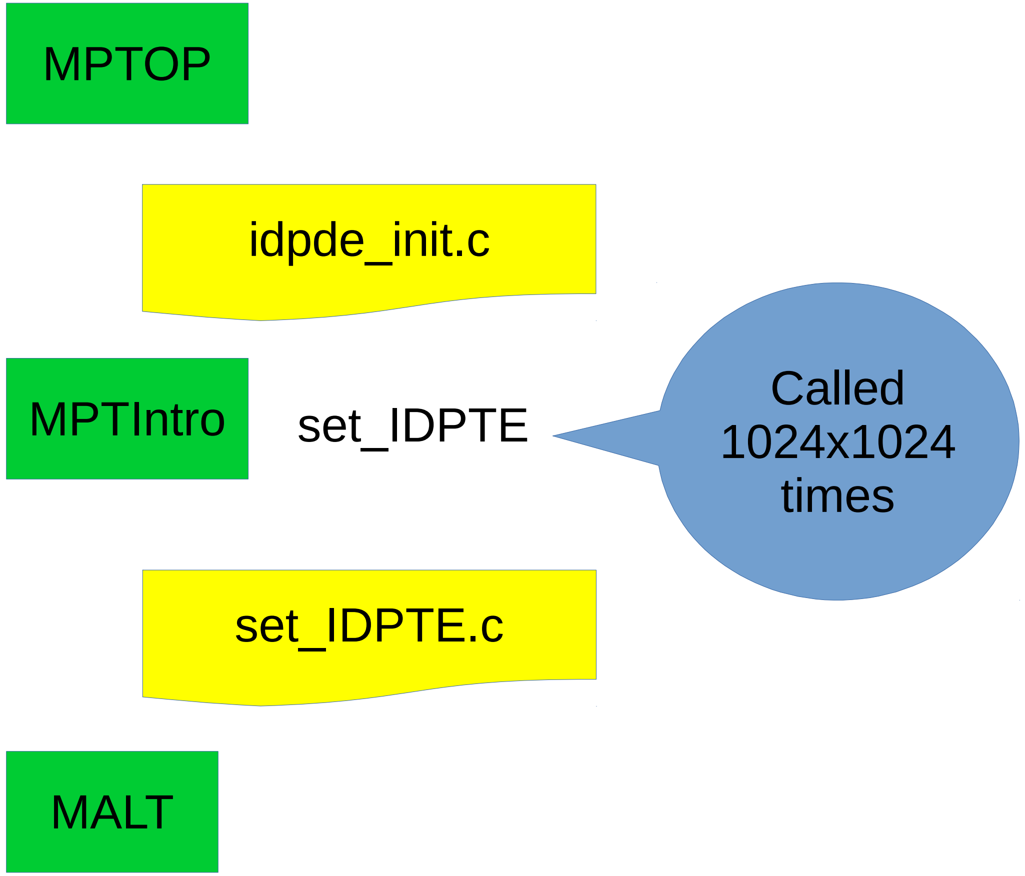
Called 1024x1024 times

set_IDPTE.c

MALT

```
extern unsigned int IDPMap_LOC[1024][1024];

     void set_IDPTE
     (unsigned int pde_index,
      unsigned int vadr, unsigned int perm)
     {
         IDPMap_LOC[pde_index][vadr] =
         (pde_index * 1024 + vadr) * 4096 + perm;
     }
```

Example: Page Tables



MPTOP

idpde_init.c

MPTIntro

set_IDPTE

Called 1024x1024 times

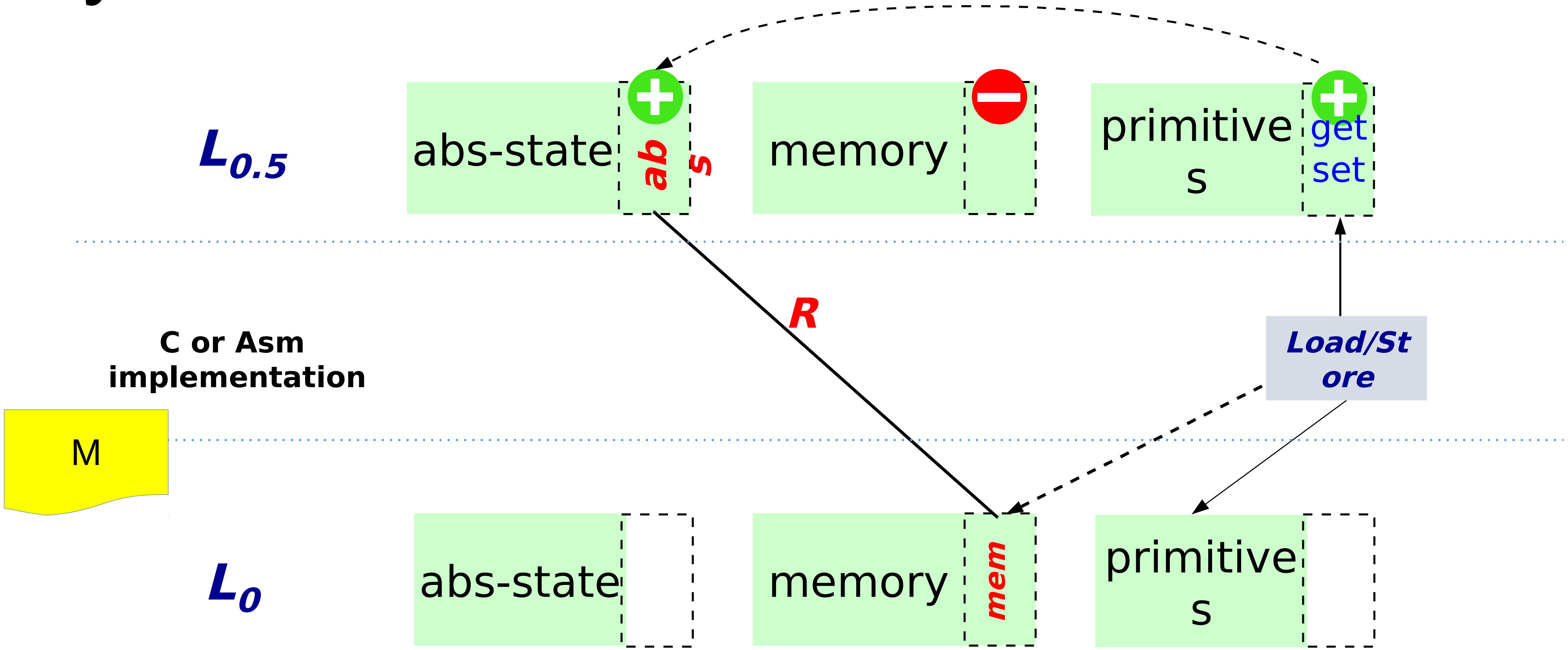
set_IDPTE.c

MALT

```
extern unsigned int IDPMap_LOC[1024][1024];

     void set_IDPTE
     (unsigned int pde_index,
      unsigned int vadr, unsigned int perm)
     {
         IDPMap_LOC[pde_index][vadr] =
         (pde_index * 1024 + vadr) * 4096 + perm;
     }
```

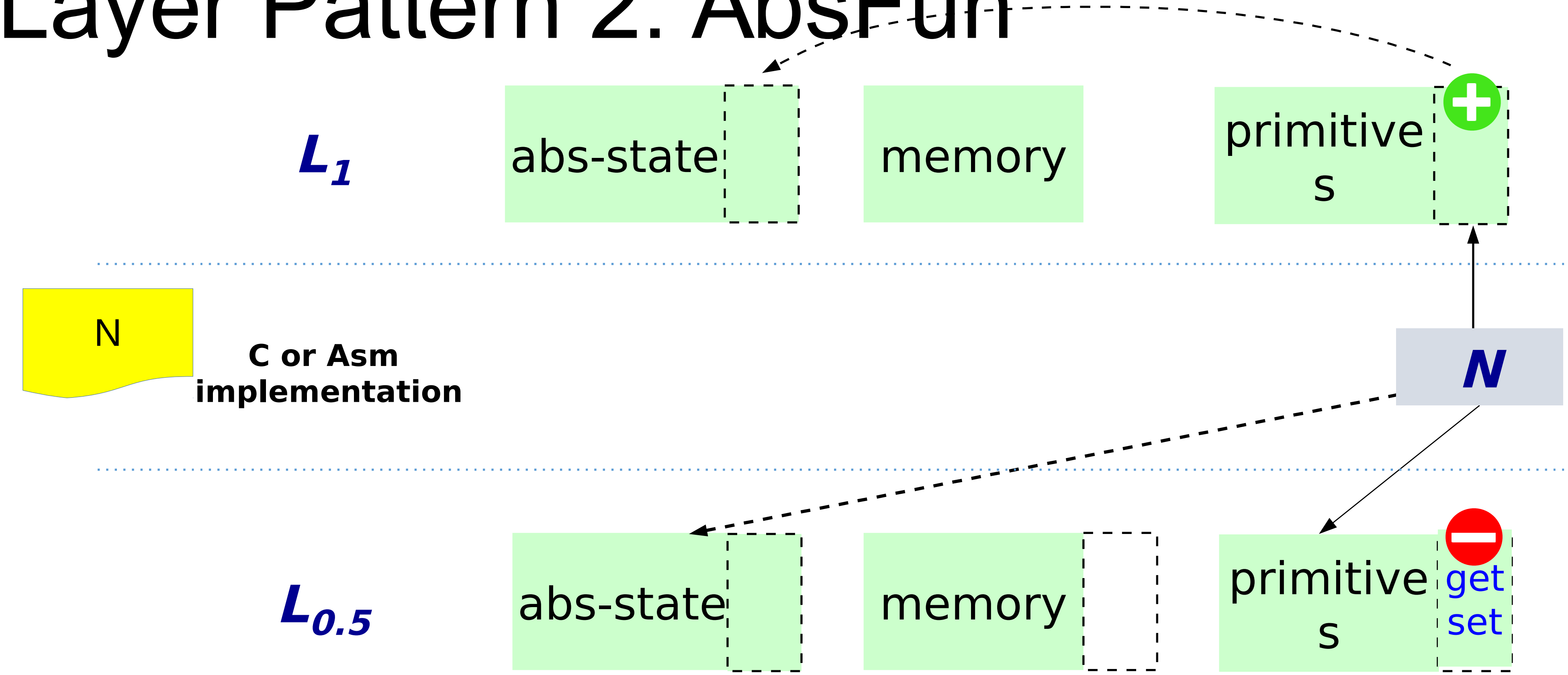
This function should be inlined in idpde_init.**s** !

Layer Pattern 1: Getter/Setter



Hide concrete memory; replace it with Abstract State
Only the getter and setter primitives can access memory

Layer Pattern 2: AbsFun



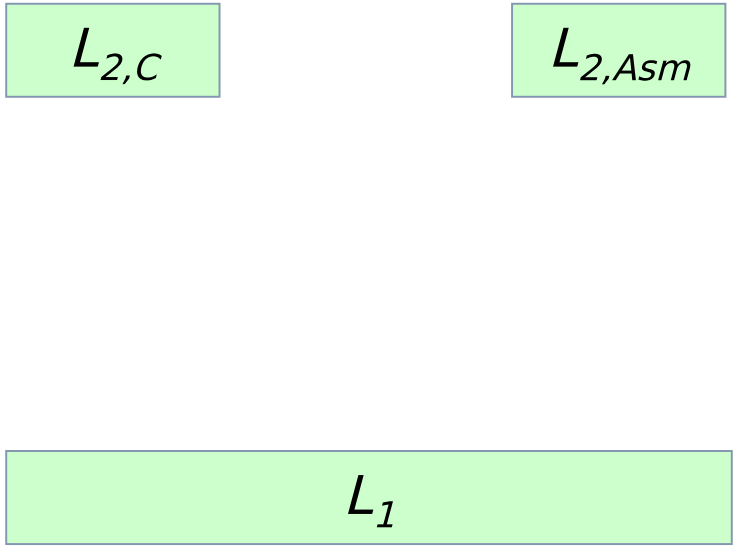
Memory does not change
New implementation code does not access memory directly!

Refinement proof : general pattern

L_2

L_1

Refinement proof : general pattern



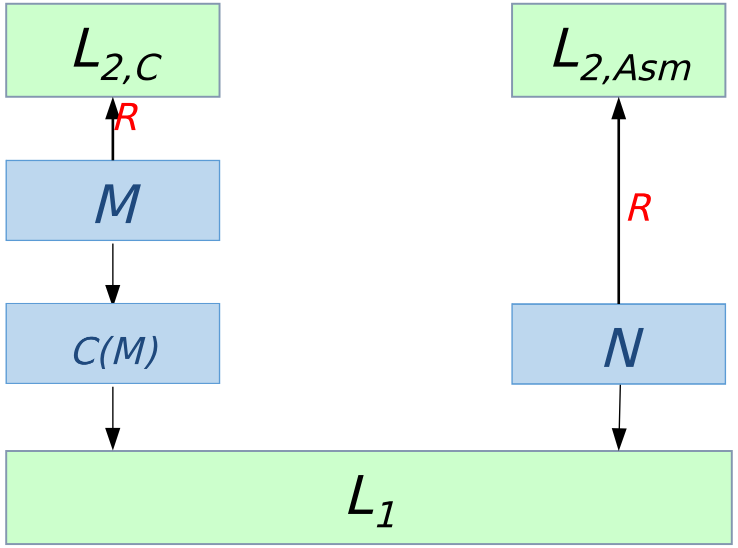
$L_{2,C}$

$L_{2,Asm}$

Same abstract state

L_1

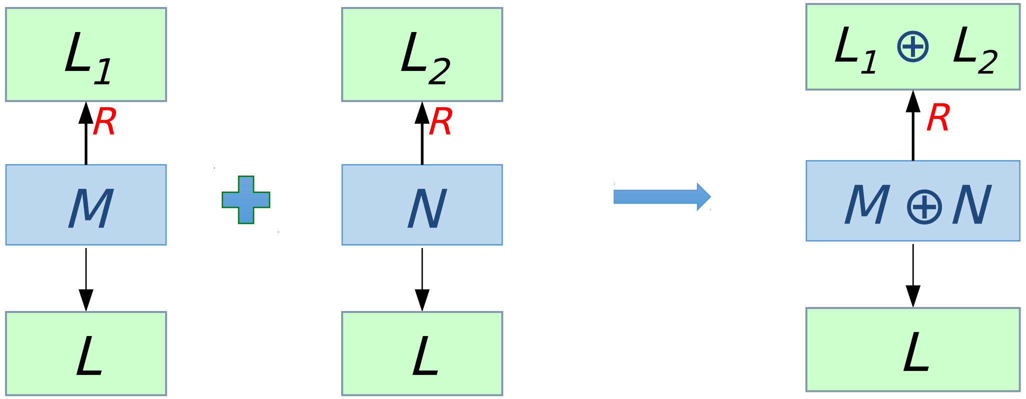
Refinement proof : general pattern



Same abstract state

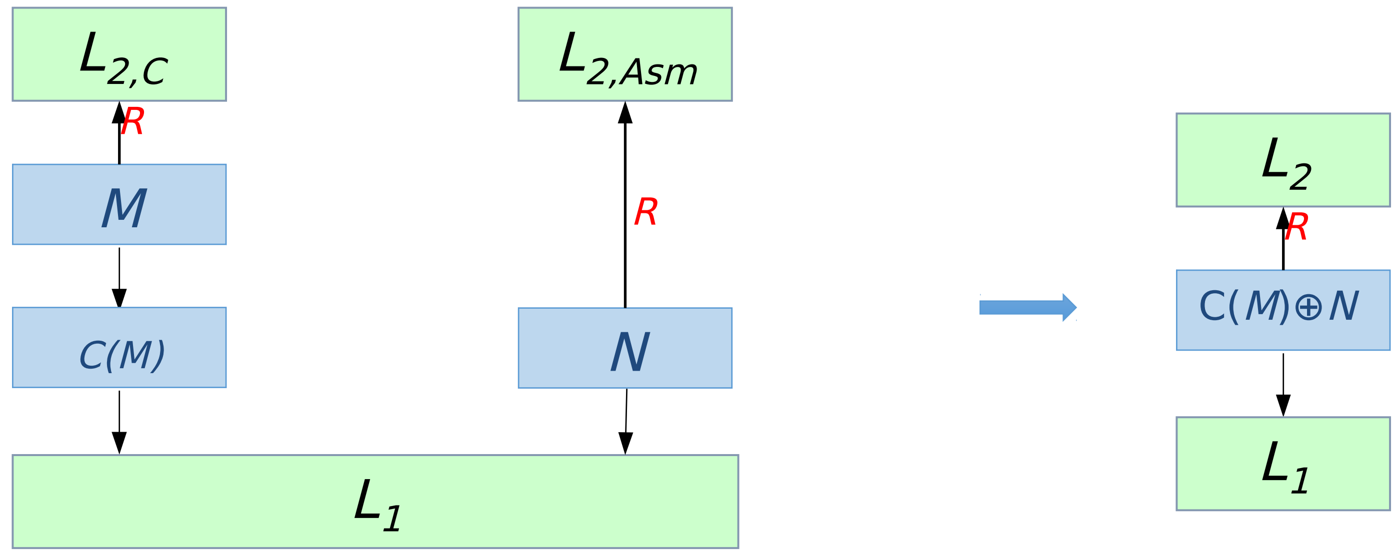
Same refinement relation

LayerLib: Horizontal Composition



- L_1 and L_2 must have the same abstract state

Refinement proof : general pattern



$L_{2,C}$

R

M

$C(M)$

$L_{2,Asm}$

R

N

L_1

L_2

R

$C(M) \oplus N$

L_1

Current limitations

- Non-global caller memory blocks are read-only
  - Due to CompCert assumption in Linear-to-Mach
  - Function call arguments must be protected
  - No arguments to top-most callee
  - 2 solutions (in progress) : copy arguments ? Use permissions ?

Limitations

- CompCertX supports no callbacks
  - Functions can only call :
    - Functions of the same module
    - Primitives of the immediate layer below
    - Lower-layer primitives must be explicitly exposed in the immediate layer
      - passed through explicitly in refinement proofs
  - Enough for CertiKOS
    - Proof by example : callbacks are not needed
    - Avoid breaches of abstraction

POPL 2015 Mumbai, India January 12-18

Our Contributions

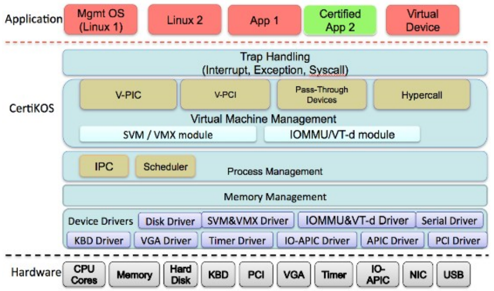
Artifact Evaluated AEC POPL Consistent Complete Well Documented Easy to Reuse

- We introduce deep specification and present a language-based formalization of certified abstraction layer

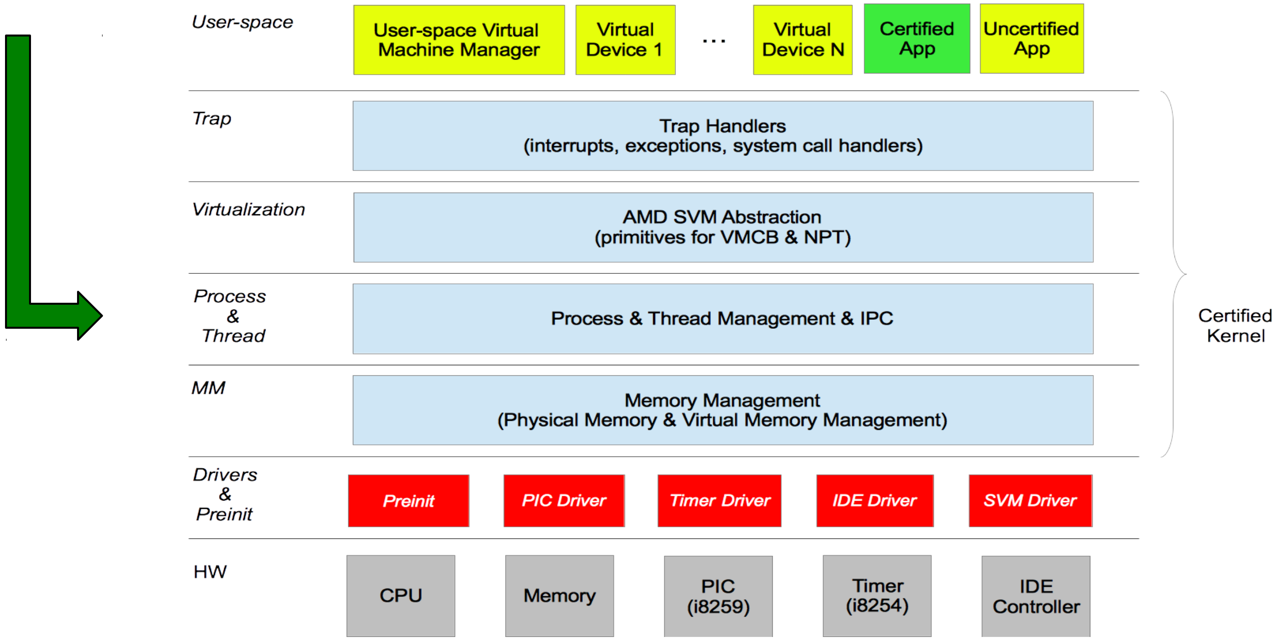
- We developed new languages & tools in Coq
  - A formal layer calculus for composing certified layers
  - ClightX for writing certified layers in a C-like language
  - AsmX for writing certified layers in assembly
  - CompCertX that compiles ClightX layers into AsmX layers

- We built multiple certified OS kernels in Coq
  - mCertiKOS-hyper consists of 37 layers, took less than one-person-year to develop, and can boot Linux as a guest

Case Study: mCertiKOS



Single-core version of CertiKOS (developed under DARPA CRASH & HACMS programs), 3 kloc, can boot Linux



User-space
User-space Virtual Machine Manager
Virtual Device 1
…
Virtual Device N
Certified App
Uncertified App

Trap
Trap Handlers
(interrupts, exceptions, system call handlers)

Virtualization
AMD SVM Abstraction
(primitives for VMCB & NPT)

Process & Thread
Process & Thread Management & IPC

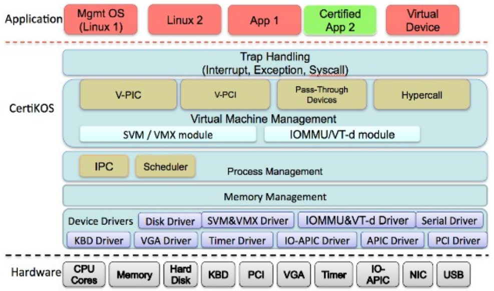
MM
Memory Management
(Physical Memory & Virtual Memory Management)

Certified Kernel

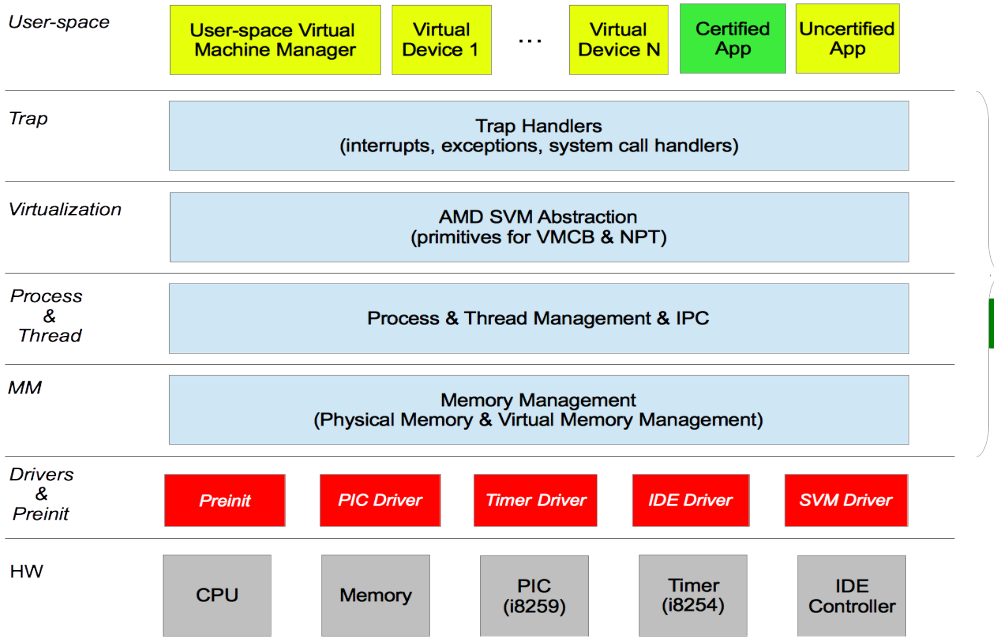
Drivers & Preinit
Preinit
PIC Driver
Timer Driver
IDE Driver
SVM Driver

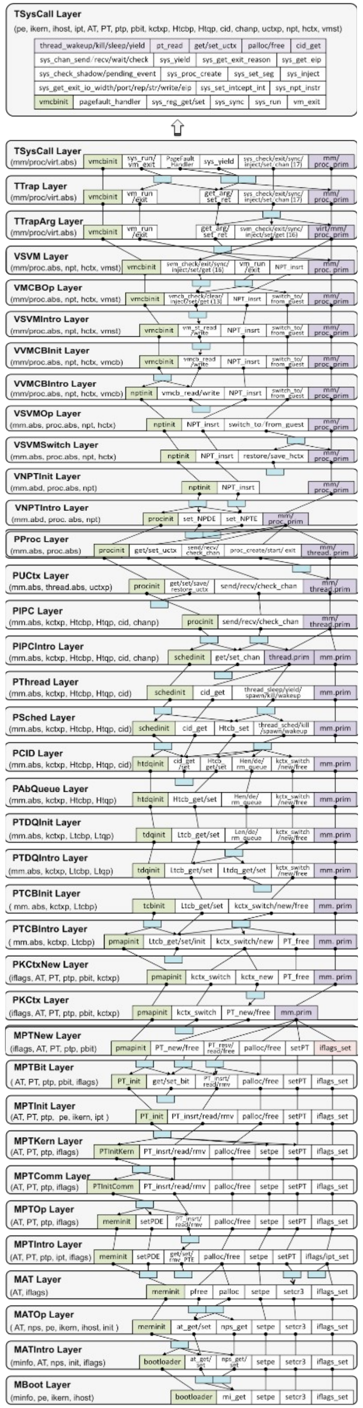
HW
CPU
Memory
PIC (i8259)
Timer (i8254)
IDE Controller

Case Study: mCertiKOS

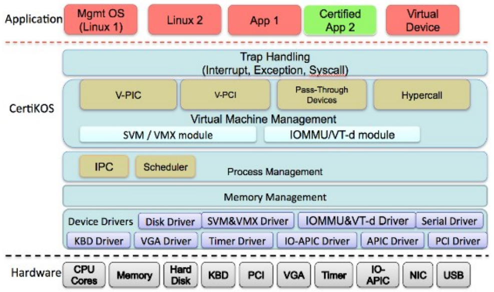


Single-core version of CertiKOS (developed under DARPA CRASH & HACMS programs), 3 kloc, can boot Linux



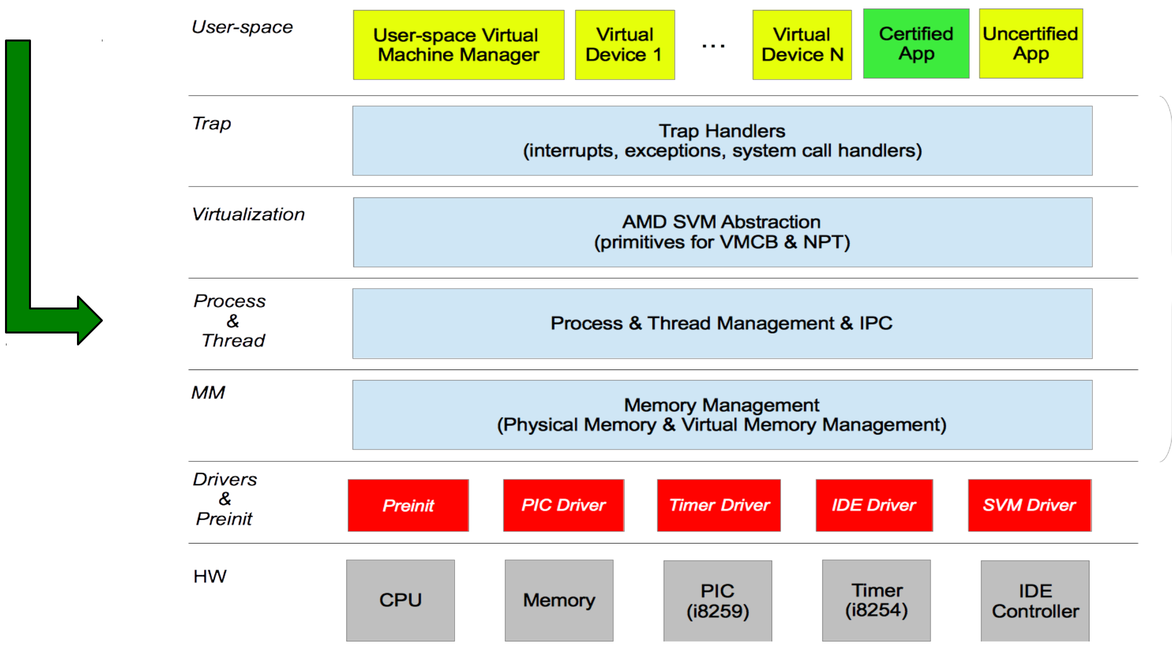


Case Study: mCertiKOS



Single-core version of CertiKOS (developed under DARPA CRASH & HACMS programs), 3 kloc, can boot Linux

Aggressive use of abstraction over deep specs (37 layers in ClightX & AsmX)



User-space: User-space Virtual Machine Manager | Virtual Device 1 | … | Virtual Device N | Certified App | Uncertified App

Trap: Trap Handlers (interrupts, exceptions, system call handlers)

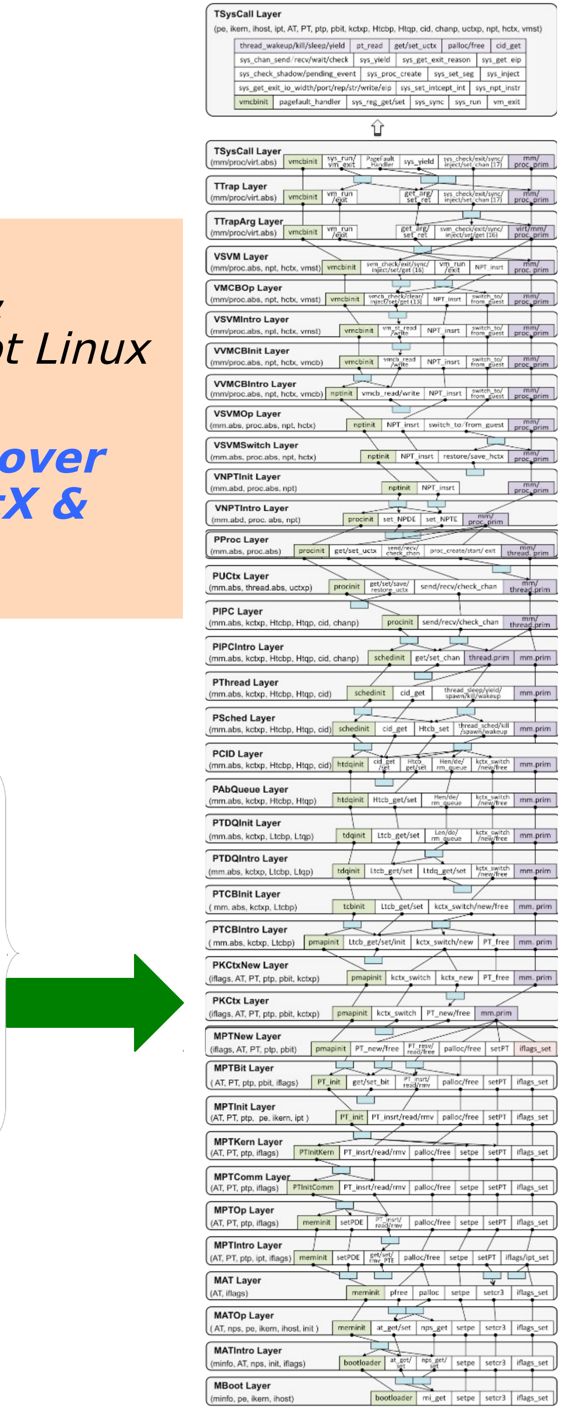
Virtualization: AMD SVM Abstraction (primitives for VMCB & NPT)

Process & Thread: Process & Thread Management & IPC

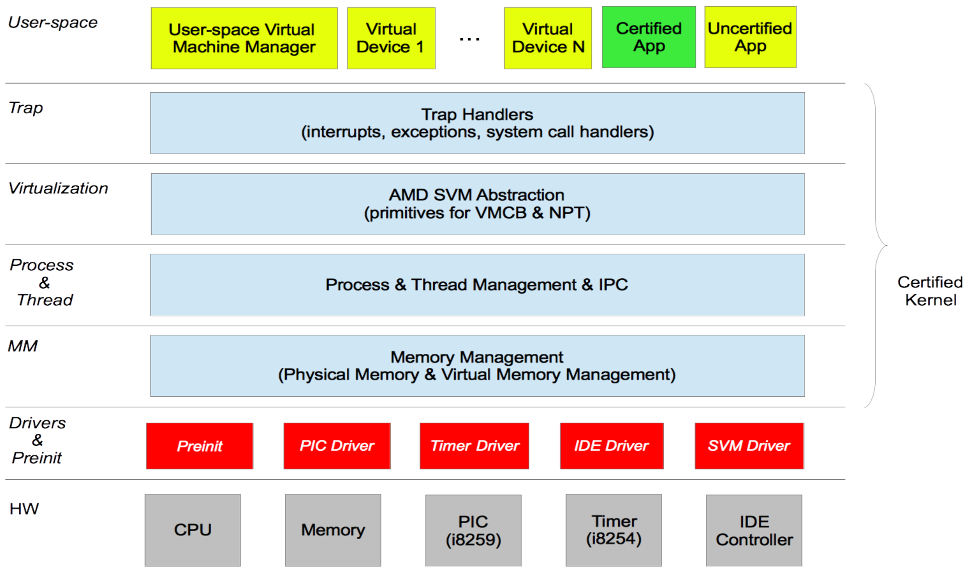
MM: Memory Management (Physical Memory & Virtual Memory Management)

Drivers & Preinit: ***Preinit*** | ***PIC Driver*** | ***Timer Driver*** | ***IDE Driver*** | ***SVM Driver***

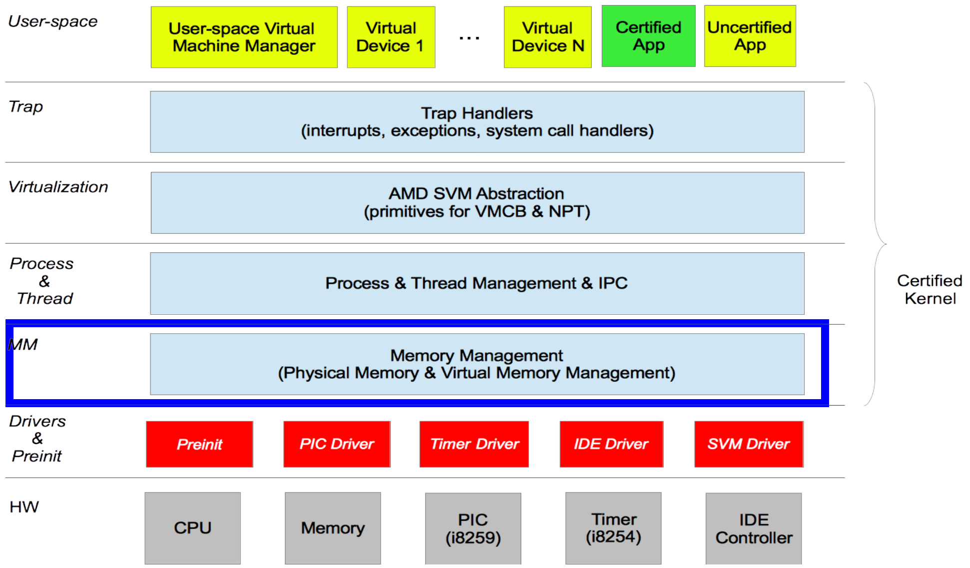
HW: CPU | Memory | PIC (i8259) | Timer (i8254) | IDE Controller



Decomposing mCertiKOS



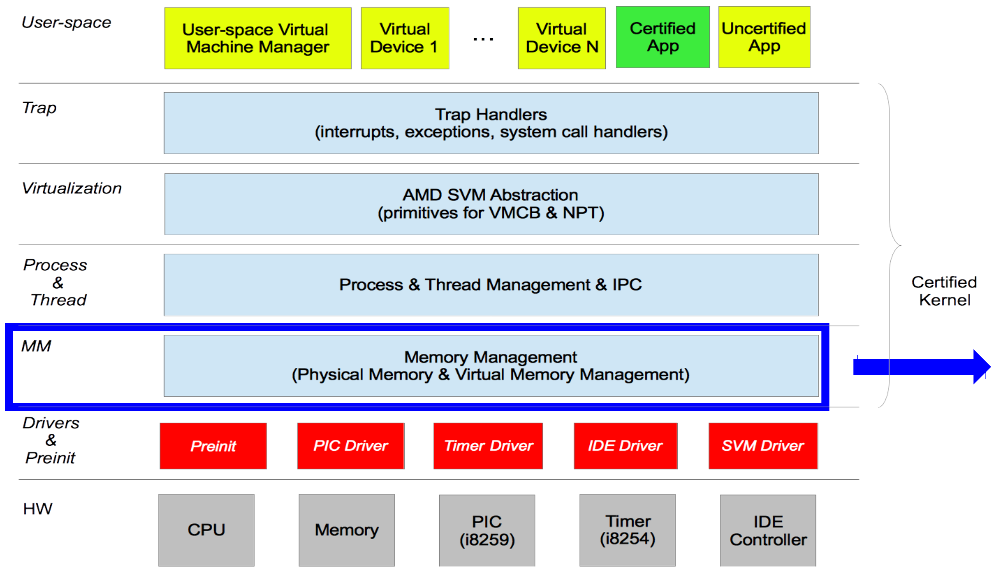
Decomposing mCertiKOS

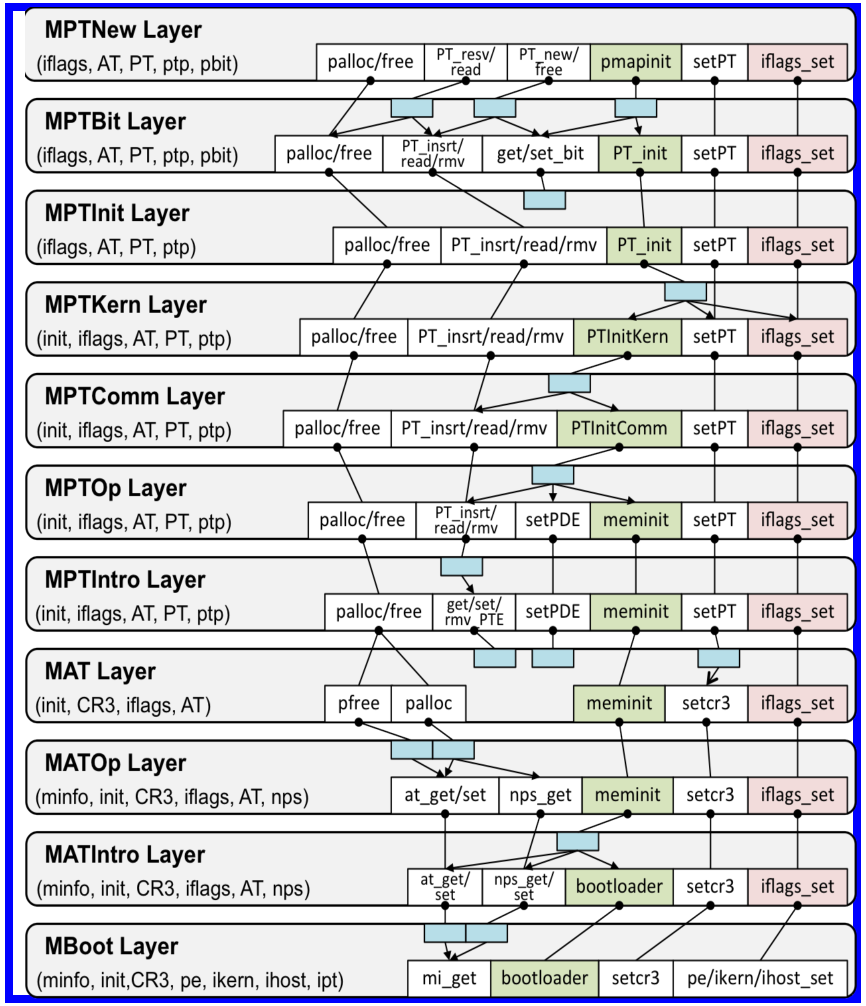


Physical Memory and Virtual Memory Management (11 Layers)

Based on the abstract machine provided by boot loader

Decomposing mCertiKOS

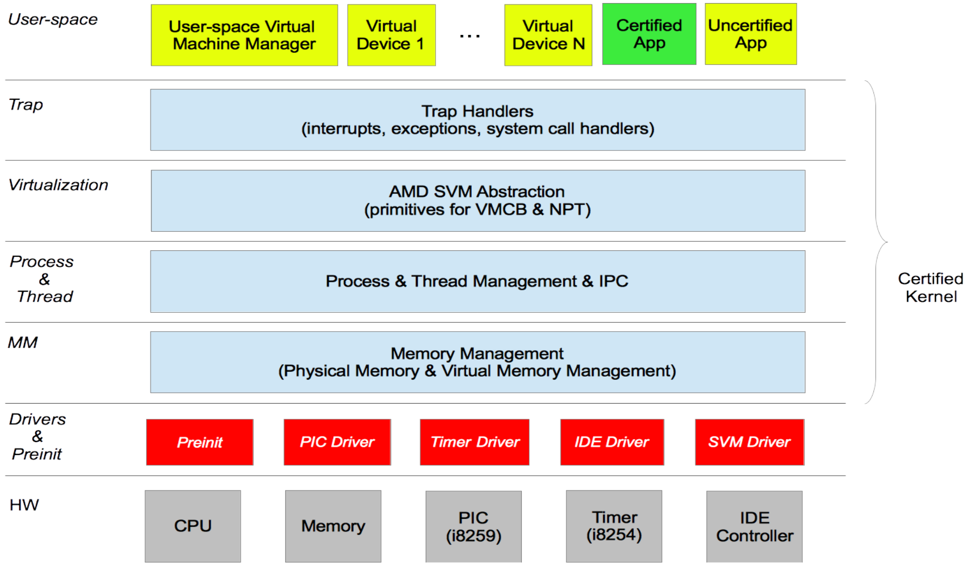




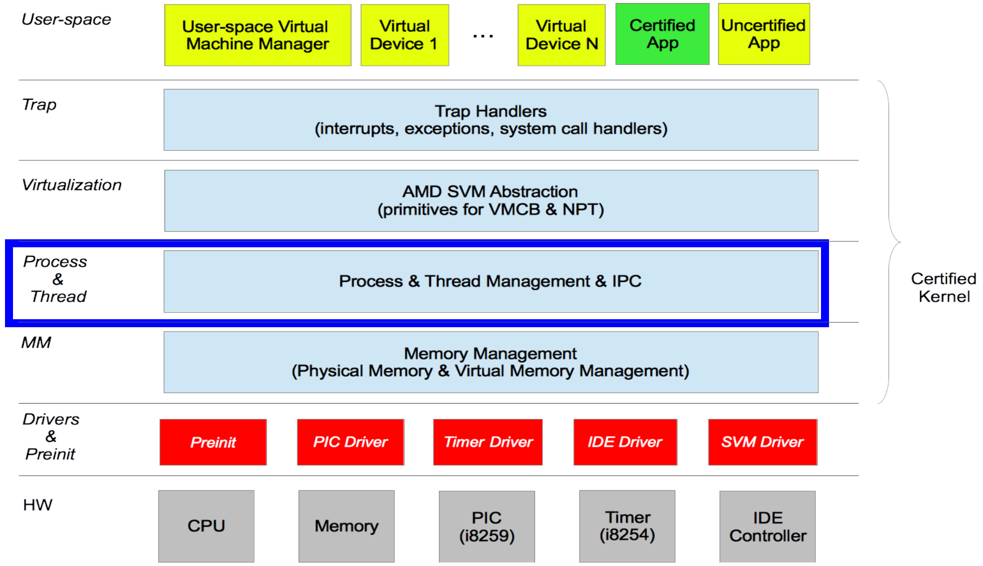
Physical Memory and Virtual Memory Management (11 Layers)

Based on the abstract machine provided by boot loader

Decomposing mCertiKOS (cont’d)

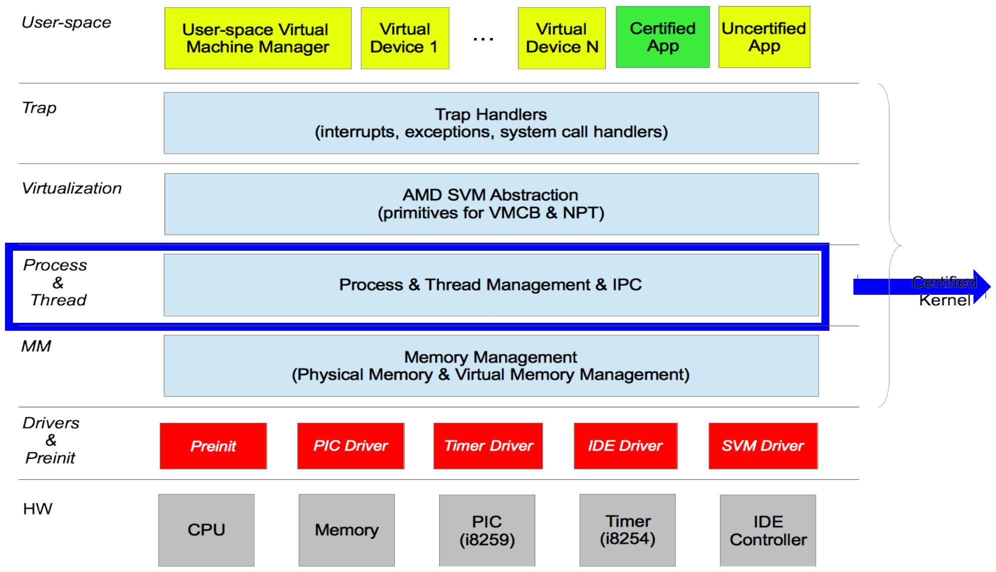


Decomposing mCertiKOS (cont'd)



Thread and Process Management (14 Layers)

Decomposing mCertiKOS (cont'd)



User-space

User-space Virtual Machine Manager

Virtual Device 1

…

Virtual Device N

Certified App

Uncertified App

Trap

Trap Handlers
(interrupts, exceptions, system call handlers)

Virtualization

AMD SVM Abstraction
(primitives for VMCB & NPT)

Process & Thread

Process & Thread Management & IPC

MM

Memory Management
(Physical Memory & Virtual Memory Management)

Drivers & Preinit

Preinit

PIC Driver

Timer Driver

IDE Driver

SVM Driver

HW

CPU

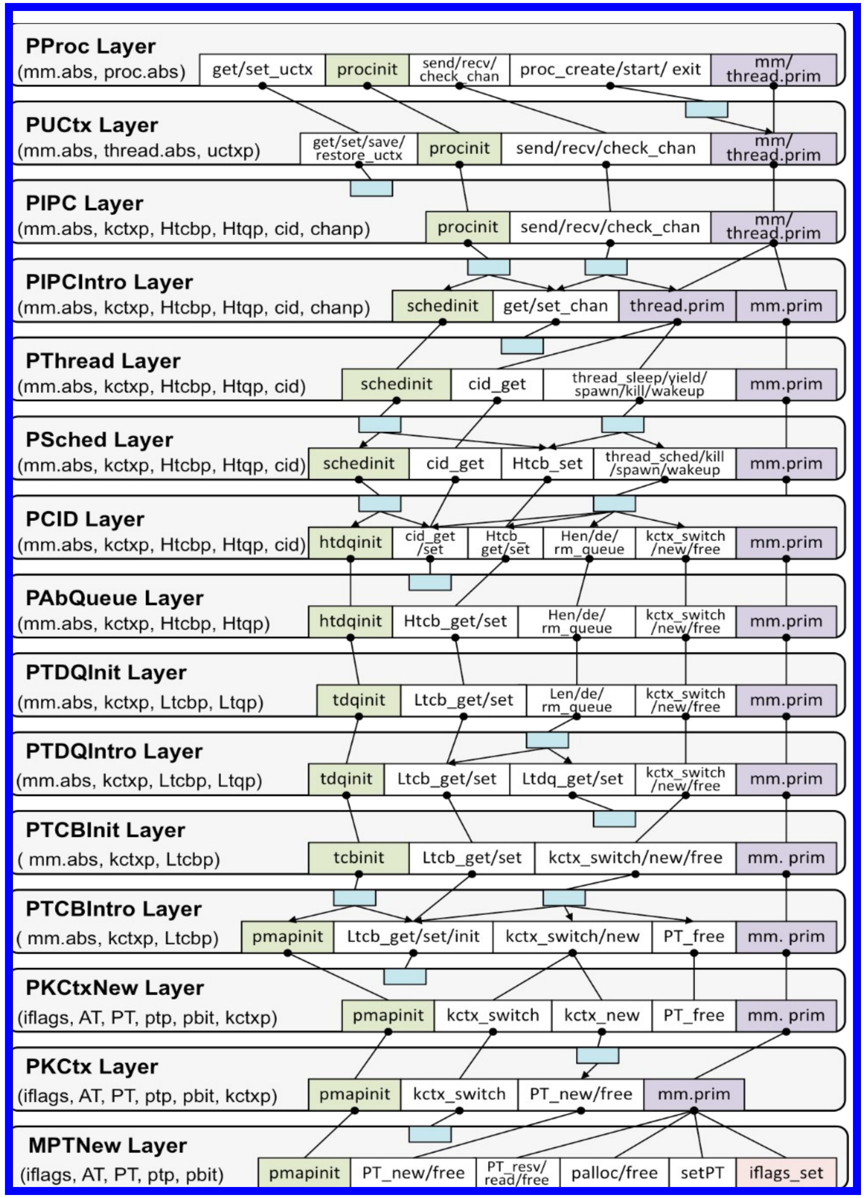
Memory

PIC (i8259)

Timer (i8254)

IDE Controller

Certified Kernel



PProc Layer
(mm.abs, proc.abs)
get/set_uctx | procinit | send/recv/ check_chan | proc_create/start/ exit | mm/ thread.prim

PUCtx Layer
(mm.abs, thread.abs, uctxp)
get/set/save/ restore_uctx | procinit | send/recv/check_chan | mm/ thread.prim

PIPC Layer
(mm.abs, kctxp, Htcbp, Htqp, cid, chanp)
procinit | send/recv/check_chan | mm/ thread.prim

PIPCIntro Layer
(mm.abs, kctxp, Htcbp, Htqp, cid, chanp)
schedinit | get/set_chan | thread.prim | mm.prim

PThread Layer
(mm.abs, kctxp, Htcbp, Htqp, cid)
schedinit | cid_get | thread_sleep/yield/ spawn/kill/wakeup | mm.prim

PSched Layer
(mm.abs, kctxp, Htcbp, Htqp, cid)
schedinit | cid_get | Htcb_set | thread_sched/kill /spawn/wakeup | mm.prim

PCID Layer
(mm.abs, kctxp, Htcbp, Htqp, cid)
htdqinit | cid_get /set | Htcb_ get/set | Hen/de/ rm_queue | kctx_switch /new/free | mm.prim

PAbQueue Layer
(mm.abs, kctxp, Htcbp, Htqp)
htdqinit | Htcb_get/set | Hen/de/ rm_queue | kctx_switch /new/free | mm.prim

PTDQInit Layer
(mm.abs, kctxp, Ltcbp, Ltqp)
tdqinit | Ltcb_get/set | Len/de/ rm_queue | kctx_switch /new/free | mm.prim

PTDQIntro Layer
(mm.abs, kctxp, Ltcbp, Ltqp)
tdqinit | Ltcb_get/set | Ltdq_get/set | kctx_switch /new/free | mm.prim

PTCBInit Layer
(mm.abs, kctxp, Ltcbp)
tcbinit | Ltcb_get/set | kctx_switch/new/free | mm. prim

PTCBIntro Layer
(mm.abs, kctxp, Ltcbp)
pmapinit | Ltcb_get/set/init | kctx_switch/new | PT_free | mm. prim

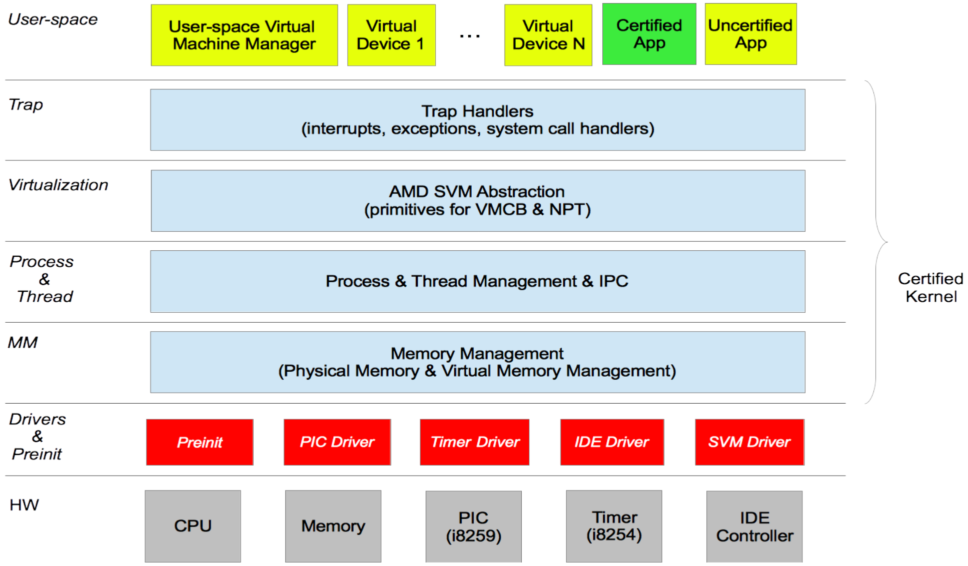
PKCtxNew Layer
(iflags, AT, PT, ptp, pbit, kctxp)
pmapinit | kctx_switch | kctx_new | PT_free | mm. prim

PKCtx Layer
(iflags, AT, PT, ptp, pbit, kctxp)
pmapinit | kctx_switch | PT_new/free | mm.prim

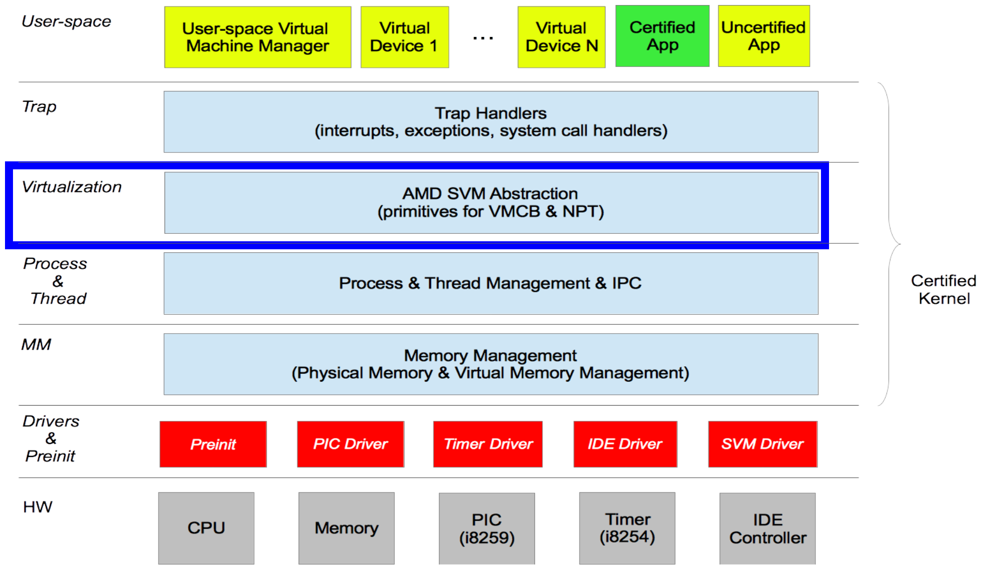
MPTNew Layer
(iflags, AT, PT, ptp, pbit)
pmapinit | PT_new/free | PT_resv/ read/free | palloc/free | setPT | iflags_set

Thread and Process Management (14 Layers)

Decomposing mCertiKOS (cont’d)

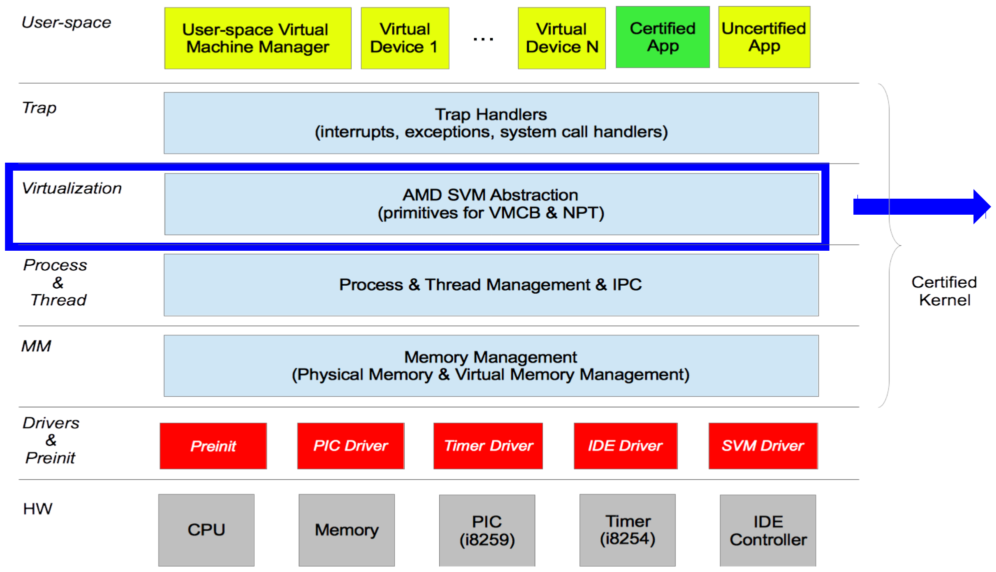


Decomposing mCertiKOS (cont’d)



Virtualization Support (9 Layers)

Decomposing mCertiKOS (cont’d)



User-space
User-space Virtual Machine Manager
Virtual Device 1
…
Virtual Device N
Certified App
Uncertified App

Trap
Trap Handlers
(interrupts, exceptions, system call handlers)

Virtualization
AMD SVM Abstraction
(primitives for VMCB & NPT)

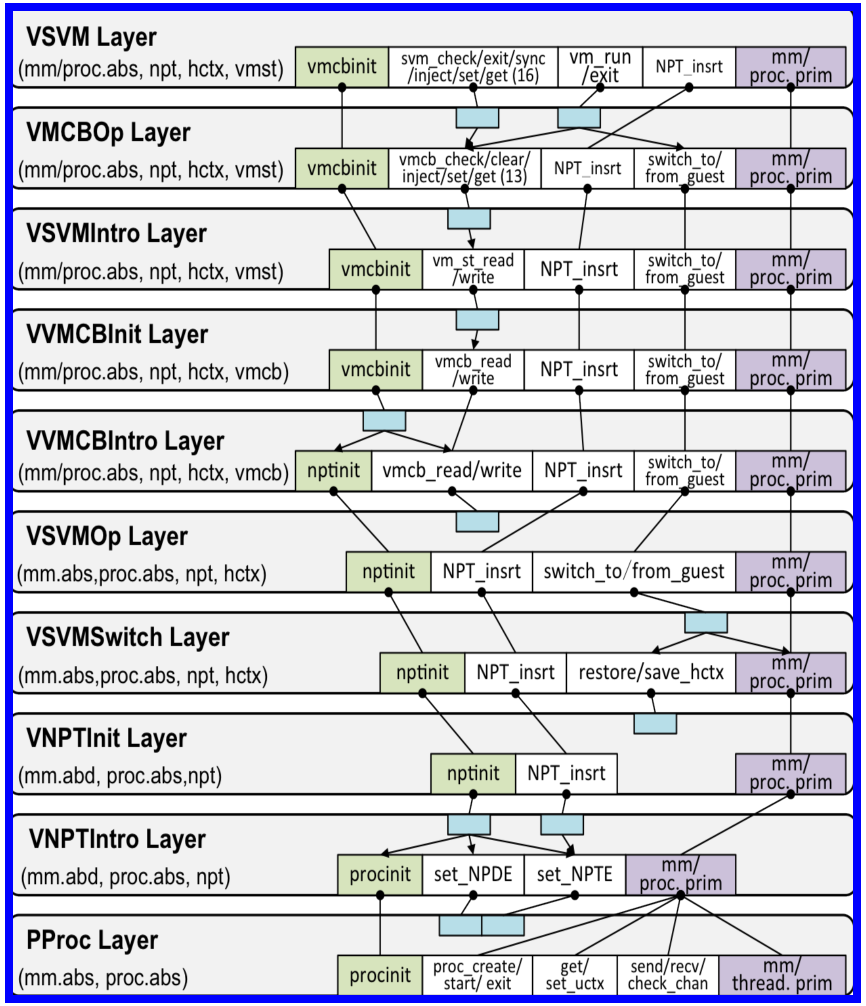
Process & Thread
Process & Thread Management & IPC

MM
Memory Management
(Physical Memory & Virtual Memory Management)

Certified Kernel

Drivers & Preinit
Preinit
PIC Driver
Timer Driver
IDE Driver
SVM Driver

HW
CPU
Memory
PIC (i8259)
Timer (i8254)
IDE Controller



VSVM Layer
(mm/proc.abs, npt, hctx, vmst)
vmcbinit | svm_check/exit/sync /inject/set/get (16) | vm_run /exit | NPT_insrt | mm/ proc. prim

VMCBOp Layer
(mm/proc.abs, npt, hctx, vmst)
vmcbinit | vmcb_check/clear/ inject/set/get (13) | NPT_insrt | switch_to/ from_guest | mm/ proc. prim

VSVMIntro Layer
(mm/proc.abs, npt, hctx, vmst)
vmcbinit | vm_st_read /write | NPT_insrt | switch_to/ from_guest | mm/ proc. prim

VVMCBInit Layer
(mm/proc.abs, npt, hctx, vmcb)
vmcbinit | vmcb_read /write | NPT_insrt | switch_to/ from_guest | mm/ proc. prim

VVMCBIntro Layer
(mm/proc.abs, npt, hctx, vmcb)
nptinit | vmcb_read/write | NPT_insrt | switch_to/ from_guest | mm/ proc. prim

VSVMOp Layer
(mm.abs,proc.abs, npt, hctx)
nptinit | NPT_insrt | switch_to/from_guest | mm/ proc. prim

VSVMSwitch Layer
(mm.abs,proc.abs, npt, hctx)
nptinit | NPT_insrt | restore/save_hctx | mm/ proc. prim

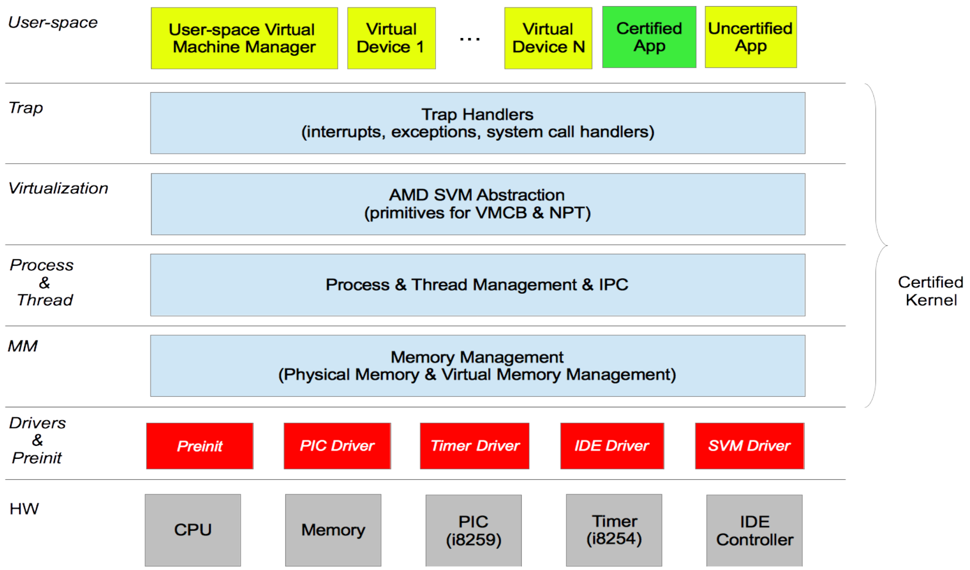
VNPTInit Layer
(mm.abd, proc.abs,npt)
nptinit | NPT_insrt | mm/ proc. prim

VNPTIntro Layer
(mm.abd, proc.abs, npt)
procinit | set_NPDE | set_NPTE | mm/ proc. prim

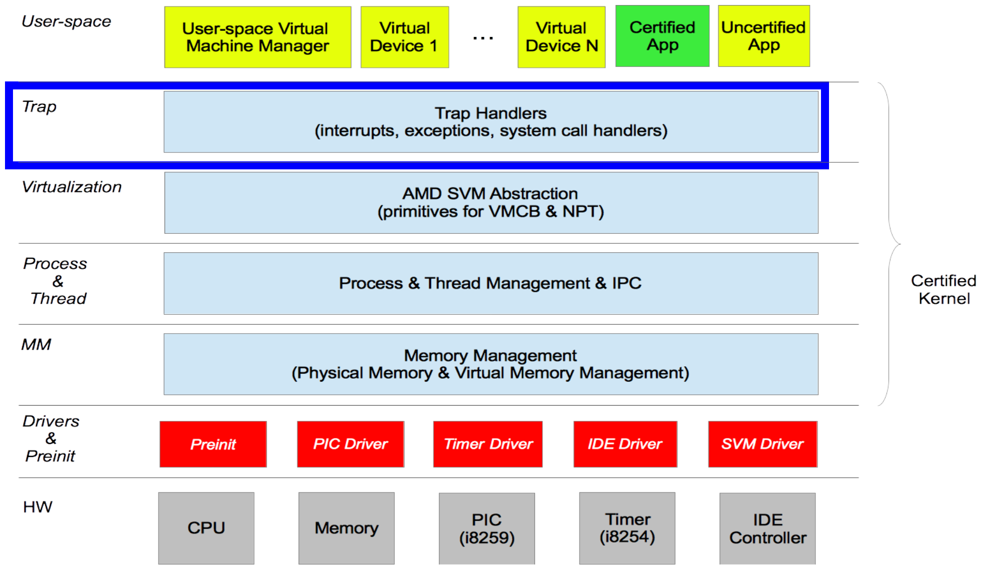
PProc Layer
(mm.abs, proc.abs)
procinit | proc_create/ start/ exit | get/ set_uctx | send/recv/ check_chan | mm/ thread. prim

Virtualization Support (9 Layers)

Decomposing mCertiKOS (cont’d)

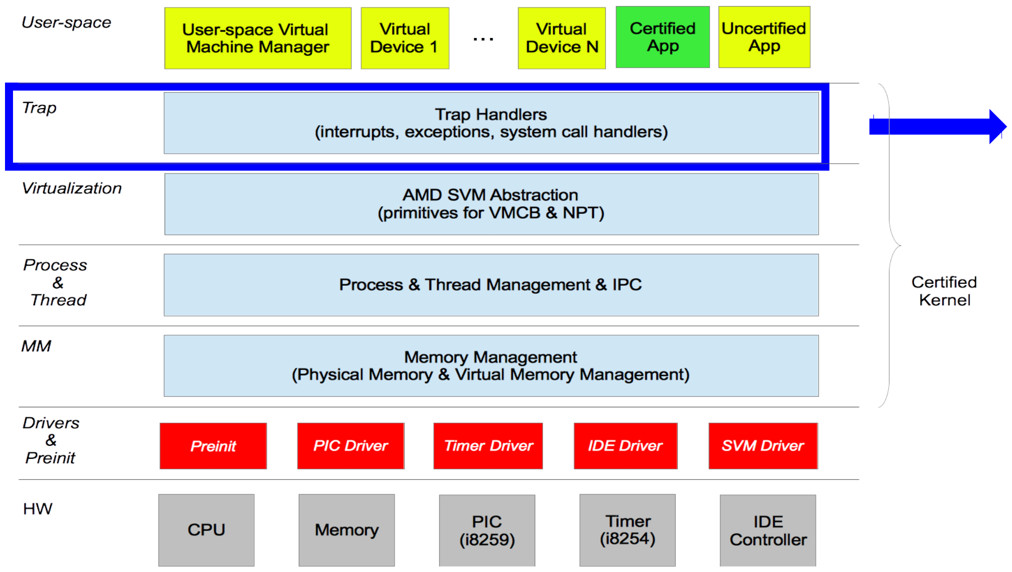


Decomposing mCertiKOS (cont'd)



Syscall and Trap Handlers (3 Layers)

Decomposing mCertiKOS (cont'd)


User-space
User-space Virtual Machine Manager
Virtual Device 1
…
Virtual Device N
Certified App
Uncertified App

Trap
Trap Handlers
(interrupts, exceptions, system call handlers)

Virtualization
AMD SVM Abstraction
(primitives for VMCB & NPT)

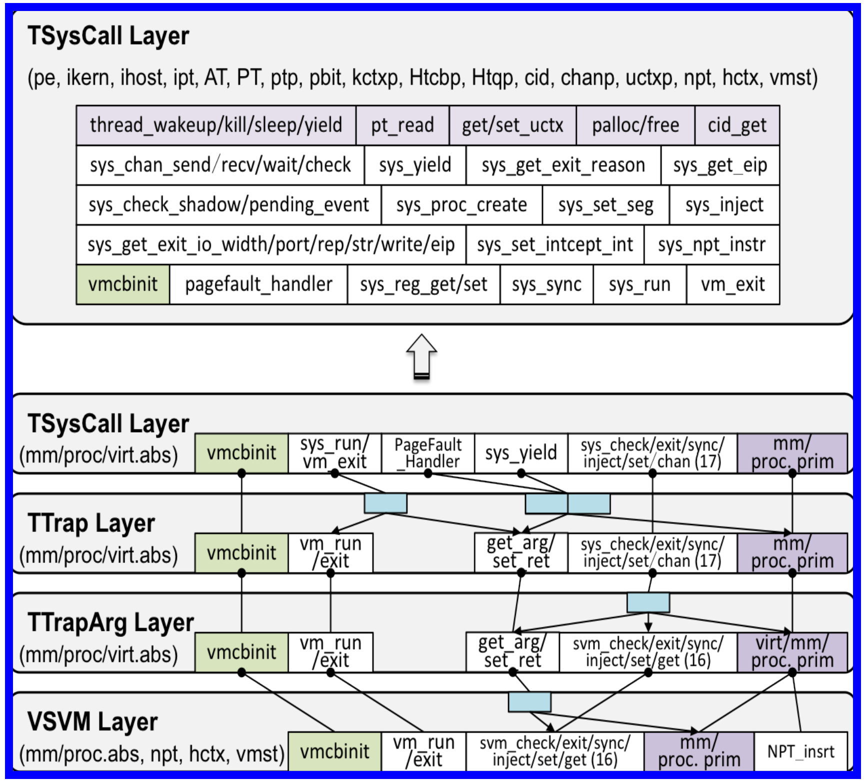
Process & Thread
Process & Thread Management & IPC

MM
Memory Management
(Physical Memory & Virtual Memory Management)

Certified Kernel

Drivers & Preinit
Preinit
PIC Driver
Timer Driver
IDE Driver
SVM Driver

HW
CPU
Memory
PIC (i8259)
Timer (i8254)
IDE Controller


TSysCall Layer
(pe, ikern, ihost, ipt, AT, PT, ptp, pbit, kctxp, Htcbp, Htqp, cid, chanp, uctxp, npt, hctx, vmst)

thread_wakeup/kill/sleep/yield | pt_read | get/set_uctx | palloc/free | cid_get
sys_chan_send/recv/wait/check | sys_yield | sys_get_exit_reason | sys_get_eip
sys_check_shadow/pending_event | sys_proc_create | sys_set_seg | sys_inject
sys_get_exit_io_width/port/rep/str/write/eip | sys_set_intcept_int | sys_npt_instr
vmcbinit | pagefault_handler | sys_reg_get/set | sys_sync | sys_run | vm_exit

TSysCall Layer
(mm/proc/virt.abs)
vmcbinit | sys_run/ vm_exit | PageFault _Handler | sys_yield | sys_check/exit/sync/ inject/set/chan (17) | mm/ proc. prim

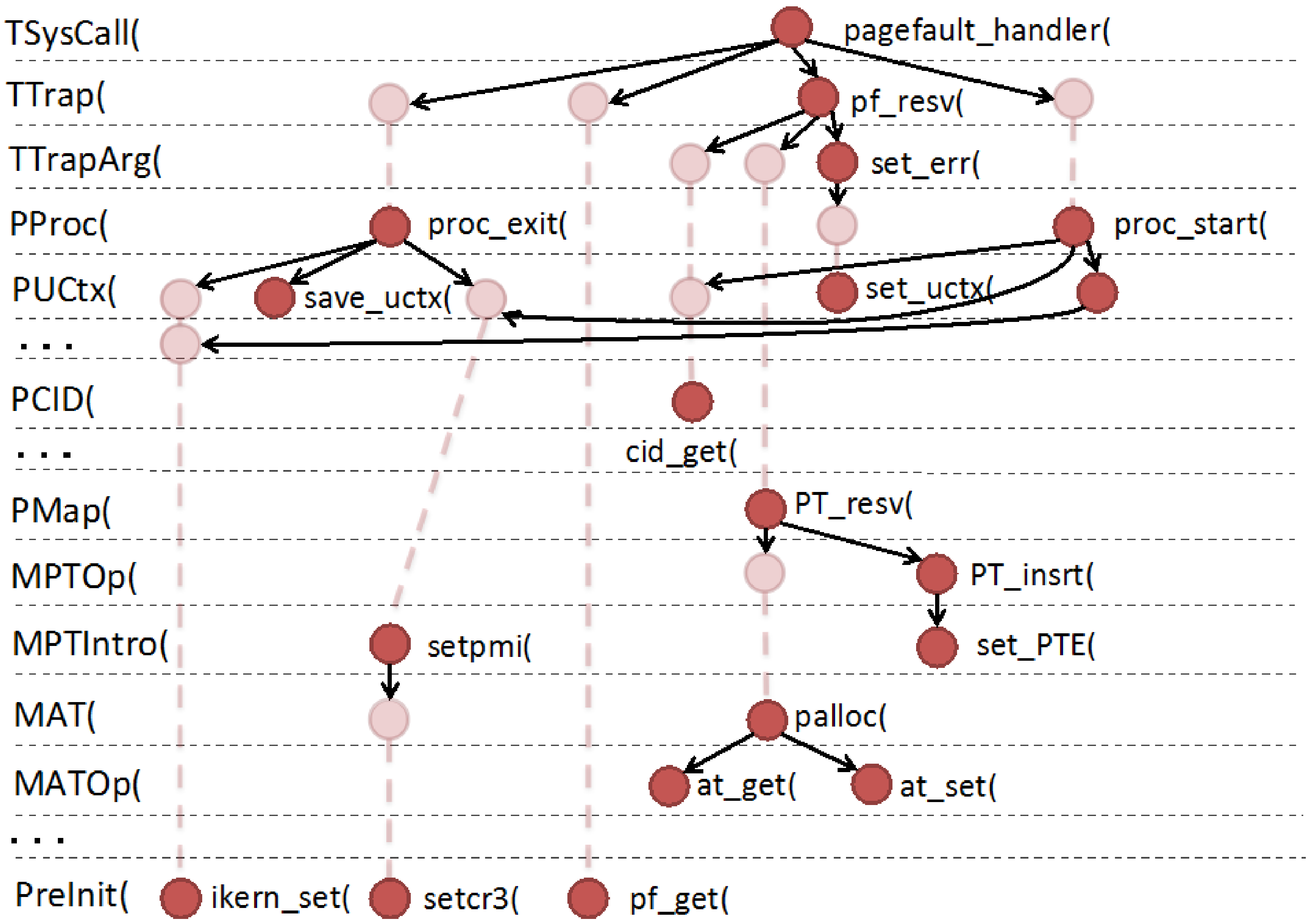
TTrap Layer
(mm/proc/virt.abs)
vmcbinit | vm_run /exit | get_arg/ set_ret | sys_check/exit/sync/ inject/set/chan (17) | mm/ proc. prim

TTrapArg Layer
(mm/proc/virt.abs)
vmcbinit | vm_run /exit | get_arg/ set_ret | svm_check/exit/sync/ inject/set/get (16) | virt/mm/ proc. prim

VSVM Layer
(mm/proc.abs, npt, hctx, vmst)
vmcbinit | vm_run /exit | svm_check/exit/sync/ inject/set/get (16) | mm/ proc. prim | NPT_insrt

Syscall and Trap Handlers (3 Layers)

Example: Page Fault Handler



TSysCall(pagefault_handler(

TTrap(pf_resv(

TTrapArg(set_err(

PProc(proc_exit(proc_start(

PUCtx(save_uctx(set_uctx(

. . .

PCID(

. . . cid_get(

PMap(PT_resv(

MPTOp(PT_insrt(

MPTIntro(setpmi(set_PTE(

MAT(palloc(

MATOp(at_get(at_set(

. . .

PreInit(ikern_set(setcr3(pf_get(

Conclusions

- Great success w. today's system software … but why?
- We identify, sharpen, & formalize two possible ingredients
  - abstraction over deep specs
  - a compositional layered methodology
- We build new lang. & tools to make layered programming *rigorous & certified* --- this leads to huge benefits:
  - simplified design & spec; reduced proof effort; better extensibility
- They also help *verification in the small*
  - hiding implementation details as soon as possible

Ongoing & Future Work

- CompCertX
  - Support for stack consumption (merge with QompCert, PLDI 2014)
  - Concurrency?
- Layer calculus
  - Make horizontal composition more powerful (disjoint abstract states...)
- Compositional verification of layered systems in practice
  - Automatic generation of ClightX code, layer spec and refinement proof
- CertiKOS
  - Device drivers, concurrency (multicore?), etc.
  - Publish the code!
  - Papers available at http://flint.cs.yale.edu