Using the Cambridge ARM model to verify the concrete machine code of seL4

Magnus Myreen[1,2], Thomas Sewell[1], Michael Norrish[1] and Gerwin Klein[1]

[1] NICTA, Australia
[2] University of Cambridge, UK

L4.verified

seL4 = a formally verified general-purpose microkernel

L4.verified

seL4 = a formally verified general-purpose microkernel

about 10,000 lines of C code and assembly

L4.verified

seL4 = a formally verified general-purpose microkernel

about 10,000 lines of C code and assembly

200,000 lines of Isabelle/HOL proofs

Assumptions

L4.verified project assumes correctness of:

- C compiler (gcc)
- inline assembly
- hardware
- hardware management
- boot code
- virtual memory

Assumptions

L4.verified project assumes correctness of:

- ~~C compiler (gcc)~~
- inline assembly
- hardware
- hardware management
- boot code
- virtual memory

The aim of this work is to remove the first assumption.

Assumptions

L4.verified project assumes correctness of:

- ~~C compiler (gcc)~~
- inline assembly
- hardware
- hardware management
- boot code
- virtual memory
- Cambridge ARM model

The aim of this work is to remove the first assumption.

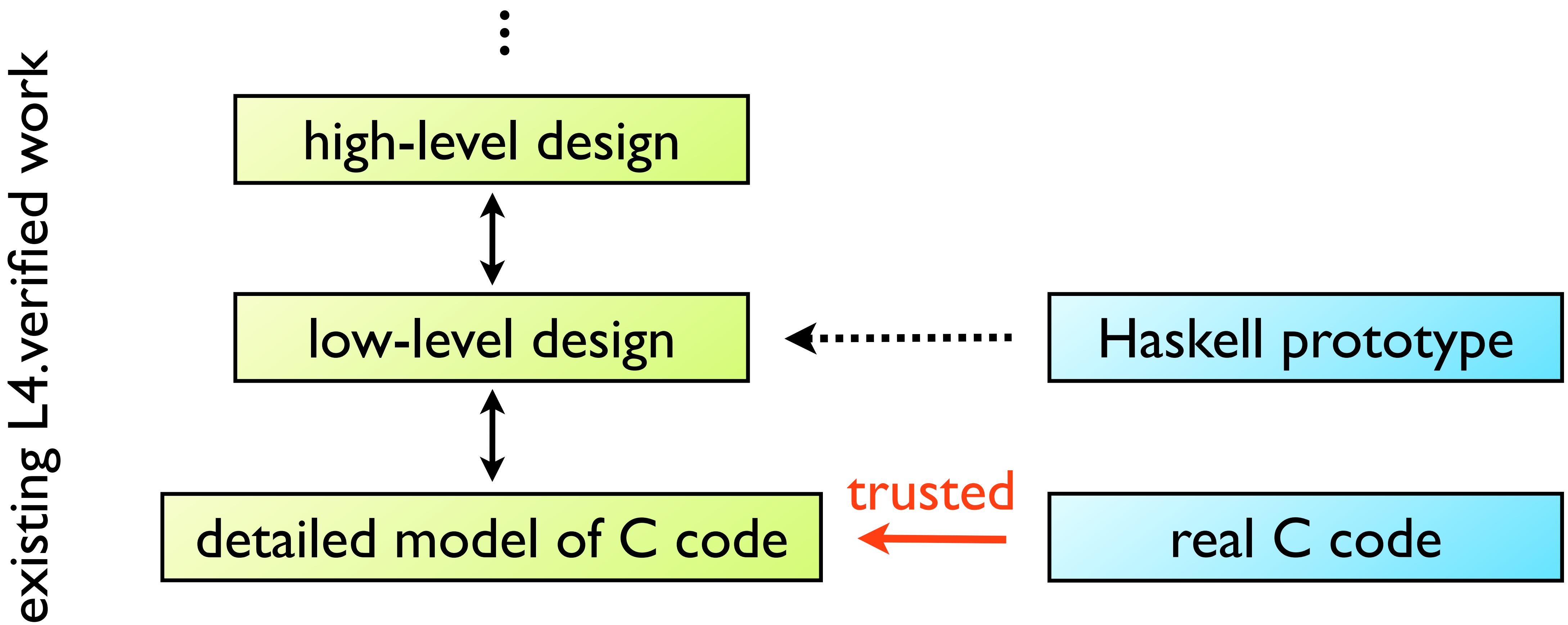
Assumptions

L4.verified project assumes correctness of:

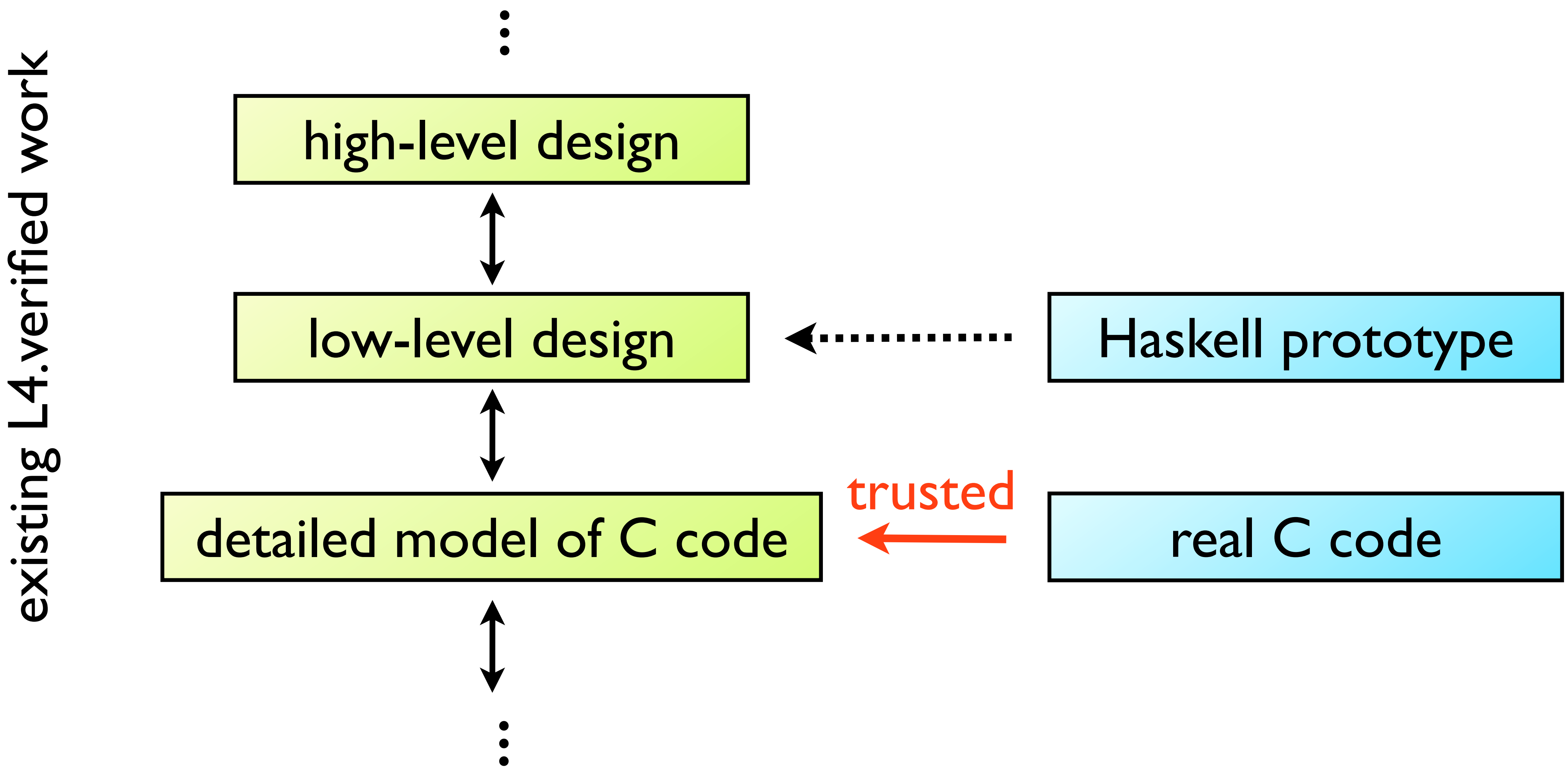
- ~~C compiler (gcc)~~
- inline assembly (?)
- hardware
- hardware management
- boot code (?)
- virtual memory
- Cambridge ARM model

The aim of this work is to remove the first assumption.

Aim: extend downwards

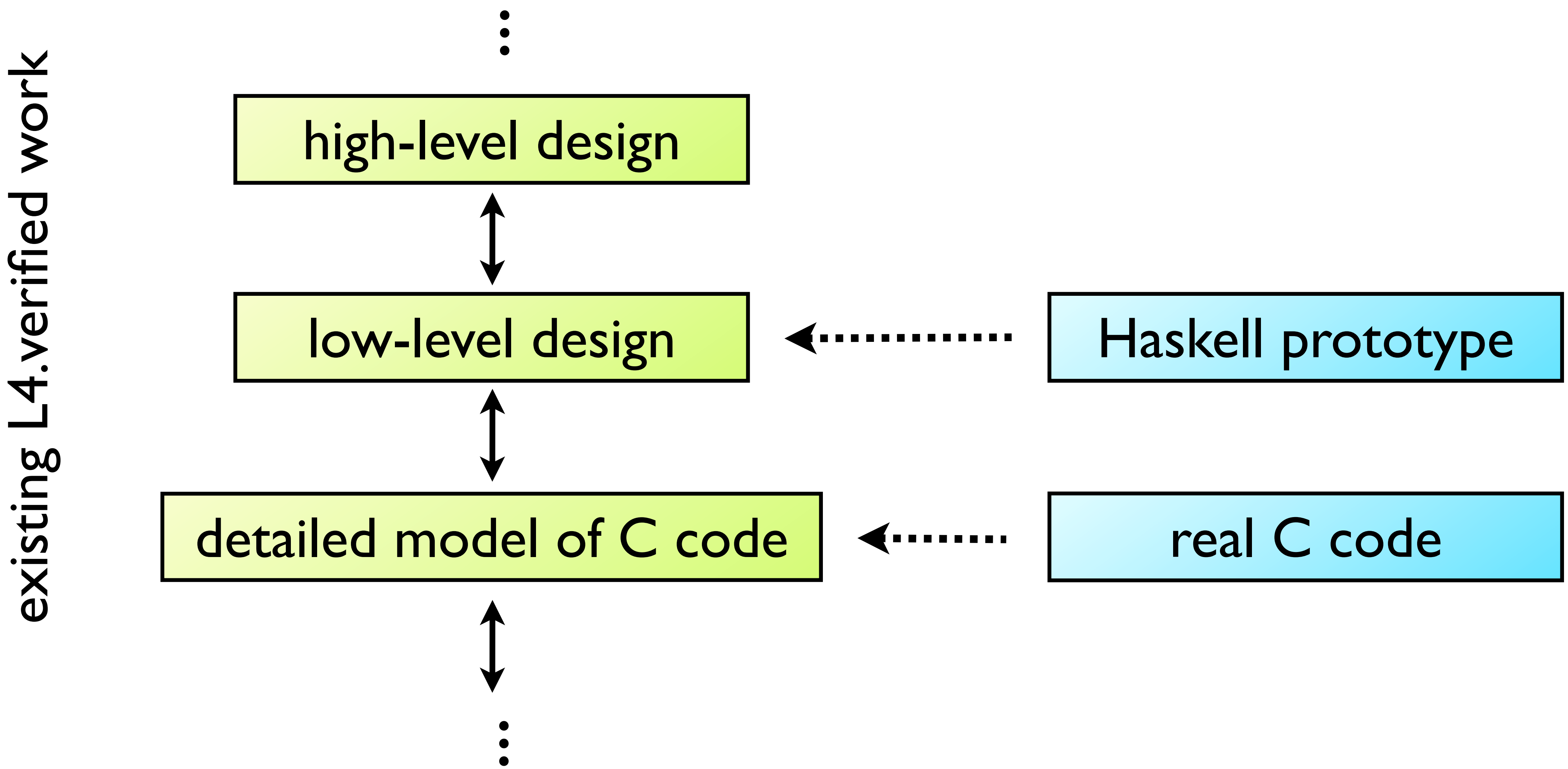


Aim: extend downwards



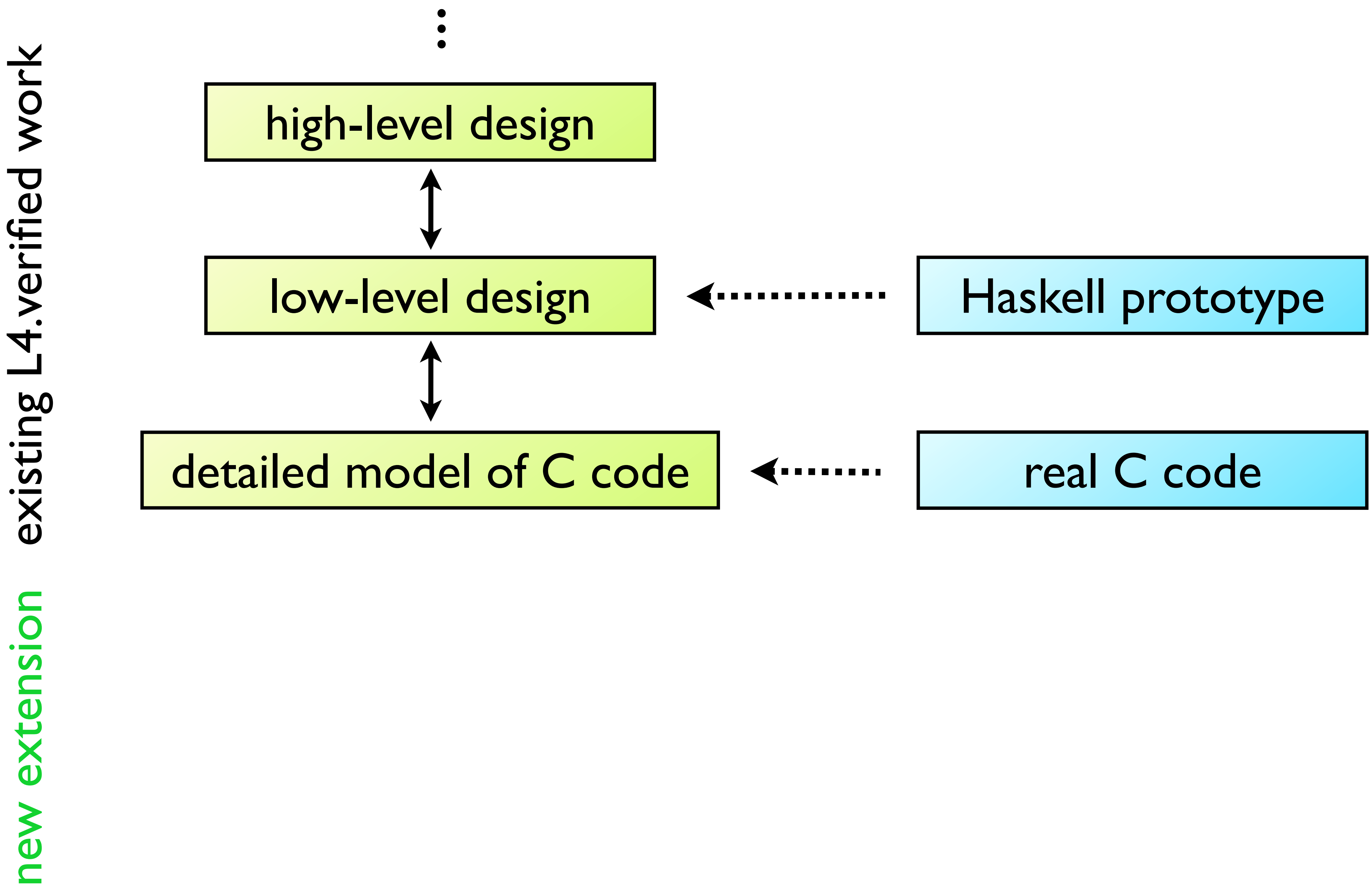
Aim: remove need to trust C compiler and C semantics

Aim: extend downwards

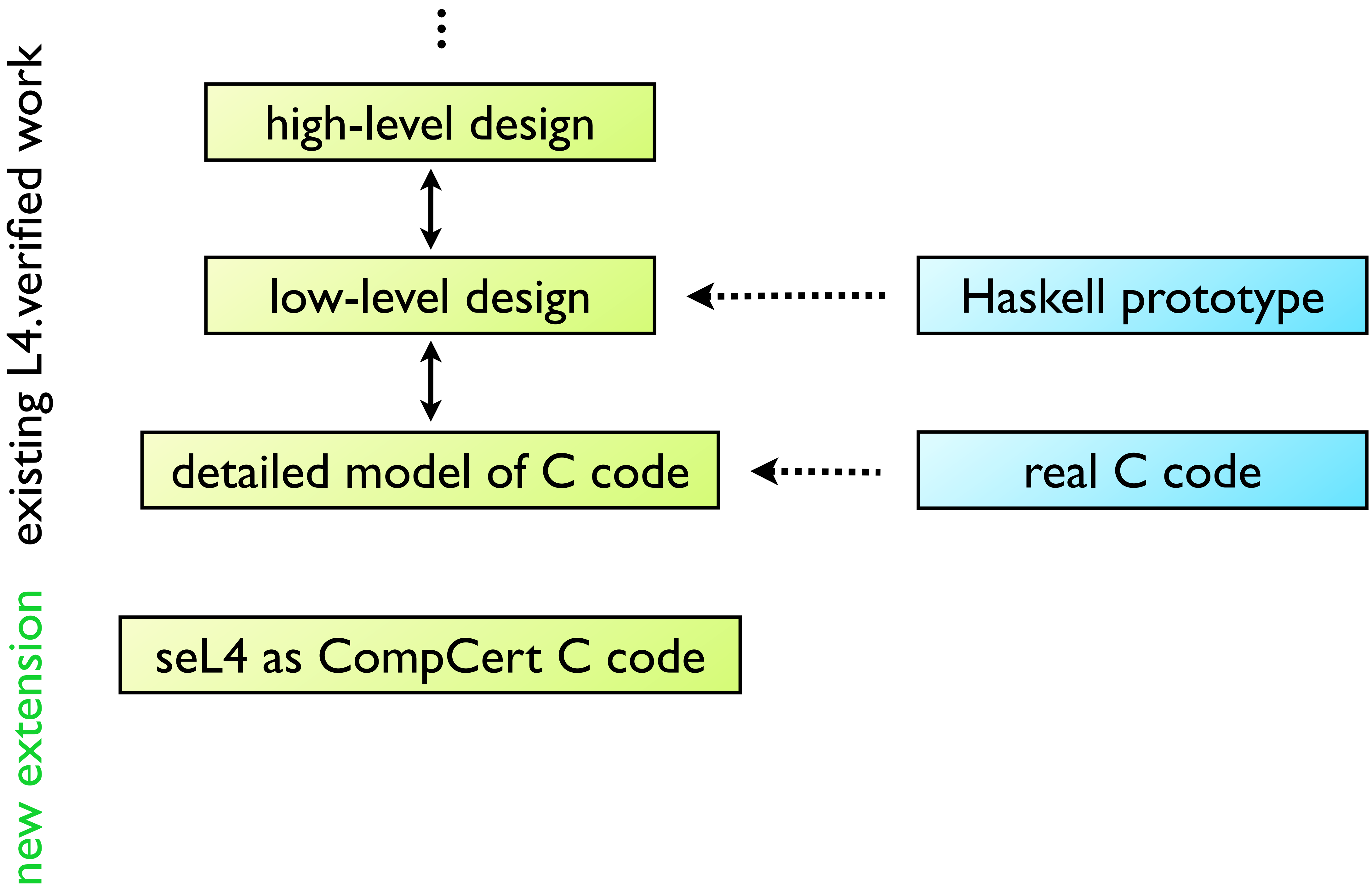


Aim: remove need to trust C compiler and C semantics

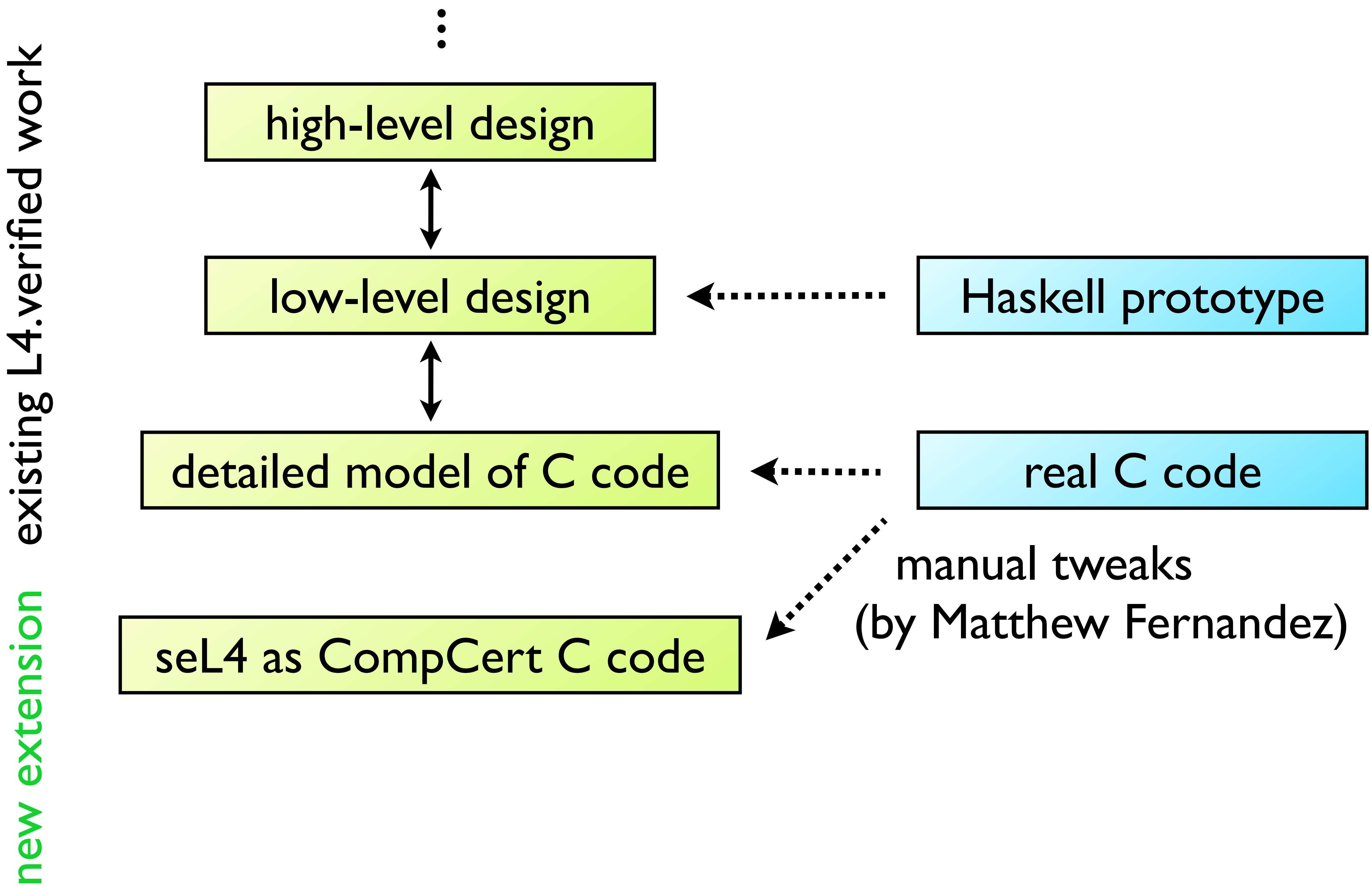
Connection to CompCert



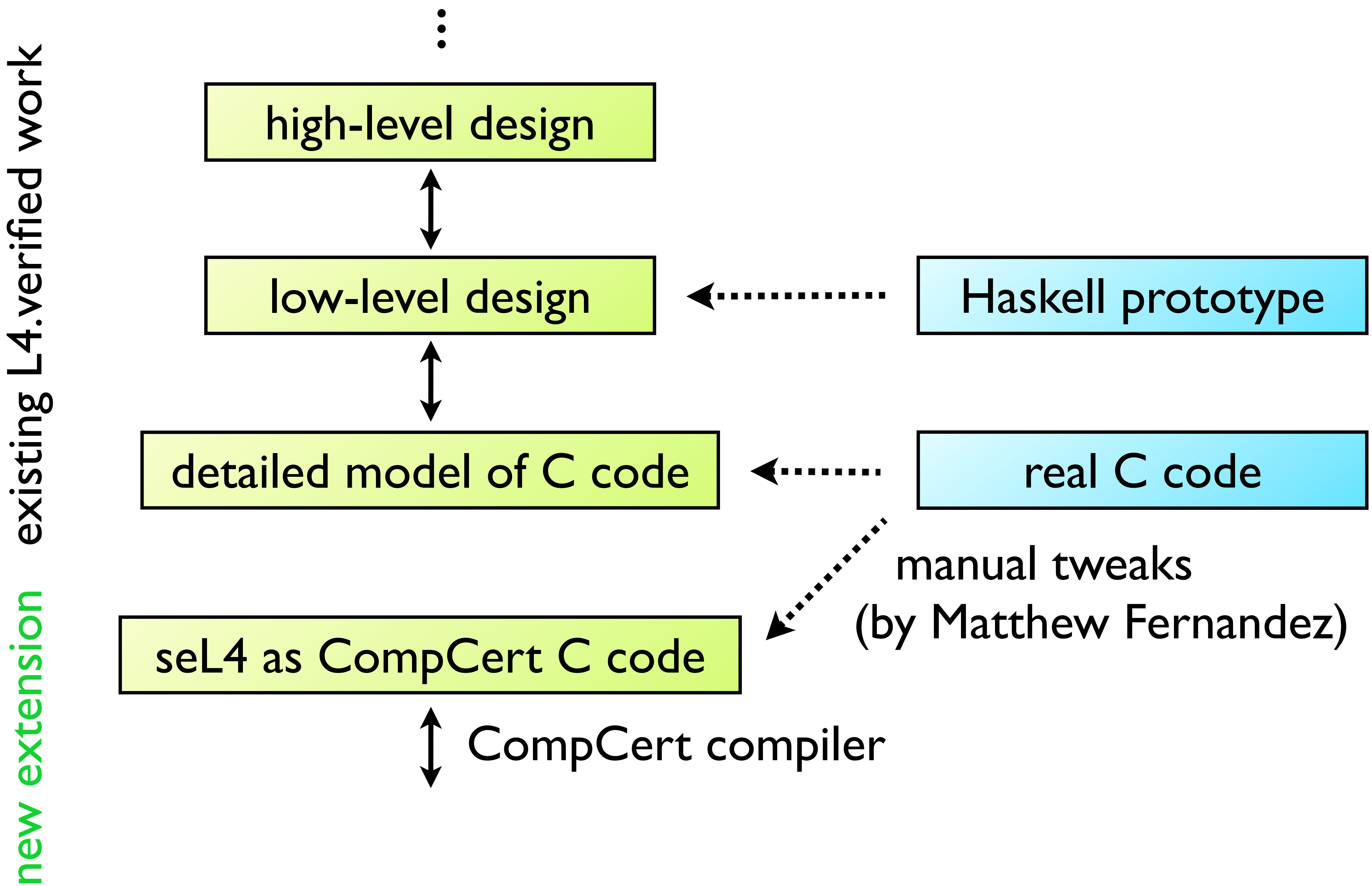
Connection to CompCert



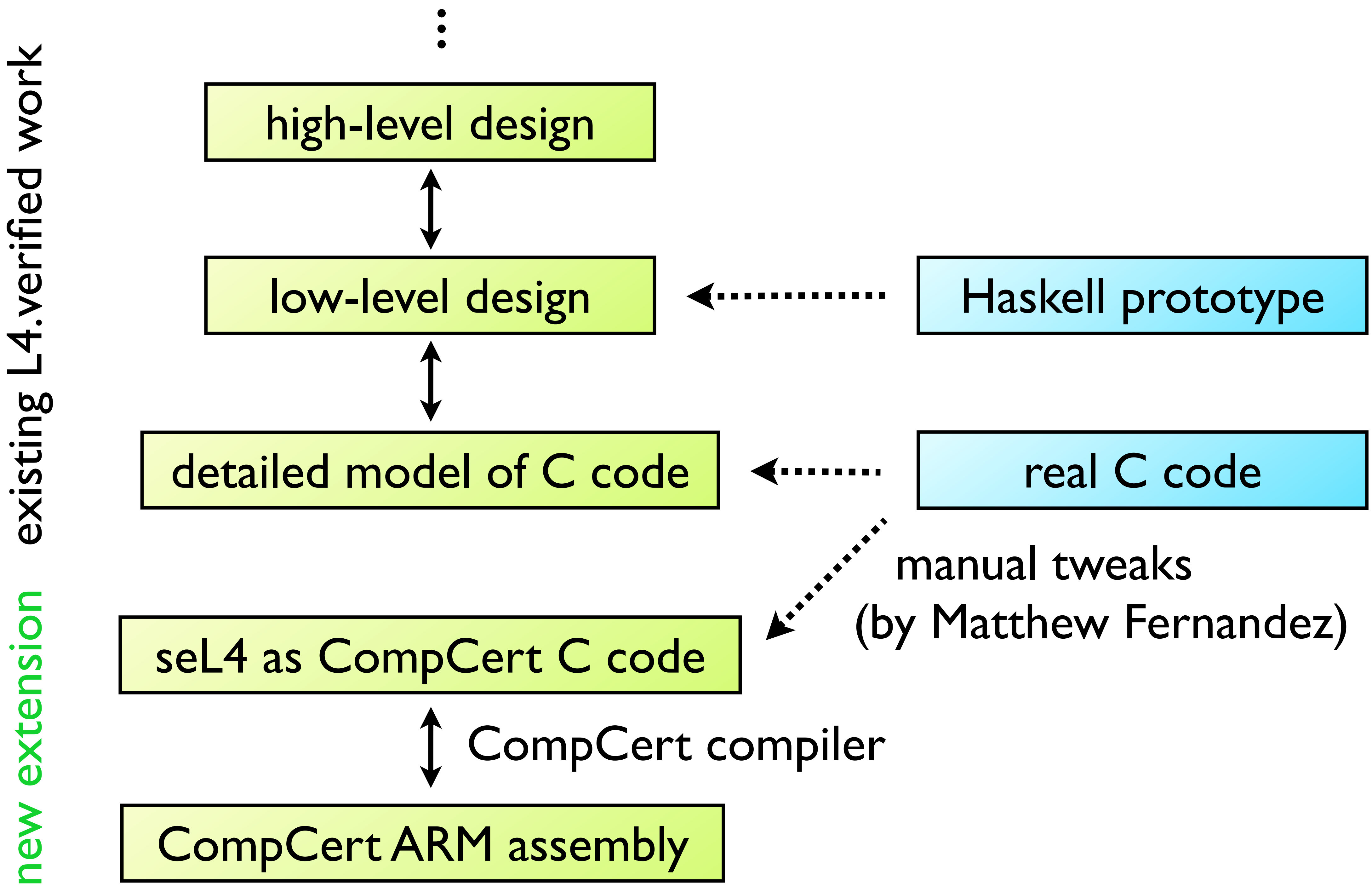
Connection to CompCert



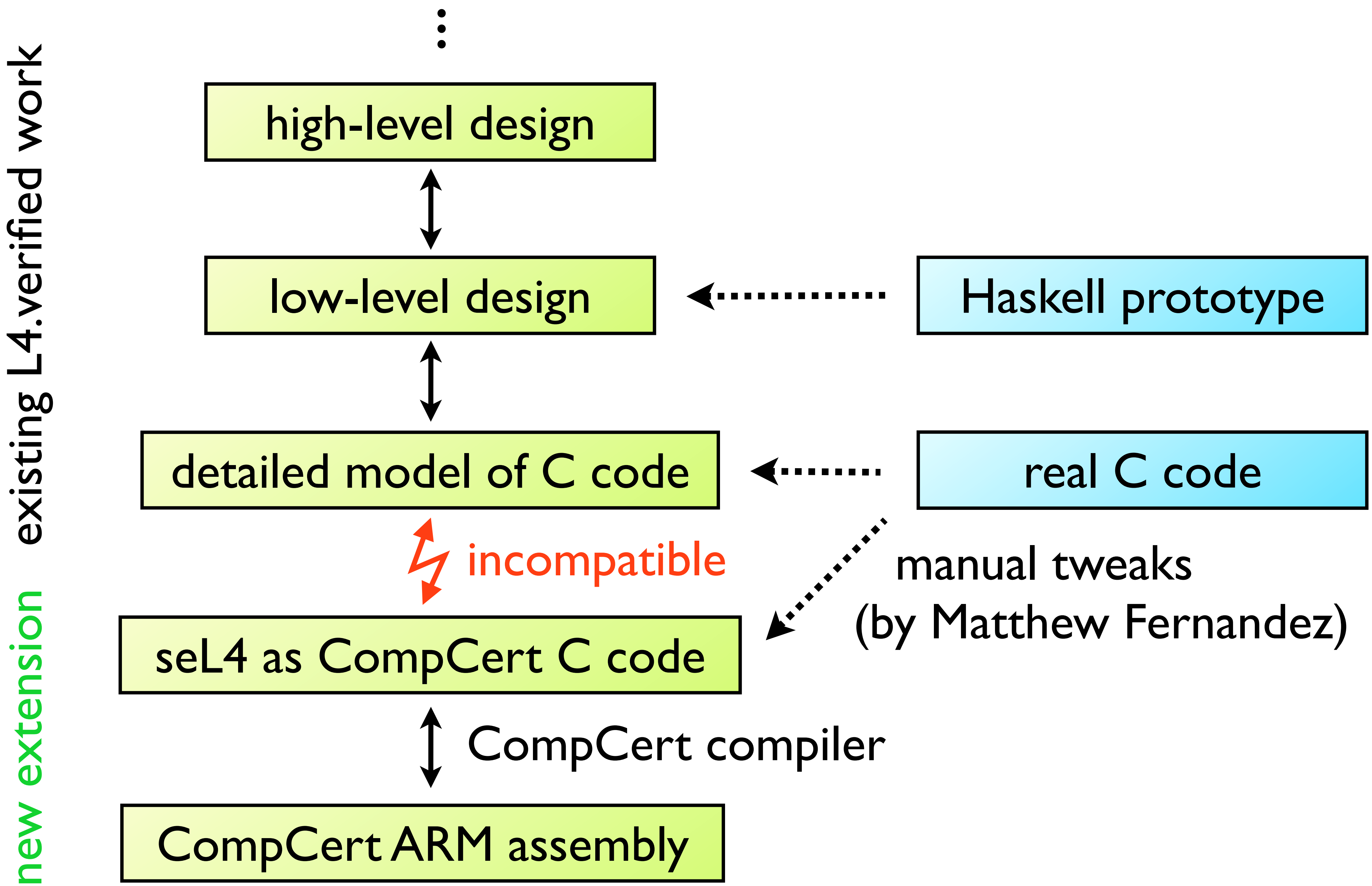
Connection to CompCert



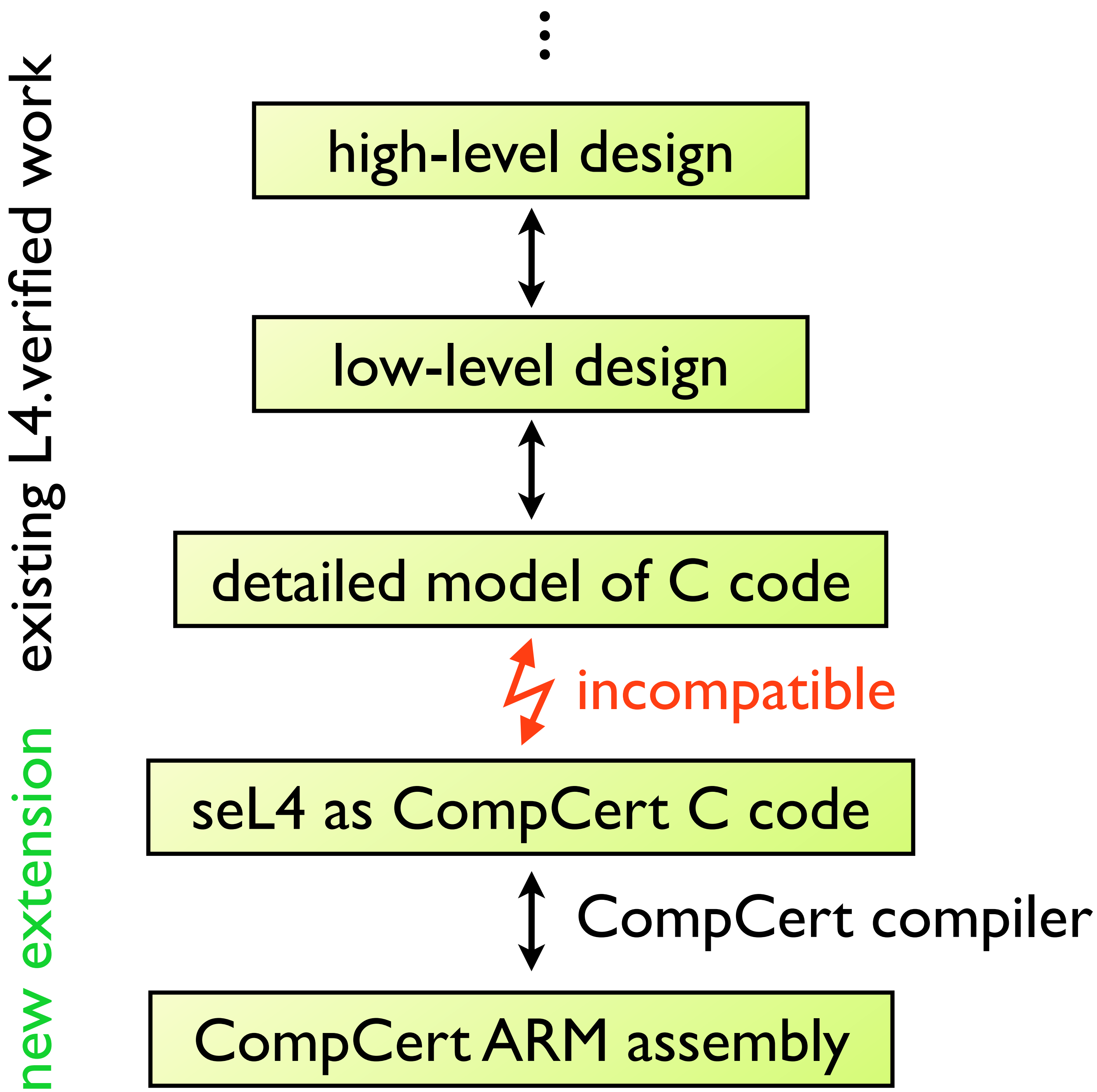
Connection to CompCert



Connection to CompCert



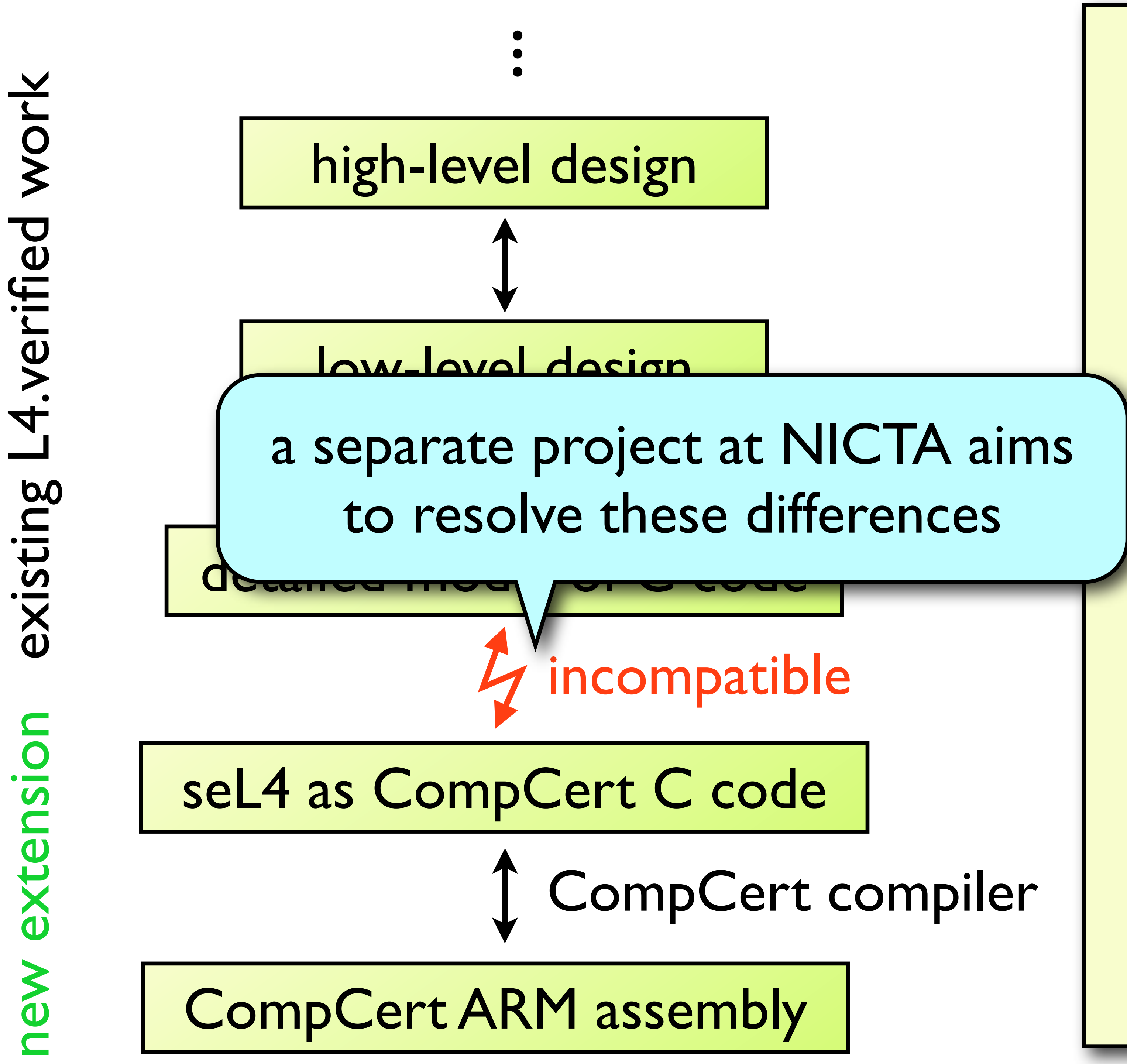
Connection to CompCert



Incompatible:

- different view on what valid C is
- pointers treated differently
- memory more abstract in CompCert C sem.
- different provers (Coq and Isabelle)

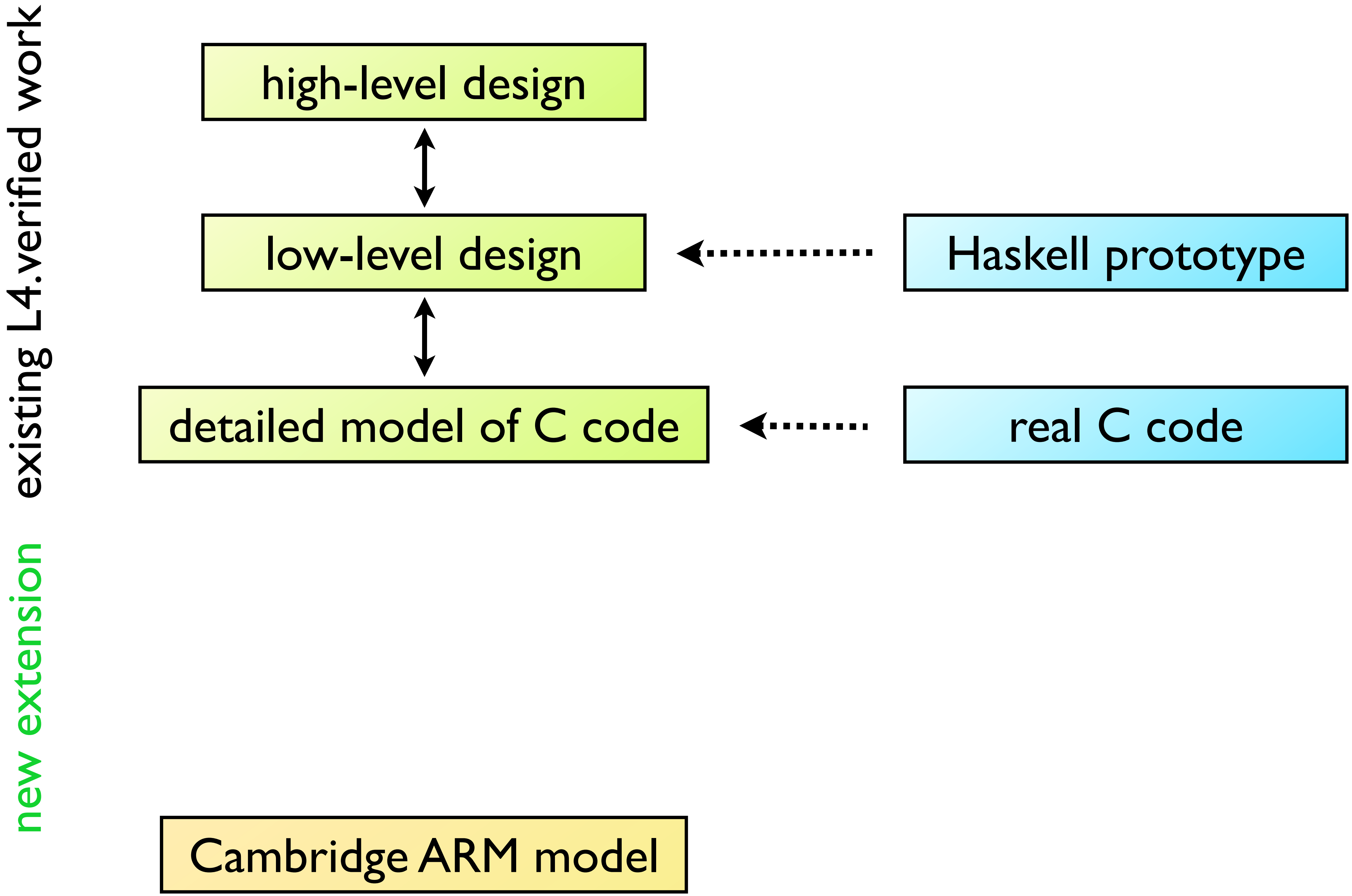
Connection to CompCert



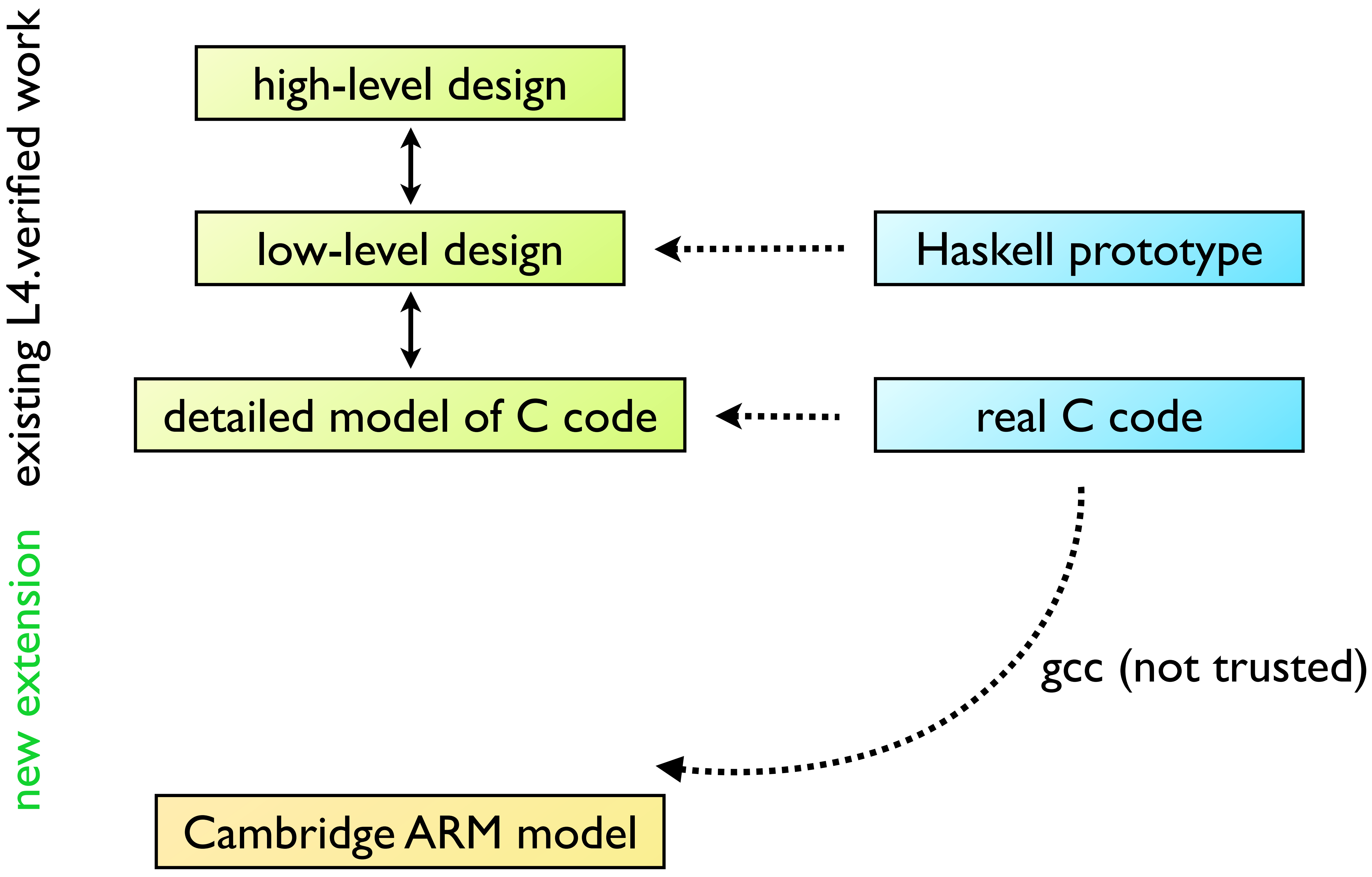
Incompatible:

- different view on what valid C is
- pointers treated differently
- memory more abstract in CompCert C sem.
- different provers (Coq and Isabelle)

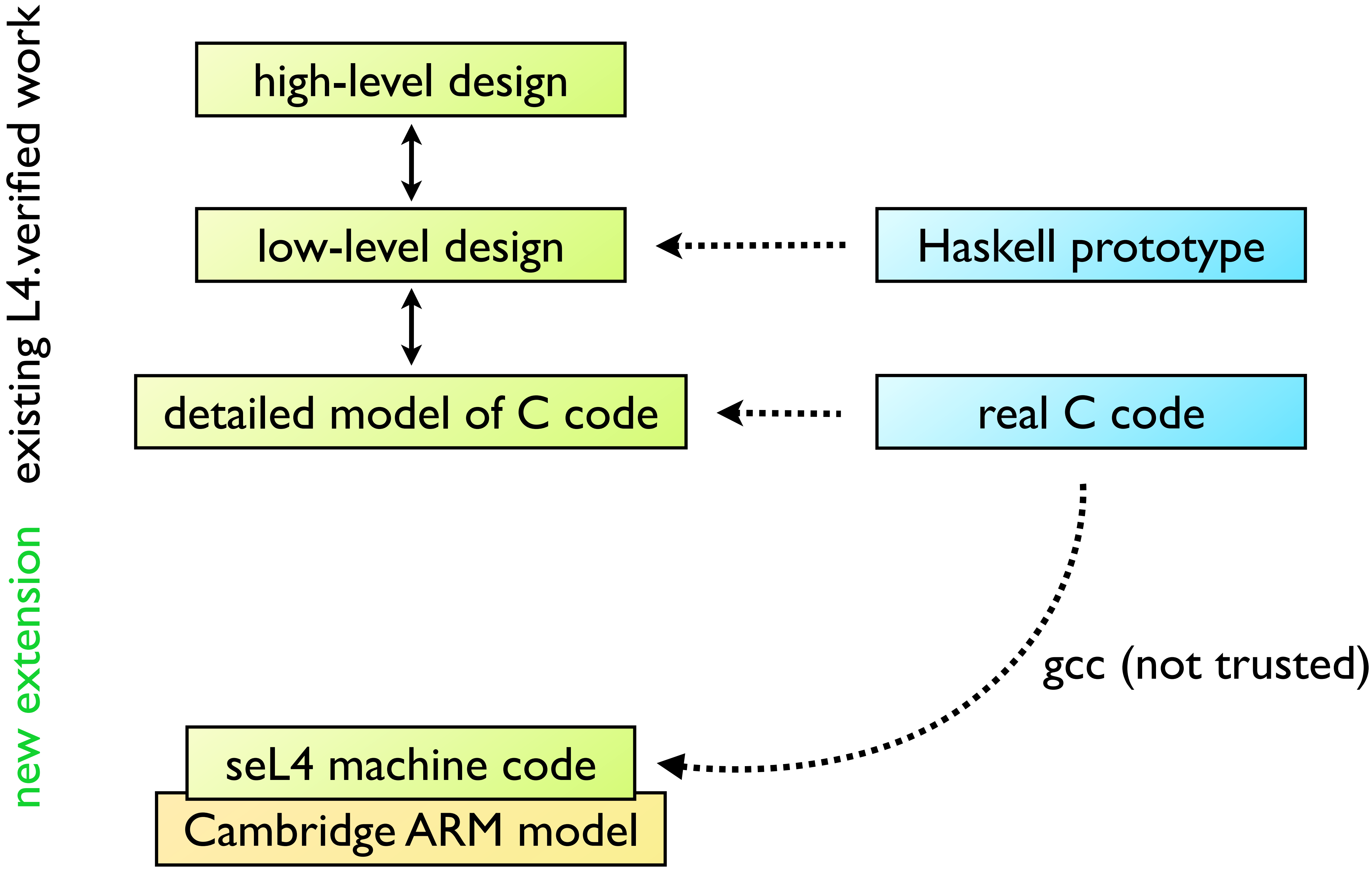
Using Cambridge ARM model



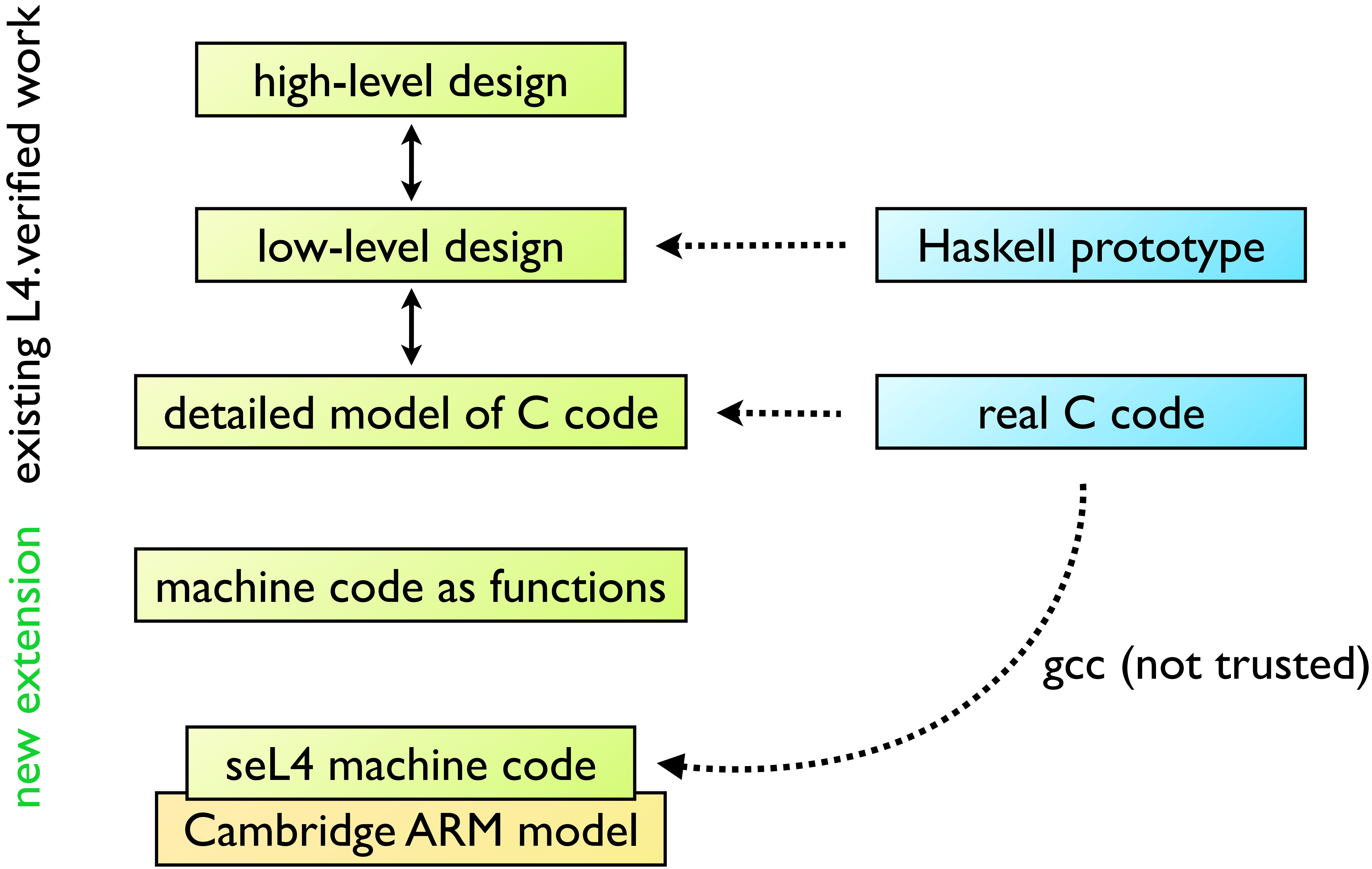
Using Cambridge ARM model



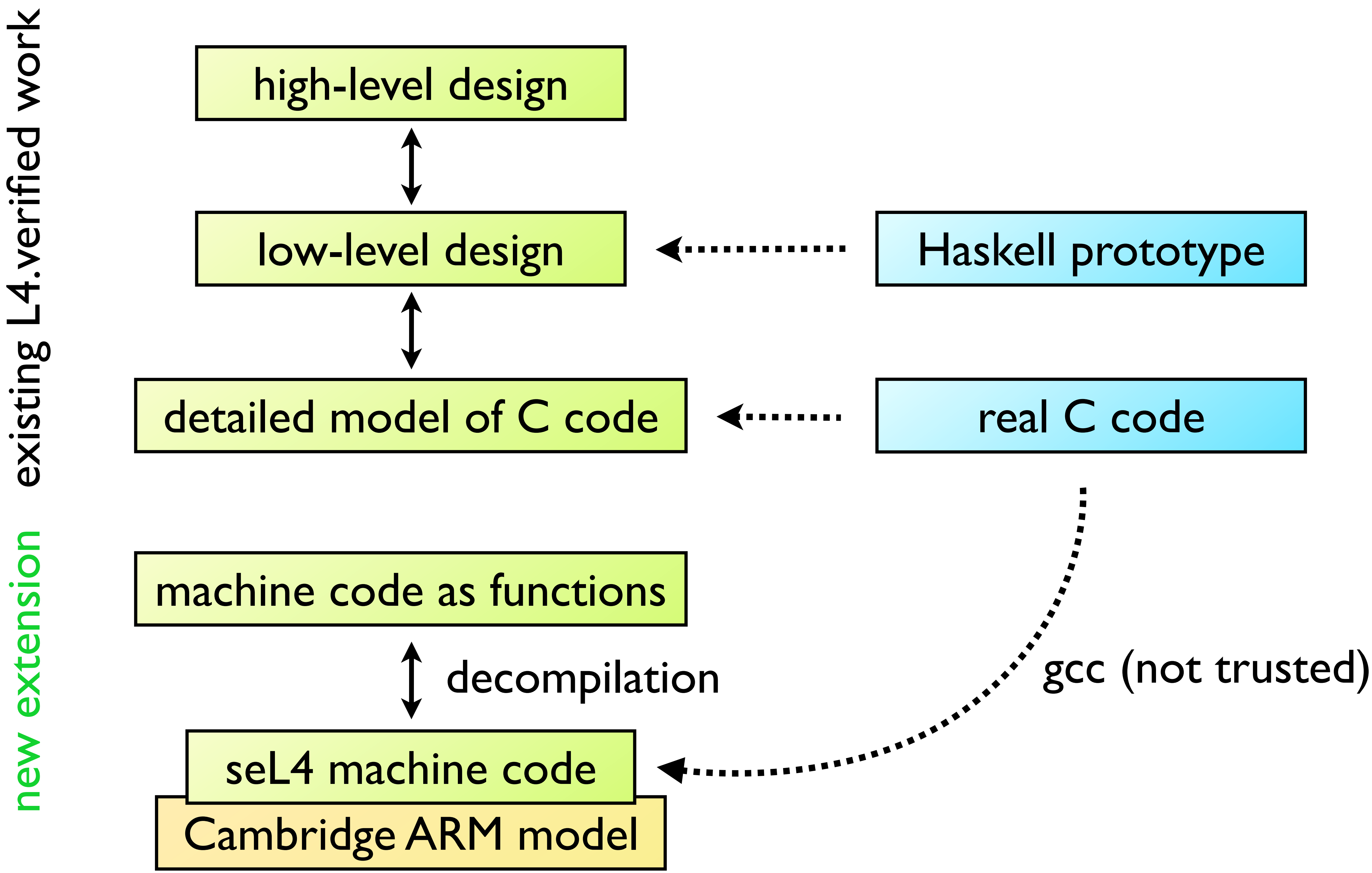
Using Cambridge ARM model



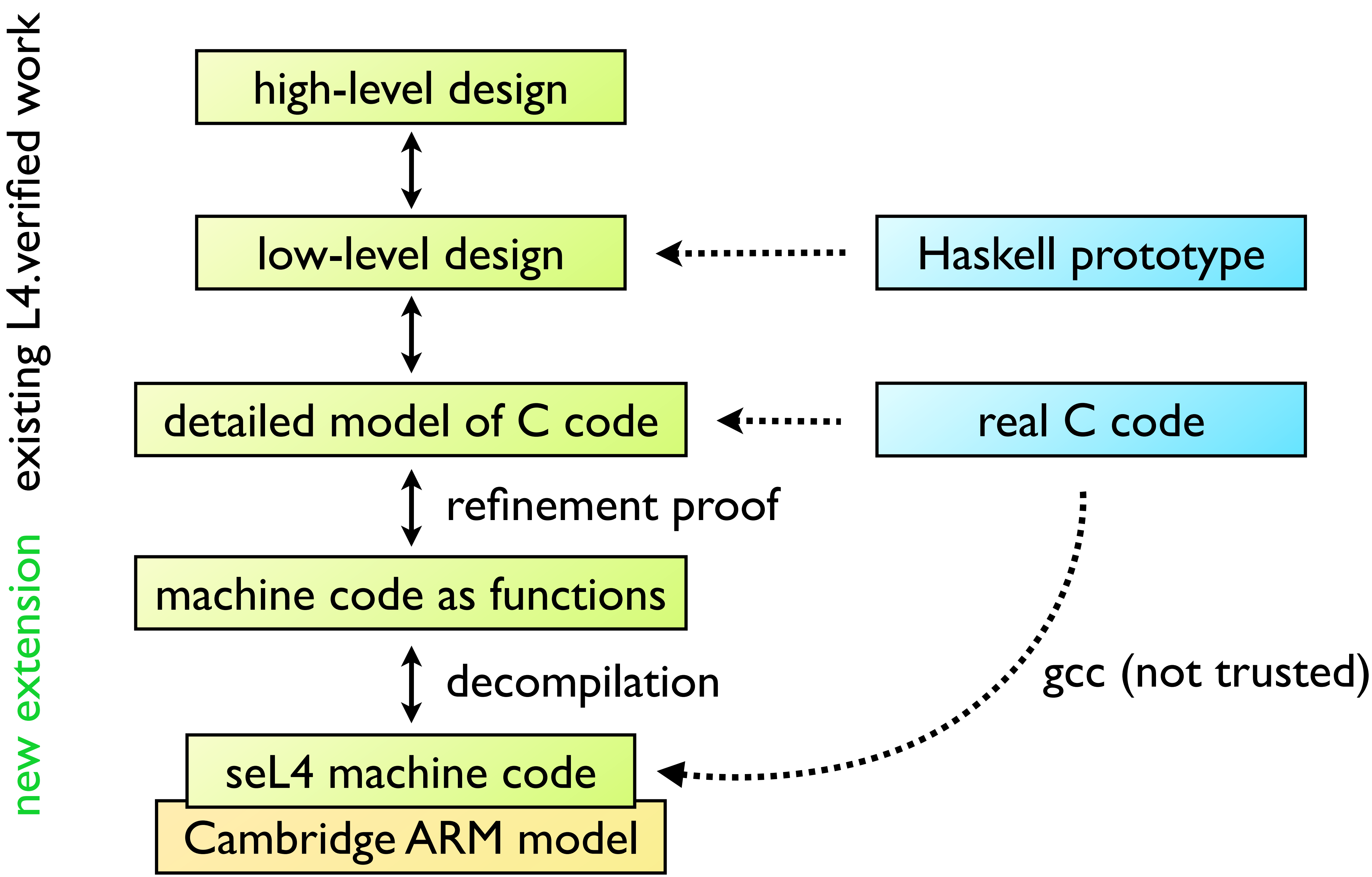
Using Cambridge ARM model



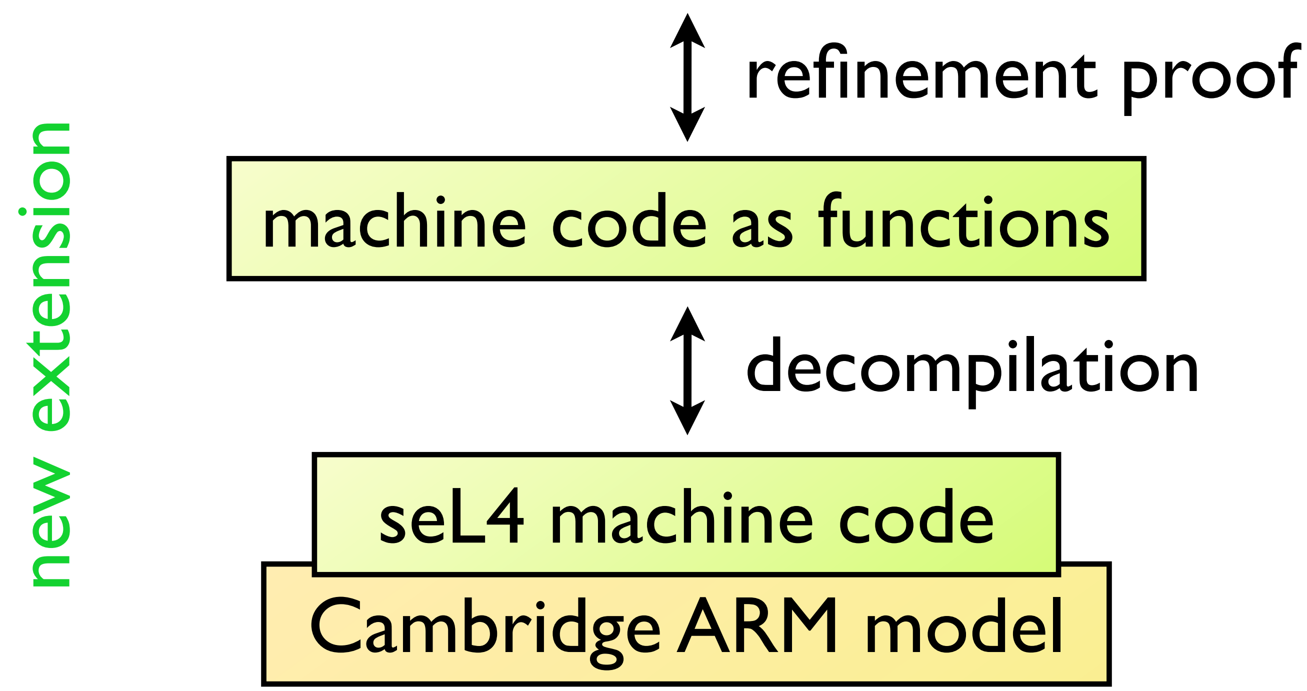
Using Cambridge ARM model



Using Cambridge ARM model



Talk outline



- automatic translation / decompilation
- progress and lessons learnt

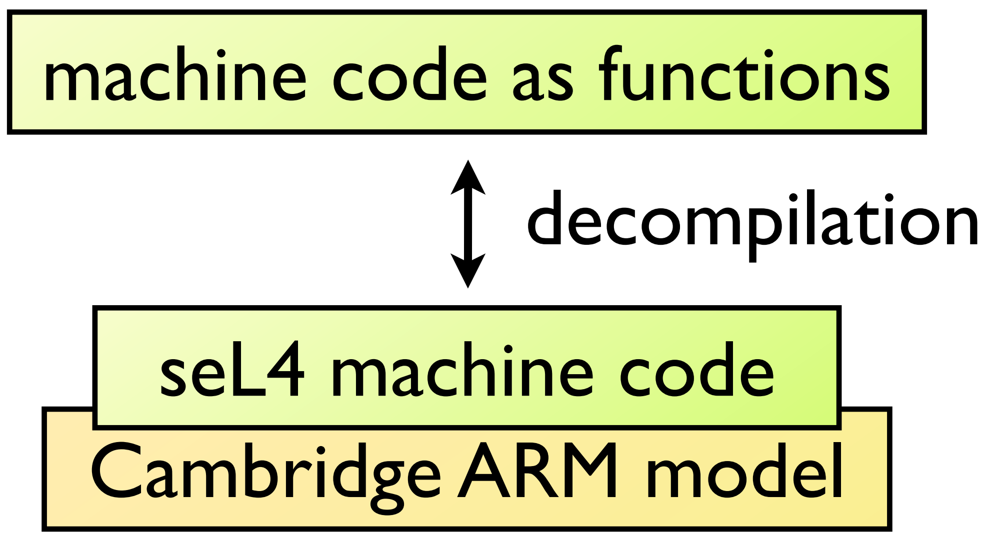
Cambridge ARM model

Cambridge ARM model developed by Anthony Fox

- high-fidelity model of the ARM instruction set architecture formalised in HOL4 theorem prover
- originates in a project on hardware verification (ARM6 verification)
- extensively tested against different hardware implementations

Web: http://www.cl.cam.ac.uk/~acjf3/arm/

Stage 1: decompilation



Decompilation

Sample C code:

```
uint avg (uint i, uint j) {
  return (i + j) / 2;
}
```

Decompilation

Sample C code:

```
uint avg (uint i, uint j) {
  return (i + j) / 2;
}
```

gcc
(not trusted)
→

machine code:

```
e0810000  add   r0, r1, r0
e1a000a0  lsr   r0, r0, #1
e12fff1e  bx    lr
```

Decompilation

Sample C code:

```
uint avg (uint i, uint j) {
  return (i + j) / 2;
}
```

gcc

(not trusted)

machine code:

```
e0810000   add   r0, r1, r0
e1a000a0   lsr   r0, r0, #1
e12fff1e   bx    lr
```

decompilation via ARM model

Resulting function:

```
avg (r0, r1) = let r0 = r1 + r0 in
               let r0 = r0 >> 1 in
                 r0
```

Decompilation

Sample C code:

```
uint avg (uint i, uint j) {
  return (i + j) / 2;
}
```

gcc
(not trusted)

machine code:

```
e0810000   add   r0, r1, r0
e1a000a0   lsr   r0, r0, #1
e12fff1e   bx    lr
```

decompilation via ARM model

Resulting function:

```
avg (r0, r1) = let r0 = r1 + r0 in
               let r0 = r0 >> 1 in
               r0
```

HOL4 certificate theorem:

```
{ R0 i * R1 j * LR lr * PC p }
 p : e0810000 e1a000a0 e12fff1e
{ R0 (avg(i,j)) * R1 _ * LR _ * PC lr }
```

Decompilation

Sample C code:

```
uint avg (uint i, uint j) {
  return (i + j) / 2;
}
```

gcc
(not trusted)

machine code:

```
e0810000   add   r0, r1, r0
e1a000a0   lsr   r0, r0, #1
e12fff1e   bx    lr
```

return instruction

decompilation

Resulting function:

```
avg (r0, r1) = let r0 = r1 + r0 in
               let r0 = r0 >> 1 in
                 r0
```

HOL4 certificate theorem:

{ R0 i * R1 j * LR lr * PC p }
p : e0810000 e1a000a0 e12fff1e
{ R0 (avg(i,j)) * R1 _ * LR _ * PC lr }

Decompilation

Sample C code:

```
uint avg (uint i, uint j) {
  return (i + j) / 2;
}
```

gcc (not trusted)

machine code:

```
e0810000   add   r0, r1, r0
e1a000a0   lsr   r0, r0, #1
e12fff1e   bx    lr
```

return instruction

decompilation

bit-string arithmetic

Resulting function:

```
avg (r0, r1) = let r0 = r1 + r0 in
               let r0 = r0 >> 1 in
                r0
```

HOL4 certificate theorem:

{ R0 i * R1 j * LR lr * PC p }
p : e0810000 e1a000a0 e12fff1e
{ R0 (avg(i,j)) * R1 _ * LR _ * PC lr }

Decompilation

Sample C code:

```
uint avg (uint i, uint j) {
  return (i + j) / 2;
}
```

gcc

(not trusted)

machine code:

```
e0810000   add   r0, r1, r0
e1a000a0   lsr   r0, r0, #1
e12fff1e   bx    lr
```

return instruction

decompilation

bit-string arithmetic

Resulting function:

avg (r0, r1) = let r0 = r1 + r0 in
let r0 = r0 >> 1 in
r0

bit-string right-shift

HOL4 certificate theorem:

{ R0 i * R1 j * LR lr * PC p }
p : e0810000 e1a000a0 e12fff1e
{ R0 (avg(i,j)) * R1 _ * LR _ * PC lr }

Decompilation

Sample C code:

```
uint avg (uint i, uint j) {
  return (i + j) / 2;
}
```

gcc
(not trusted)

machine code:

```
e0810000   add   r0, r1, r0
e1a000a0   lsr   r0, r0, #1
e12fff1e   bx    lr
```

return instruction

decompilation

bit-string arithmetic

Resulting function:

```
avg (r0, r1) = let r0 = r1 + r0 in
               let r0 = r0 >> 1 in
               r0
```

bit-string right-shift

HOL4 certificate theorem:

{ R0 i * R1 j * LR lr * PC p }
p : e0810000 e1a000a0 e12fff1e
{ R0 (avg(i,j)) * R1 _ * LR _ * PC lr }

separation logic: *

Decompilation

How to decompile:

```
e0810000  add   r0, r1, r0
e1a000a0  lsr   r0, r0, #1
e12fff1e  bx    lr
```

Decompilation

How to decompile:

```
e0810000

e1a000a0

e12fff1e
```

```
e0810000  add   r0, r1, r0
e1a000a0  lsr   r0, r0, #1
e12fff1e  bx    lr
```

Decompilation

{ R0 i * R1 j * PC p }
p+0 : e0810000
{ R0 (i+j) * R1 j * PC (p+4) }

{ R0 i * PC (p+4) }
p+4 : e1a000a0
{ R0 (i >> 1) * PC (p+8) }

{ LR lr * PC (p+8) }
p+8 : e12fff1e
{ LR lr * PC lr }

How to decompile:

```
e0810000  add   r0, r1, r0
e1a000a0  lsr   r0, r0, #1
e12fff1e  bx    lr
```

1. derive Hoare triple theorems using Cambridge ARM model

Decompilation

{ R0 i * R1 j * PC p }
p+0 : e0810000
{ R0 (i+j) * R1 j * PC (p+4) }

{ R0 i * PC (p+4) }
p+4 : e1a000a0
{ R0 (i >> 1) * PC (p+8) }

{ LR lr * PC (p+8) }
p+8 : e12fff1e
{ LR lr * PC lr }

2

{ R0 i * R1 j * LR lr * PC p }
p : e0810000 e1a000a0 e12fff1e
{ R0 ((i+j)>>1) * R1 j * LR lr * PC lr }

How to decompile:

```
e0810000   add   r0, r1, r0
e1a000a0   lsr   r0, r0, #1
e12fff1e   bx    lr
```

1. derive Hoare triple theorems using Cambridge ARM model
2. compose Hoare triples

Decompilation

{ R0 i * R1 j * PC p }
p+0 : e0810000
{ R0 (i+j) * R1 j * PC (p+4) }

{ R0 i * PC (p+4) }
p+4 : e1a000a0
{ R0 (i >> 1) * PC (p+8) }

{ LR lr * PC (p+8) }
p+8 : e12fff1e
{ LR lr * PC lr }

2

{ R0 i * R1 j * LR lr * PC p }
p : e0810000 e1a000a0 e12fff1e
{ R0 ((i+j)>>1) * R1 j * LR lr * PC lr }

3

avg (i,j) = (i+j)>>1

How to decompile:

```
e0810000   add   r0, r1, r0
e1a000a0   lsr   r0, r0, #1
e12fff1e   bx    lr
```

1. derive Hoare triple theorems using Cambridge ARM model
2. compose Hoare triples
3. extract function

(Loops result in recursive functions.)

Decompiling seL4: Challenges

- seL4 is ~12,000 lines of machine code
- compiled using gcc -O2
- must be compatible with L4.verified proof

Decompiling seL4: Challenges

- seL4 is ~12,000 lines of machine code
  - ✓ decompilation is compositional
- compiled using gcc -O2
- must be compatible with L4.verified proof

Decompiling seL4: Challenges

- seL4 is ~12,000 lines of machine code
  - ✓ decompilation is compositional
- compiled using gcc -O2
  - ✓ gcc implements ARM/C calling convention
- must be compatible with L4.verified proof

Decompiling seL4: Challenges

- seL4 is ~12,000 lines of machine code
  - ✓ decompilation is compositional
- compiled using gcc -O2
  - ✓ gcc implements ARM/C calling convention
- must be compatible with L4.verified proof
  - ➡ stack requires special treatment

Stack visible in m. code

C code:

```
uint avg8 (uint x0, x1, x2, x3, x4, x5, x6, x7) {
  return (x0+x1+x2+x3+x4+x5+x6+x7) / 8;
}
```

Stack visible in m. code

C code:

```
uint avg8 (uint x0, x1, x2, x3, x4, x5, x6, x7) {
  return (x0+x1+x2+x3+x4+x5+x6+x7) / 8;
}
```

Some arguments are passed on the stack,

Stack visible in m. code

C code:

```
uint avg8 (uint x0, x1, x2, x3, x4, x5, x6, x7) {
  return (x0+x1+x2+x3+x4+x5+x6+x7) / 8;
}
```

Some arguments are passed on the stack,

gcc

```
add   r1, r1, r0
add   r1, r1, r2
ldr   r2, [sp]
add   r1, r1, r3
add   r0, r1, r2
ldmib sp, {r2, r3}
add   r0, r0, r2
add   r0, r0, r3
ldr   r3, [sp, #12]
add   r0, r0, r3
lsr   r0, r0, #3
bx    lr
```

Stack visible in m. code

C code:

```
uint avg8 (uint x0, x1, x2, x3, x4, x5, x6, x7) {
  return (x0+x1+x2+x3+x4+x5+x6+x7) / 8;
}
```

gcc

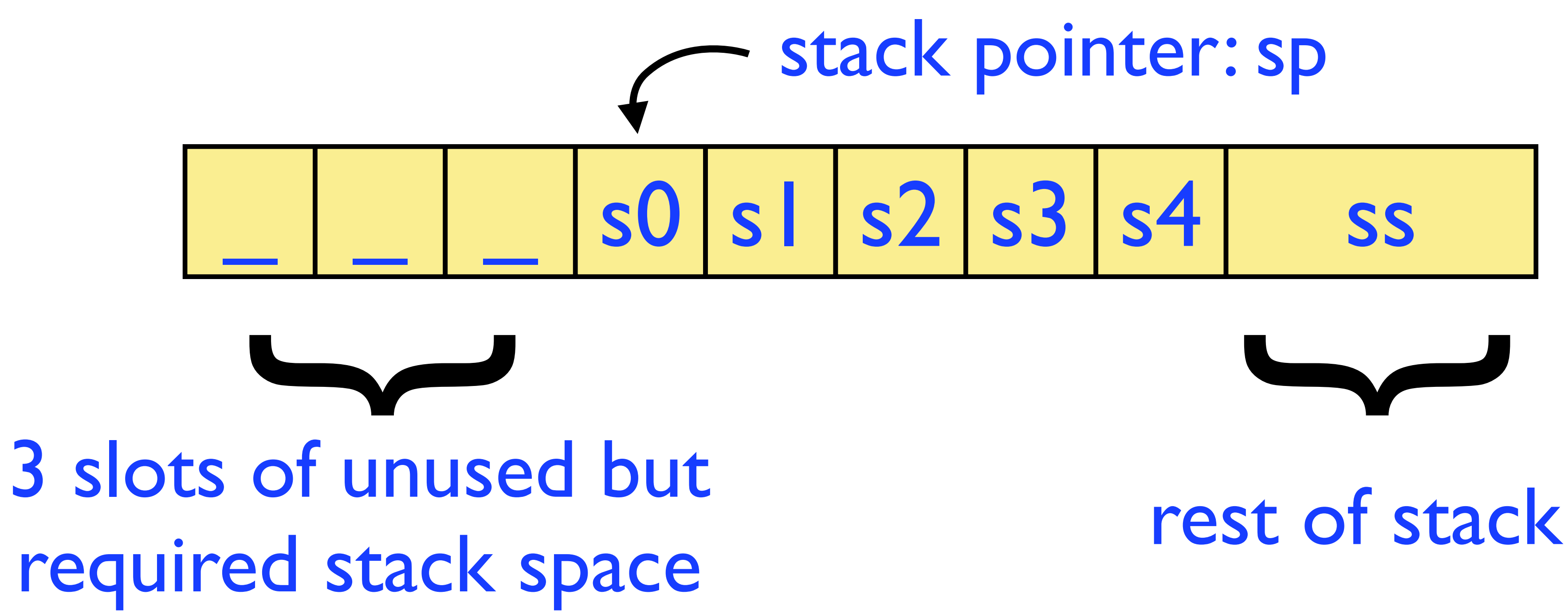
```
add   r1, r1, r0
add   r1, r1, r2
ldr   r2, [sp]
add   r1, r1, r3
add   r0, r1, r2
ldmib sp, {r2, r3}
add   r0, r0, r2
add   r0, r0, r3
ldr   r3, [sp, #12]
add   r0, r0, r3
lsr   r0, r0, #3
bx    lr
```

Some arguments are passed on the stack, and cause memory ops in machine code

... that are not present in C semantics.

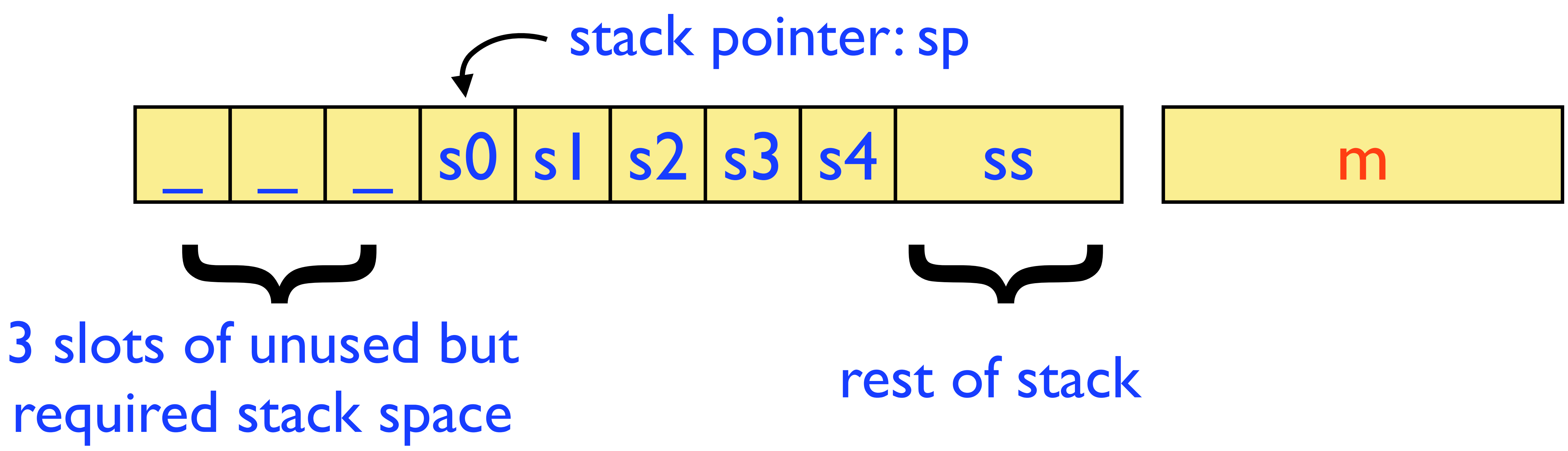
Solution

Use separation-logic inspired approach



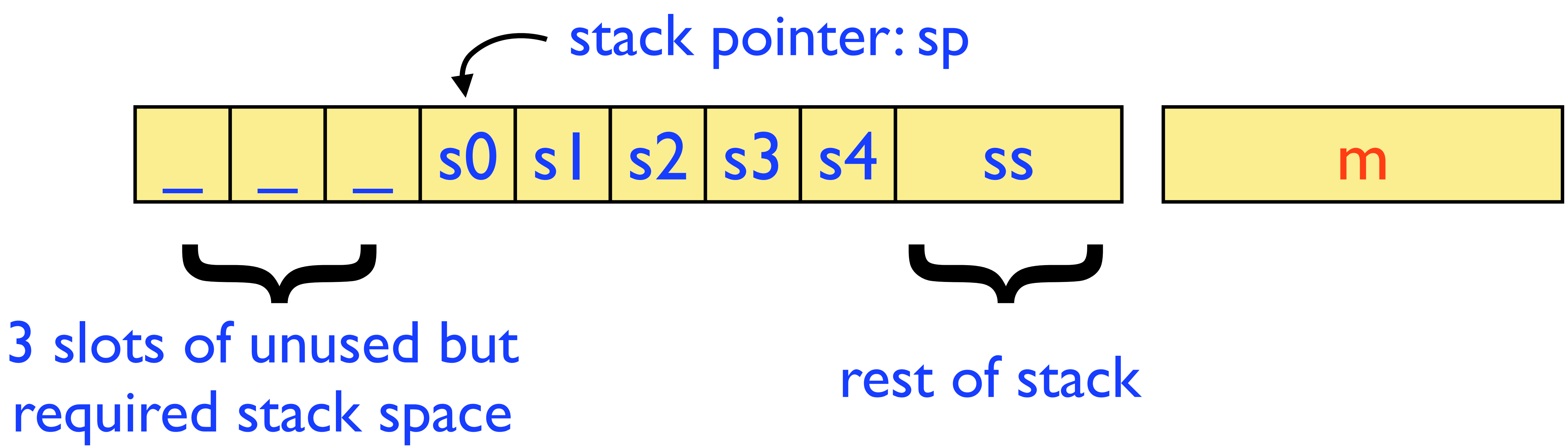
Solution

Use separation-logic inspired approach



Solution

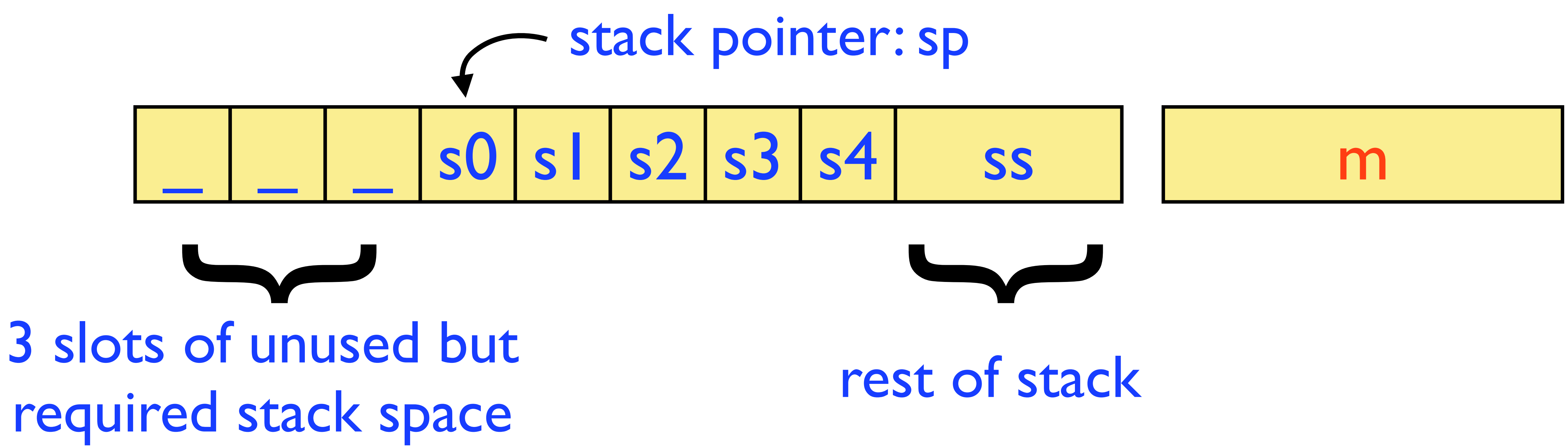
Use separation-logic inspired approach



stack sp 3 (s0::s1::s2::s3::s4::ss)

Solution

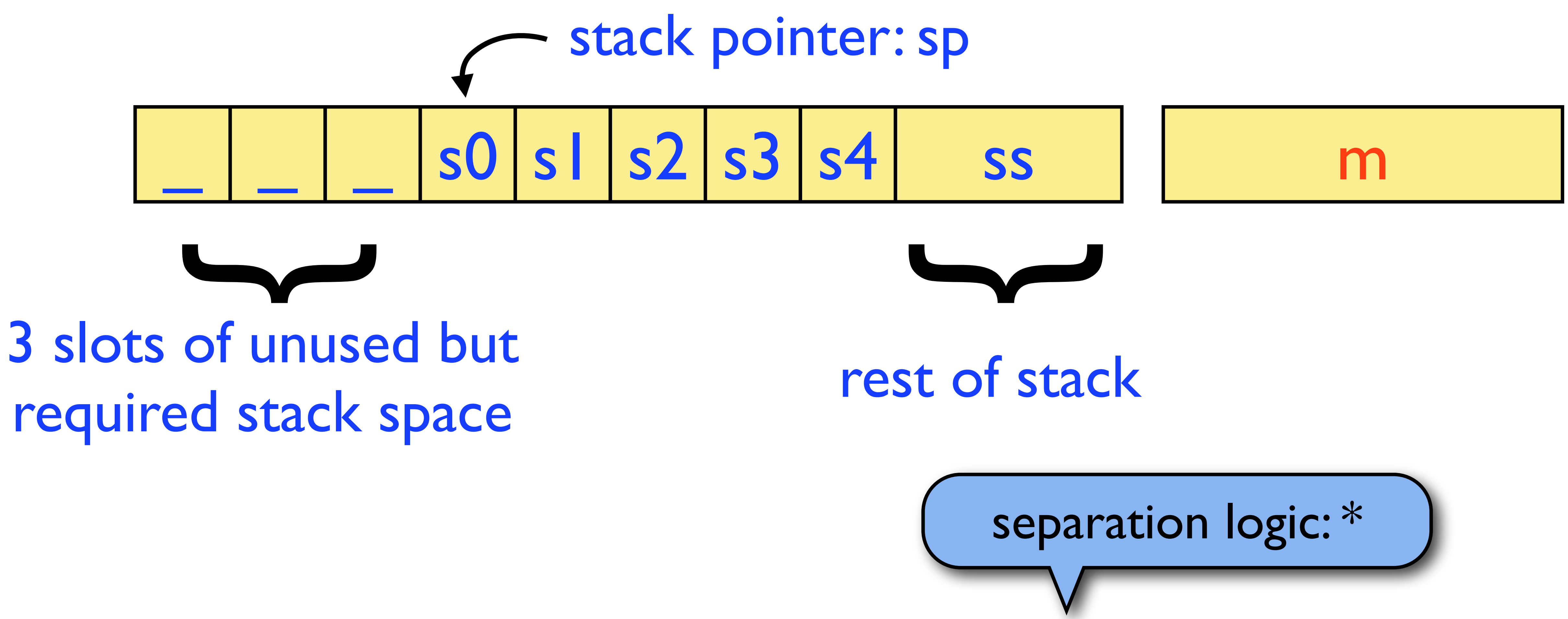
Use separation-logic inspired approach



stack sp 3 (s0::s1::s2::s3::s4::ss) * memory m

Solution

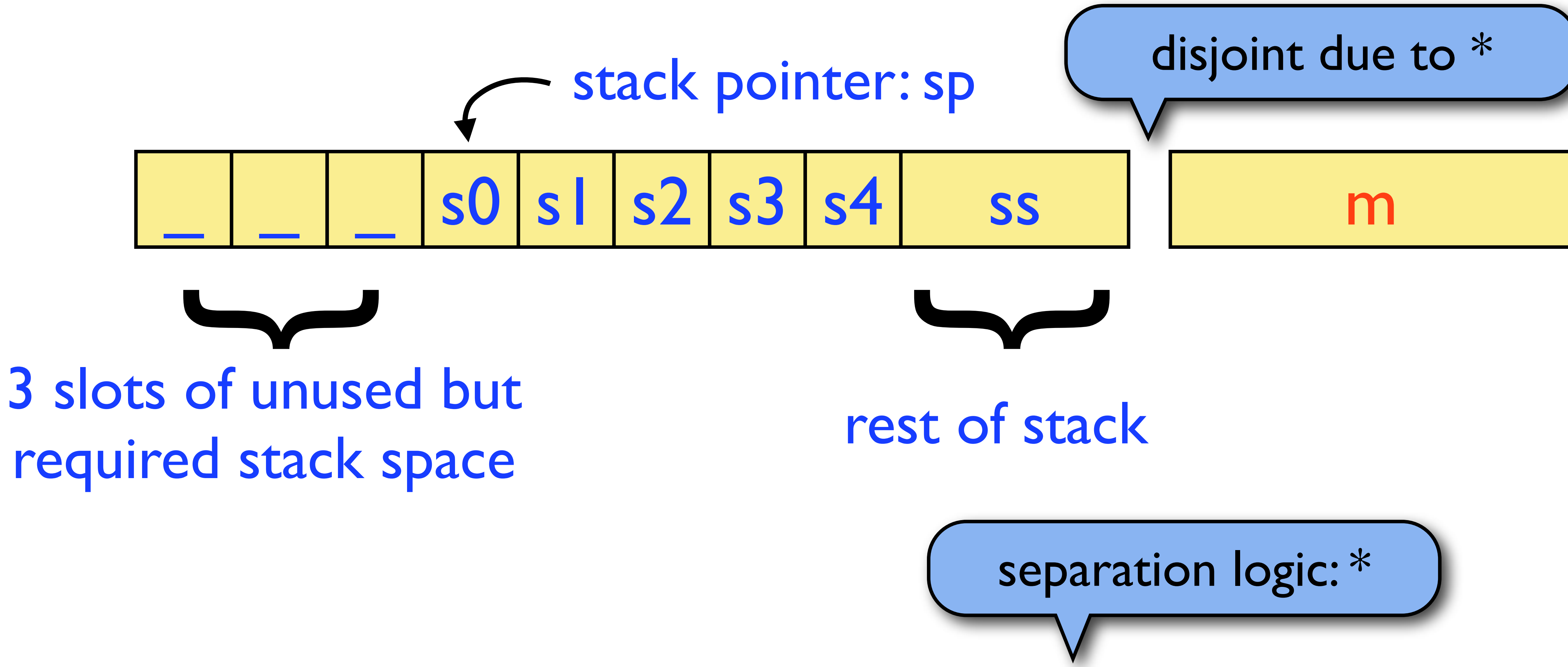
Use separation-logic inspired approach



stack sp 3 (s0::s1::s2::s3::s4::ss) * memory m

Solution

Use separation-logic inspired approach



stack sp 3 (s0::s1::s2::s3::s4::ss) * memory m

Solution (cont.)

```
add r1, r1, r0
add r1, r1, r2
ldr  r2, [sp]
add r1, r1, r3
add r0, r1, r2
ldmib sp, {r2, r3}
add r0, r0, r2
add r0, r0, r3
ldr  r3, [sp, #12]
add r0, r0, r3
lsr  r0, r0, #3
bx   lr
```

Method:

1. static analysis to find stack operations,
2. derive stack-specific Hoare triples,
3. then run decompiler as before.

Solution (cont.)

```
add   r1, r1, r0
add   r1, r1, r2
ldr   r2, [sp]
add   r1, r1, r3
add   r0, r1, r2
ldmib sp, {r2, r3}
add   r0, r0, r2
add   r0, r0, r3
ldr   r3, [sp, #12]
add   r0, r0, r3
lsr   r0, r0, #3
bx    lr
```

Method:

1. static analysis to find stack operations,
2. derive stack-specific Hoare triples,
3. then run decompiler as before.

Result

Stack load/stores become straightforward assignments.

```
add r1, r1, r0
add r1, r1, r2
ldr   r2, [sp]
add r1, r1, r3
add r0, r1, r2
ldmib sp, {r2, r3}
add r0, r0, r2
add r0, r0, r3
ldr   r3, [sp, #12]
add r0, r0, r3
lsr   r0, r0, #3
bx   lr
```

→

```
avg8(r0,r1,r2,r3,s0,s1,s2,s3) =
  let r1 = r1 + r0 in
  let r1 = r1 + r2 in
  let r2 = s0 in
  let r1 = r1 + r3 in
  let r0 = r1 + r3 in
  let (r2,r3) = (s1,s2) in
  let r0 = r0 + r2 in
  let r0 = r0 + r3 in
  let r3 = s3 in
  let r0 = r0 + r3 in
  let r0 = r0 >> 3 in
    r0
```

Result

Stack load/stores become straightforward assignments.

<u>Additional benefit:</u>

automatically proved certificate theorem
states explicitly stack shape/usage:

{ stack sp n (s0::s1::s2::s3::s) * ... * PC p }
p : code
{ stack sp n (s0::s1::s2::s3::s) * ... * PC lr }

```
bx   lr
```

```
r0
```

Result

Stack load/stores become straightforward assignments.

<u>Additional benefit:</u>

automatically proved certificate theorem

states explicitly st

four arguments passed on stack

{ stack sp n (s0::s1::s2::s3::s) * ... * PC p }

p : code

{ stack sp n (s0::s1::s2::s3::s) * ... * PC lr }

bx lr

r0

Result

Stack load/stores become straightforward assignments.

Additional benefit:

automati

states explici y st

does not require temp space, works for “any n”

four arguments passed on stack

{ stack sp n (s0::s1::s2::s3::s) * ... * PC p }
p : code
{ stack sp n (s0::s1::s2::s3::s) * ... * PC lr }

bx lr

r0

Result

Stack load/stores become straightforward assignments.

<u>Additional benefit:</u>

automati

states explici y st

does not require temp space, works for “any n”

four arguments passed on stack

{ stack sp n (s0::s1::s2::s3::s) * ... * PC p }
 p : code
{ stack sp n (s0::s1::s2::s3::s) * ... * PC lr }

promises to leave stack unchanged

bx   lr

r0

Other C-specifics

- struct as return value
  - case of passing pointer of stack location
  - stack assertion strong enough
- switch statements
  - position dependent
  - must decompile elf-files, not object files
- infinite loops in C
  - make gcc go weird
  - must be pruned from control-flow graph

Progress

A 6-week visit to NICTA resulted in:

75 % of seL4 decompiled

Progress

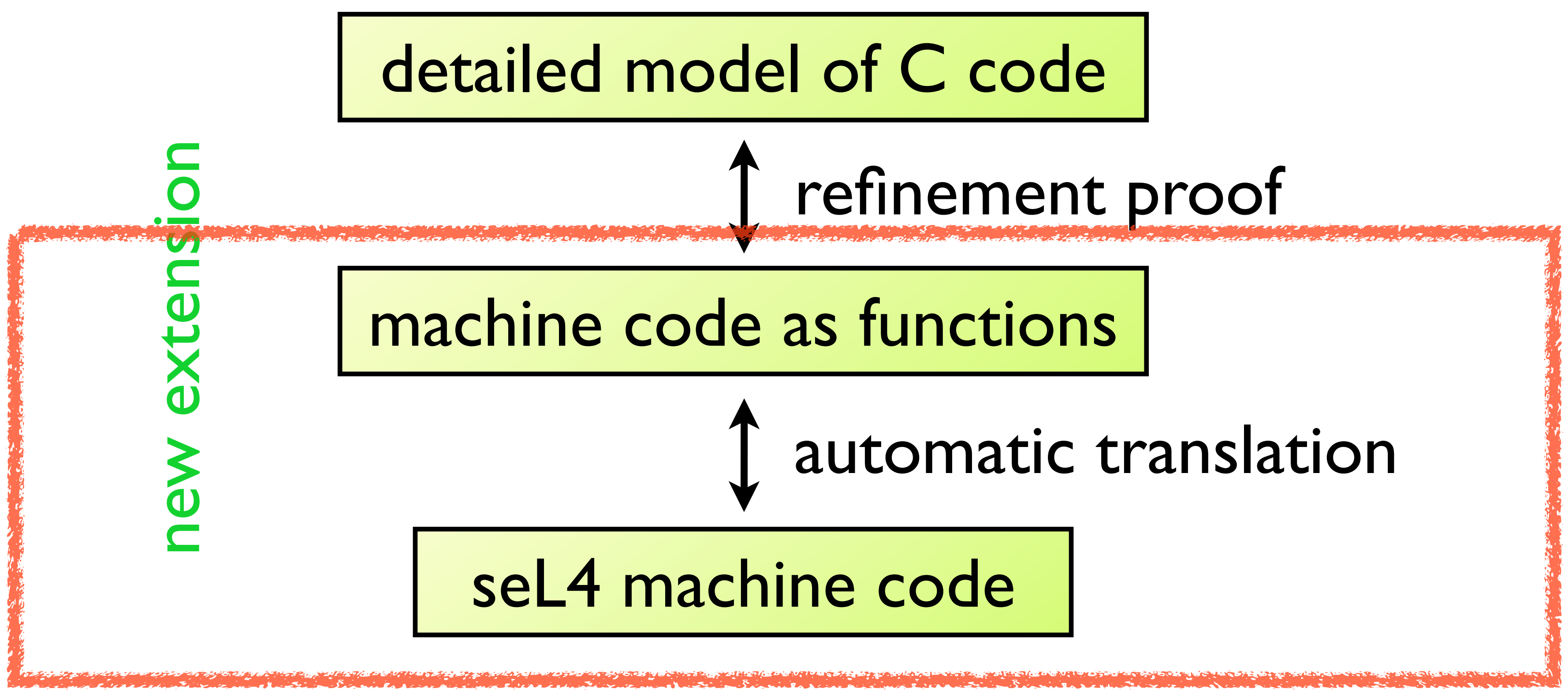
A 6-week visit to NICTA resulted in:

75 % of seL4 decompiled

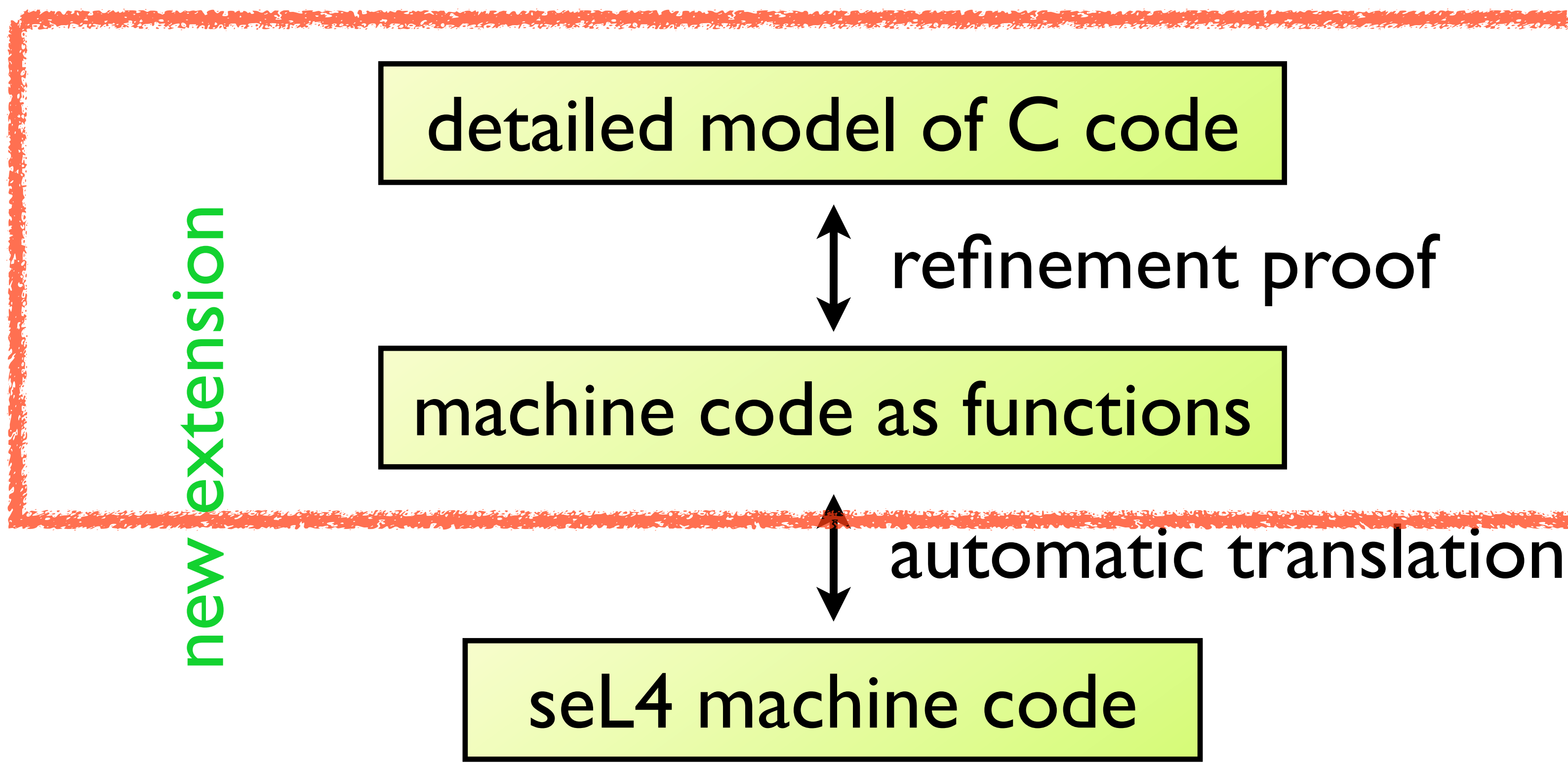
Next visit scheduled for end of this year:

- complete decompilation
  (make stack heuristic stronger)
- concentrate on stage 2... (next slide)

Moving on to stage 2



Moving on to stage 2



Proving C refinement

Approach 1:

- use a verification condition generator (VCG) to prove C Hoare-triple theorems, approximately:

  { true } code { state_after = code_fun(state_before) }

Aim:

- make solution as automatic as possible
- must deal with reordering of load/store instructions

Proving C refinement

Approach 1:

- use a verification condition generator (VCG) to prove C Hoare-triple theorems, approximately:

{ true } code { state_after = code_fun(state_before) }

simplified for easier presentation

Aim:

- make solution as automatic as possible
- must deal with reordering of load/store instructions

Proving C refinement

Approach 2:

- compose C code inside existing correctness C Hoare triple, e.g.

$$\{\ \text{pre}\ \}\ (\text{Assign } f;\ \text{Assign } g)\ \{\ \text{post}\ \} \;=\; \{\ \text{pre}\ \}\ (\text{Assign } (g \circ f))\ \{\ \text{post}\ \}$$

- then prove, for almost any pre, post:

$$\{\ \text{pre}\ \}\ \text{code}\ \{\ \text{post}\ \} \;\Longrightarrow\; \{\ \text{pre}\ \}\ (\text{Assign code\_fun})\ \{\ \text{post}\ \}$$

Proving C refinement

Approach 2:

Solution to inlined assembly:

naturally compatible with decompilations of inlined assembly, e.g.

{ pre } (Assign inline_asm_fun) { post }

Gets around the problem of C's __asm__.

{ pre } (Assign code_fun) { post }

Final part:

Lessons learnt

gcc: weird and wonderful

Wonderful:

- gcc -O2 produces good/clever code
- decompilation can be made to work on its output
- gcc -O0 produces simple “reference” machine code

Weird:

- fails to spot a few ‘obvious’ optimisations
- gcc -O2 sometimes invents new subroutines

Hardest part?

Hardest part?

So far: connection with C semantics.

Hardest part?

So far: connection with C semantics.

C semantics best avoided?

Hardest part?

So far: connection with C semantics.

C semantics best avoided?

Ideally avoid C altogether:

- use verification-friendly domain-specific language

Hardest part?

So far: connection with C semantics.

C semantics best avoided?

Ideally avoid C altogether:

- use verification-friendly domain-specific language

HASP?

Hardest part?

So far: connection with C semantics.

C semantics best avoided?

Ideally avoid C altogether:

- use verification-friendly domain-specific language

HASP?

... but C is the reality of OS code

- a simple “hacker’s semantics of C” ?

“a hacker’s C semantics”

“a hacker’s C semantics”

Possibility: use decompilation from gcc -O0
as semantics of C code.

“a hacker’s C semantics”

Possibility: use decompilation from gcc -O0
as semantics of C code.

✓ approach reflects the observation:
“OS hackers use C as convenient way to write assembly”

“a hacker’s C semantics”

Possibility: use decompilation from gcc -O0
as semantics of C code.

✓ approach reflects the observation:
“OS hackers use C as convenient way to write assembly”

✓ potentially simpler than current C semantics

“a hacker’s C semantics”

Possibility: use decompilation from gcc -O0
as semantics of C code.

- ✓ approach reflects the observation:
  “OS hackers use C as convenient way to write assembly”
- ✓ potentially simpler than current C semantics
- ✓ does not require trusting gcc
  - proof relates only to the generated machine code

“a hacker’s C semantics”

Possibility: use decompilation from gcc -O0 as semantics of C code.

- ✓ approach reflects the observation: “OS hackers use C as convenient way to write assembly”
- ✓ potentially simpler than current C semantics
- ✓ does not require trusting gcc
  - proof relates only to the generated machine code
- ✓ separately prove transition from gcc -O0 to gcc -O2

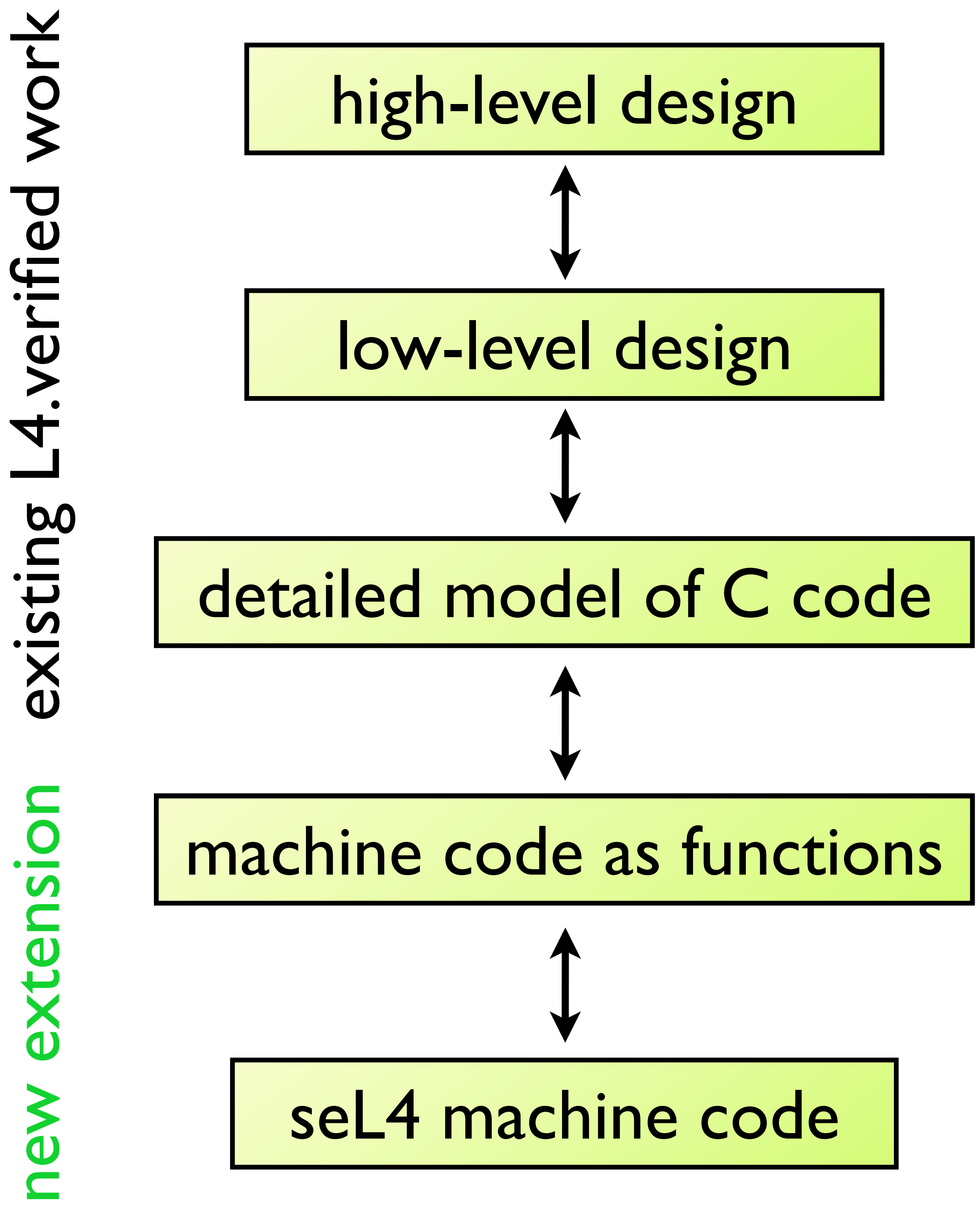
“a hacker’s C semantics”

Possibility: use decompilation from gcc -O0
as semantics of C code.

- ✓ approach reflects the observation:
  “OS hackers use C as convenient way to write assembly”
- ✓ potentially simpler than current C semantics
- ✓ does not require trusting gcc
  - proof relates only to the generated machine code
- ✓ separately prove transition from gcc -O0 to gcc -O2

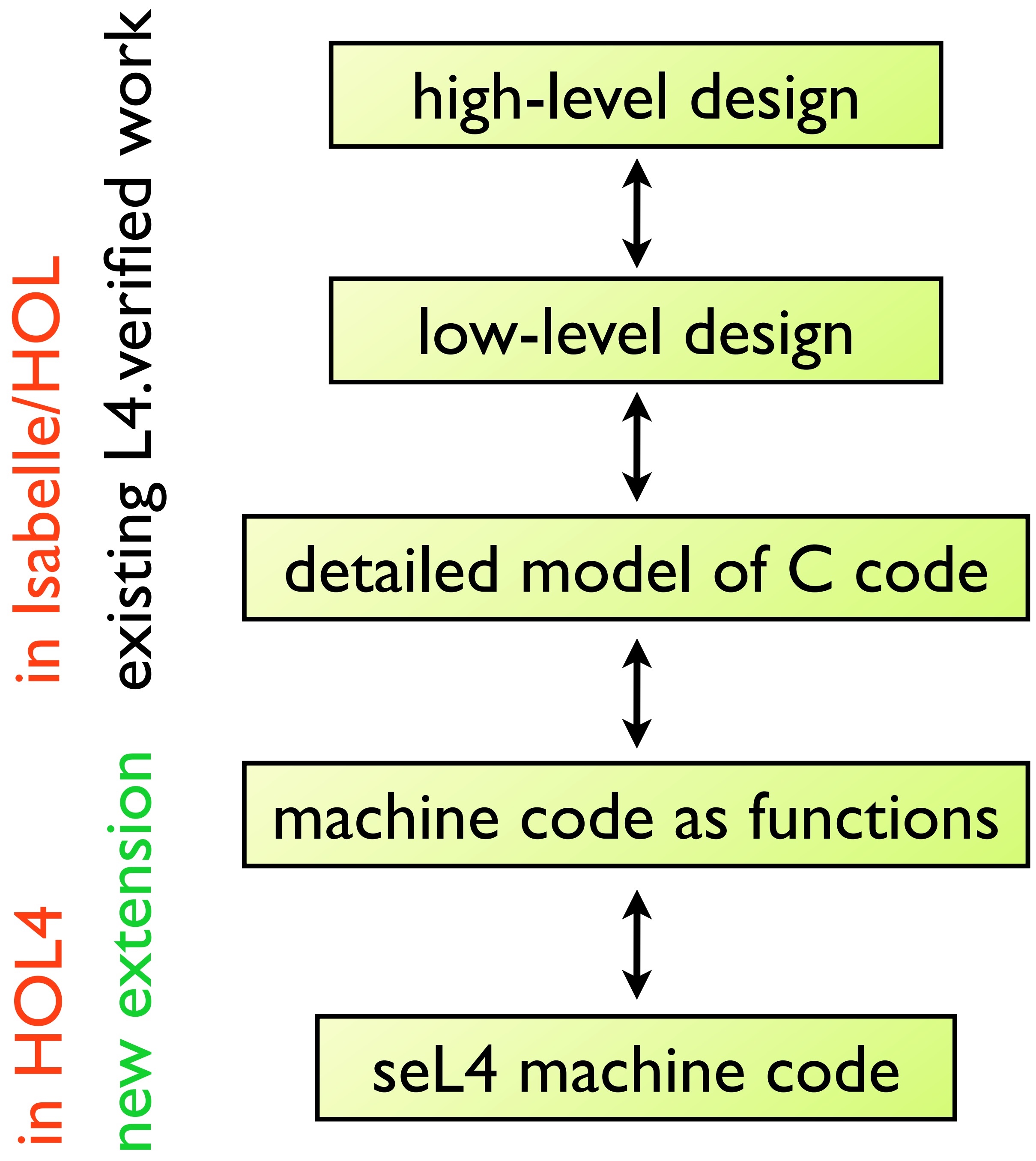
➡ impossible: current L4.verified proofs tied to C sem.

Connecting provers



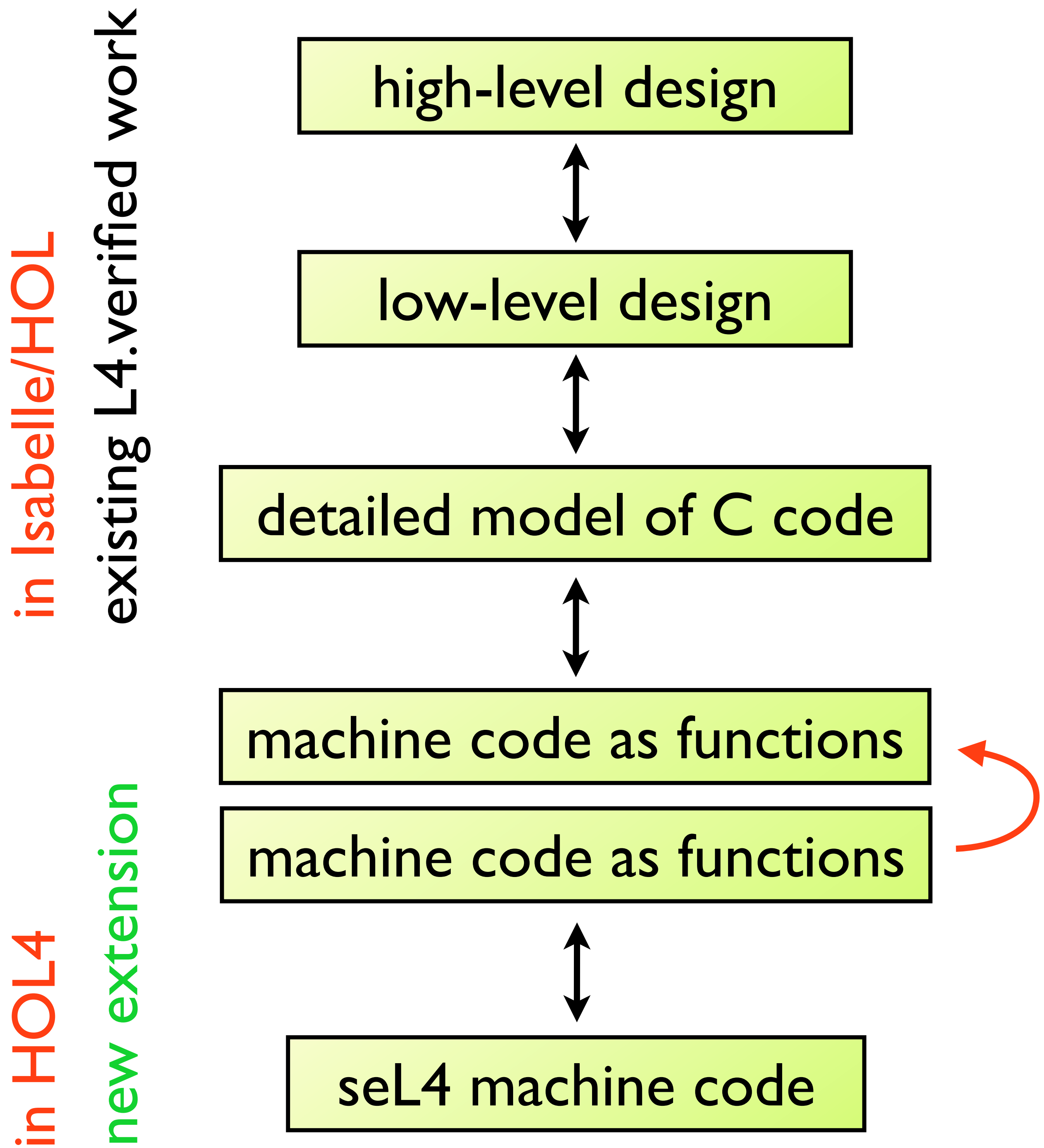
In general, hard.
Easy in this case.

Connecting provers



In general, hard.
Easy in this case.

Connecting provers



In general, hard.
Easy in this case.

automatic translation of definitions from HOL4 to Isabelle/HOL

Summary

L4.verified is being extended downwards using the Cambridge ARM model

* work in progress

Summary

L4.verified is being extended downwards
using the Cambridge ARM model

* work in progress

Aim:

- remove need to trust gcc and C

Summary

L4.verified is being extended downwards using the Cambridge ARM model

* work in progress

Aim:

- remove need to trust gcc and C

Lesson learnt:

- decompilation scales!
  (at least to 10,000 ARM instructions)

Summary

L4.verified is being extended downwards using the Cambridge ARM model

* work in progress

Questions?

Aim:

- remove need to trust gcc and C

Lesson learnt:

- decompilation scales!
  (at least to 10,000 ARM instructions)