To appear in PLDI'13

Translation Validation for a Verified OS Kernel

Thomas Sewell[1], Magnus Myreen[2], 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 in L4.verified

L4.verified project assumes correctness of:

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

Assumptions in L4.verified

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 in L4.verified

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 in L4.verified

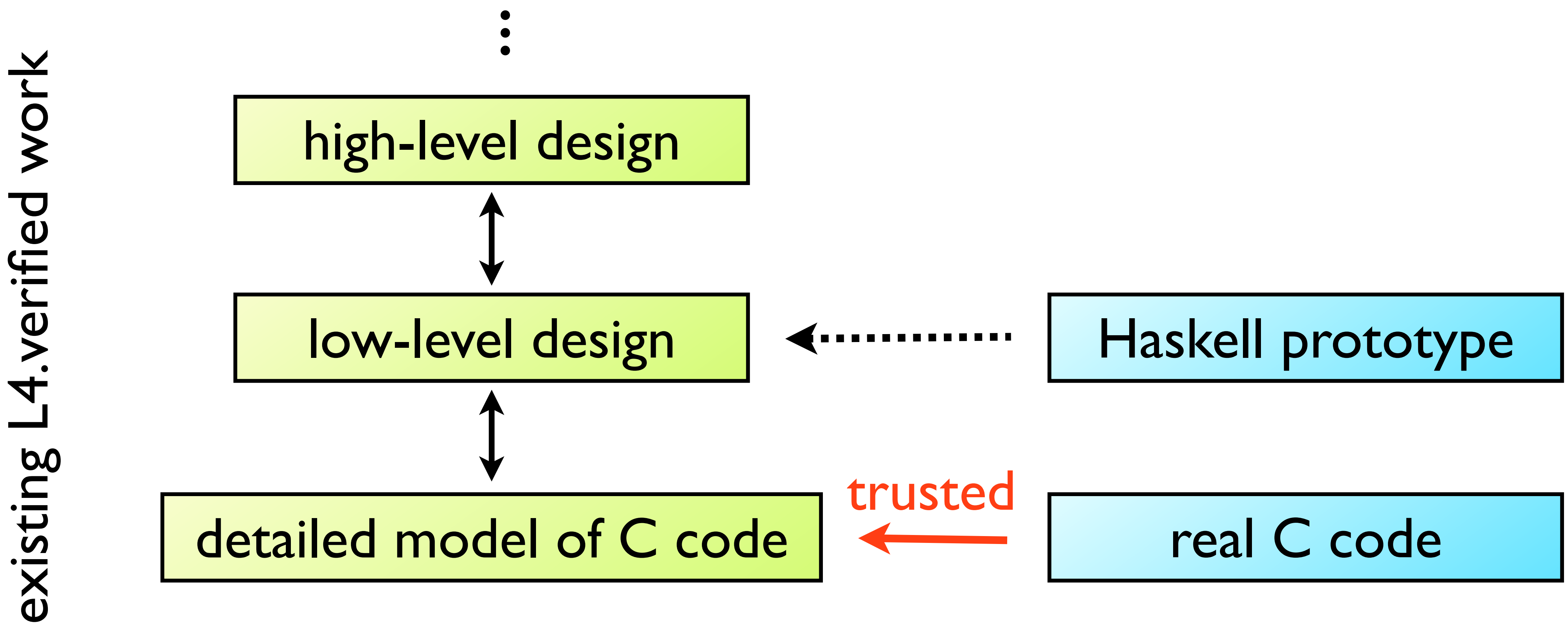
L4.verified project assumes correctness of:

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

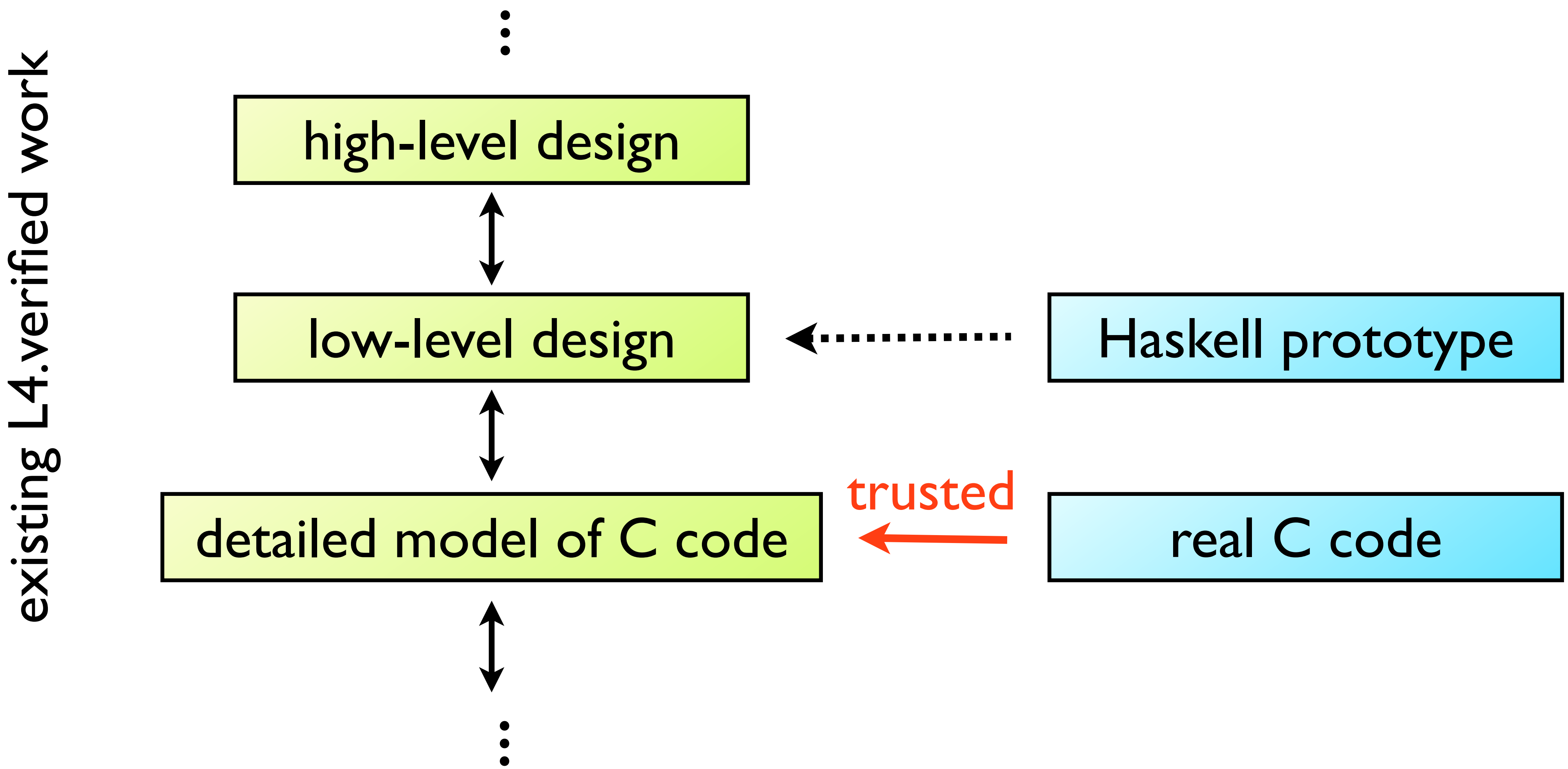
The aim of this work is to remove the first assumption.

And also to validate L4.verified's C semantics.

Aim: extend downwards

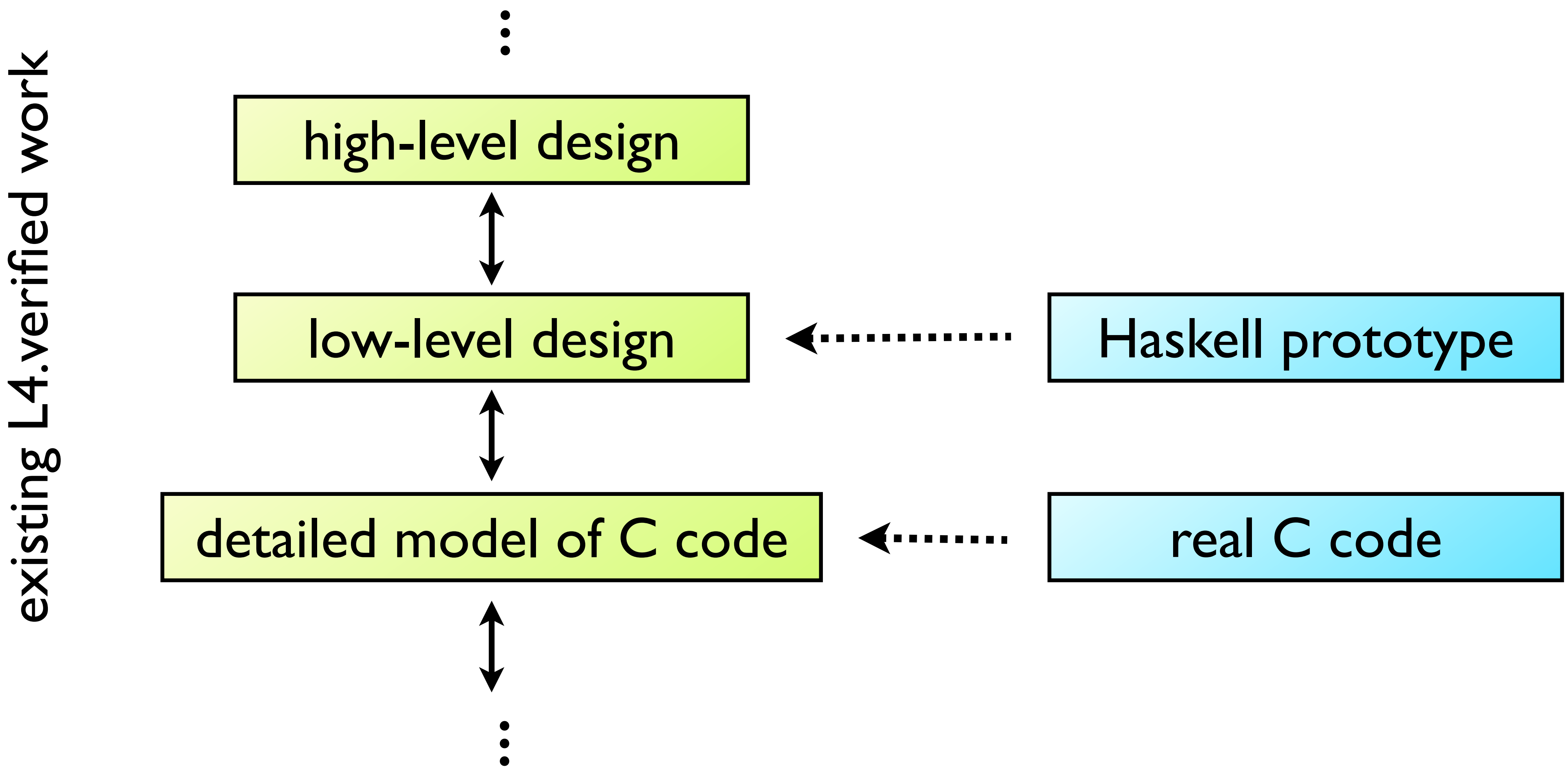


Aim: extend downwards



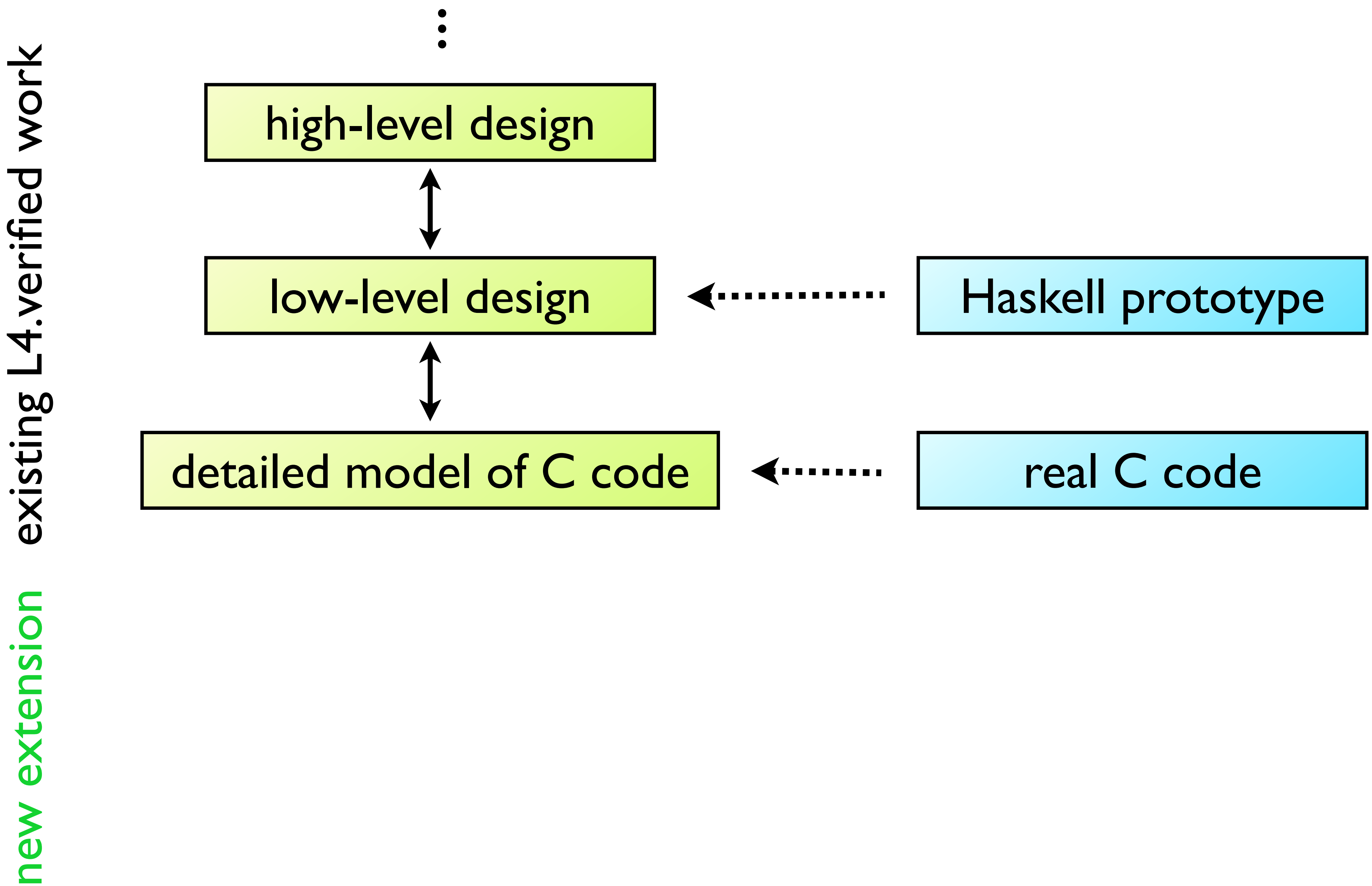
Aim: remove need to trust C compiler and C semantics

Aim: extend downwards

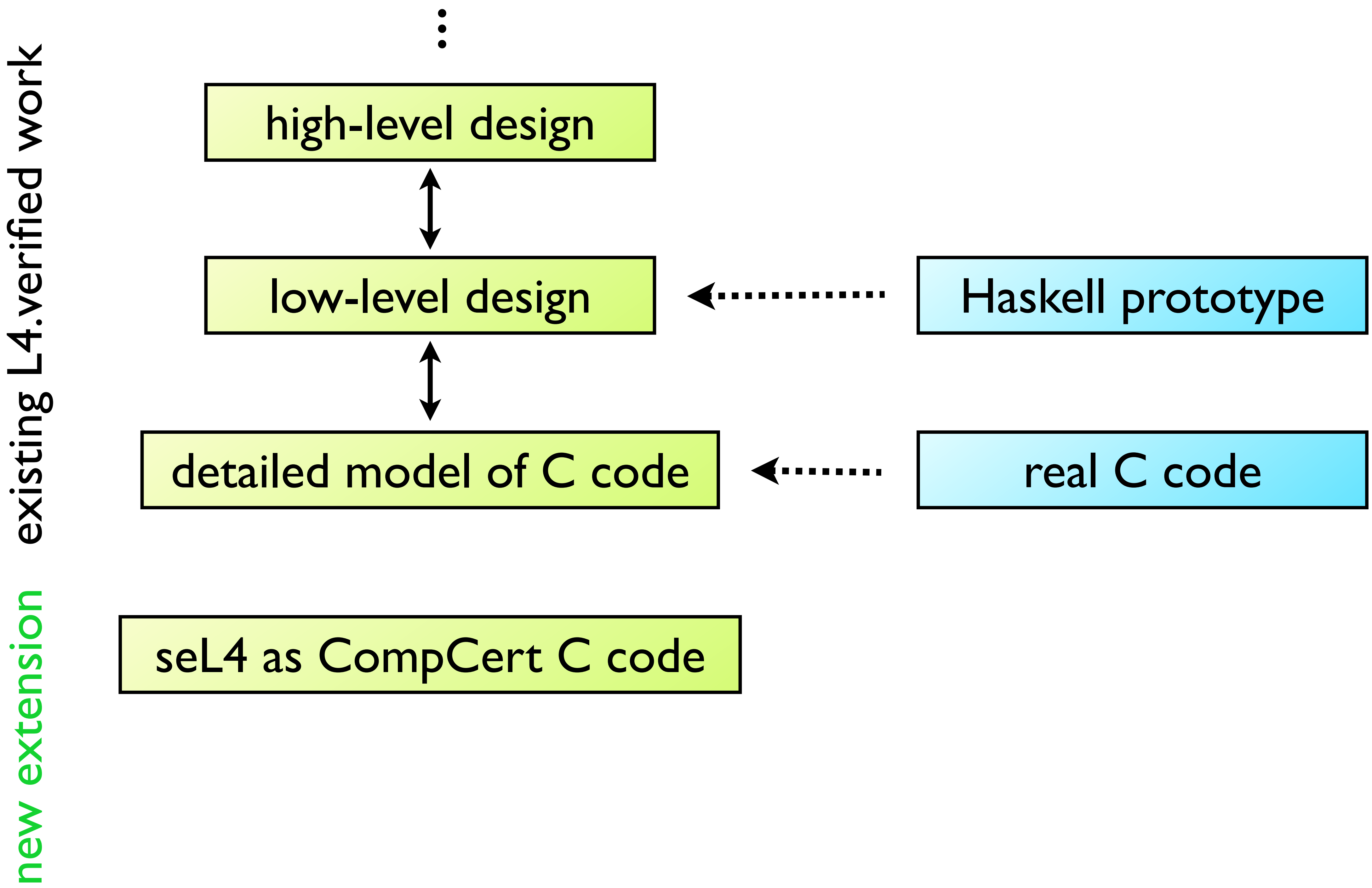


Aim: remove need to trust C compiler and C semantics

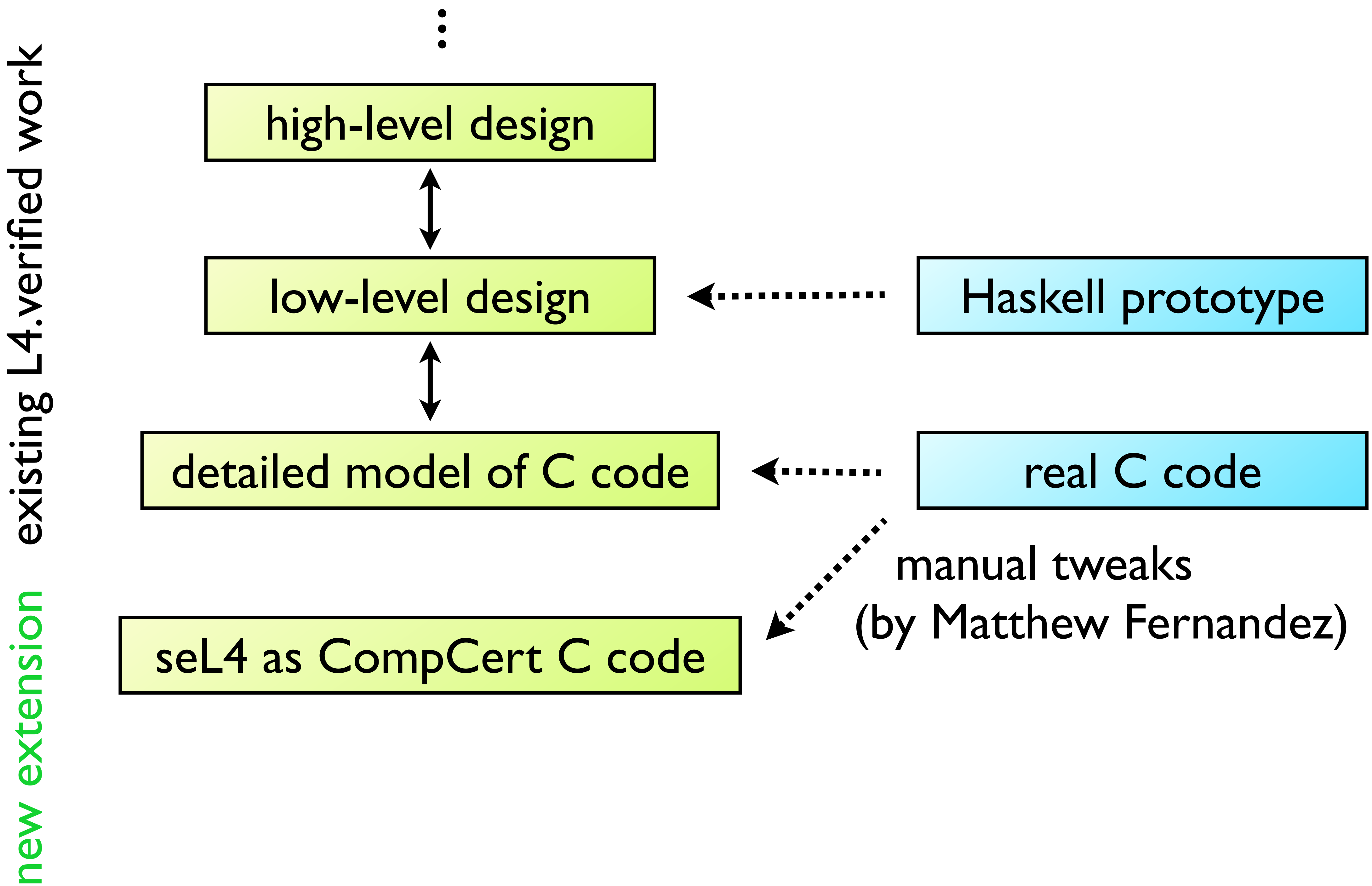
Connection to CompCert



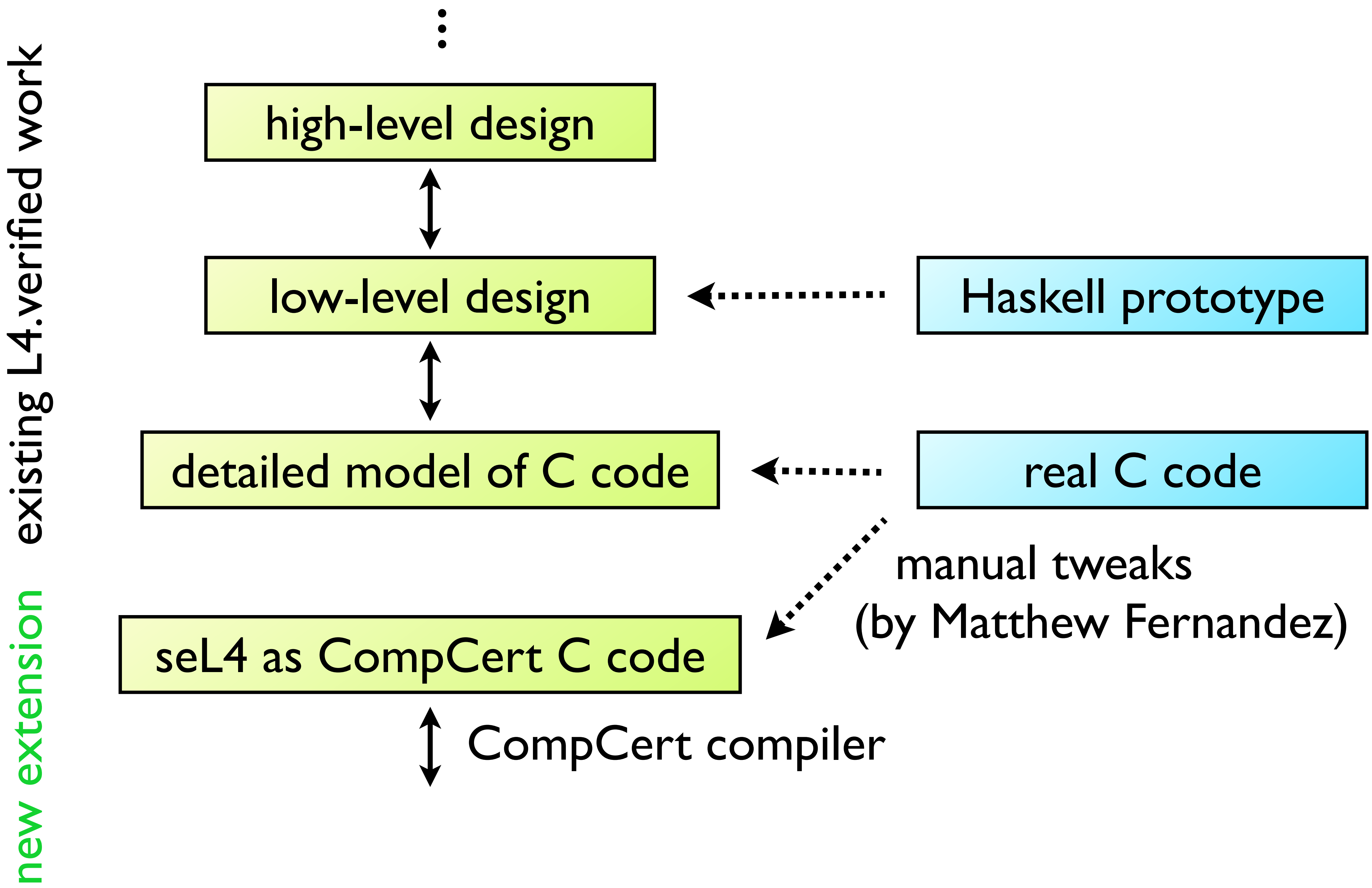
Connection to CompCert



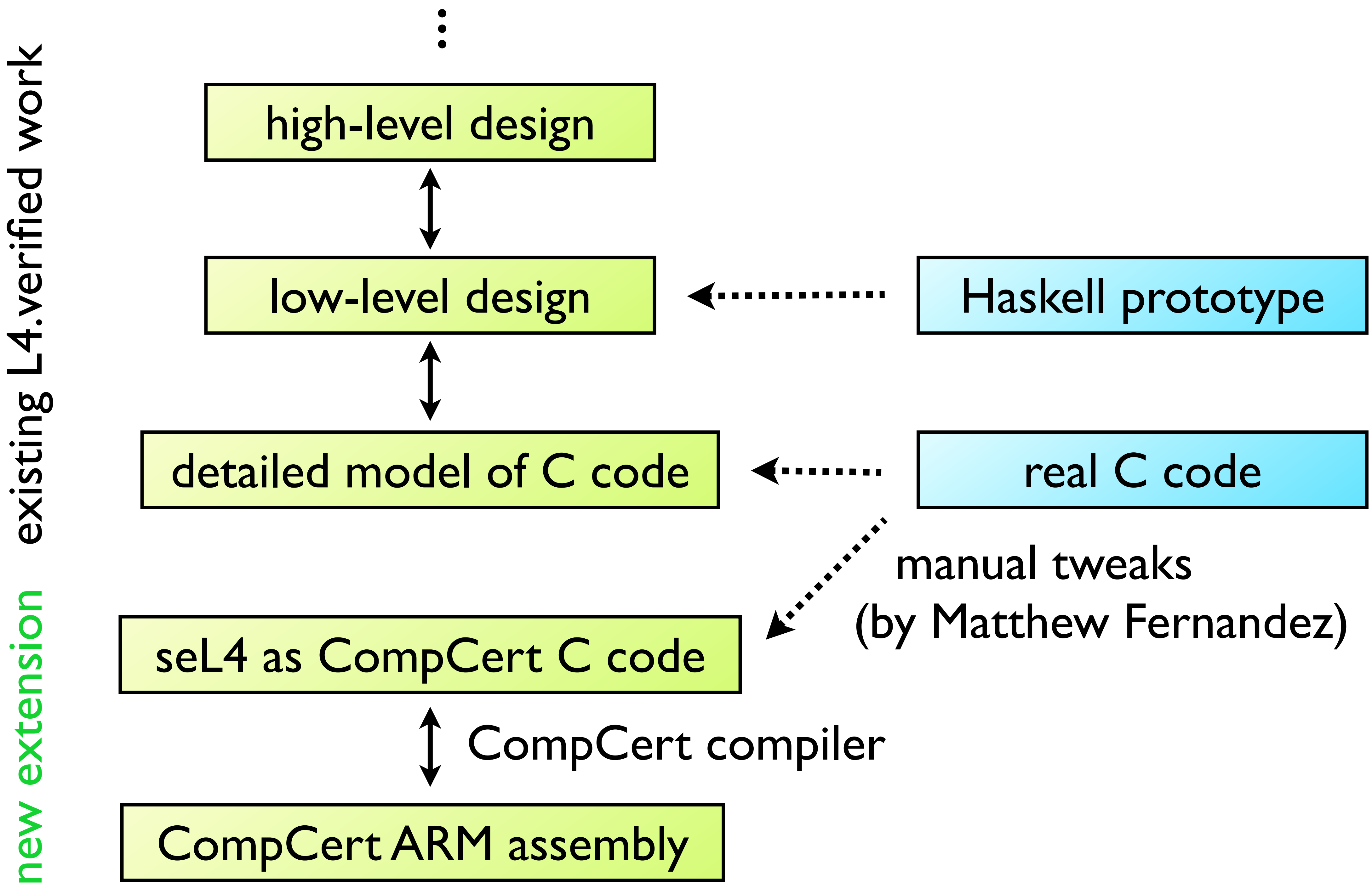
Connection to CompCert



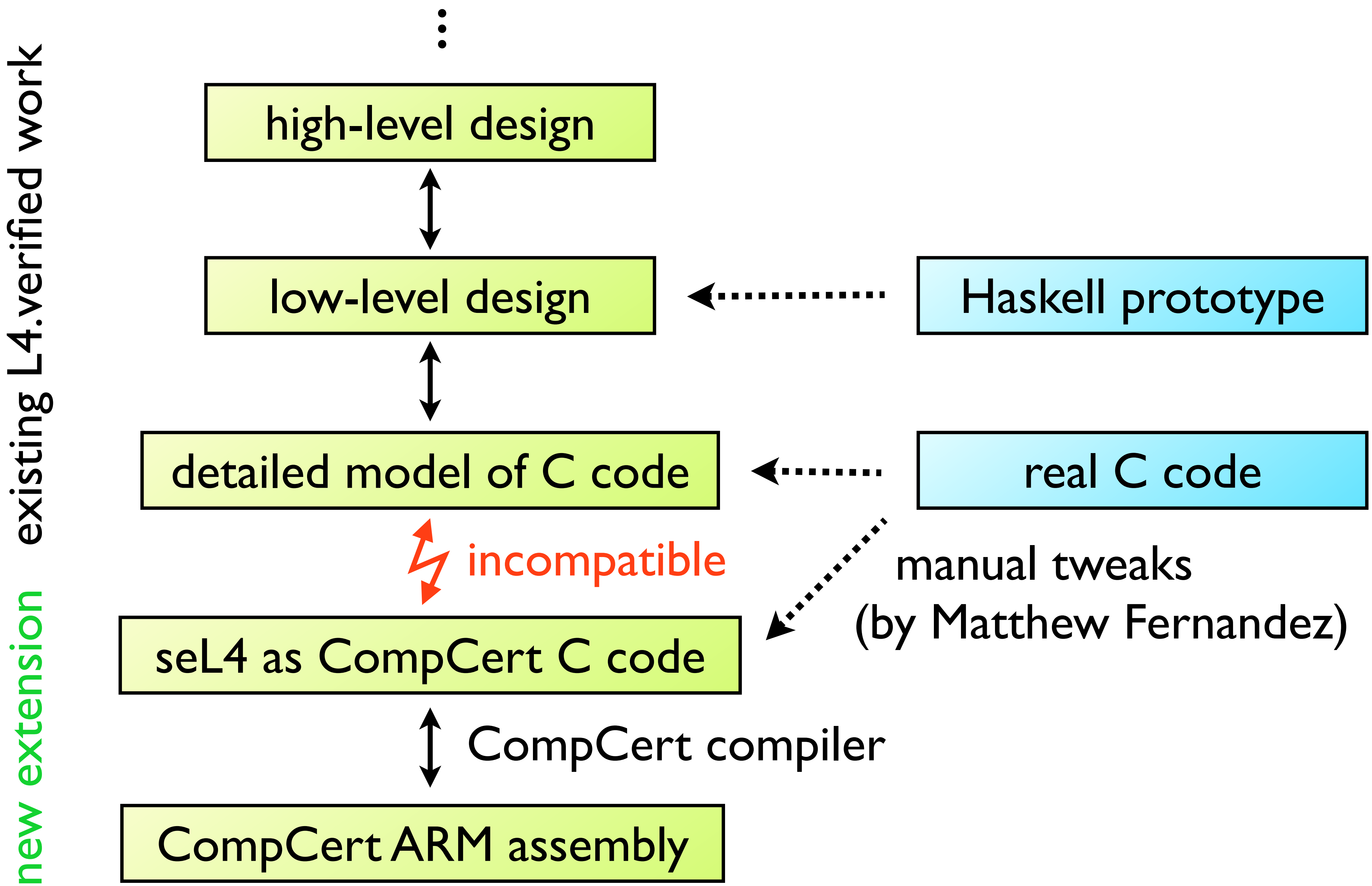
Connection to CompCert



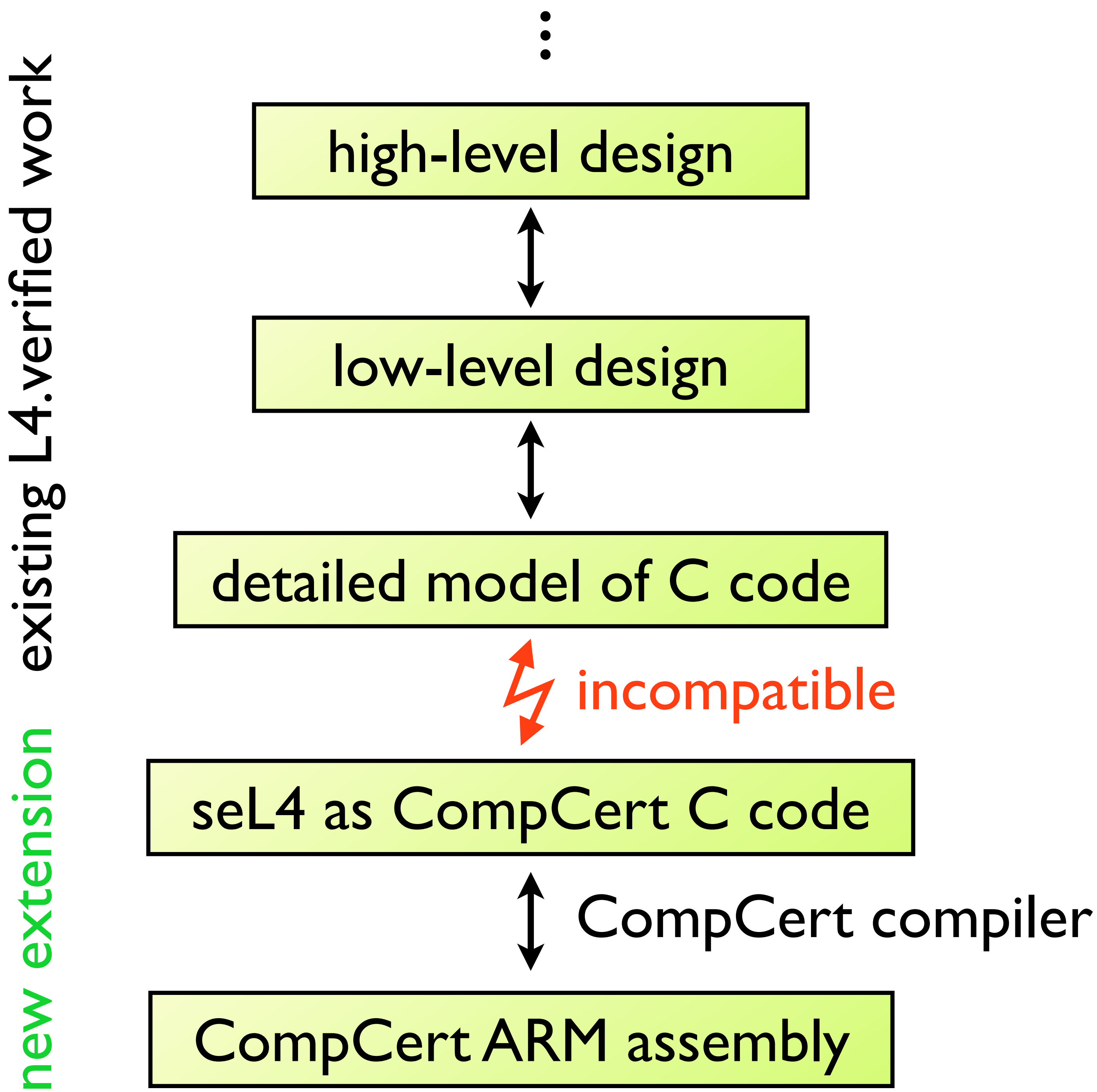
Connection to CompCert



Connection to CompCert



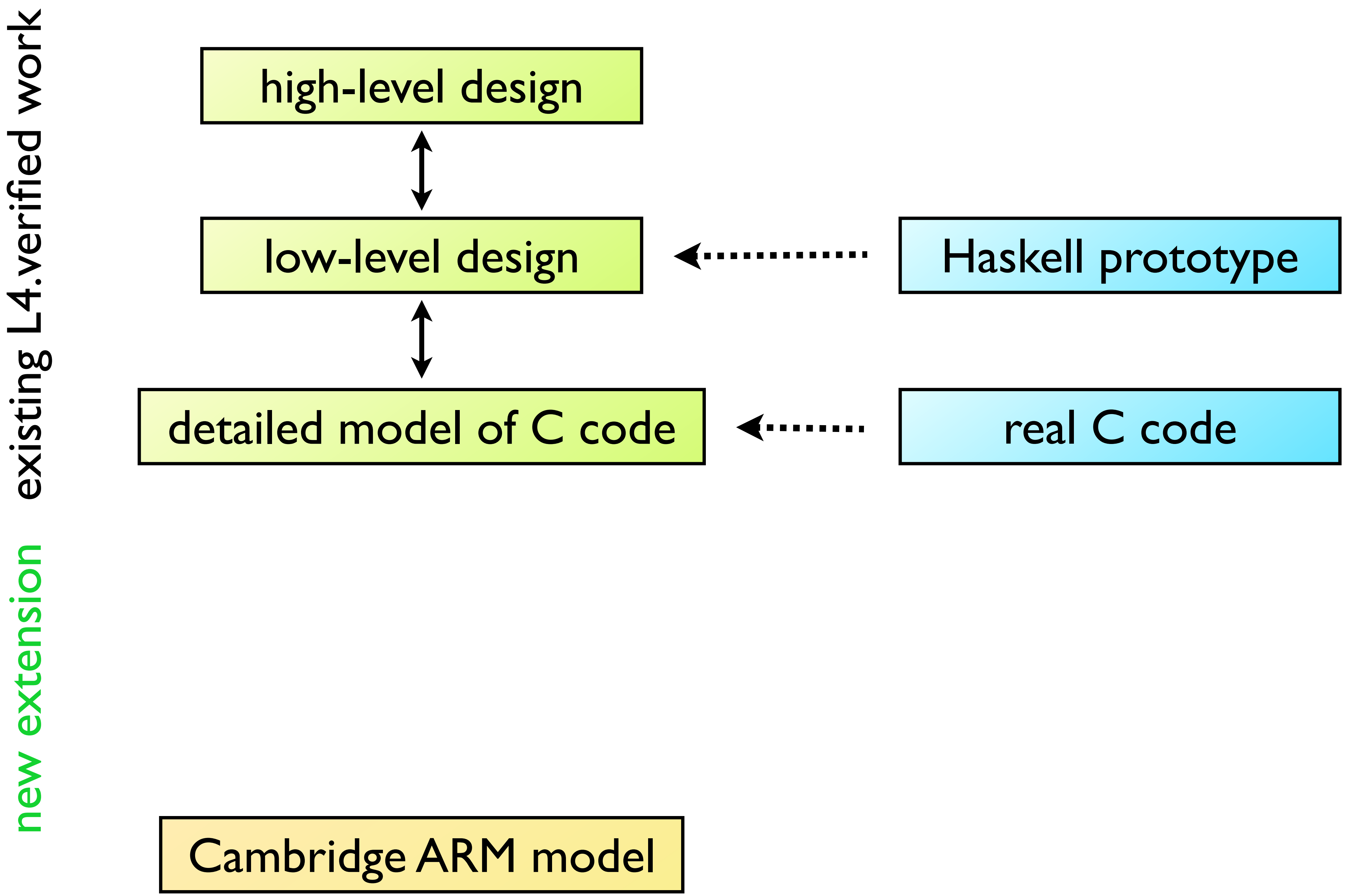
Connection to CompCert



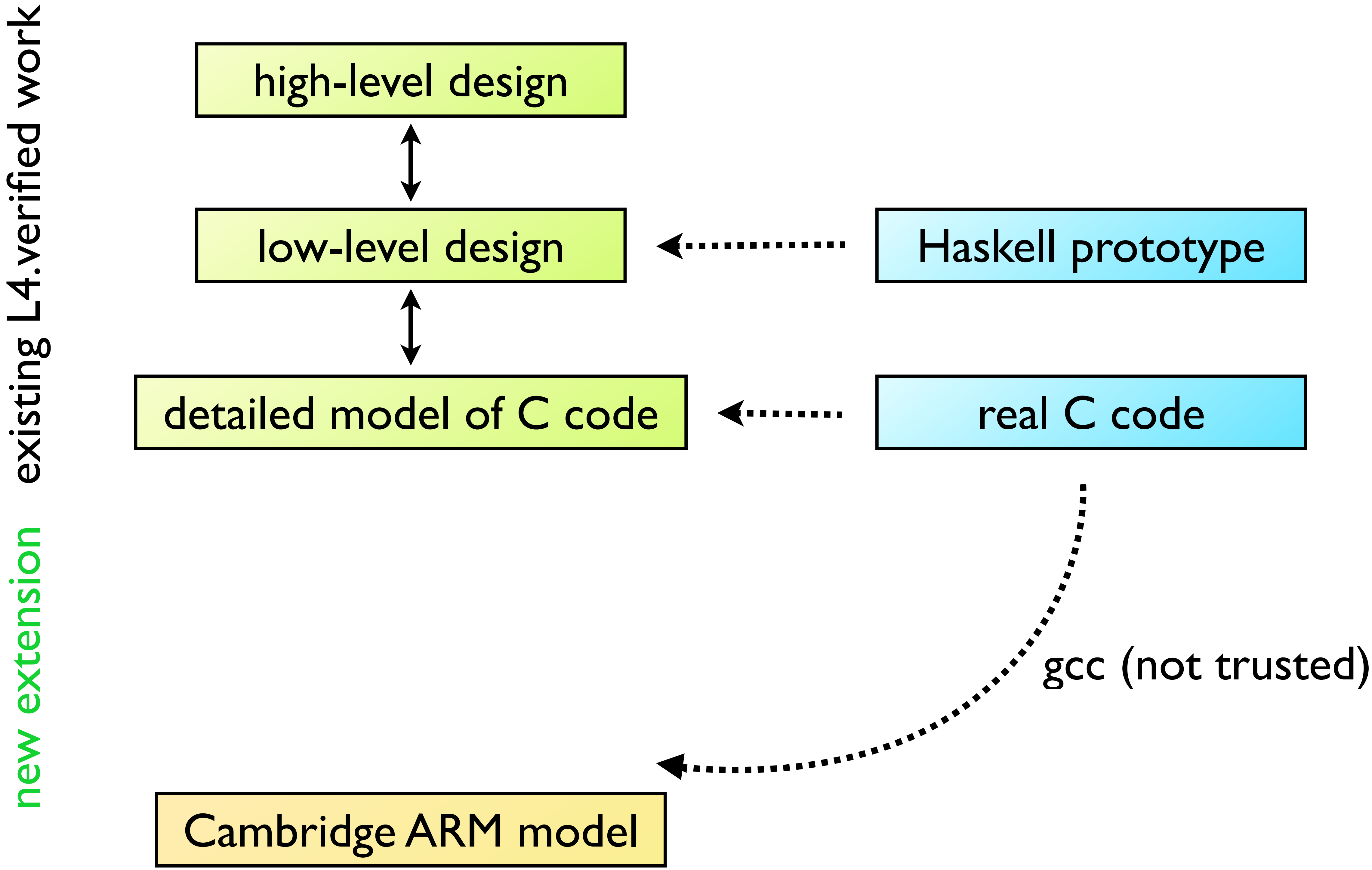
Incompatible:

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

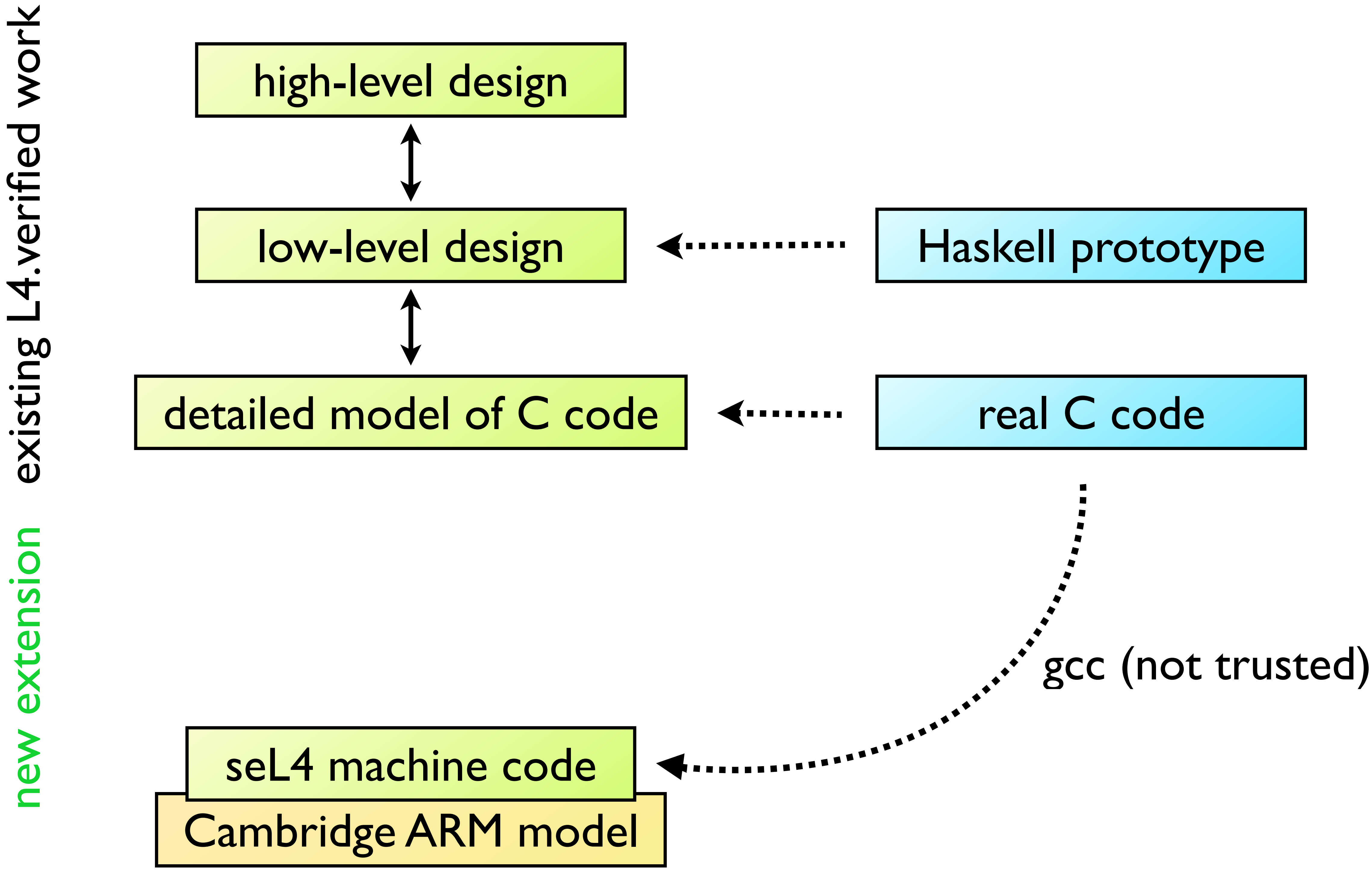
Using Cambridge ARM model



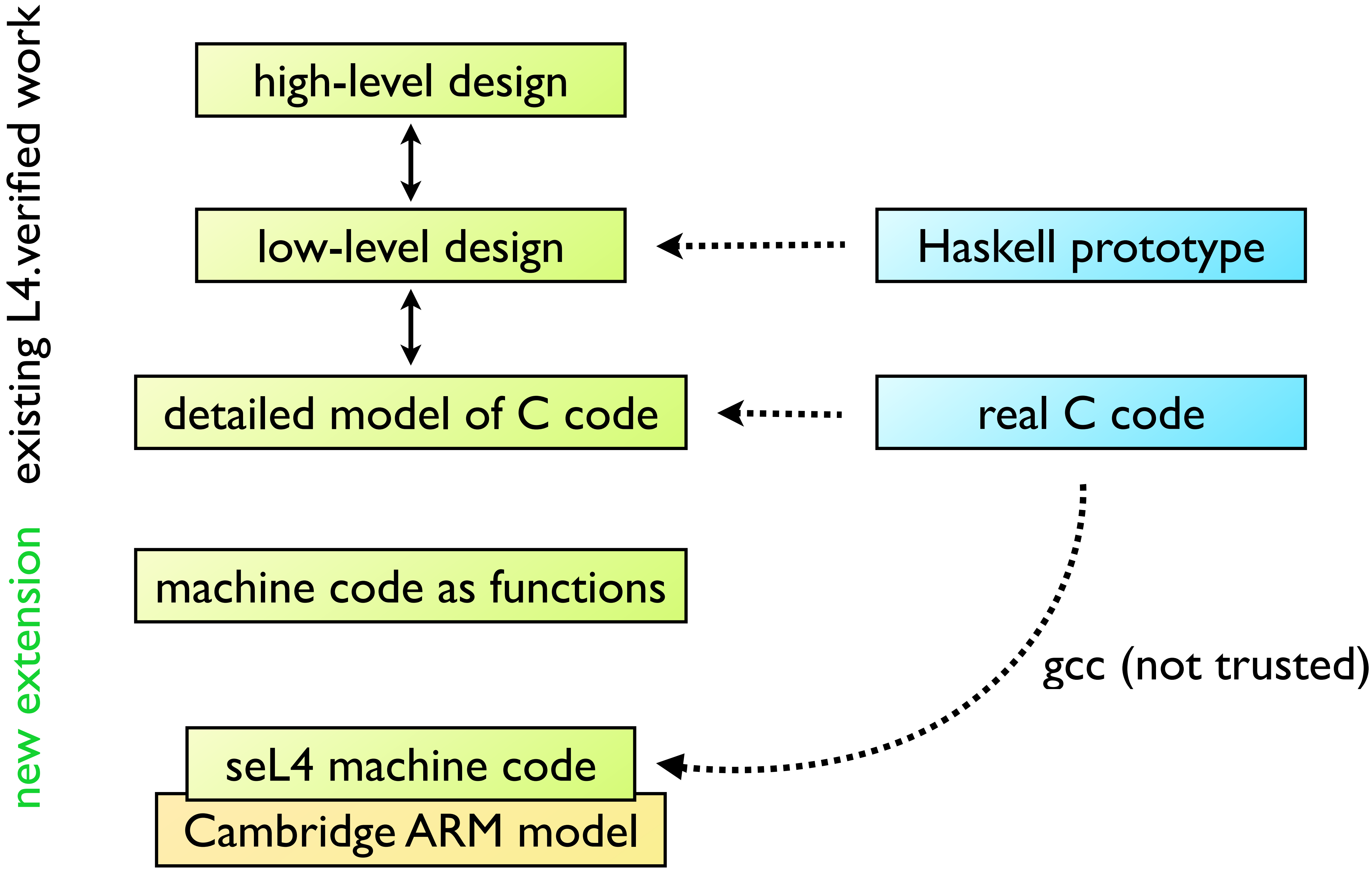
Using Cambridge ARM model



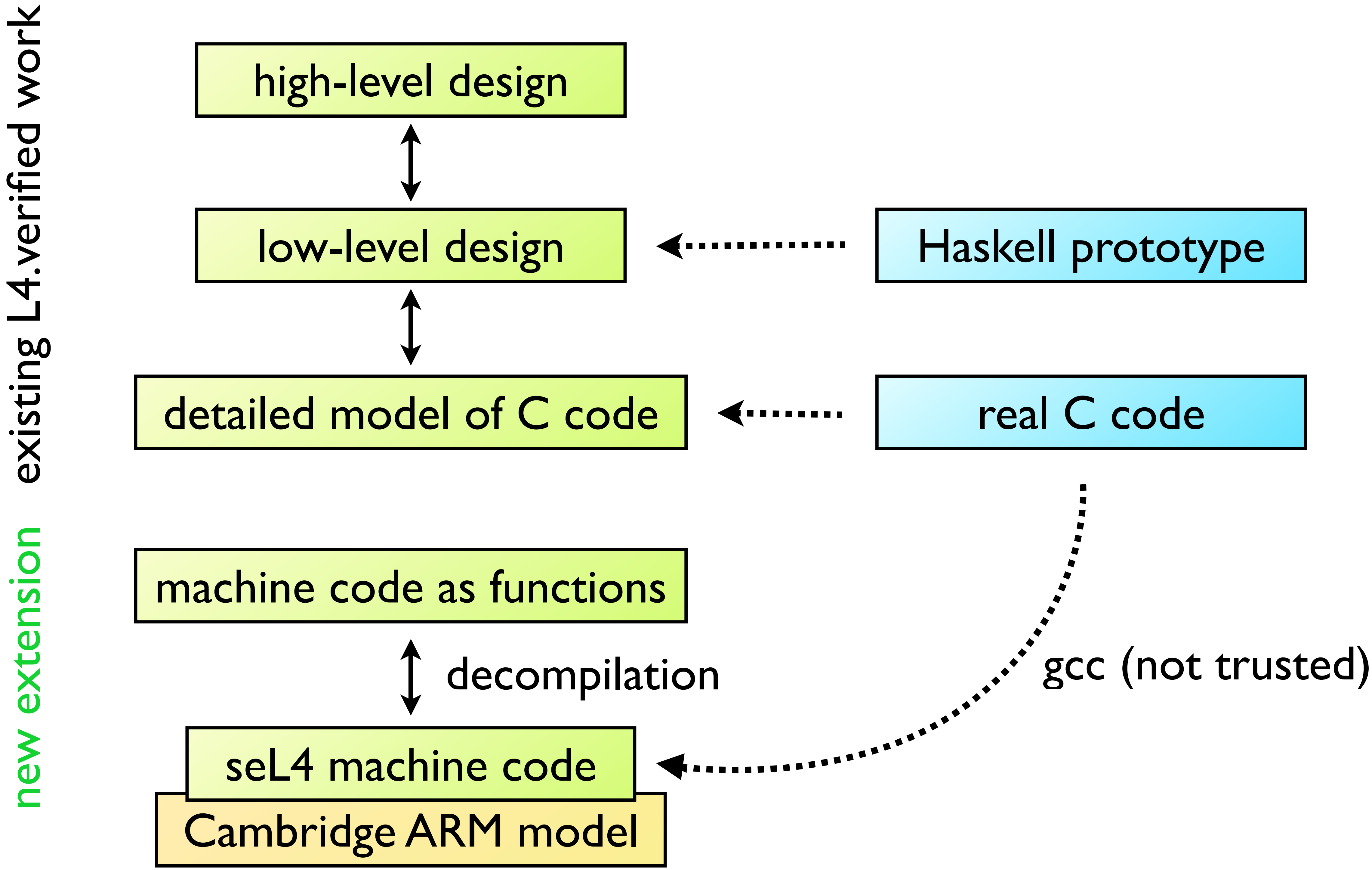
Using Cambridge ARM model



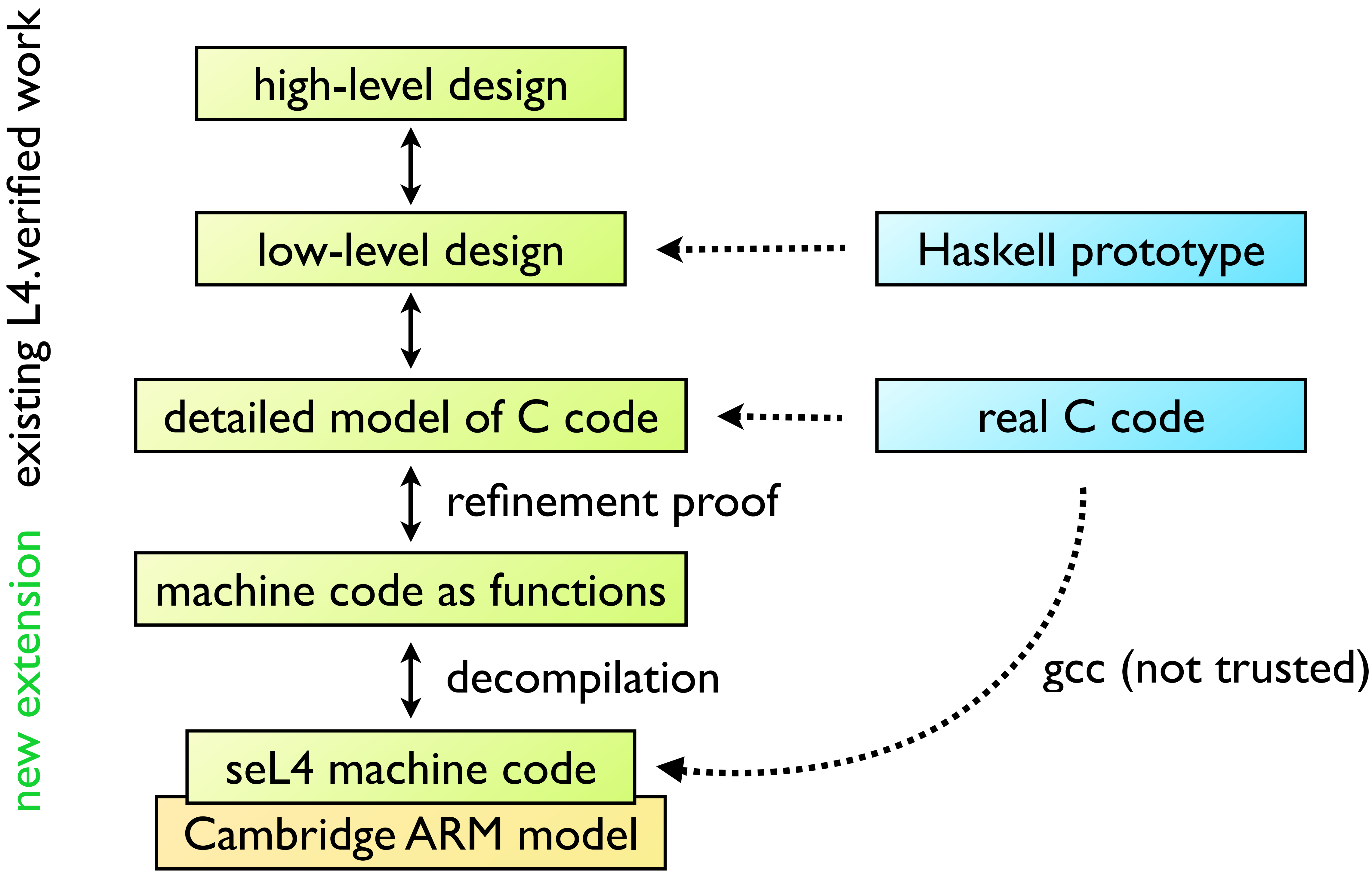
Using Cambridge ARM model



Using Cambridge ARM model



Using Cambridge ARM model

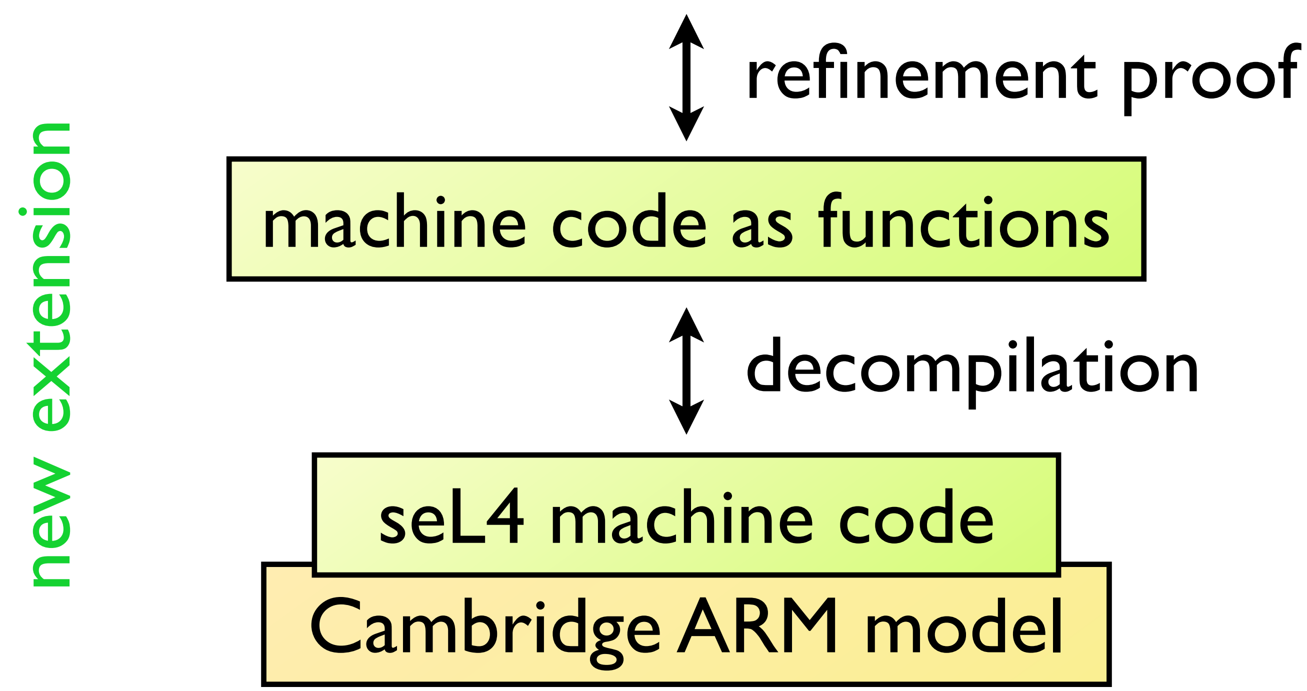


Translation validation

Translation Validation efforts:

- Pnueli et al, 1998. Introduce translation validation. Want to maintain a compiler correctness proof more easily.
- Necula, 2000. Translation validation for a C compiler. Also wants to pragmatically support compiler quality.
- Many others for many languages and levels of connection to compilers.
- …
- Sewell & Myreen, 2013. Not especially interested in compilers. Want to validate a source semantics.

Talk outline



Part 1: automatic translation / decompilation

Part 2: pseudo compilation and refinement proof (SMT)

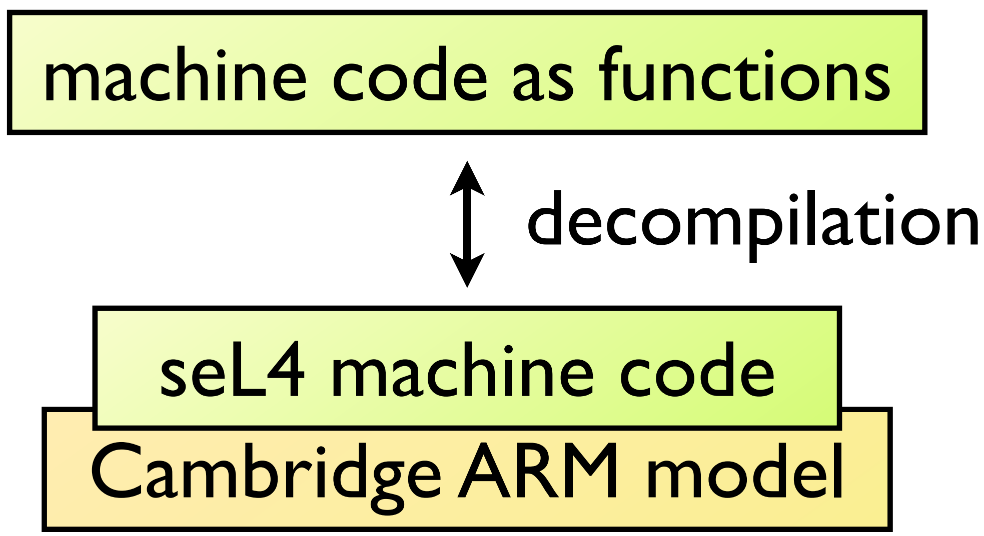
Cambridge ARM model

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

```
e0810000

e1a000a0

e12fff1e
```

How to decompile:

```
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 -O1 and 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 -O1 and 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 -O1 and 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 -O1 and gcc -O2
  - ✓ gcc implements ARM/C calling convention
- must be compatible with L4.verified proof
  - ➡ stack requires special treatment

Stack visible in machine 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 machine 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 machine 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 machine 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

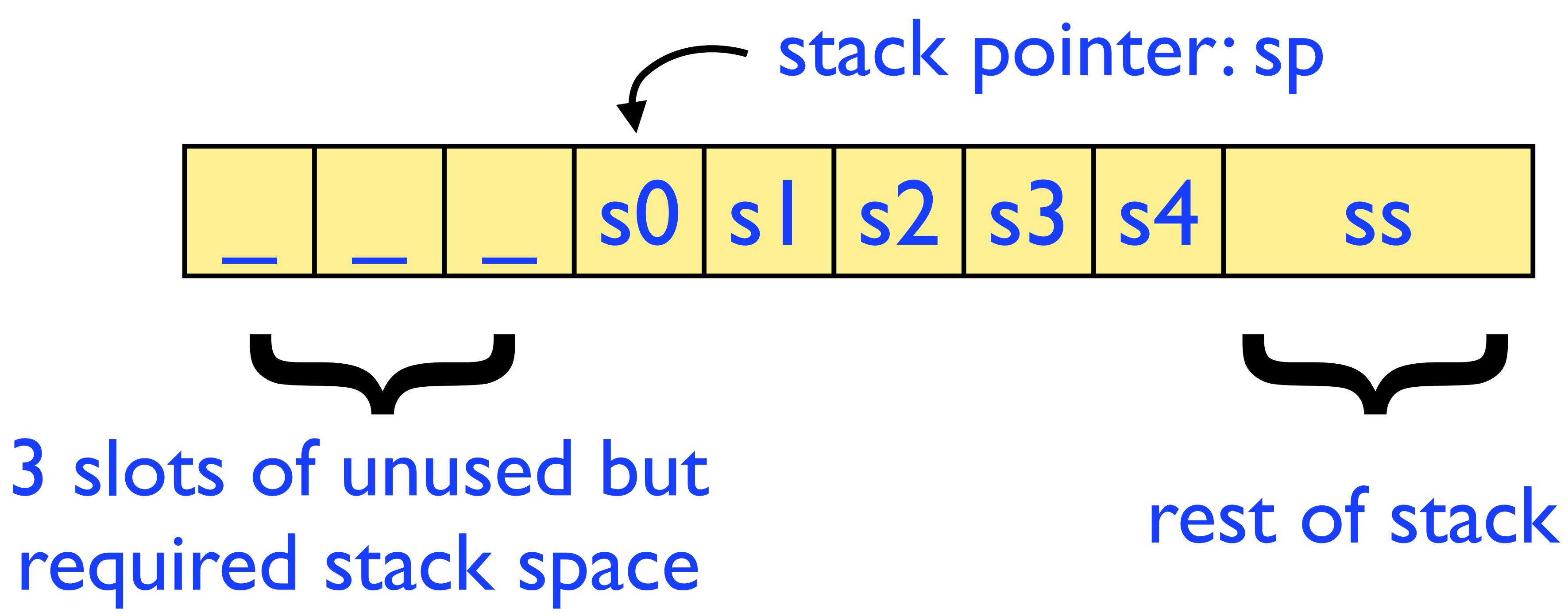
```
add  r1, r1, r0
add  r1, r1, r2
ldr  r2, [sp]
add  r1, r1, r3
add  r0, r1, r2
ldmib sp, {r2, r3}
add  r0, r0, r2
add  r0, r0, r3
ldr  r3, [sp, #12]
add  r0, r0, r3
lsr  r0, r0, #3
bx   lr
```

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

... that are not present in C semantics.

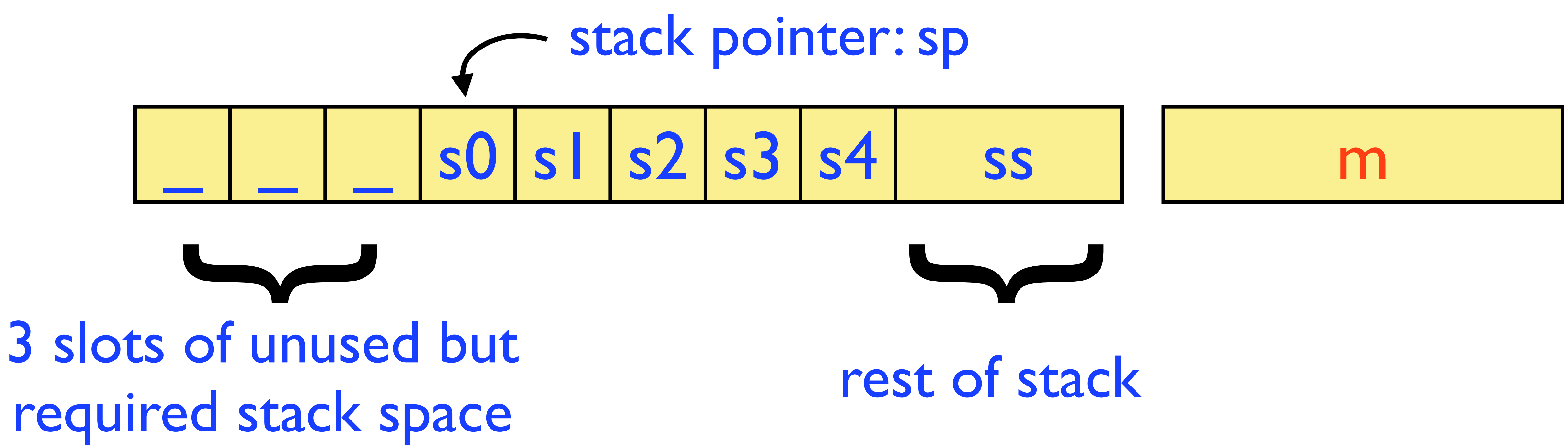
Solution

Use separation-logic inspired approach



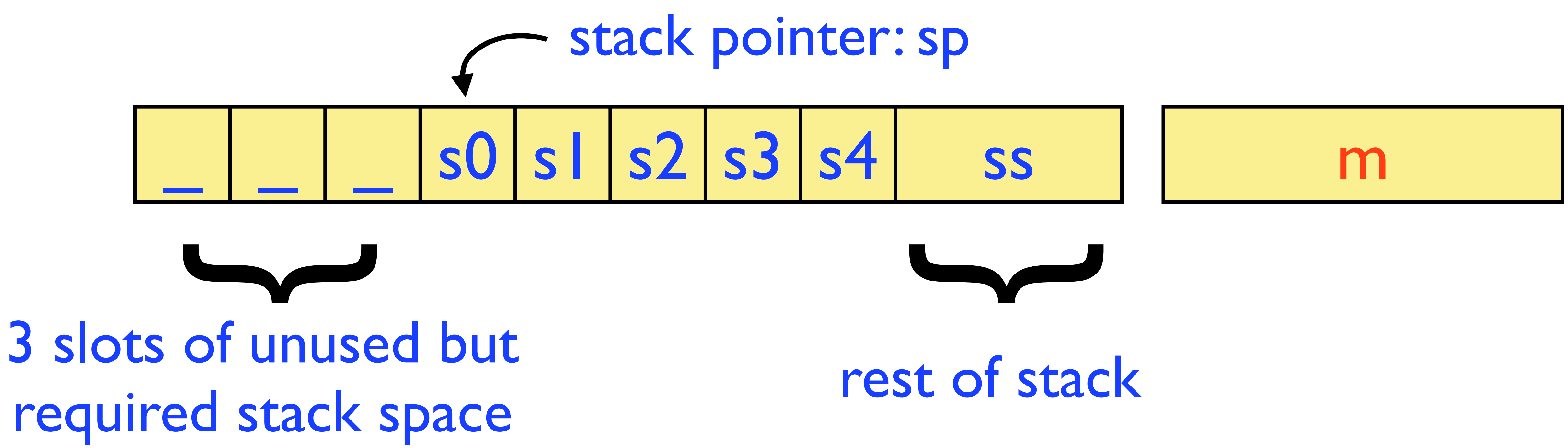
Solution

Use separation-logic inspired approach



Solution

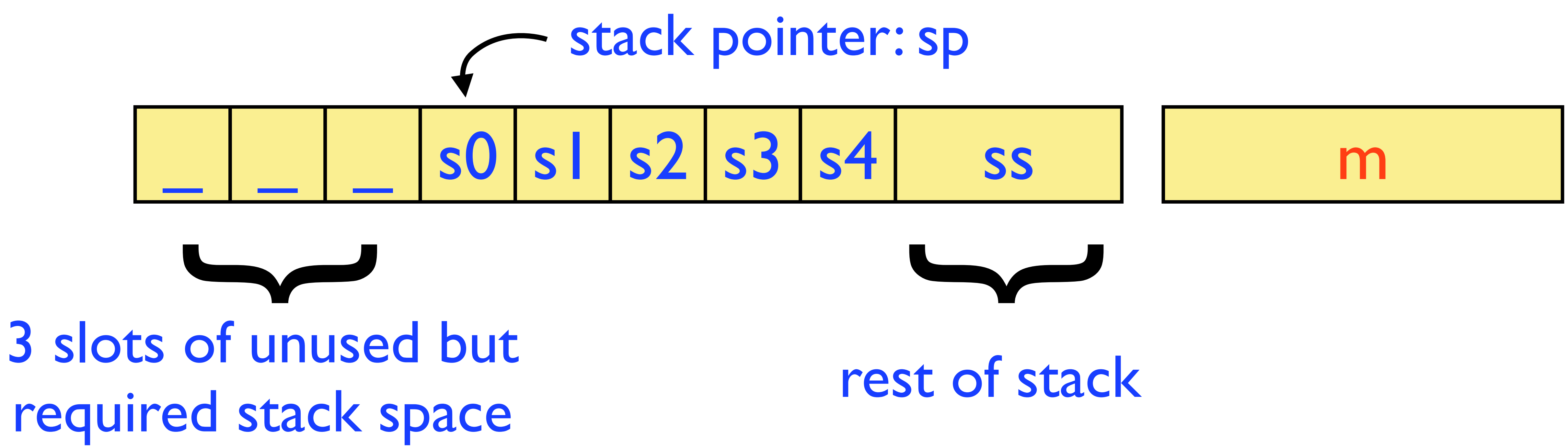
Use separation-logic inspired approach



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

Solution

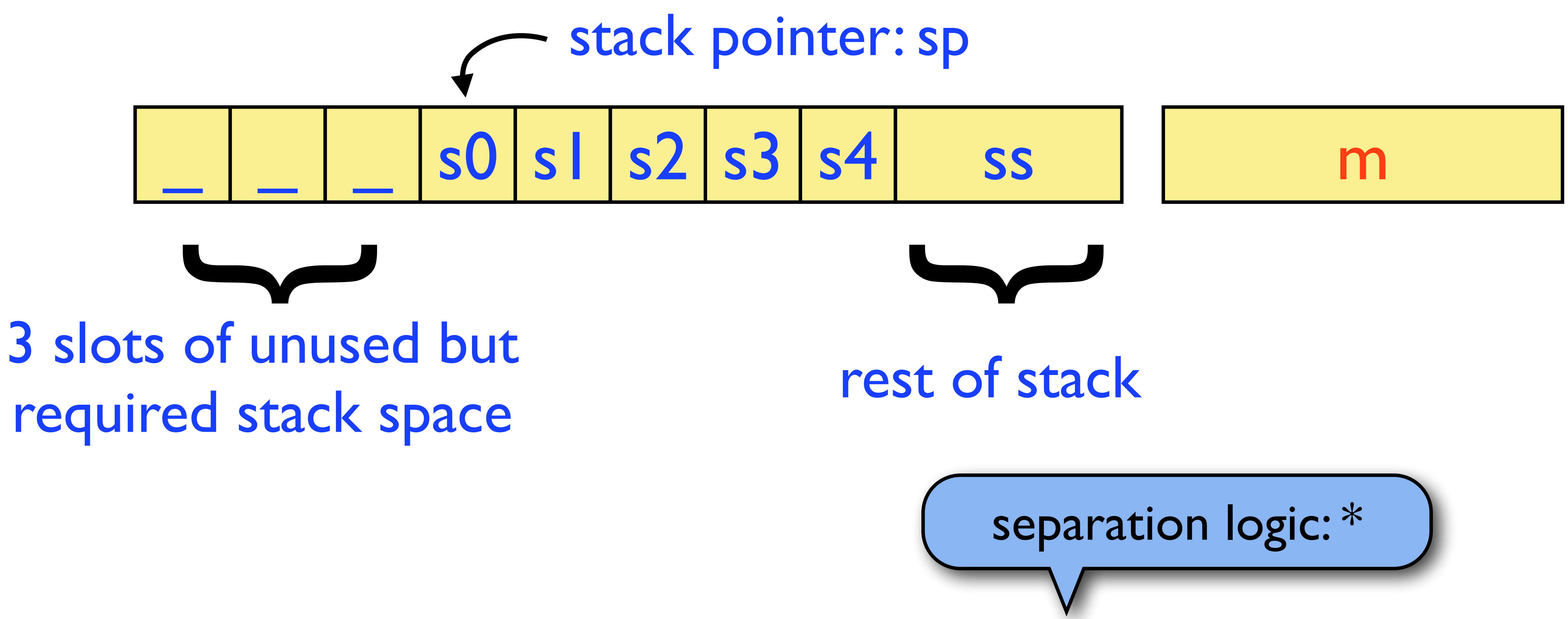
Use separation-logic inspired approach



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

Solution

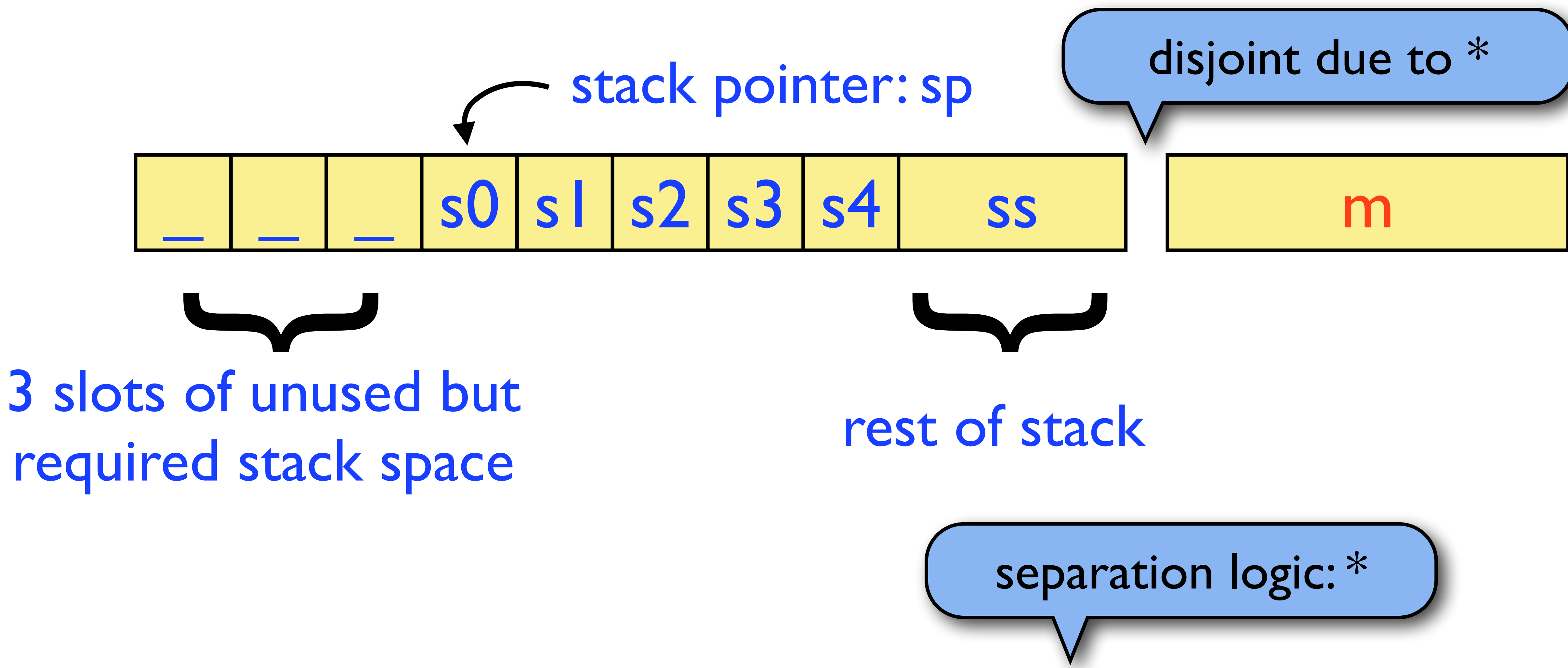
Use separation-logic inspired approach



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.

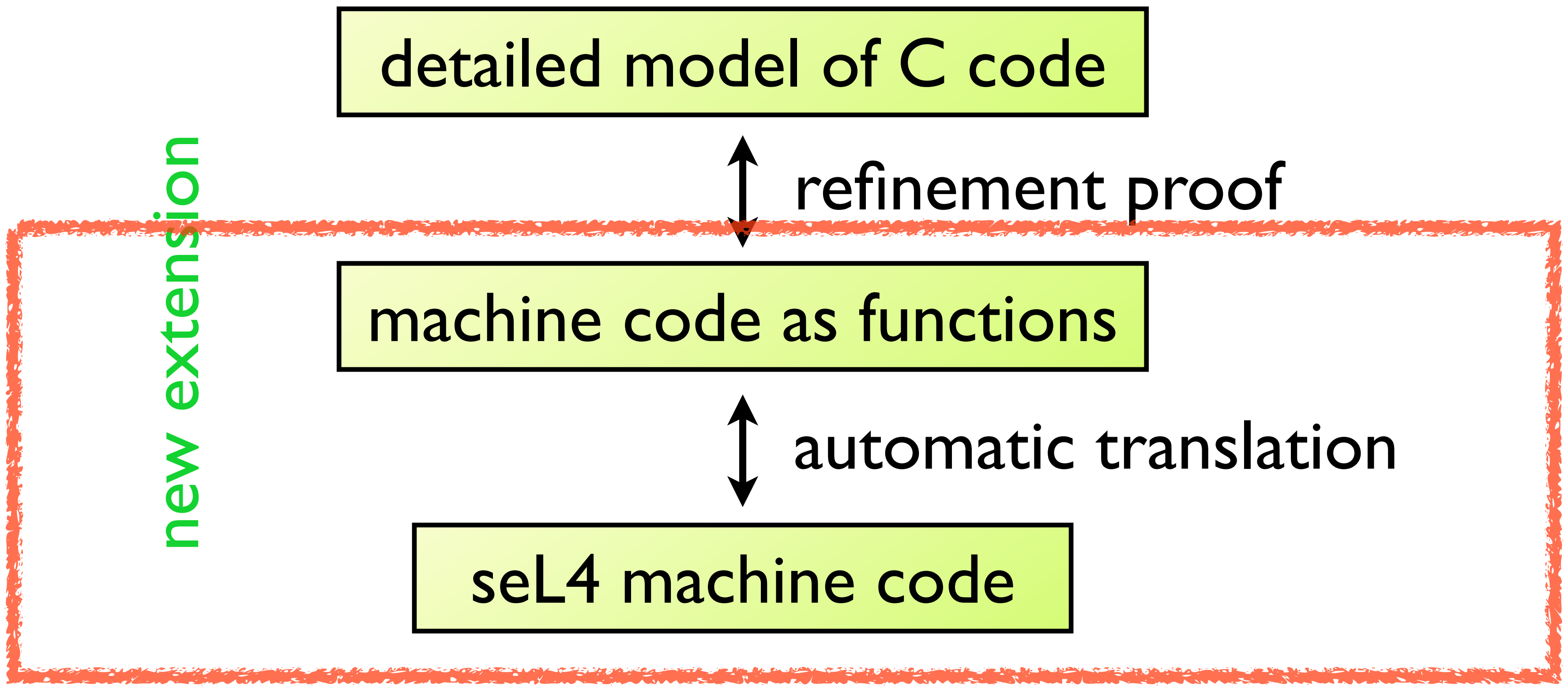
```
add r1, r1, r0
add r1, r1, r2
ldr  r2, [sp]
add r1, r1, r3
add r0, r1, r2
ldmib sp, {r2, r3}
add r0, r0, r2
add r0, r0, r3
ldr  r3, [sp, #12]
add r0, r0, r3
lsr  r0, r0, #3
bx   lr
```

```
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
```

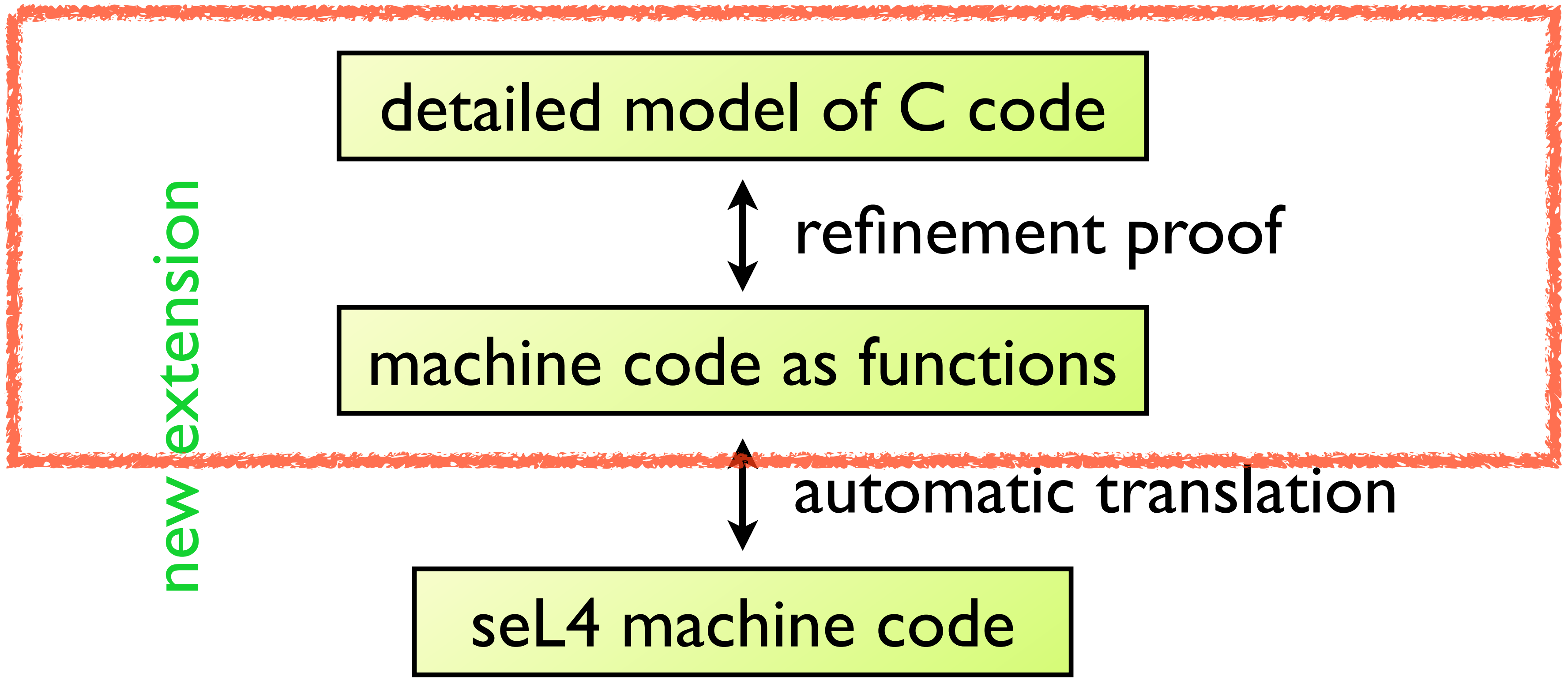
Other C-specifics

- 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

Moving on to stage 2



Moving on to stage 2

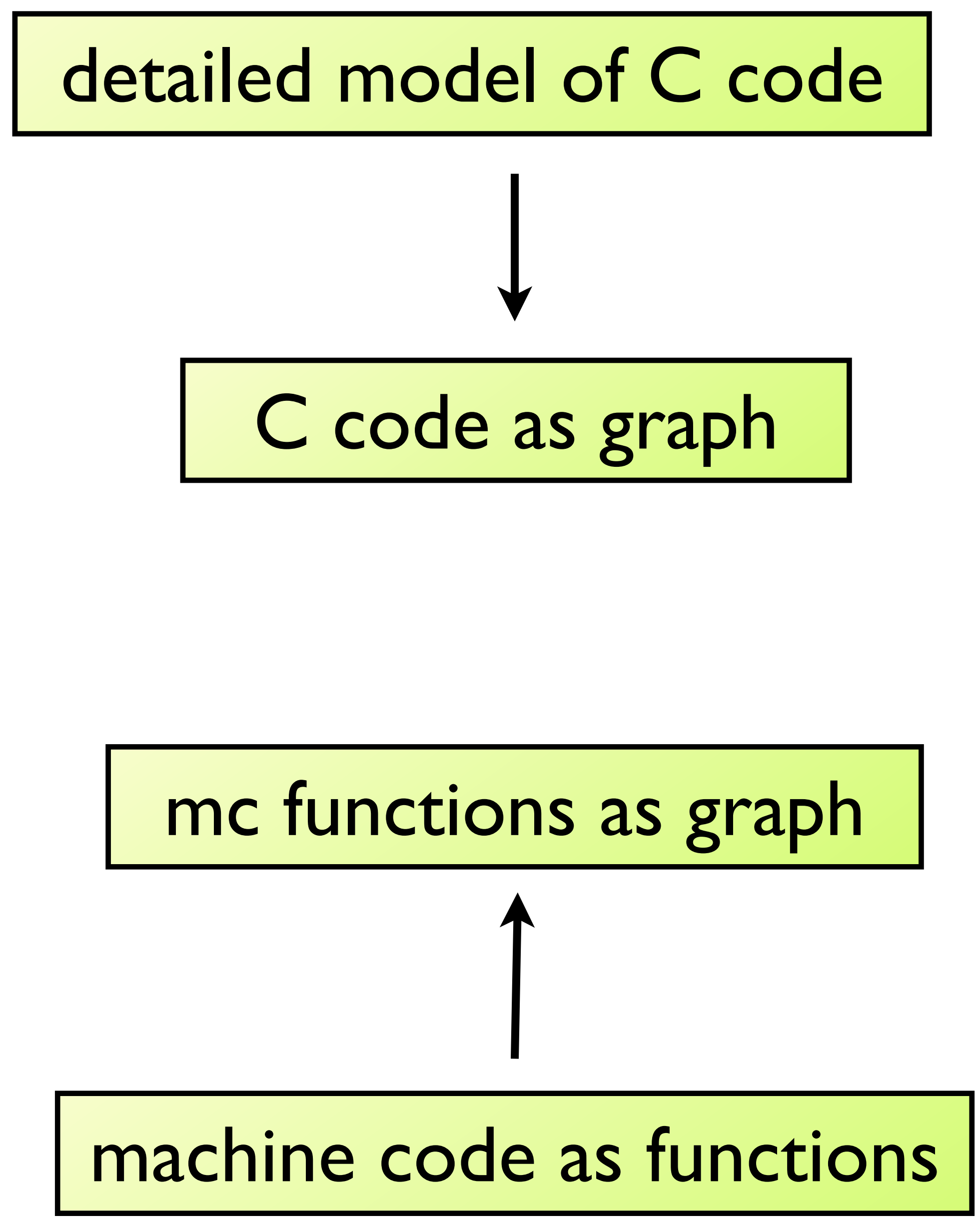


Approach for refinement proof

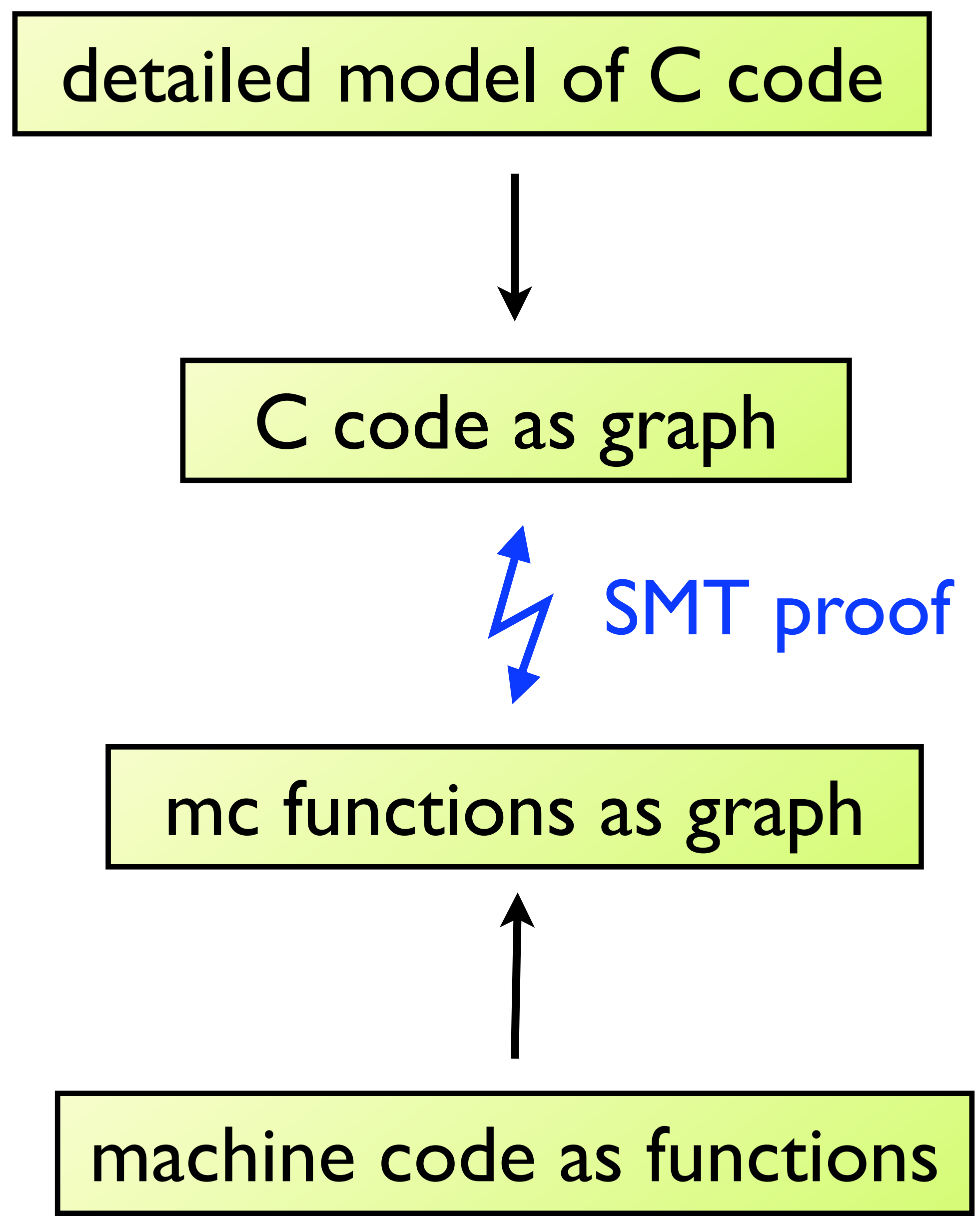
detailed model of C code

machine code as functions

Approach for refinement proof



Approach for refinement proof

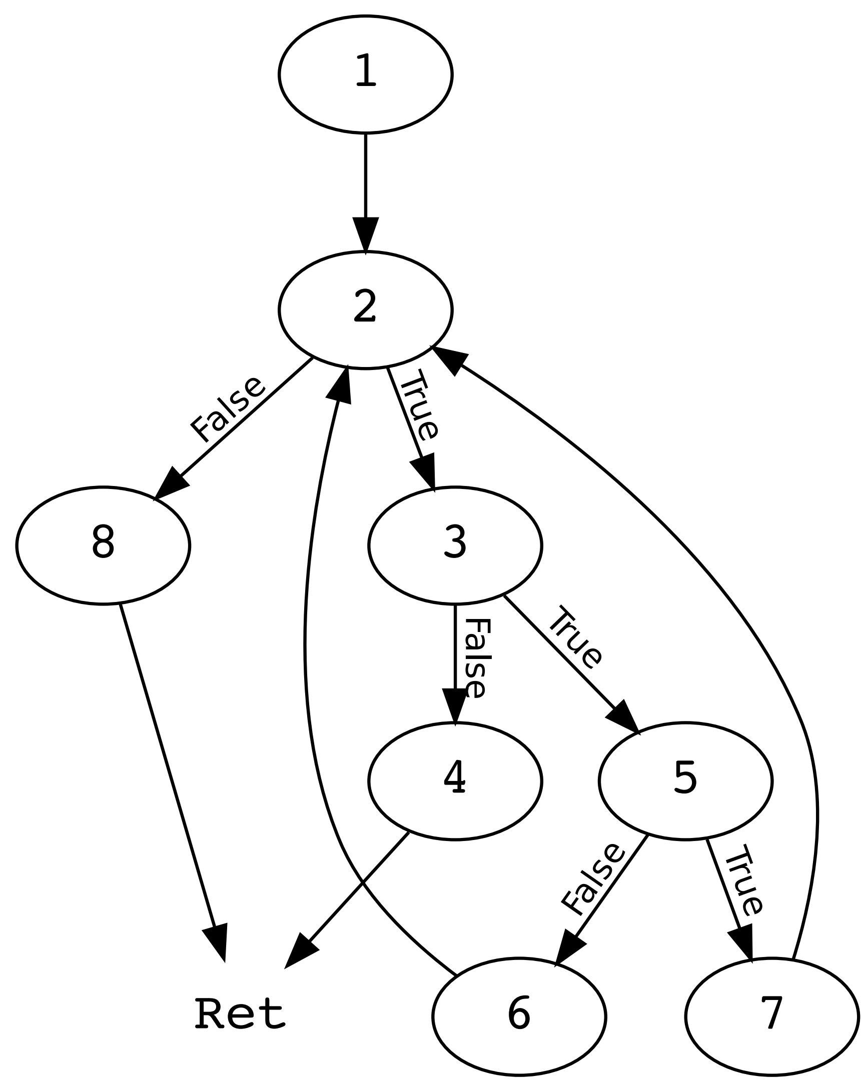


Translating C into graphs

```
struct node *
find (struct tree *t, int k) {
  struct node *p = t->trunk;
  while (p) {
    if (p->key == k)
      return p;
    else if (p->key < k)
      p = p->right;
    else
      p = p->left;
  }
  return NULL;
}
```

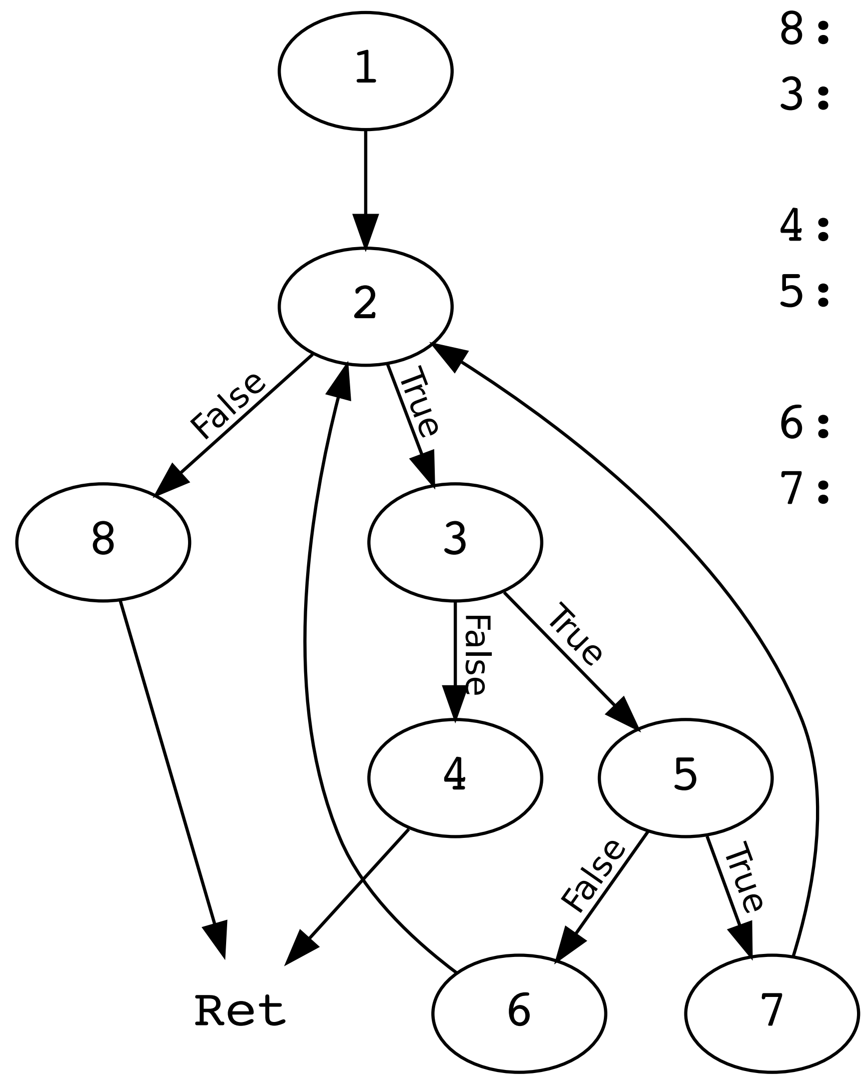
Translating C into graphs

```
struct node *
find (struct tree *t, int k) {
  struct node *p = t->trunk;
  while (p) {
    if (p->key == k)
      return p;
    else if (p->key < k)
      p = p->right;
    else
      p = p->left;
  }
  return NULL;
}
```



Translating C into graphs

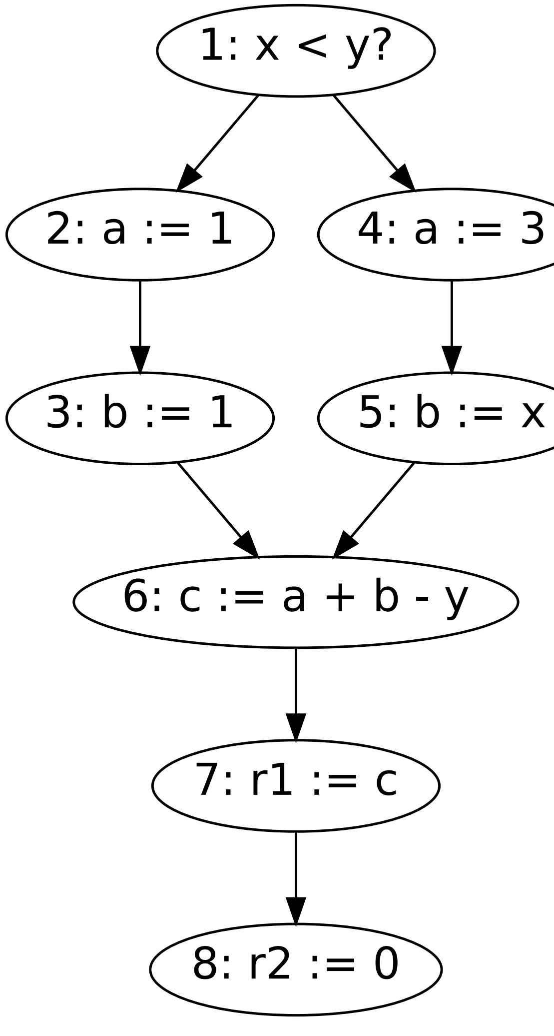
```
struct node *
find (struct tree *t, int k) {
  struct node *p = t->trunk;
  while (p) {
    if (p->key == k)
      return p;
    else if (p->key < k)
      p = p->right;
    else
      p = p->left;
  }
  return NULL;
}
```




```
1: p := Mem[t + 4];

2: p == 0 ?

8: ret := 0
3: Mem[p] == k ?

4: ret := p;
5: Mem[p] < k ?

6: p := Mem[p + 4];
7: p := Mem[p + 8];
```

Translating mc functions into graphs

```
f x y =
  let (a, b) = if x < y
    then (1, 2)
    else (3, x)
  in let c = a + b - y
  in (c, 0)
```



The SMT proof step

Following Pnuelli's original translation validation, we split the proof step:

Part 1: proof search (proof script construction)

Part 2: proof checking (checking the proof script)

The SMT proof step

Following Pnuelli's original translation validation, we split the proof step:

Part 1: proof search (proof script construction)

Part 2: proof checking (checking the proof script)

The proof scripts consist of a state space description and a tree of proof rules: Restrict, Split and Leaf.

The SMT proof step

Following Pnuelli's original translation validation, we split the proof step:

Part 1: proof search (proof script construction)

Part 2: proof checking (checking the proof script)

The proof scripts consist of a state space description and a tree of proof rules: Restrict, Split and Leaf.

The heavy lifting is done by calls to SMT solvers for both the proof search and checking.

Translating graphs into SMT exps

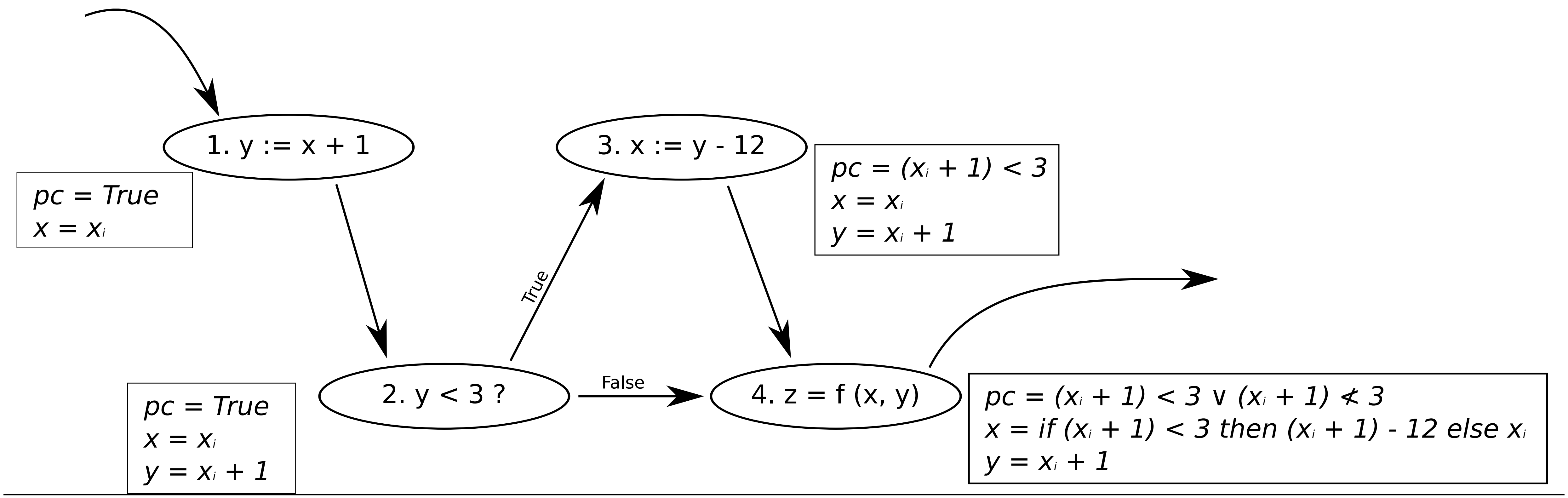
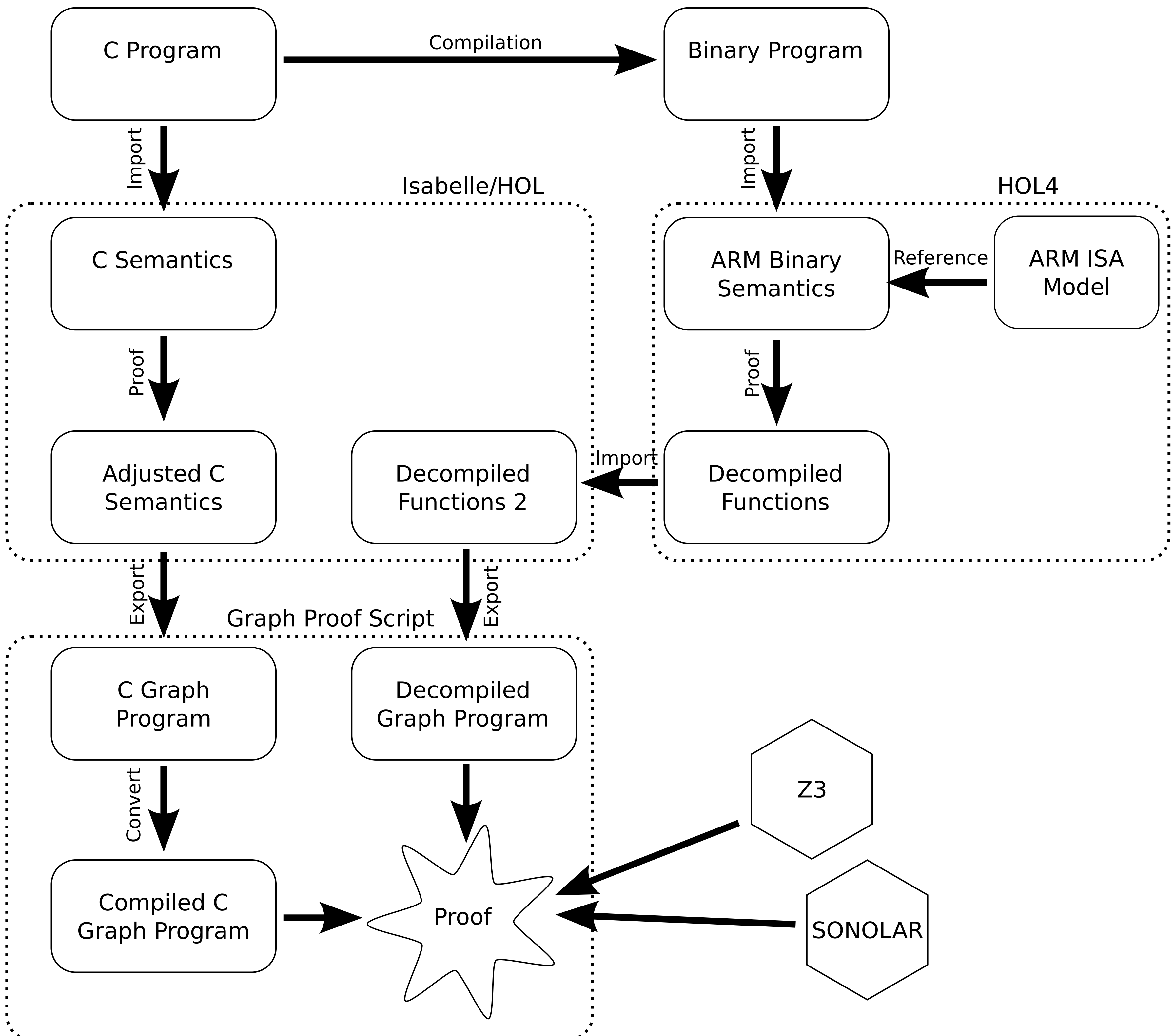


Figure 5. Example Conversion to SMT

Here: 'pc' is the accumulated path condition and variables (x, y etc.) are values w.r.t. inputs (x_i, y_i, etc.)

(The actual translation avoids a blow up in size...)

Full workflow



Results and Summary

We have (almost) proved a full connection between the verified C and seL4 binary.

	gcc -O1	gcc -O2
Instructions in Binary	11 736	12 299
Decompiled Functions	260	259
- Placeholders		3
Function Pairings	260	225
Successes	234	145
Failures	0	18
Aborted	26	62
- Machine Operations	21	13
- Nested Loops	3	2
- Machine Operations Inlined	2	47
Time Taken in Proof	59m	4h 23m

Table 1. Decompilation and Proof Results