ENTROPY 2018 (ENabling TRust through Os Proofs...and beYond), France

Verification of the GCC-generated binary of the seL4 microkernel

Thomas Sewell[1], Magnus Myreen[2], Gerwin Klein[1]

the clever PhD student who did the hard part of the work

today's speaker (borrowing some slides from Sewell)

[1] Data61 CSIRO & UNSW, Australia
[2] Chalmers, Sweden

DATA 61 CSIRO UNSW SYDNEY CHALMERS UNIVERSITY OF TECHNOLOGY

L4.verified

seL4 = a formally verified general-purpose microkernel

about 10,000 lines of C code and assembly

> 500,000 lines of Isabelle/HOL proofs

Assumptions in L4.verified

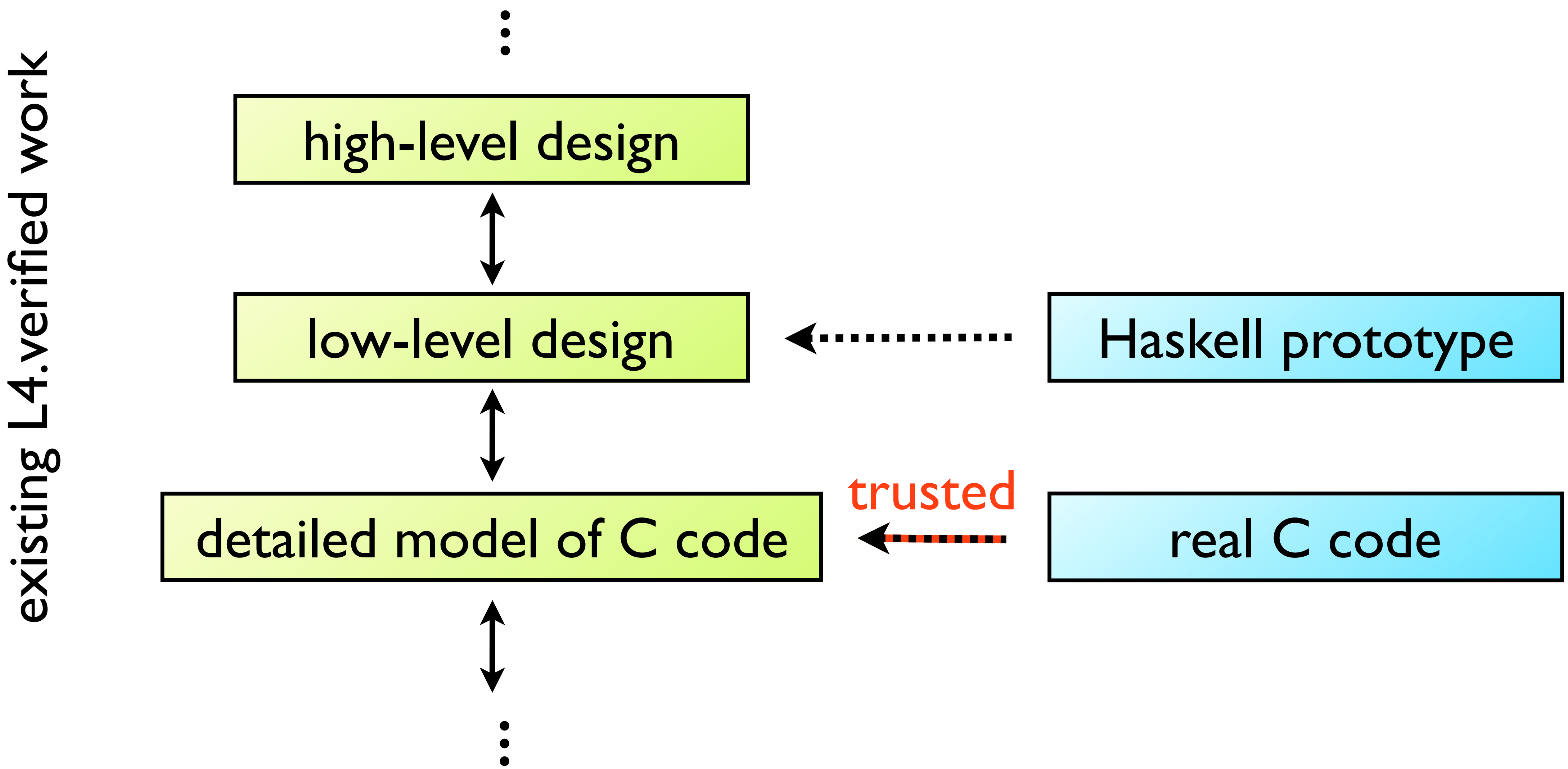
L4.verified project assumes correctness of:

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

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

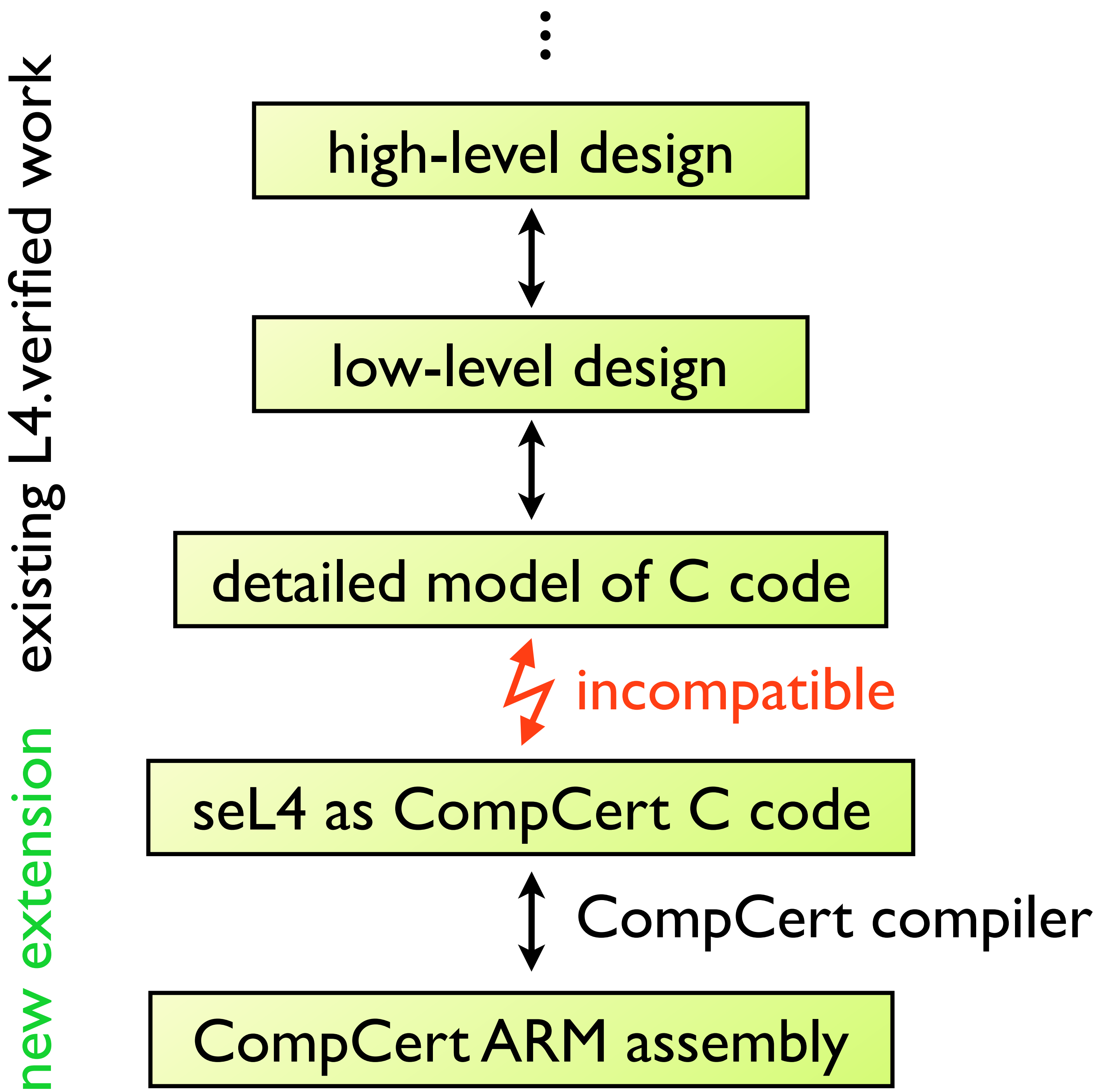
And also to validate L4.verified's C semantics.

Aim: extend downwards



Aim: remove need to trust C compiler and C semantics

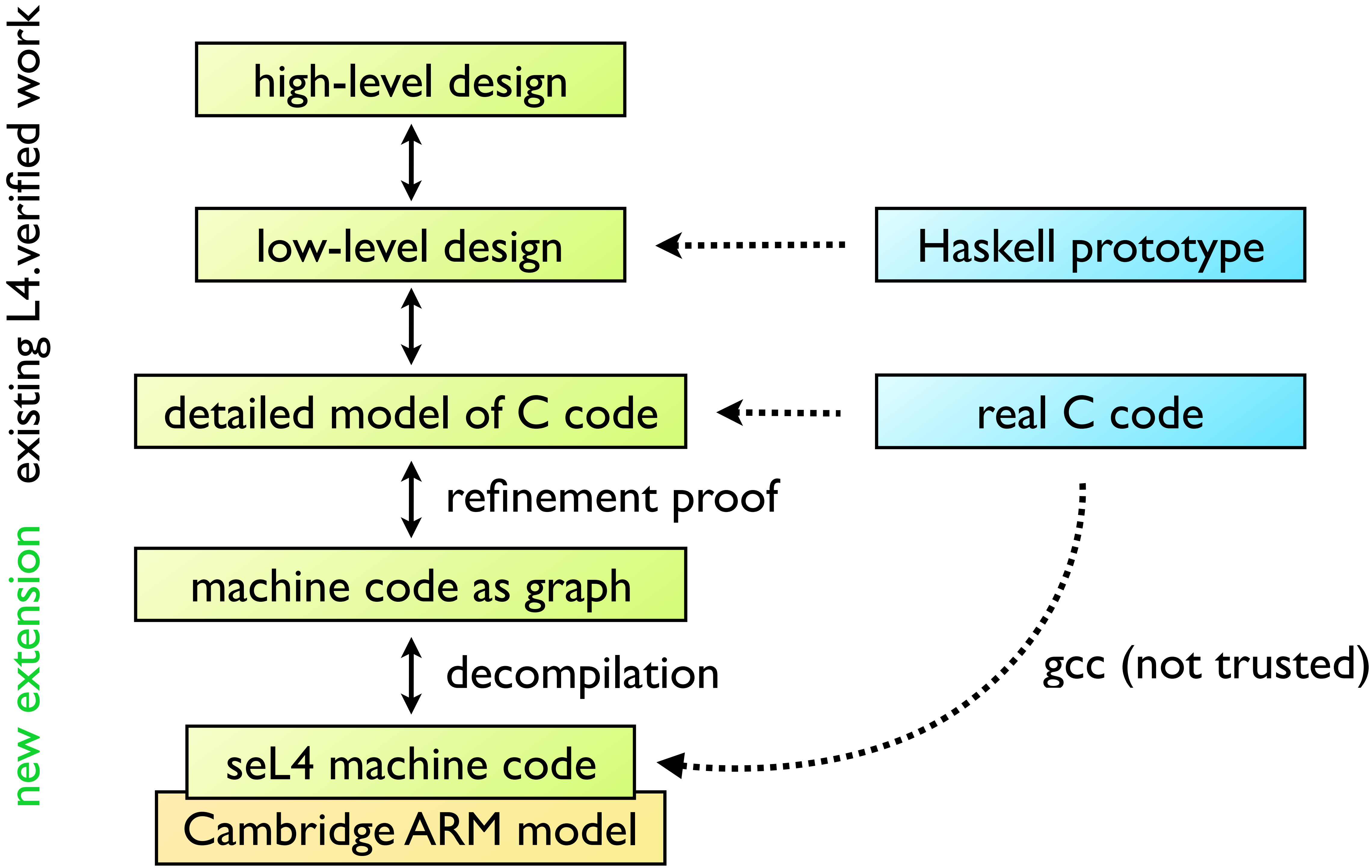
Connection to CompCert



Incompatible:

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

Using Cambridge ARM model

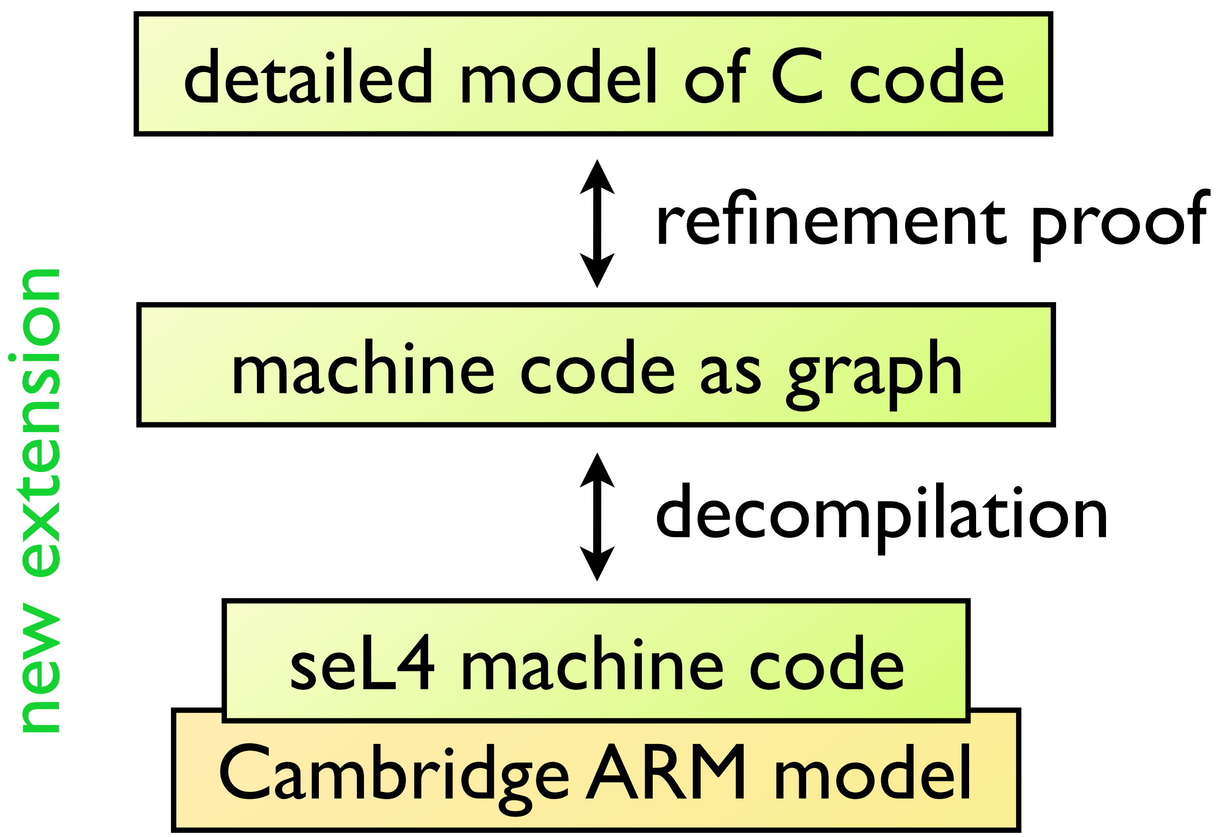


Translation validation

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



Talk ~~Part~~ 1: proof-producing decompilation

- generating functions / graphs
- stack vs heap

Talk ~~Part~~ 2: pseudo compilation and SMT refinement proof

- C semantics
- SMT proof search and proof checking
- examples
- complicated cases

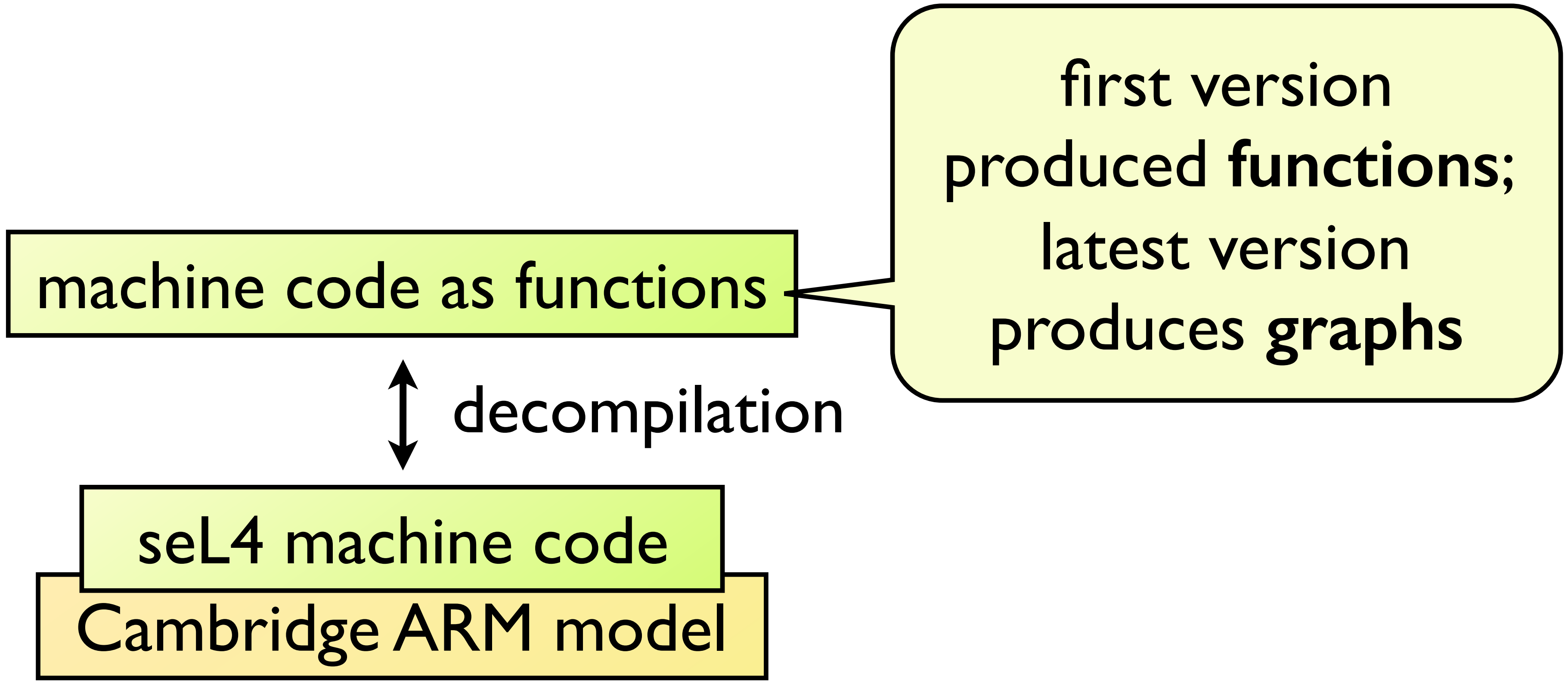
Cambridge ARM model

$\boxed{\text{Cambridge ARM model}}$ developed by Anthony Fox

- detailed 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/

Part 1: decompilation



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

word arithmetic

Resulting function:

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

word 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

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

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

{ LR lr * PC (p+8) }
p+8 :
{ 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 ARM instructions (lines of assembly)
  - ✓ 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 is 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, and cause memory ops in machine code

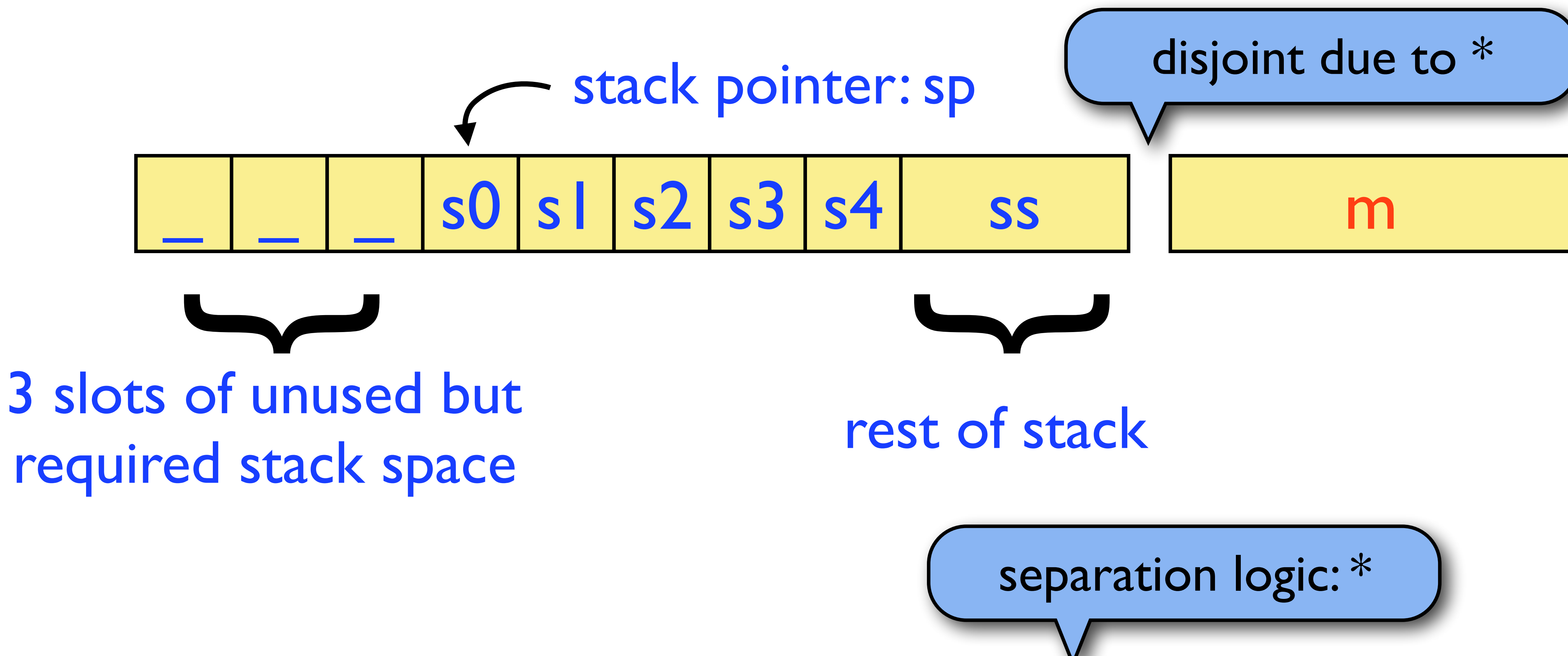
gcc

```
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
```

... that are not present in C semantics.

Solution (early version)

Use separation-logic inspired approach



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

Solution

```
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.

The new triples make it seems as if stack accesses are separate from the rest of memory.

Result (early version)

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
```

What about arrays on the stack?

with sometimes too little information

The new triples make it seems as if stack accesses are separate from the rest of memory.

Disadvantage: the au

Later version

Stack load/stores become accesses to “stack memory”.

```
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,sp,stack) =
  let r1 = r1 + r0 in
  let r1 = r1 + r2 in
  let r2 = stacks(sp) in
  let r1 = r1 + r3 in
  let r0 = r1 + r3 in
  let (r2,r3) = (stack(sp+4),stack(sp+8)) in
  let r0 = r0 + r2 in
  let r0 = r0 + r3 in
  let r3 = stack(sp+12) in
  let r0 = r0 + r3 in
  let r0 = r0 >> 3 in
   r0
```

results in proof obligations higher up

In certificate theorems: stack * heap

Correct memory after compilation

Our C semantics forbids pointers to the stack.

We also eliminate padding, clearly separating:

- the heap, under user control.
- the stack, under compiler control.

Enables a simple notion of correct compilation:

$$\forall(\mathit{in}, \mathit{in\_heap}) \in \mathrm{domain}(\mathfrak{C}).\ \mathfrak{C}(\mathit{in}, \mathit{in\_heap}) = \mathfrak{B}(\mathit{in}, \mathit{in\_heap})$$

This would be difficult with higher level optimisations.

C semantics

binary (machine code) semantics

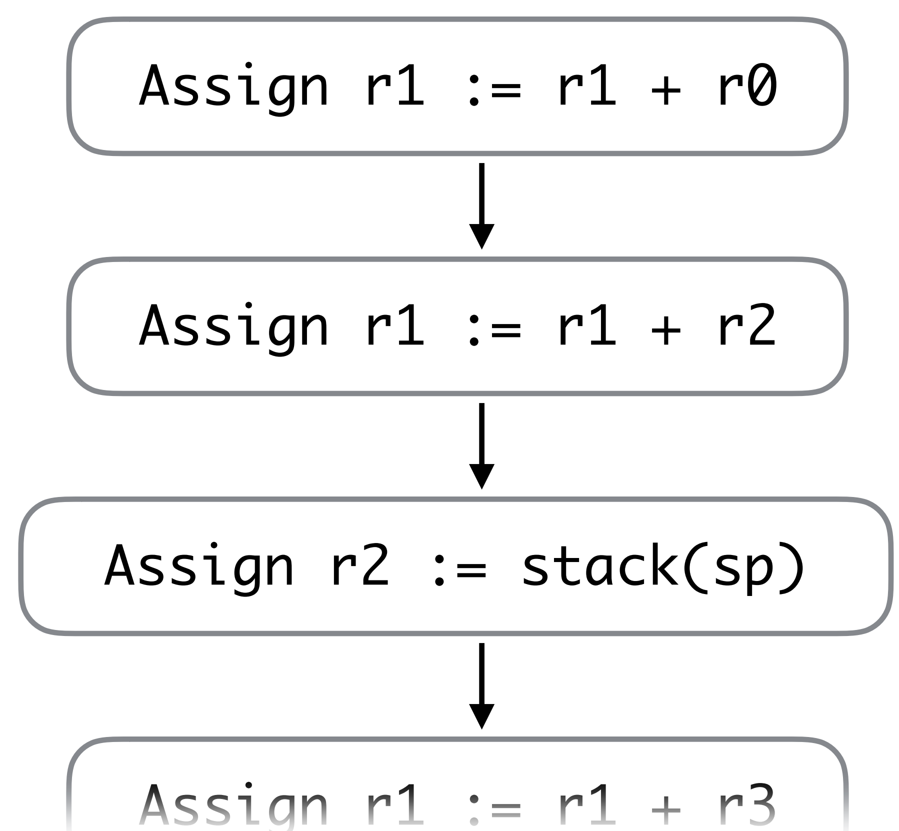
Other tricky cases

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

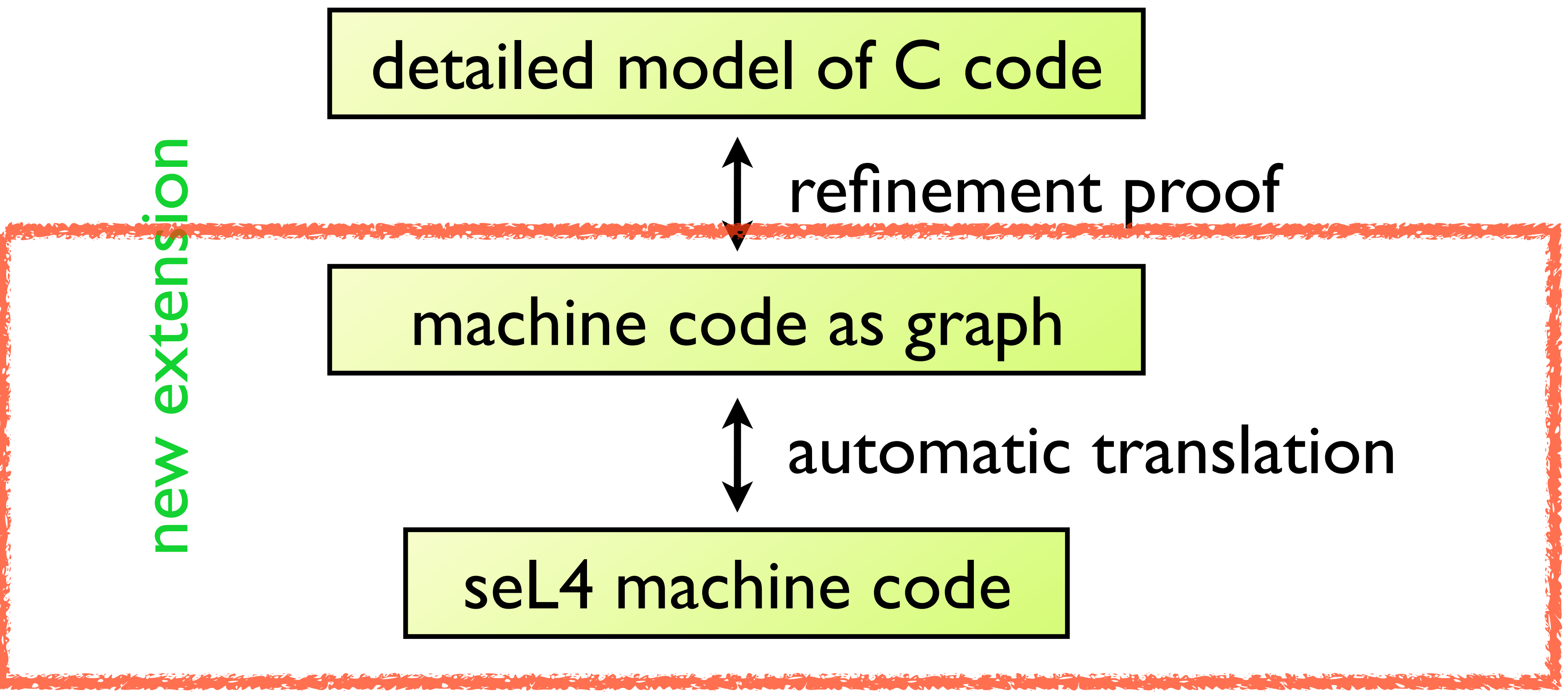
Latest decompiler

- produces a graph instead of a function
  - functions are good for interactive proofs
  - graphs seem better for automation here

```
avg8(r0,r1,r2,r3,sp,stack) =
  let r1 = r1 + r0 in
  let r1 = r1 + r2 in
  let r2 = stacks(sp) in
  let r1 = r1 + r3 in
  let r0 = r1 + r3 in
  let (r2,r3) = (stack(sp+4),stack(sp+8)) in
  let r0 = r0 + r2 in
  let r0 = r0 + r3 in
  let r3 = stack(sp+12) in
```



Moving to Part 2



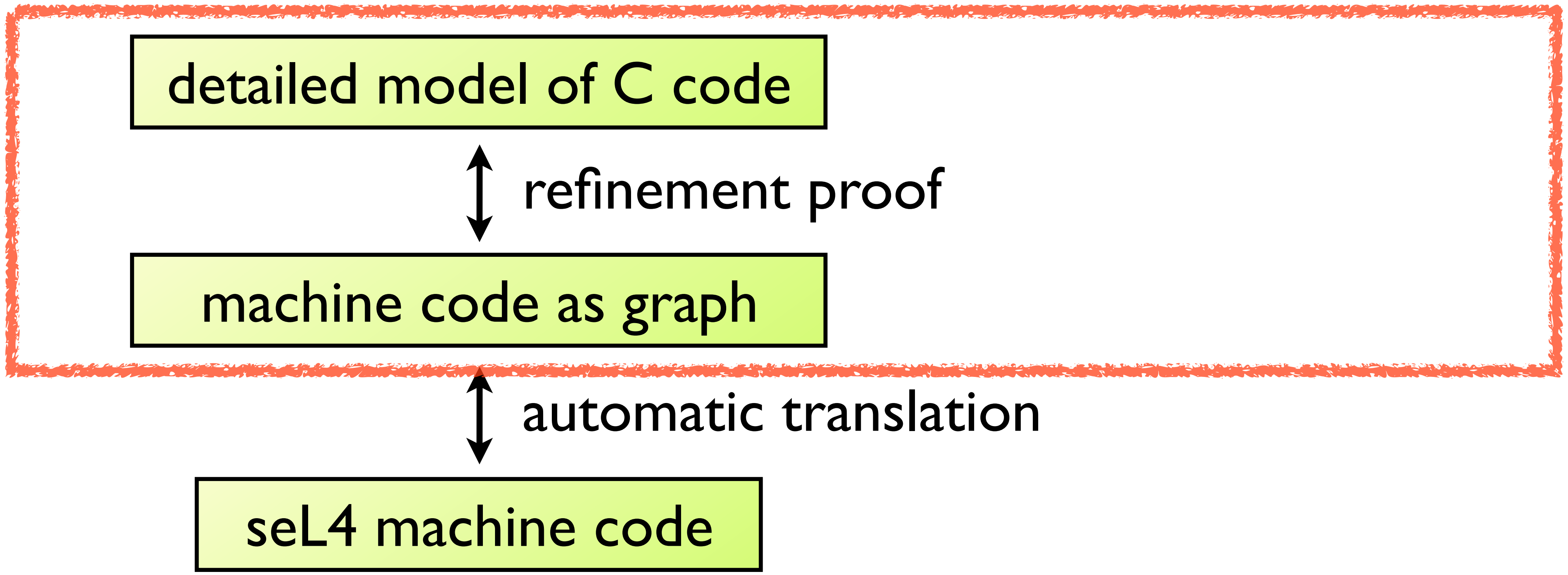
Moving to Part 2

Questions about Part 1?

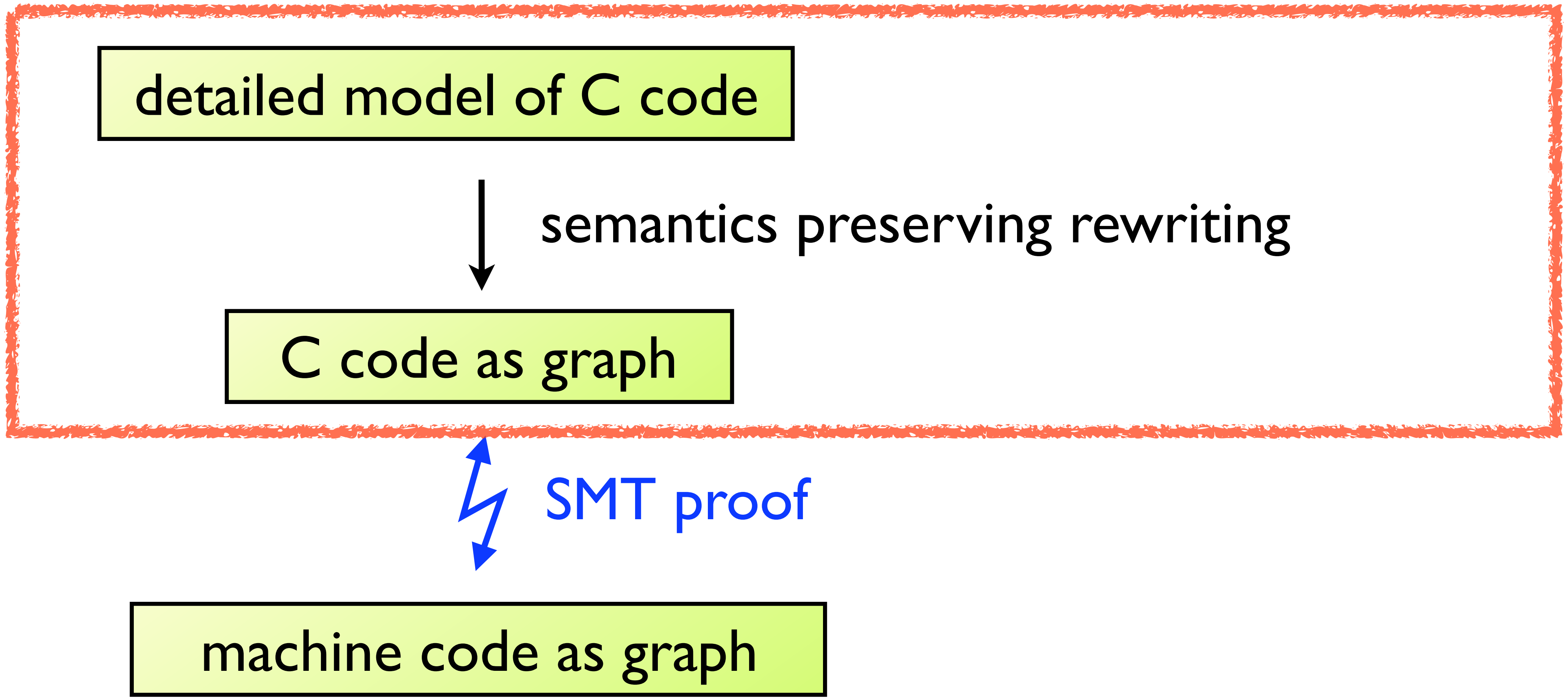
… before we continue to Part 2

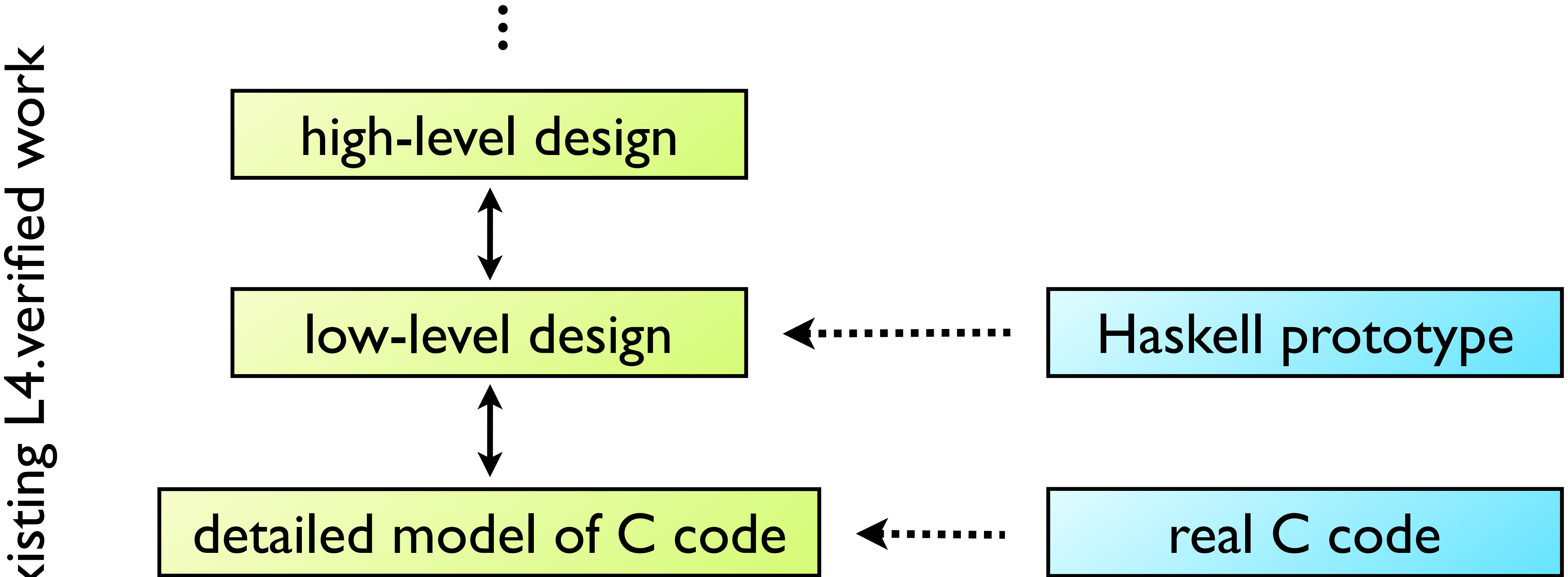
Sydney Harbour Bridge during construction

Part 2


detailed model of C code
refinement proof
machine code as graph
automatic translation
seL4 machine code

Approach for refinement proof





the C semantics is produced on import into Isabelle/HOL

```
if (...)  {...}   ⇒   IF (...)  THEN ...  FI
f (1, 2);         ⇒   CALL f_'proc (1, 2);;
x ++;             ⇒    Guard {´x <=s ´x + 1}
                           (´x :== ´x + 1);;
*p = *q;          ⇒    Guard {ptr_valid ´p}
                         Guard {ptr_valid ´q}
                           mem :== h_upd ´p
                             (h_val ´q ´mem) ´mem
```

partial semantics to account for **undefined behaviour**

Why not just trust the C compiler?

The `ptr_valid` assertion used in `Guard` is subtle.

The **object rule** says that a pointers may come from arithmetic within an object, `&` and `malloc`.

What about casts from numbers?

```
(pt_t *)(pt[x] & 0xFFFFF000)
```

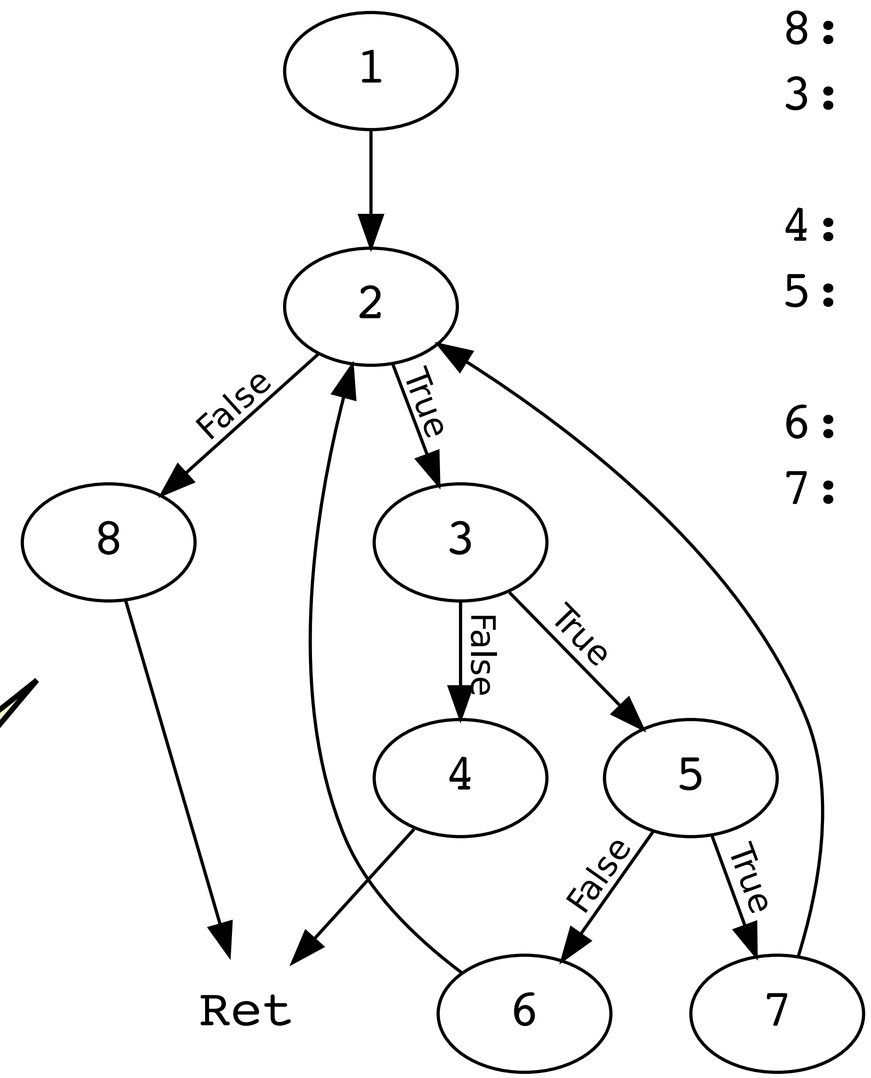
There are multiple interpretations of the C language.

- **NICTA seL4:** Liberal, portable assembler, soundy.
  - Strict aliasing rule but not object rule.
- **CompCert:** Conservative.

Translating C into graphs

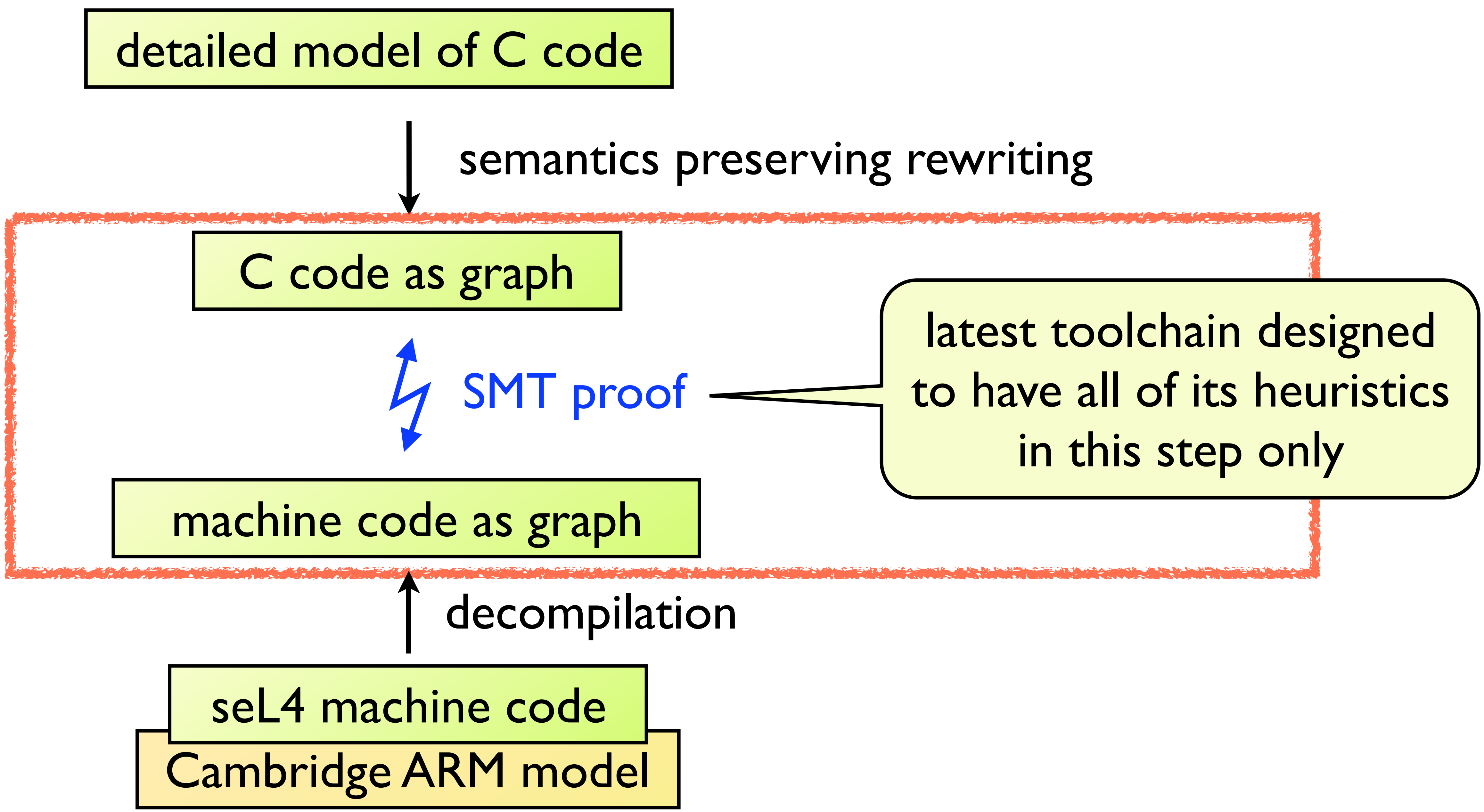
```
struct node *
find (struct tree *t, int k) {
  struct node *p = t->trunk;
  while (p) {
    if (p->key == k)
      return p;
    else if (p->key < k)
      p = p->right;
    else
      p = p->left;
  }
  return NULL;
}
```

```
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];
```



the ptr_valid assertions are omitted from the figure

Bridging the gap



The SMT proof step

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

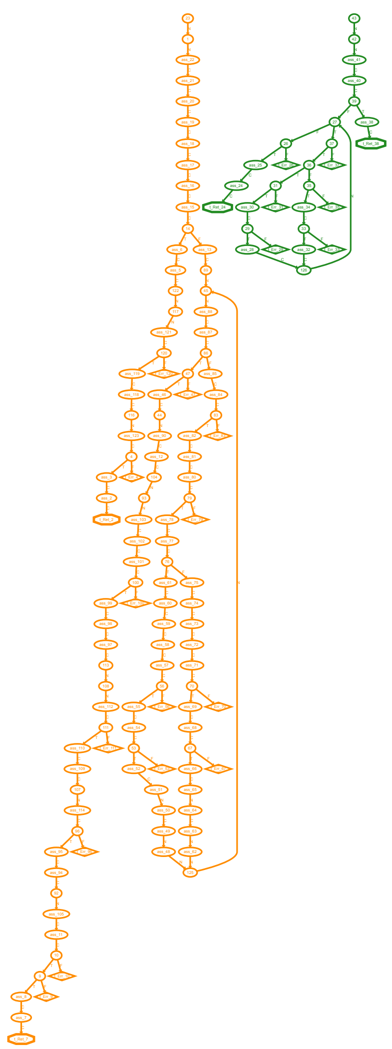
Part 1: proof search (proof script construction)

Part 2: proof checking (checking the proof script)

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

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

Generated proof scripts



Proof objects contain:

- An **inlining** of all needed function bodies into one space.
- **Restrict** rules, which observe that a given point in a loop may be reached only *n* times.
- **Split** rules, which observe that a C loop point is reached as often as a loop point in the binary.
  - Checked by *k*-induction.
  - Parameter *eqs* must relate enough of binary state to C state to relate events after the loop.

Translating graphs into SMT exps

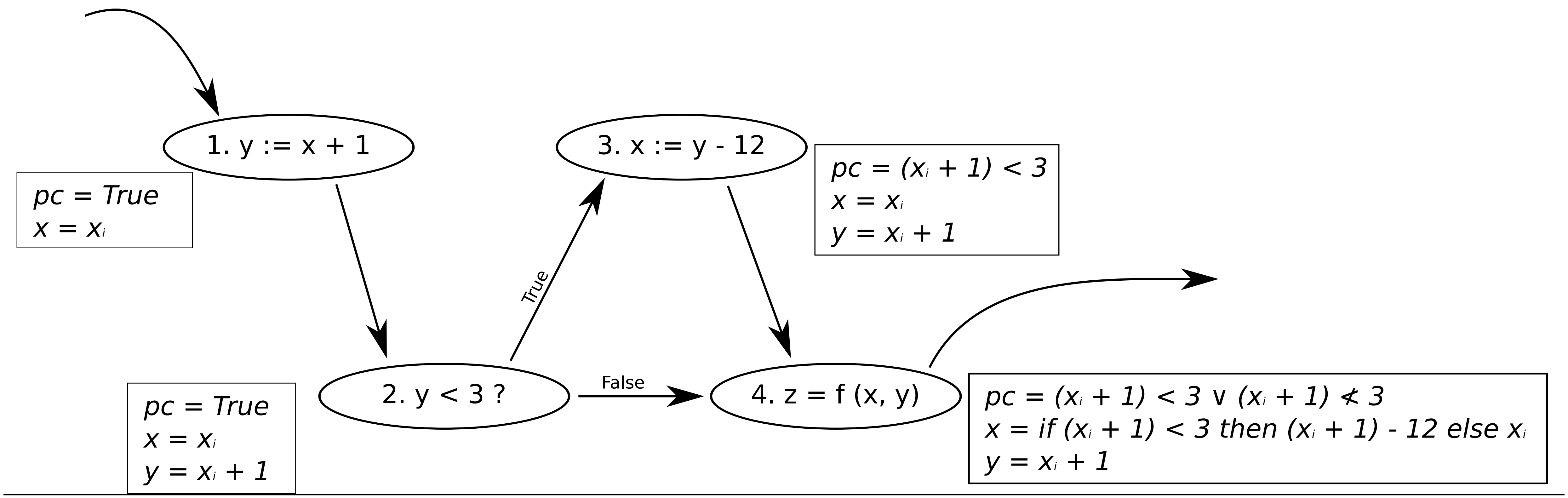


Figure 5. Example Conversion to SMT

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...)

Easy for SMT (1)

```
int
f1 (unsigned int x) {
  return ((x >> 4) & 15) == 3;
}
int
f2 (unsigned int x) {
  return (x & (15 << 4)) == (3 << 4);
}
int
f3 (unsigned int x) {
  return ((x << 24) >> 28) == 3;
}
int
f4 (unsigned int x) {
  return ((x & (15 << 4)) | (3 << 4)) == 0;
}
```

Word games: solved problem.

- "Bit Vector" SMT theory.

Easy for SMT (2)

```
void
f (struct foo *x, int y) {
  struct foo f = *x;
  f.a += y;
  f.b -= y;
  f = do_the_thing (f);
  *x = f;
}
```

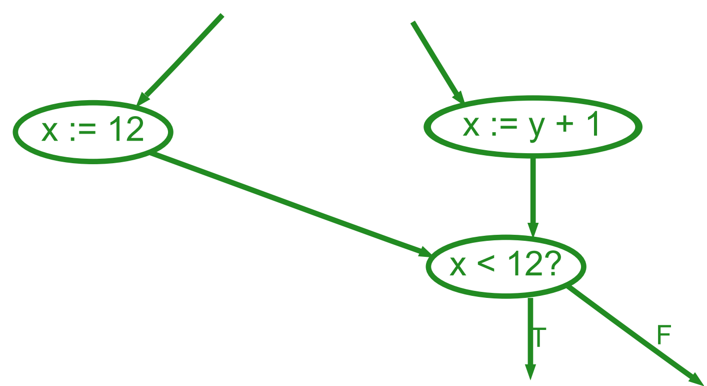
Memory optimisation: mostly solved problem.

- "Array" SMT theory.
- `QF_ABV` SMT logic.

SMT use summary

SMT problems generated contain:

- Fixed-length values and arithmetic: `word32, +, -, <=` etc.
- Arrays to model the heap: `heap :: word30 => word32`.
- If-then-else operators to handle multiple paths.



- Validity assertions and needed inequalities:
  $\texttt{ptr1\_valid} \mathbin{\&} \texttt{ptr2\_valid} \Rightarrow ptr1 > ptr2 + 7 \vee ptr2 > ptr1 + 15.$

Strong compatibility with **SMTLIB2 QF_ABV**.

Examples

Example 1

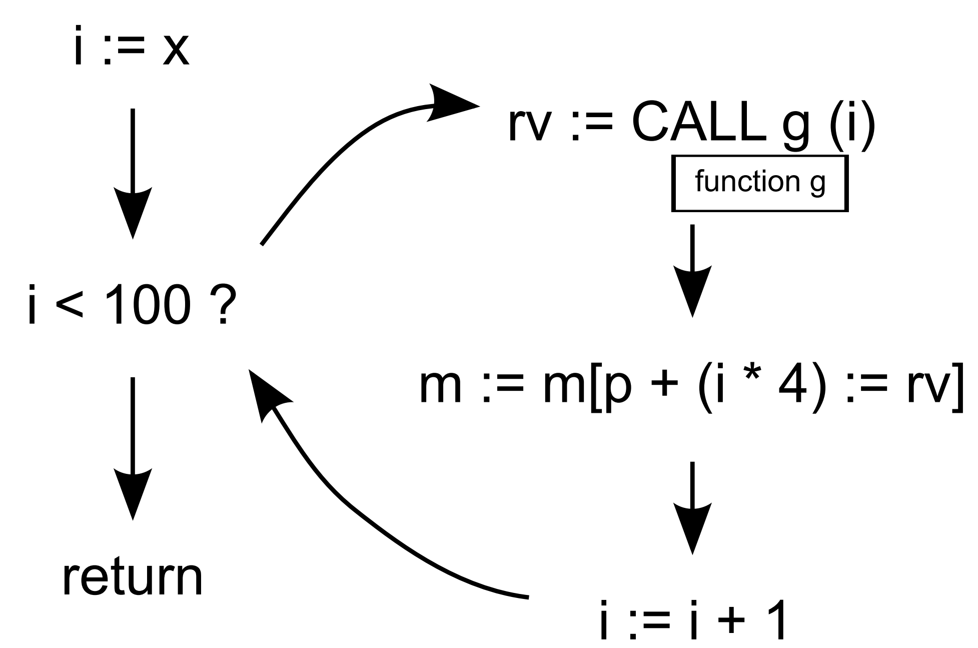
```c
int
g (int i) {
  return i * 8 + (i & 15);
}

void
f (int *p, int x) {
  int i;

  for (i = x; i < 100; i ++) {
    p[i] = g (i);
  }
}
```

```
00000000 <g>:
   0:   e200300f        and     r3, r0, #15
   4:   e0830180        add     r0, r3, r0, lsl #3
   8:   e12fff1e        bx      lr

0000000c <f>:
   c:   e3510063        cmp     r1, #99 ; 0x63
  10:   e52d4004        push    {r4} ; (str r4, [sp, #-4]!)
  14:   ca000021        bgt     a0 <f+0x94>
  18:   e1a02181        lsl     r2, r1, #3
  1c:   e201c00f        and     ip, r1, #15
  20:   e2813001        add     r3, r1, #1
  24:   e2614063        rsb     r4, r1, #99       ; 0x63
  28:   e08cc002        add     ip, ip, r2
  2c:   e0801101        add     r1, r0, r1, lsl #2
  30:   e3530064        cmp     r3, #100          ; 0x64
  34:   e2044001        and     r4, r4, #1
  38:   e481c004        str     ip, [r1], #4
  3c:   e2820008        add     r0, r2, #8
  40:   0a000016        beq     a0 <f+0x94>
  44:   e3540000        cmp     r4, #0
  48:   0a000006        beq     68 <f+0x5c>
  4c:   e203200f        and     r2, r3, #15
  ...
  94:   e2800008        add     r0, r0, #8
  98:   e2821004        add     r1, r2, #4
  9c:   1affffff1       bne     68 <f+0x5c>
  a0:   e49d4004        pop     {r4}              ; (ldr r4, [sp], #4)
  a4:   e12fff1e        bx      lr
```

Example 1 (cont)

```
void
f (int *p, int x) {
  int i;

  for (i = x; i < 100; i ++) {
    p[i] = g (i);
  }
}
```

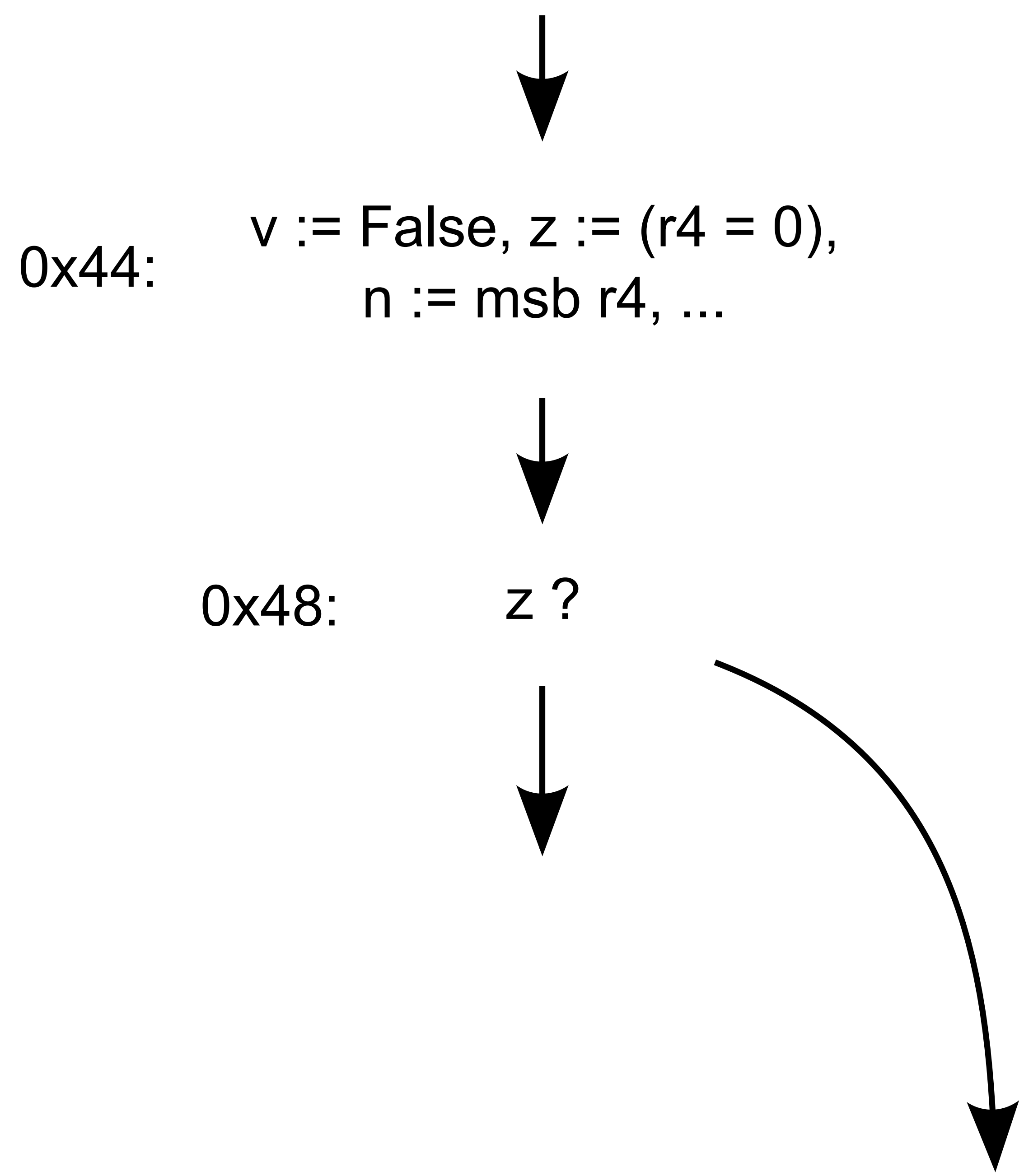
The C code as a graph:



Example 1 (cont)

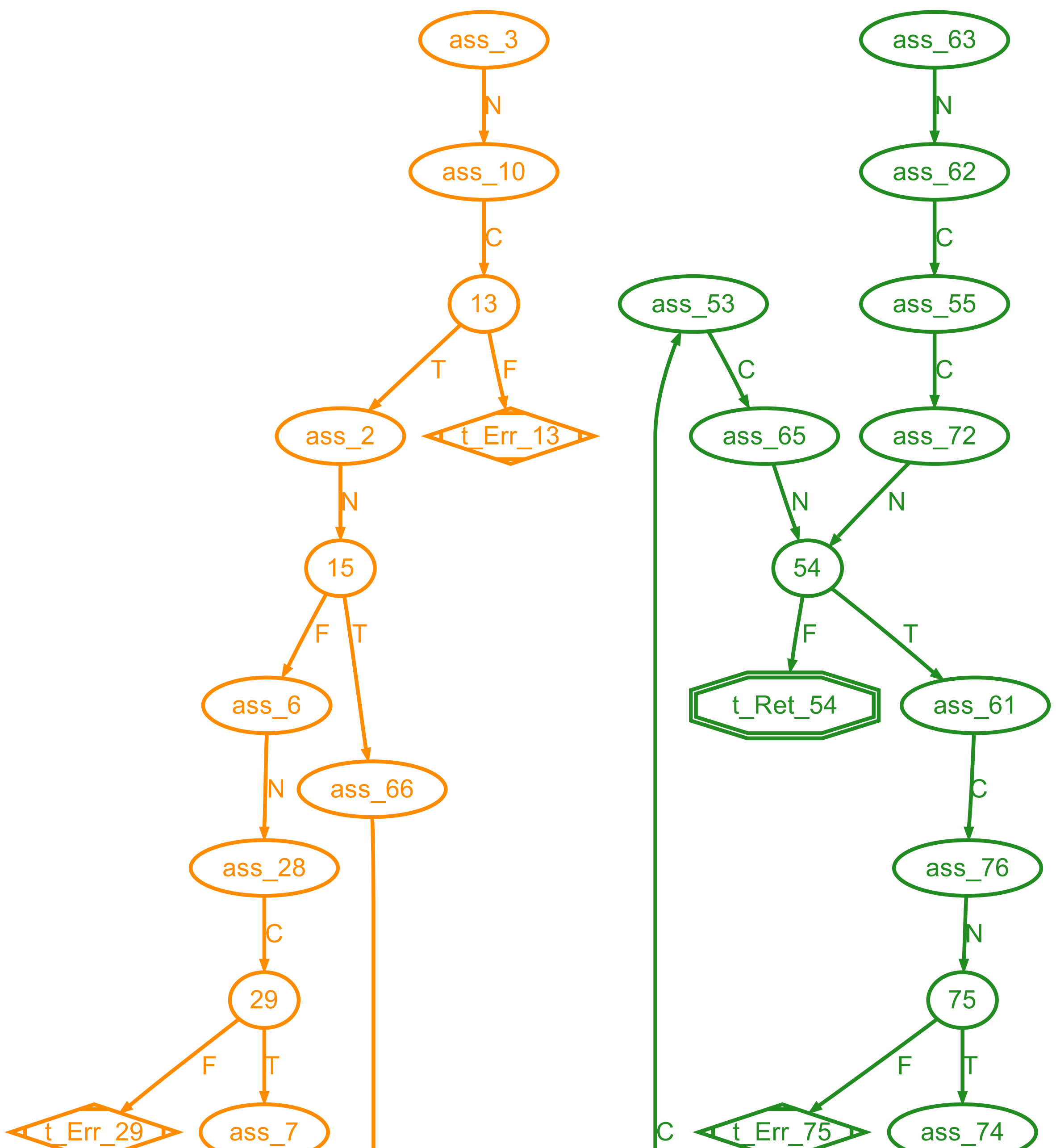
```
00000000 <g>:
   0:   e200300f        and     r3, r0, #15
   4:   e0830180        add     r0, r3, r0, lsl #3
   8:   e12fff1e        bx      lr

0000000c <f>:
   c:   e3510063        cmp     r1, #99 ; 0x63
  10:   e52d4004        push    {r4} ; (str r4, [sp, #-4]!)
  14:   ca000021        bgt     a0 <f+0x94>
  18:   e1a02181        lsl     r2, r1, #3
  1c:   e201c00f        and     ip, r1, #15
  20:   e2813001        add     r3, r1, #1
  24:   e2614063        rsb     r4, r1, #99      ; 0x63
  28:   e08cc002        add     ip, ip, r2
  2c:   e0801101        add     r1, r0, r1, lsl #2
  30:   e3530064        cmp     r3, #100         ; 0x64
  34:   e2044001        and     r4, r4, #1
  38:   e481c004        str     ip, [r1], #4
  3c:   e2820008        add     r0, r2, #8
  40:   0a000016        beq     a0 <f+0x94>
  44:   e3540000        cmp     r4, #0
  48:   0a000006        beq     68 <f+0x5c>
  4c:   e203200f        and     r2, r3, #15
  ...
  94:   e2800008        add     r0, r0, #8
  98:   e2821004        add     r1, r2, #4
  9c:   1affffff1       bne     68 <f+0x5c>
  a0:   e49d4004        pop     {r4}             ; (ldr r4, [sp], #4)
  a4:   e12fff1e        bx      lr
```

The machine code as a graph:



Example 1 (cont)

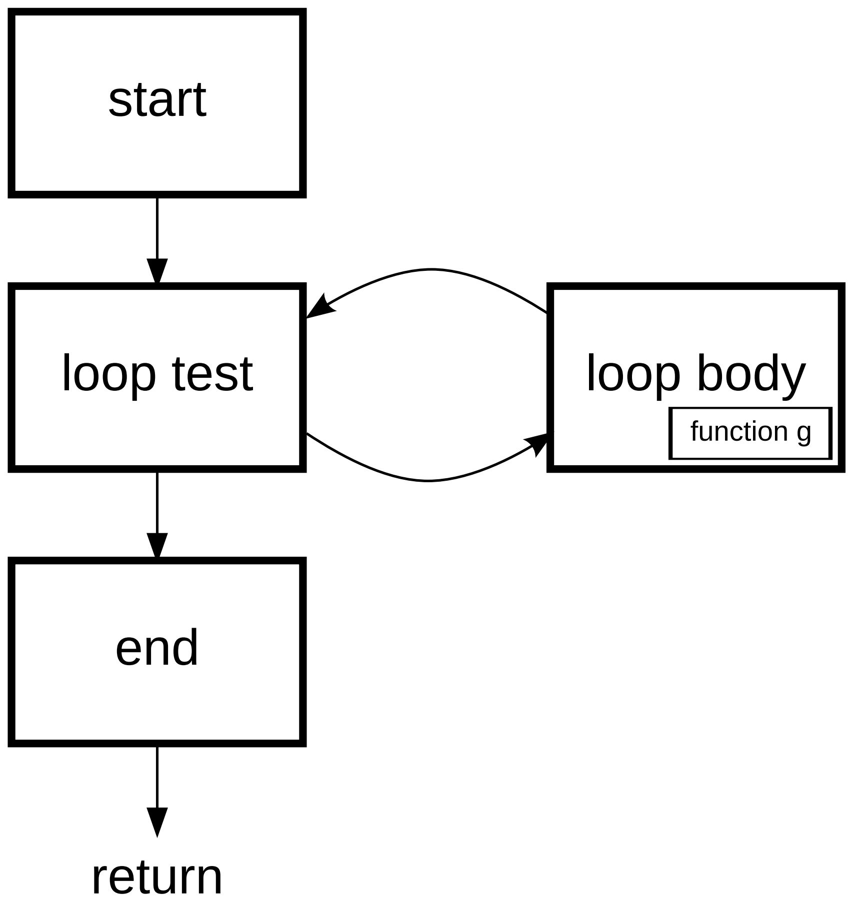
We are to prove that these compute the same:

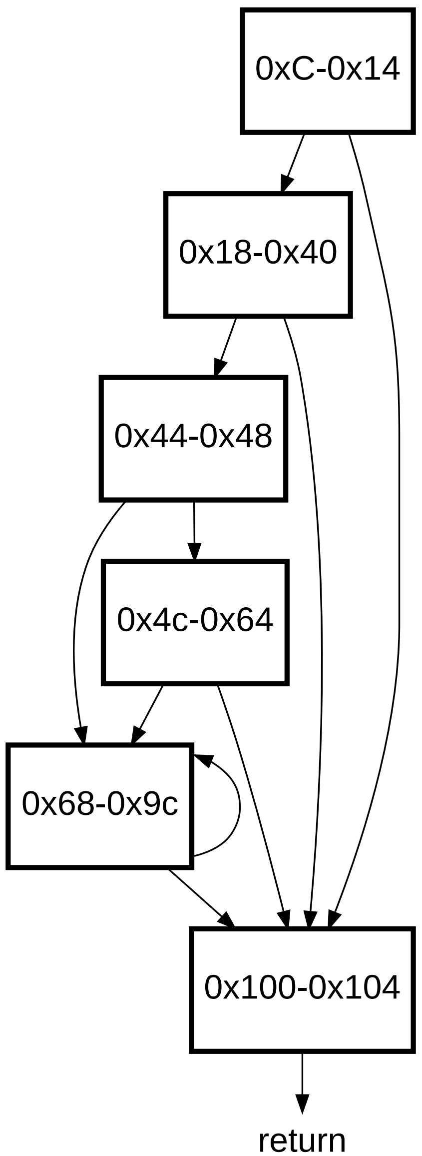


Example 1 (cont)

We are to prove that these compute the same:

(simplified view of graphs)





Example 1 (cont)

What is going on?

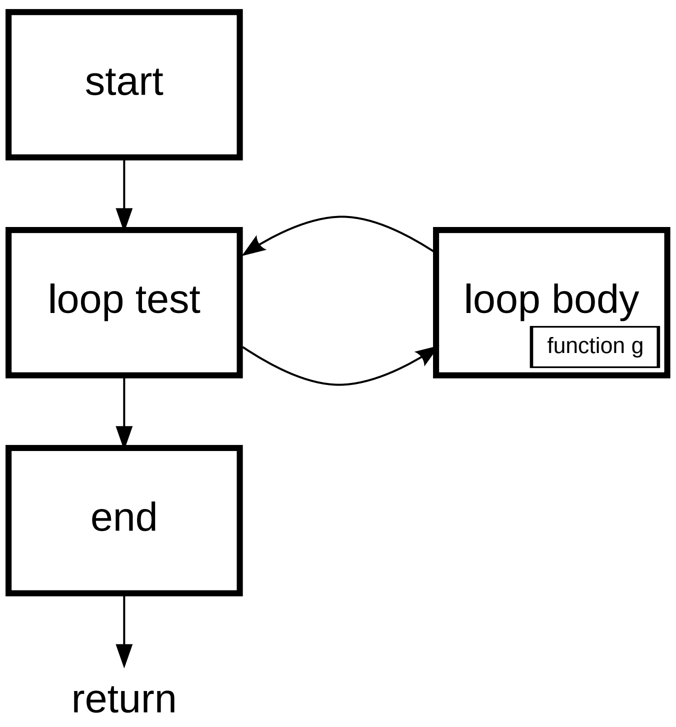
```
for (i = x; i < 100; i ++) {
```

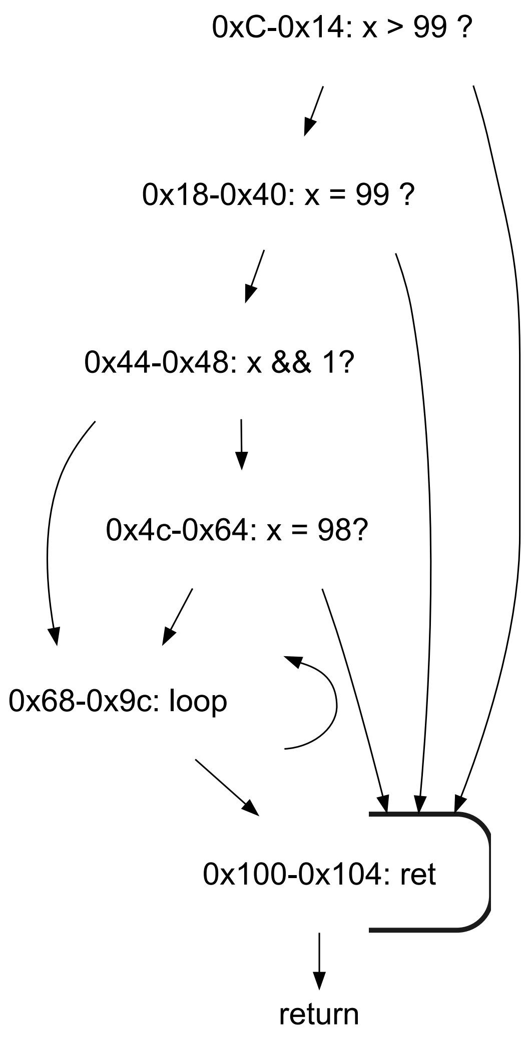
The loop has been unrolled.

The branches all encode `i < 100`.

Proof of correctness:

- relate the sequences of loop body visits.

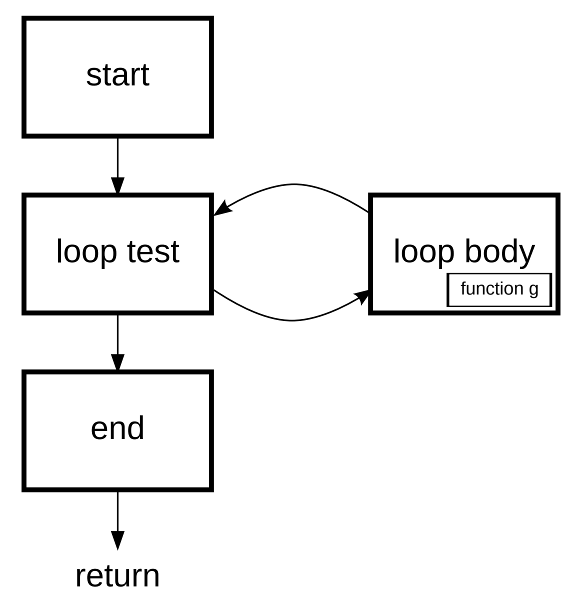


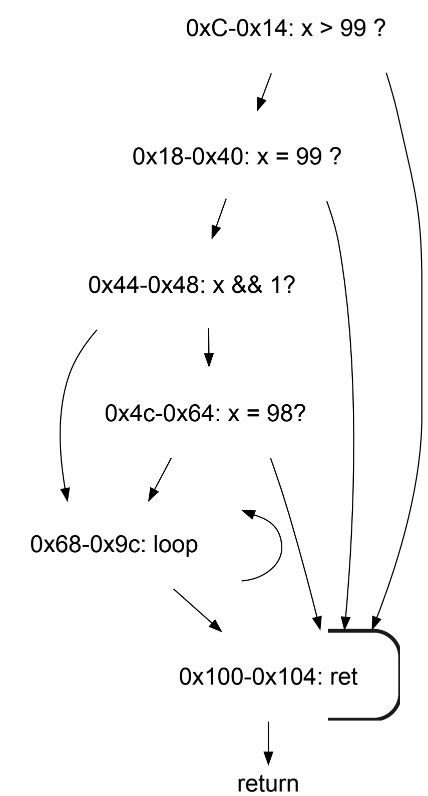


Example 1 (cont)

Proof of correctness:

1. **Case split** on execution of `04c`:
    - Consider even case
2. Relate visits to `0x68` to visits 3, 5, 7, . . . to body by **induction**.
3. **Case split** on related sequences:
    - Infinite case.
    - Init case: < 4 visits to body. Expand.
    - Loop case: $2n$ visits to body for some $n > 1$. Expand.



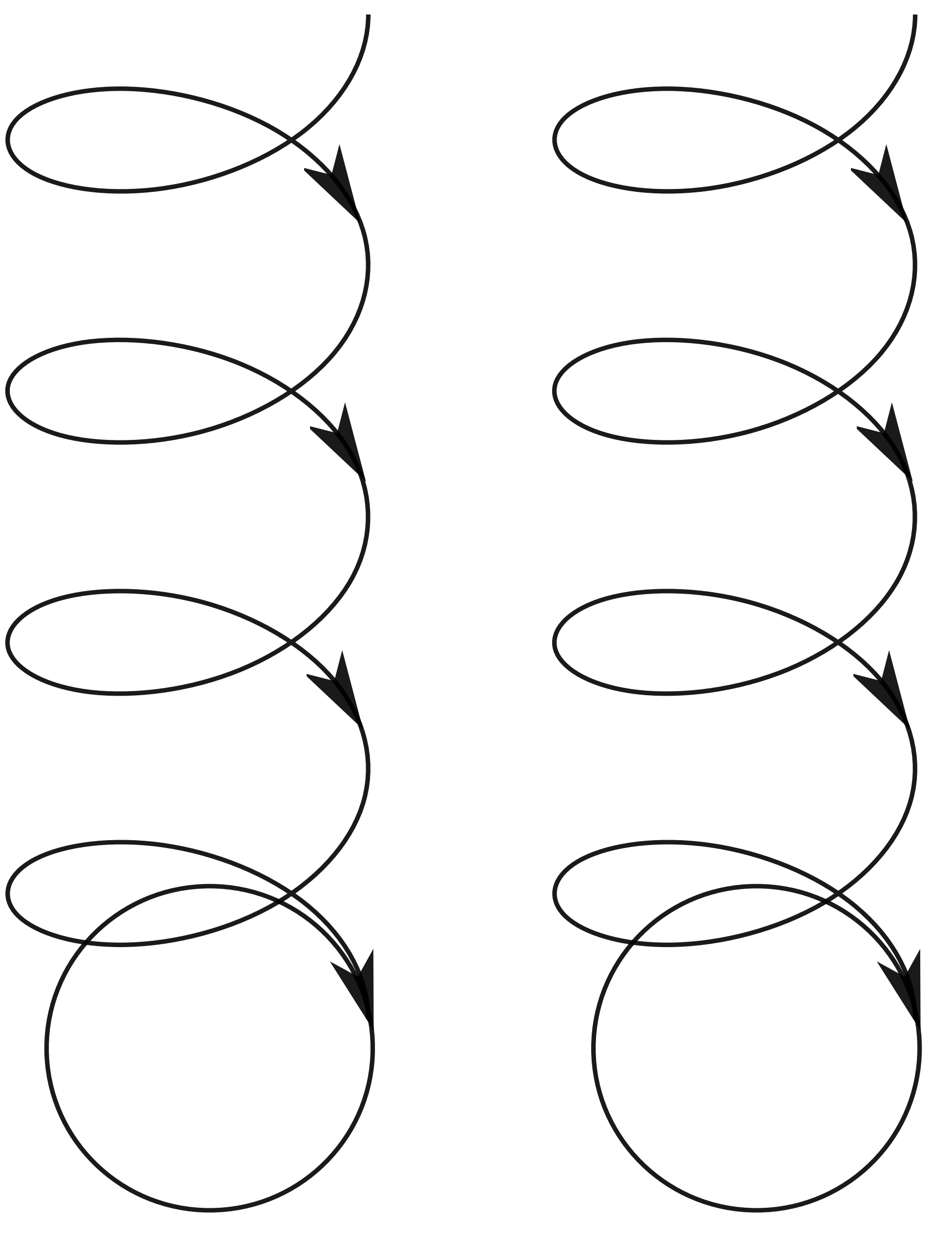


The proof search script discovers this proof automatically.

Example 1 (cont)

Proof search:

- Unroll the first few loop iterations.
- Produce SMT model.
- Look for coincidences.
- Check for counterexamples.

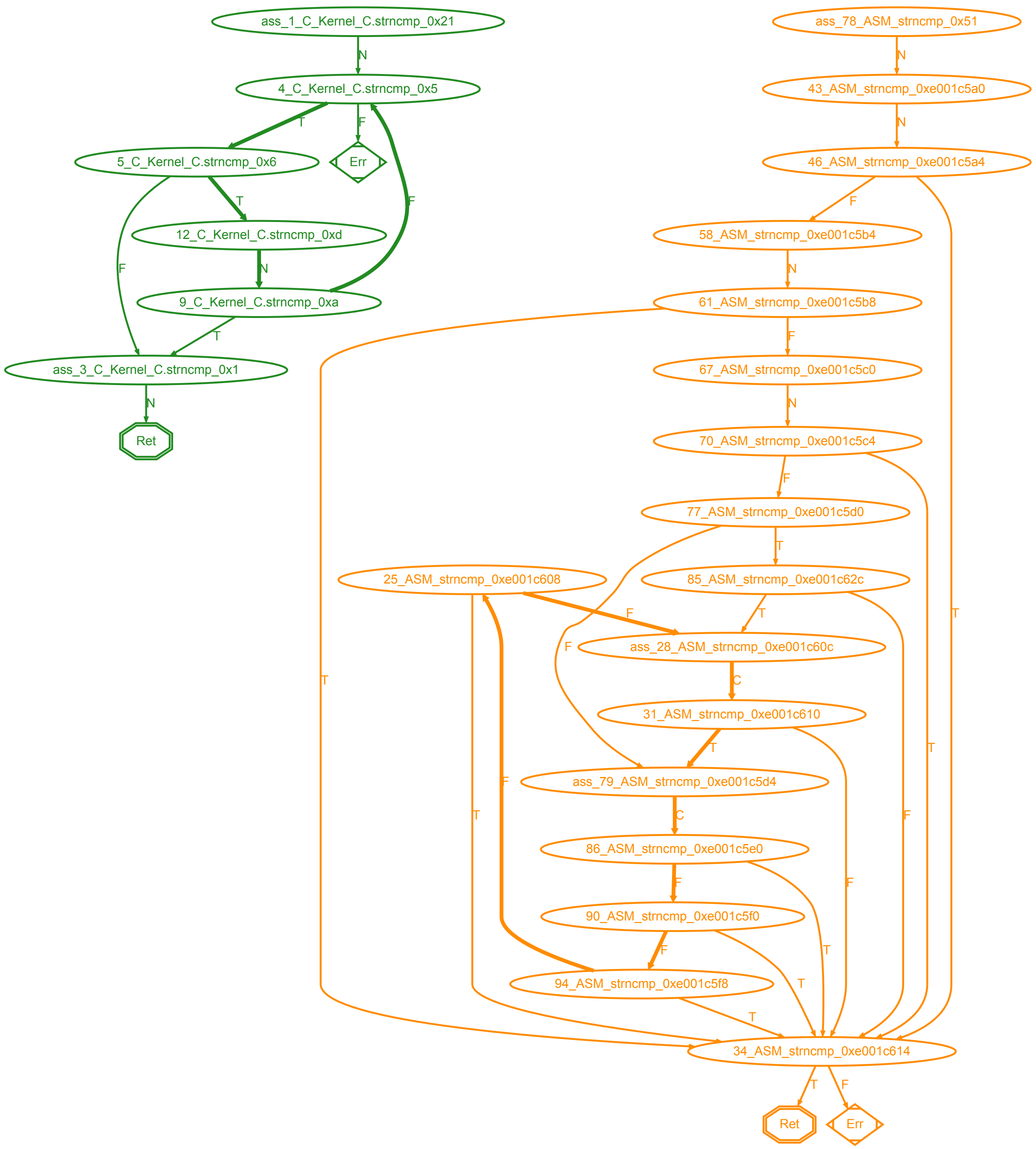


Example 2: string compare

```
int
strncmp(const char* s1, const char* s2, int n)
{
    word_t i;
    int diff;
    for (i = 0; i < n; i++) {
        diff = ((unsigned char*)s1)[i]
                - ((unsigned char*)s2)[i];
        if (diff != 0 || s1[i] == '\0') {
            return diff;
        }
    }
    return 0;
}
```

```
e001c598 <strncmp>:
e001c598: e3520000  cmp r2, #0
e001c59c: e92d0030  push {r4, r5}
e001c5a0: 01a00002  moveq r0, r2
e001c5a4: 0a00001a  beq e001c614 <strncmp+0x7c>
e001c5a8: e5d03000  ldrb r3, [r0]
e001c5ac: e5d15000  ldrb r5, [r1]
e001c5b0: e0535005  subs r5, r3, r5
e001c5b4: 11a00005  movne r0, r5
e001c5b8: 1a000015  bne e001c614 <strncmp+0x7c>
e001c5bc: e3530000  cmp r3, #0
e001c5c0: 01a00003  moveq r0, r3
e001c5c4: 0a000012  beq e001c614 <strncmp+0x7c>
e001c5c8: e3120001  tst r2, #1
e001c5cc: e1a03000  mov r3, r0
e001c5d0: 0a000011  beq e001c61c <strncmp+0x84>
e001c5d4: e2850001  add r0, r5, #1
e001c5d8: e2855002  add r5, r5, #2
e001c5dc: e1520000  cmp r2, r0
e001c5e0: 9a000013  bls e001c634 <strncmp+0x9c>
e001c5e4: e5f3c001  ldrb ip, [r3, #1]!
e001c5e8: e5f14001  ldrb r4, [r1, #1]!
e001c5ec: e05c0004  subs r0, ip, r4
e001c5f0: 1a000007  bne e001c614 <strncmp+0x7c>
e001c5f4: e35c0000  cmp ip, #0
e001c5f8: 0a000005  beq e001c614 <strncmp+0x7c>
e001c5fc: e5f3c001  ldrb ip, [r3, #1]!
e001c600: e5f14001  ldrb r4, [r1, #1]!
e001c604: e05c0004  subs r0, ip, r4
e001c608: 1a000001  bne e001c614 <strncmp+0x7c>
e001c60c: e35c0000  cmp ip, #0
e001c610: 1affffef  bne e001c5d4 <strncmp+0x3c>
e001c614: e8bd0030  pop {r4, r5}
e001c618: e12fff1e  bx lr
e001c61c: e5f15001  ldrb r5, [r1, #1]!
e001c620: e5f3c001  ldrb ip, [r3, #1]!
e001c624: e05c0005  subs r0, ip, r5
e001c628: e3a05001  mov r5, #1
e001c62c: 0afffff6  beq e001c60c <strncmp+0x74>
e001c630: eafffff7  b e001c614 <strncmp+0x7c>
e001c634: e3a00000  mov r0, #0
e001c638: eafffff5  b e001c614 <strncmp+0x7c>
```

Example 2: string compare (cont)



Example 2: string compare

i < n might not be used for the first few iterations in generated code

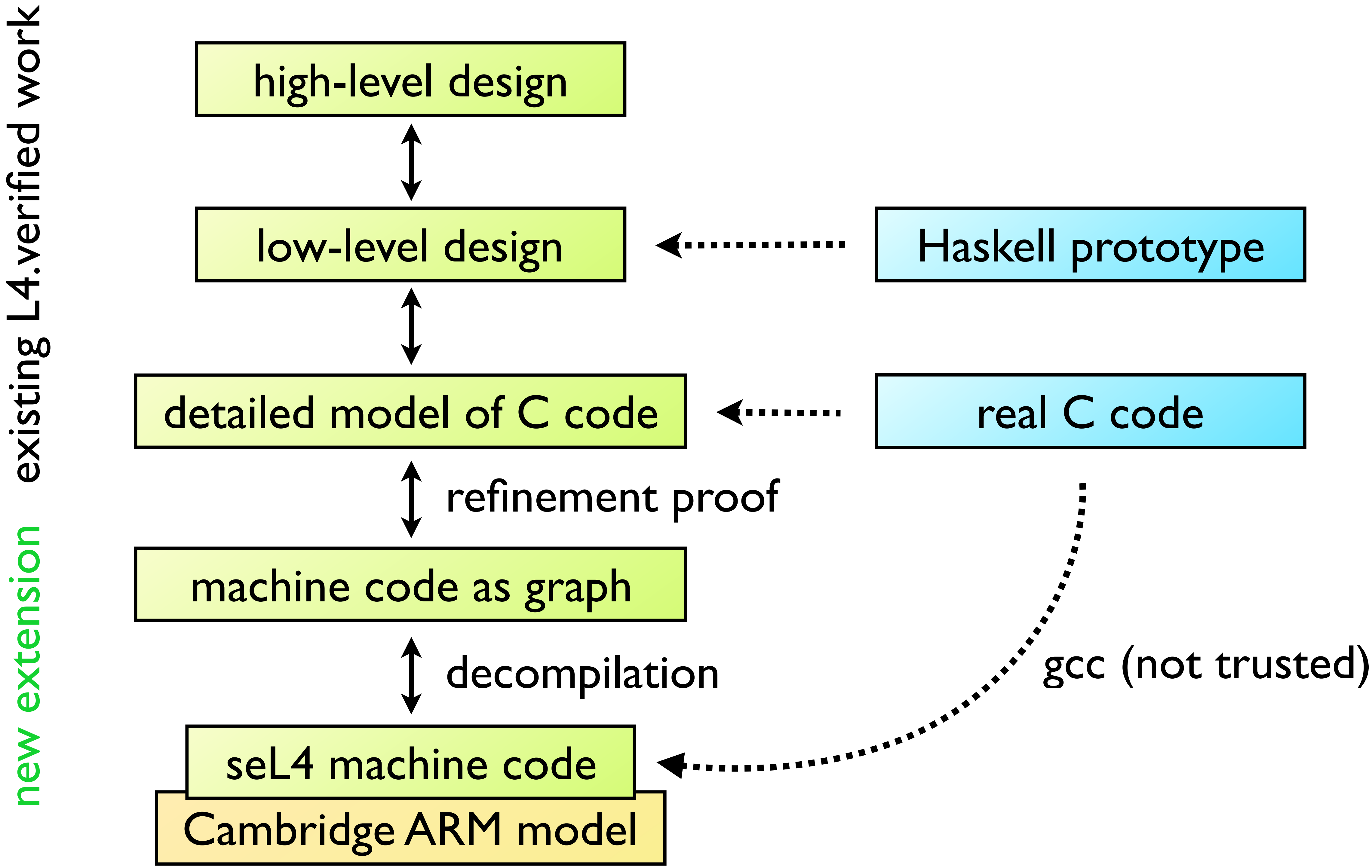
```
int
strncmp(const char* s1, const char* s2, int n)
{
    word_t i;
    int diff;
    for (i = 0; i < n; i++) {
        diff = ((unsigned char*)s1)[i]
               - ((unsigned char*)s2)[i];
        if (diff != 0 || s1[i] == '\0') {
            return diff;
        }
    }
    return 0;
}
```

can waste hours of CPU time…

Complications:

1. structure is different (complex induction required, case split on parity)
2. usual strategy of looking for coincidences doesn't work (because values of i, s1 and s2 might not be there)
3. compiler optimises linear variables and might track a combination of them (e.g. s1+i+4)
4. ignoring linear variables doesn't work because memory stays the same

Big picture (again)



Summary

Translation validation
can be used to formally check
the output of GCC -O1 and
(very nearly) -O2.

Validates the C semantics as
used for the seL4 proofs.

Questions?