Machine-code verification

Experience of tackling medium-sized case studies using decompilation into logic

ACL2'14, Vienna

Magnus Myreen

Currently at UNIVERSITY OF CAMBRIDGE but soon at CHALMERS UNIVERSITY OF TECHNOLOGY .

Why machine code?

Computer systems:

- computer networks
- multi-language implementations
- source code (Java, Lisp, C etc.)
- bytecode or LLVM
- machine code
- hardware
- electric charge

Proofs only target a model of reality.

Why machine code?

Computer systems:

- computer networks
- multi-language implementations
- source code (Java, Lisp, C etc.)
- bytecode or LLVM
- machine code
- hardware
- electric charge

Proofs only target a model of reality.

(Tests run on the ‘real thing’, but are not as insightful.)

Why machine code?

Computer systems:

computer networks

multi-language implementations

source code (Java, Lisp, C etc.)

bytecode or LLVM

machine code

hardware

electric charge

a (mostly) well specified interface

- extensive manuals
- least ambiguous(?), cf. C semantics

Proofs only target a model of reality.

(Tests run on the 'real thing', but are not as insightful.)

Why machine code?

Computer systems:

Ultimately all program verification ought to reach real machine code.

- computer networks
- multi-language implementations
- source code (Java, Lisp, C etc.)
- bytecode or LLVM
- machine code
- hardware
- electric charge

a (mostly) well specified interface

- extensive manuals
- least ambiguous(?), cf. C semantics

Proofs only target a model of reality.

(Tests run on the 'real thing', but are not as insightful.)

Machine code

Machine code,

```
E1510002 B0422001 C0411002 01AFFFFFB
```

is impossible to read, write or maintain manually.

Machine code

Machine code,

`E1510002 B0422001 C0411002 01AFFFFB`

is impossible to read, write or maintain manually.

However, for theorem-prover-based formal verification:

machine code is clean and tractable!

Machine code

Machine code,

`E1510002 B0422001 C0411002 01AFFFFB`

is impossible to read, write or maintain manually.

However, for theorem-prover-based formal verification:

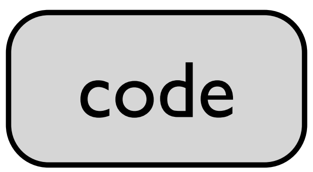
machine code is clean and tractable!

Reason:

- all types are concrete: `word32`, `word8`, bool.
- state consists of a few simple components: a few registers, a memory and some status bits.
- each instruction performs only small well-defined updates.

Challenges of Machine Code

machine code



Challenges of Machine Code

machine code

code

correctness

{P} code {Q}

Challenges of Machine Code

machine code

code

ARM/x86/PowerPC model
(1000...10,000 lines each)

correctness

{P} code {Q}

Challenges of Machine Code

machine code

code

ARM/x86/PowerPC model
(1000...10,000 lines each)

correctness

{P} code {Q}

Challenges:

- several large, detailed models
- unstructured code
- very low-level and limited resources

This talk

Part 1: my approach (PhD work)

Part 2: verification of existing code

Part 3: construction of correct code

This talk

Part 1: my approach (PhD work)

Part 2: verification of existing code

Part 3: construction of correct code

This talk

Part 1: my approach (PhD work)

- automation: code to spec
- automation: spec to code

Part 2: verification of existing code

Part 3: construction of correct code

This talk

Part 1: my approach (PhD work)

- automation: code to spec
- automation: spec to code

Part 2: verification of existing code

- verification of gcc output for microkernel (7,000 lines of C)

Part 3: construction of correct code

This talk

Part 1: my approach (PhD work)

- automation: code to spec
- automation: spec to code

Part 2: verification of existing code

- verification of gcc output for microkernel (7,000 lines of C)

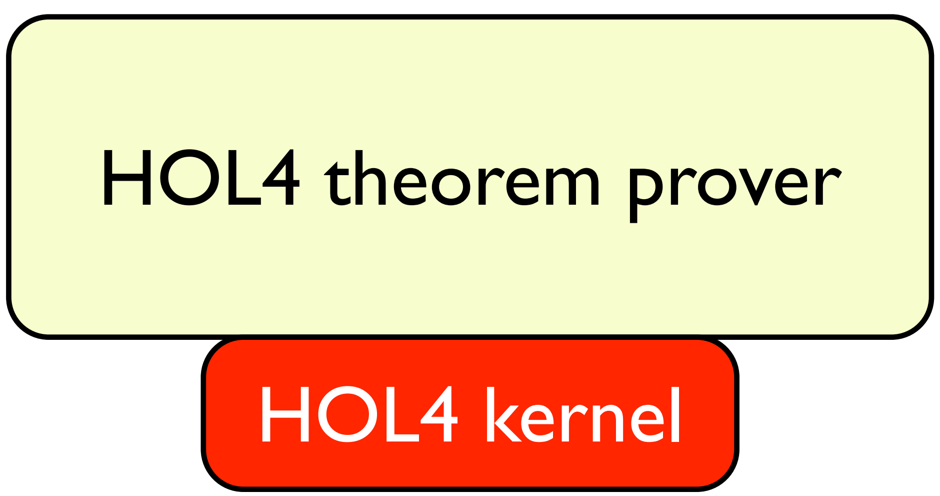
Part 3: construction of correct code

- verified implementation of Lisp that can run Jared Davis' Milawa

HOL: fully-expansive LCF-style prover

The aim is to prove deep functional properties of machine code.

Proofs are performed in HOL4 — a fully expansive theorem prover

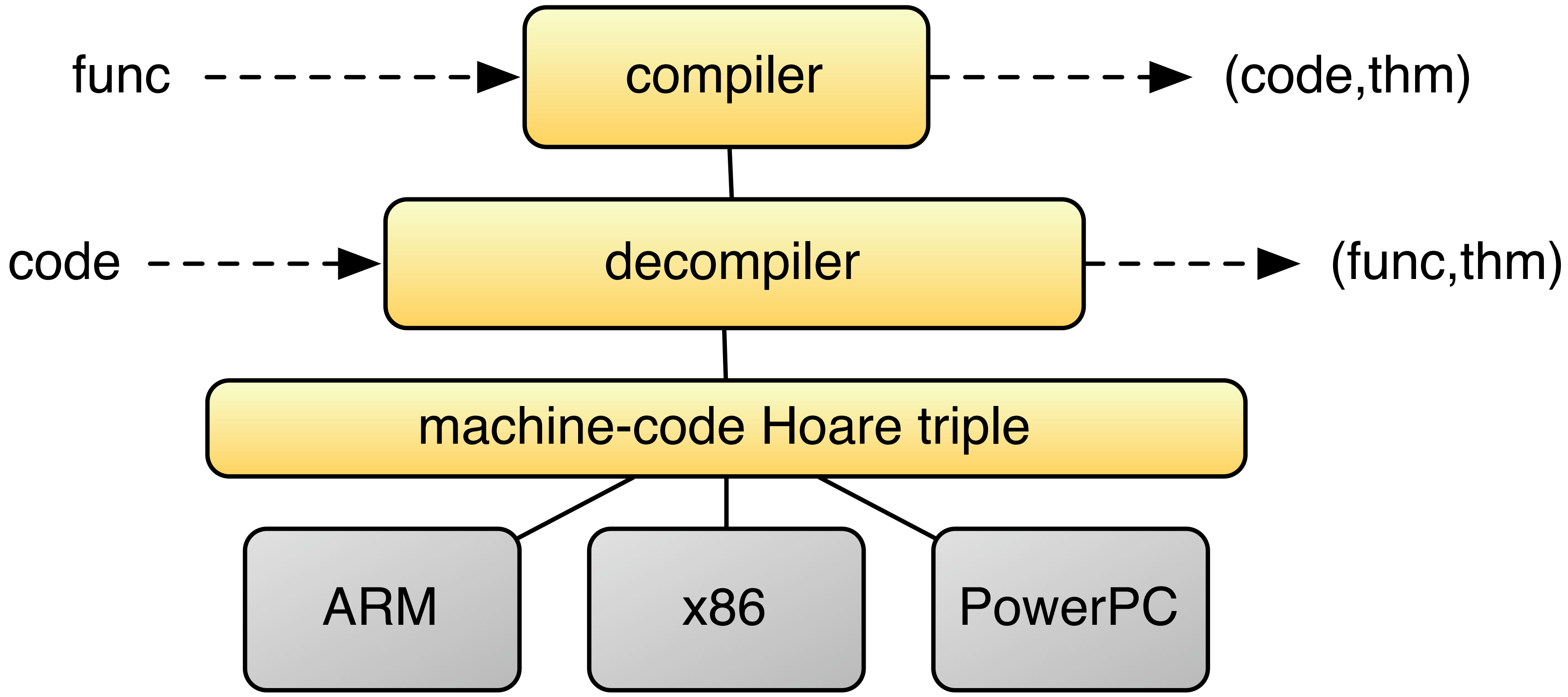


All proofs expand at runtime into primitive inferences in the HOL4 kernel.

The kernel implements the axioms and inference rules of higher-order logic.

Infrastructure

During my PhD, I developed the following infrastructure:



. . . each part will be explained in the next slides.

Models of machine code

Machine models borrowed from work by others:

ARM model, by Fox [TPHOLs'03]

- covers practically all ARM instructions, for old and new ARMs
- still actively being developed

x86 model, by Sarkar et al. [POPL'09]

- covers all addressing modes in 32-bit mode x86
- includes approximately 30 instructions

PowerPC model, originally from Leroy [POPL'06]

- manual translation (Coq $\longrightarrow$ HOL4) of Leroy's PowerPC model
- instruction decoder added

Hoare triples

Each model can be evaluated, e.g. ARM instruction `add r0,r0,r0` is described by theorem:

```
|- (ARM_READ_MEM ((31 >< 2) (ARM_READ_REG 15w state)) state =
    0xE0800000w) ∧ ¬state.undefined ⇒
   (NEXT_ARM_MMU cp state =
     ARM_WRITE_REG 15w (ARM_READ_REG 15w state + 4w)
     (ARM_WRITE_REG 0w
      (ARM_READ_REG 0w state + ARM_READ_REG 0w state) state))
```

Hoare triples

Each model can be evaluated, e.g. ARM instruction `add r0,r0,r0` is described by theorem:

```
|- (ARM_READ_MEM ((31 >< 2) (ARM_READ_REG 15w state)) state =
    0xE0800000w) ∧ ¬state.undefined ⇒
   (NEXT_ARM_MMU cp state =
     ARM_WRITE_REG 15w (ARM_READ_REG 15w state + 4w)
     (ARM_WRITE_REG 0w
      (ARM_READ_REG 0w state + ARM_READ_REG 0w state) state))
```

As a total-correctness machine-code Hoare triple:

```
|- SPEC ARM_MODEL
     (aR 0w x * aPC p)
     {(p,0xE0800000w)}
     (aR 0w (x+x) * aPC (p+4w))
```

Informal syntax for this talk:

$\{\,\mathrm{R0}\ x * \mathrm{PC}\ p\,\}$

p : E0800000

$\{\,\mathrm{R0}\ (x{+}x) * \mathrm{PC}\ (p{+}4)\,\}$

Definition of Hoare triple

$$\{p\}\ c\ \{q\} \quad\iff\quad \begin{array}{l} \forall s\ r.\ \ (p * r * \mathsf{code}\ c)\ s \implies \\ \qquad \exists n.\ (q * r * \mathsf{code}\ c)\ (\mathsf{run}\ n\ s) \end{array}$$

Definition of Hoare triple

frame

$$\{p\}\ c\ \{q\} \quad \iff \quad \forall s\ r.\ \ (p * r * \mathsf{code}\ c)\ s \implies$$
$$\exists n.\ (q * r * \mathsf{code}\ c)\ (\mathsf{run}\ n\ s)$$

Definition of Hoare triple

frame   code separate

$$\{p\}\ c\ \{q\} \quad \iff \quad \forall s\ r.\ \ (p * r * \mathsf{code}\ c)\ s \implies$$
$$\exists n.\ (q * r * \mathsf{code}\ c)\ (\mathsf{run}\ n\ s)$$

Definition of Hoare triple

frame

code separate

$$\{p\}\ c\ \{q\} \quad \Longleftrightarrow \quad \forall s\ r.\ \ (p * r * \textsf{code}\ c)\ s \implies$$
$$\exists n.\ (q * r * \textsf{code}\ c)\ (\textsf{run}\ n\ s)$$

total correctness

Definition of Hoare triple

frame

code separate

$$\{p\}\ c\ \{q\} \quad \iff \quad \begin{aligned}&\forall s\ r.\ \ (p * r * \mathsf{code}\ c)\ s \implies \\ &\qquad \exists n.\ (q * r * \mathsf{code}\ c)\ (\mathsf{run}\ n\ s)\end{aligned}$$

total correctness

machine code sem.

Definition of Hoare triple

separating conjunction

frame

code separate

$$\{p\}\ c\ \{q\} \quad \Longleftrightarrow \quad \forall s\ r.\ \ (p * r * \mathsf{code}\ c)\ s \implies$$
$$\exists n.\ (q * r * \mathsf{code}\ c)\ (\mathsf{run}\ n\ s)$$

total correctness

machine code sem.

Definition of Hoare triple

separating conjunction

frame　　code separate

$$\{p\}\ c\ \{q\} \quad \Longleftrightarrow \quad \forall s\ r.\ \ (p * r * \mathsf{code}\ c)\ s \implies$$
$$\exists n.\ (q * r * \mathsf{code}\ c)\ (\mathsf{run}\ n\ s)$$

total correctness　　machine code sem.

Program logic can be used directly for verification.

Definition of Hoare triple

separating conjunction

frame   code separate

$$\{p\}\ c\ \{q\} \quad\iff\quad \forall s\ r.\ \ (p * r * \mathsf{code}\ c)\ s \implies$$
$$\exists n.\ (q * r * \mathsf{code}\ c)\ (\mathsf{run}\ n\ s)$$

total correctness   machine code sem.

Program logic can be used directly for verification.
But direct reasoning in this embedded logic is tiresome.

Decompiler

Decompiler automates Hoare triple reasoning.

Decompiler

Decompiler automates Hoare triple reasoning.

Example: Given some ARM machine code,

```
 0: E3A00000      mov r0, #0
 4: E3510000   L: cmp r1, #0
 8: 12800001      addne r0, r0, #1
12: 15911000      ldrne r1, [r1]
16: 1AFFFFFB      bne L
```

Decompiler

Decompiler automates Hoare triple reasoning.

Example: Given some ARM machine code,

```
 0: E3A00000       mov r0, #0
 4: E3510000    L: cmp r1, #0
 8: 12800001       addne r0, r0, #1
12: 15911000       ldrne r1, [r1]
16: 1AFFFFFB       bne L
```

the decompiler automatically extracts a readable function:

$$
\begin{array}{lcl}
f(r_0, r_1, m) & = & \text{let } r_0 = 0 \text{ in } g(r_0, r_1, m) \\
g(r_0, r_1, m) & = & \text{if } r_1 = 0 \text{ then } (r_0, r_1, m) \text{ else} \\
 & & \quad \text{let } r_0 = r_0{+}1 \text{ in} \\
 & & \quad \text{let } r_1 = m(r_1) \text{ in} \\
 & & \quad \quad g(r_0, r_1, m)
\end{array}
$$

Decompilation, correct?

Decompiler automatically proves a certificate theorem:

$$f_{pre}(r_0, r_1, m) \Rightarrow$$
$$\{\, (\mathsf{R0}, \mathsf{R1}, \mathsf{M}) \text{ is } (r_0, r_1, m) * \mathsf{PC}\ p * \mathsf{S}\,\}$$
p : `E3A00000 E3510000 12800001 15911000 1AFFFFFB`
$$\{\, (\mathsf{R0}, \mathsf{R1}, \mathsf{M}) \text{ is } f(r_0, r_1, m) * \mathsf{PC}\ (p + 20) * \mathsf{S}\,\}$$

which informally reads:

for any initially value (r_0, r_1, m) in reg 0, reg 1 and memory,
the code terminates with $f(r_0, r_1, m)$ in reg 0, reg 1 and memory.

Decompilation verification example

To verify code: prove properties of function f,

$$\begin{array}{lcl} \forall x\, l\, a\, m.\;\; list(l,a,m) & \Rightarrow & f(x,a,m) = (length(l), 0, m) \\ \forall x\, l\, a\, m.\;\; list(l,a,m) & \Rightarrow & f_{pre}(x,a,m) \end{array}$$

since properties of f carry over to machine code via the certificate.

Decompilation verification example

To verify code: prove properties of function f,

$$\forall x\ l\ a\ m.\ \ list(l,a,m) \ \Rightarrow\ f(x,a,m) = (length(l), 0, m)$$
$$\forall x\ l\ a\ m.\ \ list(l,a,m) \ \Rightarrow\ f_{pre}(x,a,m)$$

since properties of f carry over to machine code via the certificate.

Proof reuse: Given similar x86 and PowerPC code:

```
31C085F67405408B36EBF7
```

```
38A000002C14000040820010 7E80A02E38A500014BFFFFF0
```

which decompiles into f' and f'', respectively. Manual proofs above can be reused if $f = f' = f''$.

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.)

Decompiler implementation

Implementation:

- ML program which fully-automatically performs forward proof,
- no heuristics and no dangling proof obligations,
- loops are handled by a special loop rule which introduces tail-recursive functions:

$$tailrec(x) = \text{if } G(x) \text{ then } tailrec(F(x)) \text{ else } D(x)$$

with termination and side-conditions H collected as:

$$pre(x) = (\text{if } G(x) \text{ then } pre(F(x)) \text{ else true}) \wedge H(x)$$

Details in Myreen et al. [FMCAD'08].

Comparison of approaches

```
 0: E3A00000      mov r0, #0
 4: E3510000   L: cmp r1, #0
 8: 12800001      addne r0, r0, #1
12: 15911000      ldrne r1, [r1]
16: 1AFFFFFB      bne L
```

Comparison of approaches

```
 0: E3A00000      mov r0, #0
 4: E3510000   L: cmp r1, #0
 8: 12800001      addne r0, r0, #1
12: 15911000      ldrne r1, [r1]
16: 1AFFFFFB      bne L
```

direct manual proof using definition of instruction set model

Comparison of approaches

```
 0: E3A00000      mov r0, #0
 4: E3510000   L: cmp r1, #0
 8: 12800001      addne r0, r0, #1
12: 15911000      ldrne r1, [r1]
16: 1AFFFFFB      bne L
```

direct manual proof using definition of instruction set model

- tedious and proofs complete tied to model

Comparison of approaches

```
 0: E3A00000      mov r0, #0
 4: E3510000   L: cmp r1, #0
 8: 12800001      addne r0, r0, #1
12: 15911000      ldrne r1, [r1]
16: 1AFFFFFB      bne L
```

direct manual proof using definition of instruction set model

- tedious and proofs complete tied to model

symbolic simulation

Comparison of approaches

```
 0: E3A00000      mov r0, #0
 4: E3510000   L: cmp r1, #0
 8: 12800001      addne r0, r0, #1
12: 15911000      ldrne r1, [r1]
16: 1AFFFFFB      bne L
```

direct manual proof using definition of instruction set model

- tedious and proofs complete tied to model

symbolic simulation

- automatic except at looping points, proofs tied to model

Comparison of approaches

```
 0: E3A00000      mov r0, #0
 4: E3510000   L: cmp r1, #0
 8: 12800001      addne r0, r0, #1
12: 15911000      ldrne r1, [r1]
16: 1AFFFFFB      bne L
```

direct manual proof using definition of instruction set model

- tedious and proofs complete tied to model

symbolic simulation

- automatic except at looping points, proofs tied to model

proof using program logic

Comparison of approaches

```
 0: E3A00000      mov r0, #0
 4: E3510000   L: cmp r1, #0
 8: 12800001      addne r0, r0, #1
12: 15911000      ldrne r1, [r1]
16: 1AFFFFFB      bne L
```

direct manual proof using definition of instruction set model

- tedious and proofs complete tied to model

symbolic simulation

- automatic except at looping points, proofs tied to model

proof using program logic

- some reusable proofs, but tedious

Comparison of approaches

```
 0: E3A00000       mov r0, #0
 4: E3510000    L: cmp r1, #0
 8: 12800001       addne r0, r0, #1
12: 15911000       ldrne r1, [r1]
16: 1AFFFFFB       bne L
```

direct manual proof using definition of instruction set model

- tedious and proofs complete tied to model

symbolic simulation

- automatic except at looping points, proofs tied to model

proof using program logic

- some reusable proofs, but tedious

verification condition generation

Comparison of approaches

```
 0: E3A00000      mov r0, #0
 4: E3510000   L: cmp r1, #0
 8: 12800001      addne r0, r0, #1
12: 15911000      ldrne r1, [r1]
16: 1AFFFFFB      bne L
```

direct manual proof using definition of instruction set model

- tedious and proofs complete tied to model

symbolic simulation

- automatic except at looping points, proofs tied to model

proof using program logic

- some reusable proofs, but tedious

verification condition generation

- largely automatic, but requires annotating the machine code(!)

Comparison of approaches

```
 0: E3A00000      mov r0, #0
 4: E3510000   L: cmp r1, #0
 8: 12800001      addne r0, r0, #1
12: 15911000      ldrne r1, [r1]
16: 1AFFFFFB      bne L
```

direct manual proof using definition of instruction set model

- tedious and proofs complete tied to model

symbolic simulation

- automatic except at looping points, proofs tied to model

proof using program logic

- some reusable proofs, but tedious

verification condition generation

- largely automatic, but requires annotating the machine code(!)

decompilation into logic

Comparison of approaches

```
 0: E3A00000      mov r0, #0
 4: E3510000   L: cmp r1, #0
 8: 12800001      addne r0, r0, #1
12: 15911000      ldrne r1, [r1]
16: 1AFFFFFB      bne L
```

direct manual proof using definition of instruction set model

- tedious and proofs complete tied to model

symbolic simulation

- automatic except at looping points, proofs tied to model

proof using program logic

- some reusable proofs, but tedious

verification condition generation

- largely automatic, but requires annotating the machine code(!)

decompilation into logic

- model-specific part is automatic, does not req. annotations

Comparison of approaches

```
 0: E3A00000      mov r0, #0
 4: E3510000   L: cmp r1, #0
 8: 12800001      addne r0, r0, #1
12: 15911000      ldrne r1, [r1]
16: 1AFFFFFB      bne L
```

direct manual proof using definition of instruction set model

- tedious and proofs complete tied to model

symbolic simulation

- automatic except at looping points, proofs tied to model

proof using program logic

- some reusable proofs, but tedious

verification condition generation

- largely automatic, but requires annotating the machine code(!)

decompilation into logic

- model-specific part is automatic, does not req. annotations
- can implement *proof-producing compilation* (next slide)

Comparison of approaches

```
 0: E3A00000       mov r0, #0
 4: E3510000    L: cmp r1, #0
 8: 12800001       addne r0, r0, #1
12: 15911000       ldrne r1, [r1]
16: 1AFFFFFB       bne L
```

direct manual proof using definition of instruction set model

- tedious and proofs complete tied to model

symbolic simulation

- automatic except at looping points, proofs tied to model

pro

-

veri

- de(!)

> decompilation into logic
>
> =
>
> symbolic simulation + support for loops (tail-rec.),
>
> done over a program logic (not machine model)

decompilation into logic

- model-specific part is automatic, does not req. annotations
- can implement *proof-producing compilation* (next slide)

Proof-producing compilation

Synthesis often more practical. Given function f,

$$f(r_1) \;=\; \text{if } r_1 < 10 \text{ then } r_1 \text{ else let } r_1 = r_1 - 10 \text{ in } f(r_1)$$

our *compiler* generates ARM machine code:

```
E351000A      L:  cmp r1,#10
2241100A          subcs r1,r1,#10
2AFFFFFC          bcs L
```

and automatically proves a certificate HOL theorem:

$$\begin{array}{l} \vdash \;\{\,\mathsf{R1}\ r_1 * \mathsf{PC}\ p * \mathsf{s}\,\} \\ \quad\; p : \mathtt{E351000A\ 2241100A\ 2AFFFFFC} \\ \quad\; \{\,\mathsf{R1}\ f(r_1) * \mathsf{PC}\ (p{+}12) * \mathsf{s}\,\} \end{array}$$

Compilation, example cont.

One can prove properties of f since it lives inside HOL:

$$\vdash \; \forall x.\ f(x) = x \bmod 10$$

Properties proved of f translate to properties of the machine code:

$$\vdash \; \{\mathrm{R1}\ r_1 * \mathrm{PC}\ p * \mathrm{s}\}$$
$$p : \texttt{E351000A 2241100A 2AFFFFFC}$$
$$\{\mathrm{R1}\ (r_1 \bmod 10) * \mathrm{PC}\ (p{+}12) * \mathrm{s}\}$$

Additional feature: the compiler can use the above theorem to extend its input language with: let $r_1 = r_1$ mod 10 in _

Implementation

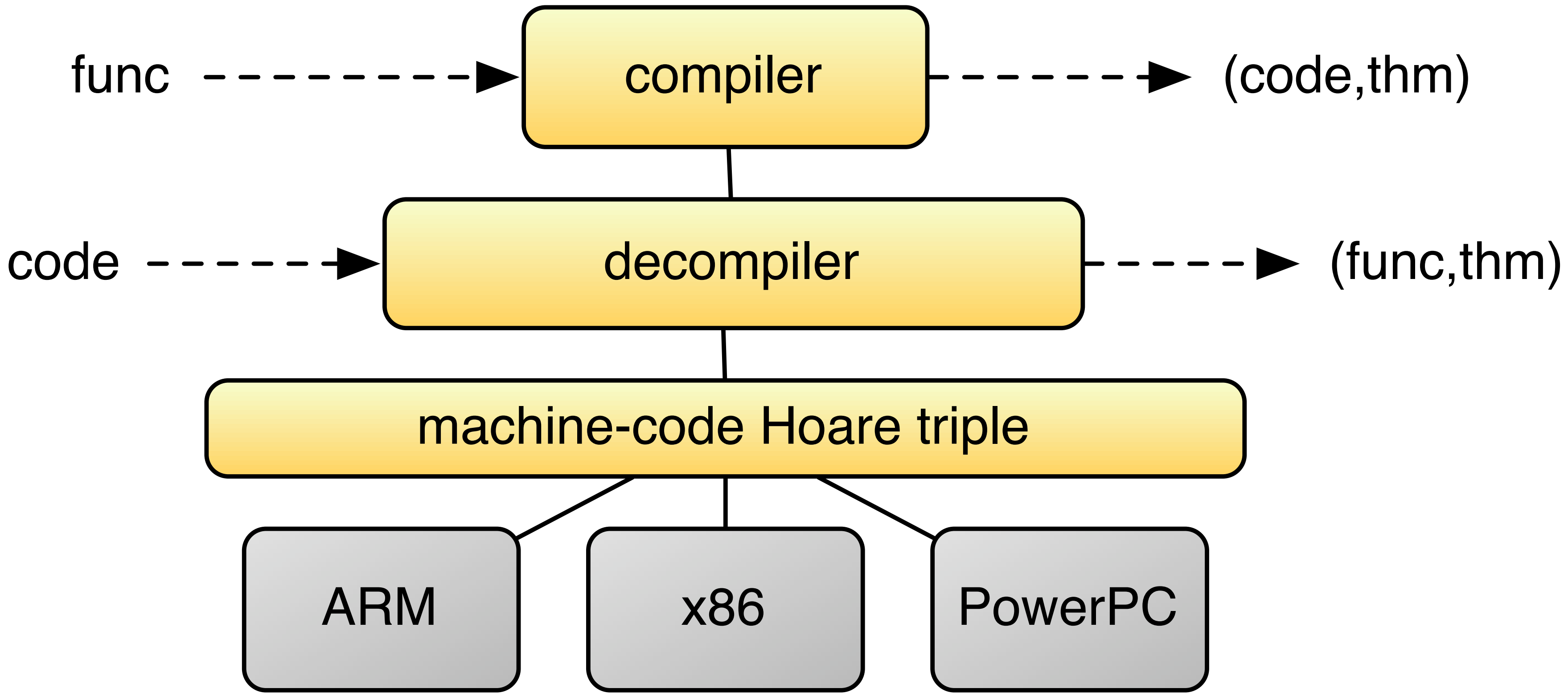
To compile function f:

1. generate, without proof, code from input f;
2. decompile, with proof, a function f' from generated code;
3. prove $f = f'$.

Features:

- code generation completely separate from proof
- supports many light-weight optimisations without any additional proof burden: instruction reordering, conditional execution, dead-code elimination, duplicate-tail elimination, ...
- allows for significant user-defined extensions

Infrastructure (again)



.

This talk

Part 1:

- automation: code to spec
- automation: spec to code

Part 2: verification of existing code

- verification of gcc output for microkernel (7,000 lines of C)

Part 3:

- verified
  that can run Jared Davis'

L4.verified

seL4 = a formally verified general-purpose microkernel

(Work by Gerwin Klein's team at NICTA, Australia)

L4.verified

seL4 = a formally verified general-purpose microkernel

about 7,000 lines of C code and assembly

(Work by Gerwin Klein's team at NICTA, Australia)

L4.verified

seL4 = a formally verified general-purpose microkernel

about 7,000 lines of C code and assembly

200,000 lines of Isabelle/HOL proofs

(Work by Gerwin Klein's team at NICTA, Australia)

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.

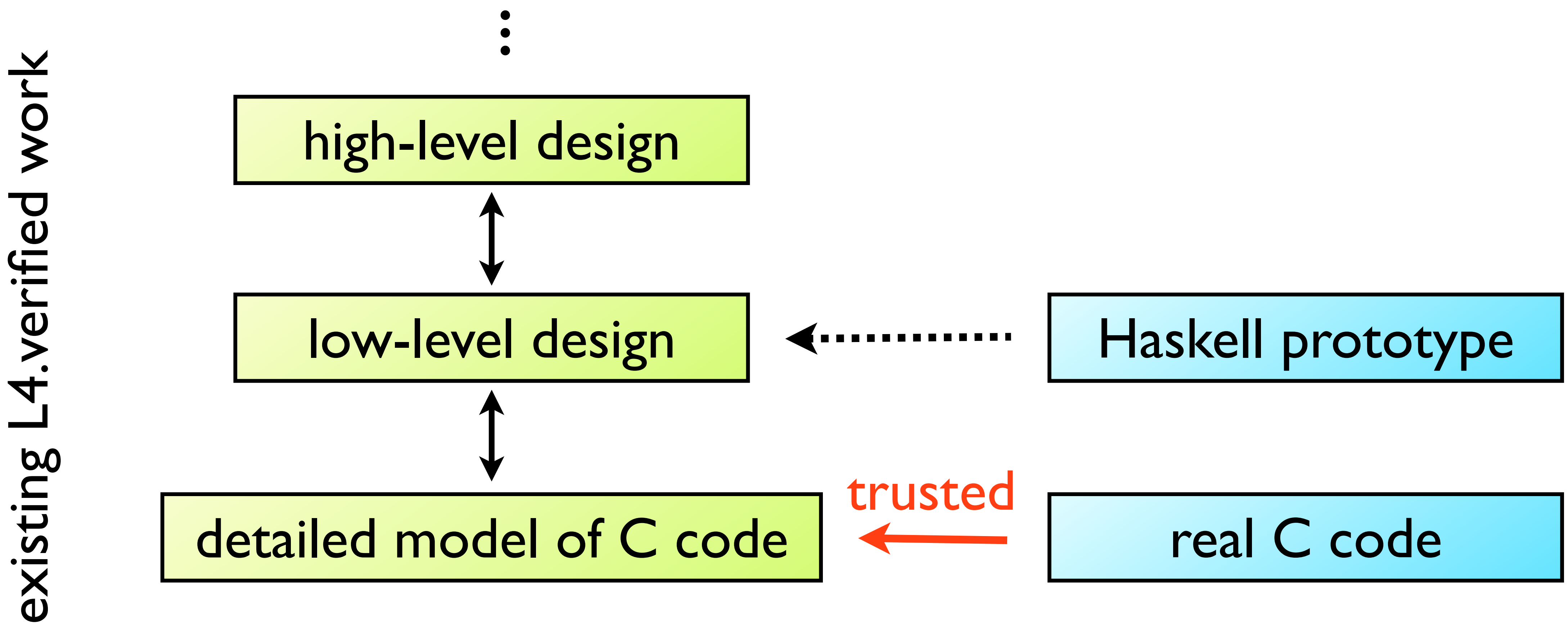
Assumptions

L4.verified project assumes correctness of:

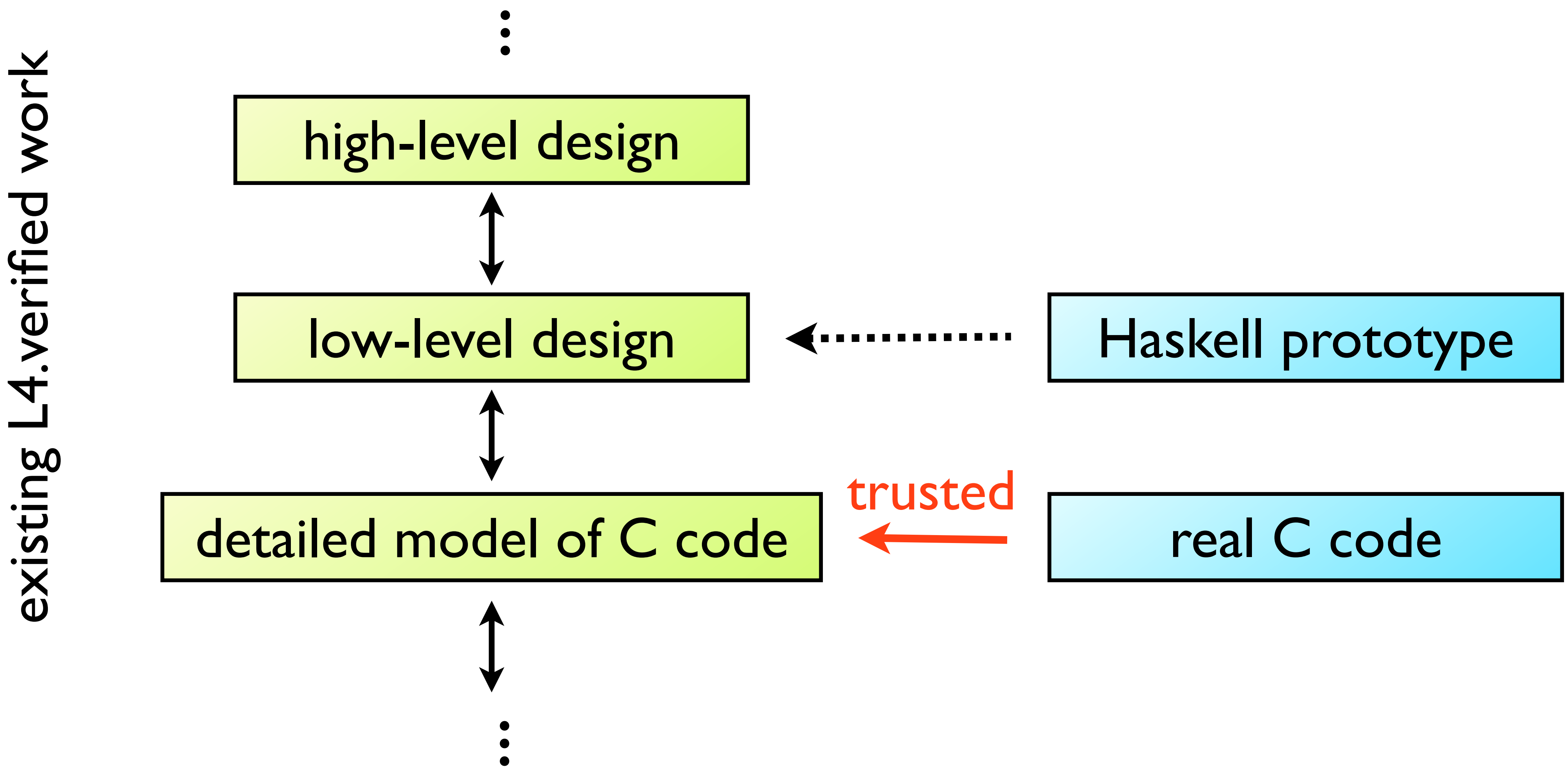
- ~~C compiler (gcc)~~
- inline assembly (?)
- hardware
- hardware management
- boot code (?)
- virtual memory
- Cambridge ARM model

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

Aim: extend downwards

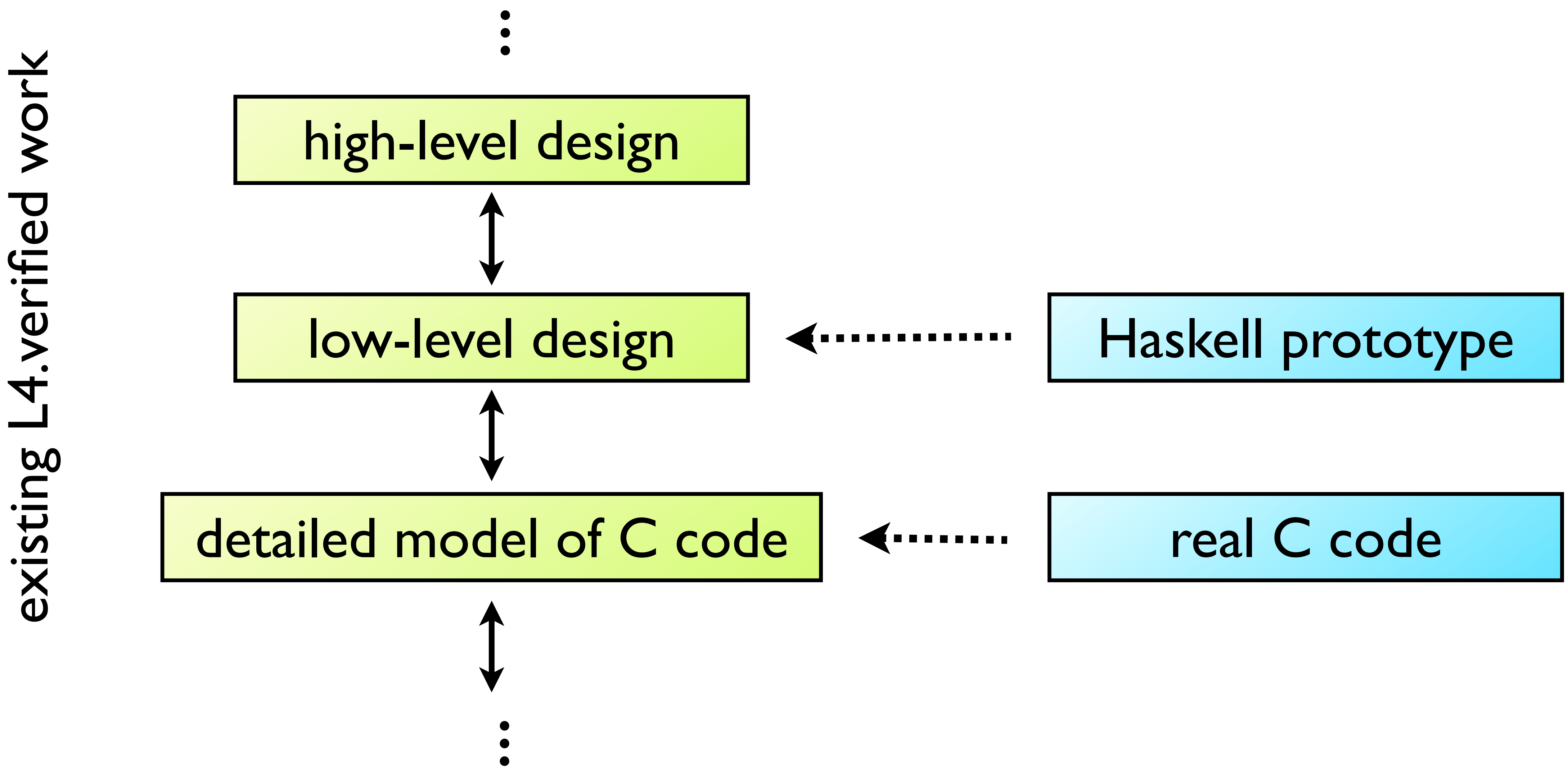


Aim: extend downwards



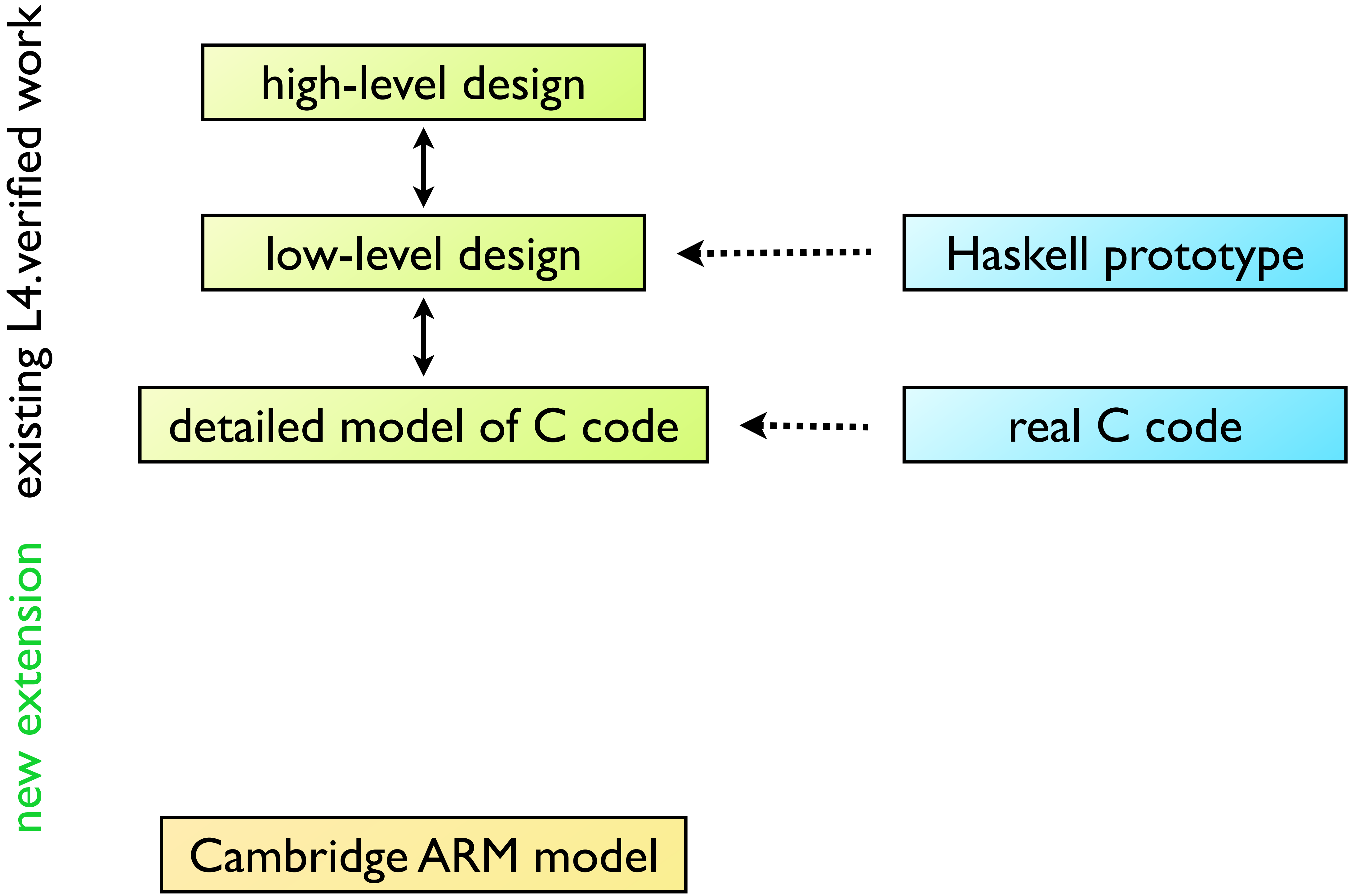
Aim: remove need to trust C compiler and C semantics

Aim: extend downwards

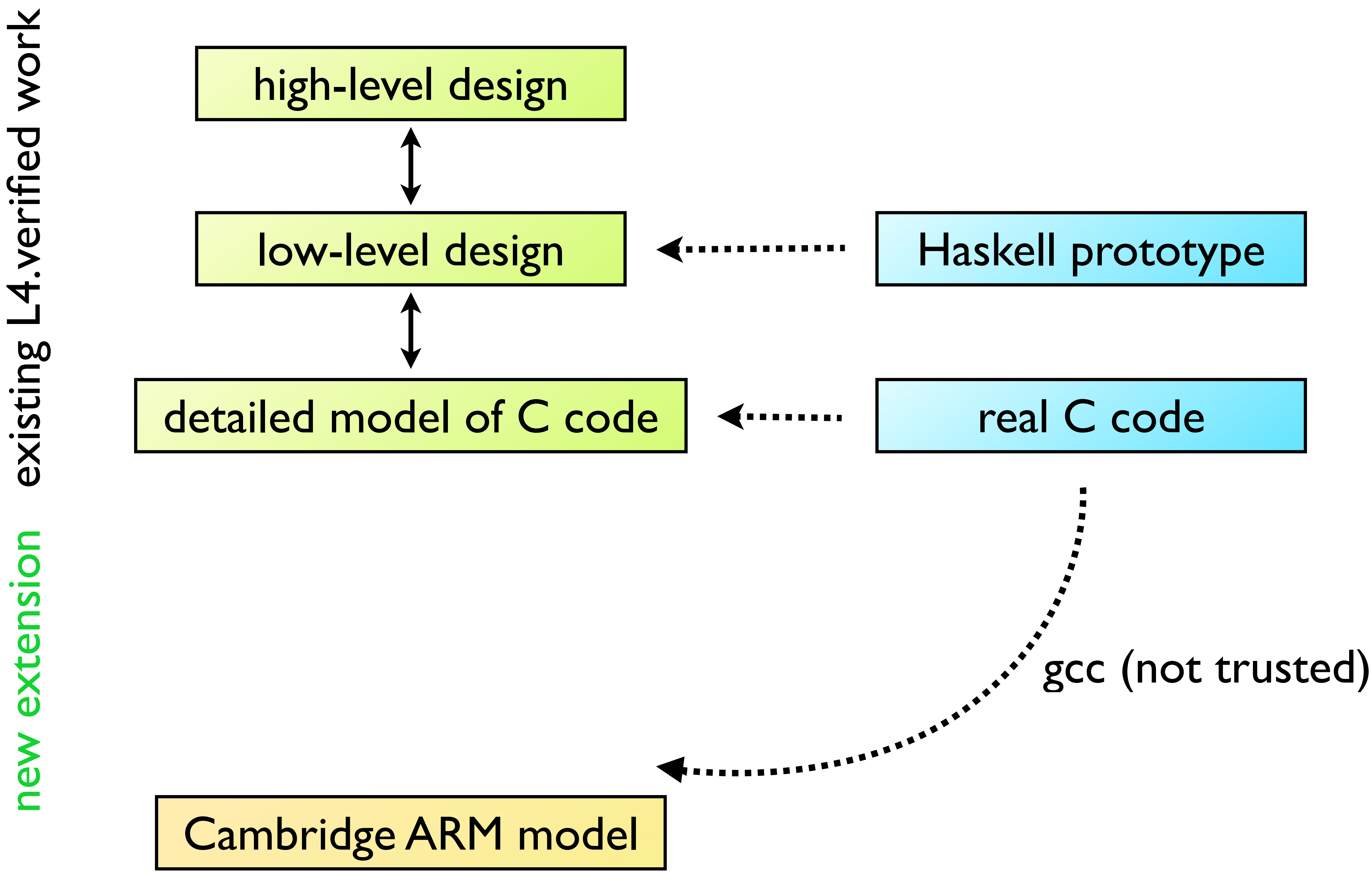


Aim: remove need to trust C compiler and C semantics

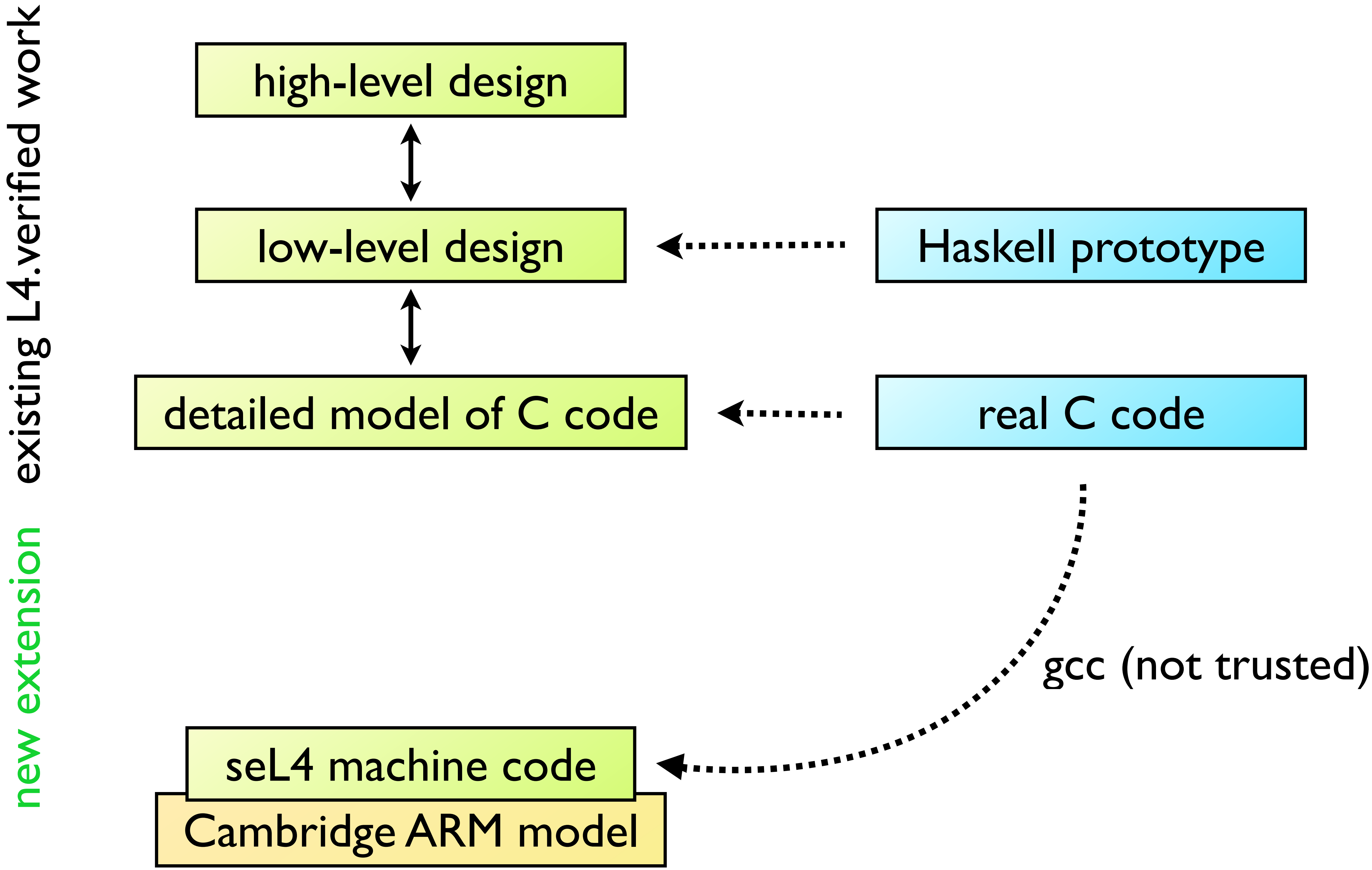
Using Cambridge ARM model



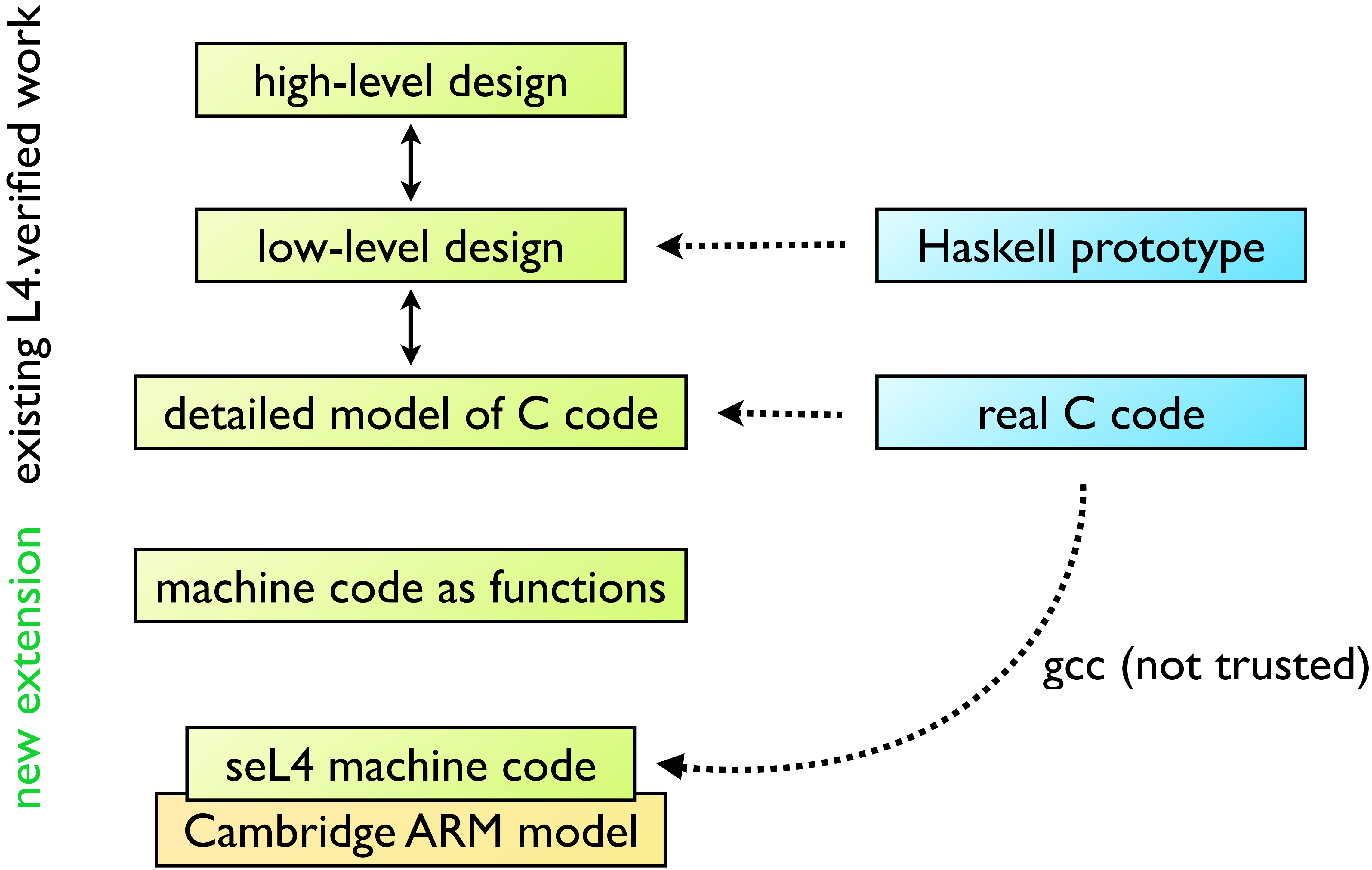
Using Cambridge ARM model



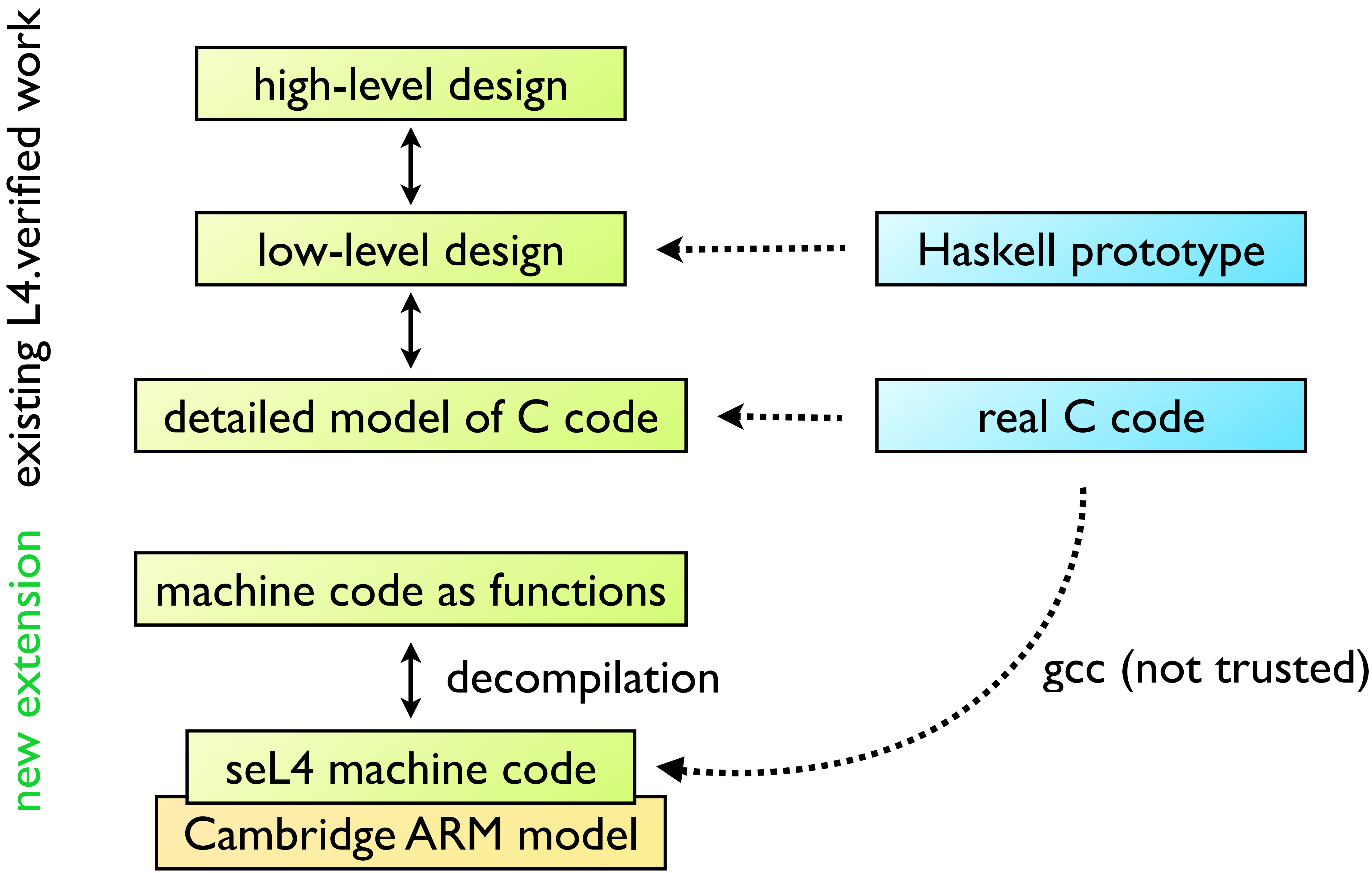
Using Cambridge ARM model



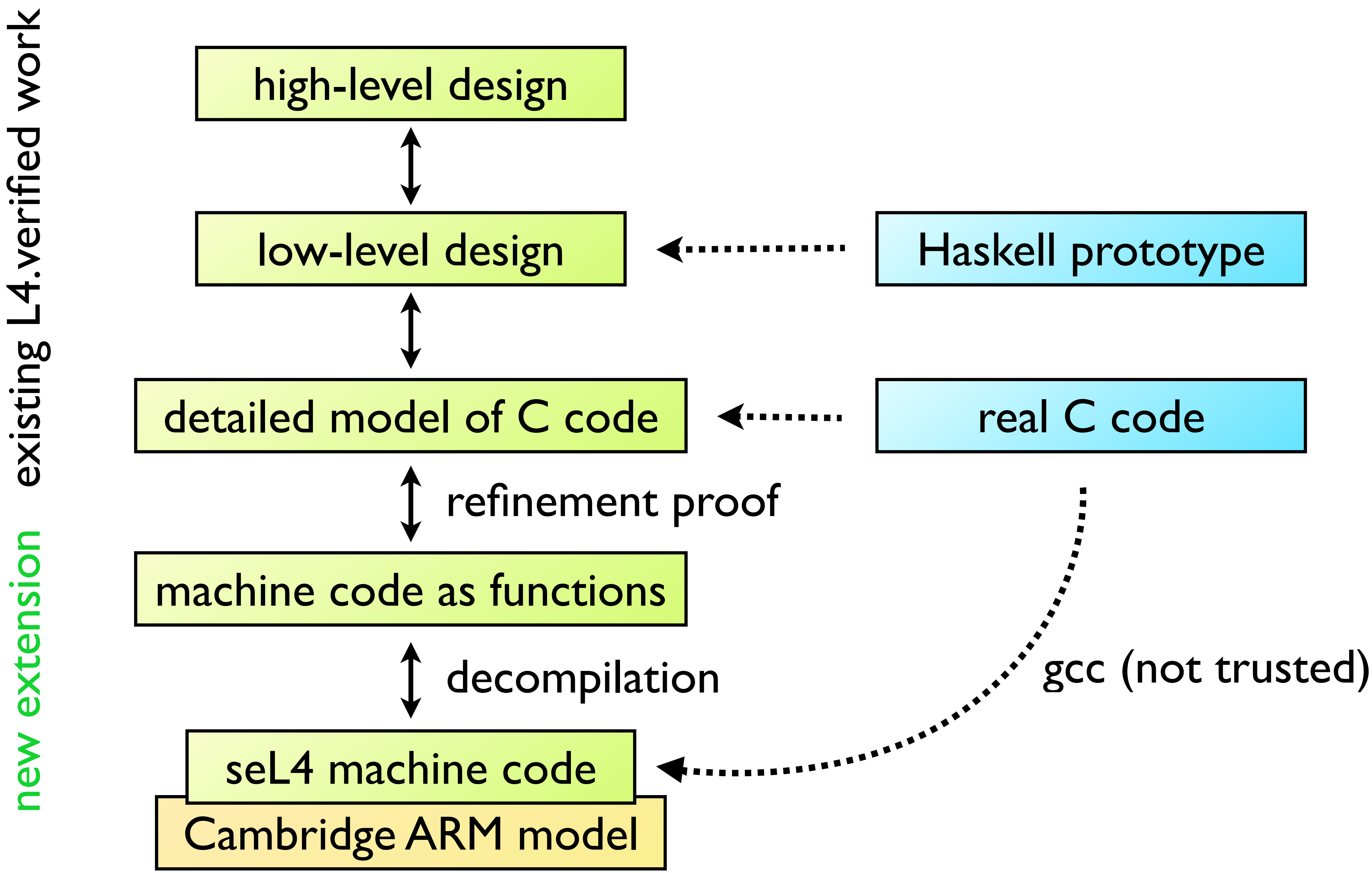
Using Cambridge ARM model



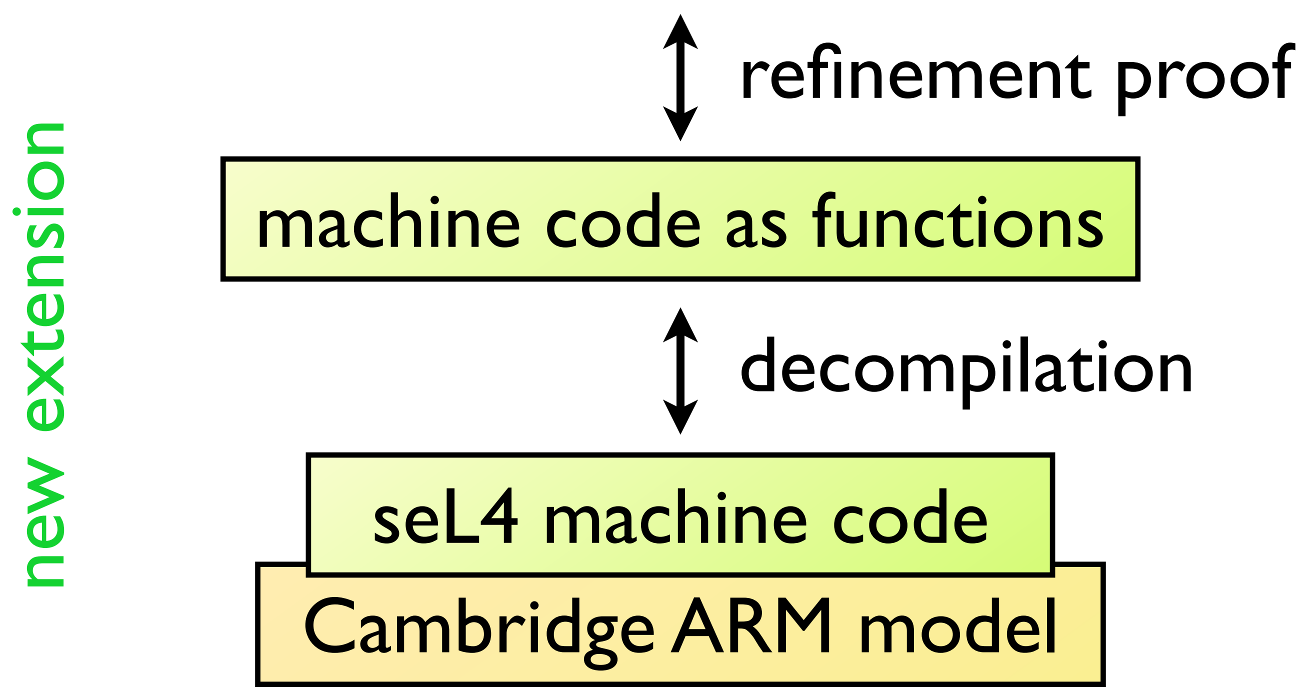
Using Cambridge ARM model



Using Cambridge ARM model

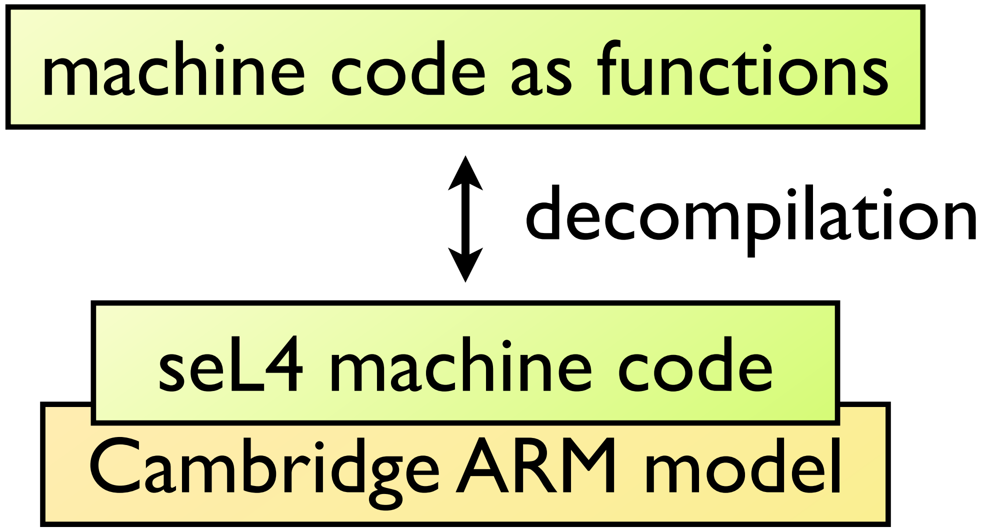


Approach



- decompilation by me
- refinement proof by Thomas Sewell (NICTA)

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:

```c
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
```

decompilation

return instruction

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: *

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;
}
```

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

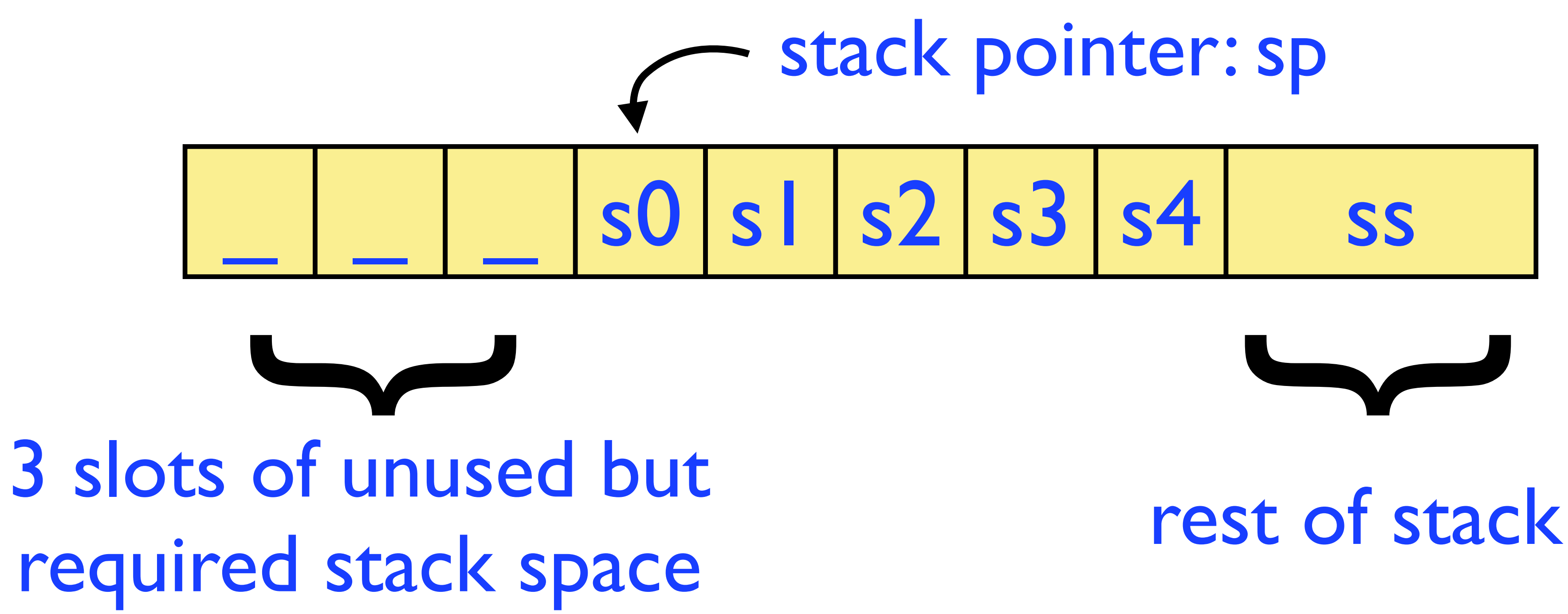
gcc

```
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.

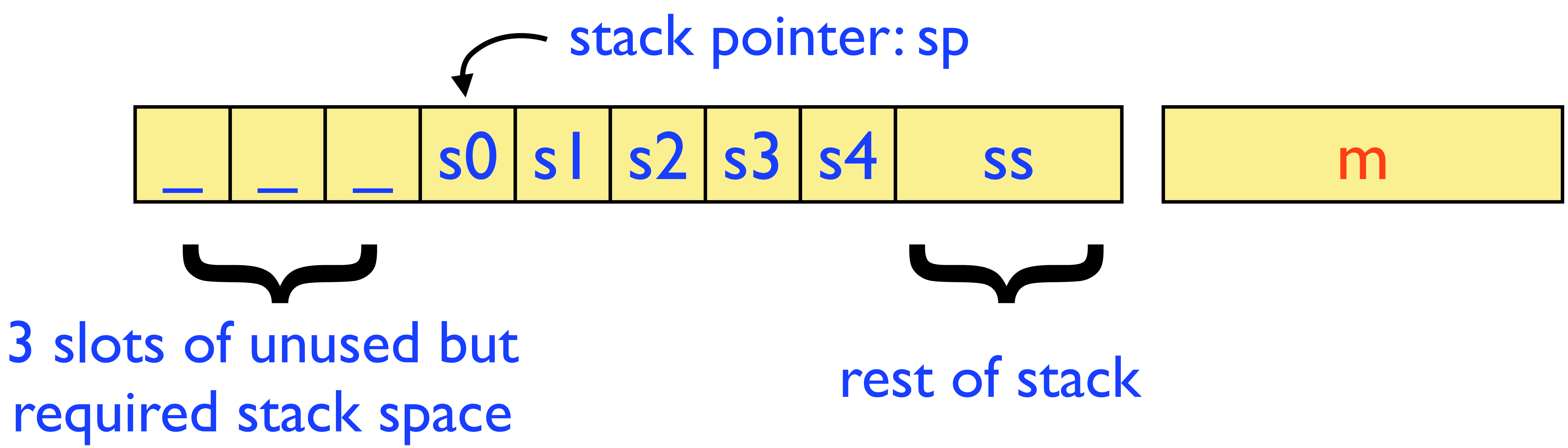
Solution

Use separation-logic inspired approach



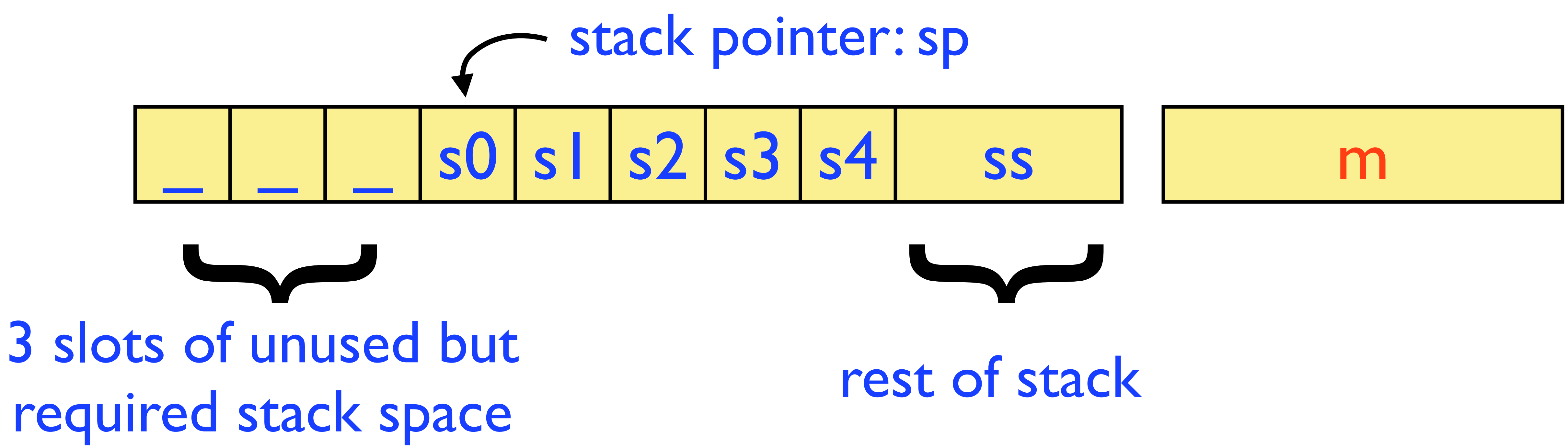
Solution

Use separation-logic inspired approach



Solution

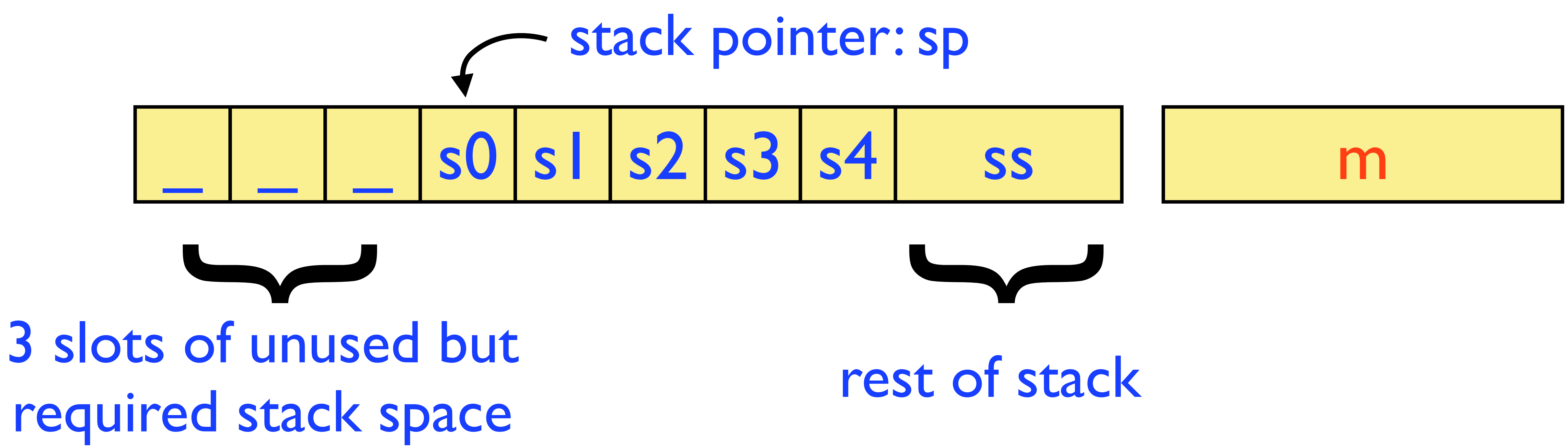
Use separation-logic inspired approach



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

Solution

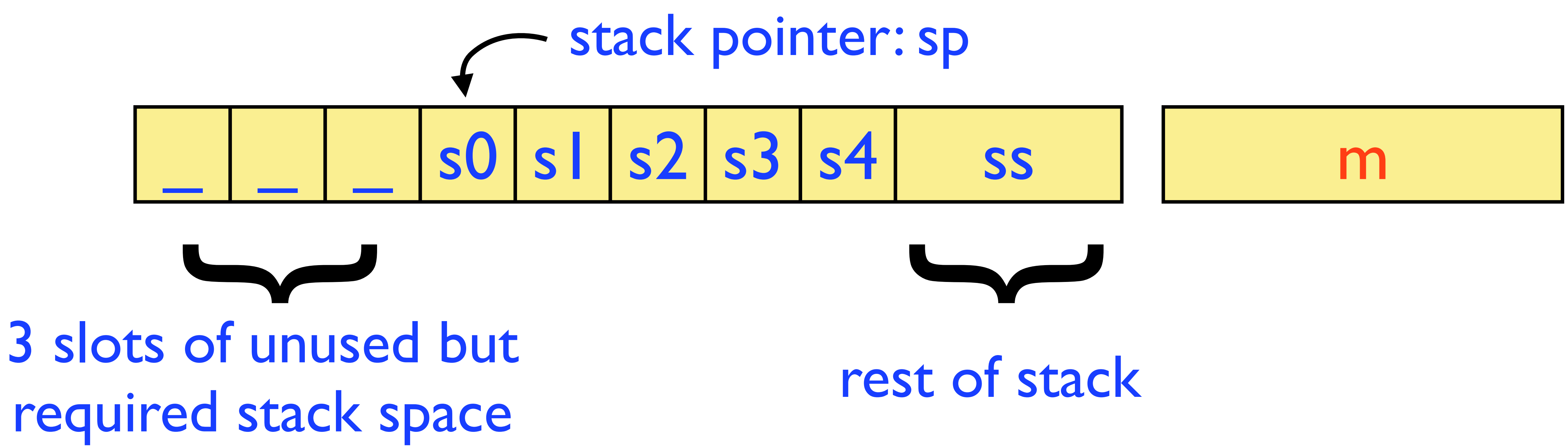
Use separation-logic inspired approach



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

Solution

Use separation-logic inspired approach

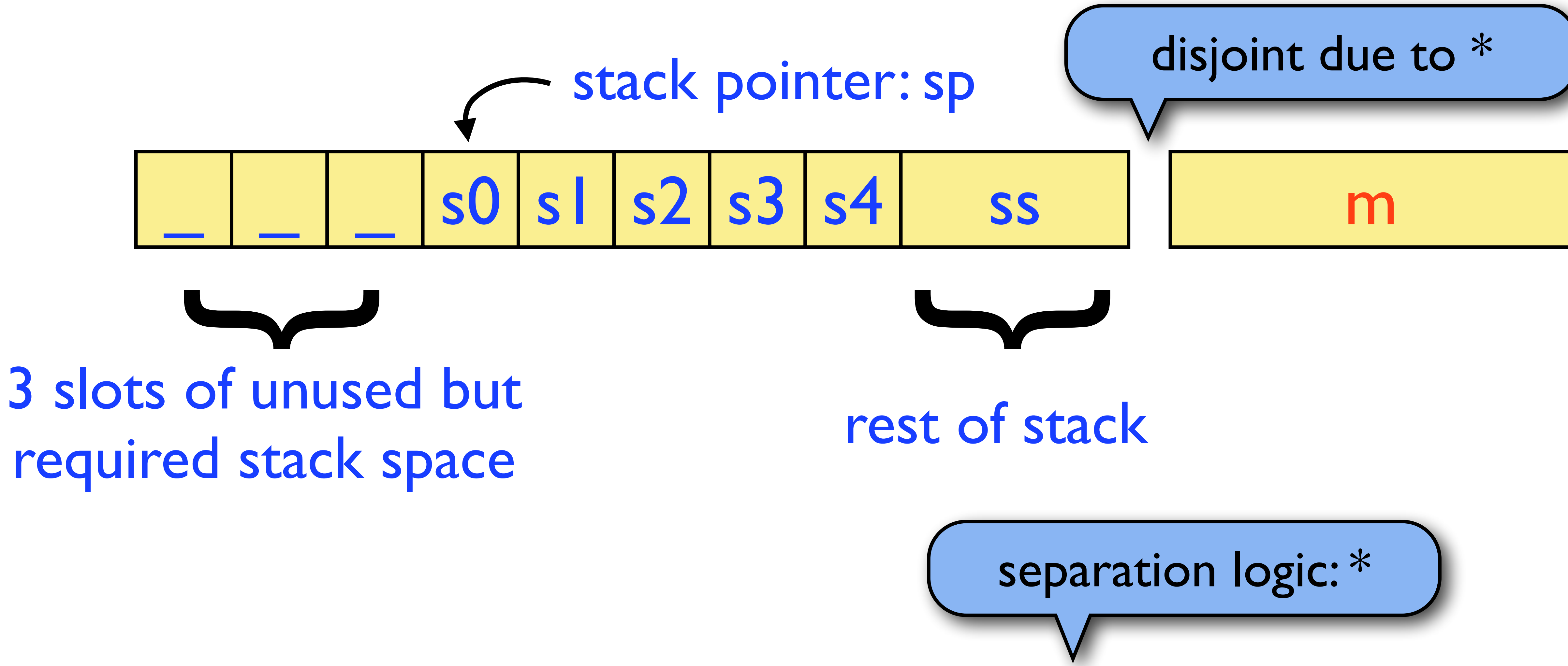


separation logic: *

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

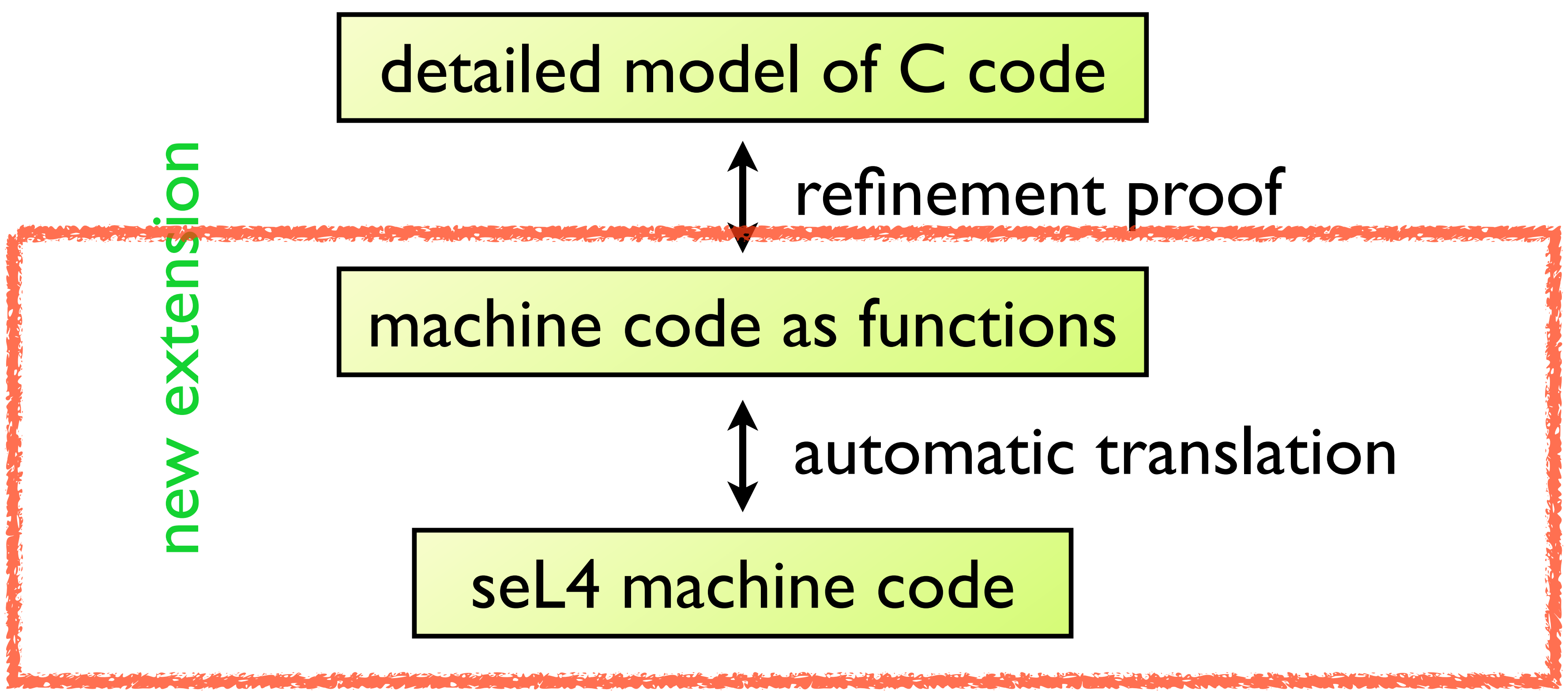
bx lr

r0

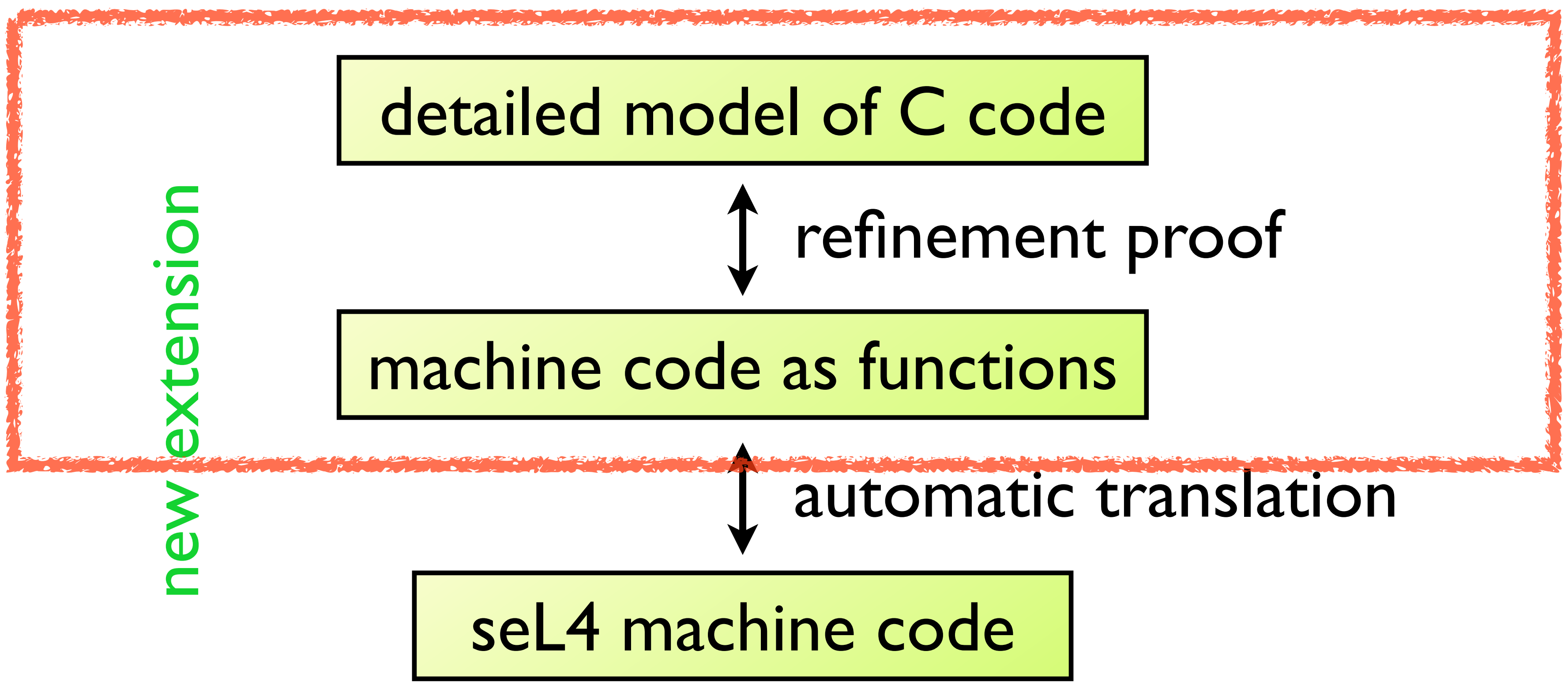
Other C-specifics

- 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

Moving on to stage 2

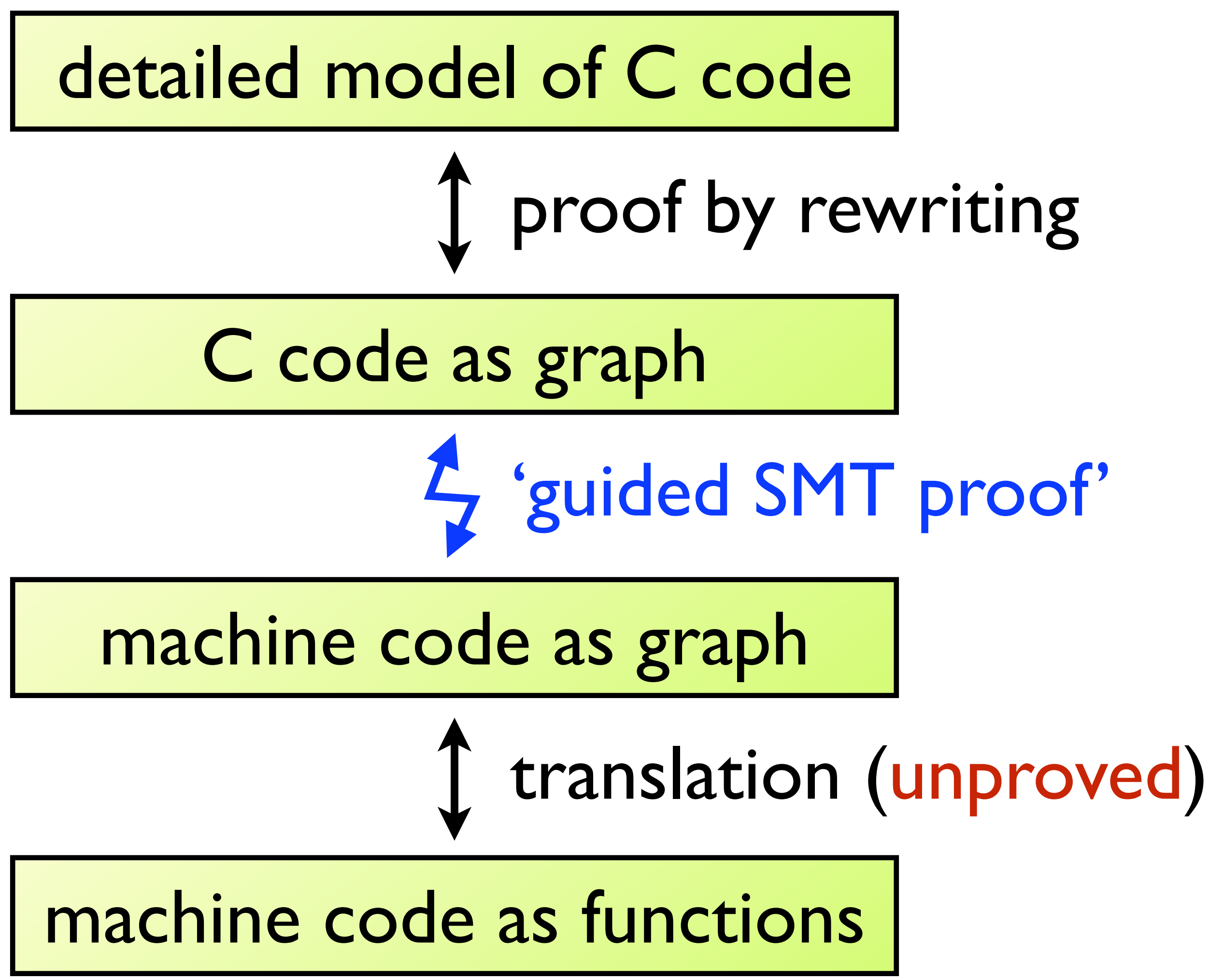


Moving on to stage 2

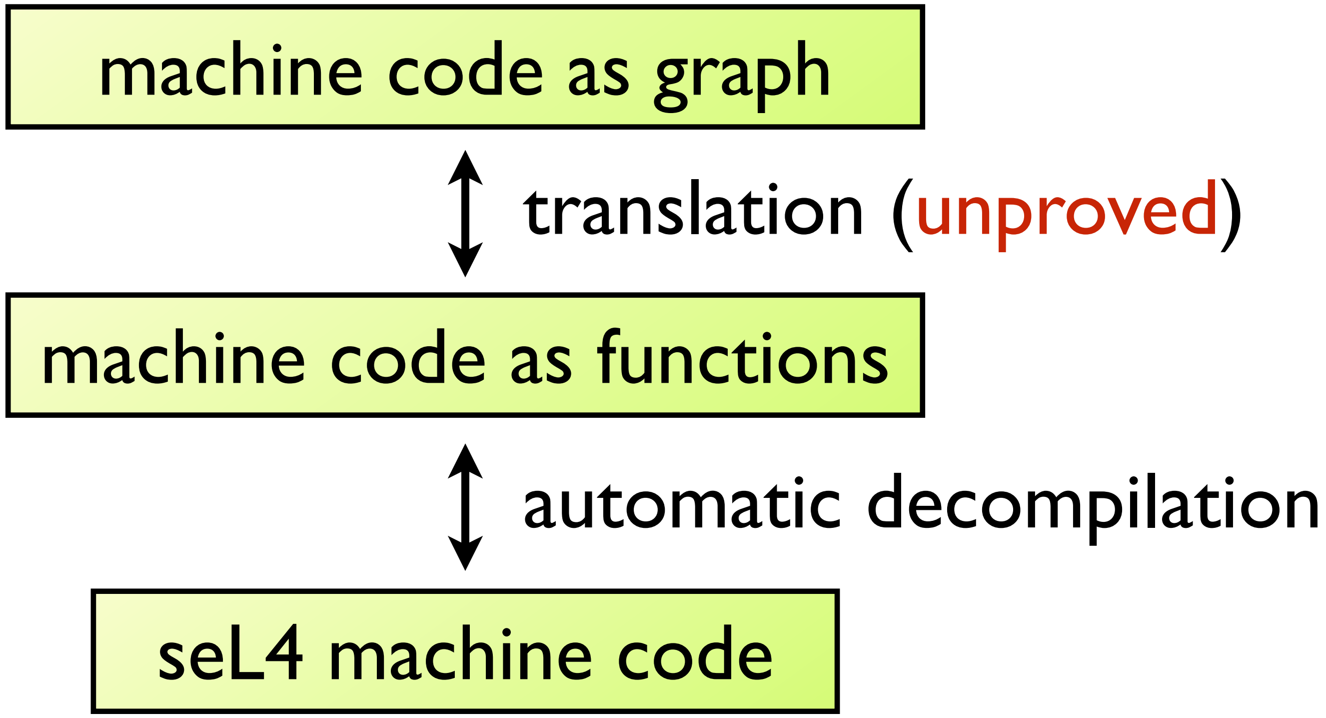


Refinement proof

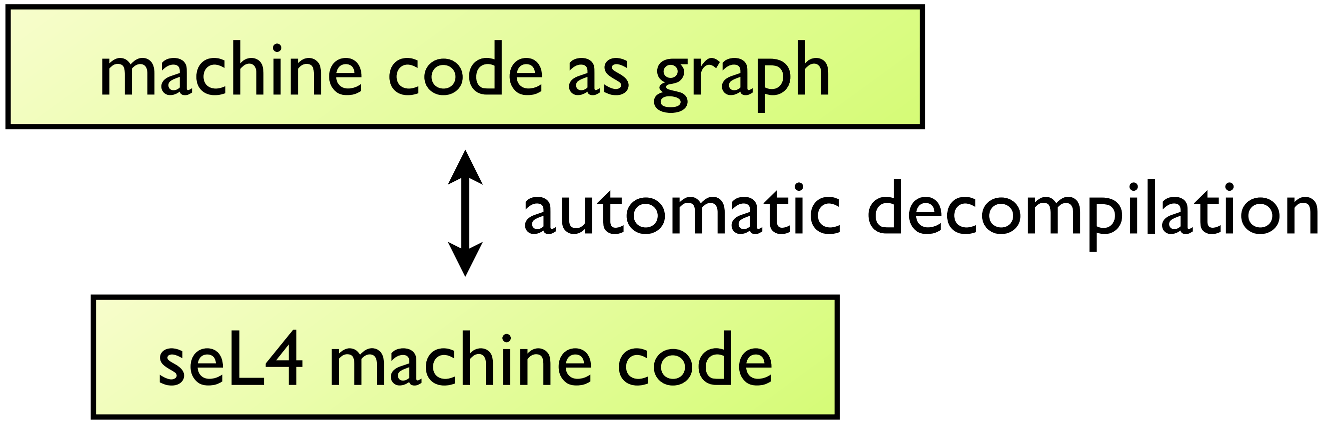
(Work by Thomas Sewell, NICTA)



Graph language



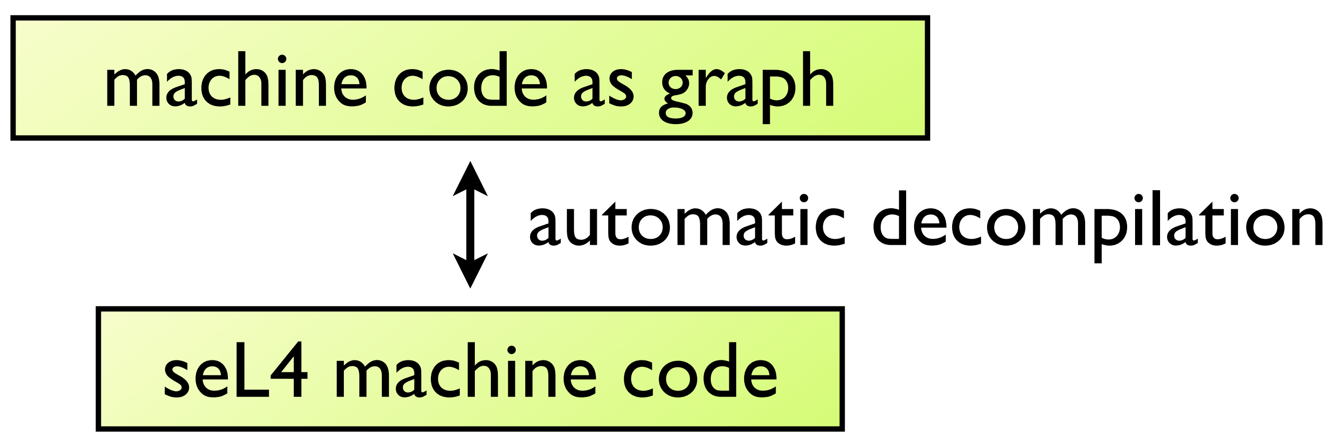
Graph language



Graph language

Node types:

- state update
- test-and-branch
- call



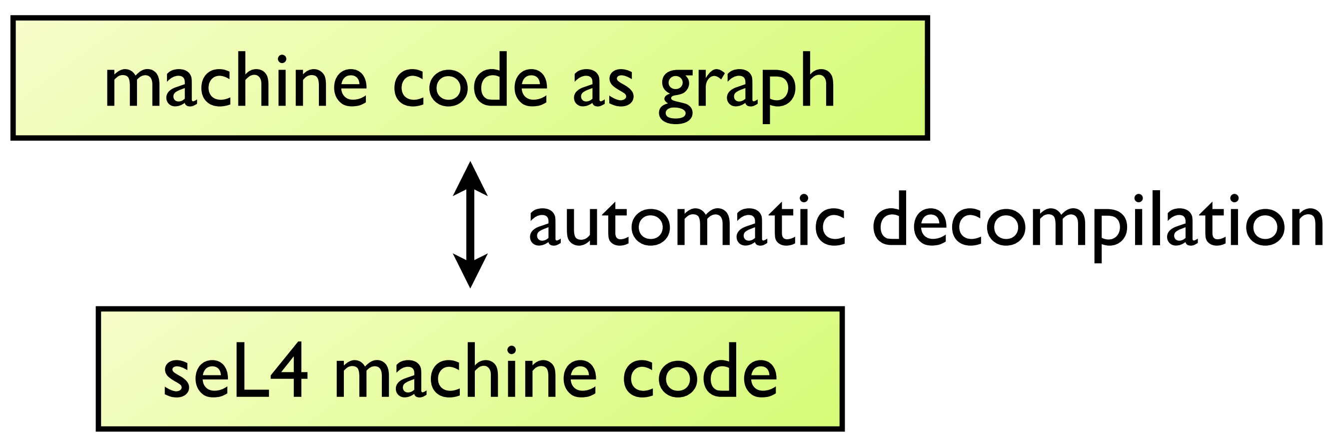
Graph language

Node types:

- state update
- test-and-branch
- call

Next pointers:

- node address
- return (from call)
- error



Graph language

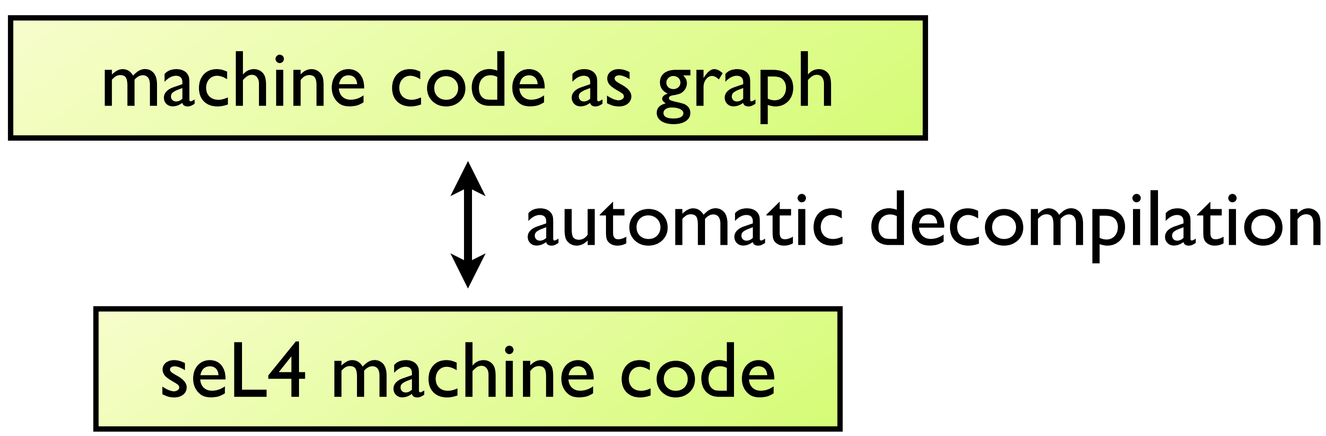
Node types:

- state update
- test-and-branch
- call

Next pointers:

- node address
- return (from call)
- error

Theorem: any exec in graph, can be done in machine code



Graph language

Node types:

- state update
- test-and-branch
- call

Next pointers:

- node address
- return (from call)
- error

Theorem: any exec in graph, can be done in machine code

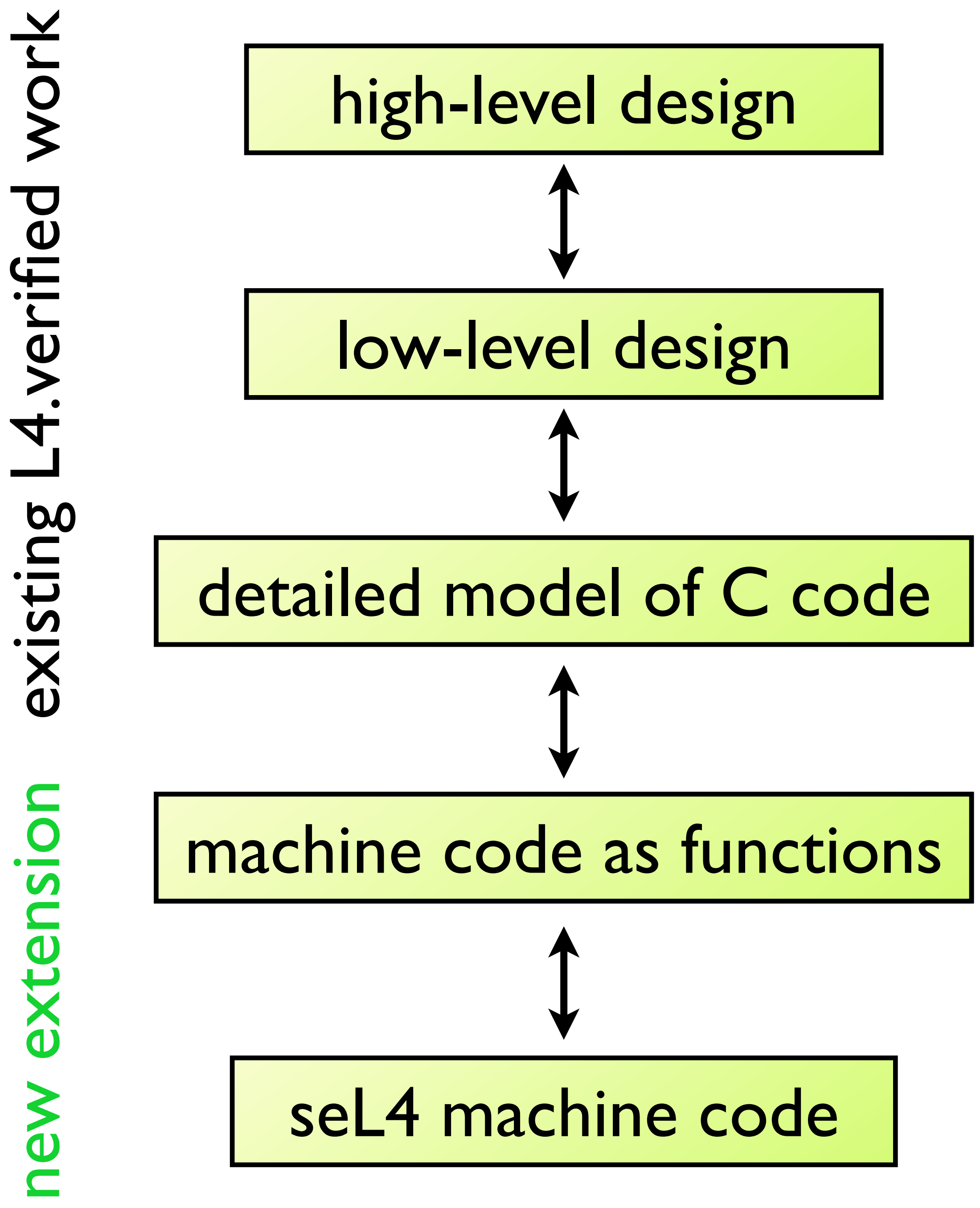
machine code as graph

automatic decompilation

seL4 machine code

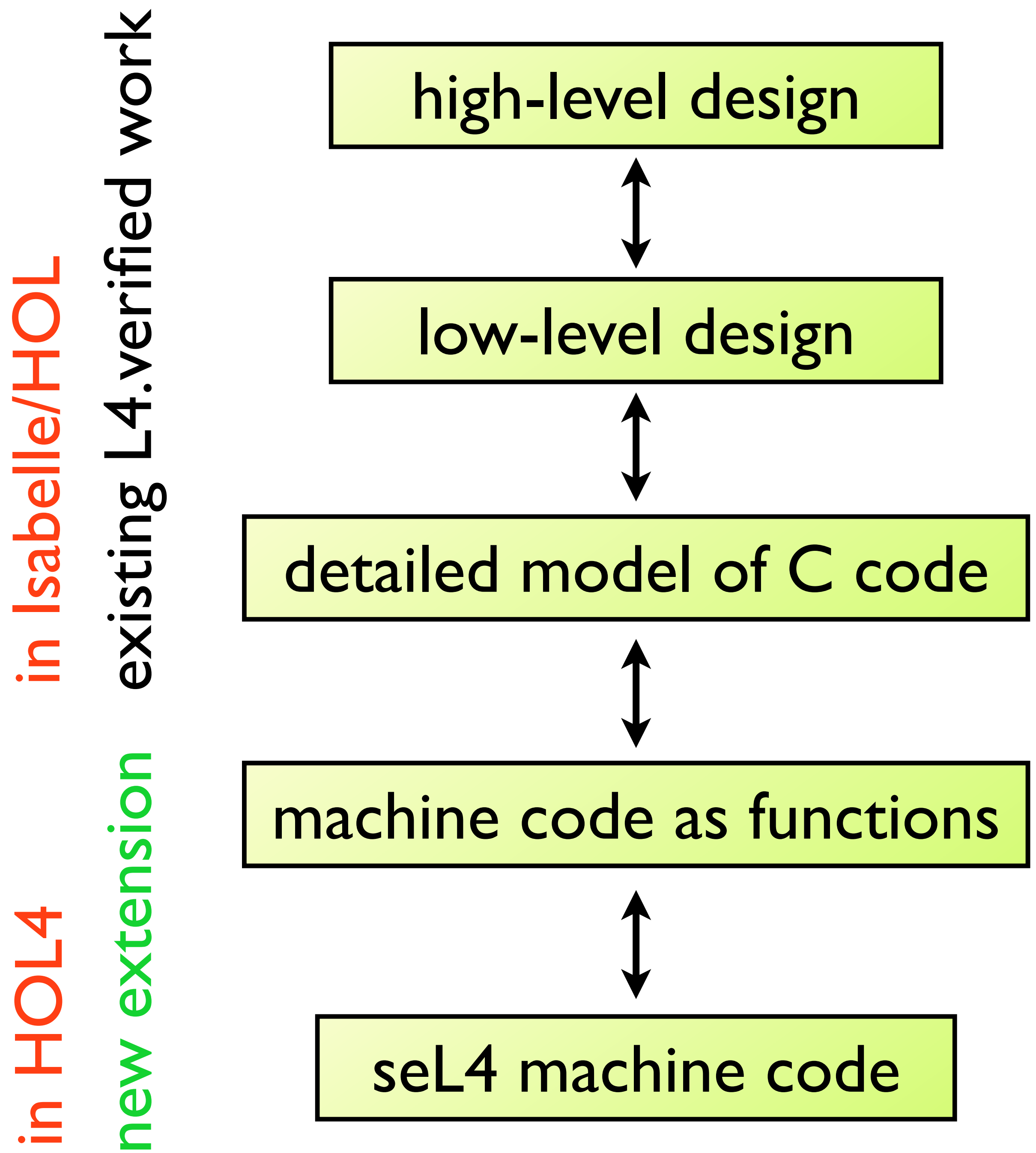
Potential to suit other applications better, e.g. safety analysis.

Connecting provers



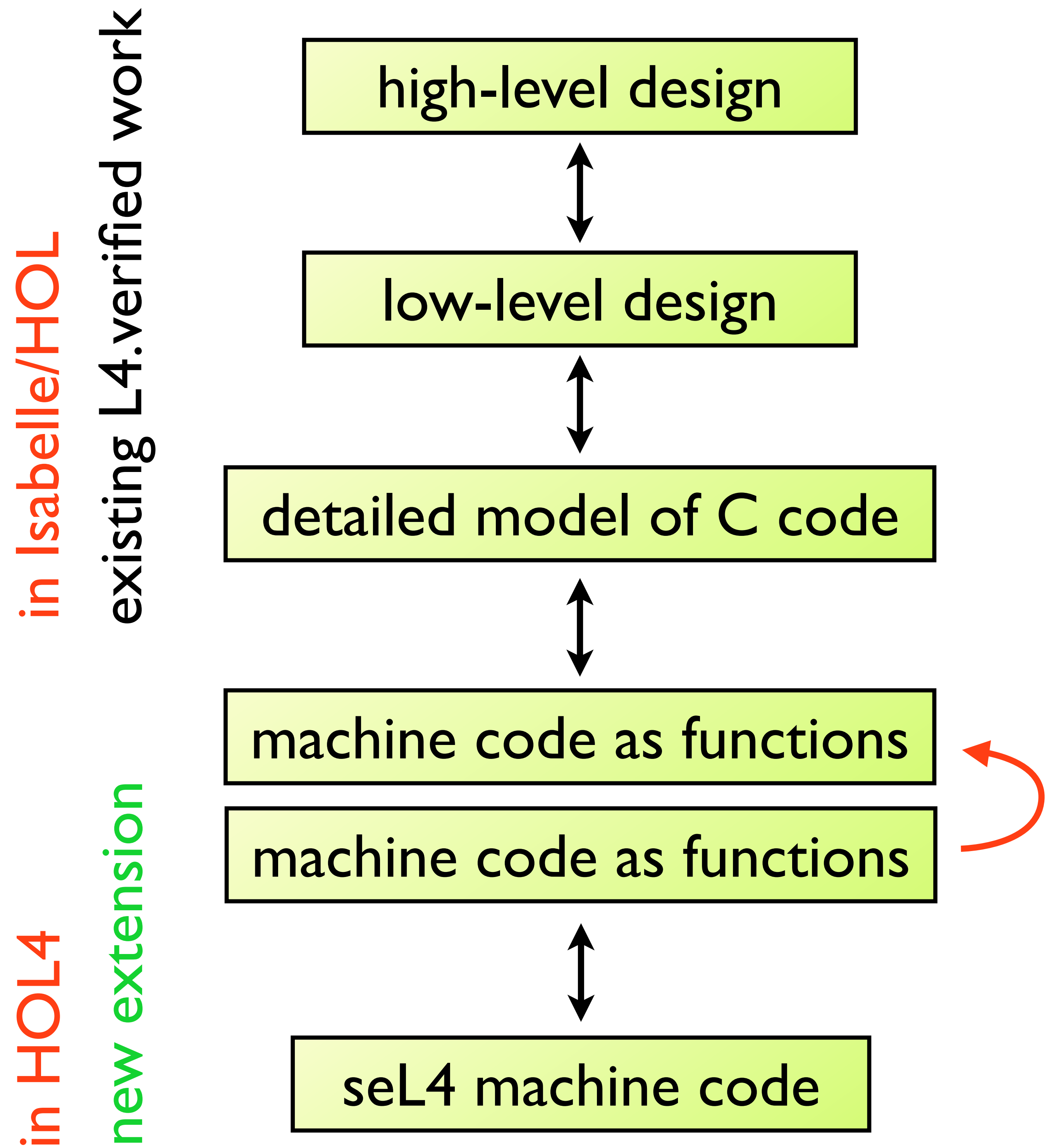
In general, hard.
Easy in this case.

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

Looking back

Success: gcc output for -O1 and -O2 on seL4 decompiles.

Looking back

Success: gcc output for -O1 and -O2 on seL4 decompiles.

However:

stack analysis brittle and requires expert user to debug,

latest version avoids stack analysis,

latest version produces graphs (instead of functions)

Looking back

Success: gcc output for -O1 and -O2 on seL4 decompiles.

However:

stack analysis brittle and requires expert user to debug,

latest version avoids stack analysis,

latest version produces graphs (instead of functions)

A one-fits-all decompilation target?

graph — good for automatic analysis/proofs

functions — readable, good for interactive proofs

Looking back

Success: gcc output for -O1 and -O2 on seL4 decompiles.

However:

stack analysis brittle and requires expert user to debug,

latest version avoids stack analysis,

latest version produces graphs (instead of functions)

A one-fits-all decompilation target?

graph — good for automatic analysis/proofs

functions — readable, good for interactive proofs

Should decompilation be over program logic or machine model?

This talk

Part 1:

- automation: code to spec
- automation: spec to code

Part 2:

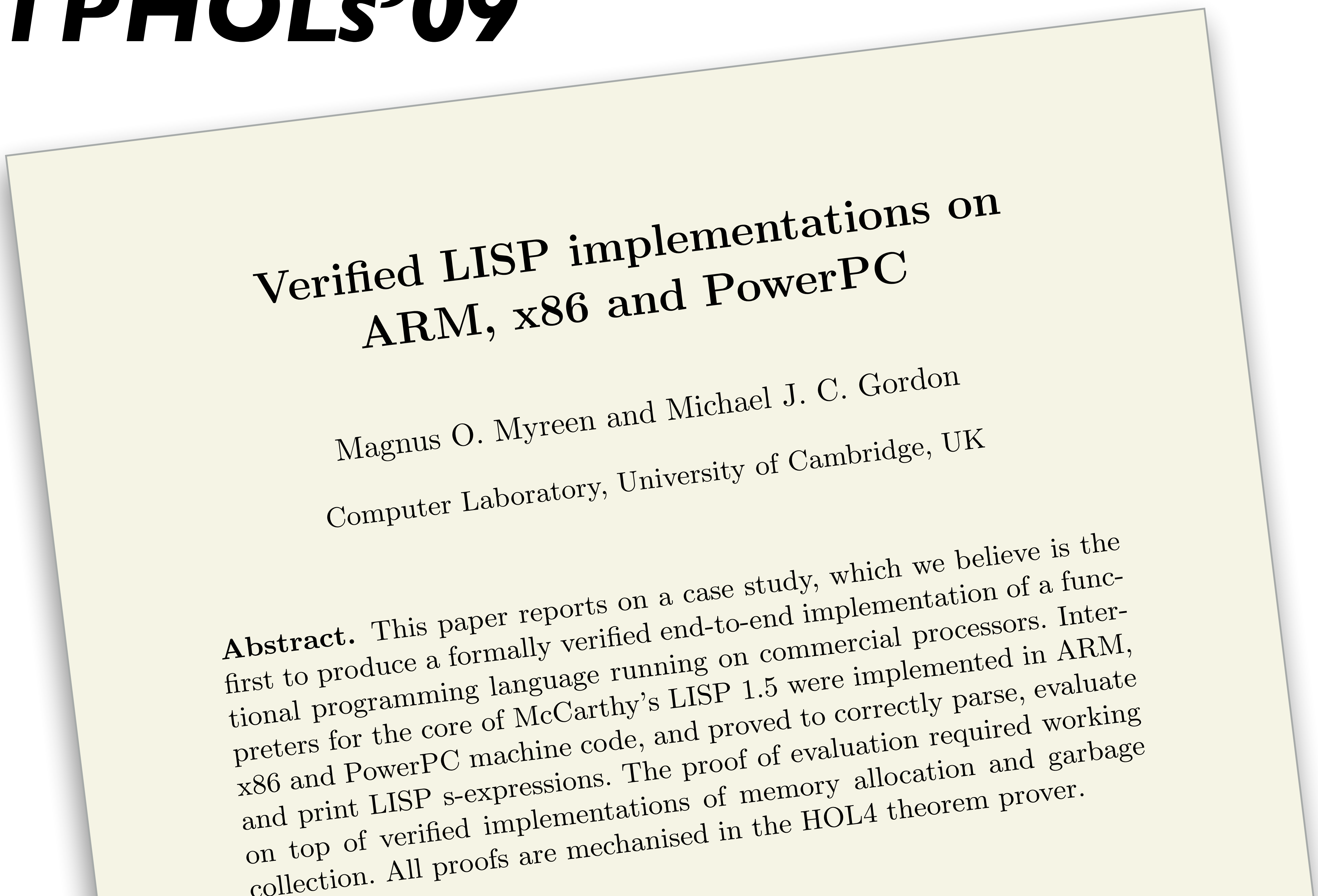
- verification of microkernel

Part 3: construction of correct code

- verified implementation of Lisp that can run Jared Davis' Milawa

Inspiration: Lisp interpreter

TPHOLs'09



Verified LISP implementations on ARM, x86 and PowerPC

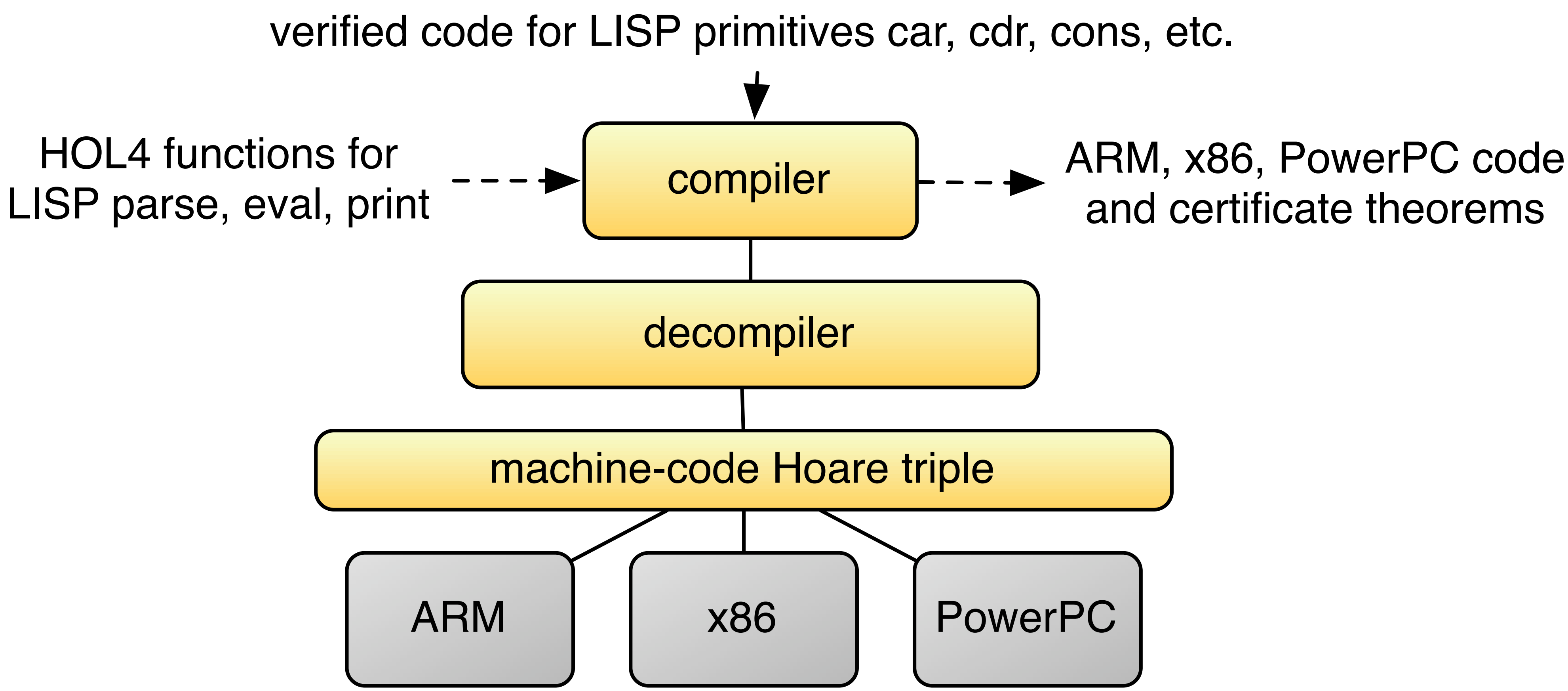
Magnus O. Myreen and Michael J. C. Gordon

Computer Laboratory, University of Cambridge, UK

Abstract. This paper reports on a case study, which we believe is the first to produce a formally verified end-to-end implementation of a functional programming language running on commercial processors. Interpreters for the core of McCarthy's LISP 1.5 were implemented in ARM, x86 and PowerPC machine code, and proved to correctly parse, evaluate and print LISP s-expressions. The proof of evaluation required working on top of verified implementations of memory allocation and garbage collection. All proofs are mechanised in the HOL4 theorem prover.

A verified Lisp interpreter

Idea: create LISP implementations via compilation.



Lisp formalised

LISP s-expressions defined as data-type SExp:

$$\begin{aligned} \mathsf{Num} &: \mathbb{N} \rightarrow \mathsf{SExp} \\ \mathsf{Sym} &: \mathsf{string} \rightarrow \mathsf{SExp} \\ \mathsf{Dot} &: \mathsf{SExp} \rightarrow \mathsf{SExp} \rightarrow \mathsf{SExp} \end{aligned}$$

LISP primitives were defined, e.g.

$$\begin{aligned} \mathsf{cons}\ x\ y &= \mathsf{Dot}\ x\ y \\ \mathsf{car}\ (\mathsf{Dot}\ x\ y) &= x \\ \mathsf{plus}\ (\mathsf{Num}\ m)\ (\mathsf{Num}\ n) &= \mathsf{Num}\ (m + n) \end{aligned}$$

The semantics of LISP evaluation was taken to be Gordon's formalisation of 'LISP 1.5'-like evaluation

Extending the compiler

We define heap assertion 'lisp $(v_1, v_2, v_3, v_4, v_5, v_6, l)$' and prove implementations for primitive operations, e.g.

$$
\begin{array}{l}
\text{is\_pair } v_1 \;\Rightarrow \\
\{\ \text{lisp}\ (v_1, v_2, v_3, v_4, v_5, v_6, l) * \text{pc}\ p\ \} \\
\ p : \texttt{E5934000} \\
\{\ \text{lisp}\ (v_1, \text{car}\ v_1, v_3, v_4, v_5, v_6, l) * \text{pc}\ (p+4)\ \}
\end{array}
$$

$$
\begin{array}{l}
\text{size } v_1 + \text{size } v_2 + \text{size } v_3 + \text{size } v_4 + \text{size } v_5 + \text{size } v_6 < l \;\Rightarrow \\
\{\ \text{lisp}\ (v_1, v_2, v_3, v_4, v_5, v_6, l) * \text{pc}\ p\ \} \\
\ p : \texttt{E50A3018 E50A4014 E50A5010 E50A600C ...} \\
\{\ \text{lisp}\ (\text{cons}\ v_1\ v_2, v_2, v_3, v_4, v_5, v_6, l) * \text{pc}\ (p+332)\ \}
\end{array}
$$

with these the compiler understands:

$$
\begin{array}{l}
\text{let } v_2 = \text{car } v_1 \text{ in } \ldots \\
\text{let } v_1 = \text{cons } v_1\ v_2 \text{ in } \ldots
\end{array}
$$

Reminder

{ 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.)

Reminder

{ 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:

We change these triples to be about lisp heap. Result: more abstraction.

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

(Loops result in recursive functions.)

Running the Lisp interpreter

Nintendo DS lite (ARM)

MacBook (x86)

old MacMini (PowerPC)

```
(pascal-triangle '((1)) '6)
```

returns:

```
((1 6 15 20 15 6 1)
 (1 5 10 10 5 1)
 (1 4 6 4 1)
 (1 3 3 1)
 (1 2 1)
 (1 1)
 (1))
```

Can we do better than a simple Lisp interpreter?

Two projects meet

Jared Davis

A self-verifying theorem prover

Milawa

Magnus Myreen

Verified Lisp implementations

verified **LISP** *on*
ARM, x86, PowerPC

Two projects meet

My theorem prover is written in Lisp.
Can I try your verified Lisp?

Jared Davis

A self-verifying theorem prover

Milawa

Magnus Myreen

Verified Lisp implementations

verified **LISP** *on*
ARM, x86, PowerPC

Two projects meet

My theorem prover is written in Lisp.
Can I try your verified Lisp?

Sure, try it.

Jared Davis

A self-verifying theorem prover

Milawa

Magnus Myreen

Verified Lisp implementations

verified **LISP** *on*
ARM, x86, PowerPC

Two projects meet

My theorem prover is written in Lisp.
Can I try your verified Lisp?

Sure, try it.

Does your Lisp support ..., ... and ...?

Jared Davis

A self-verifying theorem prover

M

Milawa

Magnus Myreen

Verified Lisp implementations

verified **LISP** *on*
ARM, x86, PowerPC

Two projects meet

My theorem prover is written in Lisp.
Can I try your verified Lisp?

Sure, try it.

Does your Lisp support ..., ... and ...?

No, but it could ...

Jared Davis

A self-verifying theorem prover

Milawa

Magnus Myreen

Verified Lisp implementations

verified **LISP** *on*
ARM, x86, PowerPC

Running Milawa

Milawa

verified **LISP** *on*
ARM, x86, PowerPC
(TPHOLs 2009)

Running Milawa

Milawa

verified **LISP** *on*
ARM, x86, PowerPC
(TPHOLs 2009)

Milawa's bootstrap proof:

Running Milawa

Milawa

verified **LISP** *on*
ARM, x86, PowerPC
(TPHOLs 2009)

Milawa's bootstrap proof:

- 4 gigabyte proof file:
  >500 million unique conses

Running Milawa

Milawa

verified **LISP** *on*
ARM, x86, PowerPC
(TPHOLs 2009)

Milawa's bootstrap proof:

- 4 gigabyte proof file:
  >500 million unique conses
- takes 16 hours to run on a state-of-the-art runtime (CCL)

Running Milawa

Milawa

verified **LISP** *on*
ARM, x86, PowerPC
(TPHOLs 2009)

Milawa's bootstrap proof:

- 4 gigabyte proof file:
  >500 million unique conses
- takes 16 hours to run on a state-of-the-art runtime (CCL)

hopelessly "toy"

Running Milawa

Milawa

Jitawa: *verified* **LISP**
with JIT compiler

Milawa's bootstrap proof:

- 4 gigabyte proof file:
  >500 million unique conses
- takes 16 hours to run on a state-of-the-art runtime (CCL)

Contribution:

- a new verified Lisp which is able to host the Milawa thm prover

work by Jared Davis

A short introduction to Milawa

- Milawa is styled after theorem provers such as NQTHM and ACL2,
- has a small trusted logical kernel similar to LCF-style provers,
- ... but does not suffer the performance hit of LCF's fully expansive approach.

work by Jared Davis

Comparison with LCF approach

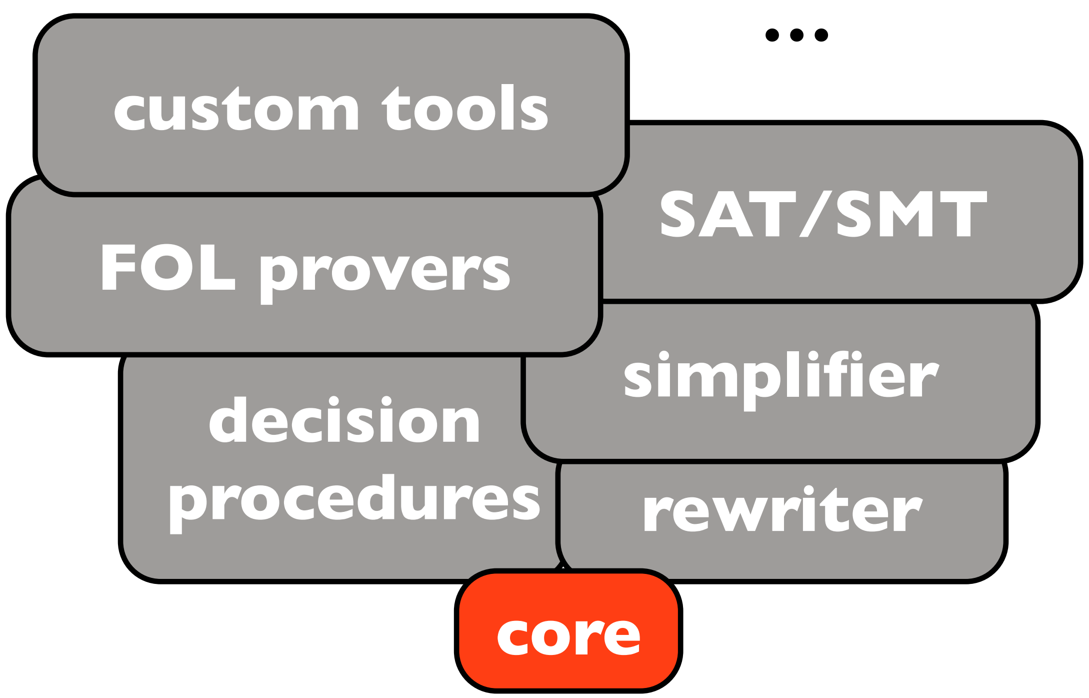
core

LCF-style approach

- all proofs pass through the core's primitive inferences
- extensions steer the core

work by Jared Davis

Comparison with LCF approach

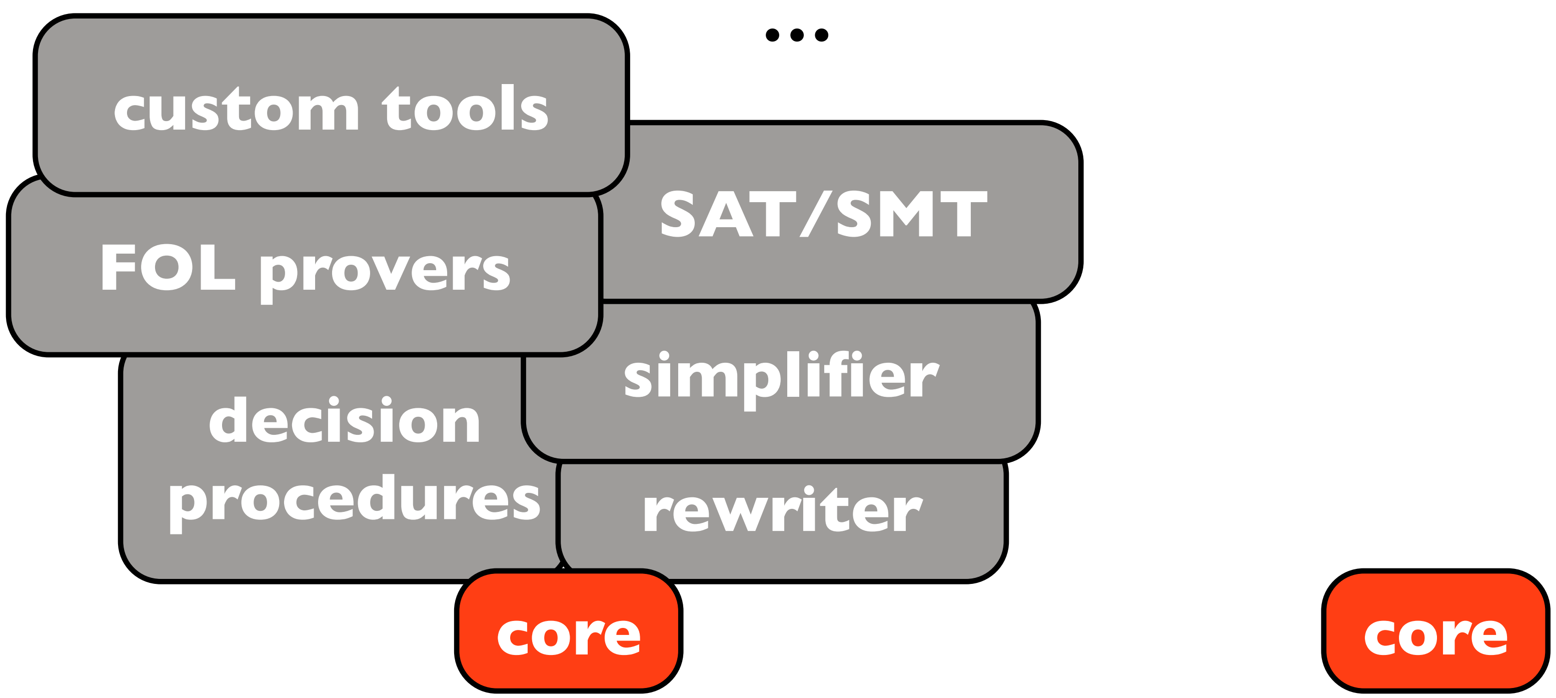


LCF-style approach

- all proofs pass through the core's primitive inferences
- extensions steer the core

work by Jared Davis

Comparison with LCF approach



LCF-style approach

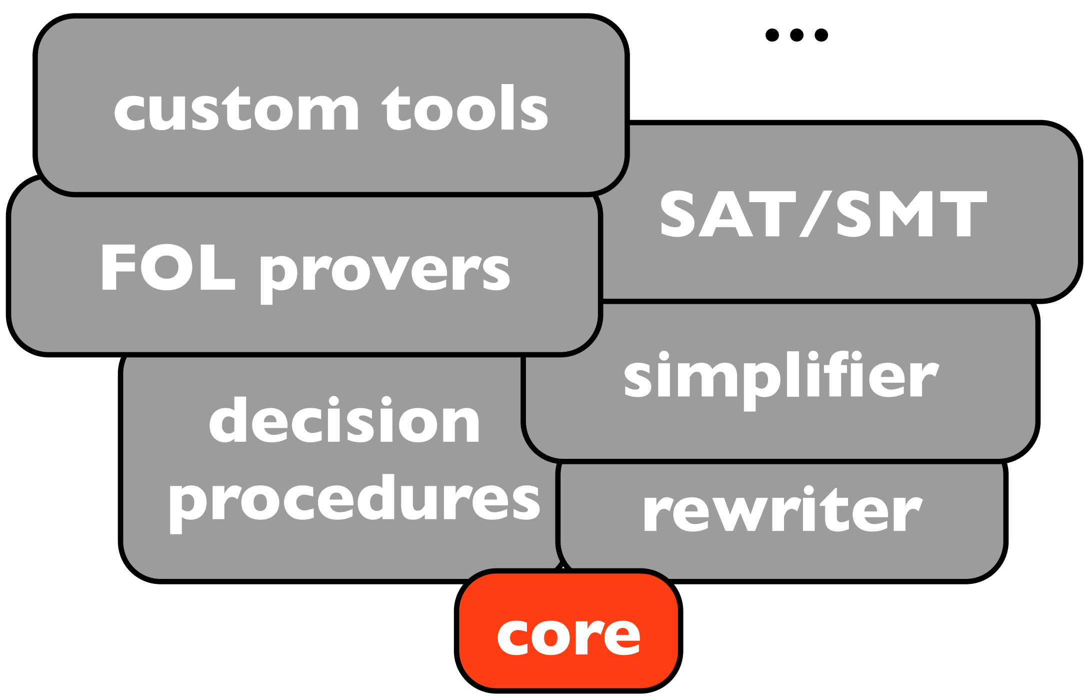
- all proofs pass through the core's primitive inferences
- extensions steer the core

the Milawa approach

- all proofs must pass the core
- the core proof checker can be replaced at runtime

work by Jared Davis

Comparison with LCF approach



LCF-style approach

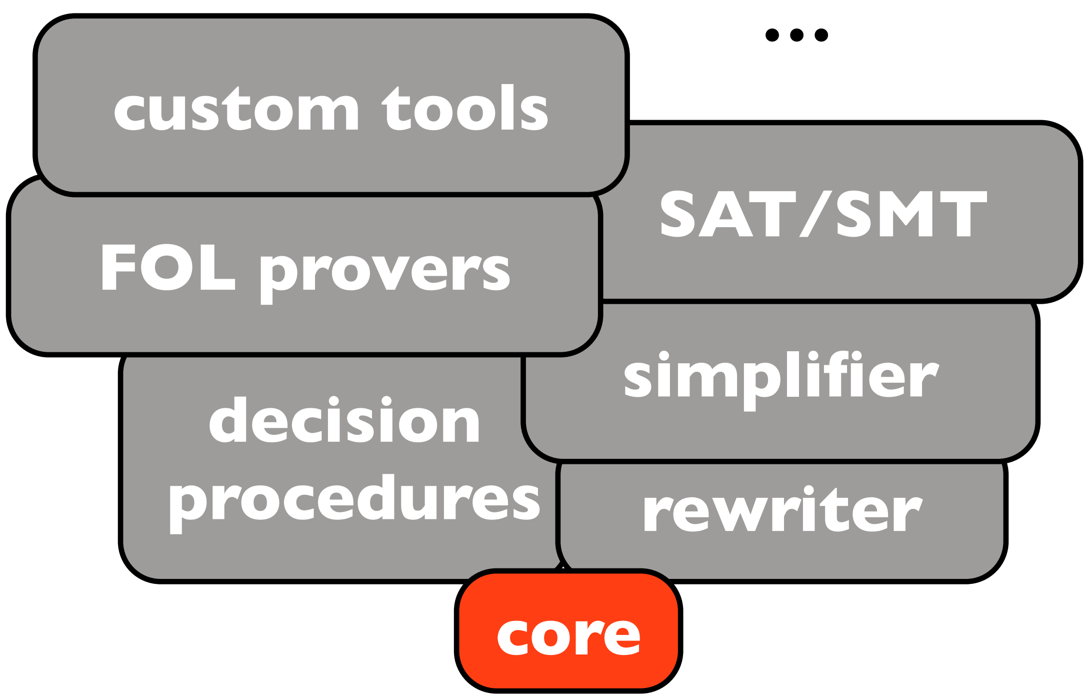
- all proofs pass through the core's primitive inferences
- extensions steer the core

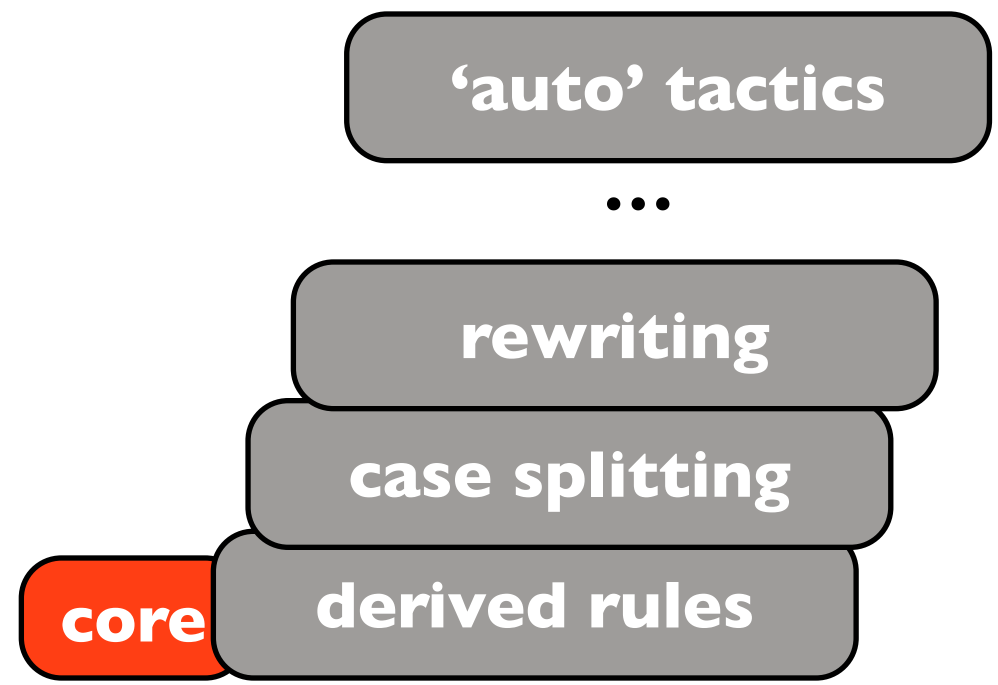
the Milawa approach

- all proofs must pass the core
- the core proof checker can be replaced at runtime

work by Jared Davis

Comparison with LCF approach





LCF-style approach

- all proofs pass through the core's primitive inferences
- extensions steer the core

the Milawa approach

- all proofs must pass the core
- the core proof checker can be replaced at runtime

work by Jared Davis

Requirements on runtime

Milawa uses a subset of Common Lisp which

is for most part first-order pure functions over natural numbers, symbols and conses,

uses primitives: `cons car cdr consp natp symbolp equal + - < symbol-< if`

macros: `or and list let let* cond first second third fourth fifth`

and a simple form of lambda-applications.

(Lisp subset defined on later slide.)

work by Jared Davis

Requirements on runtime

...but Milawa also

- uses destructive updates, hash tables
- prints status messages, timing data
- uses Common Lisp's checkpoints
- forces function compilation
- makes dynamic function calls
- can produce runtime errors

(Lisp subset defined on later slide.)

work by Jared Davis

Requirements on runtime

...but Milawa also

- ~~uses destructive updates, hash tables~~
- ~~prints status messages, timing data~~
- ~~uses Common Lisp's checkpoints~~
- forces function compilation
- makes dynamic function calls
- can produce runtime errors

(Lisp subset defined on later slide.)

work by Jared Davis

Requirements on runtime

...but Milawa also

- ~~uses destructive updates, hash tables~~
- ~~prints status messages, timing data~~
- ~~uses Common Lisp's checkpoints~~

not necessary

- forces function compilation
- makes dynamic function calls
- can produce runtime errors

runtime must support

(Lisp subset defined on later slide.)

Runtime must scale

Designed to scale:

Runtime must scale

Designed to scale:

- just-in-time compilation for speed
  - functions compile to native code

Runtime must scale

Designed to scale:

- just-in-time compilation for speed
  - functions compile to native code
- target 64-bit x86 for heap capacity
  - space for 2^{31} (2 billion) cons cells (16 GB)

Runtime must scale

Designed to scale:

- just-in-time compilation for speed
  - functions compile to native code
- target 64-bit x86 for heap capacity
  - space for 2^{31} (2 billion) cons cells (16 GB)
- efficient scannerless parsing + abbreviations
  - must cope with 4 gigabyte input

Runtime must scale

Designed to scale:

- just-in-time compilation for speed
  - functions compile to native code
- target 64-bit x86 for heap capacity
  - space for 2^{31} (2 billion) cons cells (16 GB)
- efficient scannerless parsing + abbreviations
  - must cope with 4 gigabyte input
- graceful exits in all circumstances
  - allowed to run out of space, but must report it

Workflow

1. specified input language: syntax & semantics
2. verified necessary algorithms, e.g.
   - compilation from source to bytecode
   - parsing and printing of s-expressions
   - copying garbage collection
3. proved refinements from algorithms to x86 code
4. plugged together to form read-eval-print loop

Workflow

~30,000 lines of HOL4 scripts

1. specified input language: syntax & semantics
2. verified necessary algorithms, e.g.
   - compilation from source to bytecode
   - parsing and printing of s-expressions
   - copying garbage collection
3. proved refinements from algorithms to x86 code
4. plugged together to form read-eval-print loop

AST of input language

$term$	$::=$	Const $sexp$		$sexp$	$::=$	Val num
	$\mid$	Var $string$			$\mid$	Sym $string$
	$\mid$	App $func$ ($term$ list)			$\mid$	Dot $sexp$ $sexp$
	$\mid$	If $term$ $term$ $term$				
	$\mid$	LambdaApp ($string$ list) $term$ ($term$ list)				
	$\mid$	Or ($term$ list)				
	$\mid$	And ($term$ list)	(macro)			
	$\mid$	List ($term$ list)	(macro)			
	$\mid$	Let (($string \times term$) list) $term$	(macro)			
	$\mid$	LetStar (($string \times term$) list) $term$	(macro)			
	$\mid$	Cond (($term \times term$) list)	(macro)			
	$\mid$	First $term$ $\mid$ Second $term$ $\mid$ Third $term$	(macro)			
	$\mid$	Fourth $term$ $\mid$ Fifth $term$	(macro)			
$func$	$::=$	Define $\mid$ Print $\mid$ Error $\mid$ Funcall				
	$\mid$	PrimitiveFun $primitive$ $\mid$ Fun $string$				
$primitive$	$::=$	Equal $\mid$ Symbolp $\mid$ SymbolLess				
	$\mid$	Consp $\mid$ Cons $\mid$ Car $\mid$ Cdr $\mid$				
	$\mid$	Natp $\mid$ Add $\mid$ Sub $\mid$ Less				

compile: AST $\longrightarrow$ bytecode list

bytecode	$::=$	Pop	pop one stack element
	$\mid$	PopN *num*	pop n stack elements
	$\mid$	PushVal *num*	push a constant number
	$\mid$	PushSym *string*	push a constant symbol
	$\mid$	LookupConst *num*	push the nth constant from system state
	$\mid$	Load *num*	push the nth stack element
	$\mid$	Store *num*	overwrite the nth stack element
	$\mid$	DataOp *primitive*	add, subtract, car, cons, ...
	$\mid$	Jump *num*	jump to program point n
	$\mid$	JumpIfNil *num*	conditionally jump to n
	$\mid$	DynamicJump	jump to location given by stack top
	$\mid$	Call *num*	static function call (faster)
	$\mid$	DynamicCall	dynamic function call (slower)
	$\mid$	Return	return to calling function
	$\mid$	Fail	signal a runtime error
	$\mid$	Print	print an object to stdout
	$\mid$	Compile	compile a function definition

How do we get just-in-time compilation?

We have verified compilation algorithm:

compile: AST → bytecode list

but compiler must produce real x86 code....

How do we get just-in-time compilation?

We have verified compilation algorithm:

compile: AST → bytecode list

but compiler must produce real x86 code....

Solution:

- bytecode is represented by numbers in memory that are x86 machine code
- we prove that jumping to the memory location of the bytecode executes it

How do we get just-in-time compilation?

Treating code as data:

$$\forall p\ c\ q. \quad \{p\}\ c\ \{q\} \;=\; \{p * \textsf{code}\ c\}\ \emptyset\ \{q * \textsf{code}\ c\}$$

(POPL'10)

Solution:

- bytecode is represented by numbers in memory that <u>are</u> x86 machine code
- we prove that jumping to the memory location of the bytecode executes it

How do we get just-in-time compilation?

Treating code as data:

$$\forall p\ c\ q.\quad \{p\}\ c\ \{q\} \;=\; \{p * \textsf{code}\ c\}\ \emptyset\ \{q * \textsf{code}\ c\}$$

(POPL'10)

Definition of Hoare triple:

$$\begin{aligned}\{p\}\ c\ \{q\} \quad = \quad \forall s\ r.\ & (p * r * \textsf{code}\ c)\ s \implies \\ & \exists n.\ (q * r * \textsf{code}\ c)\ (\textsf{run}\ n\ s)\end{aligned}$$

I/O and efficient parsing

Jitawa implements a read-eval-print loop:

Use of external C routines adds assumptions to proof:

- reading next string from stdin
- printing null-terminated string to stdout

Read-eval-print loop

- Result of reading lazily, writing eagerly
- Eval = compile then jump-to-compiled-code
- Specification: read-eval-print until end of input

$$\frac{\textsf{is\_empty}\ (\textsf{get\_input}\ io)}{(k, io) \xrightarrow{\textsf{exec}} io}$$

$$\frac{\begin{array}{l}\neg\textsf{is\_empty}\ (\textsf{get\_input}\ io) \wedge \\ \textsf{next\_sexp}\ (\textsf{get\_input}\ io)) = (s, rest) \wedge \\ (\textsf{sexp2term}\ s, [], k, \textsf{set\_input}\ rest\ io) \xrightarrow{\textsf{ev}} (ans, k', io') \wedge \\ (k', \textsf{append\_to\_output}\ (\textsf{sexp2string}\ ans)\ io') \xrightarrow{\textsf{exec}} io''\end{array}}{(k, io) \xrightarrow{\textsf{exec}} io''}$$

Correctness theorem

Top-level correctness theorem:

$$\{\,\text{init\_state}\ io * \text{pc}\ p * \langle \text{terminates\_for}\ io \rangle\,\}$$
$$p : \text{code\_for\_entire\_jitawa\_implementation}$$
$$\{\,\text{error\_message} \vee \exists io'.\ \langle ([], io) \xrightarrow{\text{exec}} io' \rangle * \text{final\_state}\ io'\,\}$$

Correctness theorem

There must be enough memory and I/O assumptions must hold.

ness theorem:

$$\{\,\mathsf{init\_state}\ io * \mathsf{pc}\ p * \langle \mathsf{terminates\_for}\ io \rangle\,\}$$

$$p : \mathsf{code\_for\_entire\_jitawa\_implementation}$$

$$\{\,\mathsf{error\_message} \vee \exists io'.\ \langle ([], io) \xrightarrow{\mathsf{exec}} io' \rangle * \mathsf{final\_state}\ io'\,\}$$

Correctness theorem

There must be enough memory and I/O assumptions must hold.

ness theorem:

$$\{\, \mathsf{init\_state}\ io * \mathsf{pc}\ p * \langle \mathsf{terminates\_for}\ io \rangle \,\}$$

$$p : \mathsf{code\_for\_entire\_jitawa\_implementation}$$

$$\{\, \mathsf{error\_message} \vee \exists io'.\ \langle ([], io) \xrightarrow{\mathsf{exec}} io' \rangle * \mathsf{final\_state}\ io' \,\}$$

Each execution is allowed to fail with an error message.

Correctness theorem

There must be enough memory and I/O assumptions must hold.

ness theorem:

$$\{\,\text{init\_state}\ io * \text{pc}\ p * \langle \text{terminates\_for}\ io \rangle\,\}$$

$$p : \text{code\_for\_entire\_jitawa\_implementation}$$

$$\{\,\text{error\_message} \vee \exists io'.\ \langle ([\,], io) \xrightarrow{\text{exec}} io' \rangle * \text{final\_state}\ io'\,\}$$

Each execution is allowed to fail with an error message.

If there is no error message, then the result is described by the high-level op. semantics.

Correctness theorem

There must be enough memory and I/O assumptions must hold.

This machine-code Hoare triple holds only for terminating executions.

ne

$$\{\ \text{init\_state}\ io * \text{pc}\ p * \langle \text{terminates\_for}\ io \rangle\ \}$$
$$p : \text{code\_for\_entire\_jitawa\_implementation}$$
$$\{\ \text{error\_message} \vee \exists io'.\ \langle ([], io) \xrightarrow{\text{exec}} io' \rangle * \text{final\_state}\ io'\ \}$$

Each execution is allowed to fail with an error message.

If there is no error message, then the result is described by the high-level op. semantics.

Correctness theorem

There must be enough memory and I/O assumptions must hold.

This machine-code Hoare triple holds only for terminating executions.

ne

$$\{\,\mathsf{init\_state}\ io * \mathsf{pc}\ p * \langle \mathsf{terminates\_for}\ io \rangle\,\}$$
$$p : \mathsf{code\_for\_entire\_jitawa\_implementation}$$
$$\{\,\mathsf{error\_message} \lor \exists io'.\ \langle ([], io) \xrightarrow{\mathsf{exec}} io' \rangle * \mathsf{final\_state}\ io'\,\}$$

list of numbers

Each execution is allowed to fail with an error message.

If there is no error message, then the result is described by the high-level op. semantics.

Verified code

```
$ cat verified_code.s

    /*  Machine code automatically extracted from a HOL4 theorem.  */
    /*  The code consists of 7423 instructions (31840 bytes).      */

     .byte   0x48, 0x8B, 0x5F, 0x18
     .byte   0x4C, 0x8B, 0x7F, 0x10
     .byte   0x48, 0x8B, 0x47, 0x20
     .byte   0x48, 0x8B, 0x4F, 0x28
     .byte   0x48, 0x8B, 0x57, 0x08
     .byte   0x48, 0x8B, 0x37
     .byte   0x4C, 0x8B, 0x47, 0x60
     .byte   0x4C, 0x8B, 0x4F, 0x68
     .byte   0x4C, 0x8B, 0x57, 0x58
     .byte   0x48, 0x01, 0xC1
     .byte   0xC7, 0x00, 0x04, 0x4E, 0x49, 0x4C
     .byte   0x48, 0x83, 0xC0, 0x04
     .byte   0xC7, 0x00, 0x02, 0x54, 0x06, 0x51
     .byte   0x48, 0x83, 0xC0, 0x04
     ...
```

Running Milawa on Jitawa

Running Milawa's 4-gigabyte booststrap process:

CCL	16 hours	
SBCL	22 hours	
Jitawa	128 hours	(8x slower than CCL)

Running Milawa on Jitawa

Running Milawa's 4-gigabyte booststrap process:

CCL	16 hours	
SBCL	22 hours	
Jitawa	128 hours	(8x slower than CCL)

Jitawa's compiler performs almost no optimisations.

Running Milawa on Jitawa

Running Milawa's 4-gigabyte booststrap process:

CCL	16 hours	
SBCL	22 hours	
Jitawa	128 hours	(8x slower than CCL)

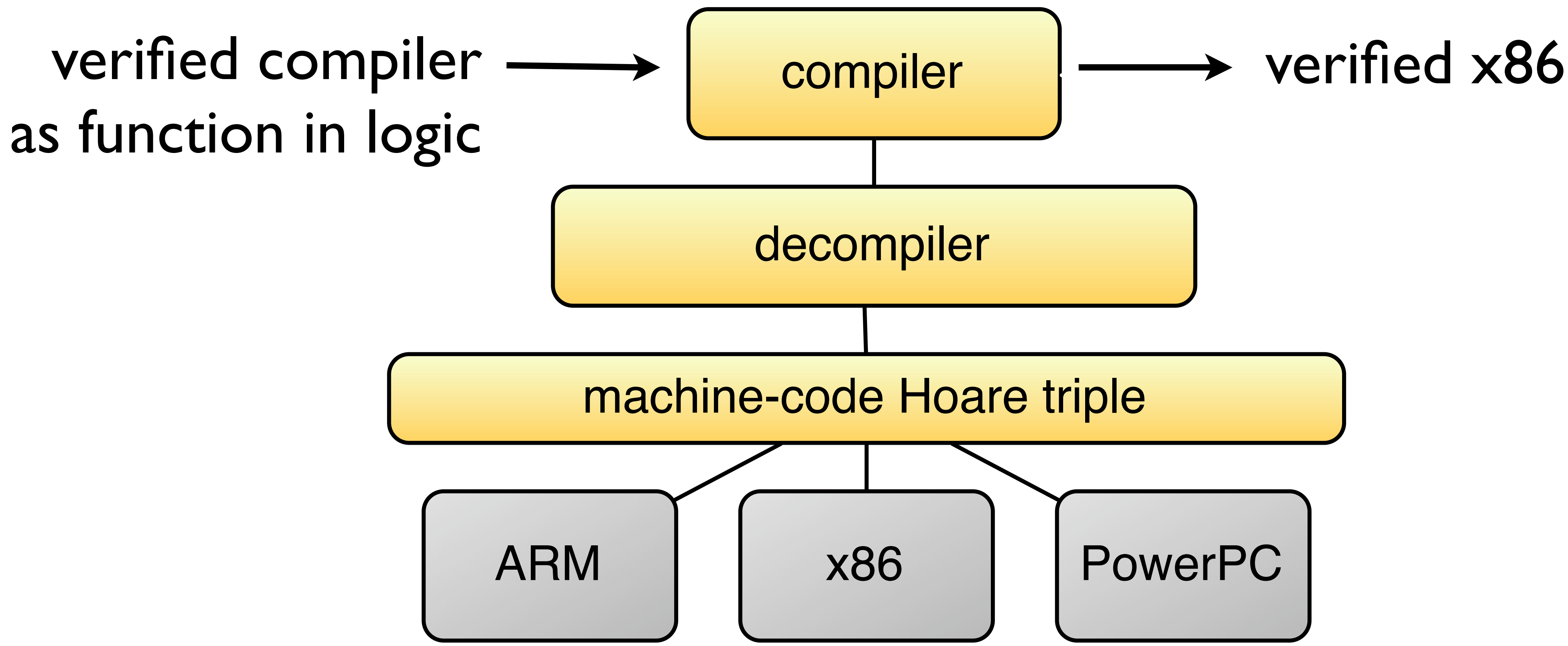
Jitawa's compiler performs almost no optimisations.

Parsing the 4 gigabyte input:

CCL	716 seconds	(9x slower than Jitawa)
Jitawa	79 seconds	

Looking back…

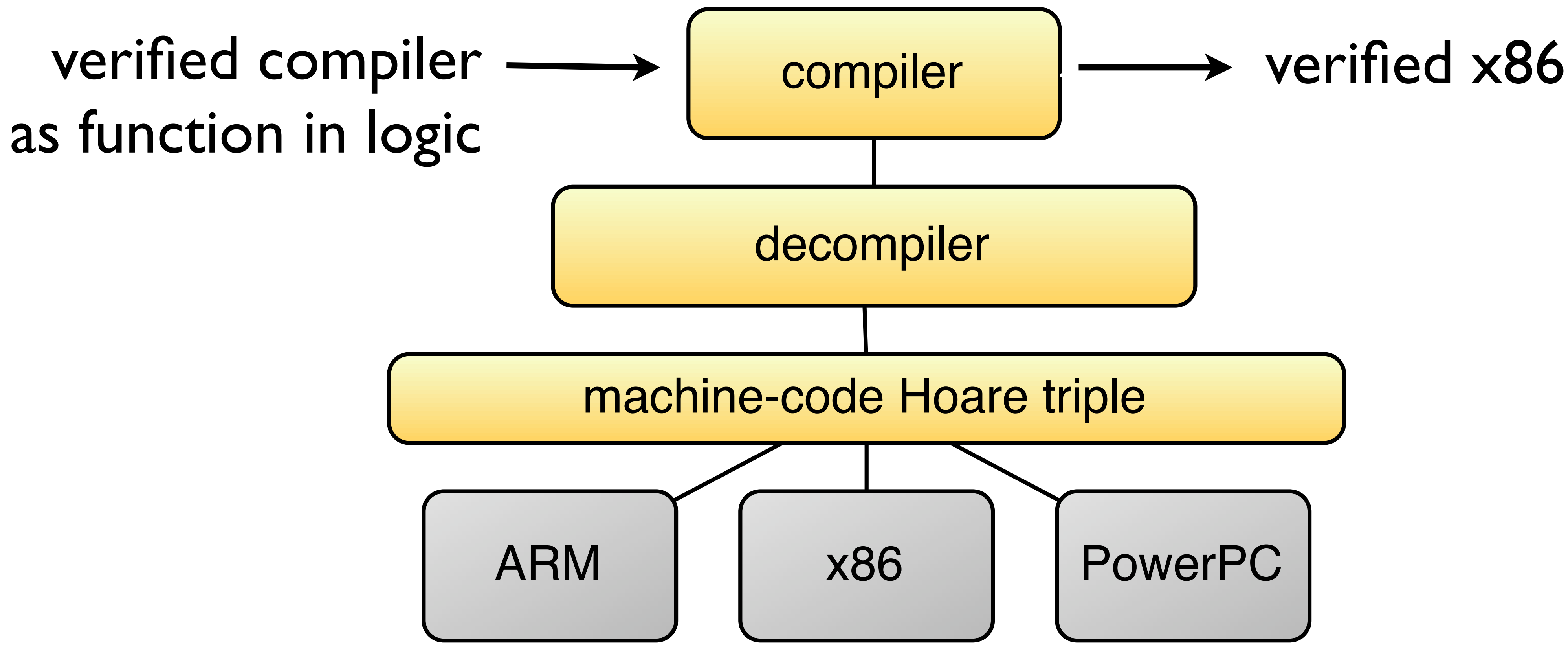
The x86 for the compile function was produced as follows:



Very cumbersome....

Looking back...

The x86 for the compile function was produced as follows:



Very cumbersome....

...should have compiled the verified compiler using itself!

Bootstrapping the compiler

Instead: we bootstrap the verified compile function,
we evaluate the compiler on a deep embedding
of itself within the logic:

EVAL ``compile COMPILE``

derives a theorem:

compile COMPILE = compiler-as-machine-code

The first(?) bootstrapping of a formally verified compiler.

Bootstrapping the compiler

Instead: we bootstrap the verified compile function,
we evaluate the compiler on a deep embedding
of itself within the logic:

EVAL ``compile COMPILE``

in Lisp (eval '(compile compile)) ?

derives a theorem:

compile COMPILE = compiler-as-machine-code

The first(?) bootstrapping of a formally verified compiler.

Ramana Kumar
(Uni. Cambridge)

Magnus Myreen
(Uni. Cambridge)

Michael Norrish
(NICTA, ANU)

Scott Owens
(Uni. Kent)

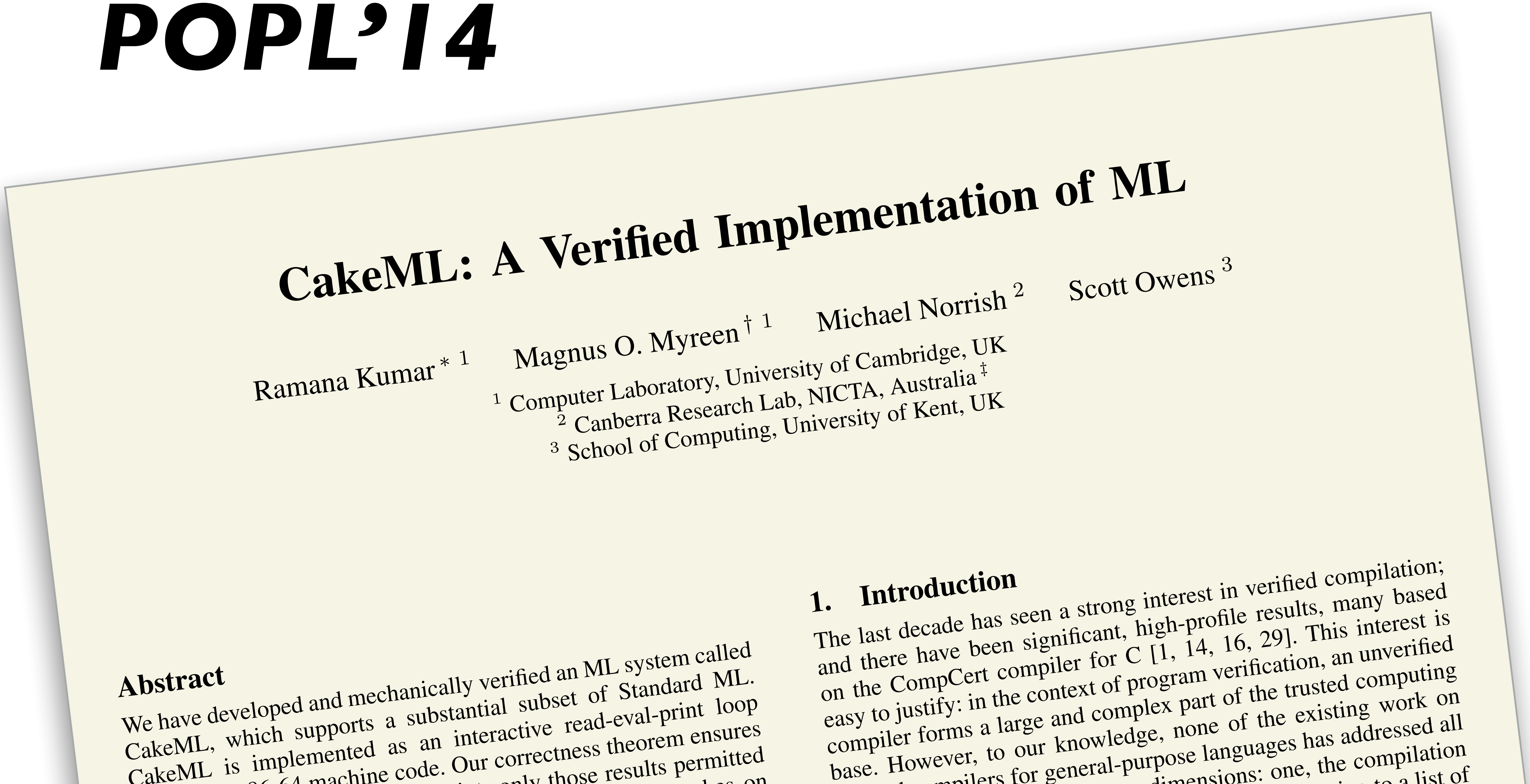
Ramana Kumar
(Uni. Cambridge)

Magnus Myreen
(Uni. Cambridge)

Michael Norrish
(NICTA, ANU)

Scott Owens
(Uni. Kent)

POPL'14



CakeML: A Verified Implementation of ML

Ramana Kumar * [1] Magnus O. Myreen † [1] Michael Norrish [2] Scott Owens [3]

[1] Computer Laboratory, University of Cambridge, UK
[2] Canberra Research Lab, NICTA, Australia ‡
[3] School of Computing, University of Kent, UK

Abstract

We have developed and mechanically verified an ML system called CakeML, which supports a substantial subset of Standard ML. CakeML is implemented as an interactive read-eval-print loop ... x86-64 machine code. Our correctness theorem ensures ... only those results permitted

1. Introduction

The last decade has seen a strong interest in verified compilation; and there have been significant, high-profile results, many based on the CompCert compiler for C [1, 14, 16, 29]. This interest is easy to justify: in the context of program verification, an unverified compiler forms a large and complex part of the trusted computing base. However, to our knowledge, none of the existing work on ... compilers for general-purpose languages has addressed all ... dimensions: one, the compilation ... to a list of

This talk

Part 1: my approach (PhD work)

- automation: code to spec
- automation: spec to code

Part 2: verification of existing code

- verification of gcc output for microkernel (7,000 lines of C)

Part 3: construction of correct code

- verified implementation of Lisp that can run Jared Davis' Milawa

Summary

Techniques from my PhD

- automation: code to spec
- automation: spec to code

worked for two non-trivial case studies:

- verification of gcc output for microkernel (7,000 lines of C)
- verified implementation of Lisp that can run Jared Davis' Milawa

Summary

Techniques from my PhD

- automation: code to spec
- automation: spec to code

worked for two non-trivial case studies:

- verification of gcc output for microkernel (7,000 lines of C)
- verified implementation of Lisp that can run Jared Davis' Milawa

Lessons were learnt:

Summary

Techniques from my PhD

- automation: code to spec
- automation: spec to code

worked for two non-trivial case studies:

- verification of gcc output for microkernel (7,000 lines of C)
- verified implementation of Lisp that can run Jared Davis' Milawa

Lessons were learnt:

- decompiler shouldn't try to be smart (stack)

Summary

Techniques from my PhD

- automation: code to spec
- automation: spec to code

worked for two non-trivial case studies:

- verification of gcc output for microkernel (7,000 lines of C)
- verified implementation of Lisp that can run Jared Davis' Milawa

Lessons were learnt:

- decompiler shouldn't try to be smart (stack)
- compile the verified compiler with itself!

Summary

Questions?

Techniques from my PhD

- automation: code to spec
- automation: spec to code

worked for two non-trivial case studies:

- verification of gcc output for microkernel (7,000 lines of C)
- verified implementation of Lisp that can run Jared Davis' Milawa

Lessons were learnt:

- decompiler shouldn't try to be smart (stack)
- compile the verified compiler with itself!