Ad hoc C: Reflections on Pragmatic Semantics

Michael Norrish
Software Systems Research Group
Three seL4 Stories

Basic C Technology (the “C parser”)
- worryingly *ad hoc*

Binary Validation (Sewell & Myreen)
- *ad hoc*, but *dispels* worry
- PLDI 2013

Automatic Abstraction (Greenaway)
- *ad hoc* complexity/effort removal
- ITP 2012
seL4, c2009

Fully featured operating system kernel

9000 lines of source code

Full functional correctness

25 person-years of effort
Python

Haskell Prototype

Formal Specification

Formal Design

C Code

Actual C Code
Making C a high integrity language?

- some industrial experience, e.g., MISRA C
- Better, CompCert is a **verified compiler**!

- but we can’t use it

C in L4.verified

- for the OS hackers: pointer arithmetic, casts, interpreting memory as untyped bytes
- for the verifiers: restrictions on side effects
- tool-chain: very dependent on `gcc -O2` and linker
C subset

Everything from C standard

• including:
  - pointers, casts, pointer arithmetic
  - data types
  - structs, padding
  - pointers into structs
  - precise finite integer arithmetic

• minus:
  - goto, switch fall-through
  - reference to local variable
  - side-effects in expressions
  - function pointers (restricted)
  - unions

• plus compiler assumptions on:
  - data layout, encoding, endianess
C, as defined by our tools

11000 lines of SML
– (+19kloc in parser/lexer code)
– turns C into Isabelle/HOL definitions
– Verifier uses Isabelle to prove refinement theorem

Errors
– In translator: unexpected goals
  ‣ (Verifier says “This isn’t what I think the C means”)
– In the code: unsolvable goals

Trusted code-base
Other Bits of C Technology

SIMPL’s Verification Condition Generator
– general tool that often “just works”

seL4 Refinement Tactic
– used to verify link between C code to and (Haskell-derived) design-level spec.

Union preprocessing
– seL4 hackers didn’t trust/like gcc’s implementation of unions
– custom tool translates out all unions into structs and casts

[TPHOLs 2009]
Binary Validation: the Problem

```c
void suspend(tcb_t *target) {
    ipcCancel(target);
    setThreadState(tar
    tcbSchedDequeue(ta
```
Binary Validation: the Problem

```c
void suspend(tcb_t *target) {
    ipcCancel(target);
    setThreadState(tar
    tcbSchedDequeue(ta
```
Binary Validation: the Problem

```c
void suspend(tcb_t *target) {
    ipcCancel(target);
    setThreadState(tar
tcbSchedDequeue(ta
```
Binary Validation: the Problem

```c
void suspend(tcb_t *target) {
    ipcCancel(target);
    setThreadState(target);
    tcbSchedDequeue(target);
}
```

**LOGIC**

**FORMAL MODEL**

**REAL WORLD**

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Instruction</th>
<th>Register Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>f0015594</td>
<td>e92d4010</td>
<td>push {r4, lr}</td>
<td></td>
</tr>
<tr>
<td>f0015598</td>
<td>ea04000</td>
<td>mov r4, r0</td>
<td></td>
</tr>
<tr>
<td>f001559c</td>
<td>ebff0028</td>
<td>bl f0015244</td>
<td></td>
</tr>
<tr>
<td>f00155a0</td>
<td>ea00004</td>
<td>mov r0, r4</td>
<td></td>
</tr>
<tr>
<td>f00155a4</td>
<td>ea01000</td>
<td>mov r1, #0</td>
<td></td>
</tr>
<tr>
<td>f00155a8</td>
<td>ebff9a5</td>
<td>bl f0013c44</td>
<td></td>
</tr>
<tr>
<td>f00155ac</td>
<td>ea00004</td>
<td>mov r0, r4</td>
<td></td>
</tr>
</tbody>
</table>

**seL4: Formal Verification and All That**


**UNIVERSITY OF CAMBRIDGE**

ARM ISA and Decompiler

Anthony Fox and Magnus Myreen
Binary Validation: Solution in a Graph

```c
void suspend(tcb_t *target) {
    import(target);
    setThreadState(tar
    tcbSchedDequeue(ta
}
```

**Translation Validation for a Verified OS Kernel**

Thomas Sewell, Magnus Myreen, Gerwin Klein

4/18
Binary Validation: Solution in a Graph

- Informal transformations

void suspend(tcb_t *target) {
    interrupt_next();
    setThreadState(target);
    tcbSchedDequeue(target);
}

- Isabelle/HOL
- PseudoCompilation

- SMT+

- HOL4
- Decompilation into Logic
- Cambridge ARM Semantics
Machine Code Decompilation

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

<avg>:

<table>
<thead>
<tr>
<th>addr</th>
<th>code</th>
<th>op</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>avg+0</td>
<td>e08100000</td>
<td>add r0, r1, r0</td>
<td>add r1 to r0</td>
</tr>
<tr>
<td>avg+4</td>
<td>e1a000a0</td>
<td>lsr r0, r0, #1</td>
<td>shift r0 right</td>
</tr>
<tr>
<td>avg+8</td>
<td>e12fff1e</td>
<td>bx lr</td>
<td>return</td>
</tr>
</tbody>
</table>

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

avg (r0, r1)
Machine code and C code eventually turn into annotated control flow graphs

```plaintext
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;
}
```

Critically, can consider one function at a time when verifying (modulo inlining).
Comparing Graphs: Challenges

**Inlining**
- if gcc inlines a function, the “C graph” has to do so too

**Loops**
- must prove that loop points are reached the same number of times
- must also cope with complete or partial unrollings

**Treatment of the Stack**
- machine code accesses to memory that are loads/stores of spilled parameters need to be recognised as such
Loops, Loops, Loops, Loops, Loops, ...

Can sometimes infer that a loop should only ever execute a fixed $n$ times.

– e.g., `for (i=0; i < 10; i++) { .... }`

– SMT solver can confirm such an inference ...
  
  • formula is finite, of size $O(n)$

– ... and establish appropriate post-condition(s)

Alternatively, use $k$-induction to prove that every visit to a loop point in the binary is matched by one in the C.

– invariant relating C and binary variables is also preserved

– offsets may be required to handle partial unrollings
Graph Equivalence Technology

Problem domain is good fit for QF_ABV SMT category
- i.e., arrays + boolean vectors

Used both Z3 and Sonolar
- Sonolar performs better on arrays (used to represent memory)
- Z3 can restart (retracting facts), making efficiency performance better
seL4, c2009

Fully featured operating system kernel

9000 lines of source code

Full functional correctness

25 person-years of effort
seL4, c2009

Fully featured operating system kernel

9000 lines of source code

Full functional correctness

25 person-years of effort

This is a disturbingly large number; where can we improve productivity?
Some background to this mess

```c
int max(int a, int b) {
    if (a < b) {
        return b;
    }
    return a;
}
```
Some background to this mess

```c
int max(int a, int b) {
    if (a < b) {
        return b;
    }
    return a;
}
```
Some background to this mess

max \ a \ b \equiv
\begin{cases} 
  b & \text{if } a > b \\
  a & \text{else}
\end{cases}

```c
int max(int a, int b) {
    if (a < b) {
        return b;
    }
    return a;
}
```
Some background to this mess

max \ a \ b ≡ 
  if \ a > \ b \ then \ b \ else \ a

int max(int a, int b) {
    if (a < b) {
        return b;
    }
    return a;
}
Some background to this mess

```
int max(int a, int b) {
    if (a < b) {
        return b;
    }
    return a;
}
```

```
max ≣
TRY
    IF ⦃ 'a < s 'b ⦄ THEN
      'ret_int :== 'b;;
      'exn_var :== Return;;
      THROW
    ELSE
      SKIP;;
    'ret_int :== 'a;;
    'exn_var :== Return;
    THROW;
    GUARD DontReach {}
      SKIP;;
  CATCH
    SKIP
END
```
Some background to this mess

```c
int max(int a, int b) {
    if (a < b) {
        return b;
    }
    return a;
}
```

```plaintext
max ≡
TRY
    IF ⦃ 'a <s 'b ⦄ THEN
      'ret_int :== 'b;;
      'exn_var :== Return;;
      THROW
    ELSE
      ...
      'ret_int :== 'a;;
      'exn_var :== Return;
      THROW;
    GUARD DontReach {}
      SKIP;;
  CATCH
    SKIP
  END
```
Some background to this mess

```c
int max(int a, int b) {
    if (a < b) {
        return b;
    }
    return a;
}
```

```ocaml
max ≣
  TRY
    IF ⦃ 'a <s 'b ⦄ THEN
      'ret_int :== 'b;;
      'exn_var :== Return;;
      THROW
    ELSE
      ...;
      'ret_int :== 'a;;
      'exn_var :== Return;
      THROW;
    GUARD DontReach {};
      SKIP;;
  CATCH
    SKIP
  END
```

```javascript
Thursday, 12 July 12

Some background to this mess

```c
int max(int a, int b) {
    if (a < b) {
        return b;
    }
    return a;
}
```

```ocaml
max ≣
  TRY
    IF ⦃ 'a <s 'b ⦄ THEN
      'ret_int :== 'b;;
      'exn_var :== Return;;
      THROW
    ELSE
      ...;
      'ret_int :== 'a;;
      'exn_var :== Return;
      THROW;
    GUARD DontReach {};
      SKIP;;
  CATCH
    SKIP
  END
```
Some background to this mess

- Uninitialised variables
- Undefined behaviour
- Pointer arithmetic
- Type casting

```c
int max(int a, int b) {
    if (a < b) {
        return b;
    }
    return a;
}
```

```max
TRY
    IF ⦃ 'a <s 'b ⦄ THEN
        'ret_int :== 'b;;      'exn_var :== Return;;      THROW
    ELSE
      ...    'ret_int :== 'a;;    'exn_var :== Return;    THROW;
    GUARD DontReach {}
      SKIP;;
  CATCH
    SKIP
  END
```
Our approach
Our approach

- Start with a conservative logical representation
Our approach

- Start with a conservative logical representation
- Automatically abstract low-level spec into higher-level spec
  - Goal: Suitable for human consumption
Our approach

- Start with a conservative logical representation
- Automatically abstract low-level spec into higher-level spec
  - Goal: Suitable for human consumption
- Automatically generate a refinement proof showing the original spec is a refinement of the generated spec
Our approach

- Start with a conservative logical representation
- Automatically abstract low-level spec into higher-level spec
  - Goal: Suitable for human consumption
- Automatically generate a refinement proof showing the original spec is a refinement of the generated spec
Our approach
Our approach
Our approach

1. Shallow Embedding
2. Peephole Optimisations
3. Exception Elimination
4. Local Variable Extraction
5. Flow-Sensitive Optimisations
6. Type Strengthening

Output Specification

C Code
max ≡
TRY
  IF ⦃ 'a <s 'b ⦄ THEN
    'ret_int :== 'b;;
    'exn_var :== Return;;
    THROW
  ELSE
    SKIP
  FI;;
  'ret_int :== 'a;;
  'exn_var :== Return;
  THROW;
GUARD  DontReach {}
  SKIP;;
CATCH
  SKIP
END

① Shallow Embedding
\[
\text{max } a \ b \equiv \\
\text{if } (a < s b) \text{ then } b \text{ else } a
\]
int max(int a, int b) {
    if (a < b) {
        return b;
    }
    return a;
}
int max(int a, int b) {
  if (a < b) {
    return b;
  }
  return a;
}
int max(int a, int b) {
    if (a < b) {
        return b;
    }
    return a;
}

max a b ≣
  if a < b then b else a
```c
int gcd(int a, int b) {
    int c;
    while (a != 0) {
        c = a;
        a = b % a;
        b = c;
    }
    return b;
}
```
int gcd(int a, int b) {
    int c;
    while (a != 0) {
        c = a;
        a = b % a;
        b = c;
    }
    return b;
}
int gcd(int a, int b) {
    int c;
    while (a != 0) {
        c = a;
        a = b % a;
        b = c;
    }
    return b;
}
void swap(int *a, int *b)
{
    int t = *a;
    *a = *b;
    *b = t;
}
void swap(int *a, int *b) {
    int t = *a;
    *a = *b;
    *b = t;
}

swap ≡
TRY
  GUARD C_Guard ⦃c_guard ´ a⦄   (´ t :== h_val (hrs_mem ´ t_hrs) ´ a);;
  GUARD C_Guard ⦃c_guard ´ a⦄   (GUARD C_Guard ⦃c_guard ´ b⦄   (´ globals :==
      t_hrs_'_update
       (hrs_mem_update
         (heap_update ´ b ´ t)))));
  GUARD C_Guard ⦃c_guard ´ b⦄   (´ globals :==
      t_hrs_'_update
       (hrs_mem_update
         (heap_update ´ b ´ t)))
CATCH
  SKIP
END
```c
void swap(int *a, int *b)
{
    int t = *a;
    *a = *b;
    *b = t;
}
```

```ocaml
swap a b ≡
do guard (λs. c_guard a);
    t ← gets (λs. h_val (hrs_mem (t_hrs_ s)) a);
    guard (λs. c_guard b);
    modify
      (λs. t_hrs_ _update
       (hrs_mem_update
         (heap_update a
          (h_val (hrs_mem (t_hrs_ s)) b))) s);
    modify
      (t_hrs_ _update
       (hrs_mem_update (heap_update b t)))
od
```
C Heap Semantics

```c
void swap(int *a, int *b)
{
    int t = *a;
    *a = *b;
    *b = t;
}
```
C Heap Semantics

```c
void swap(int *a, int *b)
{
    int t = *a;
    *a = *b;
    *b = t;
}
```
C Heap Semantics

```c
void swap(int *a, int *b)
{
    int t = *a;
    *a = *b;
    *b = t;
}
```

C Heap Semantics

```c
void swap(int *a, int *b)
{
    int t = *a;
    *a = *b;
    *b = t;
}
```
swap a b ≡
  do guard (λs. c_guard a);
     t ← gets (λs. h_val (hrs_mem (t_hrs_’ s)) a);
     guard (λs. c_guard b);
     modify
      (λs. t_hrs_’_update
        (hrs_mem_update
          (heap_update a
            (h_val (hrs_mem (t_hrs_’ s)) b)))) s);
     modify
      (t_hrs_’_update
        (hrs_mem_update (heap_update b t)))
  od
swap a b ≡
  do guard (λs. c_guard a);
     t ← gets (λs. h_val (hrs_mem (t_hrs_’ s)) a);
    guard (λs. c_guard b);
   modify
     (λs. t_hrs_’_update
       (hrs_mem_update
         (heap_update a
           (h_val (hrs_mem (t_hrs_’ s)) b)))) s);
   modify
     (t_hrs_’_update
       (hrs_mem_update (heap_update b t)))
od

Ensure “a” is aligned, non-NULL.
swap a b ≡
  do guard (λs. c_guard a);
     t ← gets (λs. h_val (hrs_mem (t_hrs_' s)) a);
     guard (λs. c_guard b);
     modify
      (λs. t_hrs_'_update
       (hrs_mem_update
        (heap_update a
         (h_val (hrs_mem (t_hrs_' s)) b)))) s);
     modify
      (t_hrs_'_update
       (hrs_mem_update (heap_update b t)))
  od

Ensure “a” is aligned, non-NULL.

Decode the bytes at “a”.

Ensure “a” is aligned, non-NULL.
swap a b ≡
  do guard (λs. c_guard a);
     t ← gets (λs. h_val (hrs_mem (t_hrs_' s)) a);
     guard (λs. c_guard b);
     modify
      (t_hrs_'_update
        (hrs_mem_update (heap_update a
          (h_val (hrs_mem (t_hrs_' s)) b))) s);
     modify
      (t_hrs_'_update
        (hrs_mem_update (heap_update b t)))
  od

Ensure “a” is aligned, non-NULL.

Decode the bytes at “a”.

Decode the bytes at “b”, encode back into bytes, store at “a”.
In Short

Treatment of Heap is Promising Work in Progress

Word Arithmetic Made More Beautiful
– having to worry about overflow is drudgery...
– make the tools do it as automatically as possible
Conclusion

Ad hoc **Proof** makes ad hoc hackery
- sound, and
- beautiful

**Tools:**