Formalizing a CAS(n) Algorithm in HOL

Joe Hurd
joe.hurd@cl.cam.ac.uk

University of Cambridge

Contents

What Does a CAS(n) Algorithm Do?

- Context is many processors and a shared memory.
- CAS(n) stands for Multiple Compare And Swap:

  if the n memory addresses $a_1, \ldots, a_n$
  contain the expected values $x_1, \ldots, x_n$
  then replace them with the values $y_1, \ldots, y_n$

- Many processors could be concurrently executing CAS(n), with potentially overlapping memory addresses.
- The important thing is that the operation must act atomically—useful for cleanly updating data-structures in a multi-threaded environment.

Primitive Atomic Operations

In this talk we assume the following primitive atomic operations:

- Many instruction sets include an atomic CAS(1):

  if the memory address a
  contains the expected value x
  then replace it with the value y

  CAS(1) returns the value that it found in the address a.
- Also assume that memory reads and writes are atomic, as well as a malloc operation to obtain fresh storage.

CAS(n) Algorithms

- There are many implementations of the CAS(n) algorithm.
- They differ in their primitive atomic operations, run-time performance and additional space requirements.
- They also have different specifications, according to their interpretation of "*the operation must act atomically*". We'll discuss this later.
- We have formalized the CAS(n) algorithm developed by Harris, Fraser and Pratt in Cambridge.
- From now on: CAS(n) refers to the CAS(n) implementation of Harris et. al.

Main Challenges for Formalization

Creating a logical model of a parallel architecture, supporting:

- independent execution of many processors communicating only through a shared memory;
- arbitrary interleaving of instructions at the granularity of the primitive atomic operations;
- higher-level constructs such as recursion;
- and a tidy way of initializing each processor.

This is where we are now. Future work will concentrate on support for specification and verification within this model.

Contents

Memory Model: Interface

Want a model of memory with the following interface:

$$
\begin{array}{rcl}
\text{minit} & : & \alpha^* \rightarrow (\alpha)\mathit{memory} \\
\text{mread} & : & (\alpha)\mathit{memory} \rightarrow \mathbb{N} \rightarrow \alpha \\
\text{mwrite} & : & (\alpha)\mathit{memory} \rightarrow \mathbb{N} \rightarrow \alpha \rightarrow (\alpha)\mathit{memory} \\
\text{malloc} & : & (\alpha)\mathit{memory} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \times (\alpha)\mathit{memory} \\
\text{mcas} & : & (\alpha)\mathit{memory} \rightarrow \mathbb{N} \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha
\end{array}
$$

Memory Model: Implementation

The underlying type is $(\alpha)\mathit{memory} = \alpha^*$.

$$
\begin{array}{rcl}
\text{minit } l & = & l \\
\text{mread } m\ l & = & \text{nth } l\ (\text{mextend\_to } m\ l) \\
\text{mwrite } m\ l\ n & = & \text{update\_nth } l\ (\text{K } n)\ (\text{mextend\_to } m\ l) \\
\text{malloc } m\ n & = & (\text{length } m,\ \text{mextend\_by } m\ n) \\
\text{mcas } m\ l\ e\ n & = & \text{let } a \leftarrow \text{mread } m\ l \\
 & & \text{in } (a,\ \text{if } a = e \text{ then mwrite } m\ l\ n \text{ else } m)
\end{array}
$$

The worker functions extend the memory as required:

$$
\begin{array}{rcl}
\text{mextend\_by } m\ n & = & \text{append } m\ (\text{klist arb } n) \\
\text{mextend\_to } m\ n & = & \text{if } n < \text{length } m \text{ then } m \\
 & & \text{else mextend\_by } m\ (\text{suc } n - \text{length } m)
\end{array}
$$

Parallel Architecture: Overview

- Identify a processor with its register file:

$$\mathit{proc} = \mathbb{S} \rightarrow \mathbb{N}$$

- The whole parallel architecture can then be modelled by a list of processors and a shared memory:

$$\mathit{arch} = (\mathbb{N})\mathit{memory} \times \mathit{proc}^*$$

- The global state advances by any non-halted processor executing a primitive atomic instruction.
- Everything happens with respect to a global program (not stored in the shared memory!) and a *"pc"* register in each processor.

Parallel Architecture: Instruction Set

An *instruction* corresponds to a primitive atomic operation:

$$
\begin{array}{rcl}
\textit{operation} & = & \mathsf{NOP} \\
& | & \mathsf{UP\ of}\ \mathbb{N} \times \mathbb{S} \\
& | & \mathsf{RD\ of}\ \mathbb{N} \times \mathbb{S} \\
& | & \mathsf{WR\ of}\ \mathbb{S} \times \mathbb{N} \\
& | & \mathsf{CAS\ of}\ \mathbb{N} \times \mathbb{S} \times \mathbb{S} \times \mathbb{S} \\
& | & \mathsf{ALLOC\ of}\ \mathbb{N} \times \mathbb{S}
\end{array}
$$

$$
\begin{array}{rcl}
\textit{instruction} & = & \mathsf{LAB\ of}\ \mathbb{S} \\
& | & \mathsf{INS\ of}\ \textit{labels} \rightarrow \textit{proc} \rightarrow \textit{operation}
\end{array}
$$

(*labels* $= \mathbb{S} \rightarrow \mathbb{N}$ stores the program labels.)

Parallel Architecture: Local Steps

- The machine_step function executes one instruction of a processor, updating the register file and the shared memory:

$$\begin{aligned}&\text{machine\_step } prog\ mem\ reg = \\ &\quad \text{let } pc \leftarrow reg \text{ “pc” in} \\ &\quad \text{let } reg' \leftarrow \text{update “pc” (suc } pc) \ reg \text{ in} \\ &\quad \text{interpret (labels } prog)\ mem\ reg'\ (\text{nth } pc\ prog)\end{aligned}$$

- A relation is used to account for halting:

$$\begin{aligned}&\text{local\_step } prog\ (m, r)\ (m', r') = \\ &\quad \neg\text{halted } prog\ r \wedge ((m', r') = \text{machine\_step } prog\ m\ r)\end{aligned}$$

Parallel Architecture: Global Steps

- This is the definition of the global step relation:

$$\begin{aligned}
&\mathsf{global_step}\ prog\ (m,p)\ (m',p') = \\
&\quad \mathsf{length}\ p' = \mathsf{length}\ p\ \wedge \\
&\quad \exists\, x. \\
&\qquad x < \mathsf{length}\ p\ \wedge \\
&\qquad (\forall\, y.\ y < \mathsf{length}\ p \wedge y \neq x \Rightarrow \mathsf{nth}\ y\ p = \mathsf{nth}\ y\ p')\ \wedge \\
&\qquad \mathsf{local_step}\ prog\ (m, \mathsf{nth}\ x\ p)\ (m', \mathsf{nth}\ x\ p')
\end{aligned}$$

- Intuitively, a global step is simply a local step in one of the processors.

Parallel Architecture: Global Steps

- For simulation, we prefer the following:

$$
\begin{array}{ll}
\vdash & \mathsf{next}\ (\mathsf{global\_step}\ prog)\ (m, p) = \\
& \mathsf{snd} \\
& (\mathsf{foldl} \\
& \quad (\lambda\,(n, s), r. \\
& \quad\quad (n + 1, \\
& \quad\quad\quad (\mathsf{if}\ \mathsf{halted}\ prog\ r\ \mathsf{then}\ s\ \mathsf{else} \\
& \quad\quad\quad\quad (\mathsf{I}\ \#\#\ (\lambda\, r'.\ \mathsf{update\_nth}\ n\ (\mathsf{K}\ r')\ p)) \\
& \quad\quad\quad\quad (\mathsf{machine\_step}\ prog\ m\ r)\ \mathsf{insert}\ s))))\ (0, \{\})\ p)
\end{array}
$$

- Grungy looking RHS, but executes quickly in the logic.
- It also avoids reasoning 'inside set comprehensions'.

Contents

Higher-Level Programming Constructs

- Problem: CAS(n) algorithm expressed as 38 lines of C-like pseudo-code, including high-level constructs such as:
  - function calls;
  - recursion;
  - structs

  but we have extremely primitive instruction set.
- Solution: Write macro instructions (of type $instruction^*$) that implement higher-level constructs.
- A compilation phase reduces programs to low-level instructions, so get proper interleaving behaviour.
- But source code of programs is possible to read (and debug!).

Higher-Level Programming Constructs

Some examples of instruction macros:

JLR $l =$

[INS ($\lambda\, labs, reg.$ UP (suc (reg *"pc"*)) *"link"*);

INS ($\lambda\, labs, reg.$ UP ($labs\ l$) *"pc"*]

PUSH $rs =$

flat

(append

[MALLOC (suc (LENGTH rs)) *"Z"*; ST *"stack"* *"Z"*; MV *"Z"* *"stack"*]

(append

(flat (map ($\lambda\, r.$ [INC *"Z"*; ST r *"Z"*]) rs)) [ZAP [*"Z"*]])

CALL $rs\ l =$

append (PUSH (*"link"* :: rs)) (append (JLR l) (POP (*"link"* :: rs)))

Implementing CAS(n)

Can now implement the functions of CAS(n):

```
[LABEL "rdcss";
 MV "arg0" "d";                                    ZAP ["arg0"];
 LABEL "rdcss-1";
 VAL "d" "dv";
 LDM "dv" ["a1"; "o1"; "a2"; "o2"];                ZAP ["a1"; "o1"; "dv"];
 VAL "a2" "a2v";                                   ZAP ["a2"];
 CAS1 "a2v" "o2" "d" "r";                          ZAP ["a2v"];
 BR (\x. ~is_Descriptor x) "r" "rdcss-2";
 MV "r" "arg0";
 CALL ["d"] "complete";
 JMP "rdcss-1";
 LABEL "rdcss-2";
 BR2 (\x y. ~(x = y)) "r" "o2" "rdcss-3";
 MV "d" "arg0";
 CALL ["r"] "complete";
 LABEL "rdcss-3";                                  ZAP ["d"; "o2"];
 MV "r" "result";                                  ZAP ["r"];
 RETURN]`;
```

Contents

- Introduction
- Formalizing a Parallel Architecture
- Formalizing CAS(n)
- **Simulating CAS(n)**
- Statement of Correctness
- Conclusion

Contents

- Introduction
- Formalizing a Parallel Architecture
- Formalizing CAS(n)
- Simulating CAS(n)
- **Statement of Correctness**
- Conclusion

Statement of Correctness

- First attempt:
  - when CAS(n) algorithm executed on only one processor, it “does the right thing”;
  - when CAS(n) algorithm is executed by many processors, the result is the same as if they had executed sequentially in some order.
- But the implementation of CAS(n) we have formalized satisfies a stronger property: it is linearizable.
- Would like to formalize this by introducing notion of time into the model.

Contents

- Introduction
- Formalizing a Parallel Architecture
- Formalizing CAS(n)
- Simulating CAS(n)
- Statement of Correctness
- **Conclusion**

Conclusion

- We have shown that it's possible to formalize low-level parallel algorithms in HOL.
- Remains to be seen how easy it will be to specify and verify them: this is the next item on the agenda!
- Would also like to extend the memory model to include memory barriers (c.f. Gordon's model of Alpha architecture), and verify a more realistic version of the CAS(n) algorithm.