Formalizing a CAS(n) Algorithm in HOL

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

University of Cambridge
Contents

- Introduction
  - Formalizing a Parallel Architecture
  - Formalizing CAS(n)
  - Simulating CAS(n)
  - Statement of Correctness
  - Conclusion
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.
In this talk we assume the following primitive atomic operations:

- Many instruction sets include an atomic CAS(1):
  
  \[
  \text{if the memory address } a \\
  \text{contains the expected value } x \\
  \text{then replace it with the value } y
  \]

  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

- Introduction
- **Formalizing a Parallel Architecture**
  - Formalizing CAS(n)
  - Simulating CAS(n)
  - Statement of Correctness
  - Conclusion
Want a model of memory with the following interface:

\[
\begin{align*}
\text{minit} : & \quad \alpha^* \rightarrow (\alpha)\text{memory} \\
\text{mread} : & \quad (\alpha)\text{memory} \rightarrow \mathbb{N} \rightarrow \alpha \\
\text{mwrite} : & \quad (\alpha)\text{memory} \rightarrow \mathbb{N} \rightarrow \alpha \rightarrow (\alpha)\text{memory} \\
\text{malloc} : & \quad (\alpha)\text{memory} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \times (\alpha)\text{memory} \\
\text{mcas} : & \quad (\alpha)\text{memory} \rightarrow \mathbb{N} \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha
\end{align*}
\]
Memory Model: Implementation

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

\[
\begin{align*}
\text{minit} \ l & = \ l \\
\text{mread} \ m \ l & = \ \text{nth} \ l \ (\text{mextend}\_\text{to} \ m \ l) \\
\text{mwrite} \ m \ l \ n & = \ \text{update}\_\text{nth} \ l \ (K \ n) \ (\text{mextend}\_\text{to} \ m \ l) \\
\text{malloc} \ m \ n & = \ (\text{length} \ m, \ \text{mextend}\_\text{by} \ m \ n) \\
\text{mcas} \ m \ l \ e \ n & = \ \text{let} \ a \leftarrow \ \text{mread} \ m \ l \\
& \hspace{1cm} \text{in} \ (a, \ \text{if} \ a = e \ \text{then} \ \text{mwrite} \ m \ l \ n \ \text{else} \ m)
\end{align*}
\]

The worker functions extend the memory as required:

\[
\begin{align*}
\text{mextend}\_\text{by} \ m \ n & = \ \text{append} \ m \ (\text{klist} \ \text{arb} \ n) \\
\text{mextend}\_\text{to} \ m \ n & = \ \text{if} \ n < \ \text{length} \ m \ \text{then} \ m \\
& \hspace{1cm} \text{else} \ \text{mextend}\_\text{by} \ m \ (\text{suc} \ n - \ \text{length} \ m)
\end{align*}
\]
Parallel Architecture: Overview

- Identify a processor with its register file:

\[ \text{proc} = \mathbb{S} \rightarrow \mathbb{N} \]

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

\[ \text{arch} = (\mathbb{N}) \text{memory} \times \text{proc}^* \]

- 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:

\[\text{operation} = \text{NOP} \mid \text{UP of } \mathbb{N} \times S \mid \text{RD of } \mathbb{N} \times S \mid \text{WR of } S \times \mathbb{N} \mid \text{CAS of } \mathbb{N} \times S \times S \times S \mid \text{ALLOC of } \mathbb{N} \times S\]

\[\text{instruction} = \text{LAB of } S \mid \text{INS of } \text{labels} \rightarrow \text{proc} \rightarrow \text{operation}\]

(labels = S → 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:

  ```
  machine_step prog mem reg =
  let pc ← reg "pc" in
  let reg' ← update "pc" (suc pc) reg in
  interpret (labels prog) mem reg' (nth pc prog)
  ```

- A relation is used to account for halting:

  ```
  local_step prog (m, r) (m', r') =
  ¬halted prog r ∧ ((m', r') = machine_step prog m r)
  ```
Parallel Architecture: Global Steps

- This is the definition of the global step relation:

\[
\text{global}_\text{step} \ prog \ (m, p) \ (m', p') = \\
\text{length} \ p' = \text{length} \ p \land \\
\exists x. \\
\quad x < \text{length} \ p \land \\
\quad (\forall y. \ y < \text{length} \ p \land y \neq x \Rightarrow \text{nth} \ y \ p = \text{nth} \ y \ p') \land \\
\text{local}_\text{step} \ prog \ (m, \text{nth} \ x \ p) \ (m', \text{nth} \ x \ p')
\]

- Intuitively, a global step is simply a local step in one of the processors.
Parallel Architecture: Global Steps

- For simulation, we prefer the following:

\[
\text{next (global\_step \textit{prog}) (m, p) = snd}
\]

\[
(\text{foldl}
\]

\[
(\lambda (n, s), r.
\]

\[
(n + 1,\]

\[
(\text{if halted \textit{prog} r then } s \text{ else}
\]

\[
(I\#\# (\lambda r'. \textit{update\_nth} n (K r') p))
\]

\[
(\text{machine\_step \textit{prog} m r} \text{ insert } s))) (0, \{\}) p)
\]

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

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

---

Formalizing a CAS(n) Algorithm in HOL – Joe Hurd – p.16/23
Higher-Level Programming Constructs

Some examples of instruction macros:

\[ \text{JLR } l = \]
\[ [\text{INS } (\lambda \text{labs, reg. UP (suc (reg "pc")) "link"}]); \]
\[ \text{INS } (\lambda \text{labs, reg. UP (labs l) "pc"}] \]

\[ \text{PUSH } rs = \]
\[ \text{flat} \]
\[ (\text{append} \]
\[ [\text{MALLOC (suc (LENGTH } rs)) "Z"; \text{ST } "stack" "Z"; \text{MV } "Z" "stack"] \]
\[ (\text{append} \]
\[ (\text{flat (map (\lambda r. [\text{INC } "Z"; \text{ST } r "Z"] } rs)) [\text{ZAP ["Z"]}]\]

\[ \text{CALL } rs l = \]
\[ \text{append (PUSH ("link :: rs)) (append (JLR } l) (\text{POP ("link :: rs"))}) \]
Implementing CAS(n)

Can now implement the functions of CAS(n):

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

Formalizing a CAS(n) Algorithm in HOL – Joe Hurd – p.18/23
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.