Operational reasoning for concurrent Caml programs and weak memory models

Tom Ridge

University of Cambridge

Abstract. This paper concerns the formal semantics of programming languages, and the specification and verification of software. We are interested in the verification of real programs, written in real programming languages, running on machines with real memory models. To this end, we verify a Caml implementation of a concurrent algorithm, Peterson's mutual exclusion algorithm, down to the operational semantics. The implementation makes use of Caml features such as higher order parameters, state, concurrency and nested general recursion. Our Caml model includes a datatype of expressions, and a small step reduction relation for programs (a Caml expression together with a store). We also develop a new proof of correctness for a modified version of Peterson's algorithm, designed to run on a machine with a weak memory.

1 Introduction

Program verification has a long and venerable history, from the pioneering work of Turing [MJ84], through to recent work such as the mechanized verification of the Four Colour Theorem [Gon05]. However, there are problems with existing approaches.

- The languages and programs are typically highly idealized, and far removed from the real languages people actually use. Attempts to address real languages are typically not foundational.
- With the advent of multicore processors, assumptions about how to model memory are no longer justified, and existing proofs are typically invalid in the case that the algorithm is running on a machine with a weak memory.

We explore the issues involved by conducting two verifications: for the first, we verify a non-trivial concurrent Caml program, Peterson's algorithm for mutual exclusion [Pet81], down to the operational semantics; for the second we verify Peterson's algorithm with a weak memory model.

The Caml code we verify appears in Fig. 1 and can be compiled and run with recent versions of OCaml[1] (for readers not familiar with Caml syntax, we describe a standard pseudocode version in Sect. 2). We formalize a semantics for Caml in Sect. 3 and the proof in Sect. 4. The complexity that arises in the

[1] http://caml.inria.fr

```
let turn = ref false in
let trys =
  let try_f = ref false in
  let try_t = ref false in
    fun x -> if x then try_t else try_f in
let rec loop = fun id ->
  (trys id := true)                                    (* set trying flag *)
  ; (turn := not id)                                   (* set turn flag   *)
  ; (let rec test = fun _ ->
      if !turn = id || not !(trys (not id)) then () else test ()
     in test ())                                       (* looping in test *)
  ; ()                                                 (* critical region *)
  ; (trys id := false)                                 (* exit crit       *)
  ; loop id
in
(Thread.create loop false, Thread.create loop true)
```

Fig. 1. Peterson's algorithm, Caml code

proofs stems from the size of individual states (Caml expressions that occupy several pages), the large number of reachable states for each thread , and the consequent intricacy of the interleavings of the two threads. Reasoning directly about the operational semantics requires clever proof techniques to mitigate these complexities. We briefly outline these techniques, which we describe in greater detail in Sect. 4.

- We factor the proof into an abstract part and an operational part. We isolate the core interleavings of the threads by constructing an abstract model of Peterson's algorithm. The Caml implementation is a refinement of this abstract model.
- The Caml part of the proof is an invariant proof. We use the theorem prover to symbolically execute along paths of the system, proving the invariant along the way.
- We employ a strong induction hypothesis, which makes use of a notion of "recurrent states". We consider paths starting from these recurrent states, and whenever we encounter another recurrent state, we apply the induction hypothesis, and so terminate symbolic execution along that particular branch. The induction scheme, with the notion of recurrent states, is useful even for non-concurrent, infinite state systems.
- To reduce the complexity of the state space of the parallel composition of the two threads, $T_1|T_2$, we replace a thread, T_2 say, with an invariant I_2, which represents possible interference of T_2, but is much easier to work with. We also consider $I_1|T_2$. This is similar to rely-guarantee reasoning [Jon81], and allows us to perform a thread-modular verification. The rely-guarantee style reasoning extends smoothly to multiple threads.
- In order for the induction to work smoothly we eliminate the interleaving between T_1 and I_2, so that T_1 is always making progress. We construct a

thread-like object T_1', that executes roughly as T_1, but incorporates possible interference from I_2 into each T_1 step. It is the system T_1' that we symbolically execute.

These techniques allow us to reduce the Caml proof to little more than simplification, which we use to symbolically execute the thread-like objects, and to check the invariant at each stage. In fact, the proof could be entirely automated. *Moreover, the techniques apply directly to similar algorithms such as Simpson's 4-slot for which we already have abstract proofs [Rid].* We claim that our approach is reproducible and can be applied to many other algorithms: given an abstract proof of an algorithm, proving the connection to a Caml (or other) implementation can often be reduced to little more than symbolic execution using the theorem prover's simplifier.

To address the second problem, we demonstrate reasoning about Peterson's algorithm at an abstract level with a very weak memory model. This requires us to formalize a model of weak memory (Sect. 6), modify Peterson's algorithm, and to develop a new proof of correctness (Sect. 7). In essence, the proof of Peterson's algorithm with a weak memory model follows the outline of the original abstract proof, but includes proof patches to cope with the additional complexities. We include background information on weak memory models in Sect. 5.

Mechanization of these proofs is vital, especially as we move towards verification of realistic full scale languages: the reachable states of the operational semantics are simply too complex for informal proof. Moreover, reasoning about concurrent algorithms even without weak memory models is considered difficult. With the additional complexity of weak memory models, we found mechanization to be invaluable. For example, the process of constructing the proof revealed several possible execution paths that we were not aware of, and which required us to alter our proofs.

Our contributions are:

- We develop a mechanized verification of a Caml implementation of Peterson's algorithm, down to the operational semantics.
- The Caml implementation is a refinement of an abstract model of Peterson's algorithm. *This is important because there are many abstract verifications which we would like to reuse for more complex program models.*
- We develop some novel proof techniques which we have sketched above, and which we describe more fully in Sect. 4.
- We develop a mechanized proof of correctness for a version of Peterson's algorithm, with a weak memory model.

Small-step operational semantics are the most successful means of language specification available. In principle, it is clear that operational reasoning is sufficient to prove the correctness of programs. If the language and programs are specified by means of operational semantics, the *most direct form of proof is by using operational reasoning.* The motivation of this work is to promote operational reasoning as a means of program verification.

```
trys[0]   = 0
trys[1]   = 0
turn      = 0
```

```
P0: trys[0] = 1
    turn = 1
    while( trys[1] && turn == 1 );
            // do nothing
    // critical section
    ...
    // end of critical section
    trys[0] = 0
```

```
P1: trys[1] = 1
    turn = 0
    while( trys[0] && turn == 0 );
                  // do nothing
    // critical section
    ...
    // end of critical section
    trys[1] = 0
```

Fig. 2. Peterson's algorithm, informal pseudocode

Basic definitions Given an underlying set of states S, a *state transition system* or *sts* on S is a pair consisting of a set of *start states* $\subseteq S$, and a *transition relation* $\subseteq S \times S$. Given an sts L, a *sequence* p is a function from $\mathbb{N}$ to S, such that adjacent points $p\ n, p\ (n+1)$ are related by the transition relation of L. A *path* is a sequence that starts in a start state. A state a is *reachable* if there is a path p, and position n, such that $p\ n = a$. If L and K are defined on the same set of underlying states S, the *parallel composition* of L and K is the sts formed by taking the union of the start states, and the union of the transition relations. An *abstraction relation* R from an sts L to an sts K relates each start state of L to some start state of K, and if a is reachable by L, and related to b, and L makes a transition from a to a', then K can make a transition from b to b', and moreover a' and b' are also related by R. The key fact about an abstraction relation is that for every path of L, there is a path of K such that corresponding positions in the paths are related. Abstraction relations are a form of simulation relations.

Isabelle/HOL The mechanizations described here were conducted in the Isabelle/HOL theorem prover. List cons is represented in Isabelle syntax by an infix #, and updating a function f at argument a with value v is written $f(a := v)$. Isabelle also has two implications, $\longrightarrow$ and $\Longrightarrow$. For the purposes of this paper, they may be considered equivalent. Apart from that, Isabelle's syntax is close to informal mathematics.

2 Background on Peterson's algorithm

In this section, we describe Peterson's algorithm informally using pseudocode, give a formal model of the abstract algorithm in HOL, give the key lemmas needed to prove the mutual exclusion property, and state the mutual exclusion property itself.

In Fig. 1 we give the Caml version of Peterson's algorithm that we verify. By way of introduction, we here present an informal pseudocode version, Fig. 2, and explain how it works. The algorithm uses two variables, `trys` (an array

```
peterson-trans i s s' ≡

(* j is the other process *)
let j = ¬ i in

(* extract parts of the state *)
let ((turn,trys),pcs) = s in
let ((turn',trys'),pcs') = s' in
let (itry,ipc) = (trys i, pcs i) in
let (itry',ipc') = (trys' i, pcs' i) in
let (jtry, jpc) = (trys j, pcs j) in
let (jtry', jpc') = (trys' j, pcs' j) in

(* nothing happens to j *)
(jpc' = jpc ∧ jtry' = jtry)

(* process i takes a step *)
∧ (
```

```
(* looping in any state *)
((ipc',itry',turn') = (ipc,itry,turn))

∨ ((ipc,ipc') = (NonCrit,SetTry)
  ∧ (itry',turn') = (itry,turn))

∨ ((ipc,ipc') = (SetTry,SetTurn)
  ∧ (itry',turn') = (True,turn))

∨ ((ipc,ipc') = (SetTurn,Test)
  ∧ (itry',turn') = (itry,j))

∨ ((ipc,ipc') = (Test,Crit)
  ∧ ((jtry = False) ∨ (turn = i))
  ∧ (itry',turn') = (itry,turn))

∨ ((ipc,ipc') = (Crit,Exit)
  ∧ (itry',turn') = (itry,turn))

∨ ((ipc,ipc') = (Exit,NonCrit)
  ∧ (itry',turn') = (False,turn)))
```

Fig. 3. Thread transitions for Peterson's algorithm, in HOL

of booleans) and `turn`. A `trys[0]` value of 1 indicates that thread `P0` wants to enter the critical section, similarly `trys[1] = 1` indicates that `P1` wants to enter the critical section. The variable `turn` holds the id of the thread whose turn it is. Entrance to the critical section is granted for `P0` if `P1` does not want to enter its critical section or if `P1` has given priority to `P0` by setting `turn` to 0. The algorithm guarantees that the two threads are never executing their critical sections simultaneously.

It is straightforward to formalize an abstract state transition system for Peterson's algorithm. For each thread, the *program counter*, or *pc*, is either non-critical, about to set the try variable, about to set the turn variable, testing the turn variable and the other thread's try variable, in the critical section, or exiting the critical section by unsetting the try variable.

datatype *pc* = *NonCrit* | *SetTry* | *SetTurn* | *Test* | *Crit* | *Exit*

We identify *thread id* with a boolean rather than an integer.The *turn* and *try* variables are booleans (*try* is indexed by thread id). The system state is a tuple consisting of the turn and try variables, and the program counters for each thread. The transition relation for a thread is parameterised on the thread id.

types *thread-id* = *bool* **types** *turn* = *thread-id* **types** *trys* = (*thread-id* ⇒ *bool*)
types *pcs* = (*thread-id* ⇒ *pc*) **types** *peterson-state* = (*turn* ∗ *trys*) ∗ *pcs*

constdefs *peterson-trans* :: *thread-id* ⇒ *peterson-state* ⇒ *peterson-state* ⇒ *bool*

We define the transitions in Fig. 3. Each thread starts in its non-critical state, with all variables initialised to false. The state transition system is a parallel composition of the two threads.

constdefs *peterson-starts* :: *peterson-state set*
peterson-starts ≡ {((*False*,λ *x*. *False*),λ *x*. *NonCrit*)}

constdefs *peterson-sts* :: *peterson-state sts*
peterson-sts ≡
par (*peterson-starts*, *peterson-trans False*) (*peterson-starts*, *peterson-trans True*)

We first characterise the thread states where the try variable is set, following Lemma 10.5.1 in [Lyn96]. The key lemma, *lemma-10-5-2*, is proved by induction on n. Mutual exclusion follows directly, since *turn* cannot be both i and j.

lemma *lemma-10-5-2*:
∀ *i p*. *is-path peterson-sts p*
⟶ (∀ *n*.
let ((*turn*,*trys*),*pcs*) = *p n in*
let (*itry*,*ipc*) = (*trys i*, *pcs i*) *in*
let j = ¬*i in*
let (*jtry*,*jpc*) = (*trys j*, *pcs j*) *in*
ipc = *Crit* ∧ *jpc* ∈ {*Test*,*Crit*}
⟶ (*turn* = *i*))

lemma *mutual-exclusion*:
∀ *p*. *is-path peterson-sts p*
⟶ (∀ *n i*.
let ((*turn*,*trys*),*pcs*) = *p n in*
let (*itry*,*ipc*) = (*trys i*, *pcs i*) *in*
let j = ¬*i in*
let (*jtry*,*jpc*) = (*trys j*, *pcs j*) *in*
¬ (*ipc* = *Crit* ∧ *jpc* = *Crit*))

3 Caml formalization

In this section we give the datatype of Caml expressions, describe the small-step operational semantics for Caml, and give the state transition system for the Caml implementation of Peterson's algorithm.

If we examine the code in Fig. 1, we see that we need to model several features of Caml: general recursion, nested recursion, state, higher order parameters, and concurrency. We do not model features, such as modules, that are not required for our example. Adding additional features would not essentially alter the proof, however, which rests largely on symbolic evaluation of the code.

We declare a datatype of Caml expressions in Isabelle/HOL. This includes the usual functional core (including *let*), constants (including *fix*), state (assignment, reference, dereference, and locations) and concurrency (*Par*). A trace expression is not part of the standard Caml syntax. It is semantically neutral (i.e. *Trace(a,e)* behaves as e), and it removes the need to quote Caml expressions explicitly.

```
datatype 'a exp =

    Var var
  | Lam 'a lambda
  | App 'a exp * 'a exp
  | LetVal 'a exp * 'a lambda

  | Const const
  | Assign 'a exp * 'a exp
  | Deref 'a exp
  | Ref 'a exp
  | Loc loc

  | Par 'a exp * 'a exp

  | Trace 'a * 'a exp

and 'a lambda = Lambda var * 'a exp
```

We use the `-dparsetree` OCaml compiler switch to obtain an abstract syntax tree of the Caml code, which we import into Isabelle/HOL, modulo some pretty printing of the thread pair as a *Par*. We first modify the Caml code by tagging subexpressions with trace information, in particular, we modify the `()` expression in the critical section to `Trace("Crit",())`.

```
constdefs peterson :: string exp
 peterson ≡
  LetVal(Trace(''turn'',Ref (Const FFalse)),Lambda(''turn'',LetVal(Trace(''trys'',...
```

We need to model how Caml expressions evaluate or reduce. Expressions reduce in the context of a store. Our store is a pair (n, w) where n is the next free location in the store w, a map from locations to expressions. Our reduction relation is then a relation between (n, w, e), the current store and expression, with (n', w', e'), the store and expression at the next step. To make symbolic evaluation easier, we phrase this as a function *reduce*′ from (n, w, e) to a list of possible next states, which we later lift to a relation *rreduce*. We give an excerpt from this definition, illustrating that variables and lambdas do not reduce, and the reduction of a function value (lambda expression) e_1 applied to a value e_2 is by substituting $e2$ in the body of the function.

```
primrec
  reduce' n w (Var v) = []
  reduce' n w (Lam l) = []
  reduce' n w (App e1e2) = reduce'-3 n w e1e2
  reduce'-3 n w (e1,e2) = (
    if is-value e1 then (
      if is-value e2 then (
        if is-Lam e1 then [(n,w,subst-lambda e2 (dest-Lam e1))]
        else ...)))
```

We combine the Caml definition of Peterson's algorithm with the Caml reduction relation to get a state transition system for the Caml implementation of Peterson's algorithm.

types *caml* = *loc* ∗ (*loc* ⇒ *string exp*) ∗ *string exp*

constdefs *peterson-starts* :: *caml set*
peterson-starts ≡ {(*loc-counter*, λ *x*. *null-exp*, *peterson*)}

constdefs *peterson-sts* :: *caml sts*
peterson-sts ≡ (*peterson-starts*, *rreduce*)

4 Proof of correctness

In this section, we outline the proof of correctness. The difficulty of the proof arises from the complexity of the state space. For example, in the abstract model, we can show a property is invariant by case splitting on a thread's program counter.For the Caml thread, which is represented as a Caml expression, case splitting does not make sense, and we have to find some other way to characterise the possible states the thread might take.

Given that we have added trace information to identify the critical section, our statement of correctness informally says that it is not the case that the Caml expression is a *Par* of two threads executing in their critical section. A thread is represented as a Caml sequence of the current expression, and the remaining expressions to execute. A sequence is just a *let* of the current expression, and the rest of the sequence as the body of the *let*. This gives rise to the following statement of correctness.

lemma *mutual-exclusion*:
∀ *p n*. *is-path peterson-sts p* ⟶ ¬ (∃ *e1 l1 e2 l2*. *let* (*n*,*w*,*e*) = *p n in*
e = *Par* (*LetVal* (*Trace* (''*Crit*''*,e1*)*,l1*), *LetVal* (*Trace* (''*Crit*''*,e2*)*,l2*)))

The proof proceeds by constructing an abstraction relation from the Caml sts to the abstract sts.

lemma *is-abstraction*: *is-abstraction PetersonML.sts Peterson2.peterson-sts R*

The abstraction relates Caml states matching $Trace(''Crit'', e)$ to abstract states with program counter *Crit*. If the Caml threads were both in their critical section, then by the main property of abstraction relations, so too would the two abstract threads. Since we have shown that this is impossible, we get mutual exclusion for the Caml implementation. To show that R is indeed an abstraction relation, we have to show that, for reachable states of the Caml system, transitions can be matched by transitions of the abstract system. That is, we have to show an invariant of all reachable states.

The main barrier to this approach is the complexity of the reachable states of the Caml implementation. Intermediate program expressions can grow substantially larger than the original program (because *let rec* duplicates a body of code, and we have nested use of *let rec*, and we also have two thread subexpressions which duplicate the outer loop code). Individual instructions, which execute in a

single step in the abstract model, can take many steps to execute in the concrete model, especially as many features are defined on top of the relatively small core. For example, the Caml code corresponding to the *Test* instructions can take more than 40 steps to execute. This in turn influences the interleavings that can occur. These factors combine to cause a blowup in the complexity of the state space.

Although intermediate states are extremely large, the state space is still finite. This means it is at least theoretically possible to enumerate all the reachable states and prove mutual exclusion this way. This would not tell us anything about proving general programs correct with an operational semantics. We therefore resolve to develop general techniques that do not take any advantage of the finiteness of the state space.

Let the two Caml thread transition systems be T_i, T_j, where $\{i, j\} = \{true, false\}$. By definition of *reachable*, we must prove $\forall p\ \forall n\ (\textit{is-path}\ (\textit{par}\ T_i\ T_j)\ p \rightarrow I\ (p\ n))$ where I states that if T_i takes a step, then so too can the abstract model of T_i, all respecting the abstraction relation (we also prove the equivalent for T_j). We attempt to prove this by cases on n. If $n = 0$, then we prove $I(p\ 0)$, else if $n = 1$ we prove $I(p\ 1)\ \ldots$, at each stage using the knowledge of the previous state to determine the next state (using the definition of Caml reduction). Where reduction may result in several alternatives, for example when reading the value of an undetermined variable (we symbolically execute parametric expressions), we have a branch in the proof. In this way, the machine keeps track of the possible Caml expression at each stage. This allows us to avoid spelling out which states are reachable.

Eventually, the path will loop back to a state already seen. Rather than paths, we consider sequences p which may start in one of these recurrent states. Instead of proving $\forall p\ \forall n\ (\ldots \rightarrow I(p\ n))$ we prove the equivalent $\forall n\ \forall p\ (\ldots \rightarrow I(p\ n))$, by complete induction on n (then cases on n, as above). The induction statement contains quantification over a higher-order object, the function p, which makes it a relatively strong use of induction. Suppose $n > 30$, and $(p\ 30)$ is a recurrent state (not necessarily occurring elsewhere on p). Then consider $n' = n - 30$, and $p' = \textit{drop}\ 30\ p$, i.e. p' is the subsequence of p starting at position 30. We can invoke the induction hypothesis on n' (since $n' < n$) with p' as the path (which starts in a recurrent state) to show $I(p'\ n')$, i.e. $I(p\ n)$ and we have finished the proof. This technique does not require the state space to be finite, or for computations to terminate, indeed our threads never terminate. To preempt possible mis-readings, we emphasize that the recurrent states are not necessarily states that have been seen previously on the path.

We wish to avoid considering the interleavings of each thread, so we use a form of rely-guarantee style reasoning [Jon81]. Our system is a parallel composition of two threads T_i, T_j. Write this as $T_i|T_j$. The lemma we want to prove is that if T_i takes a step, so too can the abstract model. Then we replace the state of T_j with some arbitrary (but reachable for $T_i|T_j$) state of T_j, and replace the transitions of T_j with I_j where I_j represents the invariant guaranteed by (an arbitrary number of steps of) T_j. In our case, I_j simply states that the *trys j*

and *turn* variables in the Caml state are either *Const TTrue* or *Const FFalse*, and the *trys i* variable is preserved. A key point is that the reachability of $T_i|I_j$ is at least that of $T_i|T_j$. This technique extends smoothly to cope with more than two threads.

At every step we have to consider that I_j may make a transition, in which case we invoke the induction hypothesis, since the state of T_j is unchanged. This leads to lengthy proofs. We can reduce the proof size by half in the following way. Instead of considering transitions of $T_i|I_j$, we consider transitions of T_i', where $(T_i'\ x\ x') \leftrightarrow \exists y\ (T_i\ x\ y \wedge I_j\ y\ x')$. In effect, our transitions include steps of T_i together with interference from I_j. The key point, again, is that the reachability of T_i' is at least that of $T_i|I_j$, and therefore at least that of $T_i|T_j$. Note that we have now modularised the proof, since we are considering an sts T_i' which is very close to the sts for our original single thread T_i. Again, this technique extends smoothly to cope with more than two threads.

We have reduced the problem to examining paths of a single "thread" sts T_i' that looks similar to T_i. The proofs consist of little more than simplification to execute the reduction steps, and to check abstract transitions match concrete transitions, and some manual instantiation of the induction hypothesis when reaching a recurrent state. So the parts of the proof which are specific to Peterson's algorithm are reduced to simplification, whilst the techniques described above can be captured in general lemmas, and reused in other settings. This concludes our discussion of the Caml verification.

5 Background on weak memory models

Programmers usually understand concurrent programs based on a notion of "sequential consistency" [Kaw00], i.e. there is some sequential execution of the program which validates the observed behaviour. For example, suppose two concurrent threads $T_{\mathrm{r}}, T_{\mathrm{w}}$ execute as follows. All variables are initialized to 0. T_{w} writes 1 to location x, then 2 to location y. T_{r} reads 2 from location y, then 0 from location x.

$$\begin{aligned} T_{\mathrm{w}}:&\ x \leftarrow 1;\ y \leftarrow 2;\\ T_{\mathrm{r}}:&\ y \rightarrow 2;\ x \rightarrow 0; \end{aligned}$$

With a sequentially consistent memory, such an execution would be impossible. $y \rightarrow 2$ must occur after $y \leftarrow 2$, and so after $x \leftarrow 1$, in which case $x \rightarrow 0$ should be $x \rightarrow 1$. Although this type of reasoning is usually valid on single processor machines, it is costly to provide such behaviour in multiprocessor machines, so most multicore systems provide weaker guarantees about the order in which reads and writes are seen. In a weak memory model, where no guarantee is provided about the order of reads and writes to *different* memory locations, the execution above might be perfectly feasible. However, most algorithms are verified with a sequentially consistent model of memory. In the presence of weak memory models, existing algorithms must be modified (or new algorithms invented), along with new proofs of correctness. Practical experience bears out

these claims. For example, Holzmann [HB06] describes how a standard implementation of Peterson's algorithm was observed to fail when executing on the Intel Pentium D processor.

Weak memory models are difficult to program against. Indeed, it is known [Kaw00] that problems such as mutual exclusion are impossible to solve, even with relatively strong memory models, without explicit synchronization primitives, such as a "memory barrier" instruction. A memory barrier ensures that all writes on all processors are flushed to main memory. Although these operations can provide the guarantees programmers need, they are typically very expensive to execute, so that there is a desire to reduce their use as much as possible.

In general there is a three-way trade off, between the strength of the synchronization primitive, the extent to which the primitive is used in an algorithm, and the strength of the memory model. In this work, we fix our synchronization primitive as a per location memory barrier. We must now choose between a strong algorithm (many memory barriers) coupled to a weak memory model, or a weak algorithm with a strong memory model. We choose a strong algorithm, i.e. we emphasize portability between memory models over efficiency of the algorithm. We modify Peterson's algorithm slightly to work with a weak memory. With reference to the original pseudocode, Fig. 2, we include memory barriers after the first write of each thread to their `try` variables, and after writes to the `turn` variable.

6 Weak memory model formalization

Compared to the previous abstract formalization, the model is more complex. Reads and writes no longer refer directly to memory. Memory locations must be reified in the model, so that we can talk about a write to a particular memory location. We model the history of writes by maintaining a list, *pending-writes*, which is updated every time a thread writes to memory.

types *thread-id* = *bool* **datatype** *loc* = *Turn* | *Try thread-id*

datatype *pc* = *NonCrit* | *SetTry* | *Barrier loc* | *SetTurn* | *Test* | *Crit* | *Exit*

types *val* = *bool* **types** *memory* = *loc* ⇒ *val*
types *pending-writes* = (*thread-id* ∗ *loc* ∗ *val*) *list*
types *pcs* = *thread-id* ⇒ *pc* **types** *peterson-state* = *memory* ∗ *pending-writes* ∗ *pcs*

pending-writes is a purely logical construct that reflects the history of the trace into the current state, allowing easier invariant proofs. It is *not* intended that a real implementation maintain a cache of pending writes. The *memory*, a map from locations to values, is also a logical construct. The interpretation of the *memory* at location *loc* will turn out to be the following: *if there have been no writes by any thread to loc since the last memory barrier, then two threads reading from loc agree on the value, which is (memory loc).* Our proofs are valid for any memory model where the memory barrier satisfies this property.

$\vee$ $((ipc,ipc') = (Barrier\ Turn, Test)$
$\wedge$ *barrier Turn* $(memory,pws)$ $(memory',pws'))$

$\vee$ $((ipc,ipc') = (Test,Crit)$
$\wedge$ $(read\ memory\ pws\ (i,Try\ j,False) \vee read\ memory\ pws\ (i,Turn,i))$
$\wedge$ $(memory',pws') = (memory,pws))$

Fig. 4. Peterson's algorithm, modified for weak memory, excerpt

On executing a memory barrier for a location *loc*, if there is some pending write for *loc*, then the memory is updated by taking the last write that some thread made to *loc*. The thread that is chosen need not be the thread executing the memory barrier instruction. All pending writes for *loc* are then dropped from the list of pending writes. If there are no pending writes, memory remains unaltered.

constdefs *barrier* :: $loc \Rightarrow memory * pending\text{-}writes \Rightarrow memory * pending\text{-}writes \Rightarrow bool$

In accordance with our logical interpretation of pending writes, the *write* operation simply appends the write to the list of pending writes. Since we aim to keep our memory model as weak as possible, we declare but do not define the *read* operation. We will later constrain *read* with properties that are required for the correctness proof. The transitions are phrased in a similar style to before, and we give an excerpt only, Fig. 4.

constdefs *write* :: $(thread\text{-}id * loc * val) \Rightarrow pending\text{-}writes \Rightarrow pending\text{-}writes$
write tidlocv pws $\equiv$ *tidlocv*#*pws*

consts *read* :: $memory \Rightarrow pending\text{-}writes \Rightarrow (thread\text{-}id * loc * val) \Rightarrow bool$

7 Proof of correctness

The weak memory proof follows the outline of the previous abstract proof. The read operation depends on the memory and the pending writes. Our first task is to characterise the pending writes to each location, given the state of a thread. For example, in the following lemma, invariant P shows that if thread i is in the non-critical state, then the pending writes to i's try variable are either empty, or contain a single write by i, setting the try variable to false. The lemma is named because it logically precedes Lemma 10.5.1; it does not appear in [Lyn96].

lemma *lemma-10-5-0*:
defines *P-def*: $P \equiv \lambda\ i\ memory\ pws\ pcs\ ipc.$
let pws-Try-i = *filter* $(\lambda\ (tid,loc,v).\ loc = Try\ i)$ *pws in*
if ipc $\in \{NonCrit, SetTry\}$ *then pws-Try-i* $\in \{[],[(i,Try\ i,False)]\}$

else if ipc = Barrier (Try i) then pws-Try-i $\in$ {[(*i*,*Try i*,*True*)], [(*i*,*Try i*,*True*),(*i*,*Try i*,*False*)]}
else pws-Try-i = []
defines *R-def*: *R* $\equiv$ λ *i memory pws pcs ipc.*
let pws-i-Turn = filter (λ (*tid*,*loc*,*v*). *tid* = *i* $\wedge$ *loc* = *Turn*) *pws in*
if ipc = Barrier Turn then pws-i-Turn $\in$ {[(*i*,*Turn*,$\neg$*i*)],[]}
else pws-i-Turn = []
shows $\forall$ *p. is-path peterson-sts p*
$\longrightarrow$ ($\forall$ *n i.*
let (*memory*,*pws*,*pcs*) = *p n in*
let ipc = pcs i in
P i memory pws pcs ipc
$\wedge$ *R i memory pws pcs ipc*)

One might imagine that placing a memory barrier after every write to a location is sufficient to ensure correctness. This is not correct for our model of memory, because it is still possible to read from a location with a pending write, in which case the read is unconstrained. Another subtlety is the interaction of the memory barriers. For example, thread i may have written to variable *turn*, and be on the point of doing a memory barrier to force the write to memory. In the meantime, the other thread j may also write to *turn*, and complete the subsequent memory barrier. In this case, i will execute the memory barrier but i's pending write will have been discarded by j's memory barrier. This case can be seen in the characterisation of the pending writes, since if i's *pc* is *Barrier Turn*, the pending writes to the turn variable may be the empty list. In this scenario, i's write may never reach memory at all.

With *lemma-10-5-0*, we can prove the equivalent of *lemma-10-5-1*, i.e. that if thread i's program counter lies in the range *SetTurn* to *Exit*, and thread j reads i's try variable to be v, then v must be true. The proof requires a restriction on the behaviour of the *read* operation: if there have been no writes to a location since the last memory barrier for that location, and two threads read that location, then they read the same value. It is easiest to express this as the existence of a function *memory* from locations to values, which gives the common value that threads must read in case there are no pending writes. We use a *locale* to capture this assumption, and the proof is conducted within this locale, in the scope of the assumption.

locale *l* = **assumes** *a*:
$\forall$ *memory pws tid loc v. filter* (λ (*tid*,*loc*$'$,*v*). *loc*$'$ = *loc*) *pws* = []
$\longrightarrow$ (*read memory pws* (*tid*,*loc*,*v*) = (*memory loc* = *v*))

We can now prove the equivalent of *lemma-10-5-2*, property Q in the following lemma statement. We require an auxiliary fact to handle the case that a thread's write to the *turn* variable has been discarded by the memory barrier of the other thread. For example, it is conceivable that j writes to *turn*, i writes to *turn*, i performs a memory barrier, and subsequently enters the critical section. j then performs a memory barrier (but there are no pending writes), reads i's write

to *turn* which gives priority to j, and so enters the critical section. However, if i enters the critical section, then i's memory barrier must have committed j's pending write to the *turn* variable. In which case, j is prevented from entering the critical section. Some work is required to phrase this reasoning as the invariant P. As before, our final correctness theorem follows directly from Q, since *memory Turn* cannot be both i and j simultaneously.

lemma (**in** *l*) *lemma-10-5-2*:
defines *P-def*:
$P \equiv \lambda\ i\ ipc\ jpc\ memory\ pws.$
$ipc = Crit$
$\wedge\ jpc = Barrier\ Turn$
$\wedge\ filter\ (\lambda(tid,loc,c).\ loc = Turn)\ pws = []$
$\longrightarrow memory\ Turn = i$
defines *Q-def*:
$Q \equiv \lambda\ i\ ipc\ jpc\ memory\ pws.$
$ipc = Crit$
$\wedge\ jpc \in \{Test,Crit\}$
$\longrightarrow memory\ Turn = i$
shows
$\forall\ p.\ is\text{-}path\ peterson\text{-}sts\ p$
$\longrightarrow (\forall\ n\ i.$
$let\ (memory,pws,pcs) = p\ n\ in$
$let\ ipc = pcs\ i\ in$
$let\ j = \neg\ i\ in$
$let\ jpc = pcs\ j\ in$
$P\ i\ ipc\ jpc\ memory\ pws$
$\wedge\ Q\ i\ ipc\ jpc\ memory\ pws)$

8 Related work

A substantial amount of program verification has used Hoare logics [Hoa69]. A state of the art example of mechanized Hoare reasoning for while-programs with state is [MN05]. The language treated is significantly simpler than that here. The work of Homeier [HM95] also treats a very simple language. This work includes a verification condition generator, essentially a tool that constructs a syntax-directed proof for the program (augmented with user hints and invariants), and returns the steps that could not be proven automatically as "verification conditions".

Our work uses operational reasoning directly above an operational semantics. Operational reasoning is much more flexible then Hoare logic, since any technique allowed by the meta-logic (in our case higher-order logic) is directly usable. On the other hand, we know of no essential advantages of Hoare logic over operational reasoning. Strother Moore has also followed this course. The detail of the operational models mandate mechanical support, but current theorem provers are more than up to the task. Indeed, Strother Moore states[2] "had there been decent theorem provers in the 1960s, Floyd and Hoare would never have had to invent Floyd-Hoare semantics!" His work [LM04] focuses on operational Java program verification. However, he first compiles the Java to bytecode, and then verifies this. Correctness properties are phrased as properties of the bytecode, and reasoning occurs above the bytecode, not above the original Java program. The examples treated, such as an "add one" program, and a Java function that implements factorial, are significantly simpler than the work presented here.

[2] http://www.cs.utexas.edu/users/moore/best-ideas/vcg/index.html.

Concurrency, and the problem of the state space explosion in interleaving between threads, is also not tackled. A very rare example of operational reasoning applied to a high-level language is the work of Compton [Com05] who verifies a version of Stenning's protocol for a restricted model of Caml and UDP. This work is similar, but again much simpler, than that presented here.

Of course, where concurrency is involved, all these examples assume a sequentially consistent memory model. There has been much work on weak memory models. The PhD of Kawash [Kaw00] provides good references into the literature. Mechanization usually takes the form of model checking. Examples include [MLP04], where the authors tackle the problem by analysing dependencies between reads and writes; and [BAM06], where the authors use a constraint based approach based on the SAT encoding method in [GYS04]. As far as interactive mechanization goes, Mike Gordon has some early unpublished work[3] on memory models of the Alpha processor.

For weak memory models, model checking offers a tractable approach to many of the main problems. However, model checking is not always an option. The thrust of our work is to see if traditional reasoning techniques, employed when *designing* algorithms such as Peterson's, can scale to the weak memory case. Our experience leads us to believe that they can, but that the effort is great, and mechanical assistance seemingly mandatory. Put simply, it is very hard for humans to design concurrent algorithms if they are restricted to reasoning about the interleaving of thread transitions. Weak memory models add even more complexity. However, new ways of thinking (and proving facts) about weak memory may alleviate the problem.

9 Conclusion

We presented the verification of a Caml implementation of Peterson's algorithm, down to the operational semantics. The proofs were non-trivial, and we spent considerable effort phrasing the lemmas and shaping the induction statements, as described in Sect. 4. However, once the overall structure of the proof was in place, the actual body of the proof was mostly symbolic execution. The lemmas, induction schemes, and proof techniques are general, and have the potential to scale to real programs in real languages, so that future verifications will benefit from the experience described here. We also addressed the problem of weak memory models by verifying a version of Peterson's algorithm running on a weak memory model.

Our model of Caml included all the features necessary to support the example of Peterson's algorithm. However, we are interested in verifying other programs which use more Caml features. To this end, we are working with several others on a highly realistic model of Caml, using the Ott tool [SNO+] to state our definitions. We intend to develop metatheory for this model, and operational proofs of program correctness for other Caml programs. The next program we

[3] See http://www.cl.cam.ac.uk/research/hvg/FTP/FTP.html.

intend to verify is a Caml implementation of a fully automatic theorem prover for first order logic [RM05].

References

[BAM06] Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. Bounded model checking of concurrent data types on relaxed memory models: A case study. In Thomas Ball and Robert B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 489–502. Springer, 2006.

[Com05] Michael Compton. Stenning's protocol implemented in UDP and verified in Isabelle. In Mike D. Atkinson and Frank K. H. A. Dehne, editors, *CATS*, volume 41 of *CRPIT*, pages 21–30. Australian Computer Society, 2005.

[Gon05] Georges Gonthier. A computer-checked proof of the Four Colour Theorem, 2005. http://research.microsoft.com/~gonthier/4colproof.pdf.

[GYS04] Ganesh Gopalakrishnan, Yue Yang, and Hemanthkumar Sivaraj. QB or not QB: An efficient execution verification tool for memory orderings. In Rajeev Alur and Doron Peled, editors, *CAV*, volume 3114 of *Lecture Notes in Computer Science*, pages 401–413. Springer, 2004.

[HB06] G.J. Holzmann and Dragan Bosnacki. The design of a multi-core extension of the Spin Model Checker. In *Formal Methods in Computer Aided Design (FMCAD)*, November 2006.

[HM95] Peter V. Homeier and David F. Martin. A mechanically verified verification condition generator. *Comput. J*, 38(2):131–141, 1995.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12, 1969.

[Jon81] C. B. Jones. Development methods for computer programs including a notion of interference. Technical Report PRG-25, Programming Research Group, Oxford University Computing Laboratory, 1981.

[Kaw00] Jalal Y. Kawash. *Limitations and capabilities of weak memory consistency systems*. PhD thesis, Computer Science, University of Calgary, 2000.

[LM04] H. Liu and J. S. Moore. Java program verification via a JVM deep embedding in ACL2. *Lecture Notes in Computer Science*, 3223:184–200, 2004.

[Lyn96] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[MJ84] F. L. Morris and C. B. Jones. An early program proof by Alan Turing. *Annals of the History of Computing*, 6(2):193–143, 1984.

[MLP04] Samuel P. Midkiff, Jaejin Lee, and David A. Padua. A compiler for multiple memory models. *Concurrency and Computation: Practice and Experience*, 16(2-3):197–220, 2004.

[MN05] Farhad Mehta and Tobias Nipkow. Proving pointer programs in higher-order logic. *Information and Computation*, 199:200–227, 2005.

[Pet81] Gary L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters, 12(3):115-116*, June 1981.

[Rid] Tom Ridge. Simpson's four slot algorithm in Isabelle/HOL. Available online at http://www.cl.cam.ac.uk/~tjr22.

[RM05] Tom Ridge and James Margetson. A mechanically verified, sound and complete theorem prover for FOL. In *Proceedings of TPHOLs 2005*, 2005.

[SNO+] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective tool support for the working semanticist. Accepted to ICFP 2007: The 12th ACM SIGPLAN International Conference on Functional Programming.