A memory model for multicore OCaml

Stephen Dolan KC Sivaramakrishnan
University of Cambridge

Introduction

When multiple threads are executed in parallel, the results of reading and writing to shared references can be surprising. For instance, suppose that the following two snippets of code run in parallel, where x and y are integer references initially 0:

```
thread 1:
1. x := 1;
2. let a = !y in
   ...

thread 2:
3. y := 1
4. let b = !x in
   ...
```

This could result in $a=0$, $b=1$, or in $b=0$, $a=1$ if one of the threads runs to completion before the other begins. Surprisingly, on most machines, it could also result in $a=0$, $b=0$, due to a hardware optimisation called *store buffering*.

This and other even more surprising behaviours are possible due to optimisations performed by hardware, optimisations performed by compilers, and complex interactions between them. Specifying the precise behaviour of shared references sufficiently clearly that programmers can reason about them is the job of the memory model.

Generally, programs should use high-level synchronisation constructs like fork/join or Reagents rather than dealing directly with shared mutable state, making the details of memory models irrelevant. However, certain low-level, high-performance concurrency libraries need efficient memory primitives, and a detailed memory model is necessary to know what they do.

We propose a memory model for OCaml, broadly following the design of axiomatic memory models for languages such as C++ and Java, but with a number of differences to provide stronger guarantees and easier reasoning to the programmer, at the expense of not admitting every possible optimisation.

Here, we concentrate mostly on the design goals and general properties of the memory model. An introduction to the details for non-experts is available at:

`https://github.com/ocamllabs/ocaml-multicore/wiki/Memory-model`

For memory-model experts, a formalisation of the memory model using the `herd` [1] tool is available at:

`https://github.com/ocamllabs/ocaml-memory-model`

The compilation scheme for the memory model to weakly-ordered PowerPC machines has been proven (on paper), and mechanically verified for small litmus tests (using the `memalloy` tool [2]).

Atomics and data races

Like many languages, we distinguish *atomic* and *non-atomic* locations. The mutable locations available in current OCaml (`ref` cells, mutable fields, mutable object instance variables) are classed as non-atomic, and atomic references are available through the new `Atomic` module. Atomic locations have sequentially consistent behaviour, while non-atomic locations admit some relaxed behaviours (but are more efficient).

Atomic locations should be used whenever a variable is used for synchronisation between multiple threads. For instance, the following program uses a flag of type `bool Atomic.t` (initially `false`) and a variable `message` of type `int ref` (initially `0`) to send a message between threads:

```
thread 1:
1. message := 42
2. Atomic.set flag true

thread 2:
3. let seen = Atomic.get flag in
4. let value = !message in
5. if seen then print_int value;
```

If the flag were an ordinary reference instead of an `Atomic.t`, then it would be possible for this program to print `0`, since lines 1 and 2 would not be executed in any particular order (nor lines 3 and 4).

A program with no concurrent accesses (except concurrent reads) to the same non-atomic location is said to be *data-race free*. Data-race freedom is an property that programs should strive for: data-race-free programs have sequentially consistent behaviour, and need not worry about strange relaxed memory effects.

The C++ memory model imposes data-race freedom as a burden on the programmer: the compiler is allowed to assume that the program has no data races, and programs with data races have undefined behaviour. That is, if two threads race on a non-atomic boolean flag, the result is not that the flag's value is unpredictable, but that the entire program may do anything at all. This is not compositional: a race in one part of the program can in principle cause an unrelated part to give the wrong answer. In practice, inlining followed by aggressive optimisations

can make "unrelated" parts of the program to have more subtle interactions than one might imagine.

Our proposed memory model does not make an assumption of data race freedom. It is still advisable to avoid data races: reasoning about behaviour in the presence of data races is difficult. However, a data race on a particular non-atomic location will make only accesses to that location difficult to reason about, rather than infecting the entire program.

Compilation scheme

Compilation to machines with relatively strong memory models such as `x86` is straightforward: non-atomic accesses map to ordinary load and store instructions, and atomic accesses map to atomic instructions.

The challenging case is compilation to weakly-ordered machines such as ARM and PowerPC. Naively compiling non-atomic accesses to ordinary loads and stores will expose the programmer to some truly odd behaviour, and when compiling atomics a careful choice must be made between the various levels of atomicity and ordering the instruction set provides.

The most important tricky case in our compilation scheme concerns initialisation of newly-allocated objects. If one thread performs `x := (1, 2)`, then if another thread running in parallel reads `x` and sees the new tuple, it must also see the correct contents rather than uninitialised memory. On ARM and PowerPC machines, this will not happen by default as those machines maintain the ordering of neither loads nor stores.

A similar issue arises in the Java memory model. The general solution is that a fence instruction must be inserted somewhere between the initialisation and the publication of the object (that is, the mutation that makes it accessible from other threads). Java chooses to place this fence just after initialisation, to preserve the performance of mutation. We choose to place this fence on those mutations which may publish new objects, to preserve the performance of functional code.

Compiler optimisations

We aim not to inhibit the sort of compiler optimisations that OCaml currently does, although preserving maximum freedom for the compiler to optimise is a lesser goal with our proposal than ensuring it is possible to reason about even racy programs.

To our knowledge, the OCaml compiler currently does no optimisations incompatible with our proposal, although the proposal does somewhat constrain which optimisations may be applied in future. An example of such an optimisation, which would be incompatible with our proposed memory model, is *rematerialisation* of loads to mutable locations. That is, a compiler for a single-threaded language may validly transform the following code:

```
let a = !global in
let b = x + y in
let c = a + 1 in
...
```

into the following:

```
let a = !global in
let b = x + y in
let c = !global + 1 in
...
```

on the basis that `!global` cannot change during the computation of `x+y`. (It is not usually profitable to introduce more memory accesses, but it can be under high register pressure).

This optimisation is incompatible with our proposal, as are others which introduce new memory accesses to shared mutable state. However, we believe that this will not be unduly constraining for OCaml, and the benefits of being able to reason about all programs outweigh the cost of missed optimisation opportunities. The converse optimisation, eliminating redundant memory accesses, is much more important and is compatible with the proposal.

Discussion and future work

Memory models are subtle beasts, and it is difficult to do justice to one in two pages. Nonetheless, we believe that the model proposed here is reasonably efficient on current hardware, while providing guarantees to the programmer strong enough to reason about.

With the in-development `arm64` backend for multicore OCaml, we have begun experimenting with the memory model's implementation on weakly-ordered processors, and our next task is detailed performance testing of programs using atomic variables for synchronisation. On the theory side, future work involves mechanised proofs of correctness for the compilation schemes and validity of optimisations (currently, we have only informal paper proofs, supplemented with exhaustive testing for small programs).

Acknowledgements We thank Mark Batty, Peter Sewell, Susmit Sarkar and Shaked Flur for memory-model discussions, and John Wickerson for help testing the compilation schemes to PowerPC.

References

[1] J. Alglave, L. Maranget, and M. Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *TOPLAS*, 36(2):7:1–7:74, July 2014.

[2] J. Wickerson, M. Batty, T. Sorensen, and G. A. Constantinides. Automatically comparing memory consistency models. In *POPL '17*, pages 190–204. ACM, 2017.