Advanced Functional Programming

Software Engineering Chair and Programming Systems Lab

Small-group work

Questions for *Composable Memory Transactions* by Tim Harris and others. It appeared in the Proceedings of the ACM SIGPLAN 2005 Symposium on Principles and Practice of Parallel Programming, pages 48–60.

1. What destroys composability of lock-based synchronization? On the other hand, what makes the approach proposed by Harris et al. composable?
2. How does Haskell's type system help to implement composable memory transaction? How would a design for ML look like?
3. Before `atomic` commits a transaction, it *validates* it. What for, and how does this work?

Homework

1. Read *Fun with Phantom Types* by Ralf Hinze.
2. Summarize the paper *in your own words* on one page. Put your name and student ID on your summary and drop off a printout at office **525/45** until Monday, January 16th at noon (12 am). If the door is closed, slide your printout under the door. No Emails.

Email conversation with Tim Harris

Tim,

we are reading your paper in my seminar course on Advanced Functional Programming. I believe that this paper makes an important contribution to the discussion how to deal with concurrency. I'd like to ask a question that is triggered by the comparison with select(2) that you provide in your paper: while select(2) is not composable, it is fair. It waits for file descriptors from a set to become ready, not preferring any of the these. The orElse opererator is composable but but it also forces me to provide an order: I can try to execute one transaction and, if this fails, another. But I cannot try to execute both. Maybe I am confusing things here, but I was missing the inherent parallelism of select. Maybe there is even a tradeoff between waiting in parallel and composability, I wonder. I would appreciate your thoughts.

-- Christian

Hi,

That's an interesting observation -- you're right in that orElse doesn't meet the usual definition of fairness in that an alternative that is always ready to run might never be selected.

However, some of the examples that motivate orElse rely on this deliberate bias -- for instance, suppose that "WaitForA" is a blocking operation and we want to turn it into one that returns a true/false value according to whether it would block (in pseudo-code):

```
    { WaitForA(); return false; } 'orElse' { return true; }
```

In that case it's crucial that orElse is not "fair".

Another motivation for a biased orElse is that a program can always randomise the order in which alternatives are composed. That might deal with the kind of starvation problem that you describe. Is it possible to build a biased one from a non-deterministic one? Probably not without leaving the STM monad (or adding support for parallelism within it).

Of course, an implementation of orElse could actually run both alternatives in parallel, tentatively performing the second alternative in a separate transaction in the hope that the effects are already prepared if the first alternative blocks. A related idea we've talked about would be something like "A 'andThen' B" where both alternatives run in the hope that "B" does not depend on "A" (this being checked dynamically by the STM, re-running B after A if there is a conflict).

Thanks,

Tim