Verifying Concurrent ML programs

a research proposal

Gergely Buday
Eszterházy Károly University
Gyöngyös, Hungary

Synchron 2016 Bamberg
December 2016

Concurrent ML

- is a synchronous language
- a CML program consists of threads
- communicating via message passing
- (instead of shared memory)
- with rendezvous communication
- either the sender or the receiver blocks

Reppy on concurrent programming

Writing correct concurrent programs is a difficult task. In addition to the bugs that may arise in sequential programming, concurrent programs suffer from their own particular kinds of problems, such as races, deadlock, and starvation.

What makes this even more difficult is that concurrent programs execute nondeterministically, which means that a program may work one time and fail the next.

Reppy: Concurrent Programming in ML

Reppy on concurrent programming cont'd

A common suggestion to address this problem is to use formal methods and logical reasoning to verify that one's program satisfies various properties. While this is a useful pedagogical tool, it does not scale well to large programs.

Reppy: Concurrent Programming in ML

Application of concurrency

- operating systems
- telecommunication
- startups: Whatsapp (Erlang), Bleacher Report (Elixir)
- multicore systems
- user interfaces: avoiding the event loop, *poor man's concurrency*
- reactive systems

Asynchronous vs synchronous message passing

in asynchronous message passing, the message queue of a channel might grow without limit, causing a memory overflow

there is another problem with memory overflow: it is likely to be far removed in time and in place from the source of the problem.

while in the synchronous case deadlock is the most frequent bug, which is easily detected

thus, detecting and fixing bugs in a synchronous message-passing program is easier

Reppy: Concurrent Programming in ML

Asynchronous vs synchronous message passing cont'd

In theory, an asynchronous system is faster than a synchronous, but in practice those systems need synchronisation as well, possibly in the form of acknowledgement messages, leading to a re-implementation of synchronous communication, which might be slower done by hand.

In rendezvous communication, the sender and the receiver has ***common knowledge****. "This property makes synchronous message passing easier to reason about, since it avoids a kind of interference where the state of the sender when the message is sent is different than when the message is received"*

Reppy: Concurrent Programming in ML

CML: spawning a thread and communication on channels

```
val spawn : (unit -> unit) -> thread_id
```

contrast to Unix: the child can run when the parent terminates

```
val channel : unit -> 'a chan
val send : 'a chan * 'a -> unit
val recv : 'a chan -> 'a
```

CML: events

events are possible communications, much like a function with a unit argument is a possible computation in basic functional programming:

```
fun greet () = print "Hello World!"
```

one can create an event from a channel with *recvEvent* and then synchronise on it and get the corresponding value with *sync*:

```
val recvEvt : 'a chan -> 'a event
val sync    : 'a event -> 'a
```

so *recv* is not a primitive, but a composition of event creation and synchronisation:

```
val recv = sync o recvEvt
```

CML: selective communication

Or, nondeterministic choice

the run-time system chooses one from the enabled events from a list

and the communication takes place

```
val select : 'a event list -> 'a
```

Its event-based counterpart is *choose*:

```
val choose : 'a event list -> 'a event
```

where the obvious

```
val select = sync o choose
```

equation holds

CML: wrap: post-synchronisation

```
val wrap : ('a event * ('a -> 'b)) -> 'b event

fun add (inCh1, inCh2, outCh) =
forever
()
(fn () =>
 let val (a, b) = select [
            wrap (recvEvt inCh1,
                    fn a => (a, recv inCh2)),
            wrap (recvEvt inCh2,
                    fn b => (recv inCh1, b)) ]
  in
      send (outCh, a + b)
 end)
```

Charguéraud on program verification

- through Hoare triples
- defining programs directly in a theorem prover

  shallow embedding

- defining the semantics of the language in the logic of the theorem prover, and in turn use these definitions to write programs, and then prove correctness of these

  deep embedding

Deep embedding: SECD versus CEK

The semantics of a functional language can be defined through an abstract machine

> *The SECD and CEK machines produce the same result for any expression that has a result, but they compute the result in a different way.*

Felleisen: SECD, Tail Recursion and Continuations

SECD vs CEK

Landin's SECD: Stack, Environment, Control and Dump

> *In the SECD machine, context is created by function calls, where the current stack, environment, and control are packaged into a dump. The stack provides a working area for assembling application arguments.*
>
> *This view of context accumulation is natural when approaching computation from the Pascal or C model.*

CEK: Control, Environment and Continuation

> *In the CEK machine, context is created when evaluating an application function or argument, regardless of whether the function or argument is a complex expression.*
>
> *This view of context accumulation is natural when approaching computation from the lambda calculus model.*

Felleisen: SECD, Tail Recursion and Continuations

Concurrency semantics: Berry, Turner, Milner

Berry, Turner, Milner: A semantics for ML concurrency primitives

The authors give a transitional semantics for a language very similar to CML

While Reppy arrived to the *event* type constructor via pragmatic reasons, the authors reached to the same conclusion via semantic reasoning

they claim that it is not possible to use the relational operational semantics of the Definition of Standard ML

> *because there is no way to specify*
> *potentially infinite interleaved evaluations*

Concurrency semantics: Havelund

Havelund: The Fork Calculus, Phd Thesis

Havelund develops various formalisms for concurrent languages

He finds the *fork* operator especially important to handle

Concentrates on process equivalence, which is usually a bisimulation. This needs to be a congruence as well.

Besides equational reasoning, develops modal logic for specification, based on Hennessy-Milner logic

Building on Extended ML, he creates Extended Concurrent ML

Concurrency semantics: Extended Concurrent ML

The third branch includes further elaboration of the specification language for CML. We consider the work presented in this thesis of major importance for such future work.

In order to provide a useful specification logic, many examples have to be written, possibly suggesting modifications of the logic proposed here. Also, a proof theory needs to be defined including proof rules.

Finally, proofs cannot be done solely by hand, so a proof theory must be supported by software tools. One cannot hope for a fully automatic theorem prover for a language like ECML, but one can hope for a user-driven proof-support involving editing, proof checking, and automated theorem proving in particular simple cases.

Havelund: The Fork Calculus, PhD Thesis

Concurrency semantics: other authors

Mosses and Musicante: An action semantics for ML concurrency primitives

Amadio, Leth, Thomsen: From a concurrent lambda-calculus to the pi-calculus

Debbabi and Bolignano: A semantic theory for ML higher-order concurrency primitives

Jeffrey, Ferreira, Hennessy and Rathke: a number of papers

Concurrency semantics: Priami PhD Thesis

Priami: Enhanced Operational Semantics for Concurrency, PhD Thesis

> *From an algebraic point of view the main difference between truly concurrent and interleaving semantics concerns the constructors which model the parallel composition of processes.*
>
> *Within interleaving theories these constructors can be always expressed as a combination of other operators of the language not modelling parallel composition. The combination yields the well known expansion law. In other word, parallel composition is a derived operator.*
>
> *On the other hand, non interleaving theories assume parallel composition as a primitive operator, i.e. they cannot be expressed as combinations of other constructors of the language.*

Concurrency semantics: small or big-step

Traditionally, concurrency is modelled using small-step operational semantics as a concurrent computation does not necessarily lead to a final value, what is the thought behind big-step semantics. Uustalu debates this stance:

> *First, big-step operational semantics can handle divergence as well as small-step semantics, so that both terminating and diverging behaviors can be reasoned about uniformly.*
>
> *Big-step semantics that account for divergence properly are achieved by working with coinductive semantic entities (transcripts of possible infinite computation paths or nonwellfounded computation trees) and coinductive evaluation.*

Uustalu: Coinductive big-step semantics for concurrency

Concurrency semantics: small or big-step cont'd

Second, contrary to what is so often stated, concurrency is not inherently small-step, or at least not more inherently than any kind of effect produced incrementally during a program's run (e.g., interactive output).

Big-step semantics for concurrency can be built by borrowing the suitable denotational machinery, except that we do not want to use domains and fixpoints to deal with partiality, but coinductively defined sets and corecursion.

Uustalu: Coinductive big-step semantics for concurrency

Tool: the theorem prover Isabelle

Proven track record: Archive of Formal Proofs

The seL4 formally verified microkernel, 9000 lines of C code

Supports coinduction and corecursion

Large and helpful community

Tackling languages with binders: Nominal Isabelle

In λ-calculus, $\lambda x.x =_\alpha \lambda y.y$

an orderly exchange of bound variables: α-equivalence

a general theory of this: Pitts: Nominal Sets

an implementation of this: Nominal Isabelle by Christian Urban

relevant application for me:

Parrow, Borgström, Eriksson, Gutkovas, Weber: Modal Logics for Nominal Transition Systems

Research plan

Learning the tools: Isabelle and Nominal Isabelle

Pilot project: verifying basic abstract machines in Isabelle

Open research: evaluate various concurrency semantic methods

Create a specification language for CML programs, following temporal logic and Havelund's Extended CML

Implement a program verifier with a chosen semantic method

Develop case studies

Write up

Questions?