Erlang in ACL2: A CPS Evaluator

Kuter Bora[1] Mark Greenstreet[2]

[1]Max Planck Institute for Security and Privacy

[2]University of British Columbia

16 November 2026

Motivation

- Real world software is largely concurrent/parallel.
- Concurrency begs for formal reasoning.
  - Humans fail when the try to “think of all interleaving” of execution.
  - Most SW verification in ACL2 is for sequential code.
- Erlang was designed for implementing large-scale, reliable concurrent software.
  - Erlang is a functional language – simpler to reason.
  - Support for fault-tolerance is built into the Erlang process model.
- There is some support for formal reasoning about Erlang programs, but most has been ad hoc.
- We need a industrial-scale way to reason about concurrent programs (in Erlang)

Contributions

- We have a continuation-passing style evaluator in ACL2 for a large subset of Erlang. Continuations provide a unified approach for modeling concurrency, communication, and exceptions.
- We have proven properties of some simple Erlang programs using this evaluator.
- This is work-in-progress.
  - The proofs we have show what we need to do for the next few steps of developing this infrastructure.
  - Our goal is to demonstrate a practical way to reason about realistic concurrent programs.

What is Erlang?

- “Erlang is a concurrent language consisting of communicating components where the components are written in a functional language.” [4].
- Erlang has been widely used, especially for telecommunications applications – see https://erlang-companies.org.
- Like ACL2, Erlang is an untyped, functional language – there is a natural semantic match.

Sequential Erlang: Sum of Natural Numbers

```
sum_of_nats(0) -> 0;
sum_of_nats(N) when is_integer(N), 0 < N ->
  N + sum_of_nats(N - 1).
```

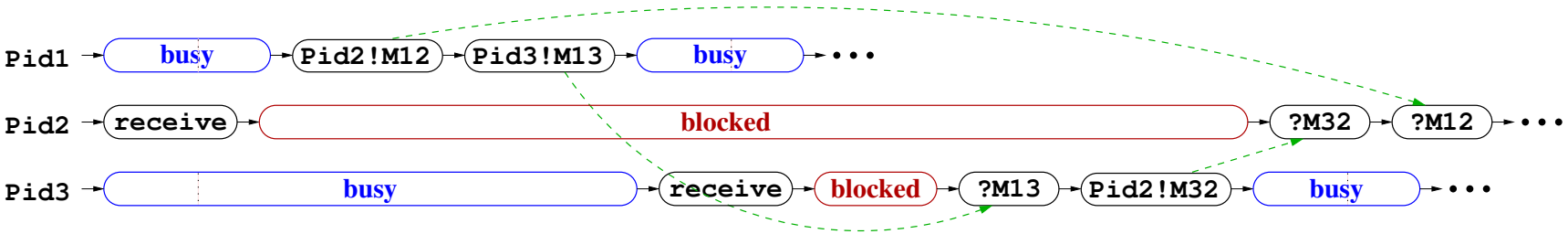
- `sum_of_nats` computes the sum of first `N` natural numbers.
- Functions are defined as lists of clauses separated by semicolons. Erlang makes extensive use of pattern matching.
  - The first clause matches when the value of `N` is 0.
  - The second clause matches when `N` is a positive integer.
  - If no clause matches, an exception will be thrown.

Concurrent Erlang: A Prime Server

```
prime_server(KnownPrimes) when is_list(KnownPrimes) ->
  receive
    exit -> ok;
    {prime_please, Pid} when is_pid(Pid) ->
      Pid ! {prime, self(), hd(KnownPrimes)},
      prime_server(next_prime(KnownPrimes)) end.
```

- The prime server receives requests for prime numbers.
- It delivers primes to the requesting processes in ascending order in which the requests were received.
- Messages are sent with a send expression: `Dst ! Message`, and received with a `receive` expression.
- Erlang is often used to implement reactive programs. Such a program may never terminate (discussed later).

Message passing in Erlang



- For any two (not necessarily distinct) pids, `pid1` and `pid2`, Erlang guarantees the messages sent from `pid1` to `pid2` are enqueued in `pid2`'s message queue in the order that they were sent by `pid1`.
- There are no other ordering guarantees.

How do we formalize Erlang?

- Using evaluators to define the semantics of a language is common practice in ACL2.
- We use an abstract scheduler, that, given a program consisting of multiple processes, chooses a process and runs it to a pre-emption point, and continues by choosing the next process to run.
- We use continuations to model processes – a continuation holds the process state when the process is preempted and another process runs.

What is CPS?

```
sum_of_nats(0, F) -> F(0);
sum_of_nats(N, F) when is_integer(N), 0 < N ->
  sum_of_nats(N - 1, fun(X) -> F(X + N) end).
```

- A continuation is a function that says “what to do next” [3].
- Prior use of continuation in ACL2 includes [13]:
  - [13] builds on [19] that provided a way to reason about partial, tail recursive functions.
  - [13] extends this to arbitrary functions because any recursive function can be made tail recursive by using continuations.
  - We are not aware of previous work in ACL2 using continuations to model concurrency.

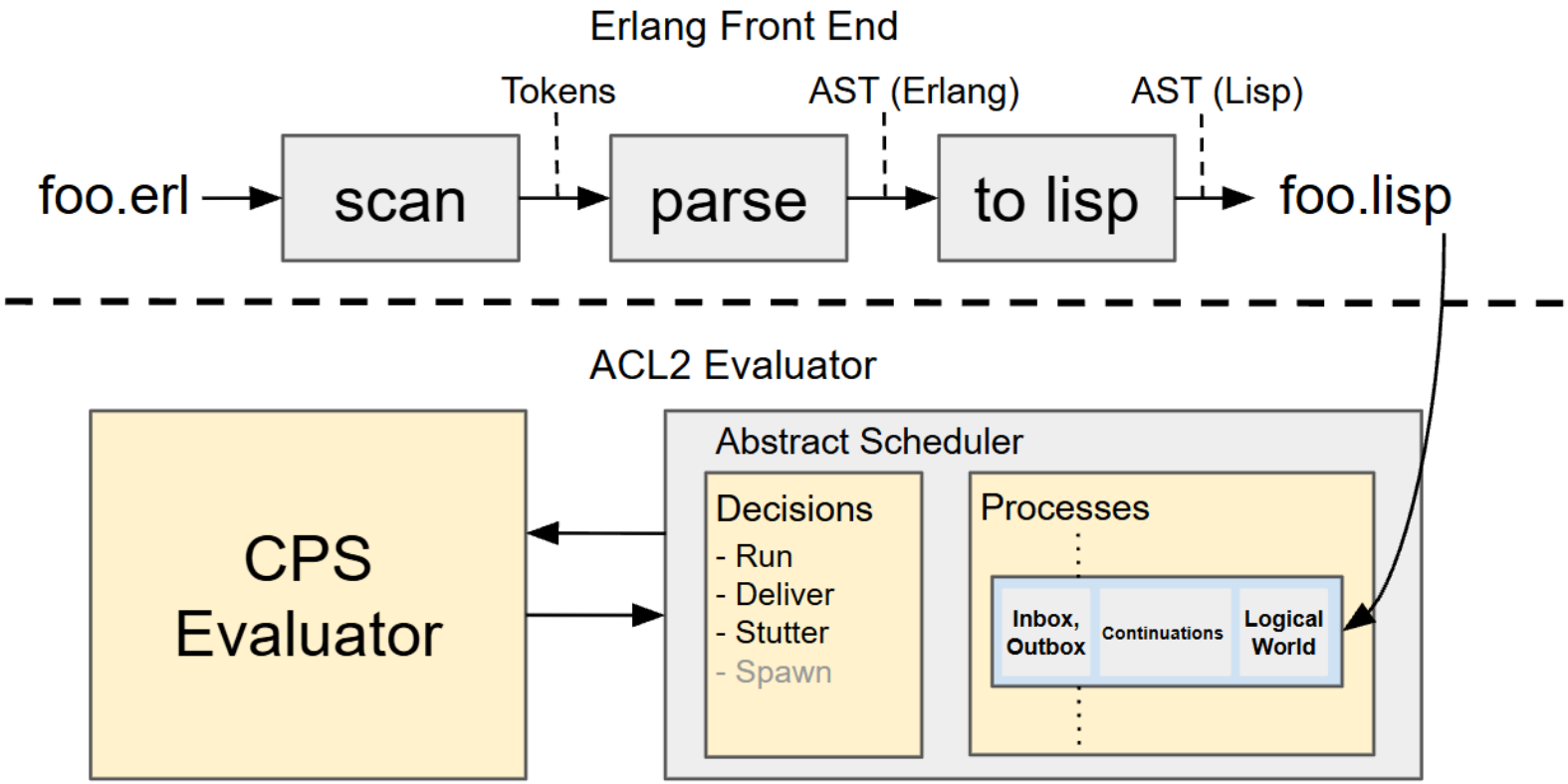
How is CPS useful?

```
eval_erlang((:receive cls), F) ->
  fun(Message) -> F(receive(Message, cls) end;

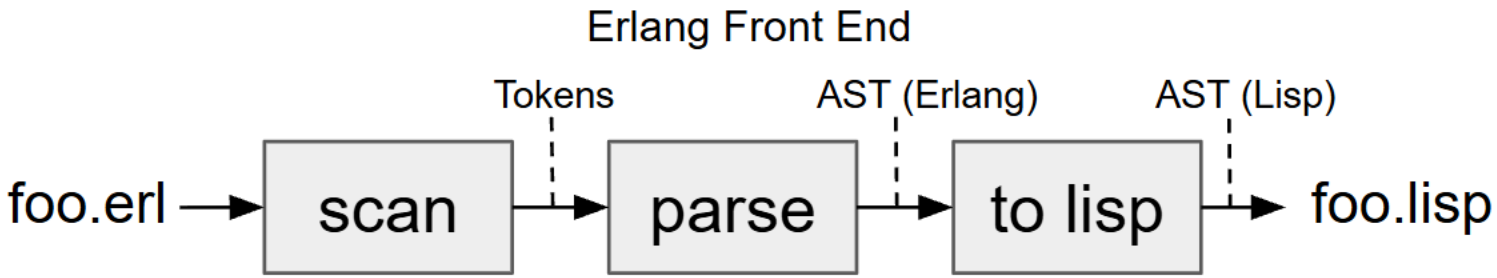
eval_erlang((:binop + A B), klst) ->
  eval_erlang(A, fun(X) ->
    eval_erlang(B, fun(Y) -> X + Y end) end.
```

- Each process can be represented as a continuation.
- The evaluator runs sequential code as usual.
- If there is a `receive`, we just return the continuation from that point.
- The scheduler can take a list of processes, each of which is a continuation, and then decide to run or deliver messages.

Implementation



Translating Erlang



- We use the existing the Erlang scanner and parser, then convert the result to Lisp.
- For example, `1 + 2.` is parsed as:

  ```
  {op + {integer, 1} {integer, 2}}
  ```

  which we translate as:

  ```
  (:binop + (:integer 1) (integer 2))
  ```

- We use `fty` to define a type for Erlang ASTs.

Continuations, Message-queues, and the Logical World

- Each process consists of a list of continuations, a logical world, and message-queues (handled by the scheduler).
- We use `fty::deftagsum` to define the possible continuations, where each continuation encodes a single Erlang operation.
- The logical world contains the function definitions.
- The message-queues (inbox and outbox) store pending messages.

Termination of the Evaluator

- We give up immediately. Erlang is often used to implement non-terminating programs.
- Instead of proving termination, we bound each continuation with a fuel.
- The evaluator decrements fuel for each new continuation, and exits with an out-of-fuel status if (zp fuel).
- TODO: use `defchoose` to find a sufficient amount of fuel, if it exists.
  - This should be able to build on [13].

Abstract Scheduler

- At each step, the scheduler decides to either;
  - run a process by invoking the CPS evaluator.
  - deliver a message from an outbox to the destination inbox.
  - or stutter.
- In the current implementation, the only fairness condition is that if there is valid non-stuttering action, then some non-stuttering action will be taken.
- Messages sent from the same source process to the same destination are delivered in order.
- If a process's next continuation is a `receive`, and it is run, it will try to receive a matching message from its inbox in order of arrival.

Examples

We show two examples of proving properties of Erlang programs using our evaluator.

- the sum of the natural numbers up to N is $\frac{N^2+N}{2}$.
- Schwartz's parallel algorithm for reduce (simplified) computes the sum of the values contributed by each worker process.

Example 1: `sum_of_nats`

```
sum_of_nats(0) -> 0;
sum_of_nats(N) when is_integer(N), 0 < N ->
  N + sum_of_nats(N - 1).
```

- We want to show that `sum_of_nats(N)` = $\frac{N^2+N}{2}$, i.e., the closed form of sum.
- We assume that:
  - `sum_of_nats` is defined in the logical world.
  - there is enough fuel for evaluation to terminate without running out.

Proof Strategy

- We found that reasoning directly about CPS evaluations is tedious and produces brittle proofs.
- It is easier to reason about Lisp functions, so we define:

```
(defun sum-of-nats (n)
  (if (= n 0)
      0
      (+ n (sum-of-nats (1- n)))))
```

- We show that evaluating `sum_of_nats(N)` is equivalent to evaluating the Lisp function with the value of `N`.
- Then, we can use the existing community books to complete the proof.

Reasoning about continuations

- Continuations break the computation down into a sequence of tiny steps.
  - Reasoning about such sequences is tedius and very sensitive to the details of the CPS evaluator.
- Often, we want to reason about blocks of code between `receive` operations that execute without exceptions:
  - Such blocks can be treated as atomic operations, and converted to more user-friendly expressions.
  - Can we do this automatically by rewriting?

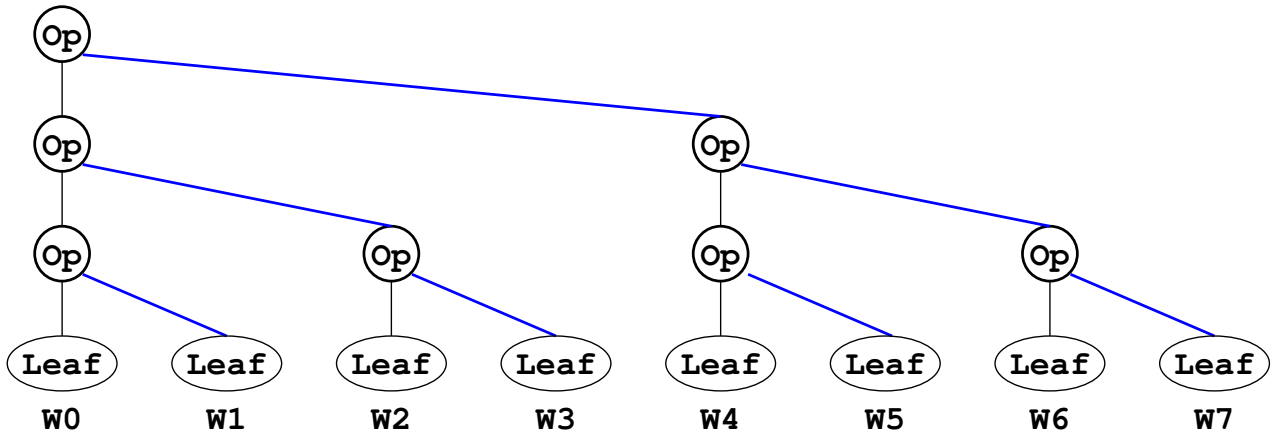
Rewriting CPS terms into pretty expressions

- We need an hypothesis that the CPS evaluation of an AST does not throw an exception – separate the reasoning exceptions from the "normal" execution path.
- For most expressions (without `try...catch` terms), a hypothesis that the top-level term doesn't throw an exception implies that no subterms throw exceptions.
- Rewriting doesn't derive that the no-exceptions hypothesis propagates down the expression tree:
  - Its a free variable term – how does ACL2 find which term is this subterms parent?
- Forward chaining works for one level of the tree,
  - Then the heuristic to decide whether or not to apply a forward-chaining rule blocks further propagation of the hypothesis.
  - We assume this is intended as a loop-breaker.

Proposed Solution

- Write a clause-processor to simply CPS terms.
- We have proof-of-concept version for a very tiny subset of Erlang.
- The general one will build on `rewrite$` using books that will be described in the next talk.

Example 2: Schwartz's Reduce



- The problem addressed by Reduce:
  - Some task is divided across `P` workers.
  - Each worker computes some result on its part of the task.
  - We need to combine these results.
- We consider a simple example, where workers simply send their indices up the tree, and the combine operation is $+$. This is equivalent to computing `sum_of_nats` for the number of workers.

Reduce: Proof Strategy

- We define `wtree-p` and `create-wtree`, a recognizer and constructor for processes that have the required tree structure.
- We define an invariant which states that the value held by the process when it reaches termination is the sum of the values at the leaves of the subtree for which the process is the root.
- We show that the invariant holds after each possible step of the scheduler.
- This is a work in progress:
  - The proofs work, but are very brittle.
  - The invariant causes many case splits, causing poor performance.
  - The reduce code is simplified quite a bit.
  - We will get back to this when we have a solution simplifying CPS terms into human-friendly terms as described above.

Conclusions and Future Work

- We can use CPS to model a concurrent program's execution.
  - We have built a CPS evaluator for Erlang in ACL2.
    - Continuations can capture all of Erlang's semantics.
    - We can use this evaluator to reason about sequential and concurrent Erlang programs. However, it is tedious and brittle.
    - The evaluator is executable by using `defattach` to provide an executable scheduler.
      - We have run the Erlang examples shown here.
  - This is still a work in progress.
    - Generating Lisp-equivalent functions for the evaluator can be an efficient proof strategy.
    - However, there are many issues with propagating hypotheses to subterms of partially symbolic ASTs.
- Questions?

Bibliography

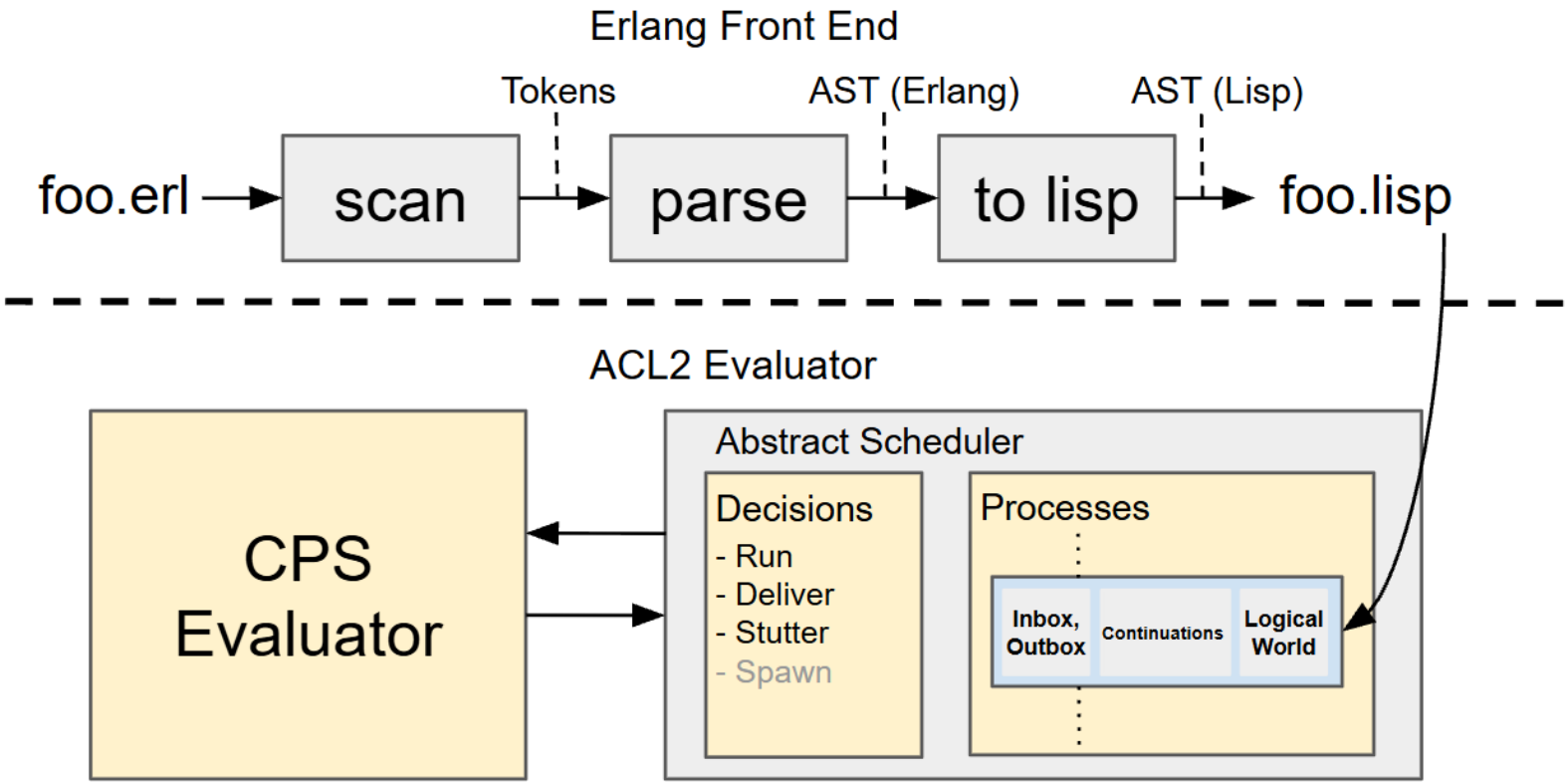
An abridged version of the bibliography in the paper.

[3] Andrew W. Appel (2007): *Compiling with Continuations*. Cambridge University Press, doi:10.1017/CBO9780511609619.

[4] Joe Armstrong (2007): A history of Erlang. In: *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, pp. 6-1–6-26, doi:10.1145/1238844.1238850.

[13] David Greve (2009): Assuming termination. In: *Proceedings of the Eighth International Workshop on the ACL2 Theorem Prover and its Applications*, pp. 114–122, doi:10.1145/1637837.1637856.

[19] Panagiotis Manolios & J Strother Moore (2003): Partial functions in ACL2. *Journal of Automated Reasoning* 31(2), pp. 107–127, doi:10.1023/B:JARS.0000009505.07087.34.


Erlang Front End
Tokens
AST (Erlang)
AST (Lisp)
foo.erl
scan
parse
to lisp
foo.lisp
ACL2 Evaluator
CPS
Evaluator
Abstract Scheduler
Decisions
- Run
- Deliver
- Stutter
- Spawn
Processes
Inbox,
Outbox
Continuations
Logical
World