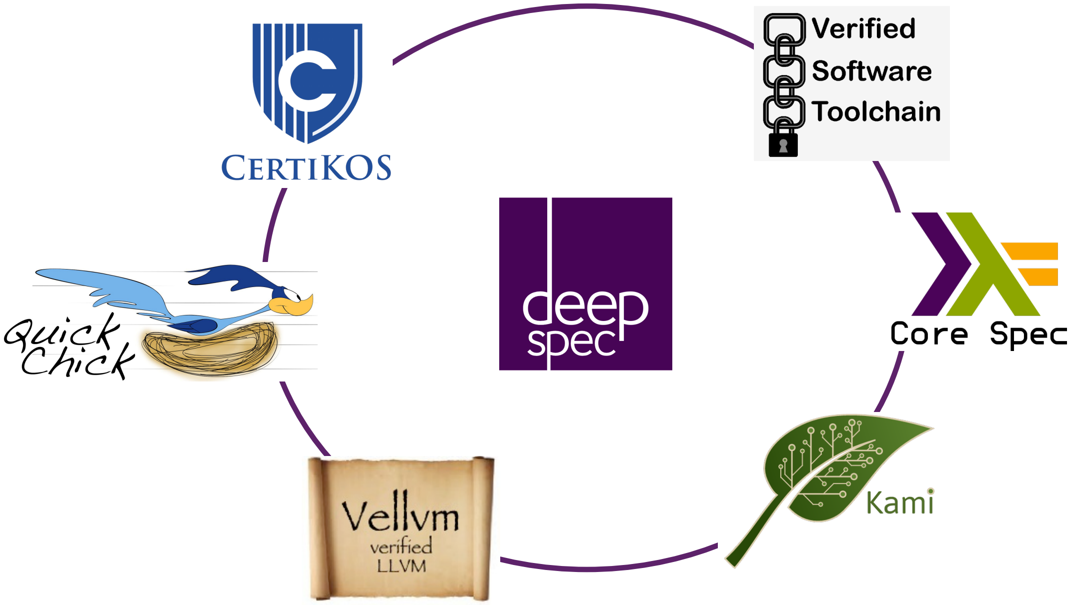
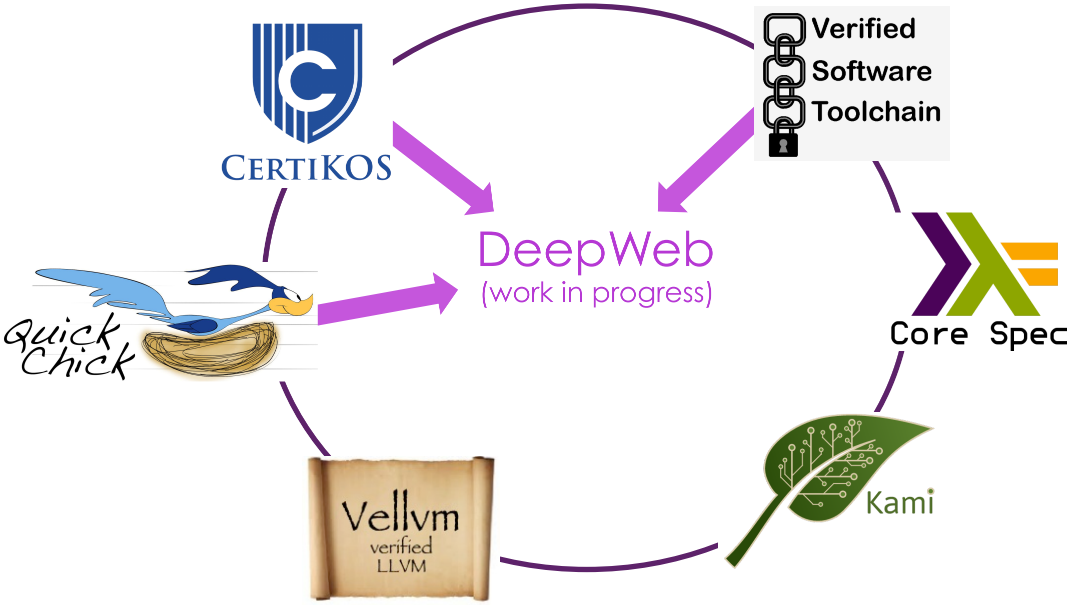
Interaction Trees in Coq

Steve Zdancewic
WG 2.8
2018



deepspec.org

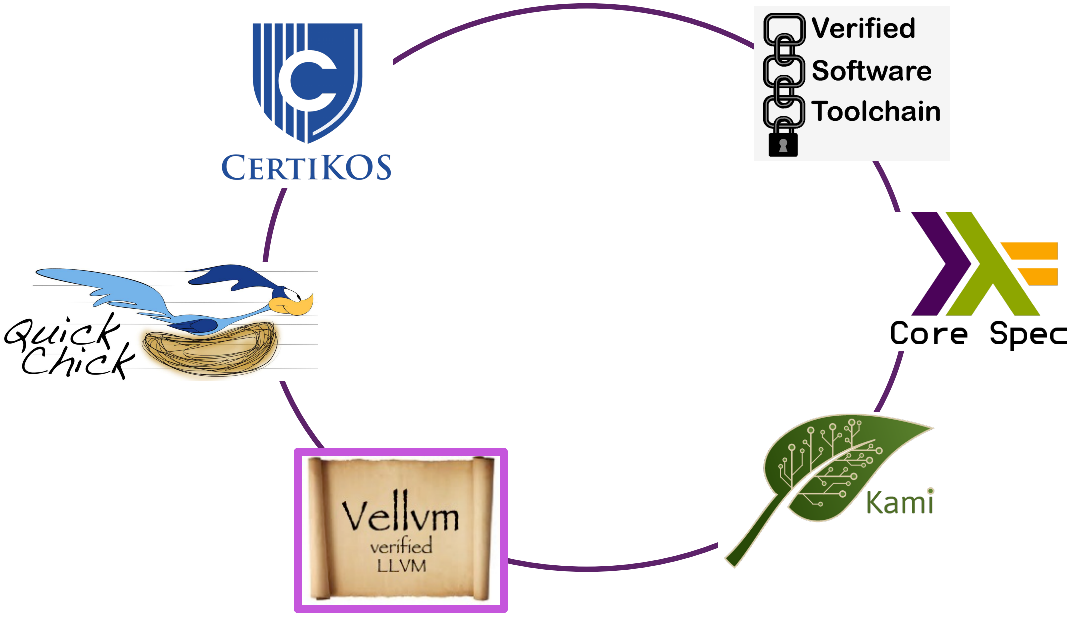
DeepSpec Integration Experiments



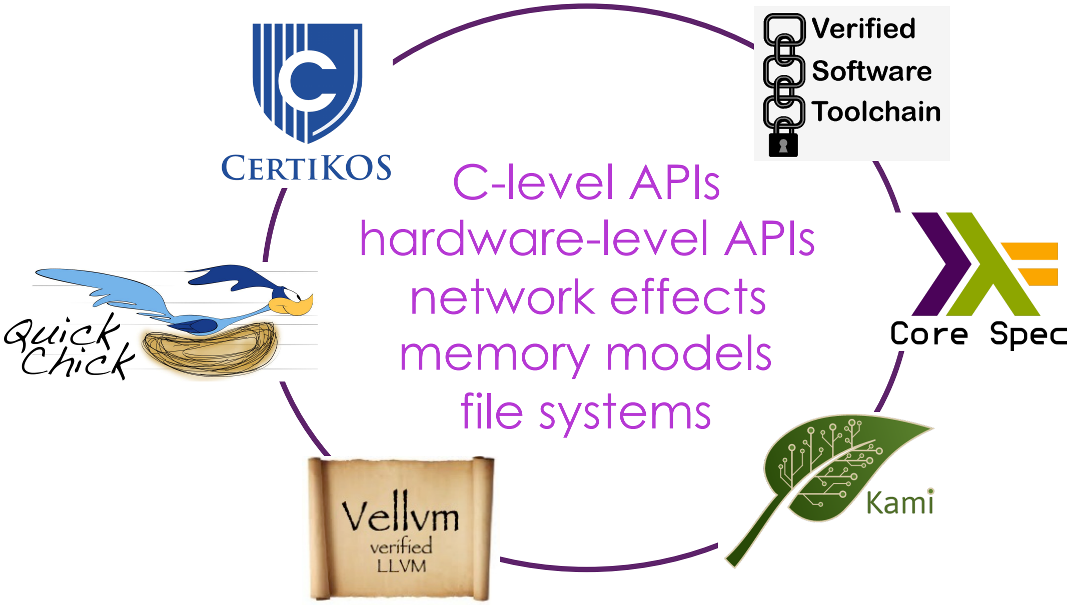
DeepWeb
(work in progress)

- (Eventual) Goal: web server implemented in C
- Running on top of CertiKOS
- Verified using VST
- Intermediate steps checked by QuickChick

Vellvm: Verified LLVM IR



- Compiler intermediate representation semantics
- Parameterized by the memory model
- github.com/vellvm/vellvm



C-level APIs
hardware-level APIs
network effects
memory models
file systems

- Coq descriptions of many different systems
- Need a common way of describing their behaviors
  - various levels of abstraction
  - different interfaces
  - modularity / extensibility

Interaction Trees

Coq adaptation of
Freer Monads, More Extensible Effects [Kiselyov & Ishii – 2015]

(see also: algebraic effects)

1. Explain Interaction Trees

2. Demo some "toy" examples in Coq

3. Come back to DeepSpec

```
CoInductive M (Event : Type -> Type) X :=
| Ret (x:X)
| Vis {Y: Type} (e : Event Y) (k : Y -> M Event X)
| Tau (k: M Event X)
| Err (s:string)
.
```

(potentially) inifinite structure

```
CoInductive M (Event : Type -> Type) X :=
| Ret (x:X)
| Vis {Y: Type} (e : Event Y) (k : Y -> M Event X)
| Tau (k: M Event X)
| Err (s:string)
.
```

named "M" (for "monad")

```
CoInductive M (Event : Type -> Type) X :=
| Ret (x:X)
| Vis {Y: Type} (e : Event Y) (k : Y -> M Event X)
| Tau (k: M Event X)
| Err (s:string)
.
```

parameterized by the type of
observable events

```
CoInductive M (Event : Type -> Type) X :=
| Ret (x:X)
| Vis {Y: Type} (e : Event Y) (k : Y -> M Event X)
| Tau (k: M Event X)
| Err (s:string)
.
```

yielding a
value of type X

```
CoInductive M (Event : Type -> Type) X :=
| Ret (x:X)
| Vis {Y: Type} (e : Event Y) (k : Y -> M Event X)
| Tau (k: M Event X)
| Err (s:string)
.
```

```
CoInductive M (Event : Type -> Type) X :=
| Ret (x:X)
| Vis {Y: Type} (e : Event Y) (k : Y -> M Event X)
| Tau (k: M Event X)
| Err (s:string)
.
```

yield a result (return of the monad)

```
CoInductive M (Event : Type -> Type) X :=
| Ret (x:X)
| Vis {Y: Type} (e : Event Y) (k : Y -> M Event X)
| Tau (k: M Event X)
| Err (s:string)
.
```

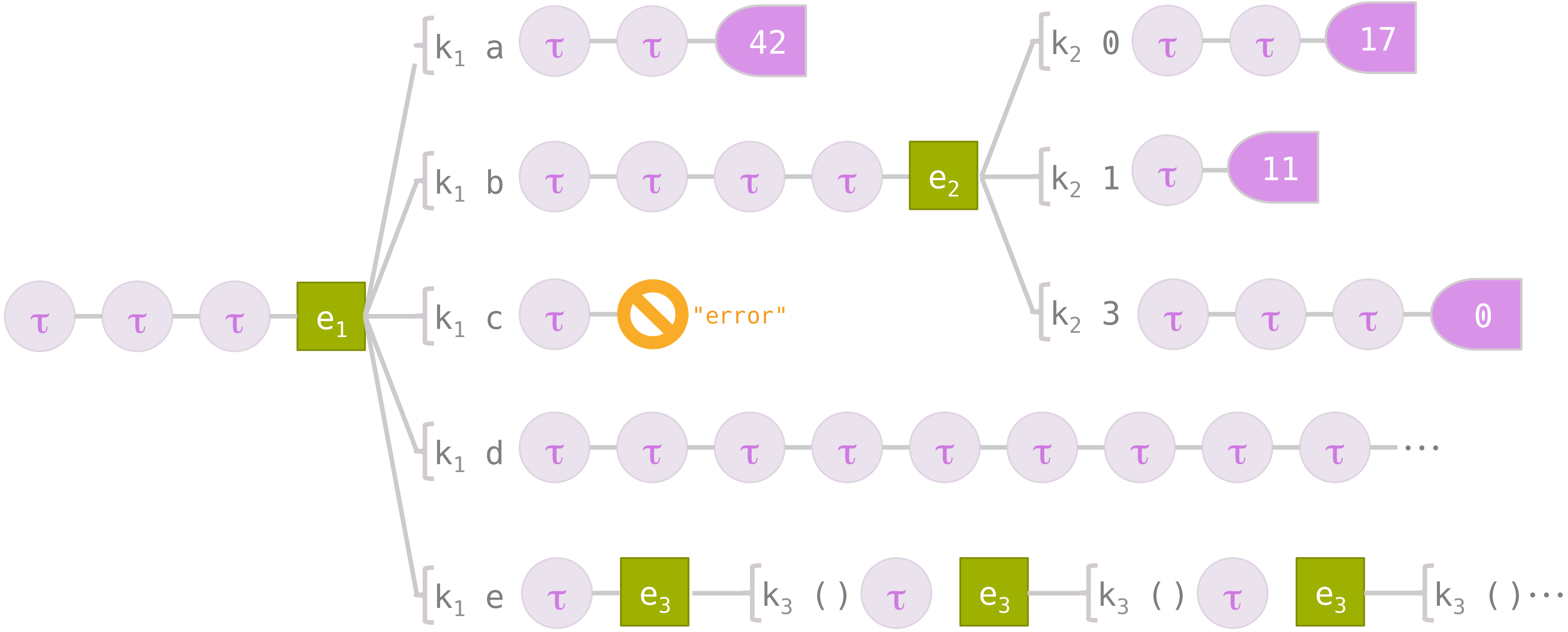
"visible" effect e
interacts with environment to get a value of type Y
k – the continuation that accepts the response

```
CoInductive M (Event : Type -> Type) X :=
| Ret (x:X)
| Vis {Y: Type} (e : Event Y) (k : Y -> M Event X)
| Tau (k: M Event X)
| Err (s:string)
.
```

internal, hidden step of computation

```
CoInductive M (Event : Type -> Type) X :=
| Ret (x:X)
| Vis {Y: Type} (e : Event Y) (k : Y -> M Event X)
| Tau (k: M Event X)
| Err (s:string)
.
```

error / aborted computation
(needed only for convenience)


τ
τ
τ
e_1
k_1 a
τ
τ
42
k_1 b
τ
τ
τ
τ
e_2
k_2 0
τ
τ
17
k_2 1
τ
11
k_2 3
τ
τ
τ
0
k_1 c
τ
"error"
k_1 d
τ
τ
τ
τ
τ
τ
τ
τ
τ
…
k_1 e
τ
e_3
k_3 ( )
τ
e_3
k_3 ( )
τ
e_3
k_3 ( )…

Good Qualities of Interaction Trees

- `(M E)` is a monad
  - bind is defined coinductively
- Behavioral Equivalences
  - strong bisimulation
  - up to Tau (insert a finite no. of Tau's anywhere)
  - not too hard to define new simulation relations
- Extractable from Coq
  - yields a way of (externally) running computations described by interaction trees
  - interpretation of events can be defined in the metalanguage (e.g. OCaml)

(demo)

Applications

- Vellvm Semantics
  - control-flow graphs, LLVM memory model
- DeepWeb
  - web server events (HTTP get/put)
- Verifiable Software Toolchain
  - socket API

LLMV IR Memory Model

```
(* IO interactions for the LLVM IR *)
Inductive IO : Type -> Type :=
| Alloca : ∀ (t:dtyp), (IO dvalue)
| Load   : ∀ (t:dtyp) (a:dvalue), (IO dvalue)
| Store  : ∀ (a:dvalue) (v:dvalue), (IO unit)
| GEP    : ∀ (t:dtyp) (v:dvalue) (vs:list dvalue), (IO dvalue)
| ItoP   : ∀ (i:dvalue), (IO dvalue)
| PtoI   : ∀ (a:dvalue), (IO dvalue)
| Call   : ∀ (f:string) (args:list dvalue), (IO dvalue)
.
```

"outputs" of the Call event

type of the result

Network IO

```
(* IO interactions for sockets *)
Inductive networkE : Type -> Type :=
| Listen : endpoint_id -> networkE unit
| Accept : endpoint_id -> networkE connection_id
| ConnectTo : endpoint_id -> networkE connection_id
| CloseConn : connection_id -> networkE unit
| Recv : connection_id -> positive -> networkE (option string)
| Send : connection_id -> string -> networkE unit
.
```

OS-level API

```
(* OS-level refinement of Network-level Spec *)
Inductive SocketAPI1 : Type -> Type :=
  | Socket_Socket (domain : Z) (type : Z) (protocol : Z) :
                   SocketAPI1 (SocketError + sockfd)
  | Socket_Close (fd : sockfd): SocketAPI1 (SocketError + unit)
  | Socket_BindAndListen (fd : sockfd) : SocketAPI1 (SocketError + unit)
  | Socket_Accept (fd : sockfd) : SocketAPI1 (SocketError + sockfd)
  | Socket_Recv (fd : sockfd) (num_bytes : Z):
      SocketAPI1 (SocketError + string)
  | Socket_Send (fd : sockfd) (msg : string):
      SocketAPI1 (SocketError + unit)
.
```

Fancier IO Specs

- Combinators at the Event level
  - to "mix and match" behaviors
  - made palatable via typeclasses

```
(* Example: combine nondeterminism, failure, Sockets *)

Definition SocketM (T : Type) :=
    (nondetE +' failureE +' SocketAPI.SocketAPI1) T.
```

Uses

- Writing effectful programs in Coq
- Giving specifications by "zipping"
  - relating a "client" to a "server"
  - for testing / proving
- Transducers: change levels of abstraction
  - e.g. from "high-level" LLVM memory model to "low-level" machine model

Technical Challenges

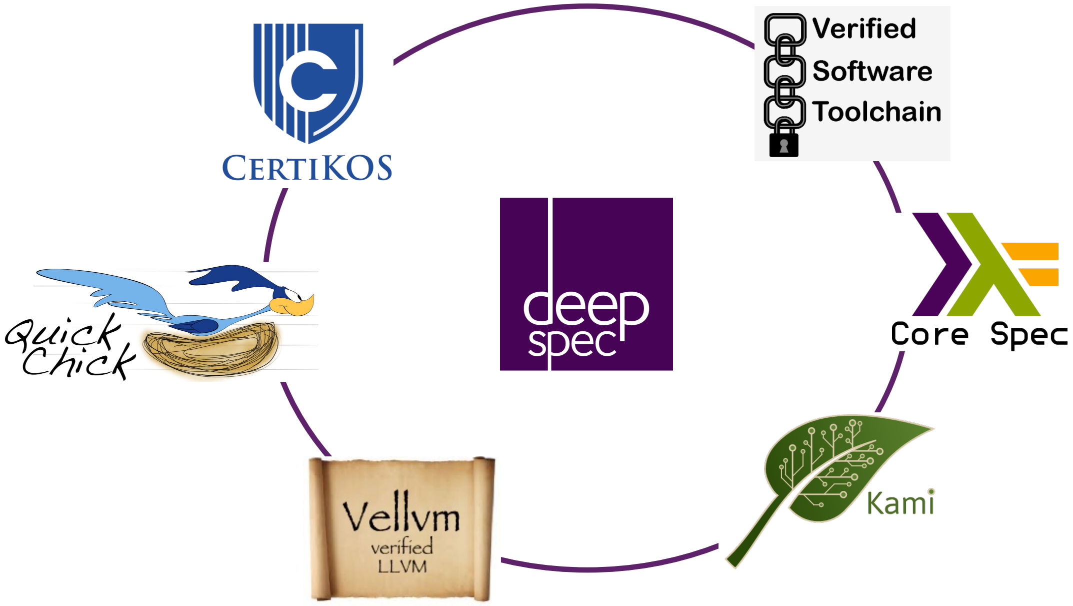
- Coinduction in Coq
  - syntactic productivity constraints are a pain
  - Gil Hur's paco library helps (somewhat)

- Proofs of some basic facts surprisingly tricky to prove
  - e.g. congruence of bind up to Tau
  - several possible ways to define EquivUpToTau

Work in Progress

- Library & automation support for Interaction Trees
- Vellvm proofs about more complex memory models
  - int2ptr / ptr2int
- VST: Semantics in CompCert
  - Interaction trees as "ghost state" in separation logic

Interaction Trees are Fun



deepspec.org

```
Definition bind_body {E X Y}
           (s : M E X)
           (go : M E X -> M E Y)
           (t : X -> M E Y) : M E Y :=
  match s with
  | Ret x => t x
  | Vis e k => Vis e (fun y => go (k y))
  | Tau k => Tau (go k)
  | Err s => Err s
  end.

Definition bindM {E X Y}
           (s: M E X)
           (t: X -> M E Y) : M E Y :=
  (cofix go (s : M E X) :=
      bind_body s go t) s.
```