Modular Semantics for LLVM IR

Steve Zdancewic
DeepSpec @ PLDI
2018

Collaborators

- Lennart Berringer
- Joachim Breitner
- Dmitri Garbuzov
- Olek Gierczak
- Gil Hur
- Jeehon Kang
- Nicolas Koh
- Yao Li
- Yishuai Li
- William Mansky
- Milo M.K. Martin
- Santosh Nagarakatte
- Benjamin Pierce
- Christine Rizkallah
- Viktor Vafeiadis
- Li-yao Xia
- Yannick Zakowski
- Jianzhou Zhao

CertiKOS

Verified Software Toolchain

QuickChick

deep spec

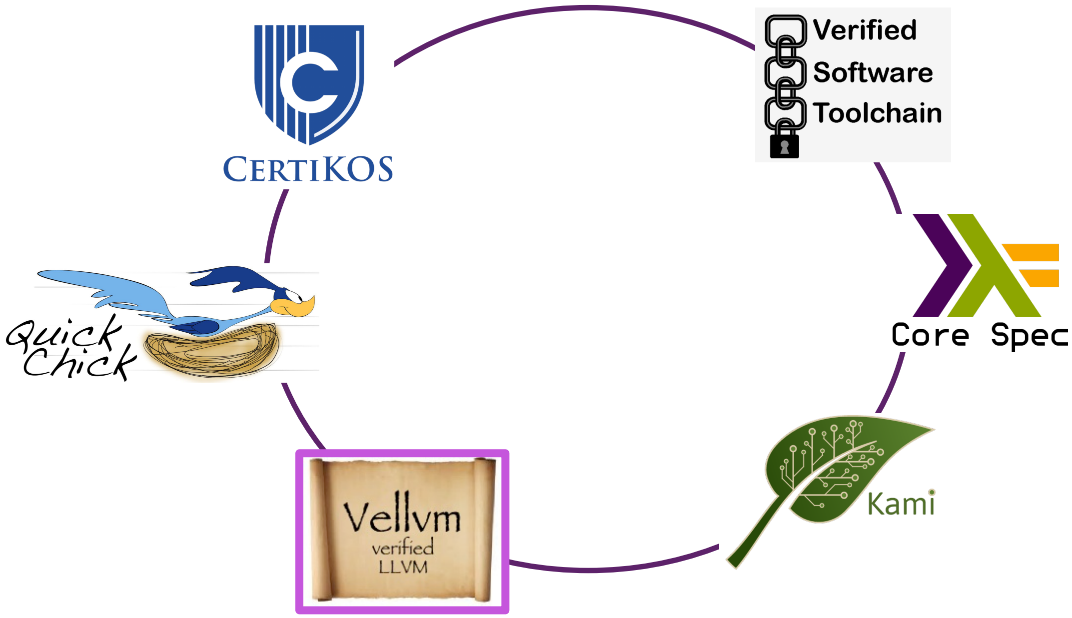
Core Spec

Vellvm verified LLVM

Kami

deepspec.org

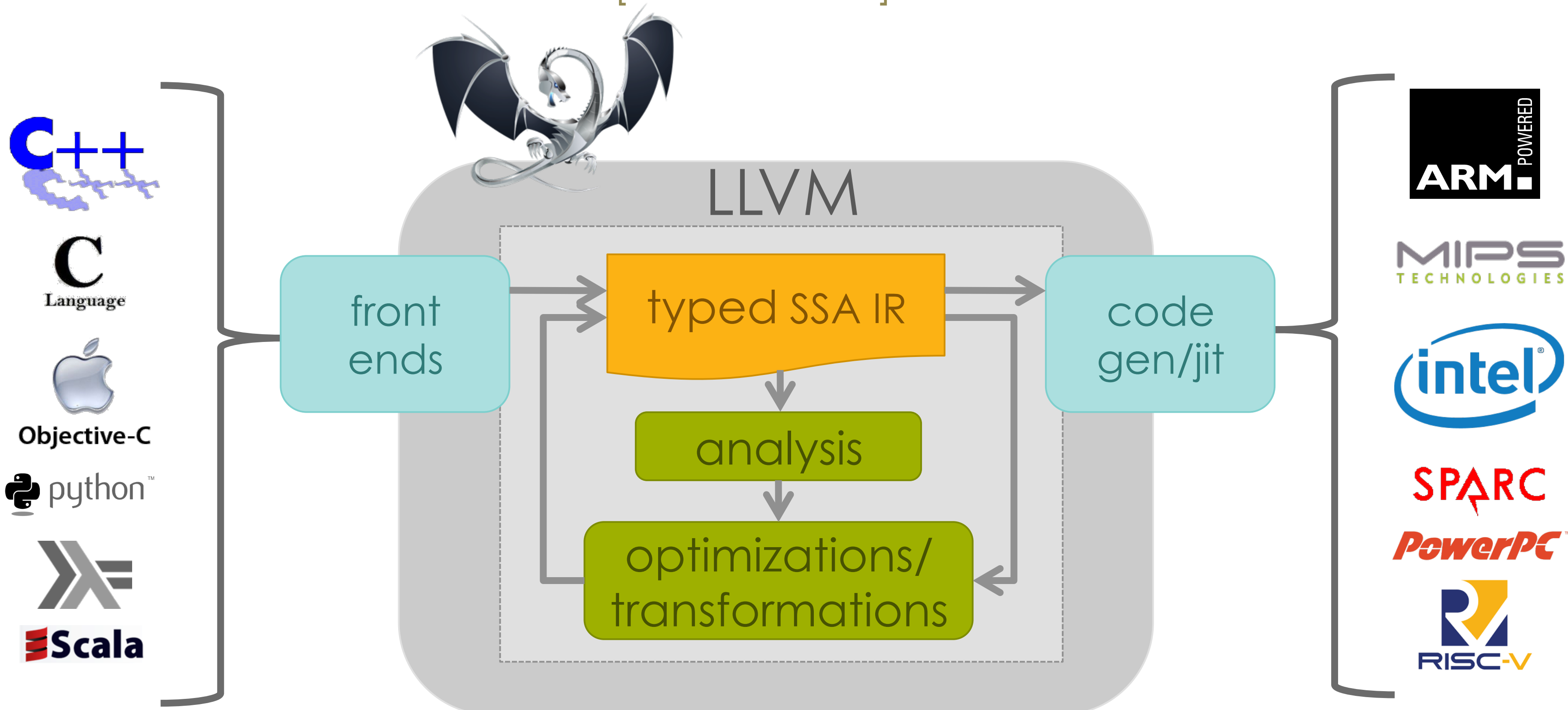
Vellvm: Verified LLVM IR



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

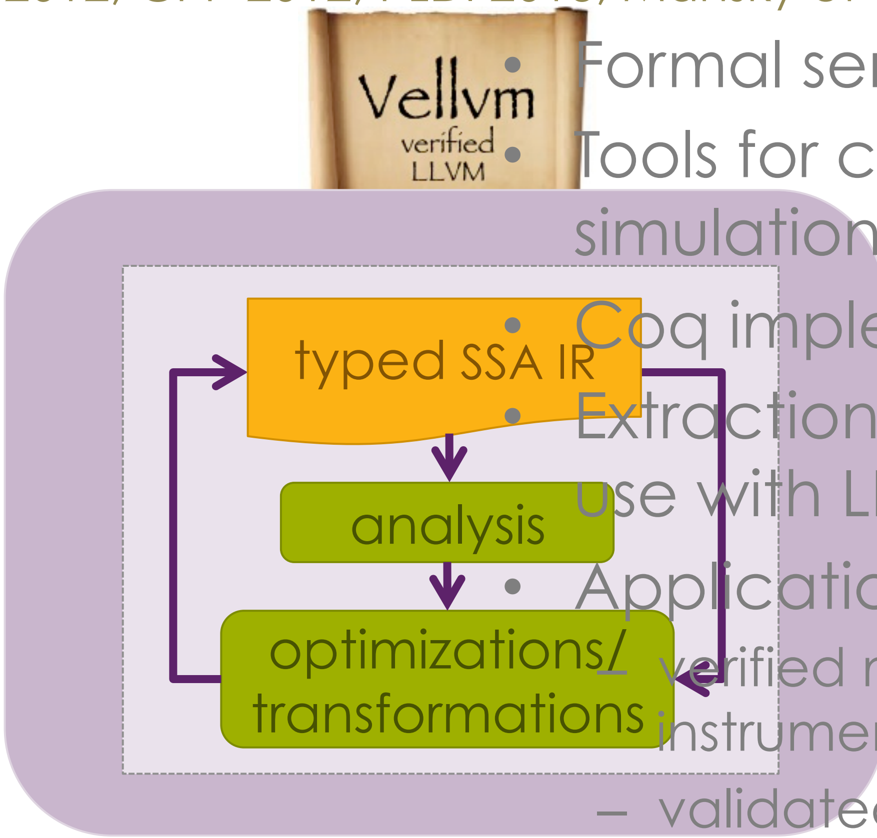
LLVM Compiler Infrastructure

[Lattner et al.]



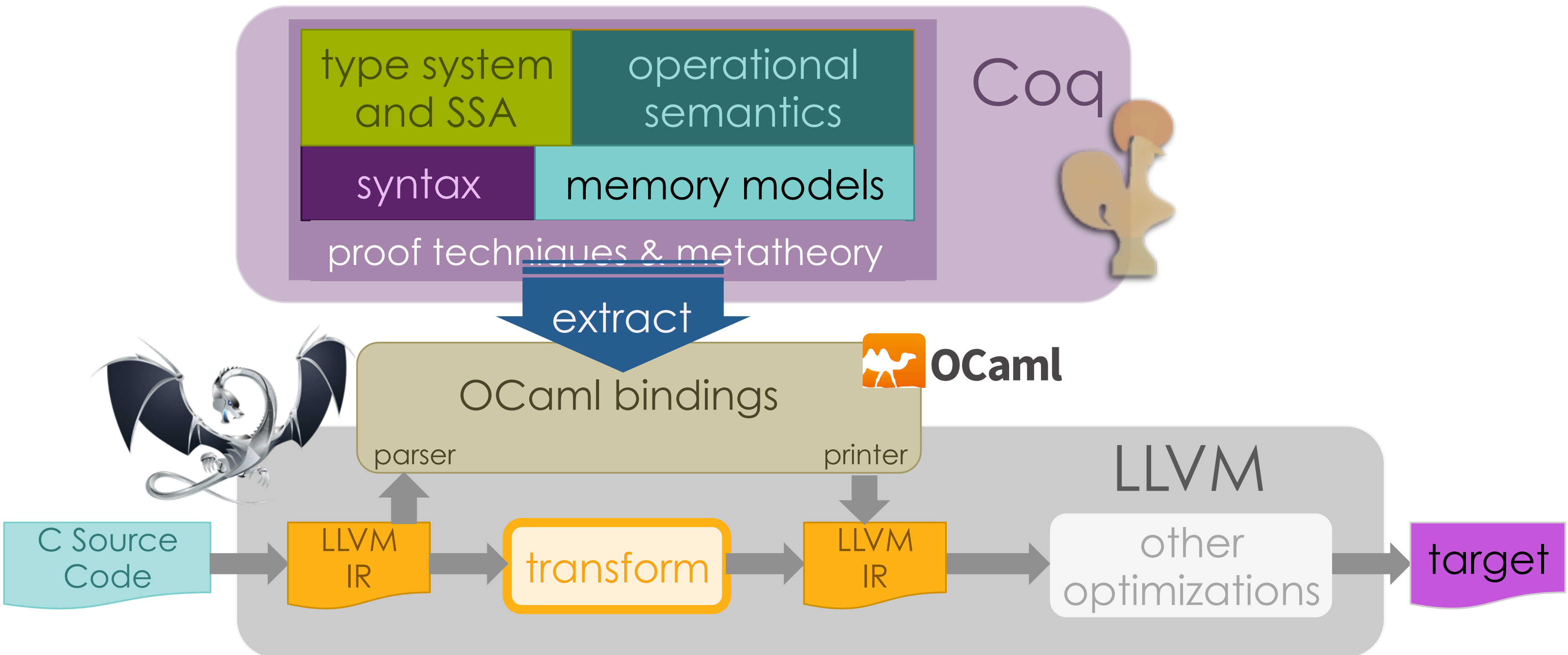
The Vellvm Project

[Zhao et al. POPL 2012, CPP 2012, PLDI 2013, Mansky et al. CAV2015]

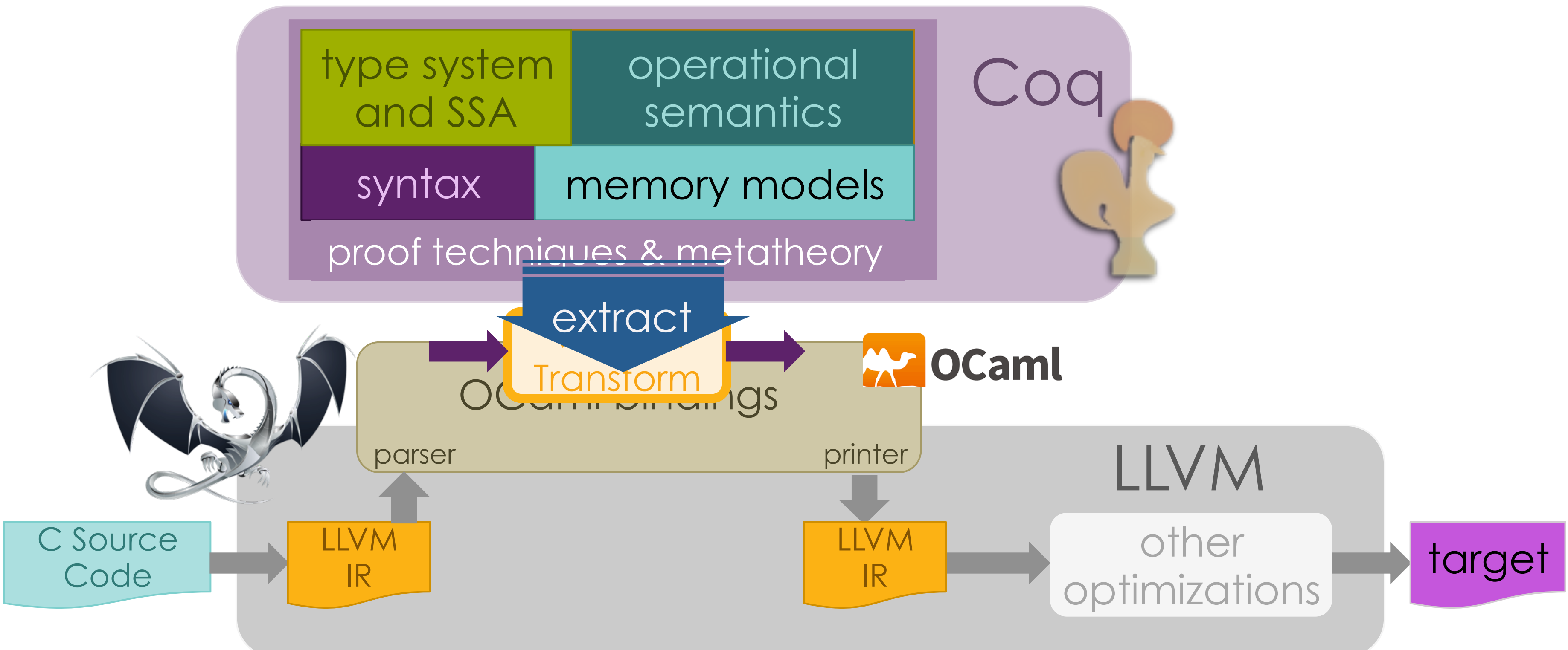


- Formal semantics
- Tools for creating simulation proofs
- Coq implementation
- Extraction of passes for use with LLVM compiler
- Applications:
  - verified memory safety instrumentation
  - validated optimizations

Vellvm Framework



Vellvm Framework



LLVM IR Semantics

SSA ≈ functional program
[Kelsey 1995 / Appel 1998]
+

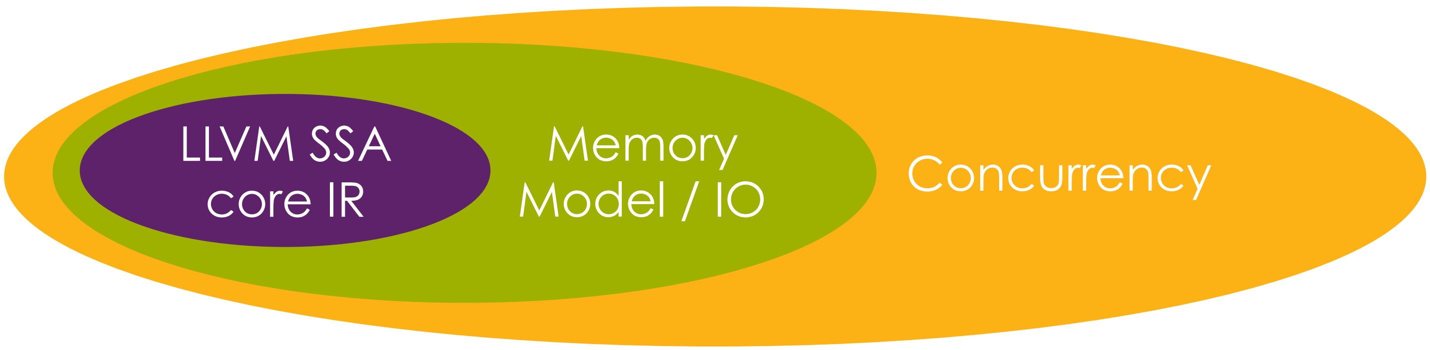
- Undefined values / poison
- Effects
  - structured heap load/store
  - system calls (I/O)
- Types & Memory Layout
  - structured, recursive types
  - type-directed projection
  - type casts

We know how to model this and prove properties about the models.

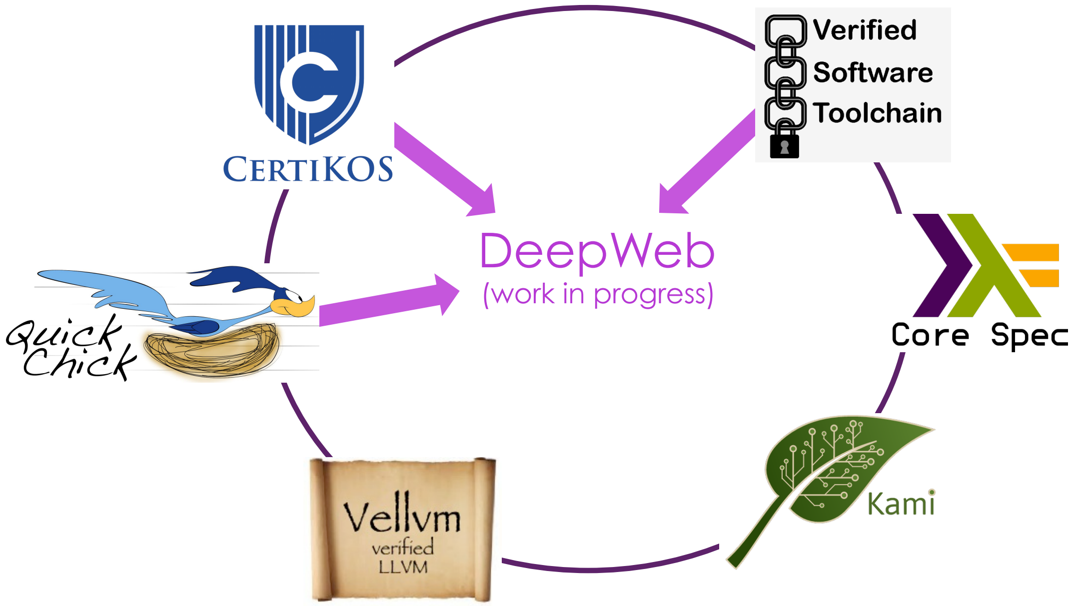
We are starting to figure out how to decompose them *modularly*.

Modular Semantics

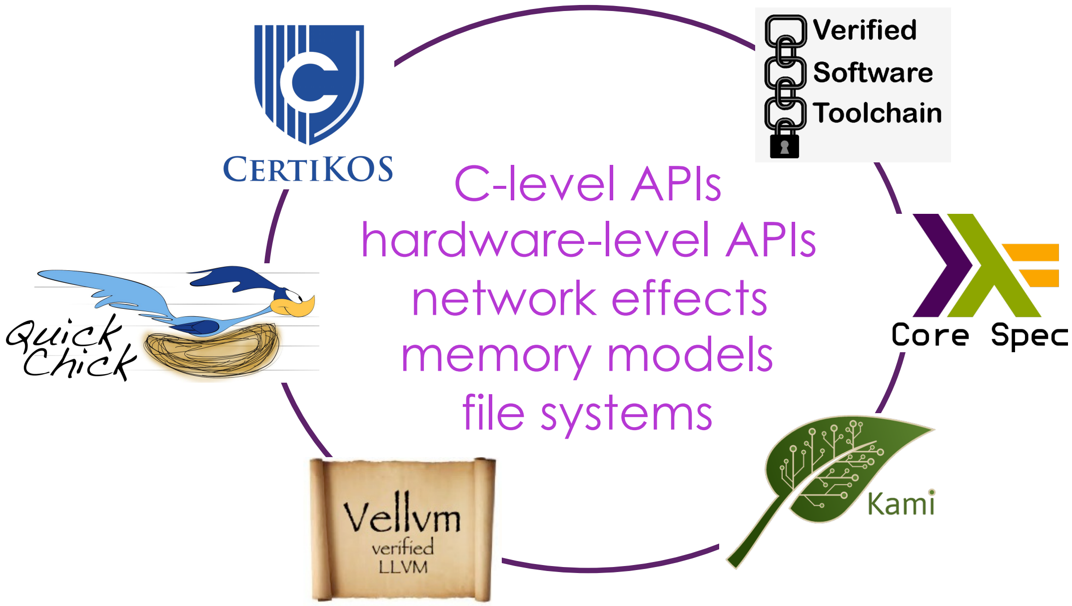
- Factor out memory model [CAV 15]
  - similar to linking/separate compilation
- For:
  - concurrency
  - more extensibility/robustness to changes
  - better support for casts [PLDI 15]



DeepSpec Integration Experiments



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



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

1. Explain Interaction Trees

2. Describe "toy" Vellvm

3. Come back to DeepSpec

Interaction Trees

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

(see also: algebraic effects)

```
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)
.
```

(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)
.
```

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)
.
```

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)
.
```

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)
.
```

```
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)
.
```

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)
.
```

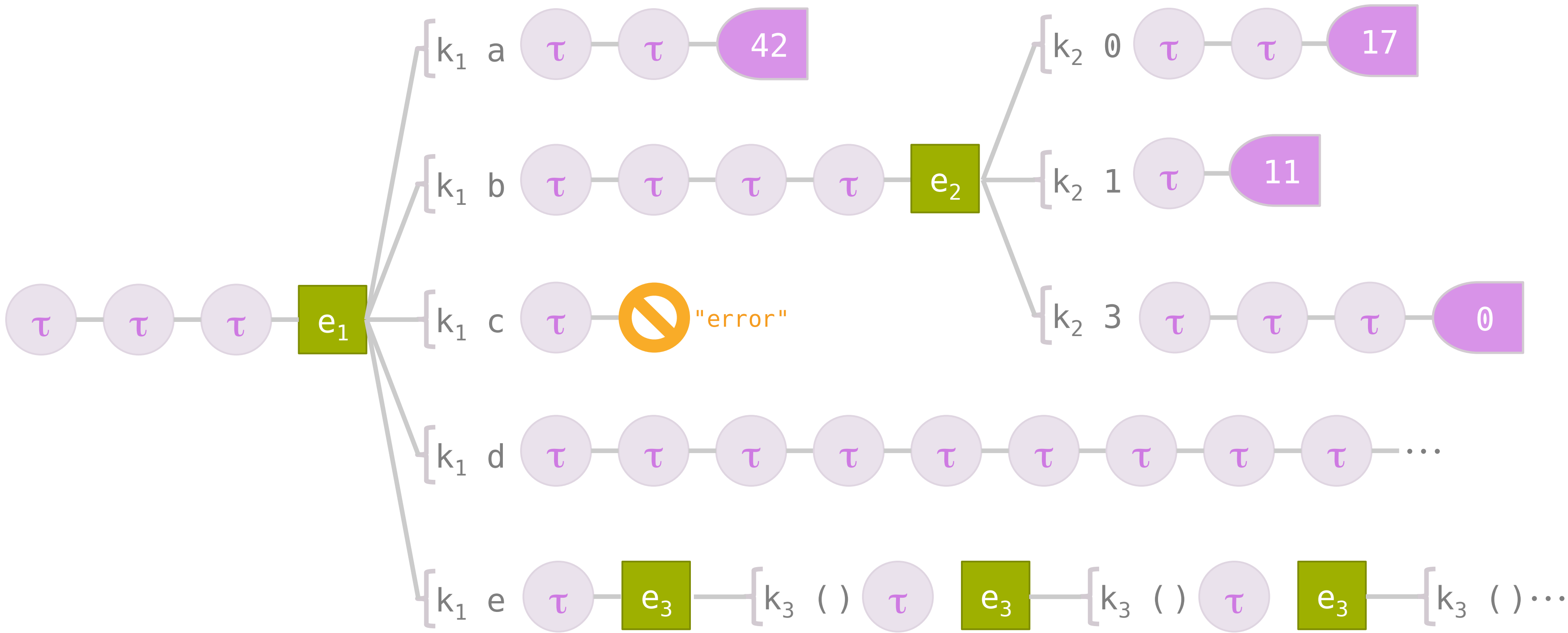
- "visible" effect e
  interacts with environment to get a value of type Y
  k – the continuation that accepts the response

```
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)
.
```

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)

Applications

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

1. Explain Interaction Trees
2. Describe "toy" Vellvm
3. Come back to DeepSpec

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)
.
```

output values of the Call event

type of the result provided by the environment

Memory Interaction Interface

```
Module Mem.

  Definition addr := nat.
  Inductive val :=  | N (n:nat)  | A (a:addr)  .

  Inductive IO : Type -> Type :=
  | Alloca : IO addr
  | Load   : ∀ (a:addr), (IO val)
  | Store  : ∀ (a:addr) (v:val), (IO unit)
  | Call   : ∀ (f:funid) (v:val), (IO val)
  .

  Definition Interactions (X:Type) := M IO X.

  Global Instance functor_IO : Functor Interactions := (@mapM IO).
  Global Instance monad_IO : (@Monad Interactions) (@mapM IO) := ….
  Global Instance equiv_ixns {X} : (@Equiv (Interactions X)) := EquivUpToTau eq.
End Mem.
```

Operations (simplified)

Memory Interaction Interface

```
Module Mem.

  Definition addr := nat.
  Inductive val :=  | N (n:nat)  | A (a:addr)  .

  Inductive IO : Type -> Type :=
  | Alloca : IO addr
  | Load   : ∀ (a:addr), (IO val)
  | Store  : ∀ (a:addr) (v:val), (IO unit)
  | Call   : ∀ (f:funid) (v:val), (IO val)
  .

  Definition Interactions (X:Type) := M IO X.

  Global Instance functor_IO : Functor Interactions := (@mapM IO).
  Global Instance monad_IO : (@Monad Interactions) (@mapM IO) := ….
  Global Instance equiv_ixns {X} : (@Equiv (Interactions X)) := EquivUpToTau eq.
End Mem.
```

instantiate the monad

External Interface

```
Module Ext.
  Inductive IO : Type -> Type :=
  | Call    : ∀ (f:funid) (v:Mem.val), (IO Mem.val)
  .

  Definition Interactions (X:Type) := M IO X.

  Global Instance functor_IO : Functor Interactions := (@mapM IO).
  Global Instance monad_IO   : (@Monad Interactions) (@mapM IO) := ….
  Global Instance equiv_ixns {X} : (@Equiv (Interactions X)) := EquivUpToTau eq.
End Ext.
```

Interface contains only the external call

Memory Model

Transduces memory interactions to external traces.

```
Definition MemoryModel :=
      ∀ X, Mem.Interactions X -> Ext.Interactions X.
```

MM Implementation

```
Definition memory := list Mem.val.
Definition load (m:memory) (a:nat) := nth_default (Mem.N 0) m a.
Definition store (m:memory) := replace m.


(* Effects handlers *)
Definition mem_step X (io : Mem.IO X) m :  (memory * X) + (Ext.IO X) :=
  match io with
  | Mem.Alloca    => inl (m ++ [Mem.N 0], (length m))
  | Mem.Load a    => inl (m, load m a)
  | Mem.Store a v => inl (store m a v, ())
  | Mem.Call f v  => inr (Ext.Call f v)
  end.
```

```
(* Map the handler over the trace *)
Definition mem (m:memory) : MemoryModel :=
  fun X t =>
    (cofix go (m:memory) t :=
       match t with
       | Tau d'    => Tau (go m d')
       | Vis io k =>
         match mem_step io m with
         | inl (m', v) => Tau (go m' (k v))
         | inr eo      => Vis eo (fun v => go m (k v))
         end
       | Ret x => Ret x
       | Err x => Err x
       end) m t.
```

interpret the effects

Metatheory

```
Lemma mem_eutt : ∀ X (s t : Mem.Interactions X),
    s ≡ t ->
    ∀ m, (mem m s) ≡ (mem m t).

Lemma mem_swap :  ∀ {X} m a b v1 v2 (k1 k2 : () -> Mem.Interactions X),
    a <> b ->
    (k1 ()) ≡ (k2 ()) ->
    (mem m (Vis (Mem.Store a v1) (fun _ => Vis (Mem.Store b v2) k1)))
      ≡
    (mem m (Vis (Mem.Store b v2) (fun _ => Vis (Mem.Store a v1) k2))).

Lemma mem_load2 : ∀ {X} m a (k : Mem.val -> Mem.val -> Mem.Interactions X),
    (mem m (Vis (Mem.Load a) (fun v => Vis (Mem.Load a) (fun w => k v w))))
      ≡
    (mem m (Vis (Mem.Load a) (fun w => k w w))).
```

Programming Language / IR

```
Inductive cmd :=
| LET (v:var) (e:exp)
| ALLOCA (x:var)
| LOAD  (x:var) (e:exp)
| STORE (e1:exp) (e2:exp)
| CBR (e:exp) (b1 b2:list cmd)
| …
.
```

Define IR language using standard techniques…
(mostly omitted here)

Operational Semantics

```
Definition interpret (e:env) (code:list cmd) : Mem.Interactions unit :=
  (cofix go (e:env) (code:list cmd) : Mem.Interactions unit :=
     match code with
     | [] => Ret ()

     | LET x exp :: k =>
       do v <- eval e exp;
       Tau (go (add e x v) k)

     | LOAD x exp :: k =>
       do a <- eval e exp;
       match a with
       | Mem.A a => Vis (Mem.Load a) (fun v => go (add e x v) k)
       | _ => raise "loaded non-address"
       end

…
```

Interpret the program in the interaction tree monad

Operational Semantics

```
Definition interpret (e:env) (code:list cmd) : Mem.Interactions unit :=
  (cofix go (e:env) (code:list cmd) : Mem.Interactions unit :=
     match code with
     | [] => Ret ()

     | LET x exp :: k =>
       do v <- eval e exp;
       Tau (go (add e x v) k)

     | LOAD x exp :: k =>
       do a <- eval e exp;
       match a with
       | Mem.A a => Vis (Mem.Load a) (fun v => go (add e x v) k)
       | _ => raise "loaded non-address"
       end
```

Use Tau to "hide" internal steps

…

Operational Semantics

```
Definition interpret (e:env) (code:list cmd) : Mem.Interactions unit :=
  (cofix go (e:env) (code:list cmd) : Mem.Interactions unit :=
     match code with
     | [] => Ret ()

     | LET x exp :: k =>
       do v <- eval e exp;
       Tau (go (add e x v) k)

     | LOAD x exp :: k =>
       do a <- eval e exp;
       match a with
       | Mem.A a => Vis (Mem.Load a) (fun v => go (add e x v) k)
       | _ => raise "loaded non-address"
       end

  …
```

Expose other effects like "load" using Vis.

```
Lemma load_elim : ∀ x y ex m e code,
    not_used_in x ex ->
    (mem m (interpret e ([LOAD x ex; LOAD y ex]++code)))
      ≡
    (mem m (interpret e ([LOAD x ex; LET y (Var x)]++code))).
```

In this (sequential) memory model, two loads of the same address can be replaced by one load and a "let".

(demo)

1. Explain Interaction Trees

2. Describe "toy" Vellvm

3. Come back to DeepSpec

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

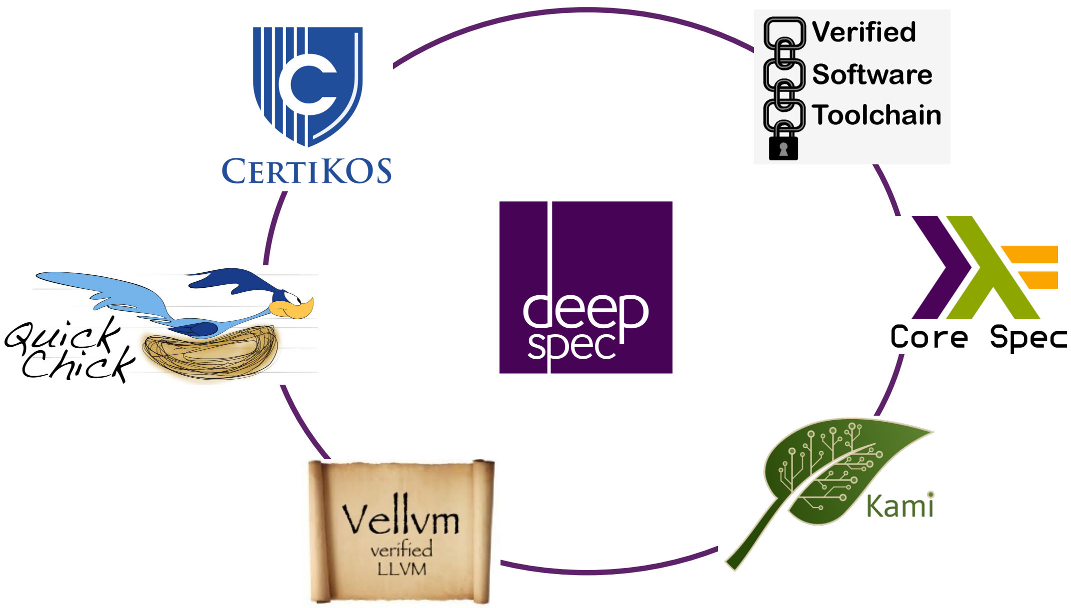
- Writing effectful programs in Coq
- Giving specifications by "zipping"
  - relating a "client" to a "server"
  - for testing / proving
- Transducers: change levels of abstraction
  - e.g. from "high-level" LLVM memory model to "low-level" machine model

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

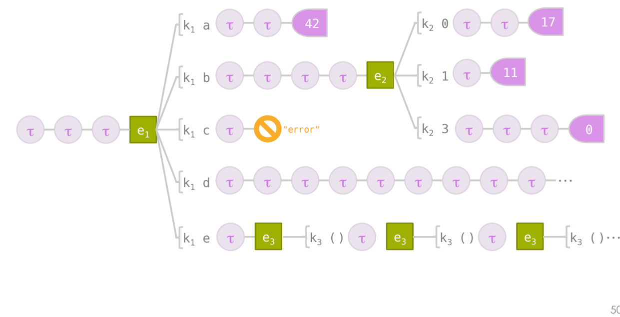
Work in Progress

- Library & automation support for Interaction Trees
- Verfied Software Toolchain
  - Interaction trees as "ghost state" in separation logic
  - connection to CompCert
- Vellvm proofs about more complex memory models
  - int2ptr / ptr2int
- Using the new Vellvm semantics for interesting optimizations



deepspec.org

github.com/vellvm/vellvm



Interaction Trees

promising abstraction for
Coq formalization
of interactive behaviors

```
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.
```