National Science Foundation Expedition in Computing

the science of deep specification

Verification of interacting systems in VST

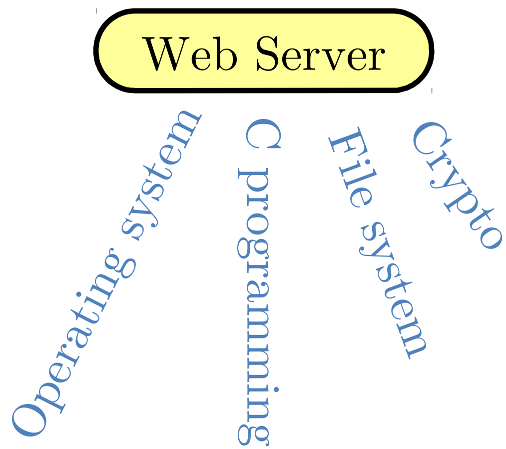
a.k.a Towards a verified web server, part II

<u>Lennart Beringer</u>, William Mansky

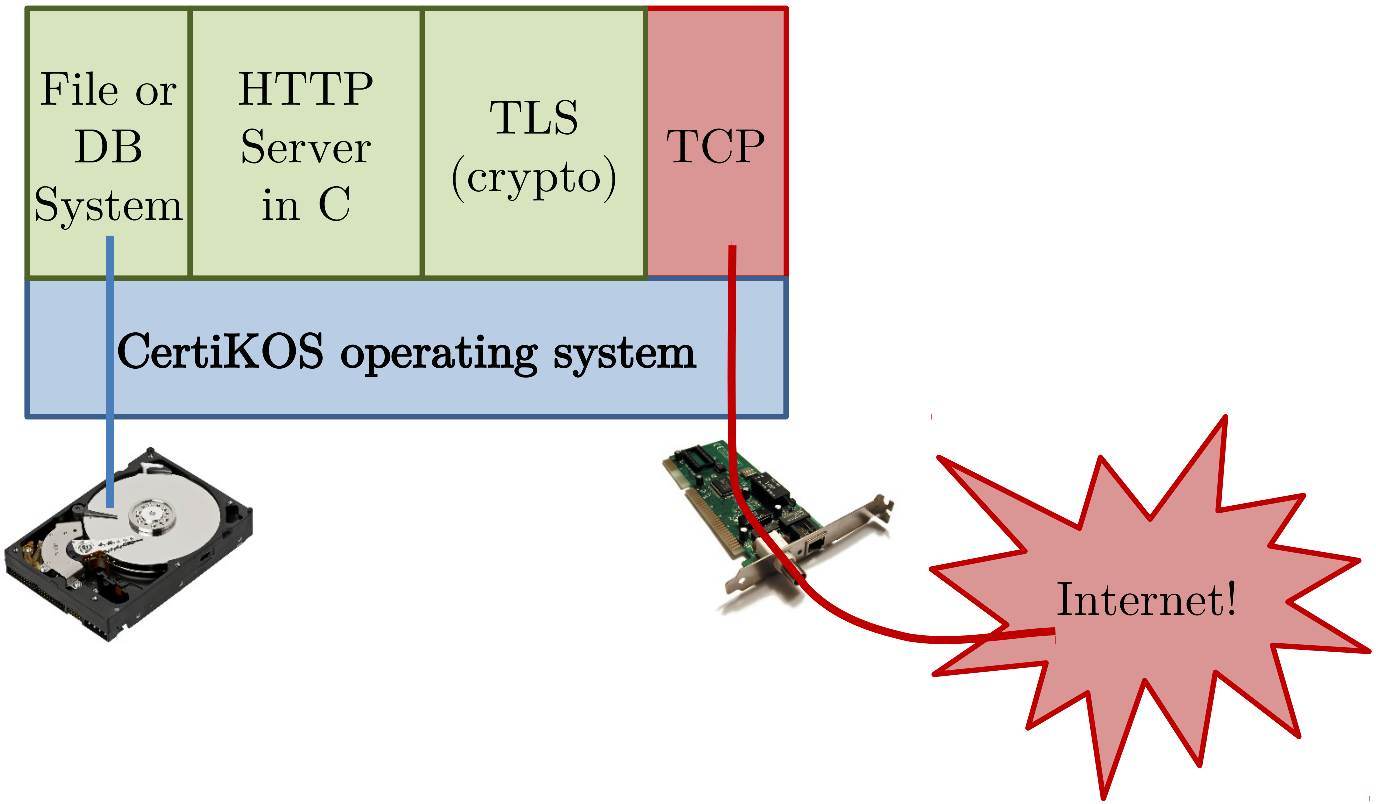
Joachim Breitner, **Nicolas Koh**, **Yao Li**, **Yishuai Li**, Benjamin C. Pierce, **Li-Yao Xia**, Steve Zdancewic

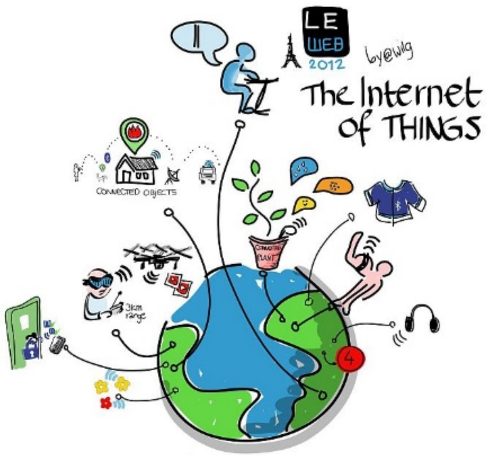
Wolf Honore

Demo vehicle for DeepSpec Techniques

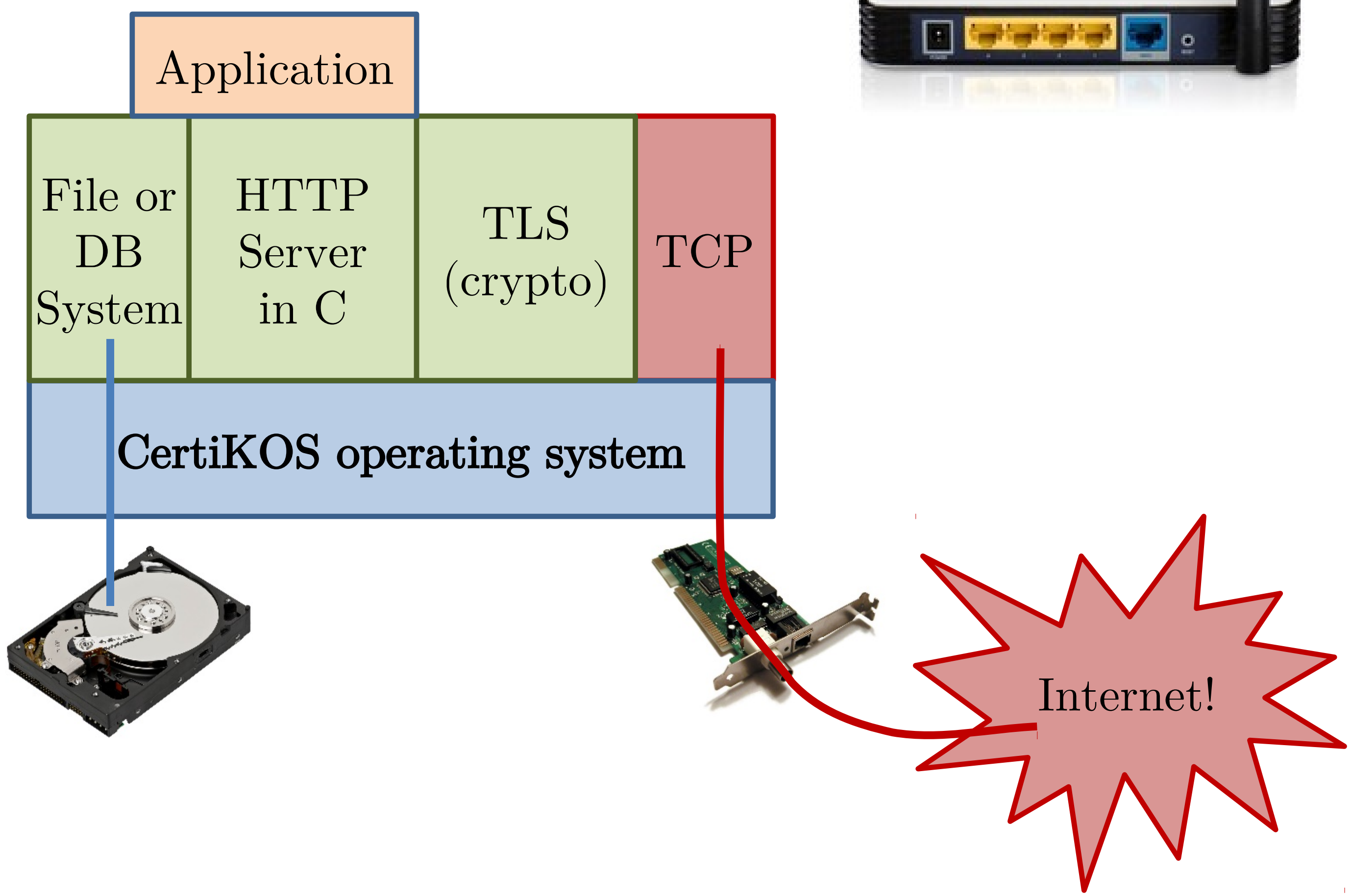


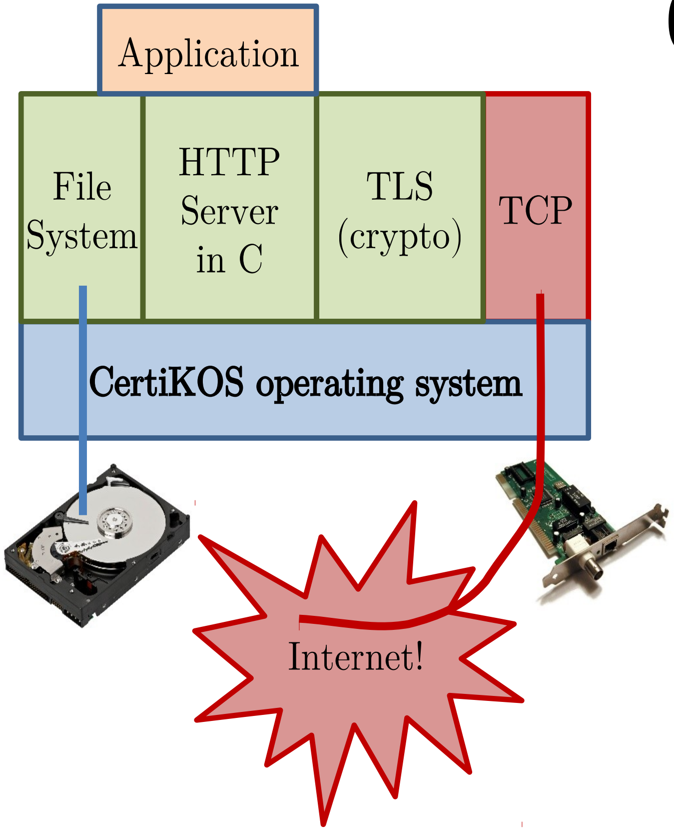
DeepSpec Web Server





Server as a library

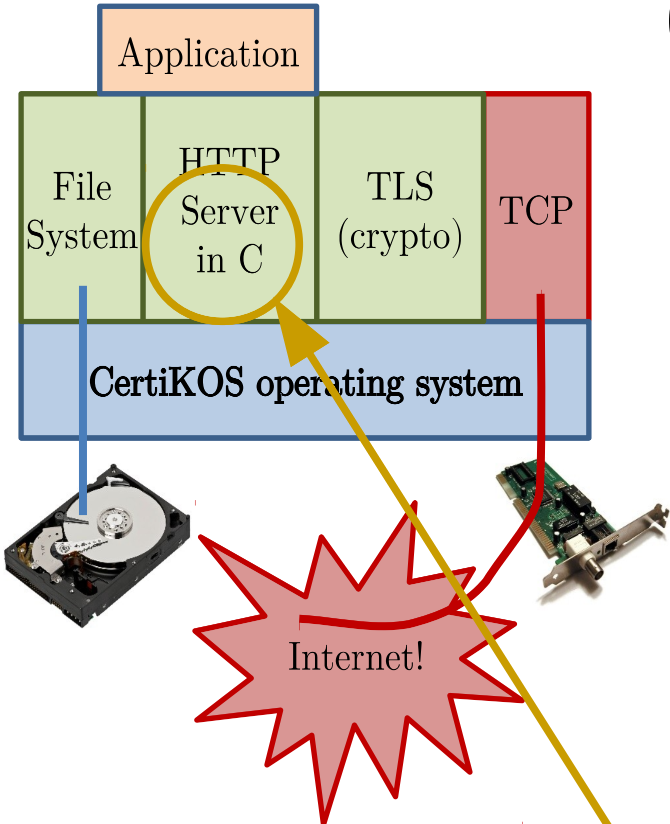




Challenges

- complexity / code size
- interfaces between components at best informally / intuitively specified
- specifications based on internal view of implementations unconnected to specification of externally visible behavior (messages sent on network)
- ...

Challenges

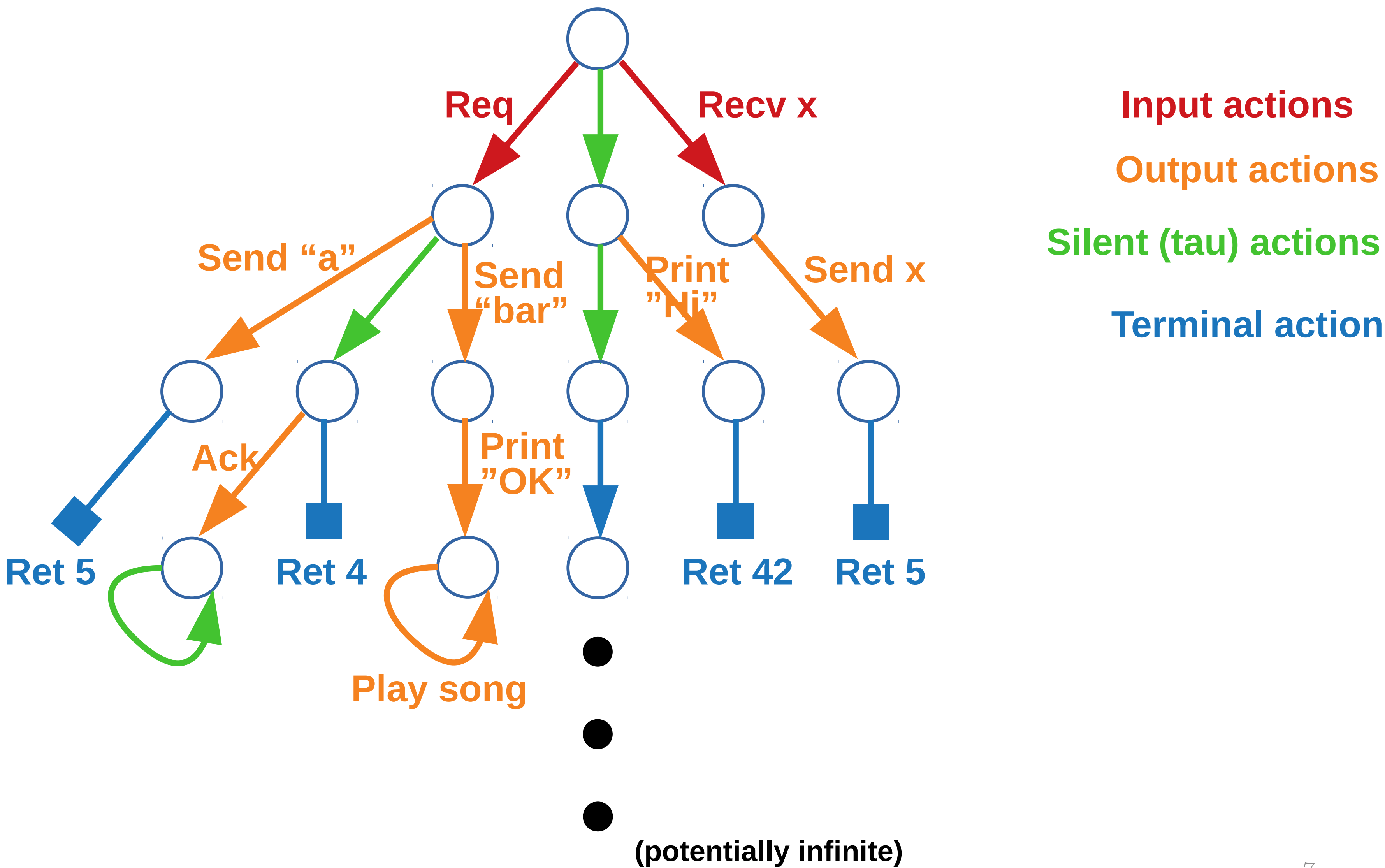


- complexity / code size
- interfaces between components at best informally / intuitively specified
- specifications based on internal view of implementations unconnected to specification of externally visible behavior (messages sent on network)
- ...

Today: specification + verification of effectful code in VST

- OS primitives for sending and receiving network packets
-
- link model-level specification of interactions in Gallina / Quickchick to code-level specs in VST

Interaction trees



Interaction trees in Coq: Definition

Specifies possibly infinite executions

It's a monad...

Parametric on external events / visible actions

Return type

Leaf: terminating computation with return value

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

DeepWeb/Free/Monad/Free.v

Externally visible action of type [Event Y], plus a Y-dependent continuation k
(Y=unit: output)

Silent / internal action: step + continuation k

Interaction: Properties + Constructions

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

Monad structure: Ret +

Definition bindM {E X Y} (s: M E X) (t: X -> M E Y) : M E Y := …

(notation y ← s ;; t y).

Single-event tree: Definition liftE E X (e : E X) : M E X := Vis e (fun x => Ret x).

Discard result: Definition ignore {E X}: M E X -> M E unit := ...

Silent infinite loop: CoFixpoint spin {E X} : M E X := Tau spin.

Interacting inf. loop: CoFixpoint forever {E X} (x : M E X) : M E void := x;; forever x.

… and more – see Custom/Monad.v and DeepWeb/Free/Monad/Free.v

Interaction: Properties + Constructions

Questions:

1. **How to link interaction trees to C code in VST ?**

2. **How to specify OS primitives for sending and receiving messages ?**

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

Monad structure: Ret +

Definition bindM {E X Y} (s: M E X) (t: X -> M E Y) : M E Y := …

(notation y ← s ;; t y).

Single-event tree: Definition liftE E X (e : E X) : M E X := Vis e (fun x => Ret x).

Discard result: Definition ignore {E X}: M E X -> M E unit := ...

Silent infinite loop: CoFixpoint spin {E X} : M E X := Tau spin.

Interacting inf. loop: CoFixpoint forever {E X} (x : M E X) : M E void := x;; forever x.

… and more – see Custom/Monad.v and DeepWeb/Free/Monad/Free.v

Interaction: Properties + Constructions

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

Questions:

1. **How to link interaction trees to C code in VST ?**
2. **How to specify OS primitives for sending and receiving messages ?**

Monad structure: Ret +

```
Definition bindM {E X Y} (s: M E X) (t: X -> M E Y) : M E Y := ..
(notation y <- s ;; t y).
```

Today:

1. **Motivate ITREE separation logic predicates**
2. **Two examples: character-IO, swap server**

Single-event tree: Definition liftE E X (e : E X) : M E X := Vis e (fun x => Ret x).

Discard result: Definition ignore {E X}: M E X -> M E unit := ...

Silent infinite loop: CoFixpoint spin {E X} : M E X := Tau spin.

Interacting inf. loop: CoFixpoint forever {E X} (x : M E X) : M E void := x;; forever x.

… and more – see Custom/Monad.v and DeepWeb/Free/Monad/Free.v

Interaction trees in VST

(maintained in SEP clause)

- embed tree **t** in a novel **spatial** predicate **ITREE (t)**.

```
PRE ...
  PROP(…) LOCAL (…) SEP(…; ITREE (r ← x;; k r))
…
POST ...
  EX v,
  PROP(v =...) LOCAL (…) SEP(…; ITREE (k v))
```

- Forward direction (“consuming” style)
  - cf. actions in process calculi
  - enables referring to input in continuation via binding
  - compatibility with partial-correctness (eg infinite loops)

Interaction trees in VST: semantic foundation

VST's supports ghost state, allowing users to specify

ghost variables / locations that can hold arbitrary Coq values

VST/veric/juicy_extspec.v

Definition **has_ext** {Z:Type} (ora:Z):mpred := ….

Informal reading:

- ghost location 0 stores value ora:Z
- shared ownership: location can be
  - **read** by code **and** the external world (OS)
  - **modified** **only** by external world

Interaction trees in VST: semantic foundation

VST's supports ghost state, allowing users to specify

ghost variables / locations that can hold arbitrary Coq values

VST/veric/juicy_extspec.v

Definition **has_ext** {Z:Type} (ora:Z):mpred := ….

Informal reading:
- ghost location 0 stores value ora:Z
- shared ownership: location can be
  - **read** by code **and** the external world (OS)
  - **modified** **only** by external world

First idea: use has_ext directly:

Definition **ITREE** (t: M …):mpred := has_ext t.

- program and external world agree on current "protocol stage", and only OS can change this
- proof rules essentially embed external function call effects as Itree reductions the ghost state

Example: getchar/putchar

1. Definition of I-Trees for read / write interactions

```
(* The type of actions: reading yields an int, writing yields a unit *)

Inductive IO_event : Type -> Type :=

| ERead : IO_event int

| EWrite (c : int) : IO_event unit.
```

```
(* Define shorthands for each of the two operations*)

Definition read : M IO_event int := embed ERead.

Definition write (c : int) : M IO_event unit := embed (EWrite c).
```

```
(*Instantiate the I-Tree monad M – overall return type is unit*)

Definition IO_itree := M IO_event unit.
```

I-Trees in VST: getchar/putchar

2. VST axioms for OS / library functions

```
Definition putchar_spec :=
DECLARE _putchar

  WITH c : int, k : unit -> IO_itree

  PRE [ _c OF tint ]

    PROP ()
    LOCAL (temp _c (Vint c))
    SEP (ITREE (write c ;; k))

  POST [ tint ]

    PROP ()
    LOCAL (temp ret_temp (Vint c))
    SEP (ITREE k).
```

```
Definition getchar_spec :=
DECLARE _getchar

  WITH k : int -> IO_itree
  PRE [ ]

    PROP ()
    LOCAL ()
    SEP (ITREE (r <- read ;; k r))

  POST [ tint ]

   EX i : int,
    PROP (- 2^7 <= Int.signed i < 2^7)
    LOCAL (temp ret_temp (Vint i))
    SEP (ITREE (k i)).
```

Caution: these specs don't model error cases

I-Trees in VST: getchar/putchar

3. Write some C program: charIO/IO/io.c

```
void print_intr(uint I)
{
  unsigned int q,r;
  if (i!=0) {
    q=i/10u;
    r=i%10u;
    print_intr(q);
    putchar(r+'0');
  }
}
```

```
void print_int(uint i)
{
  if (i==0)
    putchar('0');
  else print_intr(i);
}
```

```
int main(void) {
  unsigned int n, d; char c;

  n=0;
  c=getchar();
  while (n<1000) {
    d = ((unsigned)c)-(unsigned)'0';
    if (d>=10) break;
    n+=d;
    print_int(n);
    putchar('\n');
    c=getchar();
  }
  Return 0;
}
```

I-Trees in VST: getchar/putchar

3. Write some C program: charIO/IO/io.c

```
void print_intr(uint I)
{
  unsigned int q,r;
  if (i!=0) {
    q=i/10u;
    r=i%10u;
    print_intr(q);
    putchar(r+'0');
  }
}
```

```
void print_int(uint i)
{
  if (i==0)
    putchar('0');
  else print_intr(i);
}
```

```
int main(void) {
  unsigned int n, d; char c;

  n=0;
  c=getchar();
  while (n<1000) {
    d = ((unsigned)c)-(unsigned)'0';
    if (d>=10) break;
    n+=d;
    print_int(n);
    putchar('\n');
    c=getchar();
  }
  Return 0;
}
```

4. Write some informal spec

- reads a sequence of characters, each in the range '0'..'9';

- after each one (and before the next one) prints a decimal representation of their cumulative sum, followed by a newline;

- when the cumulative sum >=1000, or when EOF is reached, stops.

I-Trees in VST: getchar/putchar

5. Translate the informal spec into Coq / ITrees

```
// Itree generating write ops for all elements of a list
Fixpoint write_list (l: list int) : IO_itree :=
match l with
   | nil => Ret tt
   | c :: rest => write c ;; write_list rest
end.

Program Fixpoint chars_of_nat n { measure n } : list int := …

Definition chars_of_Z z := chars_of_nat (Z.to_nat z).

// the function computed by print_intr
Program Fixpoint intr n { measure n } : list int :=
  match n with
  | O => []
  | _ => intr (n / 10) ++ [Int.repr (Z.of_nat (n mod 10) + char0)]
end.
```

I-Trees in VST: getchar/putchar

6. Embed Itrees in VST specs

```
void print_intr(uint I)
{
  unsigned int q,r;
  if (i!=0) {
    q=i/10u;
    r=i%10u;
    print_intr(q);
    putchar(r+'0');
  }
}
```

```
Fixpoint write_list (l: list int) :=
match l with
  | nil => Ret tt
  | c :: t => write c ;; write_list t
end.

Program Fixpoint intr n := ...
```

```
Definition print_intr_spec :=
 DECLARE _print_intr

  WITH i : Z, tr : IO_itree

  PRE [ _i OF tuint ]

    PROP (0 <= i <= Int.max_unsigned)
    LOCAL (temp _i (Vint (Int.repr i)))
    SEP (ITREE (write_list (intr (Z.to_nat i)) ;; tr))

  POST [ tvoid ]

    PROP ()
    LOCAL ()
    SEP (ITREE tr).
```

Similarly for print_int function ….

I-Trees in VST: getchar/putchar

6. Embed Itrees in VST specs

```
int main(void) {
  unsigned int n, d; char c;

  n=0;
  c=getchar();
  while (n<1000) {
    d = ((unsigned)c)-(unsigned)'0';
    if (d>=10) break;
    n+=d;
    print_int(n);
    putchar('\n');
    c=getchar();
  }
  Return 0;
}
```

```
CoFixpoint read_sum n d : IO_itree :=
  if zlt n 1000 then
     if zlt d 10 then
         write_list (chars_of_Z (n + d));;
         write (Int.repr newline);;
         c <- read;;
         read_sum (n + d) (Int.unsigned c - char0)
  else ret tt else ret tt.
```

```
Definition main_itree := c <- read;; read_sum 0
(Int.unsigned c - char0).
```

I-Trees in VST: getchar/putchar

6. Embed Itrees in VST specs

```
int main(void) {
  unsigned int n, d; char c;

  n=0;
  c=getchar();
  while (n<1000) {
    d = ((unsigned)c)-(unsigned)'0';
    if (d>=10) break;
    n+=d;
    print_int(n);
    putchar('\n');
    c=getchar();
  }
  Return 0;
}
```

CoFixpoint??? Program terminates after 1000 iterations!

```
CoFixpoint read_sum n d : IO_itree :=
  if zlt n 1000 then
    if zlt d 10 then
        write_list (chars_of_Z (n + d));;
        write (Int.repr newline);;
        c <- read;;
        read_sum (n + d) (Int.unsigned c - char0)
  else ret tt else ret tt.
```

But really finite !

```
Definition main_itree := c <- read;; read_sum 0
(Int.unsigned c - char0).
```

I-Trees in VST: getchar/putchar

6. Embed Itrees in VST specs

CoFixpoint??? Program terminates after 1000 iterations!

```
int main(void) {
  unsigned int n, d; char c;

  n=0;
  c=getchar();
  while (n<1000) {
    d = ((unsigned)c)-(unsigned)'0';
    if (d>=10) break;
    n+=d;
    print_int(n);
    putchar('\n');
    c=getchar();
  }
  Return 0;
}
```

```
CoFixpoint read_sum n d : IO_itree :=
  if zlt n 1000 then
    if zlt d 10 then
        write_list (chars_of_Z (n + d));;
        write (Int.repr newline);;
        c <- read;;
        read_sum (n + d) (Int.unsigned c - char0)
  else ret tt else ret tt.
```

But really finite !

```
Definition main_itree := c <- read;; read_sum 0
(Int.unsigned c - char0).
```

```
Definition main_spec :=
  DECLARE _main  WITH gv : globals
  PRE  [] main_pre_ext prog main_itree nil gv
  POST [ tint ] main_post prog nil gv.
```

main_post == TT, so no ITREE here

I-Trees in VST: getchar/putchar

7. Add VST boilerplate and verify individual functions

```
Definition of Vprog, Gprog Espec...

Lemma body_print_int: semax_body Vprog Gprog f_print_int print_int_spec.

Lemma body_print_intr: semax_body Vprog Gprog f_print_intr print_intr_spec.

Lemma body_main: semax_body Vprog Gprog f_main main_spec.
```

I-Trees in VST: getchar/putchar

7. Add VST boilerplate and verify individual functions

```
Definition Vprog, Gprog Espec...

Lemma body_print_int: semax_body Vprog Gprog f_print_int print_int_spec.

Lemma body_print_intr: semax_body Vprog Gprog f_print_intr print_intr_spec.

Lemma body_main: semax_body Vprog Gprog f_main main_spec.
```

Not so fast ...

Better definition of **ITREE**(t)

VST/veric/juicy_extspec.v

```
Definition has_ext {Z:Type} (ora:Z) := ….
```

iospecs.v

```
Definition ITREE (t) := EX t' : _, !!(EquivUpToTau t t') && has_ext t'.
```

Remove tau actions (yields equiv relation)

Sanity checks:

```
Lemma has_ext_ITREE t: has_ext t |-- ITREE t.
Lemma ITREE_impl t t’:  EquivUpToTau t t' -> ITREE t |-- ITREE t'.
```

In particular:

- monad laws (associativity / congruence of bind) can be lifted to ITREE
- application-level equivalences can be exploited:

```
Lemma write_list_app l1 l2:
EquivUpToTau (write_list (l1 ++ l2)) (write_list l1;; write_list l2).
```

Better definition of **ITREE**(t)

VST/veric/juicy_extspec.v

```
Definition has_ext {Z:Type} (ora:Z) := ….
```

iospecs.v

```
Definition ITREE (t) := EX t' : _, !!(EquivUpToTau t t') && has_ext t'.
```

(Remove tau actions (yields equiv relation)

Enjoy!

Sanity checks:

```
Lemma has_ext_ITREE t: has_ext t |-- ITREE t.
Lemma ITREE_impl t t':  EquivUpToTau t t' -> ITREE t |-- ITREE t'.
```

In particular:

- monad laws (associativity / congruence of bind) can be lifted to ITREE
- application-level equivalences can be exploited:

```
Lemma write_list_app l1 l2:
EquivUpToTau (write_list (l1 ++ l2)) (write_list l1;; write_list l2).
```

I-Trees in VST: getchar/putchar

8. Putting it all together ….

```
Definition of Vprog, Gprog Espec...

Lemma body_print_int: semax_body Vprog Gprog f_print_int print_int_spec.

Lemma body_print_intr: semax_body Vprog Gprog f_print_intr print_intr_spec.

Lemma body_main: semax_body Vprog Gprog f_main main_spec.
```

```
Instance Espec : OracleKind :=
            add_funspecs IO_Espec (ext_link_prog prog) Gprog.

Theorem prog_correct: semax_prog_ext prog main_itree Vprog Gprog.
```

"**main_itree** is safe for the whole **prog**"

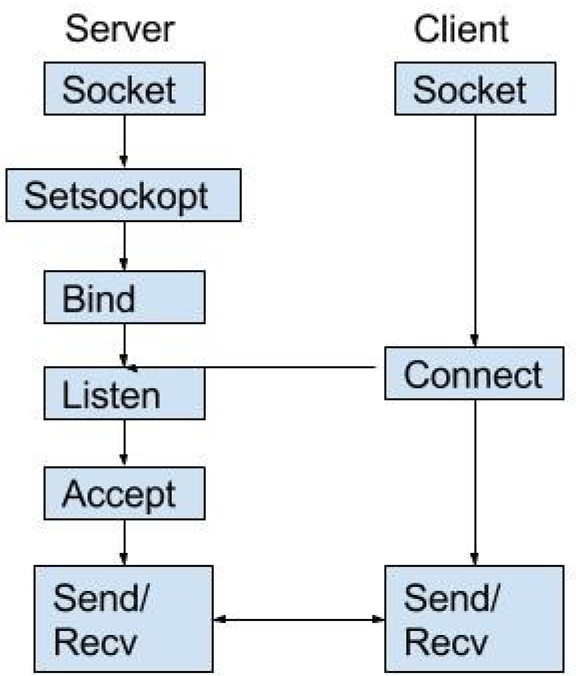
Connecting VST-Itrees to CertiKOS

Goal: show that CertiKOS indeed operates as specified, i.e. is a suitable ”external word” for those ghost-state manipulating instructions that amount to OS calls.

Challenges:

- VST’s ghost-supporting mathematical model not directly translatable into CertiKOS - concepts (“step-indexing”, “predicates-in-the heap”)
- But: erasing / compiling away step-indexing yields first-order specifications that amount to pre-post specifications over **CompCert** memories, these **are** understood by CertiKOS
- Current work: prove formal correspondence between step-erased specs and their ghost-enriched counterparts
- Approach restricted to OS calls without complex recursive structure (callbacks via function pointers), but sufficient for charaction-IO and network interactions: data sent/received is typically first-order

I-Trees in VST: sockets



https://www.geeksforgeeks.org/socket-programming-cc/

Function prototypes (server)

IPv4 / IPv6... UDP / TCP Tpyically 0

```
// returns sockfd
int socket(domain, type, protocol);

//associates socket to address/port
int bind(int sockfd,
          const struct sockaddr *addr, socklen_t addrlen);
```

Max length of queue of pending conn's

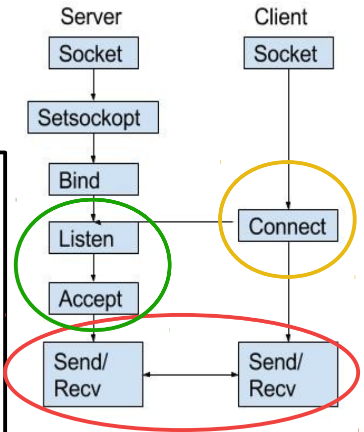
```
// put socket into "awaiting requests" mode
int listen(int sockfd, int backlog);

// establish conn with first request; return if of fresh conn
int accept(int sockfd,
              struct sockaddr *addr, socklen_t *addrlen);
```

Additional function prototype (client):

```
// try to establish new connection between sockfd and server at addr; retval indicates success
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

I-Trees in VST: sockets

1. Definition of I-Trees for socket interactions



https://www.geeksforgeeks.org/socket-programming-cc/

Network-visible events:

```
Definition connection_id : Type := nat.

Definition endpoint_id : Type := nat.

Inductive networkE : Type -> Type :=

| Listen : endpoint_id -> networkE unit

| Accept : endpoint_id -> networkE connection_id

| ConnectTo: endpoint_id -> networkE connection_id

| Shutdown : connection_id -> networkE unit

| RecvByte : connection_id -> networkE (option string)

| SendByte : connection_id -> string -> networkE unit.
```

Lib/NetworkInterface.v

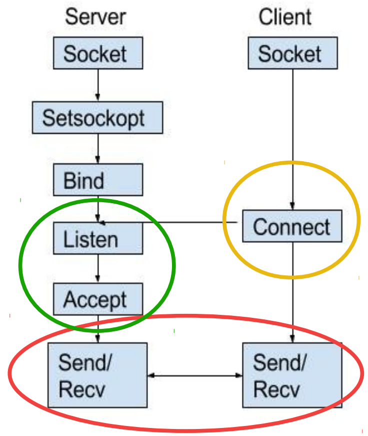
I-Trees in VST: sockets

1. Definition of I-Trees for socket interactions

Derived Forms (excerpt):

```
// Receive a string of length at most [len]. The return value [None] signals that
// a connection was closed, when modelling the [recv()] POSIX syscall.

Definition recv `{networkE -< E} `{nondetE -< E}
(c : connection_id) (len : nat) : M E (option bytes) := ….

//Send all bytes in a bytestring.

Fixpoint send `{networkE -< E} (c : connection_id)
     (bs : bytes) : M E unit := for_bytes bs (send_byte c).
```

Lib/NetworkInterface.v:



https://www.geeksforgeeks.org/socket-programming-cc/

Extend network events by failure + nondeterminism:

```
Definition SocketE : Type -> Type := (nondetE +' failureE +' networkE).
```

I-Trees in VST: sockets

1. Definition of I-Trees for socket interactions

Protocol stages of socket: these actions are **not** network-visible (OS-interaction only)

Lib/Socketinterface.v:

```
Inductive socket_status :=
 | ClosedSocket
 | OpenedSocket
 | BoundSocket (addr : endpoint_id)
 | ListeningSocket (addr : endpoint_id)
 | ConnectedSocket (conn : connection_id).

 Record SocketMap : Type := { lookup_socket: sockfd -> socket_status }.
```

again, client-side...

{ z:Z | 0 <= z <= maxFD}

We capture that status of all sockets in a seond mpred:

```
Parameter SOCKAPI: SocketMap -> mpred.
```

Will soon be replaced by another ghost variable...

I-Trees in VST: sockets

2a. ITREE predicate

Again, don’t use has_ext directly but

```
Definition SocketMonad := M SocketE.

Definition ITREE {T} (t : SocketMonad T) :=
EX t' : SocketMonad T, !!(trace_incl t t') && has_ext t'.
```

Spec/Vst/SocketSpecs.v

where

```
Definition trace_incl {R : Type} (m m' : M_ R) : Prop :=
   forall t r, is_trace_of t r m -> is_trace_of t r m'.
```

Lib/Socketinterface.v

```
Inductive is_trace_of {R : Type}: trace -> option R -> M_ R -> Prop := …
```

Lib/Socketinterface.v

generalizes equivUpToTau from previous example, yielding, e.g.

Spec/Vst/SocketSpecs.v

```
Lemma trace_bind_assoc: forall ...,

    ITREE ( b <- (a <- m ;; f a) ;; g b ) = ITREE ( a <- m ;; b <- f a ;; g b ).
```

I-Trees in VST: sockets

2b. VST axioms for send / receive

Spec/Vst/SocketSpecs.v

```
Definition recv_spec  (T : Type) :=
  DECLARE _recv

  WITH t : SocketMonad T, k : option string -> SocketMonad T, st : SocketMap,
        client_conn : connection_id, fd: sockfd, buf_ptr: val, alloc_len: Z, sh: share

  PRE [ 1%positive OF tint, 2%positive OF (tptr tvoid), 3%positive OF tuint, 4%positive OF tint ]

    PROP ( consistent_world st; lookup_socket st fd = ConnectedSocket client_conn; writable_share sh;
           trace_incl (r <- recv client_conn (Z.to_nat alloc_len) ;; k r) t)

    LOCAL (temp 1%positive (Vint (Int.repr (descriptor fd))); temp 2%positive buf_ptr;
             temp 3%positive (Vint (Int.repr alloc_len)); temp 4%positive (Vint (Int.repr 0)))

    SEP ( SOCKAPI st; ITREE t;
           data_at_ sh (Tarray tuchar alloc_len noattr) buf_ptr )

  POST [ tint ] EX result : _, st' : SocketMap, r : _, contents: _,

    PROP ( 0 <= r <= alloc_len \/ r = NO;
        r > 0 → st'=st /\ exists msg, result = inr (Some msg) /\ Zlength (val_of_string msg) = r /\
                          sublist 0 r contents = (val_of_string msg) /\
                          sublist r alloc_len contents = list_repeat (Z.to_nat (alloc_len - r)) Vundef;
      (*case EOF*)  r = 0 -> result = inr None /\ contents = list_repeat (Z.to_nat alloc_len) Vundef /\
             st' = update_socket_state st fd OpenedSocket) ;
        r < 0 -> st'=st /\ result = inl tt /\ contents = list_repeat (Z.to_nat alloc_len) Vundef;
        Zlength contents = alloc_len; consistent_world st' )

    LOCAL ( temp ret_temp (Vint (Int.repr r)) )

    SEP ( data_at sh (tarray tuchar alloc_len) contents buf_ptr; SOCKAPI st' ;
        ITREE (match result with
                    inl tt => t
                  | inr msg_opt => k msg_opt
                  end)).
```

I-Trees in VST: sockets

2b. VST axioms for send / receive

```
Definition recv_spec  (T : Type) :=                                Spec/Vst/SocketSpecs.v
  DECLARE _recv
  WITH t : SocketMonad T, k : option string -> SocketMonad T, st : SocketMap,
       client_conn : connection_id, fd: sockfd, buf_ptr: val, alloc_len: Z, sh: share
  PRE [ 1%positive OF tint, 2%positive OF (tptr tvoid), 3%positive OF tuint, 4%positive OF tint ]
   PROP ( consistent_world st; lookup_socket st fd = ConnectedSocket client_conn; writable_share sh;
          trace_incl (r <- recv client_conn (Z.to_nat alloc_len) ;; k r) t)
```

Yes, the spec for send is similar (and similarly lengthy)...

Specs for bind and socket don't involve ITREE terms, just SOCKAPI terms.

Again, these axioms need to be justified by CertiKOS.

```
         r < 0 -> st'=st /\ result = inl tt /\ contents = list_repeat (Z.to_nat alloc_len) Vundef;
         Zlength contents = alloc_len; consistent_world st' )
   LOCAL ( temp ret_temp (Vint (Int.repr r)) )
   SEP ( data_at sh (tarray tuchar alloc_len) contents buf_ptr; SOCKAPI st' ;
         ITREE (match result with
                  inl tt => t
                | inr msg_opt => k msg_opt
                end)).
```

I-Trees in VST: sockets

3. Write some C program: **dw/DeepWeb/Spec/main.cw/DeepWeb/Spec/main.c**

I-Trees in VST: sockets

3. Write some C program: **dw/DeepWeb/Spec/main.cw/DeepWeb/Spec/main.c**

... 631 LOC ......

I-Trees in VST: sockets

3. Write some C program: **dw/DeepWeb/Spec/main.cw/DeepWeb/Spec/main.c**

... 631 LOC ......

4. Write an informal spec: **dw/DeepWeb/Spec/main.cw/DeepWeb/Spec/main.c – lines 2- 10!**

I-Trees in VST: sockets

3. Write some C program: dw/DeepWeb/Spec/main.cw/DeepWeb/Spec/main.c

... 631 LOC ......

4. Write an informal spec: dw/DeepWeb/Spec/main.cw/DeepWeb/Spec/main.c – lines 2- 10!

This program implements a swap server that accepts connections from multiple clients.

On each connection, the server receives bytes from the corresponding client until the buffer is full, at which point the received bytes are deemed a "message".

The server replies to this client with the last obtained message, which could have been obtained from another client (on another connection). This last obtained message is initially set to all zeroes.

All these actions (accepting a new connection, receiving a message, and sending a message) are interleaved.

I-Trees in VST: sockets

3. Write some C program: dw/DeepWeb/Spec/main.cw/DeepWeb/Spec/main.c

... 631 LOC ......

4. Write an informal spec: dw/DeepWeb/Spec/main.cw/DeepWeb/Spec/main.c – lines 2- 10!

This program implements a swap server that accepts connections from multiple clients.

On each connection, the server receives bytes from the corresponding client until the buffer is full, at which point the received bytes are deemed a "message".

The server replies to this client with the last obtained message, which could have been obtained from another client (on another connection). This last obtained message is initially set to all zeroes.

All these actions (accepting a new connection, receiving a message, and sending a message) are interleaved.

I-Trees in VST: sockets

5. Translate the informal spec into Coq / Itrees

"ClikeSpec" : counterparts of data structures

```
Inductive connection_state : Type :=
RECVING | SENDING | DONE |
DELETED.

Record connection : Type :=
{ conn_id : connection_id;
  conn_request : string;
  conn_response : string;
  conn_response_bytes_sent : Z;
  conn_state : connection_state
}.
```

```
typedef enum state {
  RECVING,
  SENDING,
  DONE,
  DELETED,
} state;
```

```
typedef struct connection {
  socket_fd fd;
  uint32_t request_len;
  uint8_t request_buffer[BUFFER_SIZE];
  uint32_t response_len;
  uint8_t response_buffer[BUFFER_SIZE];
  uint32_t num_bytes_sent;
  enum state st;
  struct connection* next;
} connection;
```

I-Trees in VST: sockets

5. Translate the informal spec into Coq / Itrees

"ClikeSpec" : counterparts of data structures

```
typedef enum state {
  RECVING,
  SENDING,
  DONE,
  DELETED,
} state;
```

```
Inductive connection_state : Type :=
RECVING | SENDING | DONE |
DELETED.

Record connection : Type :=
{ conn_id : connection_id;
  conn_request : string;
  conn_response : string;
  conn_response_bytes_sent : Z;
  conn_state : connection_state
}.
```

```
typedef struct connection {
  socket_fd fd;
  uint32_t request_len;
  uint8_t request_buffer[BUFFER_SIZE];
  uint32_t response_len;
  uint8_t response_buffer[BUFFER_SIZE];
  uint32_t num_bytes_sent;
  enum state st;
  struct connection* next;
} connection;
```

and their operations

```
Definition upd_conn_request (conn : connection) (request : string) : connection := ...

Definition upd_conn_response (conn : connection) (response : string) : connection := ...
```

etc.

dw/DeepWeb/Spec/Swap_CLikeSpec.v

I-Trees in VST: sockets

5. Translate the informal spec into Coq / Itrees

counterparts of C functions

```
(* Wait for a message from connection [conn].   [recv] can return a partial message, in which
case we store the bytes we received to try receiving more in a later iteration. *)

Definition conn_read (buffer_size : Z) (conn: connection) (last_full_msg : string)

          : M socketE (connection * string) := . . .
```

```
static int
conn_read(connection* conn, store* last_msg_store) {
    . . .
}
```

I-Trees in VST: sockets

5. Translate the informal spec into Coq / Itrees

counterparts of C functions

```
(* Wait for a message from connection [conn].   [recv] can return a partial message, in which
case we store the bytes we received to try receiving more in a later iteration. *)

Definition conn_read (buffer_size : Z) (conn: connection) (last_full_msg : string)

          : M socketE (connection * string) := . . .
```

28 lines ;-)

```
static int
conn_read(connection* conn, store* last_msg_store) {
    . . .
}
```

20 lines

. . . similarly for the other data structures and functions of the C code . . .

I-Trees in VST: sockets

6. Embed Itrees in VST specs

```
Definition conn_read_spec (T : Type) (buffer_size : Z) :=
  DECLARE _conn_read
  WITH k : (connection * string) -> SocketM T, st : SocketMap, conn : connection,
            fd : sockfd, last_msg : string, conn_ptr : val,  msg_store_ptr : val
  PRE [ _conn OF (tptr (Tstruct _connection noattr)), _last_msg_store OF (tptr (Tstruct _store noattr) ]
  PROP ( consistent_world st; conn_state conn = RECVING ;
           consistent_state buffer_size st (conn, fd)  )
   LOCAL ( temp _conn conn_ptr ; temp _last_msg_store msg_store_ptr )
   SEP ( SOCKAPI st ;  ITREE (r <- conn_read buffer_size conn last_msg ;; k r) ;
           list_cell LS Tsh (rep_connection conn fd) conn_ptr ;
           field_at Tsh (Tstruct _store noattr) []  (rep_store last_msg) msg_store_ptr )
  POST [ tint ]   EX conn' : _, EX last_msg' : _ , EX st' : _, EX r : Z,
   PROP ( r = YES ;  consistent_world st' ;
            recv_step buffer_size (conn, fd, st, last_msg) (conn', fd, st', last_msg') )
   LOCAL ( temp ret_temp (Vint (Int.repr r)) )
   SEP ( SOCKAPI st' ; ITREE (k (conn', last_msg')) ;
            list_cell LS Tsh (rep_connection conn' fd) conn_ptr ;
            field_at Tsh (Tstruct _store noattr) []  (rep_store last_msg') msg_store_ptr ).
```

etc.

I-Trees in VST: sockets

7. Add VST boilerplate and verify individual functions

. . . .

```
Lemma body_conn_read:
 semax_body Vprog Gprog f_conn_read (conn_read_spec unit BUFFER_SIZE).
Proof. start_function. … Qed.
```

430 lines

etc.

I-Trees in VST: sockets

8. Putting it all together

dw/DeepWeb/Proofs/Vst/verif_main.v

```
Lemma body_main:  semax_body Vprog Gprog f_main (main_spec server).
Proof. … Qed.
```

dw/DeepWeb/Proofs/Vst/verif_prog.v

```
Lemma all_funcs_correct:  semax_func Vprog Gprog (prog_funct prog) Gprog.
Proof. … Qed.
```

Swap server: top level claim / proof

"**tree** is safe for the whole **prog**"

Spec/TopLevelSpec.v

Definition swap_server_correct := exists (tree : M socketE unit),

semax_prog_ext **prog** tree Vprog Gprog /\

refines_mod_network swap_observer (simplify_network tree)

Currently omits listen, shutdown

"Refinement scrambling": any scrambling that results from a trace of the low-level implementation can also arise from a trace of the high-level spec.

Swap server: top level claim / proof

"**tree** is safe for the whole **prog**"

Spec/TopLevelSpec.v

Definition swap_server_correct := exists (tree : M socketE unit),

semax_prog_ext **prog** tree Vprog Gprog /\

refines_mod_network swap_observer (simplify_network tree)

Currently omits listen, shutdown

"Refinement scrambling": any scrambling that results from a trace of the low-level implementation can also arise from a trace of the high-level spec.

Proofs/TopLevelProof.v

Theorem swap_server_is_correct : swap_server_correct.
Proof. exists **main_itree**; split. … Admitted.

(*ToDo: scaffolding macros, parts of refinement *)

Swap server: top level claim / proof

"tree is safe for
the whole prog"

Spec/TopLevelSpec.v

```
Definition swap_server_correct :=  exists (tree : M socketE unit),
    semax_prog_ext prog tree Vprog Gprog /\
    refines_mod_network swap_observer (simplify_network tree)
```

Currently omits listen, shutdown

Notation:

" prog |== swap_observer "

"Refinement sc
results from a trac
can also arise from a trace of the high-level spec.

Proofs/TopLevelProof.v

```
Theorem swap_server_is_correct : swap_server_correct.
Proof. exists main_itree; split. … Admitted.

(*all_funcs_correct verified but some scaffolding / macros ToDo *)
```

Scaling out (in all dimensions)

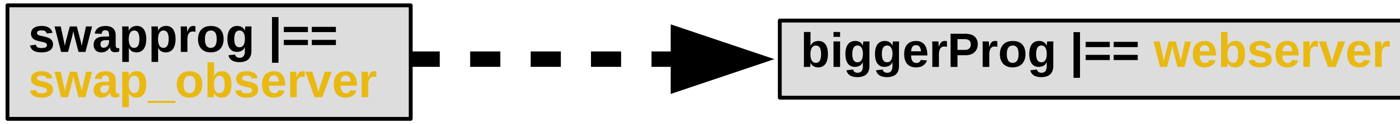
Horizontally:

swapprog |== swap_observer

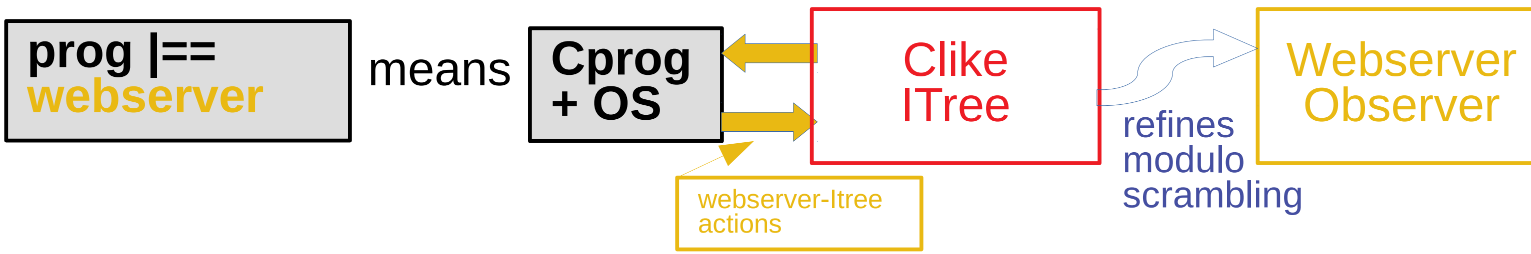
biggerProg |== webserver

Scaling out (in all dimensions)

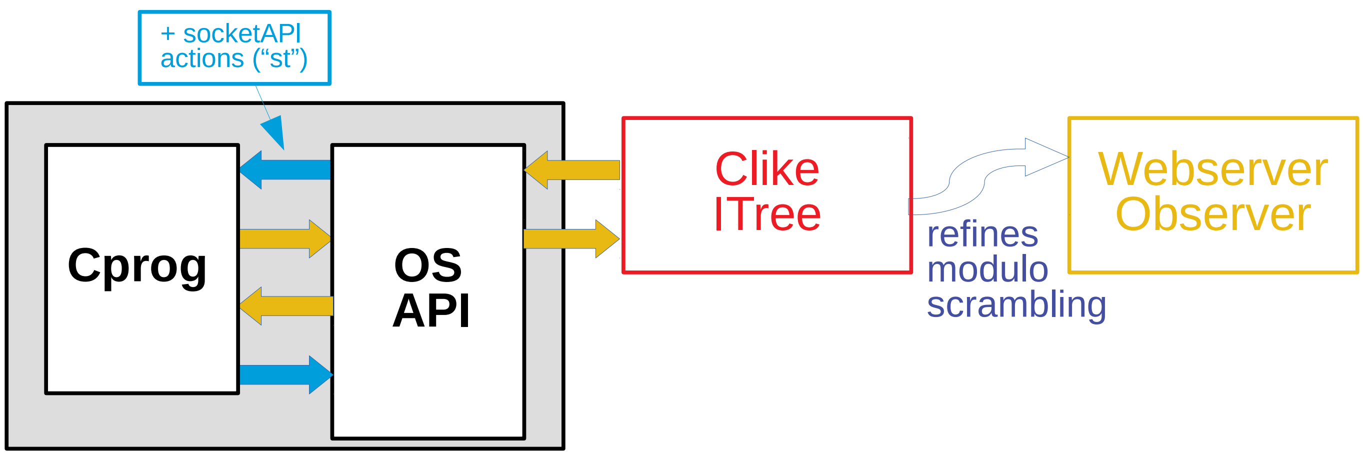
Horizontally:



Vertically:



or better



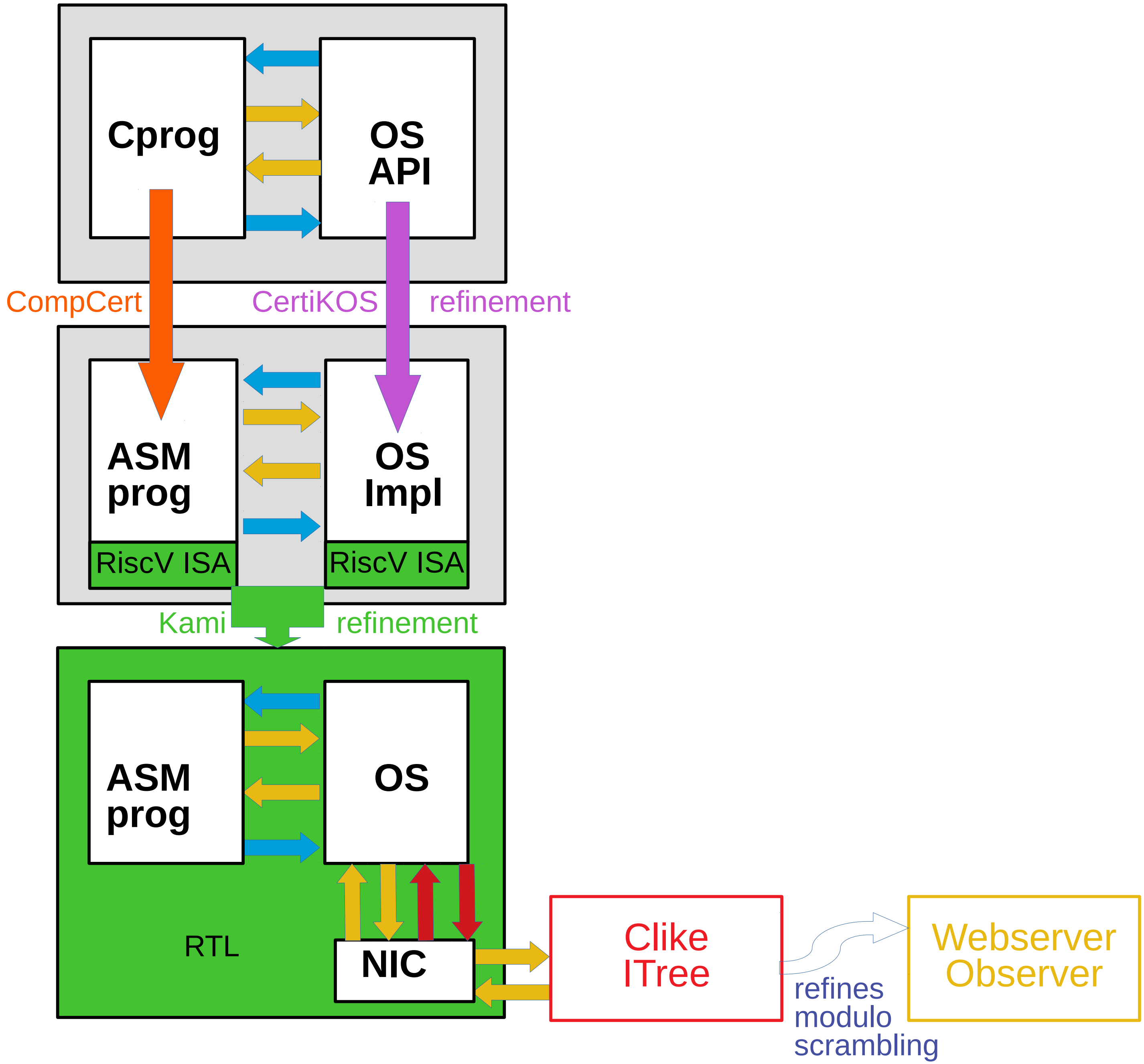
Scaling out (in all dimensions)

Horizontally: **swapprog |== swap_observer** → **biggerProg |== webserver**

Vertically: **prog |== webserver** means **Cprog + OS** ⇄ Clike ITree (refines modulo scrambling) → Webserver Observer

webserver-Itree actions

or better


Cprog
OS API
CompCert
CertiKOS
refinement
ASM prog
OS Impl
RiscV ISA
RiscV ISA
Kami
refinement
ASM prog
OS
RTL
NIC
Clike ITree
refines modulo scrambling
Webserver Observer

Challenge I: composition of various refinement notions

Challenge I: composition of various refinement notions

Challenge II:

Theorem Correct:

Cprog |== server.

Proof. ... Qed.

Integration testbed for DeepSpec Technologies