Integrating Types and Specifications for Secure Software Development

Greg Morrisett
Harvard University

Software Security

- Very few security breaches are due to weak crypto or broken protocols.
- Most are due to:
  - implementation errors (programmers)
  - configuration errors (administrators)
  - wrong decisions (users)
- Software security focuses on tools to help programmers avoid these errors.

Yesterday’s Attacks:

- Code injection via stack smashing.
- Code injection via heap spraying.
- Code synthesis via buffer overrun and “return-oriented programming”.
- Changing/revealing sensitive variables via format string attacks.
- Changing/revealing sensitive database entries via SQL injection attacks.
- Denial of service attacks via null-pointer dereference.

Today's Mitigations

- Manual code inspection (e.g., grep for strcpy)
- Testing
- No-execute for the stack
  - requires new chips, doesn't stop heap spraying or return-oriented programming.
- Stack guard, /gs switch
  - stops only some stack smashing, does not stop heap spraying.
- Heap guard, allocation randomization
  - alas, MS lookaside lists made this irrelevant, Adobe used own heap manager.
- Static analysis: Prefast, Coverity, Fortify
  - unsound else too many false positives
- Address space randomization
  - low entropy in Windows, just slows the spread

The Trend

- Most of the early attacks were based on lack of enforcement of *language-level abstractions.*
  - No one expects “a[34] := x” to change the behavior of a procedure return.
  - Frustrating because type safety is all about enforcing abstractions.
- The mitigations aren’t perfect.
  - But they have increased the difficulty of constructing an exploit that takes advantage of language-level errors.
    - Evidence: cost of a zero-day in Win7 is orders of magnitude higher than for WinXP.
- Unfortunately, they’ve simply shifted the attacks:
  - to code that Microsoft doesn’t own (drivers, Flash plug-in), or
  - a higher-level of abstraction (e.g., SQL, Javascript, APIs).
  - (Next: denial of service…)

Research

- Goal should be to get ahead of these trends.
- Type-safe languages (e.g., Java)
  - enforce language-level abstractions.
  - means we can reason at the source level instead of at the machine level.
  - key challenge: bugs in the implementation
    - for instance, JDK includes 700K lines of C code.
    - can we eliminate the compiler from the trusted computing base?
- Next-generation types
  - encode application-level security policies
  - key challenge: tradeoff between expressiveness of types, and automation.
  - key challenge: scaling these expressive type systems to full languages.

Complimentary Efforts

- Powerful static type checking for systems code
  - ruling out application-level errors
- Compiler & type-checker verification
  - ruling out errors in the implementation

What we wish we could do with types:

A range from shallow to deep properties:

```
sub : (x:array T, i:int) → T
          requires i >= 0 && i < size(x)

printf : (x:string) -> (vs:list obj) -> unit
           requires (∃ts,parses(x,ts) &&
                             have_types(vs,ts))

typechecks : (x:source) -> (y:bool)
              ensures if y then typesafe(x)
                        else True

compile : (x:source) → (y:x86)
              ensures (bisimilar(x,y))
```

Emerging languages:

- ESC/Java, JML, Spec#, Sage, M, Cyclone, etc.
  - Extend types with specifications & refinements
    - `void foo(Bar x) requires x != null`
    - Pre-, post-conditions, assertions, object invariants, etc.
    - But limited to first-order logic over a few sorts.
  - Generate verification conditions (VCs) as part of type-checking.
    - Use abstract interpretation for loop invariants.
  - Feed verification conditions to an SMT prover.

Unfortunately:

- Two very different languages.
  - An impoverished “pure math” language for models & specs.
  - An imperative language for code.
  - Serious confusion trying to mix the two.
    - e.g., what happens when a spec calls a function with effects?
- First-order logic hurts.
  - No real ability to abstract over models & specs.
  - No good frame properties.
- Provers & analyses come up short.
  - Can discharge shallow properties: x != null.
  - But not parses(x,ts), much less bisimilar(x,y).

I have some experience…

- Cyclone incorporated a form of refinement types.
  - Cyclone was a type-safe dialect of C.
  - Goal was to eliminate null-pointer checks, array-bounds checks.
  - It was actually quite effective:
    - even with a dumb theorem prover, got rid of 90% of the checks.
  - But 10% is still a lot (4,000 checks left in the compiler).
    - primary limitation was *not* the theorem prover
    - it was partially due to the synthesis of loop invariants
    - it was partially due to the lack of context/summaries (for scaling)
    - and the inability to reason about memory (aliasing) in a modular fashion.
  - And I wanted to prove *application-level* properties about my code, not just language-level.

Some semantic issues...

- There were also some tricky semantic issues with refinement types in impure programming languages.
  - { x : ref int | x := 42; true }
  - {(x,y): int*int | x/y > 10}
  - { x : int | exit(0) }
- Usual approach:
  - analyze the syntax of the predicate to rule out side effects, including exceptions, divergence, IO, nested failing contracts, etc.
  - and furthermore, you can't use *separately compiled* functions in your predicates.
  - need to reflect *effects into types* to get a modular treatment.

Type Theory

Give programmers the ability to work around short-comings of automation.

In particular, give them a way to build explicit *proofs* within the language.

- if automation can't find proof, at least the programmer can try to construct one.

Not a new idea: dependent type theory!

- in particular: Coq, Agda, ACL2, Isabelle/HOL, PRL,…
- but *many* challenges in making this practical

Programming in Coq:

```
sub(v:vector T)(i:nat)(pf:i<size(v)): T;


cmp(i:nat)(j:nat): LT{i<j} + GTE{i>=j}


checked_sub(v:vector T)(i:nat):option T
  := match cmp i (size v) with
       | LT pf => SOME(sub v i pf)
       | _ => NONE
       end
```

Another Example

```
Inductive list(T:Type) : Type :=
      | nil : list T
      | cons  : T -> list T -> list T.

Infix “::” := cons.



Fixpoint append(x y:list nat) : list nat :=
      match x with
      | nil => y
      | h::t => h::(append t y)
      end.



Lemma append_assoc :
      ∀x y z, append (append x y) z = append x (append y z).
```

Building Proofs Explicitly

```
eq_refl : ∀ (T : Type) (x : T), x = x

eq_ind_r : ∀ (T : Type) (x : T) (P : T -> Prop),
      P x -> ∀y : T, y = x -> P y

Fixpoint append_assoc(x:list nat) :
    ∀y z, append (append x y) z = append x (append y z) :=
    match x with
    | nil => fun y z => eq_refl (append (append nil y) z
    | h::t => fun y z =>
       eq_ind_r (fun l : list nat =>
                h :: l = h :: append t (append y z))
             (eq_refl (h :: append t (append y y z)))
             (append_assoc t y z)
    end.
```

Building Proofs in Practice

```
Lemma append_assoc :
     ∀x  y z, append (append x y) z = append x (append y z).

Proof.

  induction x.

    auto.

    intros y z H. simpl. rewrite H. auto.

Qed.
```

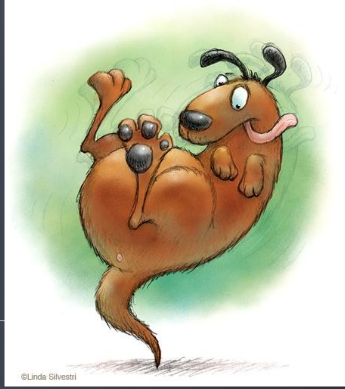
How Does All This Scale?

X.Leroy [PoPL '06]: correct, optimizing compiler from C to PowerPC:

- Build interpreters for C and PowerPC code.
- compile: (s:source) $\rightarrow$ (t:target, bisimilar(s,t))
- compiler comparable to good ugrad class
- Coq extracts Ocaml code by erasing proofs

Bottom line: it's feasible to build *mechanically* verified software using this kind of approach.

Great Progress, but…



- 4,000 line compiler:
  - 7,000 lines of
    lemmas and theorems
    - includes interpreters/models
    - much is re-usable in other contexts
  - 17,000 lines of
    proof scripts!
    - though with right tactics, could at least cut in half.
    - and keep in mind, this is a *very* deep property.
- Key research question:
  - how to keep the tail from wagging the dog?

Recent Work [PoPL'10]

- Developed a core-ML compiler
  - higher-order functions, datatypes, refs, etc.
  - CPS & closure conversion
  - Common sub-expr., dead-code, register allocation, etc.
- Compiler about 5,000 loc.
- Proofs only 2,000 loc!
  - Stronger result too: compiler will terminate and produce bisimilar result.

Adam’s Secret Weapon

The typical Coq proof is coded as a series of small steps that drive a goal down to known facts.

- This makes the proof very brittle: small changes in the code or specifications result in non-local changes in corresponding proofs.
- Adam codes a search tactic that automatically simplifies goals.
- As a result, changes to the code or specifications really only demand augmentations to the shared tactic.

Example

```
Inductive stmt : Set :=

| ...

| Seq : stmt -> stmt -> stmt

| ...



Inductive evals :
     stmt -> state -> state -> Prop :=

| ...

| evSeq : forall c1 c2 s1 s2 s3,

      evals c1 s1 s2 -> evals c2 s2 s3 ->

        evals (Seq c1 c2) s1 s3
```

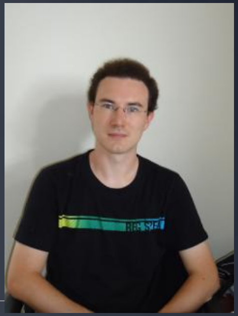
Proving Seq Associative

```
Lemma seq_assoc :
  ∀c1 c2 c3 s1 s2,
  evals (Seq c1 (Seq c2 c3)) s1 s2 ->
  evals (Seq (Seq c1 c2) c3) s1 s2.

Proof.
  intros.
  inversion H ; subst.
  inversion H5 ; subst.
  econstructor.
  econstructor.
  eapply H2.
  eapply H3.
  eapply H7.

Qed.
```

Adam style

```
Ltac mytac := repeat match goal with
    | [ |- forall _, _ ] => intro
    | [ H : evals (Seq _ _) _ _ |- _ ] =>
          inversion H ; subst ; clear H
    | [ |- evals (Seq _ _) _ _ ] => econstructor
    | _ => eauto

end.



Lemma seq_assoc : forall c1 c2 c3  s1 s2,
   evals (Seq c1 (Seq c2 c3)) s1 s2 ->
   evals (Seq (Seq c1 c2) c3) s1 s2.

Proof.

  mytac.

Qed.
```

More Recently



- Paul & Jean have developed a translation validator for the LLVM compiler.
  - tries to automatically prove the output of an optimization is equivalent to the input.
  - in essence, decompiles the code into a denotational semantics.
  - MySQL implementation: validates 95% of the functions with 11 optimizations turned on.
- Suggests that we can significantly lower the proof burden for realistic compilers.

So…

- At least for compilers, it's not only possible, but I would claim *practical* to build fully verifying compilers.
  - Just as importantly: maintain them!
- Reality: you still need very smart people to do the specifications and proofs.
  - But they can build compilers for domain-specific languages that capture important safety or security properties.
- Bottom line: we can eliminate compilers from the trusted computing base.
  - *language-enforced* security properties.

Another big problem:

Systems like Coq (and ACL2, Isabelle/HOL, etc.) are limited to *pure, total* functions:

- no hash tables, union-find, splay trees, …
- no I/O, no exceptions, no diverging computations, no concurrency, …

As a result, both Xavier's and Adam's compilers are relatively slow.

And, although we can *model* systems (e.g., kernels), we can't program them directly in Coq.

Why must we have pure functions?

Why only total functions?

At all costs, there should be no (closed) term of type False.

- i.e., there should be no proof of False.
- `fun bot()=bot():∀α.unit→α`
- If we can code `bot` in Coq:
  `bot(): False`
- Note that other things, including state, exceptions, concurrency, continuations, can lead to the same sort of problems.

A pattern: monads

As in Haskell, distinguish purity with types:

- `e : int`
  - **e** is equivalent to an integer *value*
- `e :` Cmd `int`
  - **e** is a computation or *command* which when run in a world w either diverges, or yields an int and some new world w'.
  - Because computations are delayed, they are pure.
  - So we can safely manipulate them within types and proofs.
- `e :` Cmd `False`
  - possible, but means **e** must diverge when run!

Hoare Type Theory:

By *refining* Cmd with predicates, we can capture the effects of an imperative computation within its type.

```
e  :  Cmd{P}x:int{Q}
```

When run in a world satisfying `P`, `e` either

- diverges, or else
- terminates with an integer `x` and world satisfying `Q`.

i.e., Hoare-logic meets Type Theory

Hoare Type Theory (HTT)

- Dependently-typed, pure, core functional language
  - really pure, no effects including divergence
  - importantly, functions are always pure.
  - so function calls can safely appear within predicates.
- Layer on top of this a language for building *commands*
  - c : Cmd {P}x:T{Q}
  - a delayed computation which when run in a world satisfying P
  - either diverges or returns a value x of type T
  - and puts us in a world satisfying Q
- Commands are delayed
  - So building a command doesn't have any effects
  - (The command is only run *outside* the language.)
  - So even commands can safely appear in types and predicates.

Implementing HTT

- We embedded HTT into Coq
  - (could do this in other settings like Agda)
  - Coq provides us the pure, dependently-typed core language
  - It also provides a powerful logic (CiC)
  - And an interactive theorem-proving environment
- Coq gets the dependency, proofs, etc. *right*.
  - much more powerful than GADTs or related ideas.
  - actually *simplifies* things considerably.
- So in a sense, all we're doing is suggesting how to add *effects* to Coq.
  - Not a new idea (c.f., NuPRL's bar types, W.Swiestra's Agda work)
  - One key difference is that our worlds are "bigger" than what you can encode within Coq.
  - In particular, our stores allow you to store computations.

Reasoning about pointers…

- A long standing issue with Hoare logic is finding a modular treatment of pointers to heap-allocated data.
- The key issue is this:
  - Suppose we start in a state s such that:
    - sorted(x:linked-list) ∧ non-empty(y:queue)
  - Now suppose we have a dequeue operation for y:
    - e.g., dequeue : Cmd {non-empty(y)}z:T{true}
  - We can use the rule of consequence to forget about x and then invoke the dequeue command, but then we lose information about x.
- The insight is that x and y are referring to distinct regions in memory.
  - But to take advantage of this, we need to show that each location in x is disjoint from each location in y.
  - And how do you do this without leaking implementation details?

Separation Logic

In HTT we used a form of *separation logic* (Reynolds & O'Hearn) for our specifications.

- predicates that incorporate a notion of capability or ownership.
  - emp is only satisfied by the empty heap
  - x → e is only satisfied by the heap that contains one location x, pointing to a value e.
- connectives capture disjoint ownership.
  - P * Q describes a store s that can be broken into disjoint fragments s1 and s2 such that P(s1) and Q(s2).
  - (P1 * P2 * … * Pn) captures that disjoint(Pi,Pj) for all i,j.
- commands can only access locations they are given in their spec
  - This ensures a frame condition on e.g., procedures
  - If c : Cmd{P}{Q} and s |= (P*R) then after calling c in state s, I get a state that satisfies (Q*R).

A simple, imperative ADT

```
Parameter stack : Set -> Set.

Parameter rep (T : Set) : stack T -> list T -> hprop.

Parameter empty (T : Set) :
      Cmd emp (fun s : stack T => rep s nil)

Parameter push T (s : stack T) (x : T)(ls : [list T]) :
      Cmd (rep s ls) (fun _ : unit => rep s (x :: ls)).

Parameter pop T (s : stack T) (ls : [list T]) :
      Cmd (rep s ls)
          (fun xo : option T =>
             match xo with
             | None => [ls = nil] * rep s ls
             | Some x => Exists ls' :@ list T,
                                [ls = x :: ls'] * rep s ls'
            end)
```

A linked-list implementation

```
Record node : Set := Node {
   data : T;
   next : option ptr

}.

Definition stack = ptr.

Fixpoint listRep (ls : list T) (hd : option ptr) : hprop :=
   match ls with
   | nil => [hd = None]
   | h :: t => match hd with
               | None => [False]
               | Some hd => Exists p :@ option ptr,
                            hd --> Node h p * listRep t p


               end
   end.

Definition rep (s : stack) (ls : list T) : hprop :=
   Exists p :@ option ptr, s ⟶ p * listRep ls p.
```

The push code

```
Definition push(s:stack)(x:T)(ls:[list T]) :
     Cmd (ls ~~ rep s ls)
          (fun _ : unit => ls ~~ rep s (x :: ls)).

  refine (fun s x ls => hd <- !s;

                           nd <- New (Node x hd);

                          {{s ::= Some nd}}

  ); unfold rep ; sep fail auto.
```

Each line induces a verification-condition as a predicate which is then fed to the `sep` tactic.

In this case, the tactic can easily discharge the verification conditions (when we tell it to unfold the definition of the `rep` predicate.)

For more complicated code

- We provide a generic tactic that understands separation logic.
  - e.g., `x→v * y→z -> x ≠ y`
  - these build on a library of separation lemmas
  - and other tactics included with Coq
- The tactic is parameterized so you can add domain-specific reasoning.
  - e.g., unrolling definitions like `rep`.
- In practice, works extremely well for building fully verified, imperative ADTs.
  - stacks, association lists, queues, trees, hash-tables, etc.

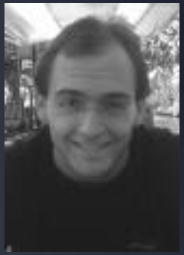
[Details in ICFP'09]

What about systems?

Is it feasible to build a complete system?

- not just state, but I/O & exceptions
- feasible to specify desired semantics?
- feasible to construct & maintain proofs?

Ysql [PoPL'10]



In-core database (c.f., MySQL) including:

- Definitions of schemas, relations, & queries
  - define meaning of queries as denotational semantics
  - define a simple cost model for queries
- Routines for I/O
  - [de]serialize tables to disk; proof that deserialize(serialize x) = x
  - query parser
- Query optimizer
  - prove correctness w.r.t. semantics
  - prove cost preservation where possible
- Execution engine
  - uses B+-trees for in-core representation
  - use Cmd monad for imperative operations
  - prove (partial) correctness w.r.t. query semantics

What’s missing?

- No concurrency
  - Integrating ideas from concurrent separation logic to make this feasible.
- Performance
  - The OCaml code that is extracted has many inefficiencies.
  - And we must trust the OCaml compiler!

Coq compilation

- Goal: verified compiler for Coq
  - need model of Coq in Coq
  - need verified extractor to core-ML
  - let Adam's compiler take over
- But there are more opportunities:
  - For the pure fragment, we have the luxury of choosing evaluation order.
    - all the advantages of Haskell & ML!
  - Opportunities for introducing parallel constructs.
    - e.g., reductions may require proof that combining operation is associative.

To wrap up

- I believe that in 10 years time, we will have the tools needed to build fully verified code in a cost-effective way.
  - at least for safety and security critical code.
- Hard challenges remain
  - concurrency is still amazingly difficult
  - maintainable specifications & proofs
- Domain-specific languages hit a sweet spot.
  - can afford to build verified checkers, compilers.
  - amortize the cost of proofs across many programs.

Proving Seq Associative

```
Lemma seq_assoc :
   ∀c1 c2 c3 s1 s2,
   evals (Seq c1 (Seq c2 c3)) s1 s2 ->
   evals (Seq (Seq c1 c2) c3) s1 s2.

Proof.
   intros.
   inversion H ; subst.
   inversion H5 ; subst.
   econstructor.
   econstructor.
   eapply H2.
   eapply H3.
   eapply H7.

Qed.
```

```
1 subgoal

==============================
∀c1 c2 c3 s1 s2,
evals (Seq c1 (Seq c2 c3)) s1 s2 ->
evals (Seq (Seq c1 c2) c3) s1 s2.
```

Proving Seq Associative

```
Lemma seq_assoc :
   ∀c1 c2 c3 s1 s2,
   evals (Seq c1 (Seq c2 c3)) s1 s2 ->
   evals (Seq (Seq c1 c2) c3) s1 s2.

Proof.
   intros.
   inversion H ; subst.
   inversion H5 ; subst.
   econstructor.
   econstructor.
   eapply H2.
   eapply H3.
   eapply H7.

Qed.
```

```
1 subgoal

===============================
∀c1 c2 c3 s1 s2,
evals (Seq c1 (Seq c2 c3)) s1 s2 ->
evals (Seq (Seq c1 c2) c3) s1 s2.
```

Proving Seq Associative

```
Lemma seq_assoc :
   ∀ c1 c2 c3 s1 s2,
   evals (Seq c1 (Seq c2 c3)) s1 s2 ->
   evals (Seq (Seq c1 c2) c3) s1 s2.

Proof.
   intros.
   inversion H ; subst.
   inversion H5 ; subst.
   econstructor.
   econstructor.
   eapply H2.
   eapply H3.
   eapply H7.

Qed.
```

```
1 subgoal

 c1 : stmt
 c2 : stmt
 c3 : stmt
 s1 : state
 s2 : state
 H : evals (Seq c1 (Seq c2 c3)) s1 s2
 ==============================
  evals (Seq (Seq c1 c2) c3) s1 s2
```

Proving Seq Associative

```
Lemma seq_assoc :
  ∀ c1 c2 c3 s1 s2,
  evals (Seq c1 (Seq c2 c3)) s1 s2 ->
  evals (Seq (Seq c1 c2) c3) s1 s2.

Proof.
  intros.
  inversion H ; subst.
  inversion H5 ; subst.
  econstructor.
  econstructor.
  eapply H2.
  eapply H3.
  eapply H7.

Qed.
```

```
1 subgoal

  c1 : stmt
  c2 : stmt
  c3 : stmt
  s1 : state
  s2 : state
  H : evals (Seq c1 (Seq c2 c3)) s1 s2
  ==============================
   evals (Seq (Seq c1 c2) c3) s1 s2
```

Proving Seq Associative

```
Lemma seq_assoc :
  ∀ c1 c2 c3 s1 s2,
  evals (Seq c1 (Seq c2 c3)) s1 s2 ->
  evals (Seq (Seq c1 c2) c3) s1 s2.

Proof.
  intros.
  inversion H ; subst.
  inversion H5 ; subst.
  econstructor.
  econstructor.
  eapply H2.
  eapply H3.
  eapply H7.

Qed.
```

```
1 subgoal

  c1 : stmt
  c2 : stmt
  c3 : stmt
  s1 : state
  s2 : state
  H : evals (Seq c1 (Seq c2 c3)) s1 s2
  s3 : state
  H2 : evals c1 s1 s3
  H5 : evals (Seq c2 c3) s3 s2
  ============================
   evals (Seq (Seq c1 c2) c3) s1 s2
```

Proving Seq Associative

```
Lemma seq_assoc :
   ∀c1 c2 c3 s1 s2,
   evals (Seq c1 (Seq c2 c3)) s1 s2 ->
   evals (Seq (Seq c1 c2) c3) s1 s2.

Proof.
   intros.
   inversion H ; subst.
   inversion H5 ; subst.
   econstructor.
   econstructor.
   eapply H2.
   eapply H3.
   eapply H7.

Qed.
```

```
1 subgoal

 c1 : stmt
 c2 : stmt
 c3 : stmt
 s1 : state
 s2 : state
 H : evals (Seq c1 (Seq c2 c3)) s1 s2
 s3 : state
 H2 : evals c1 s1 s3
 H5 : evals (Seq c2 c3) s3 s2
 ==============================
  evals (Seq (Seq c1 c2) c3) s1 s2
```

Proving Seq Associative

```
Lemma seq_assoc :
   ∀ c1 c2 c3 s1 s2,
   evals (Seq c1 (Seq c2 c3)) s1 s2 ->
   evals (Seq (Seq c1 c2) c3) s1 s2.

Proof.
   intros.
   inversion H ; subst.
   inversion H5 ; subst.
   econstructor.
   econstructor.
   eapply H2.
   eapply H3.
   eapply H7.

Qed.
```

```
1 subgoal

 c1 : stmt
 c2 : stmt
 c3 : stmt
 s1 : state
 s2 : state
 H : evals (Seq c1 (Seq c2 c3)) s1 s2
 s3 : state
 H2 : evals c1 s1 s3
 H5 : evals (Seq c2 c3) s3 s2
 s4 : state
 H3 : evals c2 s3 s4
 H7 : evals c3 s4 s2
 ===============================
  evals (Seq (Seq c1 c2) c3) s1 s2
```

Proving Seq Associative

```
Lemma seq_assoc :
  ∀ c1 c2 c3 s1 s2,
  evals (Seq c1 (Seq c2 c3))
  evals (Seq (Seq c1 c2) c3

Proof.
  intros.
  inversion H ; subst.
  inversion H5 ; subst.
  econstructor.
  econstructor.
  eapply H2.
  eapply H3.
  eapply H7.

Qed.
```

```
2 subgoals

 c1 : stmt
 c2 : stmt
 c3 : stmt
 s1 : state
 s2 : state
 H : evals (Seq c1 (Seq c2 c3)) s1 s2
 s3 : state
 H2 : evals c1 s1 s3
 H5 : evals (Seq c2 c3) s3 s2
 s4 : state
 H3 : evals c2 s3 s4
 H7 : evals c3 s4 s2
 ============================
  evals (Seq c1 c2) s1 ?969

subgoal 2 is:
 evals c3 ?969 s2
```

Proving Seq As

```
Lemma seq_assoc :
   ∀c1 c2 c3 s1 s2,
   evals (Seq c1 (Seq c2 c3)
   evals (Seq (Seq c1 c2) c3

Proof.
   intros.
   inversion H ; subst.
   inversion H5 ; subst.
   econstructor.
   econstructor.
   eapply H2.
   eapply H3.
   eapply H7.

Qed.
```

```
3 subgoals

 c1 : stmt
 c2 : stmt
 c3 : stmt
 s1 : state
 s2 : state
 H : evals (Seq c1 (Seq c2 c3)) s1 s2
 s3 : state
 H2 : evals c1 s1 s3
 H5 : evals (Seq c2 c3) s3 s2
 s4 : state
 H3 : evals c2 s3 s4
 H7 : evals c3 s4 s2
 ============================
  evals c1 s1 ?970

subgoal 2 is:
 evals c2 ?970 ?969
subgoal 3 is:
 evals c3 ?969 s2
```

Proving Seq Associative

```
Lemma seq_assoc :
   ∀c1 c2 c3 s1 s2,
   evals (Seq c1 (Seq c2 c3))
   evals (Seq (Seq c1 c2) c3)

Proof.
   intros.
   inversion H ; subst.
   inversion H5 ; subst.
   econstructor.
   econstructor.
   eapply H2.
   eapply H3.
   eapply H7.

Qed.
```

```
2 subgoals


 c1 : stmt
 c2 : stmt
 c3 : stmt
 s1 : state
 s2 : state
 H : evals (Seq c1 (Seq c2 c3)) s1 s2
 s3 : state
 H2 : evals c1 s1 s3
 H5 : evals (Seq c2 c3) s3 s2
 s4 : state
 H3 : evals c2 s3 s4
 H7 : evals c3 s4 s2
 ============================
  evals c2 s3 ?969


subgoal 2 is:
 evals c3 ?969 s2
```

Proving Seq Associative

```
Lemma seq_assoc :
   ∀c1 c2 c3 s1 s2,
   evals (Seq c1 (Seq c2 c3)) s1 s2 ->
   evals (Seq (Seq c1 c2) c3

Proof.
   intros.
   inversion H ; subst.
   inversion H5 ; subst.
   econstructor.
   econstructor.
   eapply H2.
   eapply H3.
   eapply H7.

Qed.
```

```
1 subgoal

  c1 : stmt
  c2 : stmt
  c3 : stmt
  s1 : state
  s2 : state
  H : evals (Seq c1 (Seq c2 c3)) s1 s2
  s3 : state
  H2 : evals c1 s1 s3
  H5 : evals (Seq c2 c3) s3 s2
  s4 : state
  H3 : evals c2 s3 s4
  H7 : evals c3 s4 s2
  ============================
   evals c3 s4 s2
```

Proving Seq Associative

```
Lemma seq_assoc :
   ∀c1 c2 c3 s1 s2,
   evals (Seq c1 (Seq c2 c3)) s1 s2 ->
   evals (Seq (Seq c1 c2) c3) s1 s2.

Proof.
   intros.
   inversion H ; subst.
   inversion H5 ; subst.
   econstructor.
   econstructor.
   eapply H2.
   eapply H3.
   eapply H7.

Qed.
```

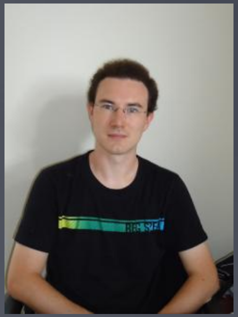
Proof completed.

Acknowledgements

A. Nanevski

A. Chlipala

M. Sozeau


JB. Tristan

A. Shinnar

R. Wisnesky

G.Malecha

P. Govereau