Harvard
School of Engineering
and Applied Sciences

Intro to Hoare Type Theory

Greg Morrisett

A pattern: Monads

As in Haskell, distinguish purity with types:

- `e : int`
  - **`e`** is equivalent to an integer *value*
- `e :` ST `int`
  - **`e`** is a *computation* 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 :` ST `False`
  - possible, but means **`e`** must diverge when run!

Hoare Type Theory:

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

`e : ST{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:

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

Why refine the type? Why not just unroll the definition as we did in type the type-inference example?

One reason is that we want to be able extract code that actually uses a mutable store.

- if we expose the definition of ST, then you could write an ST command which, say, copies the whole heap and returns it as a value!

Defining ST in Coq?

Can try to define:

```
Record res(i:heap)(A:Type)(Q:post A) :=
  mkRes { res_h : heap ;
          res_v : A ;
          res_p : Q i res_v res_h }.

ST P A Q := forall (i:heap), P h -> res i A Q.
```

but then you sacrifice:

- recursive (diverging) computations
- non-deterministic computations
- code that stores computations in the heap

So we add ST and its constructors as axioms.

What about Consistency

But can we use ST to prove False?

- No equations on computations!
  - (except for monad laws)
  - no way to *run* them, even in the logic.
- Trivial model: ST = unit.
  - silly, but ensures we haven't broken Coq.
- Intermediate model:
  - ST = predicate transformers
  - not big enough -- can't store ST's in heap!
- Denotational (category-theoretic) model:
  - Lars Birkedal & Rasmus Peterson
  - (subset of Coq corresponding to HTT)

Operational Soundness

- For HTT subset [ESOP'07]:
  - give a fairly standard, small-step operational semantics
  - assume "heap-consistency" of logic
    - we get this with the trivial model
  - proved preservation & progress
- Not an everyday type-soundness proof.
  - not in the context of Coq
  - took advantage of hereditary substitutions

A Very Simple Example:

```
Definition postinc x :=
  xv <- !x ;
  x ::= (xv + 1) ;;
  ret xv.
```

Expanding the Definition

```
Definition postinc x :=
  xv <- !x ;
  x ::= (xv + 1) ;;
  ret xv.
```

Expands into:

```
  Definition postinc x :=
    bind (read x) (λ xv =>
      bind (write x (xv + 1)) (λ _ =>
        ret xv)).
```

Primitive commands are sequenced with bind & ret.

Roadmap

```
ST(P:pre)(A:Type)(Q:post A):Type
```

- Heaps, pre- & post-conditions
- Basic monadic & state constructs
- Example: hashtable
- Modularity & separation

Modeling Heaps in Coq

```
ST(P:pre)(A:Type)(Q:post A):Type.

(* pre-conditions classify heaps, where
   a heap maps pointers to values. *)
Inductive dynamic : Type :=
  | Dyn : forall T, T -> dynamic.
heap := ptr -> option dynamic
pre := heap -> Prop

(* post-conditions relate a return value,
  the input heap and the output heap.*)
post(A:Set) := A -> heap -> heap -> Prop.
```

Some Primitives & Types:

```
(* ret just returns the value x, with no effect*)
ret(x:A) :
  ST True (y:A) (y=x ∧ final=initial).

(* allocate and initialize a fresh location *)
new(v:A) :
  ST True (y:ptr)
      (unallocated initial y ∧
       final = update initial y v).
```

Note: I’m cheating and using “initial” and “final” where I should be using lambda-bound variables, as well as some math notation.

More Primitives

```
(* read location x, getting out an A value *)
read(x:ptr) :
       ST (∃v:A,ptsto x v initial)
          (y:A)
          (final = initial ∧ ptsto x y initial)

(* write v into location x *)
write(x:ptr)(v:A) :
        ST (∃B:Type,v:B,ptsto x v initial)
           (_:unit)
           (final = update x v initial)

ptsto x y h := h x = Some y
update x y h z := if ptr_eq_dec x z then y else h x
```

Bind

```
(* sequentially compose computations *)
bind(C1: ST P1 A1 Q1)
    (C2: ∀x:A1, ST (P2 x) A2 (Q2 x)),
 ST (P1 initial ∧
      (∀z h, Q1 z initial h -> P2 z h))
    (y:A2)
      (∃z h, Q1 z initial h ∧
                 Q2 z y h final)
```

In words:

- The weakest pre-condition needed to run (C_1; C_2).
- The strongest post-condition when (C_1; C_2) terminates.

Type Inference in Ynot.

The types of our combinators *compute* a principal type.

- For loop-free code, we infer a most general specification.
- So for most code, you don’t have to give specifications.
- And you can be ensured that we aren’t going to prohibit some specification.

The bad news:

- Loops require specifications.
  - But being able to factor out iterators using lambda makes this not as bad as it first sounds.
- Inferred types are ridiculous.

Recall our Example:

```
Definition postinc x :=
  bind (read x) (λ xv =>
    bind (write x (xv + 1)) (λ _ =>
      ret xv)).

Check postinc.
```

Argh!

```
postinc(x:loc) :
   ST
   (λ i => (ref nat x i) /\
             (∀z m, m = i /\
               (∀v, ptsto x v i -> Val z = Val v) ->
                 (∃A:Type, ref A x m) /\
                   (∀(u:unit) m0, Val u = Val tt /\
                    m0 = update_loc m x (z + 1) -> True))
   nat
   (λ (y:ans nat) (i m:heap) =>
    (ref nat x i) /\
    (∃z:nat, ∃h:heap, (h = i /\
          (∀v:nat, ptsto x v i -> Val z = Val v)) /\
       (∃A:Type, ref A x h) /\
       (∃u:unit,
          ∃h0:heap,
            (Val u = Val tt /\
             h0 = update_loc h x (z + 1)) /\ m = h0 /\ y = Val x0)).
```

Semantic Type Casting

Fortunately, we can explicitly coerce when we want a different type -- it just demands a proof.

```
(* strengthen pre- and weaken post-condition *)
cast:
  ∀(C : ST P1 A Q1),
   (∀i, P2 i -> P1 i ∧
    ∀x f, Q1 x i f -> Q2 x i f) ->
          ST P2 A Q2.
```

With an Explicit Cast

```
Definition postinc x :
  ST (∃n:A,ptsto x n initial)
     (y:nat)
     (∀n, ptsto x n initial ->
        y = n ∧
        final = update x (n+1) initial).

 refine (cast (xv <- !x ;
               x ::= (xv + 1) ;;
               ret xv)) _ ; …
```

A Bigger Example: Hashtables

- Representation: array of list(key*value).
  - `table := {len:nat ; arr:array len}`
- Abstraction: (key*value) sets.
  - We *model* the hash-table as a pure set.
  - And connect the implementation with an abstract representation predicate.

    `reps(S:Set(key*value))(t:table)(h:heap)`
  - Holds when (k,v) is in S iff (k,v) is in the list at index (hash k mod len)
- Operations to create, destroy, insert, lookup, iterate, etc.

Type of Create:

```
create(n:nat) :
 ST True (y:table)
   ((* old values in memory are preserved *)
    ∀r A (v:A),ptsto r v initial ->
                ptsto r v final ∧
    (* array locations are fresh *)
    ∀j (p:j<len t),
          unallocated (vsub (arr t) j pj) i
    (* returns a table that represents {} *)
    reps {} y final).
```

Type of Lookup

```
lookup :
  ∀(k:key)(t:table)(S:kvset),
  ST (reps S t inital) (y:option value)
     ((* memory is unchanged *)
       final=initial ∧
      (* returns None when key isn't in S *)
      (∀v,(k,v) ∉ S ∧ y=None ∨
      (* or else returns some v s.t. (k,v) ∈ S *)
       ∃v, y=Some v ∧ (k,v) ∈ S ))
```

Type of Insert:

```
insert(k:key)(v:value)(t:table)(S:kvset),
 ST (reps S t) (_:unit)
  ((* only changes location at (hash k) mod n *)
   (∀ r w,(r <> vsub (arr t)((hash k) mod (len t))) ->
                ptsto r w initial -> ptsto r w final)
    ∧
   (* table t now represents {(k,v)}+S *)
   reps ({(k,v)} ∪ S) t final).
```

Modularity

Thus far, a “big footprint” approach.

– specify changes to *entire* heap.
– inconvenient specifications -- always have to make sure we specify what we don’t change.
– not modular -- had to leak implementation details in specification of insert.

So we define a “small footprint” approach based on separation logic.

– Simpler specifications.
– More importantly, hides abstraction details.
– Downside: proofs are harder?
    - Not any more – thanks to Adam’s separation tactics, we have an order of magnitude reduction in the size of the proof scripts.

STsep

We define:

```
STsep(P:pre)(A:Set)(Q: post A)
```

to be an ST computation such that:

- If we start in a heap ($h_1 + h_2$),
- where h_1 satisfies P,
- then we'll end up with a heap ($h_1' + h_2$),
- where h_1' satisfies Q,

That is, STsep ensures we don't modify locations outside our specification.

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) $\wedge$ non-empty(y:queue)
  - Now suppose we have a dequeue operation for y:
    - e.g.., dequeue : ST {non-empty(y)}x:T{true}
  - We can use the rule of consequence to forget about x and then invoke the dequeue command.
  - But then afterwards, I've lost the facts that I knew about y!
  - (This is the same reason you don't want to trade subtyping for polymorphism…)
  - The key insight is that x and y are referring to distinct regions in memory.
  - But tracking pair-wise that each x and y are disjoint is a pain.
  - And how do you do this without leaking implementation details?
    - e.g., how do I know x:T is disjoint from y:U when T & U are abstract types?

Separation Logic

So separation logic introduces three key things:

- predicates incorporate a notion of 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).
- we defined separation-style connectives on top of the core HTT.

Separating Interfaces:

```
create(n:nat) :
 STsep (emp initial)
       (t:table)
       (reps {} t final).
```

- The pre-condition specifies an “empty” footprint.
- But we can run it in any larger heap.
- Anything returned is automatically “fresh”
- And that all other locations remain unmodified.

Separating Type of Insert:

```
insert(k:key)(v:value)(t:table)(S:kvset),
  ST (reps S t initial) (_:unit)
     (reps ({(k,v)} ∪ S) t final).
```

- The pre-condition specifies only the part of the heap that is involved in the representation of S.
- So we know that no other locations are modified by insertion.
- But crucially, the set of locations that make up the representation is abstract.
- That is, no implementation details are leaked.

Abstraction

```
Record FinMap(key value:Type) :=
 { t : Type ;
   reps : set(key*value)->t->heap->Prop ;
   create(n:nat) :  STsep emp t … ;
   insert k v t S:  STsep (reps S t) unit …;
   fold : ...
 }
```

We’ve built a range of implementations that meet the interface: association lists, hash-tables, splay-trees…

- Hashtable actually dogfoods the interface.
- The fold operation captures *aggregate* effects.

Roadmap

`ST(P:pre)(A:Set)(Q:post A) : Set`

- ✓ Heaps, pre- & post-conditions
- ✓ Basic monadic & state constructs
- ✓ Example: hashtable
- ✓ Modularity & separation

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)
- Main components:
  - Definitions of schemas, relations, & queries
    - define meaning of queries as denotational semantics
    - define a simple cost model for queries
  - Routines for [de]serializing tables to disk
    - proof that deserialize(serialize x) = x
  - Query parser
    - uses a Packrat, memoizing parser library
  - 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



Our RDBMS Pipeline

(LOC)	Front-End	Back-End
Functional	1200	810
Imperative	0	920
Proof	1140	6880
Proof/ Code	**≈ 1**	**≈ 4**

Relational Algebra

Cost Respecting

Finite Map Operations

Imperative B+ Tree

SQL Query → Compilation → Optimization → Planning → Execution → Table / Relation

The table the SQL query denotes and
the table the RDBMS returns are equal (partial correctness).

To Start With…

- We need a model of the DB to state correctness.
- We start by defining schemas, tuples, and relations.
  - many possible ways to encode these in Coq
  - I’ll show you what we did, but we want to investigate others
- We then define our query language.
  - typed abstract syntax
  - denotational semantics: map queries and input relations to an output relation.

Some Coq Definitions

```
Def Schema := list Type.
Def s : Scheme :=
  (string::nat::string::nil).
Fix Tuple (T: Schema) : Type :=
 match T with
  | Nil ⇒ unit
  | Cons a b ⇒ a * Tuple b
 end.
Def t : Tuple s :=
  (“Greg”,(43,(“PL”,tt)))
Def Table (T: Schema) :  :=
  FiniteSet (Tuple T).
```

Query Abstract Syntax

```
Inductive RAExp (G: Context) : Schema → Type →
   Type  :=
 | Var : ∀(v: name), RAExp G (G v)
 | Union : ∀(t: Schema),
     RAExp G t → RAExp G t → RAExp G t
 | Select : ∀(t: Schema),
     RAExp G t → (Tuple t → bool) → RAExp G t
 | …
 | Product : ∀(t t’: Schema),
     RAExp G t → RAExp G t’ → RAExp G (t ++ t’).
```

Denotational Semantics

```
Fix denote (G: Context) (env:Env G)
  (T:Schema) (q : RAexp G T) : Table T :=
  match q with
  | Var v => env v
  | Union T q1 q2 =>
      FiniteSet.union (denote G env T q1)
                      (denote G env T q2)
  | Select T q f =>
      FiniteSet.filter
       (denote G env T q) f
  | ...
```

Verifying Query Optimization

```
Def rewrite G T : Type :=
    RAExp G T → RAExp G T.

Def semantics_preserving G T (r:
  rewrite T) : Prop :=
  ∀(q: RAExp T), denote q = denote
  (r q).

Def optimization T : Type :=
  { r   : rewrite T
  ; pf  : semantics_preserving r }
```

Some of the optimizations

- Ryan proved a whole bunch of relational algebra identities using the denotational semantics.
  - e.g., filter P1 (filter P2 R) = filter P2 (filter P1 R)
- Then he implemented a number of textbook query optimizations.
  - e.g., select P2 (select P1 Q) → select (P1 and P2) Q
- And constructed proofs of equivalence.
  - note, manipulating typed AST was a pain
  - See Ryan's next talk for more on this.

One More Syntax Issue

- Need to parse queries, [de]serialize tables
- Built a packrat parsing combinator library
  - the types of the combinators tell you what grammar (really transducer) they implement.
  - packrat parsing uses memoization (i.e., refs) to avoid some backtracking
  - but that's a whole other story…
- Allows us to validate the parser against a grammar.
- Allows us to prove a roundtrip theorem for tables:
  deserialize(serialize(T)) = T

The B+-trees

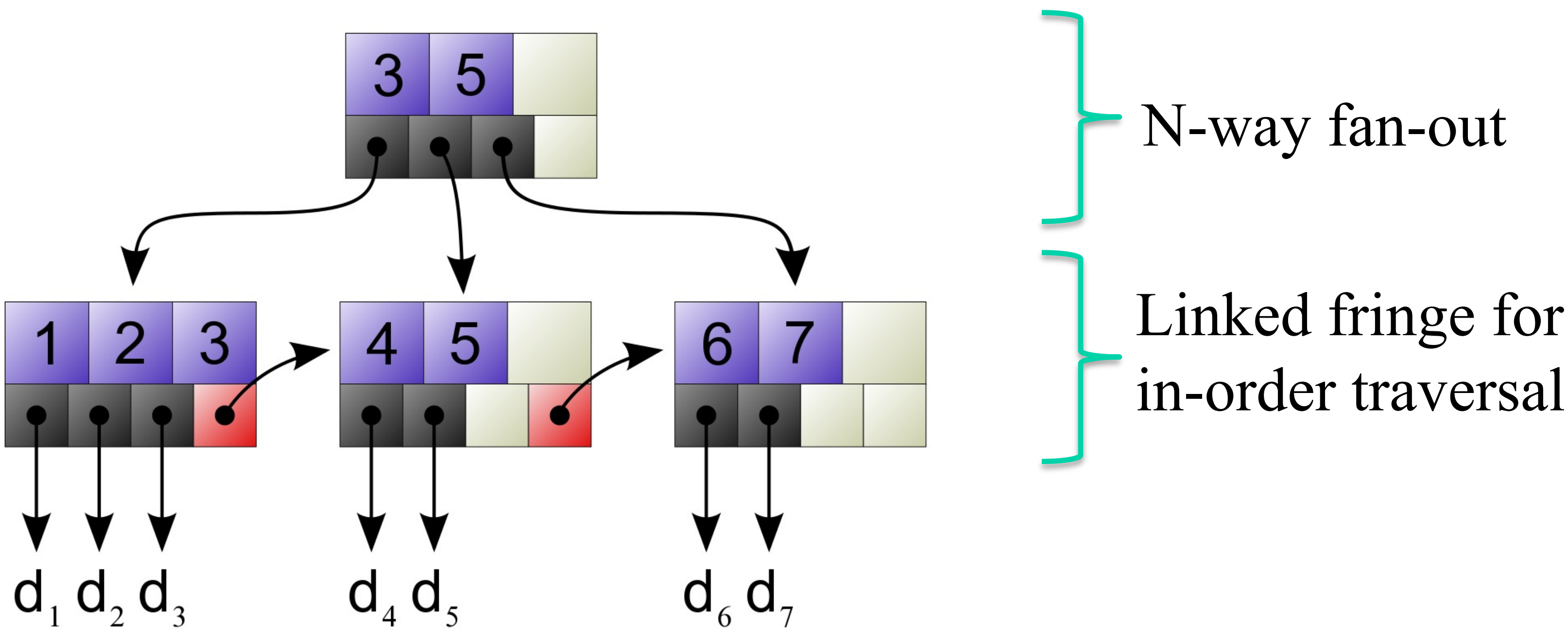
- An interface mediates between the query engine and the B+-trees.
- The interface captures the idea that a tree (an ADT) represents a (functional) finite-map.
  - important: we want to be able to swap alternative implementations, with alternative internal invariants.
- The operations on the tree are reflected in the pre- and post-conditions as (functional) operations on the finite-map.
- So building the query engine in terms of the interface is easy – just have to reason about finite maps.

An Imperative Finite Map ADT

```
Class Fmap (K V: Type) : Type := {
 handle : Type;
 model  : Type := list (K*V);
 rep    : handle → model → heap → Prop;
 add   : ∀(m: model) (k: K) (v: V) (h: handle),
           Cmd (rep h m )
              (fun _: unit ⇒ rep h ((k,v)::m));
 lookup  : ∀(m: model) (k: K) (h: handle),
           Cmd (rep h m )
              (fun vopt : option V ⇒ rep h m * [vopt =
  find k m]);
 ;
 iterate : ... ;
...
```

The Implementation

- Generalized Binary Search Trees



The “rep” predicate

- Recall we are supposed to relate the B+-tree to some (functional) list of key-value pairs.
- Intuitively, rep t l should hold when the leaves of t are some permutation of the list l.
- But in addition to this fact, we want to capture what it means for a B+-tree to be well formed.
  - e.g., balance conditions
  - in practice, we relate the B+-tree to a functional tree without a skirt, but that’s balanced

The Challenge

- Trees are nice for separation logic
  - each sub-tree is disjoint
- But the B+-tree is really two data structures that physically share:
  - a tree and a linked list of the leaves
  - inserting a key/value wants to view things as a tree
    - but also link into the list of leaves
  - iteration wants to view things as a linked list
  - so finding an appropriate representation predicate is kind of tricky.

Part of the B+ Tree Rep

$$
\begin{array}{l}
\texttt{repTree}\ 0\ r\ optr\ (p', ls) \iff \\
\quad [r = p'] * \exists ary.\ r \mapsto \texttt{mkNode}\, 0\ ary\ optr\ * \\
\qquad \texttt{repLeaf}\ ary\ |ls|\ ls
\end{array}
$$

$$
\begin{array}{l}
\texttt{repTree}\ (h+1)\ r\ optr\ (p', (ls, nxt)) \iff \\
\quad [r = p'] * \exists ary.r \mapsto \texttt{mkNode}\ (h+1)\ ary\ (\texttt{ptrFor}\ nxt) * \\
\qquad \texttt{repBranch}\ ary\ (\texttt{firstPtr}\ nxt)\ |ls|\ ls\ * \\
\qquad \texttt{repTree}\ h\ (\texttt{ptrFor}\ nxt)\ optr\ nxt
\end{array}
$$

$$
\begin{array}{l}
\texttt{repLeaf}\ ary\ n\ [v_1, ..., v_n] \iff \\
\quad ary[0] \mapsto \texttt{Some}\ v_1 * ... * ary[n-1] \mapsto \texttt{Some}\ v_n * \\
\quad ary[n] \mapsto \texttt{None} * ... * ary[SIZE-1] \mapsto \texttt{None}
\end{array}
$$

$$
\begin{array}{l}
\texttt{repBranch}\ ary\ n\ optr\ [(k_1, t_1), ..., (k_n, t_n)] \iff \\
\quad ary[0] \mapsto \texttt{Some}\ (k_1, \texttt{ptrFor}\ t_1) * \\
\qquad \texttt{repTree}\ h\ (\texttt{ptrFor}\ t_1)\ (\texttt{firstPtr}\ t_2)\ t_1\ \ * ... * \\
\quad ary[n-2] \mapsto \texttt{Some}\ (k_{n-1}, \texttt{ptrFor}\ t_{n-1}) * \\
\qquad \texttt{repTree}\ h\ (\texttt{ptrFor}\ t_{n-1})\ (\texttt{firstPtr}\ t_n)\ t_{n-1} * \\
\quad ary[n-1] \mapsto \texttt{Some}\ (k_n, \texttt{ptrFor}\ t_n) * \\
\qquad \texttt{repTree}\ h\ (\texttt{ptrFor}\ t_n)\ optr\ t_n * \\
\quad ary[n] \mapsto \texttt{None} * ... * ary[SIZE-1] \mapsto \texttt{None}
\end{array}
$$

Key Splitting Theorem

```
Theorem repTree_iff_repTrunk :
  ∀h (r : ptr) (optr : option ptr) (p :
  ptree h) (H : heap),
    repTree r optr p H <->
           (repTrunk r optr p *
            repLeaves (Some (firstPtr p))
  (leaves p) optr) H.
```

To Wrap Up

- Systems like Coq make it possible to write code and prove *deep* properties about it.
  - from simple types to full correctness.
- Provides a uniform, *modular* framework for:
  - types and specifications.
  - code, models, and proofs.
  - abstraction at all levels.
- Recent advances scale it from pure languages to effects without losing modularity.
  - monads.
  - separation logic.

But Lots to Do:

- Scaling the theory further:
  - IO, concurrency
  - liveness, information flow, …
- More automation:
  - better inference
  - adapt good decision procedures from SMT
  - termination analyses, shape analyses, etc.
- Re-think languages & environments:
  - in particular, for discharging explicit proofs

I Remain Optimistic

These obstacles will be overcome.

We won't develop *all* software with proofs of correctness, but I do believe that within another 10 years:

- type systems for mainstream languages will rule out language-level errors, and many library errors, and
- a lot more safety & security-critical code will be developed with machine-checked proofs of key properties.