The Science of Deep Specification

Benjamin C. Pierce
University of Pennsylvania

SPLASH
November, 2016
The Science of Deep Specification

Benjamin C. Pierce
University of Pennsylvania

SPLASH
November, 2016
We can’t build software that works!”
Or can we??
How did that happen?
• Better **programming languages**
  • Basic *safety guarantees* built in
  • Powerful mechanisms for *abstraction* and *modularity*
• Better **software development methodology**
• Stable **platforms and frameworks**
• Better use of **specifications**
• Better programming languages
  • Basic safety guarantees built in
  • Powerful mechanisms for abstraction and modularity
• Better software development methodology
• Stable platforms and frameworks
• Better use of specifications
  *i.e., descriptions of what software does (as opposed to how to do it)*
What are “deep” specifications?
Deep specifications are...

**Formal**
- mathematically precise

**Rich**
- precisely expressing intended behavior of complex software (a spectrum!)

**Live**
- automatically checked against actual code (not just a model)

**Two-sided**
- exercised by both implementations and clients
Deep specifications
(+ two-sided)

Type systems and other "Lightweight formal methods"

"Classic" specification languages (Z, VDM, ...)

Comprehensive informal specs (C, x86, AUTOSAR, ...)

Diagram showing a 3D cube with axes for Formal vs. Informal, Simple vs. Rich, and Disconnected vs. Live.
A Short Story
about a tiny compiler
and its specification(s)…
A datatype of stack machine instructions

Inductive instr : Type :=
| PUSH : nat -> instr
| PLUS : instr
| MINUS : instr
| MULT : instr.

Definition my_favorite_instructions : list instr :=
[PUSH 10; PUSH 4; MULT; PUSH 2; PLUS].

An example instruction sequence

(All examples in Gallina, the functional language of the Coq proof assistant)
Fixpoint execute (s : list nat) (p : list instr) : list nat :=
match (s, p) with
| (_,        nil)           => s
| (_,        (PUSH n) ::p') => execute (n    ::s)  p'
| (m::n::s', PLUS    ::p') => execute ((m+n)::s') p'
| (m::n::s', MINUS   ::p') => execute ((m-n)::s') p'
| (m::n::s', MULT    ::p') => execute ((m*n)::s') p'
| (_,        _        ::p') => execute s           p'
end.
A datatype of arithmetic expressions

Inductive exp : Type :=
| Num : nat -> exp
| Plus : exp -> exp -> exp
| Minus : exp -> exp -> exp
| Mult : exp -> exp -> exp.

Definition my_favorite_number : exp :=
   Plus (Mult (Num 10) (Num 4)) (Num 2).

An example value belonging to the type exp
A compiler from arithmetic expressions to stack instructions

Fixpoint compile (e : exp) : list instr :=
    match e with
    | Num n => [PUSH n]
    | Plus e1 e2 => compile e1 ++ compile e2 ++ [PLUS]
    | Minus e1 e2 => compile e1 ++ compile e2 ++ [MINUS]
    | Mult e1 e2 => compile e1 ++ compile e1 ++ [MULT]
    end.
Specifying our compiler...
An Informal Specification

Compiling an arithmetic expression should yield stack-machine instructions that compute the corresponding numeric result:

- (Plus e1 e2) means add the results of e1 and e2
- (Minus e1 e2) means subtract the results of e1 and e2
- (Mult e1 e2) means multiply the results of e1 and e2
A (Very) Simple Formal Specification

Fixpoint \texttt{compile} (e : \texttt{exp}) : list instr :=
match e with
| \texttt{Num} n => [\texttt{PUSH} n]
| \texttt{Plus} e1 e2 => \texttt{compile} e1 ++ \texttt{compile} e2 ++ [\texttt{PLUS}]
| \texttt{Minus} e1 e2 => \texttt{compile} e1 ++ \texttt{compile} e2 ++ [\texttt{MINUS}]
| \texttt{Mult} e1 e2 => \texttt{compile} e1 ++ \texttt{compile} e2 ++ [\texttt{MULT}]
end.
Another Simple Formal Specification

Fixpoint compile (e : exp) : list instr :=
  match e with
  | Num n => [PUSH n]
  | Plus e1 e2 => compile e1 ++ compile e2 ++ [PLUS]
  | Minus e1 e2 => compile e1 ++ compile e2 ++ [MINUS]
  | Mult e1 e2 => compile e1 ++ compile e1 ++ [MULT]
  end.

Example e1 : assert (eq (compile (Num 42)) [PUSH 42]).

Example e2 : assert (eq (compile (Plus (Num 2) (Num 2))) [PUSH 2; PUSH 2; PLUS]).
Can we do better?
Fixpoint compile (e : exp) : list instr :=
  match e with
  | Num n => [PUSH n]
  | Plus e1 e2 => compile e1 ++ compile e2 ++ [PLUS]
  | Minus e1 e2 => compile e1 ++ compile e2 ++ [MINUS]
  | Mult e1 e2 => compile e1 ++ compile e2 ++ [MULT]
  end.

Example e1 : assert (eq (compile (Num 4)) [PUSH 42]).

Example e2 : assert (eq (compile (Plus (Num 2) (Num 2))) [PUSH 2; PUSH 2; PLUS]).

For Coq savants:
Definition assert b := (b = true).
Fixpoint eval (e : exp) : nat :=
  match e with
  | Num n => n
  | Plus e1 e2 => (eval e1) + (eval e2)
  | Minus e1 e2 => (eval e1) - (eval e2)
  | Mult e1 e2 => (eval e1) * (eval e2)
end.

Example e3 :
assert (eq (execute [] (compile (Plus (Num 2) (Num 2))))
  [eval (Plus (Num 2) (Num 2))]).

“Executing the compiled code in an empty stack... yields a stack containing the result of evaluating the original expression.”

Operational semantics of the source language
Example e3 :
assert (eq (execute [] (compile (Plus (Num 2) (Num 2))))
[eval (Plus (Num 2) (Num 2))]).

Example e4 :
assert (eq (execute [] (compile (Plus (Num 5) (Num 3))))
[eval (Plus (Num 5) (Num 3))]).

Example e5 :
assert (eq (execute [] (compile (Mult (Num 0) (Num 3))))
[eval (Mult (Num 0) (Num 3))]).

Example e6 :
assert (eq (execute [] (compile (Mult (Num 2) (Num 2))))
[eval (Mult (Num 2) (Num 2))]).
Example e7 :
assert (eq (execute [] (compile (Mult (Num 3) (Num 1)))) [eval (Mult (Num 3) (Num 1))]).

Fixpoint compile (e : exp) : list instr :=
  match e with
  | Num n => [PUSH n]
  | Plus e1 e2 => compile e1 ++ compile e2 ++ [PLUS]
  | Minus e1 e2 => compile e1 ++ compile e2 ++ [MINUS]
  | Mult e1 e2 => compile e1 ++ compile e1 ++ [MULT]
  end.
Example e7:
assert (eq (execute [] (compile (Mult (Num 3) (Num 1)))) [eval (Mult (Num 3) (Num 1))]).

Fixpoint compile (e : exp) : list instr :=
 match e with
  | Num n => [PUSH n]
  | Plus e1 e2 => compile e1 ++ compile e2 ++ [PLUS]
  | Minus e1 e2 => compile e1 ++ compile e2 ++ [MINUS]
  | Mult e1 e2 => compile e1 ++ compile e2 ++ [MULT]
 end.
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Example e3:
assert (eq (execute [] (compile (Plus (Num 2) (Num 2))))
[eval (Plus (Num 2) (Num 2))]).

Example e4:
assert (eq (execute [] (compile (Plus (Num 5) (Num 3))))
[eval (Plus (Num 5) (Num 3))]).

...
Fixpoint compile (e : exp) : list instr :=
match e with
| Num n => [PUSH n]
| Plus e1 e2 => compile e1 ++ compile e2 ++ [PLUS]
| Minus e1 e2 => compile e1 ++ compile e2 ++ [MINUS]
| Mult e1 e2 => compile e1 ++ compile e2 ++ [MULT]
end.

Example e3:
assert (eq (execute [] (compile (Plus (Num 2) (Num 2))))
[eval (Plus (Num 2) (Num 2))]).

Example e4:
assert (eq (execute [] (compile (Plus (Num 5) (Num 3))))
[eval (Plus (Num 5) (Num 3))]).

...
Example e3:
assert (eq (execute [] (compile (Plus (Num 2) (Num 2)))) [eval (Plus (Num 2) (Num 2))]).

Example e4:
assert (eq (execute [] (compile (Plus (Num 5) (Num 3)))) [eval (Plus (Num 5) (Num 3))]).

...
Example e3:
assert (eq (execute [] (compile (Plus (Num 2) (Num 2))))
[eval (Plus (Num 2) (Num 2))]).

Example e4:
assert (eq (execute [] (compile (Plus (Num 5) (Num 3))))
[eval (Plus (Num 5) (Num 3))]).

Example e5:
assert (eq (execute [] (compile (Mult (Num 0) (Num 3))))
[eval (Mult (Num 0) (Num 3))]).

Example e6:
assert (eq (execute [] (compile (Mult (Num 2) (Num 2))))
[eval (Mult (Num 2) (Num 2))]).
Example e3 :  
assert (eq (execute [] (compile (Plus (Num 2) (Num 2)))) [eval (Plus (Num 2) (Num 2))]).

Example e4 : 
assert (eq (execute [] (compile (Plus (Num 5) (Num 3)))) [eval (Plus (Num 5) (Num 3))]).

Example e5 : 
assert (eq (execute [] (compile (Mult (Num 0) (Num 3)))) [eval (Mult (Num 0) (Num 3))]).

Example e6 :  
assert (eq (execute [] (compile (Mult (Num 2) (Num 2)))) [eval (Mult (Num 2) (Num 2))]).
Definition compiles_correctly (e : exp) :=
  eq (execute [] (compile e)) [eval e].

Example e3 :
  assert (eq (execute [] (compile (Plus (Num 2) (Num 2))))
    [eval (Plus (Num 2) (Num 2))]).

Example e4 :
  assert (eq (execute [] (compile (Plus (Num 5) (Num 3))))
    [eval (Plus (Num 5) (Num 3))]).

Example e5 :
  assert (eq (execute [] (compile (Mult (Num 0) (Num 3))))
    [eval (Mult (Num 0) (Num 3))]).

Example e6 :
  assert (eq (execute [] (compile (Mult (Num 2) (Num 2))))
    [eval (Mult (Num 2) (Num 2))]).
Definition compiles_correctly (e : exp) :=
    eq (execute [] (compile e)) [eval e].

Example e3 :
    assert (compiles_correctly (Plus (Num 2) (Num 2))).

Example e4 :
    assert (compiles_correctly (Plus (Num 5) (Num 3))).

Example e5 :
    assert (compiles_correctly (Mult (Num 0) (Num 3))).

Example e6 :
    assert (compiles_correctly (Mult (Num 2) (Num 2))).
Specification-Based Testing

Enumerative

Random

Concolic

etc.

etc.
Specification-Based Random Testing

Idea:

- Generate lots of random values of type `exp`
- See if `compiles_correctly` returns `true` for each of them

Haskell
QuickCheck
[Claessen&Hughes]
QuickChick compiles\_correctly.

Counterexample found after 4 tests:

\[\text{Plus}\left(\text{Plus}\left(\text{Minus}\left(\text{Num}\left(3\right)\right)\left(\text{Num}\left(0\right)\right)\right)\left(\text{Plus}\left(\text{Num}\left(3\right)\right)\left(\text{Num}\left(2\right)\right)\right)\right)\left(\text{Plus}\left(\text{Minus}\left(\text{Num}\left(0\right)\right)\left(\text{Num}\left(0\right)\right)\right)\left(\text{Mult}\left(\text{Num}\left(0\right)\right)\left(\text{Num}\left(3\right)\right)\right)\right)\right)\]
QuickChick compiles correctly.

Counterexample found after 4 tests:

\[
\begin{align*}
\text{Plus} & \ ( \text{Plus} \ ( \text{Minus} \ ( \text{Num} \ (3) \ ) \ ) \ ( \text{Num} \ (0) \ ) \ ) \\
& \ ( \text{Plus} \ ( \text{Num} \ (3) \ ) \ ( \text{Num} \ (2) \ ) \ ) \ ) \ ( \text{Plus} \\
& \ ( \text{Minus} \ ( \text{Num} \ (0) \ ) \ ) \ ( \text{Num} \ (0) \ ) \ ) \ ( \text{Mult} \ ( \\
& \ ( \text{Num} \ (0) \ ) \ ( \text{Num} \ (3) \ ) \ ) \ ) 
\end{align*}
\]
Idea:

- Generate lots of random values of type `exp`
- For each, see if `compiles_correctly` returns `true`
- If a failing example is found, perform a greedy search for a minimal failing example ("shrinking")
QuickChick compiles_correctly.

Counterexample found after 4 tests and 8 shrinks:

Minus (Num 3) (Num 0)

Fixpoint compile (e : exp) : list instr :=
  match e with
  | Num n => [PUSH n]
  | Plus e1 e2 => compile e1 ++ compile e2 ++ [PLUS]
  | Minus e1 e2 => compile e1 ++ compile e2 ++ [MINUS]
  | Mult e1 e2 => compile e1 ++ compile e2 ++ [MULT]
end.
QuickChick compiles correctly. Counterexample found after 4 tests and 8 shrinks:

Minus (Num 3) (Num 0)

Fixpoint compile (e : exp) : list instr :=
  match e with
  | Num n => [PUSH n]
  | Plus e1 e2 => compile e1 ++ compile e2 ++ [PLUS]
  | Minus e1 e2 => compile e1 ++ compile e2 ++ [MINUS]
  | Mult e1 e2 => compile e1 ++ compile e2 ++ [MULT]
  end.

Fixpoint execute (s : list nat) (p : list instr) : list nat :=
  match (s, p) with
  | (_, nil) => s
  | (_, (PUSH n) :: p') => execute (n :: s) p'
  | (m::n::s', PLUS :: p') => execute ((m+n)::s') p'
  | (m::n::s', MINUS :: p') => execute ((m-n)::s') p'
  | (m::n::s', MULT :: p') => execute ((m*n)::s') p'
  | (_, _ :: p') => execute s p'
  end.

compile leaves the arguments of Minus in the wrong order on the stack!
Beyond Testing...
What else can we do with a specification?

• **Synthesize** programs that satisfy it
• **Build** run-time monitors that check for violations
• **Prove** that an implementation satisfies it
Lemma execute_app : \(\forall p1 \ p2 \ \text{stack}, \ \text{execute} \ \text{stack} \ (p1 \ ++ \ p2) = \text{execute} \ \text{stack} \ p1 \ p2\).

Lemma execute_eval_comm : \(\forall e \ \text{stack}, \ \text{execute} \ \text{stack} \ (\text{compile} \ e) = \text{eval} \ e \ :: \ \text{stack}\).

Theorem compile_correct : \(\forall e, \ \text{assert} \ (\text{compiles_correctly} \ e)\).
Lemma execute_app : forall p1 p2 stack,
    execute stack (p1 ++ p2)
  = execute (execute stack p1) p2.

Lemma execute_eval_comm : forall e stack,
    execute stack (compile e) = eval e :: stack.

Theorem compile_correct : forall e,
    assert (compiles_correctly e).
Lemma execute_app : forall p1 p2 stack,
    execute stack (p1 ++ p2)
  = execute (execute stack p1) p2.

Lemma execute_eval_comm : forall e stack,
    execute stack (compile e) = eval e :: stack.

Theorem compile_correct : forall e,
    assert (compiles_correctly e e).
Lemma execute_app : forall p1 p2 stack, 
   execute stack (p1 ++ p2) 
   = execute (execute stack p1) p2.
Proof.
   induction p1.
   - reflexivity.
   - destruct a.
     + intros. simpl. rewrite IHp1.
       reflexivity.
     + intros. simpl.
       destruct stack as [|x [|y stack']].
       * rewrite IHp1. reflexivity.
       * rewrite IHp1. reflexivity.
       * rewrite IHp1. reflexivity.
     + intros. simpl.
       destruct stack as [|x [|y stack']].
       * rewrite IHp1. reflexivity.
       * rewrite IHp1. reflexivity.
       * rewrite IHp1. reflexivity.
       + intros. simpl.
         destruct stack as [|x [|y stack']].
         * rewrite IHp1. reflexivity.
         * rewrite IHp1. reflexivity.
         * rewrite IHp1. reflexivity.
       + intros. simpl.
         destruct stack as [|x [|y stack']].
         * rewrite IHp1. reflexivity.
         * rewrite IHp1. reflexivity.
         * rewrite IHp1. reflexivity.
Qed.
Lemma execute_app : forall p1 p2 stack,
  execute stack (p1 ++ p2)
  = execute (execute stack p1) p2.
Proof.
  induction p1.
  - reflexivity.
  - destruct a.
    + intros. simpl. rewrite IHp1.
      reflexivity.
    + intros. simpl.
      destruct stack as [|x [|y stack']].
      * rewrite IHp1. reflexivity.
      * rewrite IHp1. reflexivity.
      * rewrite IHp1. reflexivity.
      * intros. simpl.
        destruct stack as [|x [|y stack']].
        * rewrite IHp1. reflexivity.
        * rewrite IHp1. reflexivity.
        * rewrite IHp1. reflexivity.
    + intros. simpl.
      destruct stack as [|x [|y stack']].
      * rewrite IHp1. reflexivity.
      * rewrite IHp1. reflexivity.
      * rewrite IHp1. reflexivity.
  Qed.

Lemma execute_app : forall p1 p2 stack,
  execute stack (p1 ++ p2)
  = execute (execute stack p1) p2.
Proof.
  induction p1.
  - reflexivity.
  - destruct a; simpl; intros;
    destruct stack as [|x [|y stack']].
    try rewrite IHp1;
    reflexivity.
Qed.

Simple automation

No automation
Lemma execute_app : forall p1 p2 stack,
execute stack (p1 ++ p2)
= execute (execute stack p1) p2.
Proof.
  induction p1.
  - reflexivity.
  - destruct a.
    + intros. simpl. rewrite IHp1. reflexivity.
    + intros. simpl.
      destruct stack as [|x [|y stack']].
      * rewrite IHp1. reflexivity.
      * rewrite IHp1. reflexivity.
      * rewrite IHp1. reflexivity.
    + intros. simpl.
      destruct stack as [|x [|y stack']].
      * rewrite IHp1. reflexivity.
      * rewrite IHp1. reflexivity.
      * rewrite IHp1. reflexivity.
  Qed.

Simple automation

Lemma execute_app : forall p1 p2 stack,
execute stack (p1 ++ p2)
= execute (execute stack p1) p2.
Proof.
  induction p1.
  - reflexivity.
  - destruct a; simpl; intros;
    destruct stack as [|x [|y stack']].
    try rewrite IHp1; reflexivity.
Qed.

Chlipala automation

Lemma execute_app : forall p1 p2 stack,
execute stack (p1 ++ p2)
= execute (execute stack p1) p2.
Proof.
  induction p1;
  try (destruct a);
  try (destruct stack
      as [|x [|y stack']]);
  crush.
Qed.
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. …

**logical specification**

\[ \forall (e : \text{exp}), \quad \text{eq} \ (\text{execute} \ \[] \ (\text{compile} \ e)) \ [\text{eval} \ e]. \]

**executable specification**

Definition \( \text{compiles\_correctly} \ (e : \text{exp}) := \text{eq} \ (\text{execute} \ \[] \ (\text{compile} \ e)) \ [\text{eval} \ e]. \)

**unit tests**

Example e3:
assert (compiles_correctly (Plus (Num 2) (Num 2))).
Example e4:
assert (compiles_correctly (Plus (Num 5) (Num 3))).
...

**informal specification**

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. …

Fixpoint compile (e : exp) : list instr :=
match e with
| Num n => [PUSH n]
| Plus e1 e2 => compile e1 ++ compile e2 ++ [PLUS]
| Minus e1 e2 => compile e1 ++ compile e2 ++ [MINUS]
| Mult e1 e2 => compile e1 ++ compile e2 ++ [MULT]
end.
What about “two-sided”?
Theorem optimize_correct :
  forall e, 
  eval (optimize e) 
  = eval e.
nice story

does it scale??
Some recent “deep specifications”

- Rigorously tested specifications of existing real-world artifacts
- Formally verified specifications of new artifacts
“live” = “exhaustively tested”...
• Full-scale formal specifications of a range of critical interfaces
  • X86 instruction set
  • TCP protocol suite
  • Posix file system interface

• Weak memory consistency models for x86, ARM, PowerPC
• ISO C / C++ concurrency
• Elf loader format
• C language (Cerberus – also see Krebbers, K semantics, …)

“Rigorous Engineering of Mainstream Systems”

http://rems.io
• Engineers at Quviq built an executable specification based on the 3000-page AutoSAR standard for automotive software components

• QuickCheck-based testing found >200 faults in AutoSAR Basic Software, including >100 inconsistencies in the standard
“live” = “verified”...
- Accepts most of ISO C 99
- Produces machine code for PowerPC, ARM, and IA32 (x86 32-bit) architectures
- 90% of the performance of GCC (v4, opt. level 1)
- Fully verified (at the source-code level)
• 50,000 lines of Coq
  • 8k code (≈ 40k of C or Java)
  • 42k specification and proof
Verification really works!

Regehr’s Csmith project used random testing to assess all popular C compilers, and reported:

```
The striking thing about our CompCert results is that the middle-end bugs we found in all other compilers are absent. As of early 2011, the under-development version of CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying: we have devoted about six CPU-years to the task. The apparent unbreakability of CompCert supports a strong argument that developing compiler optimizations within a proof framework, where safety checks are explicit and machine-checked, has tangible benefits for compiler users.”
```
• Verified compiler from a substantial subset of Standard ML to x86-64 machine code (ARM, MIPS, and RISC-V are anticipated)

• Bootstrapped!
  • The compiler itself is implemented in CakeML, so its executable is guaranteed to implement the compilation algorithm described by its source code

• Correctness proofs use validated ISA models for machine code

• Goal is to implement a proof assistant in CakeML and use it to verify CakeML’s own correctness proof
  • TinyTCB!!
• Real-world operating-system kernel with an end-to-end proof of implementation correctness and security enforcement

• Verified down to machine code
• **Ironclad Apps**: verifying the security of a complete software stack

User can securely transmit her data to a remote machine with the guarantee that every instruction executed on that machine adheres to a formal abstract specification of the app’s behavior.

• **IronFleet**: verifying safety and liveness of distributed systems
Ongoing project aiming to build and deploy a verified HTTPS stack

drop-in replacement for the HTTPS library in mainstream web browsers, servers, etc.
Certified OS Kernels
Clean-slate design with end-to-end guarantees on extensibility, security, and resilience. Without Zero-Day Kernel Vulnerabilities.

Layered Approach
Divides a complex system into multiple certified abstraction layers, which are deep specifications of their underlying implementations.

Languages and Tools
New formal methods, languages, compilers and other tools for developing, checking, and automating specs and proofs.
• Coq framework for implementing, specifying, verifying, and compiling Bluespec-style hardware components.

• E.g., a RISC-V implementation (w 4-stage pipeline), fully verified down to RTL
Verdi

- Framework for implementing and formally verifying distributed systems
  - E.g. verified implementation of the Raft distributed consensus protocol

- **Verified system transformers** encapsulate common fault tolerance techniques
  - Developers verify an application in an idealized fault model, then apply a VST to obtain an application with analogous properties in a more adversarial environment
• The Vellvm project has built a formal specification of the intermediate representation used by the popular LLVM compiler.

• This spec has been used to build verified compiler transformations that can be plugged into LLVM. Their performance is competitive with unverified transformations.

• The specification has been validated against the LLVM test suite.
Certified compiler from Coq to C
  • and then, via CompCert, to assembly
  • (in progress)
Haskell CoreSpec is an ongoing effort to formally specify the core intermediate language of the GHC compiler and verify key compiler passes.
• C verification framework based on higher-order separation logic in Coq
• Verified implementations of OpenSSL-HMAC and SHA-256
• working on additional crypto primitives (HMAC-based Deterministic Random Byte Generation, AES), parts of TweetNaCL
Verified Textbooks!

Software Foundations
Benjamin C. Pierce
Arthur Azevedo de Amorim
Chris Casinghino
Marco Gaboardi
Michael Greenberg
Cătălin Hrițcu
Vilhelm Sjöberg
Brent Yorgey

Concrete Semantics
Tobias Nipkow · Gerwin Klein
With Isabelle/HOL

Certified Programming with Dependent Types
Adam Chlipala
A Pragmatic Introduction to the Coq Proof Assistant
And more!

• Bedrock system
• Ur/Web compiler
• CompCert TSO compiler
• CompCert static analysis tools
• Jitk and Data6 verified file systems
• Verified Fscq from MIT
• …
Why now?

Urgent need for increased confidence
+
Diminishing value of “paper proofs”
+
Progress on enabling technologies
Enabling Technologies

Better theory

• Operational semantics, etc.
• Domain-specific logics
  • E.g. Separation logic
Enabling Technologies

Better tools

- **Proof assistants**
  - Coq, Isabelle, ACL2, Twelf, HOL-light, …
- **Testing tools and methodologies**
  - QuickCheck, QuickChick, …
- **DSLs for writing specifications**
  - OTT, Lem, Redex, …
- **Languages with integrated specifications**
  - Dafny, Boogie, JML, F*, Liquid Types, Verilog PSL, Dependent Haskell, …
Enabling Technologies

Faster hardware also helps!
What next?
Goal:

Move from **one-off success stories** to **sustainable engineering practice at industrially relevant scale**
Lessons from CompCert

- OS client interface: CertiKOS hypervisor kernel
- Program Logic: Verifiable C System
- C language
- C language
- CompCert Compiler
- PowerPC ISA
- PowerPC ISA
- IBM's CPU
- Transistors
OS client interface
- CertiKOS hypervisor kernel
- C language

Program Logic
- Verifiable C System
- C language

C language

CompCert Compiler

PowerPC ISA

PowerPC ISA

IBM’s CPU

Transistors
## Research threads

<table>
<thead>
<tr>
<th>Tool</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fiat/Kami</td>
<td>RISC-V implementation, verified down to RTL</td>
</tr>
<tr>
<td>CertiCoq</td>
<td>Verified Gallina-to-CompCert-C compiler</td>
</tr>
<tr>
<td>CertiKOS</td>
<td>Verified OS / hypervisor</td>
</tr>
<tr>
<td>VST</td>
<td>Verified Software Toolchain for C</td>
</tr>
<tr>
<td>Vellvm</td>
<td>Verified LLVM</td>
</tr>
<tr>
<td>Core Haskell</td>
<td>Formal model of GHC core</td>
</tr>
<tr>
<td>QuickChick</td>
<td>Specification-based random testing in Coq</td>
</tr>
</tbody>
</table>
Course design

• Undergrad
  • Drop-in replacements for standard compiler and OS courses
  • Built around pedagogical versions of Vellvm and CertiKOS
  • Students will learn to read and interact with specifications (but not proofs)
  • Code connected to specifications via random testing

• Grad
  • New course on formally specifying and verifying systems software and hardware
... and further volumes to come!
Join us!

Summer schools
(July 13-28, 2017,
in Philadelphia)

Technical workshops
(one last spring; several more to follow)

PhD and postdoc positions

Visitors program

Visit deepspec.org to see what’s happening
and join our mailing list

Thank you!
(any (more) questions?)