Proving the Equivalence of Two Modules

COS 326
Andrew Appel
Princeton University

slides copyright 2020 David Walker and Andrew Appel
permission granted to reuse these slides for non-commercial educational purposes
When explaining our modules to clients, we would like to explain them in terms of abstract values – sets, not the lists (or maybe trees) that implement them.

From a client’s perspective, operations act on abstract values.

Signature comments, specifications, preconditions and post-conditions should be defined in terms of those abstract values.

How are these abstract values connected to the implementation?
Abstraction

user’s view:

sets of integers

\{1, 2, 3\} \quad \{4, 5\} \\
\{\} \\

implementation view:

lists of integers

\[1; 1; 2; 3; 2; 3\] \quad [ ] \quad [4, 5] \quad [4, 5, 5] \\
\[1; 2; 3\] \quad [5, 4]
user’s view:

sets of integers

{1, 2, 3}  
{ }        
{4, 5}    

implementation view:

lists of integers

[1; 1; 2; 3; 2; 3]  
[1; 2; 3]        
[ ]              
[4, 5]           
[5, 4]           
[4, 5, 5]        

there’s a relationship here, of course!

we are trying to implement the abstraction
user’s view:
sets of integers
{1, 2, 3} { } {4, 5}

implementation view:
lists of integers
[1; 1; 2; 3; 2; 3] [1; 2; 3] [4, 5] [4, 5, 5]

this relationship is a function: it converts concrete values to abstract ones

function called “the abstraction function”
Abstraction

user’s view:
sets of integers
{1, 2, 3}  {4, 5}
{}        

implementation view:
lists of integers
[1; 1; 2; 3; 2; 3] [ ] [4, 5] [4, 5, 5]
[1; 2; 3] inv(x): no duplicates [5, 4]

A Representation Invariant cuts down the domain of the abstraction function
Specifications

user’s view:

{1, 2} \rightarrow \text{add 3} \rightarrow \{1, 2, 3\}

a specification tells us what operations on abstract values do

implementation view:
A specification tells us what operations on abstract values do. In this example:

User's view:

- **inv(x)**
- **add 3**
- Input: \{1, 2\}
- Output: \{1, 2, 3\}

Implementation view:

- **inv(x)**
- **add 3**
- Input: [1; 2]
- Output: [1; 3]
a specification tells us what operations on abstract values do

user’s view:

implementation view:

inv(x)
In general, related arguments are mapped to related results. A specification tells us what operations on abstract values do.
Implementation does not correspond to the correct abstract value!
Specifications

Implementation must correspond no matter which concrete value you start with.

User's view:

\[
\{1, 2\} \xrightarrow{\text{add 3}} \{1, 2, 3\}
\]

Implementation view:

\[
[1; 2] \xrightarrow{\text{add 3}} [3; 1; 2] \quad [2; 1] \xrightarrow{\text{add 3}} [3; 2; 1]
\]

inv(x)
A more general view

to prove:
for all \( c_1 : t \), if \( \text{inv}(c_1) \) then
\[
\text{f}_{\text{abs}}(\text{abs } c_1) = \text{abs } (\text{f}_{\text{con}} c_1)
\]

\text{abstract then apply the abstract op \( \equiv \) apply concrete op then abstract}
Another Viewpoint

A specification is really just another implementation (in this viewpoint)
— but it’s often simpler (“more abstract”)

We can use similar ideas to compare any two implementations of the same signature. Just come up with a relation between corresponding values of abstract type.

We ask: Do operations like \( f \) take related arguments to related results?
What is a specification?

It is really just another implementation
   – but it’s often simpler (“more abstract”)

We can use similar ideas to compare *any two implementations of the same signature*. Just come up with a relation between corresponding values of abstract type.
One Signature, Two Implementations

module type S = 
  sig
    type t
    val zero : t
    val bump : t -> t
    val reveal : t -> int
  end

module M1 : S = 
  struct
    type t = int
    let zero = 0
    let bump n = n + 1
    let reveal n = n
  end

module M2 : S = 
  struct
    type t = int
    let zero = 2
    let bump n = n + 2
    let reveal n = n/2 - 1
  end

Consider a client that might use the module:

let x1 = M1.bump (M1.bump (M1.zero))
let x2 = M2.bump (M2.bump (M2.zero))

What is the relationship?

is_related (x1, x2) = x1 == x2/2 - 1

And it persists: Any sequence of operations produces related results from M1 and M2!
Recall: A representation invariant is a property that holds for all values of abs. type:

- if \( M.v \) has abstract type \( t \),
  - we want \( \text{inv}(M.v) \) to be true

Inter-module relations are a lot like representation invariants!

- if \( M1.v \) and \( M2.v \) have abstract type \( t \),
  - we want \( \text{is}_\text{related}(M1.v, M2.v) \) to be true

It’s just a relation between two modules instead of one
Relations may imply the Rep Inv

When defining our relation, we will often do so in a way that implies the representation invariant.

ie: a value in M1 will not be related to any value in M2 unless it satisfies the representation invariant.
module type S =
    sig
        type t
        val zero : t
        val bump : t -> t
        val reveal : t -> int
    end

module M1 : S =
    struct
        type t = int
        let zero = 0
        let bump n = n + 1
        let reveal n = n
    end

module M2 : S =
    struct
        type t = int
        let zero = 2
        let bump n = n + 2
        let reveal n = n/2 - 1
    end

is_related (x1, x2) =
    (x1 == x2/2 - 1) && x1 >= 0 && even x2

is_related (x1, x2) implies x1 >= 0

is_related (x1, x2) implies even x2 && x2 > 0
module type S =
  sig
    type t
    val zero : t
    val bump : t -> t
    val reveal : t -> int
  end

module M1 : S =
  struct
    type t = int
    let zero = 0
    let bump n = n + 1
    let reveal n = n
  end

module M2 : S =
  struct
    type t = int
    let zero = 2
    let bump n = n + 2
    let reveal n = n/2 - 1
  end

is_related (x1, x2) =
  (x1 == x2/2 - 1)

But For Now:
Consider zero, which has abstract type t.

Must prove: \( \text{is\_related}(\text{M1.zero}, \text{M2.zero}) \)

Equivalent to proving: \( \text{M1.zero} == \text{M2.zero}/2 - 1 \)

Proof:

\[
\begin{align*}
\text{M1.zero} &== 0 \quad \text{(substitution)} \\
&== 2/2 - 1 \quad \text{(math)} \\
&== \text{M2.zero}/2 - 1 \quad \text{(substitution)}
\end{align*}
\]
Consider bump, which has abstract type \( t \rightarrow t \).

Must prove for all \( v1: \text{int}, v2: \text{int} \)
if \( \text{is\_related}(v1, v2) \) then \( \text{is\_related}(\text{M1}.\text{bump} \ v1, \text{M2}.\text{bump} \ v2) \)

Proof:
(1) Assume \( \text{is\_related}(v1, v2) \).
(2) \( v1 = v2/2 - 1 \) (by def)

Next, prove:
(\( M2.\text{bump} \ v2 \))/2 - 1 == \( M1.\text{bump} \ v1 \)

\( \text{is\_related}(x1, x2) = x1 \ == \ x2/2 - 1 \)
Consider reveal, which has abstract type $t \rightarrow \text{int}$.

Must prove for all $v_1 : \text{int}$, $v_2 : \text{int}$ if $\text{is\_related}(v_1, v_2)$ then $\text{M1}$.reveal $v_1 = \text{M2}$.reveal $v_2$

Proof:
(1) Assume $\text{is\_related}(v_1, v_2)$.
(2) $v_1 = v_2/2 - 1$ (by def)

Next, prove:
$\text{M2}$.reveal $v_2 = \text{M1}$.reveal $v_1$
Summary of Proof Technique

To prove M1 == M2 relative to signature S,

– Start by defining a relation “is_related”:
  • is_related (v1, v2) should hold for values with abstract type t when v1 comes from module M1 and v2 comes from module M2

– Extend “is_related” to types other than just abstract t. For example:
  • if v1, v2 have type int, then they must be exactly the same
    – ie, we must prove: v1 == v2
  • if v1, v2 have type s1 -> s2 then we consider arg1, arg2 such that:
    – if is_related(arg1, arg2) at type s1 then we prove
    – is_related(v1 arg1, v2 arg2) at type s2
  • if v1, v2 have type s option then we must prove:
    – v1 == None and v2 == None, or
    – v1 == Some u1 and v2 == Some u2 and is_related(u1, u2) at type s

– For each val v:s in S, prove is_related(M1.v, M2.v) at type s
MODULES WITH DIFFERENT IMPLEMENTATION TYPES
One Signature, Two Implementations

module type S =
  sig
    type t
    val zero : t
    val bump : t -> t
    val reveal : t -> int
  end

module M1 : S =
  struct
    type t = int
    let zero = 0
    let bump n = n + 1
    let reveal n = n
  end

module M2 : S =
  struct
    type t = int
    let zero = 2
    let bump n = n + 2
    let reveal n = n/2 - 1
  end
Different representation types

module type S =
  sig
    type t
    val zero : t
    val bump : t -> t
    val reveal : t -> int
  end

module M1 : S =
  struct
    type t = int
    let zero = 0
    let bump x = x + 1
    let reveal x = x
  end

module M2 : S =
  struct
    type t = Zero | S of t
    let zero = Zero
    let bump x = S x
    let rec reveal x =
      match x with
      | Zero -> 0
      | S x -> 1 + reveal x
  end
Two modules with abstract type t will be declared equivalent if:

- one can *define a relation between corresponding values of type t*
- one can show that *the relation is preserved by all operations*

If we do indeed show the relation is “preserved” by operations of the module (an idea that depends crucially on the *signature* of the module) then *no client will ever be able to tell the difference between the two modules even though their data structures are implemented by completely different types!*

module type S =
  sig
    type t
    val zero : t
    val bump : t -> t
    val reveal : t -> int
  end

module M1 : S =
  struct
    type t = int
    let zero = 0
    let bump x = x + 1
    let reveal x = x
  end

module M2 : S =
  struct
    type t = Zero | S of t
    let zero = Zero
    let bump x = S x
    let rec reveal x =
      match x with
        | Zero -> 0
        | S x -> 1 + reveal x
  end

is_related (x1, x2) =
  x1 == M2.reveal x2
John Reynolds, 1935-2013
Discovered the polymorphic lambda calculus (first polymorphic type system).
Developed **Relational Parametricity**: A technique for proving the equivalence of modules.
Abstraction functions define the relationship between a concrete implementation and the abstract view of the client

- We should prove concrete operations implement abstract ones described to our customers/clients

We prove any two modules are equivalent by

- Defining a relation between values of the modules with abstract type
- We get to assume the relation holds on inputs; prove it on outputs

Rep invariants and “is_related” predicates are called logical relations
Software Verification
(preview of COS 510 “Programming Languages”)

Andrew W. Appel

Princeton University
Formal reasoning about programs

- Functional Programming
- Proving your (functional) programs correct
Formal reasoning about programs and programming languages

- Intro to Formal Logic
- Functional Programming
- Specification of programming languages
- Imperative Programming
- Type systems
- Proving your type system sound
- Hoare Logic
- Proving Hoare Logic sound
- Proving your (functional) programs correct
- Proving your (imperative) programs correct
Which of these things do we do

By machine?

- intro to Formal Logic
- Functional Programming
- Proving your (functional) programs correct

With pencil+paper?

- Specification of programming languages
- Type systems
- Proving your type system sound
- Imperative Programming
- Hoare Logic
- Proving Hoare Logic sound
- Proving your (imperative) programs correct
We can do all of these

By machine!

pencil+paper? Really?

- Intro to Formal Logic
- Functional Programming
- Proving your (functional) programs correct

- Specification of programming languages
- Type systems
- Proving your type system sound

- Imperative Programming
- Hoare Logic
- Proving Hoare Logic sound

- Proving your (imperative) programs correct
COS 510: Machine-checked, formal reasoning about programs and programming languages
EXAMPLE: LENGTH, APP
Require Import List.

Fixpoint length {A} (xs: list A) : nat :=
match xs with
| nil => 0
| x::xs' => 1 + length xs'
end.

Eval compute in length (1::2::3::4::nil).

Fixpoint app {A} (xs ys: list A) : list A :=
match xs with
| nil => ys
| x::xs' => x :: app xs' ys
end.

Eval compute in app (1::2::3::nil) (7::8::nil).

Eval compute in length (app (1::2::3::nil) (7::8::nil)).
Require Import List.

**Fixpoint** length {A} (xs: list A) : nat :=
match xs with
| nil => 0
| x::xs' => 1 + length xs'
end.

**Eval** compute in length (1::2::3::4::nil).

**Fixpoint** app {A} (xs ys: list A) : list A :=
match xs with
| nil => ys
| x::xs' => x :: app xs' ys
end.

**Eval** compute in app (1::2::3::nil) (7::8::nil).

**Eval** compute in length (app (1::2::3::nil) (7::8::nil)).
Require Import List.

Fixpoint length {A} (xs: list A) : nat :=
  match xs with
  | nil => 0
  | x::xs' => 1 + length xs'
end.

Eval compute in length (1::2::3::4::nil).

Fixpoint app {A} (xs ys: list A) : list A :=
  match xs with
  | nil => ys
  | x::xs' => x :: app xs' ys
end.

Eval compute in app (1::2::3::nil) (7::8::nil).

Eval compute in length (app (1::2::3::nil) (7::8::nil)).
Require Import List.

Fixpoint length {A} (xs: list A) : nat :=
  match xs with
  | nil => 0
  | x::xs' => 1 + length xs'
end.

Eval compute in length (1::2::3::4::nil).

Fixpoint app {A} (xs ys: list A) : list A :=
  match xs with
  | nil => ys
  | x::xs' => x :: app xs' ys
end.

Eval compute in app (1::2::3::nil) (7::8::nil).

Eval compute in length (app (1::2::3::nil) (7::8::nil)).

= 1 :: 2 :: 3 :: 7 :: 8 :: nil
: list nat
Require Import List.

Fixpoint length {A} (xs : list A) : nat :=
  match xs with
  | nil => 0
  | x::xs' => 1 + length xs'
  end.

Eval compute in length (1::2::3::4::nil).

Fixpoint app {A} (xs ys : list A) : list A :=
  match xs with
  | nil => ys
  | x::xs' => x :: app xs' ys
  end.

Eval compute in app (1::2::3::nil) (7::8::nil).

Eval compute in length (app (1::2::3::nil) (7::8::nil)).
**Theorem** app_length: forall {A} (xs ys: list A),
   length (app xs ys) = length xs + length ys.

**Proof.**

**Qed.**
Theorem app_length: forall {A} (xs ys: list A),
    length (app xs ys) = length xs + length ys.

Proof.
Qed.
Theorem app_length: forall {A} (xs ys : list A),
  length (app xs ys) = length xs + length ys.
Proof.
intros.
Qed.
Theorem app_length: forall {A} (xs ys: list A),
  length (app xs ys) = length xs + length ys.

Proof.
intros.
induction xs.
- (* base case *)
simpl.
  reflexivity.
- (* inductive case *)
simpl.
  reflexivity.
Qed.

2 subgoals
A : Type
ys : list A

length (app nil ys) = length nil + length ys

length (app (a :: xs) ys) =
length (a :: xs) + length ys
**Theorem** `app_length`:

forall `{A}` (xs : list A), ys : list A

\[ \text{length } (\text{app } xs \ ys) = \text{length } xs + \text{length } ys. \]

**Proof.**

intros.

induction `xs`.

(* base case *)

simpl.

reflexivity.

(* inductive case *)

simpl.

reflexivity.

Qed.
**Theorem** app_length: forall {A} (xs y: list A), length (app xs y) = length xs + length y.

**Proof.**

intros.
induction xs.
- (* base case *)
  simpl.
  reflexivity.
- (* inductive case *)
  simpl.
  reflexivity.
Qed.
Theorem app_length: forall {A} (xs ys : list A),
  length (app xs ys) = length xs + length ys.

Proof.
  intros.
  induction xs.
  (* base case *)
  simpl.
  reflexivity.
  (* inductive case *)
  simpl.
  reflexivity.
Qed.
**Theorem** `app_length`: for all `{A} (xs ys : list A), 
  length (app xs ys) = length xs + length ys.

**Proof**

- intros.
- induction xs.
- (* base case *)
  simpl.
  reflexivity.
- (* inductive case *)
  simpl.
  reflexivity.

`Qed`. 

1 subgoal 
A : Type 
ys : list A 

(1/1)

length ys = length ys
**Theorem** app_length: forall {A} (xs ys: list A),
  length (app xs ys) = length xs + length ys.

**Proof.**

intro xs.
induction xs.
- (* base case *)
  simpl.
  reflexivity.
- (* inductive case *)
  simpl.
  reflexivity.

Qed.
Theorem app_length: forall {A} (xs ys: list A),
  length (app xs ys) = length xs + length ys.

Proof.
  intros.
  induction xs.
  (* base case *)
  simpl.
  reflexivity.
  (* inductive case *)
  simpl.
  reflexivity.
Qed.
Theorem \textit{app\_length}: \texttt{forall \{A\} (xs ys: list A),
length (app xs ys) = length xs + length ys.}

\textbf{Proof.}

\texttt{intros.}

\texttt{induction xs.}

- (* base case *)
  \texttt{simpl.}
  \texttt{reflexivity.}

- (* inductive case *)
  \texttt{simpl.}
  \texttt{reflexivity.}

\texttt{Qed.}
**Theorem** app_length: forall {A} (xs ys: list A),
length (app xs ys) = length xs + length ys.

**Proof.**

- intros.
- induction xs.
- (* base case *)
  - simpl.
  - reflexivity.
- (* inductive case *)
  - simpl.
  - reflexivity.

Qed.

1 subgoal
A : Type
a : A
xs, ys : list A
IHxs : length (app xs ys) =
  length xs + length ys

---------------------------(1/1)
S (length (app xs ys)) =
S [length xs + length ys]

In environment
A : Type
a : A
xs, ys : list A
IHxs : length (app xs ys) =
  length xs + length ys
Unable to unify "S (length xs + length ys)"
with "S (length (app xs ys))".

Ready, proving app_length
**Theorem** `app_length`: forall `{A}` (xs ys: list A),
\[ \text{length} \ (\text{app} \ \text{xs} \ \text{ys}) = \text{length} \ \text{xs} + \text{length} \ \text{ys} \]

**Proof**

- intros.
- induction `xs`.
- `(* base case *)` simpl.
- reflexivity.
- `(* inductive case *)` simpl.
- rewrite `IHxs`.
- reflexivity.

 Qed.
Theorem app_length: forall {A} (xs ys: list A),
  length (app xs ys) = length xs + length ys.

Proof,
intros.
induction xs.
- (* base case *)
  simpl.
  reflexivity.
- (* inductive case *)
  simpl.
  rewrite IHxs.
  reflexivity.
Qed.

No more subgoals.
Theorem app_length: forall {A} (xs ys: list A),
length (app xs ys) = length xs + length ys.

Proof.
intros.
induction xs.
- (* base case *)
simpl.
reflexivity.
- (* inductive case *)
simpl.
rewrite IHxs.
reflexivity.
Qed.
Theorem app_assoc: \forall A \ (xs \ ys \ zs : list A),
app xs (app ys zs) = app (app xs ys) zs.

Proof.
intros.
induction xs.
- (* base case *)
simpl.
reflexivity.
- (* inductive case *)
simpl.
rewrite IHxs.
reflexivity.
Qed.
**Theorem** `app_assoc`: for all `{A} (xs ys zs : list A),
  `app xs (app ys zs) = app (app xs ys) zs`.

**Proof.**
intros.
induction `xs`.
- (* base case *)
simpl.
  reflexivity.
- (* inductive case *)
simpl.
  rewrite `IHxs`.
  reflexivity.
Qed.
Applications of Formal Methods
Attacking a web server

URLs
Input in web forms
Crypto keys for SSL
etc.

Client PC
Web Server

for(i=0;p[i];i++)
search[i]=p[i];

this is a really long search term that overflows a buffer
Attacking a web browser

HTML keywords
Images
Image names
URLs
etc.

for(i=0;p[i];i++)
gif[i]=p[i];

Client PC

Web Server @ badguy.com

www.badguy.com

Earn $$$ Thousands working at home!
Attacking everything in sight

E-mail client
PDF viewer
Web browser
Operating-system kernel
TCP/IP stack

*Any* application that ever sees input directly from the outside
Solution: implement the outward-facing parts of software without any bugs!

E-mail client
PDF viewer
Web browser
Operating-system kernel
TCP/IP stack

Any application that ever sees input directly from the outside
In recent years, great progress in 

- Proved-correct optimizing C compiler *(France)*
- Proved-correct ML compiler *(Sweden, Princeton)*
- Proved-correct O.S. kernels *(Australia, New Haven)*
- Proved-correct crypto *(Princeton NJ, Cambridge MA)*
- Proved-correct distributed systems *(Seattle, Israel)*
- Proved-correct web server *(Philadelphia)*
- Proved-correct malloc/free library *(Princeton, Hoboken)*
Automated verification in industry

Amazon
Microsoft
Intel
Facebook
Google
Galois, HRL, Rockwell, Bedrock, ...
• Katherine Ye ’16 verified crypto security
• Naphat Sanguansin ’16 verified crypto impl’n
• Brian McSwiggen ’18 verified B-trees
• Katja Vassilev ’19 verified dead-var elimination
• John Li ’19 verified uncurrying
• Jake Waksbaum ’20 verified Burrows-Wheeler
• Anvay Grover ’20 verified CPS-conversion
Verified Correctness and Security of mbedTLS HMAC-DRBG

Katherine Q. Ye ’16
Princeton U., Carnegie Mellon U.

Matthew Green
Johns Hopkins University

Naphat Sanguansin’16
Princeton University

Lennart Beringer
Princeton University

Adam Petcher
Oracle

Andrew W. Appel ’81
Princeton University

ABSTRACT
We have formalized the functional specification of HMAC-DRBG (NIST 800-90A), and we have proved its cryptographic security—that its output is pseudorandom—using a hybrid game-based proof. We have also proved that the mbedTLS implementation (C program) correctly implements this functional specification. That proof composes with an existing C compiler correctness proof to guarantee, end-to-end, that the machine language program gives strong pseudorandomness. All proofs (hybrid games, C program verification, compiler, and their composition) are machine-checked in the Coq proof assistant. Our proofs are modular: the hybrid game proof holds on any implementation of HMAC-DRBG that satisfies our functional specification. Therefore, our functional specification can serve as a high-assurance reference.
Prerequisites for COS 510

if you’re an undergrad

1. COS 326 Functional Programming

2. Enjoy the proofs in COS 326