Proving the Equivalence of Two Modules

COS 326
Speaker: David Walker
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\} \{4, 5\}
\{} \{}

implementation view:
lists of integers

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

there’s a relationship here, of course!

we are trying to implement the abstraction
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\}
- \{\}\n
Implementation view:
Lists of integers:
- [1; 1; 2; 3; 2; 3]
- [1; 2; 3]
- [4, 5]
- [4, 5, 5]
- [5, 4]

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

user’s view:

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

user’s view:

implementation view:

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

\{1, 2\} \xleftarrow{\text{inv}(x)} [1; 2]
A specification tells us what operations on abstract values do.
Specifications

user’s view:
- {1, 2} \rightarrow \text{add 3} \rightarrow \{1, 2, 3\}

implementation view:
- [1; 2] \rightarrow \text{add 3} \rightarrow [3; 1; 2]

a specification tells us what operations on abstract values do

In general: related arguments are mapped to related results

inv(x)
Bug! Implementation does not correspond to the correct abstract value!
Specifications

user’s view:

implementation

view:

add 3

{1, 2} → {1, 2, 3}

add 3

[1; 2] → [3; 1; 2]

add 3

[2; 1] → [3; 2; 1]

inv(x)

implementation must correspond no matter which concrete value you start with
A more general view

To prove:
For all \( c1 : t \), if \( \text{inv}(c1) \) then \( f_{\text{abs}}(\text{abs } c1) = \text{abs } (f_{\text{con}} c1) \)

Abstract then apply the abstract op == apply concrete op then abstract
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.*
Consider a client that might use the module:

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

What is the relationship?

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

And it persists: Any sequence of operations produces related results from M1 and M2!
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

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 \( \text{is\_related}(M1.v, M2.v) \) 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

rep inv for M1

is_related (x1, x2) implies even x2 && x2 > 0

rep inv for M2
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

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

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

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

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

Proof:

\[
\begin{align*}
\text{M1}.\text{zero} & = 0 \\
& = 2/2 - 1 \\
& = \text{M2}.\text{zero}/2 - 1
\end{align*}
\] (substitution)

\[
\text{is}_\text{related}(x1, x2) = x1 == x2/2 - 1
\]
Consider bump, which has abstract type \( t \rightarrow t \).

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

Proof:
(1) Assume \( \text{is\_related}(v_1, v_2) \).

(2) \( v_1 = v_2/2 - 1 \) (by def)

Next, prove:
\( (\text{M2.bump } v_2)/2 - 1 = \text{M1.bump } v_1 \)

\[
\begin{align*}
\text{(M2.bump v2)/2 - 1} & = (v2 + 2)/2 - 1 \\
& = (v2/2 - 1) + 1 \\
& = v1 + 1 \\
& = \text{M1.bump v1}
\end{align*}
\]

(by 2) (eval, reverse) (math) (eval) (by def)
Consider reveal, which has abstract type \( t \rightarrow \text{int} \).

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

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

Next, prove:
\( \text{M2.reveal } v2 == \text{M1.reveal } v1 \)

\( \text{M2.reveal } v2 
== v2/2 - 1 
== v1 
== \text{M1.reveal } v1 \)

(\( \text{eval}, \text{reverse} \))

\( \text{is\_related} \( x1, x2 \) = x1 == x2/2 - 1 \)
Summary of Proof Technique

To prove $M_1 \equiv M_2$ relative to signature $S$,

- Start by defining a relation “is_related”:
  
  - $\text{is\_related}(v_1, v_2)$ should hold for values with abstract type $t$ when $v_1$ comes from module $M_1$ and $v_2$ comes from module $M_2$

- Extend “is_related” to types other than just abstract $t$. For example:
  
  - if $v_1, v_2$ have type $\text{int}$, then they must be exactly the same
    - ie, we must prove: $v_1 == v_2$
  
  - if $v_1, v_2$ have type $s_1 \to s_2$ then we consider $\text{arg}_1, \text{arg}_2$ such that:
    
    - if $\text{is\_related}(\text{arg}_1, \text{arg}_2)$ at type $s_1$ then we prove
    - $\text{is\_related}(v_1 \text{ arg}_1, v_2 \text{ arg}_2)$ at type $s_2$
  
  - if $v_1, v_2$ have type $s$ option then we must prove:
    
    - $v_1 == \text{None}$ and $v_2 == \text{None}$, or
    - $v_1 == \text{Some } u_1$ and $v_2 == \text{Some } u_2$ and $\text{is\_related}(u_1, u_2)$ at type $s$

- For each $\text{val } v:s$ in $S$, prove $\text{is\_related}(M_1.v, M_2.v)$ at type $s$
MODULES WITH DIFFERENT IMPLEMENTATION 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 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
Reasoning about Module Equivalence

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!
Reasoning about Module Equivalence

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