

Interactive Theorem Proving

Week 11

Cezary Kaliszyk

May 26, 2015



Summary

So far

- Proof Assistants
 - HOL Light, Mizar, Coq, Declarative Proof
- Type Theory, Set Theory
 - λ_{\rightarrow} , λ_P , λ_2 , Curry-Howard

Today

- Program Extraction
- Code Generation
- Datatype Invariants

Verified Programs

Proof-carrying code

Verify properties about an application via formal proof that accompanies the executable code

- Hoare-logic, Separation logic, Dynamic logic, ...
- Model checking, Bi-simulation ...
- Equations and algorithms in a PA

Code-carrying proof

- Intuitionistic proof is equivalent gives rise to executable code
- BHK, Curry-Howard

Two approaches

- Correctness proofs: program \rightarrow proof
- Program extraction: proof \rightarrow program

Hoare logic, Separation logic, Dynamic logic

Imperative programs

- Annotated with pre- and post-conditions
- Usually predicate logic, example:
Variable x is a pointer to a well formed binary tree

Checker for the programs

- Generates proof obligations that need to be verified
- For example the *WHY* tool
 - Supports a common small imperative and functional language
 - Via plugins almost full ANSI C (cauceus, frama-c)
 - Obligations in Coq and ATP formats
- *HIP/SLEEK*
 - Separation logic: reasoning about pointers
- *KEY*-prover
 - Dynamic logic: formulas may include other subformulas (modal logic)

Example

```
data node { int val; node next; }

ll<n> == self=null & n=0 or
  self::node<_,p> * p::ll<n-1> inv n>=0;

sortl<mi,mx> == self::node<mi,null> & mi=mx or
  self::node<mi,p> * p::sortl<m,mx> & mi<=m inv mi<=mx & self!=null;

node append(node x, node y)
  requires x::ll<n>*y::ll<m> & n>0
  ensures res::ll<n+m> & res=x;
  requires x::sortl<s1,b1> * y::sortl<s2,b2> & b1<=s2
  ensures res::sortl<s1,b2>;
  requires x::sortl<s1,b1> & y=null
  ensures res::sortl<s1,b1>; {
    node t = x.next;
    if (t==null) x.next=y;
    else x.next=append(t,y);
  }
  return x;
}
```

Proof extraction

Formal Coq proof

- Type Theory

Functional Program

- Haskell
- OCaml
- (other experimental)

Example (1/3)

Inductive type

```
Inductive bintree : Set :=  
  leaf : nat -> bintree  
| node : bintree -> bintree -> bintree.
```

Recursive function

```
Fixpoint mirror (t : bintree) : bintree :=  
  match t with  
    leaf n => leaf n  
  | node t1 t2 => node (mirror t2) (mirror t1)  
  end.
```

Example (2/3)

Inductive predicate

```
Inductive Mirrored : bintree -> bintree -> Prop :=  
  Mirrored_leaf : forall n : nat, Mirrored (leaf n) (leaf n)  
| Mirrored_node : forall t1 t2 t1' t2' : bintree,  
  Mirrored t1 t1' -> Mirrored t2 t2' ->  
  Mirrored (node t1 t2) (node t2' t1').
```

Correctness

```
Lemma Mirrored_mirror : forall t : bintree, Mirrored t (mirror t).  
  induction t.  
  simpl.  
  apply Mirrored_leaf.  
  simpl.  
  apply Mirrored_node.  
  exact IHt1.  
  exact IHt2.  
Qed.
```


Example (3/3)

Proof of existence

Lemma Mirror :

```
  forall t : bintree, {t' : bintree | Mirrored t t'}.  
induction t.  
exists (leaf n).  
apply Mirrored_leaf.  
elim IHt1.  
intros t1' H1.  
elim IHt2.  
intros t2' H2.  
exists (node t2' t1').  
apply Mirrored_node.  
exact H1.  
exact H2.  
Qed.
```

Extracted program

Extraction Mirror.

```
let rec mirror = function
  | Leaf n -> Leaf n
  | Node (b0, b1) -> Node ((mirror b1), (mirror b0))
```

Extraction summary

specification

```
Inductive Mirrored : bintree -> bintree -> Prop := ...
```

Either:

implementation and correctness

```
Fixpoint mirror (t : bintree) : bintree := ...
```

```
forall t : bintree, Mirrored t (mirror t)
```

Or:

existence proof for specification

```
forall t : bintree, {t' : bintree | Mirrored t t'}
```

Coq Universe Hierarchy

Prop vs Set

- Not all terms are computationally relevant
- Curry-Howard terms do not need to be calculated

Two kinds of existential

- Predicative and impredicative
- In Set and in Prop
- `exists x : A, P x`

```
Inductive ex (A : Set) (P : A -> Prop) : Prop :=  
  ex_intro : forall x : A, P x -> ex P
```

- `{x : A | P x}`

```
Inductive sig (A : Set) (P : A -> Prop) : Set :=  
  exist : forall x : A, P x -> sig P
```

Elimination on Prop is not allowed to create Set!

Code generation

- HOL (Isabelle) specifications
- can be turned into executable programs
 - Haskell, OCaml, SML, Scala
- Shallow embedding
 - Program semantics generated from equational theorems
 - (relation to higher-order rewriting)

Example (1/2)

```
datatype 'a queue = AQueue 'a list 'a list
```

```
definition empty :: 'a queue where  
  empty = AQueue [] []
```

```
primrec enqueue :: 'a => 'a queue => 'a queue where  
  enqueue x (AQueue xs ys) = AQueue (x # xs) ys
```

```
fun dequeue :: 'a queue => 'a option * 'a queue where  
  dequeue (AQueue [] []) = (None, AQueue [] [])  
| dequeue (AQueue xs (y # ys)) = (Some y, AQueue xs ys)  
| dequeue (AQueue xs []) =  
  (case rev xs of y # ys => (Some y, AQueue [] ys))
```

```
export-code empty dequeue enqueue in SML  
module-name Example file examples/example.ML
```

```

structure Example = struct

fun foldl f a [] = a
  | foldl f a (x :: xs) = foldl f (f a x) xs;

fun rev xs = foldl (fn xsa => fn x => x :: xsa) [] xs;

fun list_case f1 f2 (a :: lista) = f2 a lista
  | list_case f1 f2 [] = f1;

datatype 'a queue = AQueue of 'a list * 'a list;

val empty : 'a queue = AQueue ([], [])

fun dequeue (AQueue ([], [])) = (NONE, AQueue ([], []))
  | dequeue (AQueue (xs, y :: ys)) = (SOME y, AQueue (xs, ys))
  | dequeue (AQueue (v :: va, [])) =
    let
      val y :: ys = rev (v :: va);
    in
      (SOME y, AQueue ([], ys))
    end;

fun enqueue x (AQueue (xs, ys)) = AQueue (x :: xs, ys);
end; (*struct Example*)

```

Code generation

- Program Refinement
 - Given various equal implementations
 - Selecting Code Equations with an attribute
- Partial programs
 - Explicit “code-abort”
- Datatype refinement
 - Abstract data-types: sets
 - List? RBT?
 - Datatypes with invariants
 - Quotients: λ -terms
- Evaluation techniques
 - In proofs?
- Example: RBT_Set

Verified programs in the large

- Extraction
 - 2000: FTA
 - every non-constant complex polynomial has a root
 - input $x^2 - 2$
 - program approximating $\sqrt{2}$
 - CompCert: optimizing compiler for C
 - SSR: 4 color theorem, Odd order theorem.
- Separation Logic
 - VCG
 - L4-Verified
- Code Generation
 - CeTA

Summary

Today

- Hoare Logic, Separation Logic, Dynamic Logic
- Program Extraction from Type Theory
- Code Generation
- Verified programs

Next time

- Deep vs Shallow Embedding
- Logical Frameworks