# Lecture 5: Structuring Proofs – Sections and Theories

July 17, 2013

# Instantiation

- Concrete schemes and abstract adversaries
- Reuse existing proofs when realizing cryptographic assumptions? (e.g., one-way trapdoor with RSA)
- Sections hide proof artifacts from final theorems, automatically infer restrictions on adversaries, and generalize theorems with module quantification.
- Cloning avoids user-level code duplication by instantiating abstract types and operators with concrete values, creating module copies with disjoint memories.

# Sections

```
section.
  declare module Adv : A{Prop,Hyp}.

  local module G1 = {
    var count:int (* some state *)
    fun main() = { (* uses A *) }
  }.

  local module Dist : D = { (* uses A *) }

  local equiv Prop_G1:
    [Prop(A).main ~ G1.main: true ==> ={res}].

  local equiv G1_Hyp:
    [G1.main ~ Hyp(D).main: true ==> ={res}].

  lemma &m final: exists (Dist<:D),
    Pr[Prop(A).main() @ m: res] = Pr[Hyp(D).main() @ m: res].
end section.
```

# Sections

- Inside the section, declared modules are independent from modules defined (declared) after it.
- **print** **axiom** final.

  yields (after the section is closed)

  ```
  lemma final (Adv:>A{Prop,Hyp}) &m:
    exists (Dist:>D),
      Pr[Prop(A).main() @ &m: res] = Pr[Hyp(D).main() @ &m:res].
  ```

- Declared modules become parameters to lemmas.
- Local lemmas disappear.
- Local modules disappear and restrictions can be dropped.

# Usages of Sections

- Simplify proofs by inferring adversary restrictions and quantifications.

```
module G(Adv:A) = { ... }.

lemma foo (Adv<:A{G}):
  equiv [ Pr[G(A).f() ~ ... ].
```

```
section.
  declare Adv<:A.
  module G = { ... }.

  lemma foo:
    equiv [ Pr[G.f() ~ ...].
end section.
```

- Generalize theorem statements by hiding proof artifacts.
  - Adversary restrictions,
  - Intermediate games,
  - Intermediate equivs, lemmas and proofs.

# Theories: Generalities

Theories provide an additional layer of generalization:

- declared abstract types yield "polymorphism",
- declared constants and operators yield "universal quantifications",
- in forms that EasyCrypt cannot reason about...
- ... but that allow efficient code reuse.

# Theories: A simple example

```
theory Monoid.
  type t.

  op one: t.
  op ( * ): t -> t -> t.

  axiom mul1m (x : t): one * x = x.
  axiom mulm1 (x : t): x * one = x.
  axiom mulmA (x y z : t): (x * y) * z = x * (y * z).
end Monoid.
```

# Cloning: A simple example

```
require import Int.
clone Monoid as MInt with
  type t <- int,
  op one = 1,
  op ( * ) <- ( * )
  proof * by smt.

print theory MInt. (* yields *)

theory MInt.
  op one: int = 1.

  lemma mul1m (x : int): one * x = x.
  lemma mulm1 (x : int): x * one = x.
  lemma mulmA (x y z : int): (x * y) * z = x * (y * z).
end MInt.
```

# Theories: Cloning and Realization

When cloning, you can:

- define (=) or override ($<-$)
  - abstract types,
  - abstract operators (and constants),
- define *abstract* sub-theories,
  - All declared types and operators are abstract,
  - the theory contains only axioms (no lemmas).
- discharge some (or all) axioms
  - by giving a single proof for all axioms (usually smt),
  - or by giving individual proofs.

# Theories: Cloning modules

- You can clone theories that contain modules.
- You get an exact copy of the module that works in a separate memory space.
- This is useful for code reuse and may have unforeseen applications in proofs.

# Cloning: An example

```
theory ROM.
  module RO = {
    var m: (word,word) map

    fun init(): unit = {
      m = empty;
    }

    fun h(x:word): word = {
      if (!in_dom x m) m.[x] =
        $dword;
      return m.[x];
    }
  }.
end ROM.
```

```
theory ROM'.
  module RO = {
    var m: (word,word) map

    fun init(): unit = {
      m = empty;
    }

    fun h(x:word): word = {
      if (!in_dom x m) m.[x] =
        $dword;
      return m.[x];
    }
  }.
end ROM'.
```

Or just use **clone** ROM **as** ROM'.

# Cloning: Some notes

- Cloning a theory that declares abstract types creates new, distinct abstract types unless you define or override them.
- Cloning is not especially suited to equipping existing theories with new algebraic structures.
- When used carelessly, cloning can cause SMT to give up.

# Summary

Several ways to generalize proofs and theorems:

- Automatically infer module dependencies and adversary restrictions using sections.
- Abstract away the proof and its artifacts and keep only the relevant theorems and hypotheses using local modules and lemmas.
- Perform crypto proofs on abstract modules and operators before instantiating them using theories and cloning.

# Lecture 5.5: Describing Distributions

July 17, 2013

# Distributions

- Discrete sub-distributions (also know as “counting”)
- $\mu$ : $\alpha$ *distr* $\to$ ($\alpha \to$ *bool*) $\to$ real.

**Example**

```
op uniform: bool distr.

axiom uniform_def (p: bool -> bool):
  mu uniform p = (1%r / 2%r) * charfun p true +
                 (1%r / 2%r) * charfun p false.
```

# Derived Operators

## Derived Operators

```
(* Probability of a particular element *)
op mu_x (d:'a distr) (x:'a): real = mu d ((=) x).

(* Total weight of the distribution *)
op weight (d:'a distr): real = mu d (lambda _, true).

(* Support of a distribution *)
op in_supp (x:'a) (d:'a distr): bool = 0%r < mu d x.

(* Point-wise equality *)
pred (==) (d1:'a distr) (d2:'a distr) = mu_x d1 == mu_x d2.
```

# General Axioms on Distributions

## General Axioms

```
(* mu d p is always within the unit interval *)
axiom mu_bounded (d:'a distr) (p:'a -> bool):
  0%r <= mu d p <= 1%r.

(* The probability of the false event is 0 *)
axiom mu_false (d:'a distr): mu d (lambda _, false) = 0%r.

(* Probability of a disjunction of events *)
axiom mu_or (d:'a distr) (p q:'a -> bool):
  mu d (cpOr p q) = mu d p + mu d q - mu d (cpAnd p q).

(* Point-wise equality is equality *)
axiom pw_eq (d d':'a distr):
  d == d' => d = d'.
```

# Some remarks

- These can be used to prove rewriting lemmas on mu that can be used with rewrite Pr.

**Example (rewrite Pr mu_or)**

$$\Pr[\ f, m : A \lor B\ ]$$
$$\downarrow$$
$$\Pr[\ f, m : A\ ] + \Pr[\ f, m : B\ ] - \Pr[\ f, m : A \land B\ ]$$

- It is better, if possible, to define distributions using mu and prove simplification lemmas for mu_x, weight and in_supp.

**Example**

**lemma** in_supp_def (b:bool): in_supp b uniform.

**lemma** uniform_x_def (b:bool): mu_x uniform b = 1%r / 2%r.

**lemma** lossless : weight uniform = 1%r.

# Summary

- A way of *axiomatically* defining discrete distributions.
- Very powerful rewriting results on probability expressions.
- Some sanity checks available.