SMLtoCoq: Automated Generation of Coq Specifications and Proof Obligations from SML Programs with Contracts

Ammar Karkour and Laila Elbeheiry

Advisor: Giselle Reis

I. INTRODUCTION

Motivation

- We're becoming dependent on software that malicious software can compromise our safety and health.
- Testing and non-mathematical techniques are not reliable methods to ensure software safety.
- **Formal verification** is a rigorous mathematical technique to formally prove code correctness.
- Formal verification can ensure the absence of bugs in software
- A common approach to formal verification is using tools known as **theorem-provers** to write theorems and prove them on a computer

Problem

- Programming languages are built for programming; they aim to facilitate writing code
- Proof assistants are built for reasoning; they aim to make facilitate writing proofs
- To prove properties about an implemented program, it is necessary to "reimplement" it in the language of a proof assistant
- This requires familiarity with both the programming language and the proof assistant making it inconvenient for programmers

Contribution Highlights

- **SMLtoCoq**: a tool that *automatically translates SML programs* without side-effects including partial functions, structures, functors, and records *into Coq specifications*
- An extension to the SML language with **function contracts** which are directly translated into Coq theorems
- A Coq version of many parts of SML's basis library
- A **case study** where we translate non-trivial SML code and prove properties on the Coq output

II. INFRASTRUCTURE

A) Overview

SMLtoCoq implements a translation of SML's abstract syntax tree (AST) into Gallina's (Coq's specification language) AST – which is subsequently used to generate Gallina code

B) HaMLet

- An implementation of SML with a front-end compiler
- Comprises of three phases:
  - **Parsing**: Returns the AST of an SML program (if syntax is correct)
  - **Elaboration:** Populates the AST with well-formedness conditions (e.g. non-exhaustive or redundant matches) and type-checking information
  - **Evaluation:** evaluates the program to a value
- We use the AST after the elaboration phase, when annotations contain useful information such as inferred types and exhaustiveness of matches that are crucial for the generated Coq code

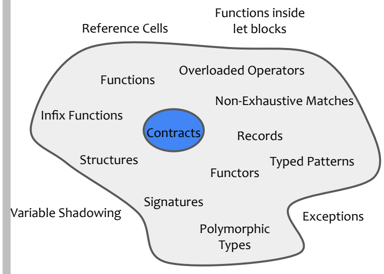
C) Coq/Gallina

- Gallina is Coq's core language for writing specifications.
- Gallina's AST is implemented it as a datatype in our system
- Some constructors were added/eliminated to the datatype to match SML's AST

III. TRANSLATION

Translating from an SML's AST $\mathcal{S}$ into a Gallina's AST $\mathcal{G}$ is defined inductively on $\mathcal{S}$.

The fragment of SML being translated



Functions

- Unless a function is total and structurally decreasing at every recursive call, its translation to Coq isn't trivial
- We use the **Equations** plugin which provides a powerful tool for defining terminating functions via pattern-matching on dependent types
- Our translation includes *pattern matching on inputs, mutually recursive functions, and partial functions*

Contracts

- We add function contracts to HaMLet
- SML contracts get translated to Coq Theorems as follows:

```
(!! f input ==> output;
    REQUIRES: precond;
    ENSURES:  postcond;  !!)
```

⇨

```
Theorem f_Theorem: forall vars,
(f input = output /\ precond =
true) -> postcond = true.
```

IV. EXAMPLES

SML Program

```
datatype treeS = emptyS
               | leafS of string
               | nodeS of treeS * treeS

fun inorder (emptyS: treeS): string list = nil
  | inorder (leafS x) = [x]
  | inorder (nodeS (tL, tR)) =
    (inorder tL) @ (inorder tR)

fun normal' (emptyS: treeS): bool = false
  | normal' (leafS _) = true
  | normal' (nodeS (tL, tR)) =
    normal' tL andalso normal' tR

fun normal (emptyS: treeS): bool = true
  | normal t = normal' t

(* normalize t --> t'
   Satifies:
   - inorder t == inorder (normalize t)
   - normal t' == true
 *)
fun normalize (emptyS: treeS): treeS = emptyS
  | normalize (leafS x) = leafS x
  | normalize (nodeS (tL, tR)) =
    (case (normalize tL, normalize tR)
       of (emptyS, tR') => tR'
        | (tL', emptyS) => tL'
        | (tL', tR')     => nodeS (tL', tR'));
```

Coq Translation

```
Inductive treeS : Type :=
  | emptyS
  | leafS : string → @ treeS
  | nodeS : (treeS * treeS) % type → @ tree

Equations inorder (x1: treeS): @ list string
  inorder emptyS := nil;
  inorder (leafS x) := [x];
  inorder (nodeS (tL, tR)) :=
    ((inorder tL)) @ ((inorder tR)).

Equations normal' (x1: treeS): bool :=
  normal' emptyS := false;
  normal' (leafS _) := true;
  normal' (nodeS (tL, tR)) :=
    (normal' tL) && (normal' tR).

Equations normal (x1: treeS): bool :=
  normal emptyS := true;
  normal t := (normal' t).

Equations normalize (x1: treeS): treeS :=
  normalize emptyS := emptyS;
  normalize (leafS x) := (leafS x);
  normalize (nodeS (tL, tR)) := (
  match ((normalize tL), (normalize tR)
  | (emptyS, tR') ⇒ tR'
  | (tL', emptyS) ⇒ tL'
  | (tL', tR') ⇒ (nodeS (tL', tR'))
  end).
```

SML Program

```
signature KEY =
sig
  type t
  val compare : t * t -> order
end

signature DICT =
sig
  type key
  type 'a entry = key * 'a
  type 'a dict

  val empty: 'a dict
  val lookup: key * 'a dict -> 'a option
  val insert: 'a entry * 'a dict -> 'a dict
end

structure IntKey :> KEY where type t=int  =
struct
    type t = int
    val compare = Int.compare
end
```

Coq Translation

```
Module Type KEY.
  Parameter t : Type.
  Parameter compare : ((t * t) % type → comparison).
End KEY.

Module Type DICT.
  Parameter key : Type.
  Definition entry {_a : Type} := (key * _a) % type.
  Parameter dict : (Type → Type).
  Parameter empty : (forall {_a : Type} , (@ dict _a)).
  Parameter lookup : (forall {_a : Type} ,
    ((key * (@ dict _a)) % type → (@ option _a))).
  Parameter insert : (forall {_a : Type} ,
    (((@ entry _a) * (@ dict _a)) % type → (@ dict _a))).
End DICT.

Module IntKey : KEY with Definition t := Z.
  Definition t := Z.
  Definition compare := Int.compare.
End IntKey.
```

```
type r = { name : string,
           age : int };

fun isBob ({name = "Bob", ...}: r) = true
  | isBob {...} = false;
```

```
Record rid₁ := { rid₁_name : string; rid₁_age : Z }.
Definition r := rid₁.

Equations isBob (x1: rid₁): bool :=
  isBob {| rid₁_age := _; rid₁_name := "Bob" |} := true;
  isBob {| rid₁_age := _; rid₁_name := _ |} := false.
```

V. FUTURE WORK

- Extend SMLtoCoq with: Functions inside let blocks, Non-trivial recursion (without the need for termination proofs), Side-effects
- Prove the correctness of the translation from SML to Gallina
- Simplify automatically generated preconditions

Carnegie Mellon University Qatar

MEETING OF THE Minds Undergraduate Research Symposium

SMLtoCoq's source code can be found at: https://github.com/meta-logic/sml-to-coq/