

# Interactive Theorem Proving and Program Verification

## Lecture 9

Pablo Buiras and Karl Palmskog



Academic Year 2019/20, Period 3–4

Based on slides by Thomas Tuerk

# Part XVI

## Obtaining Verified Programs

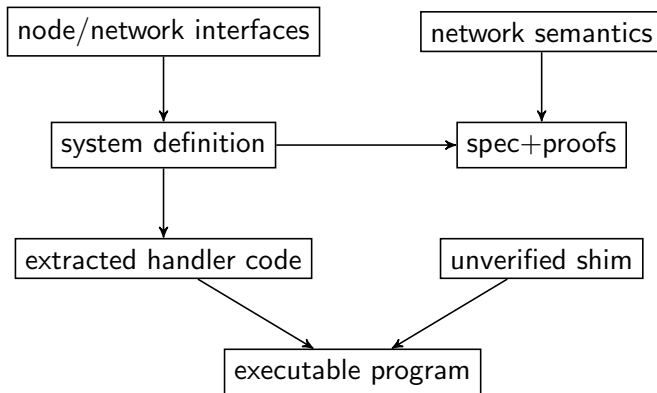


Many options available:

- using code extraction (to SML, OCaml, Haskell, ...)
- reasoning directly about deeply embedded “real” programs using their semantics
- validating compiled binaries
- using a verified compiler
- ...

- what is verified vs. what is trusted?
- TCB originally from security (and adversarial)
- verification TCB typically includes at least
  - ▶ hardware (processor, ISA, ...)
  - ▶ operating system
  - ▶ low-level system libraries
- small TCB is (nearly) always preferable

# Example: Verified Distributed System



- Standard ML extraction in HOL4 (EmitML module)
- OCaml and Haskell in Coq
- Standard ML, Scala, Haskell in Isabelle/HOL

```
open EmitML basis_emitTheory;  
  
val _ = eSML "my_theory" [  
  DATATYPE mydata,  
  DEFN myfun1_def,  
  DEFN myfun2_def,  
  DEFN myfun3_def  
];
```

- extraction is not guaranteed (via machine-checked proofs) to preserve program semantics
- a translation validation approach can establish that generated binary adheres to source language semantics
- used to analyze binaries generated by gcc for the seL4 operating system kernel
- general approach that can be used for other tasks than compilation



- verified compilers can directly produce machine code that is guaranteed be consistent with program meaning
- needed: hardware ISA semantics, source language semantics
- usually constructed as translations between many intermediate languages
- examples: CakeML, CompCert

Any binary produced by a successful evaluation of the compiler function will either

- behave exactly according to the observable behaviour of the source semantics, or
- behave the same as the source up to some point at which it terminates with an out-of-memory error.

Typical assumptions:

- external world doesn't modify allocated memory
- external procedures called by program are well-behaved

# Purely Functional vs. Imperative Code



- purely functional code usually verified by rewriting (lightweight)
- imperative code usually needs Hoare logic verification (heavyweight)
- reasoning about heaps is a lot of work (even with separation logic)
- conjecture (X. Leroy): purely functional programs are the most straightforward path to verified code

# Part XVII

## Introduction to CakeML



- bootstrapping verified compiler for SML-like language, implemented in HOL4
- can generate machine code for MIPS, x86, x86-64, ARMv8, RISC-V
- source and pre-compiled CakeML compatible with HOL4 Kananaskis-13 available online:

<https://github.com/CakeML/cakeml/releases/download/v1009/cake-x64-64.tar.gz>

<https://github.com/CakeML/cakeml/archive/v1009.tar.gz>

# Properties of the CakeML Language



- impure language in the SML family
- eagerly evaluated
- semantics given in functional big-step style
- supports IO and FFI
- all integers are unbounded

# Syntax of CakeML vs. Standard ML



- CakeML has curried Haskell-style constructor syntax
- constructors in CakeML must begin with an uppercase letter
- constructors must be fully applied
- alpha-numeric variable and function names begin with a lowercase letter
- CakeML lacks SML's records, functors, open and (at present) signatures
- CakeML capitalises True, False and Ref

# Semantics of CakeML vs. Standard ML



- right-to-left evaluation order
- CakeML has no equality types
- semantics of equality is different from SML and OCaml
- multi-argument functions



Hello World:

```
print "Hello world!\n";
```

Fibonacci with argument from CLI:

```
fun fiba i j n = if n = 0 then i else fiba j (i+j) (n-1);  
(print (n2s (fiba 0 1 (s2n (hd (CommandLine.arguments()))))));  
  print "\n")  
handle _ => print_err ("usage: " ^ CommandLine.name() ^ " <n>\n");
```

```
fun foldl f e xs =  
  case xs of [] => e  
  | (x::xs) => foldl f (f e x) xs;  
  
fun reverse xs =  
  let  
    fun append xs ys =  
      case xs of [] => ys  
      | (x::xs) => x :: append xs ys;  
    fun rev xs =  
      case xs of [] => xs  
      | (x::xs) => append (rev xs) [x]  
  in  
    rev xs  
  end;
```

# Example Using CakeML as Compiler



Download the pre-compiled CakeML, and put “hello world” program in `hello.cml`:

```
$ make hello.cake  
$ ./hello.cake  
Hello world!
```

Compiler takes 20+ hours to bootstrap!

# Verifying Programs Using CakeML



- imperative programs handled via monads in HOL4
- proof-producing synthesis in HOL4 via **translator**
- post-hoc verification using separation logic

See CakeML journal paper for overview:

<https://cakeml.org/jfp19.pdf>