A Module System for Certified Abstraction Layers

Jérémie Koenig

July 24, 2017

Introduction

Module system

Anatomy of a layer

Conclusion

CertiKOS

As a certified operating system kernel, CertiKOS comes with a computer-checked, mathematical proof of correctness for its assembly code.

We show the following contextual refinement property:

$$\forall P \,.\, [\![P \oplus \texttt{CertiKOS}]\!]_{\texttt{MBoot}} \sqsubseteq [\![P]\!]_{\texttt{TSyscall}}$$

Schematically:

CertiKOS and Compcert

The CertiKOS proof is based on Compcert in several ways:

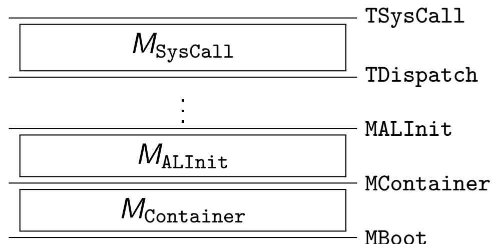
- Compcert x86 assembly serves as the basis for our machine model;
- we use it to compile our C code and proofs;
- we follow similar techniques to construct our own proof.

If the layer interfaces are seen as *languages*, adding the code of `CertiKOS` to a program can be seen as a special case of *certified compilation*.

Decomposition into layers

Because contextual refinement is transitive, we can split `CertiKOS` into *layers*:

$$\texttt{CertiKOS} = M_{\texttt{Container}} \oplus M_{\texttt{ALInit}} \oplus \ldots \oplus M_{\texttt{SysCall}}$$



At each layer, a *module* M implements an *overlay interface* L_2 in terms of an *underlay interface* L_1.

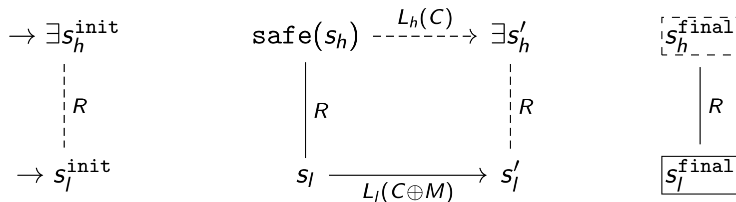
Layer interfaces

As a language, each layer interface is a specialized version of the Compcert formalization of x86 `Asm`, where:

- The memory states contain contain an extra *abstract data* component (each layer can specify its own type);
- A number of *primitives* are made available through Compcert's external functions interface, which can manipulate this extra component.

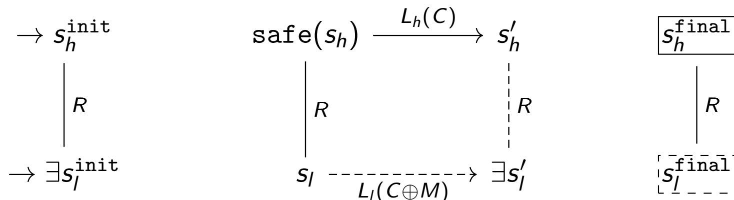
Refinement

Following Compcert, we establish our refinement theorems using *backward simulations*. That is, we can find a relation R such that:



Forward simulations

Backward simulations are often established by relying on the properties of the languages to “flip” a *forward simulation*:



Proof structure

In our case, most steps will execute in the same way in the overlay and underlay. But whenever the context invokes an abstract primitive in the overlay, we need to show that:

- the layer implementation M contains a function which implements the overlay primitive according to R, or
- the underlay contains its own corresponding abstract primitive, which is simulated by the overlay primitive according to R.

Challenge

The layered approach provides some amount of compositionality and allows us to verify code at an appropriate level of abstraction.

However, CertiKOS contains dozens of layers. Manually defining a language for each layer interface, and writing a monolithic contextual refinement proof for each layer implementation, quickly becomes tedious and repetitive.

To address this, we introduce a module system for certified abstraction layers, which captures the common recurring patterns and allows us to concentrate on what is specific to any given layer.

Introduction

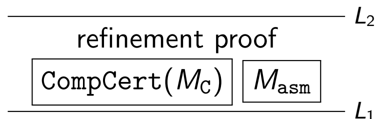
Module system

Anatomy of a layer

Conclusion

Verifying a single layer

All of our layers follow the same pattern:



There are many moving parts:

- the x86 and C abstract machines for L_1 and L_2;
- proofs of correctness for the C and assembly code;
- the Compcert certified compiler;
- a proof the the low-level code specifications refine L_2.

These components must be glued into a contextual refinement proof. How can we avoid ad-hoc boilerplate?

Modules and layer interfaces

We encapsulate what varies into *modules*, *layer interfaces*, and *simulation abstraction specifications*, which express the components of a layer on a per-function/per-primitive basis:

$$L ::= \varnothing \mid i \mapsto \sigma \mid L_1 \oplus L_2$$
$$M ::= \varnothing \mid i \mapsto \kappa \mid M_1 \oplus M_2$$
$$R ::= \mathtt{id} \mid R_1 \circ R_2 \mid \cdots$$

Then this judgement asserts that "M implements L_2 on top of L_1":

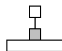
$$L_1 \vdash_R M : L_2$$

We encapsulate the common structure of the proof into the logic of our module system and its soundness proof.

The logic

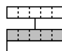
Elementary proof

$$\frac{\mathtt{VC}(L, R, \kappa, \sigma)}{L \vdash_R i \mapsto \kappa : i \mapsto \sigma}$$



Horizontal composition

$$\frac{L \vdash_R M_1 : L_1 \qquad L \vdash_R M_2 : L_2}{L \vdash_R M_1 \oplus M_2 : L_1 \oplus L_2}$$



Vertical composition

$$\frac{L_1 \vdash_R M : L_2 \qquad L_2 \vdash_S N : L_3}{L_1 \vdash_{R \circ S} M \oplus N : L_3}$$

Soundness

$$\frac{L_1 \vdash_R M : L_2}{\forall P. [\![P \oplus M]\!]_{L_1} \sqsubseteq [\![P]\!]_{L_2}}$$

$C[-] \sqsubseteq C[-]$

Building certified layers

We can build certified layers by expressing the components of our proofs in a unified framework:

- Code proofs can be stated as $L_1 \vdash_{\mathtt{id}} M : \Sigma$;
- Refinement proofs can be formulated as $\Sigma \vdash_R \varnothing : L_2$;
- The correctness theorem for Compcert can be formulated as:

$$\frac{L_1 \vdash_R^{\mathtt{C}} M : L_2}{\mathtt{CallConv}(L_1) \vdash_{R \circ \mathtt{inv} \circ \mathtt{inj}}^{\mathtt{Asm}} \mathtt{CompCert}(M) : \mathtt{CallConv}(L_2)}$$

In the following, we will only consider the Clight instance of the module system.

Introduction

Module system

Anatomy of a layer

Conclusion

Overview

Under the `CAL/tutorial/` directory of the summer school's git repository, you will find some examples of layer implementations. They are Clight layers of varying complexity.

As an exercise, you will attempt to fill in some of the missing parts. The file `CAL/tutorial/Tutorial.pdf` has some instructions. The solutions are available under `CAL/tutorial.solutions/`.

In the rest of this presentation, I will introduce some of the key interfaces in the Coq implementation of our module system, and walk through the example layer `CAL/tutorial/stack/Counter.v` to give you a sense of how it works and where to start.

General procedure

To build a certified layer on top of a given underlay L_l:

- Define its overlay interface L_h;
- Define a module M containing the code of its functions;
- Define an intermediate layer interface Σ with low-level specifications;
- Prove one-by-one that the functions implement Σ on top of L_l;
- Prove one-by-one that the low-level specifications in Σ refine the high-level specifications in L_h.

Abstract data

Before we can define a layer interface, we need to define the kind of abstract states that it will use:

```
Record layerdata :=
  ldata {
    ldata_type :> Type;
    ldata_ops : AbstractDataOps ldata_type;
    ldata_prf : AbstractData ldata_type
  }.
```

Abstract data

In addition to the type of abstract states, we specify an initial value and an invariant, then show that they satisfy some required properties:

```
Class AbstractDataOps data :=
  {
    init_data : data;
    data_inv: data -> Prop;
    [...]
  }.

Class AbstractData data `{ops: AbstractDataOps data}: Prop :=
  {
    init_data_inv: data_inv init_data;
    [...]
  }
```

(liblayers/simrel/AbstractData.v)

Extended memory model

Once we've define a kind of abstract states D, we can use Compcert with the new memory model `mwd D`, which is a pair consisting of a normal concrete Compcert memory state, and an abstract data component:

```
Context `{Hmem: BaseMemoryModel} (D: layerdata).

Definition mwd := (mem * D)%type.

[...]

Global Instance mwd_ops: Mem.MemoryModelOps mwd.
Global Instance mwd_prf: Mem.MemoryModelX mwd.
```

(liblayers/simrel/MemWithData.v)

Primitive specifications

A primitive specifications is essentially a transition relation, from a call state consisting of an initial memory state and actual arguments, to a return state consisting of a final memory state and return value:

```
Record cprimitive (D: layerdata) :=
  {
    cprimitive_step:
      csignature -> list val * mwd D -> val * mwd D -> Prop;
    [...]
    cprimitive_step_wt sg sinit v' m':
      cprimitive_step sg sinit (v', m') ->
      Val.has_type v' (typ_of_type (csig_res sg))
  }.
```

(liblayers/compcertx/CPrimitives.v)

High-level primitive specifications

Most of the time the primitives will only manipulate the abstract data component of the state. Then, the can be computed automatically from specifications written as high-level Coq functions.

For instance, if we have a function of type:

```
f :  Z -> D -> option D,
```

then the (C-style) primitive specification `cgensem f` will update the abstract state according to `f x` whenever it is called with an integer argument of value `x`.

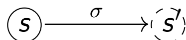
(liblayers/compcertx/GenSem.v, liblayers/compcertx/CGenSem.v)

Invariant preservation

It will be convenient when writing code proofs to be able to assume the following:

- The invariant `data_inv` hold on the abstract state component;
- The memory and arguments only contain valid pointers.

To make this possible, we need to show that our primitive specifications actually preserve this invariant:



Invariant preservation

This can be done by declaring an instance of the following class:

```
Class CPrimitivePreservesInvariant D (σ: cprimitive D) :=
    {
      cprimitive_preserves_invariant sg args m d res m' d':
        cprimitive_step D σ sg (args, (m, d)) (res, (m', d')) ->
        cprimitive_inv_init_state D args m d ->
        cprimitive_inv_final_state D res m' d';
      cprimitive_nextblock_incr sg args m d res m' d':
        cprimitive_step D σ sg (args, (m, d)) (res, (m', d')) ->
        cprimitive_inv_init_state D args m d ->
        (Mem.nextblock m ≤ Mem.nextblock m')%positive
    }.
```

There is a corresponding class `GenSemPreservesInvariant` for high-level specifications computed with `cgensem`.

Code proofs

s —σ_l→ s'

s ⇢ s' (dashed): $\texttt{Clight}(L_l,\kappa)$

$$\frac{\quad}{L_l \vdash_{\texttt{id}} i \mapsto \kappa : i \mapsto \sigma_L}$$

By composing these horizontally we obtain:

$$L_l \vdash_R M : \Sigma$$

Abstraction relations

Before we can write the refinement proofs, we need to define an *abstraction relation* between the kinds of abstract data used by the overlay and the underlay, which will specify a transformation of the data representation between the two layers:

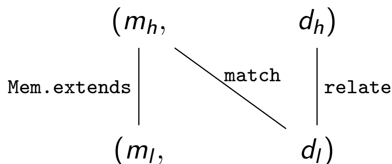
```
Record abrel (D1 D2: layerdata) :=
  {
    abrel_ops :> abrel_components D1 D2;
    abrel_prf : AbstractionRelation D1 D2 abrel_ops
  }.
```

Abstraction relation components

The relation has the following components:

```
Record abrel_components (D1 D2: layerdata) :=
  {
    abrel_relate: rel D1 D2;
    abrel_match: rel D1 mem;
    abrel_new_glbl: list (ident × list AST.init_data)
  }.
```

Schematically:



Abstraction relation properties

The relation should satisfy the following properties:

```
Class AbstractionRelation D1 D2 (R: abrel_components D1 D2) :=
  {
    abrel_relate_init_mem:
      abrel_relate D1 D2 R init_data init_data;
    abrel_match_init_mem m2:
      abrel_init_props m2 (abrel_new_glbl D1 D2 R) ->
      abrel_match D1 D2 R init_data m2;
    abrel_match_unchanged_on :>
      Monotonic
        (abrel_match D1 D2 R)
        (- ==> Mem.unchanged_on (abrel_new_glbl_mask D1 D2 R) ++> impl);
    [...]
  }.
```

Refinement proofs

$$\frac{\begin{array}{ccc} \textcircled{s_h} & \xrightarrow{\sigma_h} & s'_h \\ | R & & \vdots R \\ \textcircled{s_l} & \overset{}{\underset{\sigma_l}{\dashrightarrow}} & \exists s'_l \end{array}}{L_h \vdash_{\mathtt{inv} \circ R \circ \mathtt{inv}} i \mapsto \sigma_L : i \mapsto \sigma_H}$$

By composing these horizontally we obtain:

$$\Sigma \vdash_R \varnothing : L_h$$

Linking

Using vertical composition, we can link the code and refinement proofs to obtain the whole certified layer:

$$L_l \vdash_R M : L_h$$

As we add more whole layers on top of L_h, we can in turn vertically compose them together.

When we have everything we need, we can apply the module system's soundness proof and obtain the contextual refinement property:

$$\forall C, [\![C \oplus M]\!]_{L_l} \sqsubseteq [\![C]\!]_{L_h}$$

Introduction

Module system

Anatomy of a layer

Conclusion

Conclusion

For homework:

- Pull the latest version of the DSSS git
- Start working through `CAL/tutorial/Tutorial.pdf`

We will be around to answer any question.

The Coqdoc for the tutorial can be browsed at:
`https://certikos.github.io/tutorial-coqdoc/toc.html`.
These slides can be found under `CAL/slides.pdf`.

Tomorrow I will discuss some of the implementation techniques used in the module system.