CertiKOS Layer Tutorial

Before you begin

It takes about 20 minutes for the Coq files we will be using to compile on *really nice* hardware. On somewhat less powerful hardware, it can take closer to 30-35 minutes or longer. If you intend to follow along with the exercises, it'd be a good idea to get the build process started. If you don't intend to do the exercises or would rather read more about this tutorial before starting the build, feel free to skip this section for now and come back later.

First, make sure you are in an environment that has all the necessary dependencies. If you are using nix, this is as simple as running `nix-shell .` from the directory with `default.nix`. If you are not using nix, make sure you have the following dependencies:

- Coq version 8.6
- Ocaml version 4.02 or later
- Ocaml .opt compilers
- Menhir version 20161201 or later
- GNU Make version 3.80 or later

From the directory that contains `default.nix` and `DSSSCoqProject`, run the `configure` script. You will be presented with a list of possible targets. Choose the appropriate target from this list and re-run `configure` with that target as an argument. For example, if you are on a 64-bit Linux system, run:

```
./configure x86_64-linux
```

It will be helpful to do as much work in parallel as possible. The provided Makefile has some support for parallelizing the build. If your machine has `n` cores, you should invoke `make` with at least `n+2` worker threads. The following example shows how to invoke `make` on a machine with 4 cores:

```
make -kj 6 quick
```

After this is finished running, either there will have been an error earlier in the build than expected (some files in the `tutorial/` directory are expected to fail) or you will be ready to begin exploring the `.v` files in `tutorial/` with your Coq IDE of choice.

Introduction

CertiKOS is a certified operating system kernel being developed and researched at Yale University. Operating system kernels are quite complex on their own, and adding the burden of proving correctness can greatly amplify that complexity.

To ameliorate this problem, CertiKOS employs a layered approach to verification in which small modules are proven to implement their specifications independently. Then a linking theorem shows that the composition of those modules meets the specification of the complete system.

This tutorial will guide you through the construction of one or two such layers in order to demonstrate the process.

Skills

We assume you have certain skills coming into this tutorial:

- Basic experience using the Coq theorem prover (if you've made it at least most of the way through Software Foundations, this should be enough)
- Minimal knowledge of C
- Basic experience using the command line

The tutorial will exercise some potentially new skills:

- Producing and working with CompCert models of C program behavior
- Expressing the function of a program module in terms of high-level, functional programs and data structures
- Proving simulation relations between high-level and low-level descriptions of program behavior

Organization

```
certikos-private
+- Makefile
+- compcert     -- Modified version of Inria's CompCert verified C compiler
+- compcertx    -- Additions to and modifications of CompCert
+- coqrel
+- liblayers    -- Layer calculus library
\- tutorial     -- Subject of this tutorial. Study these.
   +- common       -- Commonly used theorems and tactics;
   |  \- TutoLib.v    -- some may be merged into liblayers in the future
   +- container -- A simple practical layer, actually used in CertiKOS
```

```
|  +- Container.v       -- Container layer (built on getter/setter)
|  +- ContainerIntro.v  -- Container getter/setter layer
|  +- ContainerType.v   -- Abstract representation of containers
|  +- container.c       -- C implementation of containers
|  \- container_intro.c -- C implementation of getters/setters
+- queue        -- Array-backed bounded queue layer
|  +- AbsQueue.v    -- Refinement layer from Queue to high-level queue
|  +- Node.v        -- Queue node layer definition
|  +- Queue.v       -- Medium-level Queue layer
|  +- QueueData.v   -- Abstract representation of a queue
|  +- QueueIntro.v  -- Getters and setters for queue head and tail
|  +- node.c        -- C definition of a queue node
|  +- queue.c       -- C implementation of bounded queue
|  \- queue_intro.c -- C implementation of getters/setters
+- stack        -- Toy layers demonstrating the basics; start here
|  +- Counter.v    -- The Counter layer
|  +- Stack.v      -- The Stack layer, built upon the Counter layer
|  +- counter.c    -- C code implementing a counter
|  \- stack.c      -- C code implementing a stack, using a counter
\- Tutorial.md  -- This document
```

Theory

Here is a high-level overview of some of the theory we will be using in building layers. References are included to relevant papers that have much greater detail. Reading them is highly recommended.

Deep Specifications

You may already be familiar with the use of type systems (like the ones used in C or Haskell) and/or contracts (like the ones in Racket, for example) to specify or constrain the behavior of programs and help prevent unintended program behaviors. These are examples of shallow specifications. Shallow specifications do not fully describe the intended functionality of a program or module; they only put bounds on behavior.

Deep specifications[1] on the other hand capture the precise functionality of a program, with every necessary detail. They also incorporate any and all relevant assumptions about the program's computational context (i.e. what primitives are available and what operations are allowed with what resources) which the implementation depends on in order to provide its functionality.

[1] http://dl.acm.org/citation.cfm?id=2676975 "Ronghui Gu et al. 2015. Deep Specifications and Certified Abstraction Layers."

Simulation Relations

A simulation relation[2] is a relation over the states of two automata such that some important properties of traces of the two automata hold for related states. For instance, in a forward simulation $A \leq_F B$ if s is a start state of A then some state in B related to s must also be a start state, and whenever a step $s' \xrightarrow{a}_A s$ is in A and s is related to u' in B there exists a state u and some trace β in B such that $u' \overset{\beta}{\Rightarrow}_B u$. Together with an additional property about A, this would let us prove that every time A makes a move, B makes an analogous move. A's behavior *simulates* the behavior of B.

Each layer of abstraction in the CertiKOS kernel defines several relations. Some of these are simulation relations. Simple, functional, deep specifications are proven to simulate more complex, imperative programs. We can then reason about these simpler deep specifications and build other layers on top of them, confident that the behavior of the resulting system will be described by the specifications.

Layers

(TODO: Ask Jeremie to help come up with a concise definition of what a layer is)

Overview

The walkthrough below will take you through the following:

Learn to build a simple layer on the raw CompCert C model

Work through the `stack/Counter.v` file. Learn the basic structure of a layer definition and practice proving the necessary lemmas and theorems involved in its specification.

Learn to build a layer on top of an existing layer.

Work through the `stack/Stack.v` file. Learn how to build a new layer on top of an existing one.

[2] http://dl.acm.org/citation.cfm?id=889596 "Nancy A. Lynch and Frits W. Vaandrager. 1994. Forward and Backward Simulations Part I: Untimed Systems."

Explore a more practical layer from CertiKOS.

Work through the `container` directory. Learn how to represent C code in Coq and then prove things about it.

Walkthrough

Before you begin, open one or more of the developments under the `tutorial` directory (such as `tutorial/stack/Counter.v`) in your Coq IDE of choice and make sure you can at least get through the `Require`s at the beginning. If you can't, you'll need to (re)compile whatever's failing. It may be easiest to just recompile everything with `make clean && make` from the top level, though this will take a *significant* amount of time.

The Counter Layer: `stack/Counter.v`

Some real easy stuff just to get warmed up:

- [ ] Read the comments after the `Require`s for a brief overview of Counter and what it's meant to do.
- [ ] `MAX_COUNTER` is defined to be equal to 10, and it will later be important that this is less than or equal to `Int.max_unsigned`. The proof of this fact is omitted. See if you can prove this fact with just one tactic.
- [ ] Read `Section AbsData.` There isn't much to do in this section because we want it to be a firm example to build on later.

The AbsData section defines the abstract data representation of the layer upon which we will build Counter (the underlay) and the abstract data representation that the Counter layer will present to its clients (the overlay). Since there are no layers beneath Counter to speak of (save the C language itself), the underlay is essentially empty.

A note on the difference between `Qed` and `Defined`: There is one. Most often, we want to be using `Qed`. But there are times when we may wish to unfold proof values later, in order to show facts about their construction. At those times, we use `Defined` in order to make these proof values transparent.

- [ ] Read `Section HighSpec.` This section defines the abstract behavior of Counter.
- [ ] Complete the proof for `decr_counter_preserves_invariant`. It should be similar to `incr_counter_preserves_invariant`. (Hint: In the context of this proof, `data_inv d` contains the information that the

existing abstract data satisfies the layer invariant. In other words, it must be true that `(counter d) <= MAX_COUNTER`. Using the `cbn` or `simpl` tactics can make this fact transparent.)

Glance through `Section Code.` The C code from which these definitions were generated is provided in comments. It shouldn't ever be necessary to write such definitions by hand. The `clightgen` command provided by CompCert can be leveraged to produce them.

- [ ] Read `Section LowSpec.` These inductive definitions capture the behavior of the C code definitions above. If it seems like this is a bit mechanical and redundant, you're not wrong. Research is being done on a tool that will automate the creation of these definitions from their respective C code and the proofs that each one is an accurate model of the original code's behavior (the proofs in the next section).
- [ ] Unspeakable things were done to the definition of `incr_counter_step`. Replace the four `False`s with appropriate hypotheses. Looking ahead to `decr_counter_step` is *not* cheating.
- [ ] Read `Section CodeLowSpecSim.` These prove simulation relations between the C modules generated by `clightgen` and the low-level specifications of the previous section.
- [ ] Since the hypotheses of `incr_counter_step` are all `False`, someone thought it would be funny to use this to "prove" the simulation relation, `incr_counter_code`. At least they had the decency to have `Admitted` that it was a joke. Replace the `contradiction` with an actual proof.
- [ ] Read `Section LowHighSpecRel.` This section defines a relation between the abstract representation of the counter, as a record with a natural number as a field, and the implementation representation, as a global variable in memory.

TODO: Exercises in this section

- [ ] Read `Section LowHighSpecSim.` The hard part is proving a simulation relation between the low-level implementation and the abstract, functional specification.

There are a lot of gorey details in these proofs due to the need to sort out exactly what is happening to memory and how it relates to what is happening to the abstract state. The connection these proofs have to the actual theory is not always apparent. Some of the work being done at Yale involves providing better tactics, lemmas, theorems, and documentation to help make working with proofs like these easier and more transparent.

- [ ] The proof of `incr_counter_refine` has been removed. If you are feeling very brave and adventurous, try to write it from scratch using the following tactics and lemmas. If you are somewhat more sane, copy and paste from `decr_counter_refine` and fix what breaks. The middle road might be the most educational: Try to write a proof on your own, but refer to `decr_counter_refine` whenever you get stuck. Here are those tactics and lemmas I promised:
  - apply
  - assert
  - assumption
  - auto
  - cbn
  - constructor
  - destr_in
  - destruct
  - do
  - eapply
  - eauto
  - econstructor
  - erewrite
  - f_equal
  - generalize
  - intros
  - intuition
  - inv
  - inverse_hyps
  - inversion
  - inv_generic_sem
  - inv_monad
  - omega
  - pose proof
  - red
  - refine_proof_tac – A great way to start
  - reflexivity
  - rewrite
  - specialize

- split
- subst
- try
- unfold
- Int.repr_unsigned
- Int.unsigned_range
- Int.unsigned_repr
- Mem.load_store_same
- Mem.nextblock_store
- Mem.perm_store_1
- Mem.store_outside_extends
- Nat2Z.inj_lt
- Z2Nat.id
- Z2Nat.inj
- Z2Nat.inj_add
- Z.divide_0_r

- [ ] Read `Section Linking.` Each function we've defined and verified so far has stood on its own, as a whole program. We need to show that these three programs can be composed into a layer.

The Stack Layer: `stack/Stack.v`

The Stack layer implements a bounded stack of bounded integers. There are still plenty of comments in this file, explaining what's going on in great detail. Feel free to read through until you reach the exercises. Be prepared to see things you've seen before (in `Counter.v`).

In order to keep things interesting, the exercises in this layer are focused around the really big differences from the `Counter` layer. There might, however, be some interesting proofs and definitions to investigate on the way. So if something piques your interest, you are encouraged to explore! Step through proofs. Rip pieces out and try to replace or reproduce them. Enjoy, and we'll meet you for the first exercise of the layer in `Section LowSpec.`

- [ ] The inductive definition of `pop_step` has been vandalized. What a pity, since it was such a good example of just how much these intermediate C module specifications can be simplified by building on top of lower layers! Replace its two False premises with premises that reflect the operations actually done by the C code.

- [ ] The proof that `pop_cprimitive` preserves the invariant has been left undone as a result of the above-mentioned vandalism. Complete it to properly define the global instance `pop_cprim_pres_inv : CPrimitivePreservesInvariant _ pop_cprimitive`.
- [ ] Try to state and prove `pop_refine`. If you go about it the same way as most of the other refinements you've encountered so far, you're going to run into trouble in a couple of places. Most of the proof you'll produce is relevant anyway, so go ahead and get as far as you can and expect to wind up with a couple of `admit`s.

These tactics may be useful:

- auto
- cbn
- constructor
- destr
- destr_in
- do
- eassumption
- eauto
- econstructor
- f_equal
- generalize
- intros
- inv
- inverse_hyps
- inversion
- inv_generic_sem
- inv_monad
- omega
- refine_proof_tac
- reflexivity
- repeat
- rewrite
- split
- unfold

And these values/theorems will probably help:

- decr_counter_high_spec
- Int.repr_signed
- Int.unsigned_repr

- MAX_COUNTER_range
- MemRel
- Nat2Z.id
- Nat2Z.inj_sub
- Nat.sub_0_r
- pop_high_spec
- stack_counter_len0
- Z.of_nat

In order to make a proper `pop_refine`, you'll need facts that come from the layer invariants. Conceptually, it would be enough to add "Assuming that the layer invariants hold …" to the beginning of the lemma statement.

- [ ] `Check inv` and have a look at its type. As a `simrel`, it can compose with other `simrels`, like `stack_R`, with the `compose` function - ∘. Composing `inv` with a `simrel` gives it enough information to tell what invariants it actually represents (it has an inferred type argument).

- [ ] Enhance the statement of the lemma `pop_refine` by composing with `inv`. You'll need it twice, actually. `inv` refers to different invariants depending on whether you compose it on the left or the right. One will be the high-level invariants, and the other will be the low-level invariants.

- [ ] Finish your proof of `pop_refine`. If you used `refine_proof_tac` in your original proof, it should be very easy. You may find the `xomega` tactic helpful.

- [ ] It all comes down to the linking theorem. An interesting thing about the linking theorem is that it need not mention `base_L` explicitly, even though the Counter layer depends on it. State and prove a linking theorem for the Stack layer. The proof will be very short using automation, but you will have had to be paying attention to state the theorem correctly.

- [ ] Once you've stated and proved (or admitted) the linking theorem for the Stack layer, you can add `stack_link` to the `linking` hint list (just uncomment the `Hint Resolve` line after the `stack_pres_inv` lemma) and change the `Admitted` to a `Qed` at the end of `stack_counter_link`.

If you've done all the exercises and the final proofs don't go through, it could mean one of two things: Either one of the exercise theorems isn't correct, or you chose a valid theorem that can't be used to prove non-exercise theorems the way we did.

Container from CertiKOS: `container/`

The Container layers demonstrate the practical use of the layer approach in a real software system. The source files do a pretty good job of explaining what containers are and what they do, but a short summary might be that containers are the accounting part of memory management in the CertiKOS kernel.

The exercises in this directory are largely about getting a representation of C code into Coq. In Counter and Stack, you may have noticed big, unwieldy `Definition`s that looked suspiciously like C ASTs. In reality, they were big, unwieldy `Definition`s of C ASTs. They were generated from C files using the `clightgen` command and then slightly modified.

Accurately representing C code in Coq is important for verification purposes. In this exercise, you will generate and edit big, unwieldy `Definition`s of C ASTs for use in `Container.v`. Find the TUTORIAL marker in `Section Code.`

- [ ] The big, unwieldy `Definition`s are gone from this section. It would be a relief, except now there are all these `Admitted`s everywhere. Have a look at `container.c`.

This is the code that needs to be represented in `Container.v`. Of course, Coq can't read C. So it will need some translation.

Before taking the next step, a question: Are you doing these exercises on an operating system that considers files to be different if they have the same name but are spelled with different capitalizations? If not, you should rename `container.c` so that the next step doesn't do anything foolish, like destroy `Container.v`.

- [ ] Make sure you've got the CompCert binaries in your path and run `clightgen container.c` (or your relevant, renamed file as the case may be). This will generate `container.v`.

It will be tempting to `Require Import` this new `container.v` file from `Container.v`. Don't do this. It's not quite what we want.

It will be tempting to start copying and pasting from this file into `Container.v`. Go ahead, but bear in mind, we only need the big, unwieldy `Definition`s, and they will need tinkering with.

Try to evaluate past the big, unwieldy `Definition`s. This will fail. They're not quite what we need yet. First of all, they have identifiers that begin with an underscore. No such identifiers exist in the context we've brought them into, so remove the underscore from the front of any identifiers that already exist and are in scope (these are usually long names, like `container_get_usage`).

In some cases (like `_ret`) the identifiers are new ones we must introduce. We've done so with several under new names (in the case of `_ret`, it's `ccc_ret`). In other cases, the identifiers denote temporary variables and you should add `Definition`s for them with unique `positive` values.

There will also be type errors. This is because of slight differences in representation between stock CompCert and what the layer calculus uses internally. Things like `Tcons` need to become `Ctypes.Tcons` in order to match the representation available in scope. (Though, don't go overboard. `Tfunction` doesn't need to change, for example.)

There is no `tbool`. Use `tuint`.

One final wrinkle. When `clightgen` converted the C code, it automatically expanded macros like `MAX_CHILDREN`. This is bad. We want that constant in particular to remain a reference to its opaque value. So where you see an `Int.repr 8` in `f_container_split`, you'll need to replace it with `Int.repr MAX_CHILDREN`.

- [ ] Finish modifying big, unwieldy `Definition`s representing C code and uncomment the `Program Definition`s accompanying them.

Great! Now we have representations of the C functions from `container.c`. Now, in order to make a low-level specification proof and later bundle modules together and state a linking theorem, we'll need `cmodule` mappings. These map identifiers like `container_alloc` to `Program Definition`s like `inlinable_f_container_alloc` using the $\mapsto$ notation. We do this so that calling `container_alloc` from some higher layer actually means something.

- [ ] Replace the `Axioms Minit`, `Mconsume`, `Malloc`, and `Msplit` with `Definition`s of proper `cmodule`s.

To make this section feasible, the only other modifications necessary in this file are uncommenting the code proofs. They should just work.

- [ ] Uncomment the code proofs in `Section CodeLowSpecSim.`

It is entirely possible that once you've uncommented the code proofs and removed the `Admitted`s, the proofs won't work. This probably means the big, unwieldy `Definition`s are somehow wrong. It's probably worth going back and seeing if you used the wrong identifier somewhere or forgot to change an 8 into a `MAX_CHILDREN`.

Just in case you think you managed to put together a big, unwieldy `Definition` that is a correct representation of the C code, but is nevertheless sufficiently different from the original version that the code proofs can't cope with it, you *could* try to change the code proofs. It's not what this section is about, but there's no wrong way to tackle the problem if you can indeed find a proof.

About this document

Contributors to this document:

- Lucas Adam Michael Paul (lucas.paul+deepspec2017@yale.edu)

References