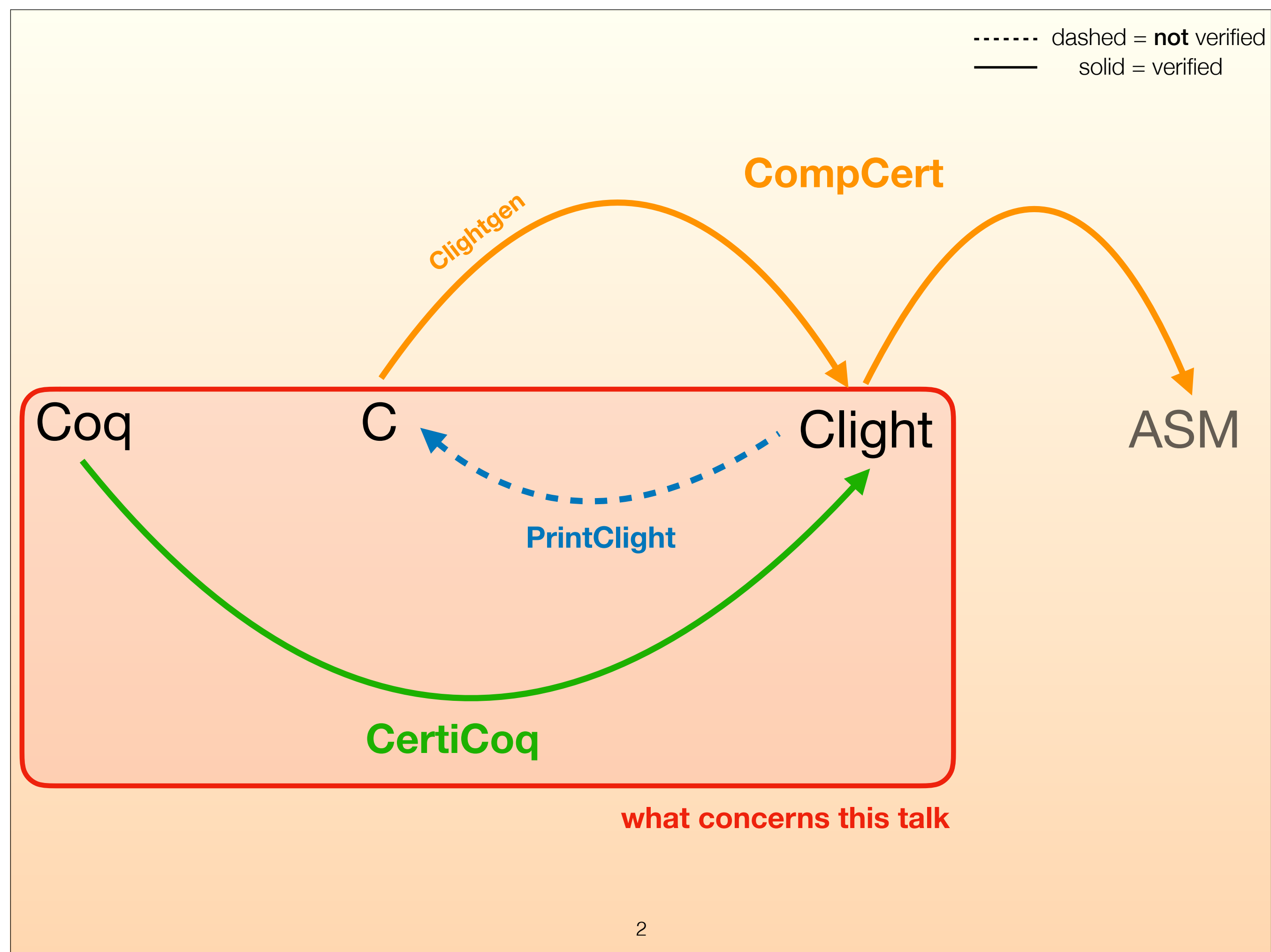
Ergonomics and verification of a **foreign function interface** between **Coq** and **C**

Joomy Korkut
Princeton University

General Exam Talk
May 14th, 2020

Hello everyone! Today I'm gonna talk about the foreign function interface, or FFI, as it is often called, between Coq and C, that I developed for the CertiCoq compiler. I'll explain the mechanisms and design decisions for the interface from an ergonomics and ease of verification point of view.

For those who are not familiar, "foreign function interface" means allowing one language to call a function from another and vice versa. Here we want to do that for Coq and C. We will achieve that by using the "glue code" that we generate. That means extra code in C generated by the compiler separate from the compiled program, that helps us interact with Coq programs in C.



To get familiar with the landscape of the problems we're trying to solve, let's first look at the languages we are dealing with and the compilers we use.

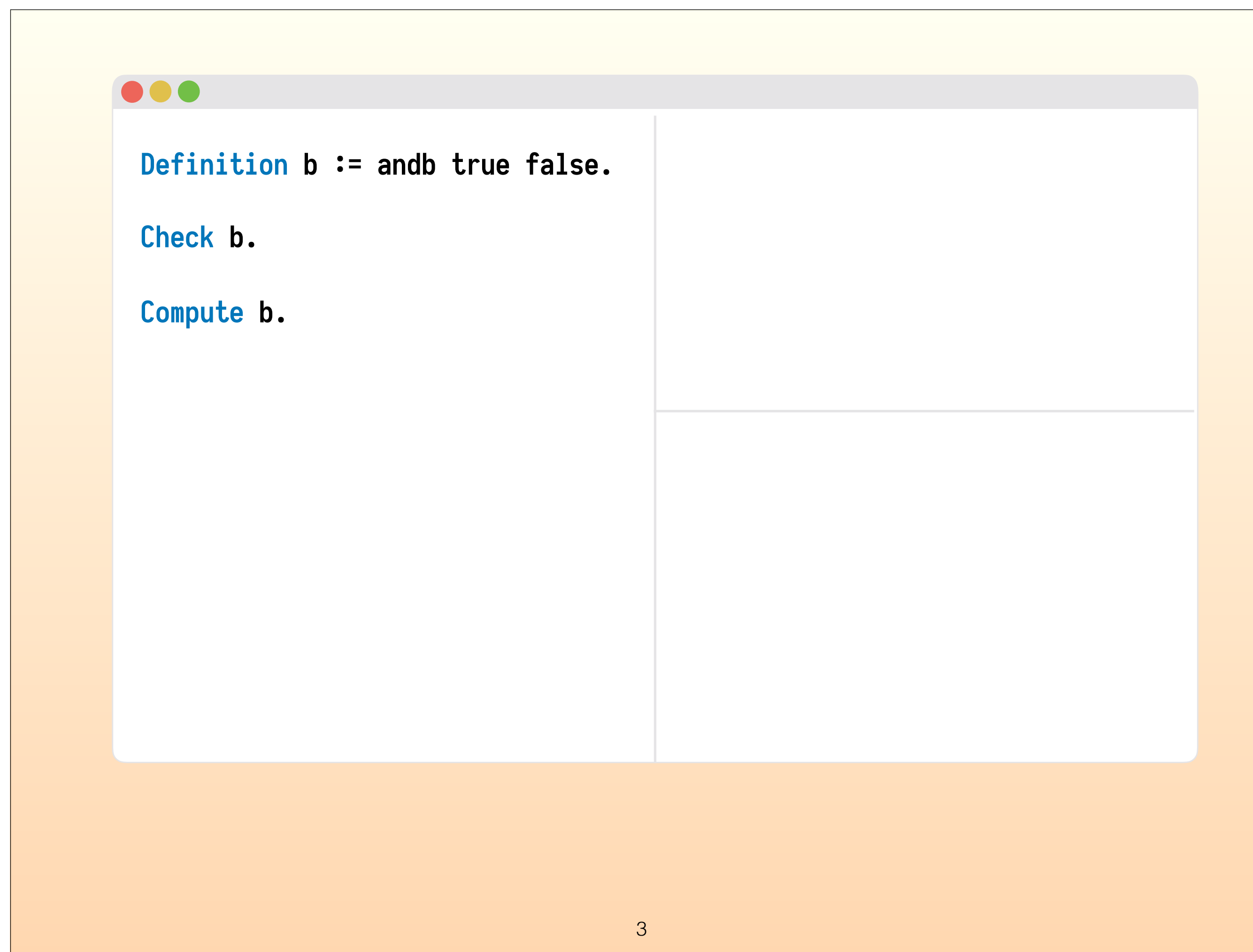
<click> CertiCoq is a verified compiler written in Coq, it is a compiler from Coq to Clight. Clight is a simpler subset of C; it doesn't have typedefs, it doesn't have enum types, it has function calls only as a statement, and such.

Clight does not have a concrete syntax, it only exists in syntax trees.
<click> But it is possible to print Clight syntax trees in C concrete syntax, which is what CertiCoq does. However, this printing is not verified, which we will overlook for now.

<click> This subset of C comes from the CompCert compiler, which is a verified compiler written in Coq, it is a compiler from C to different assembly languages.

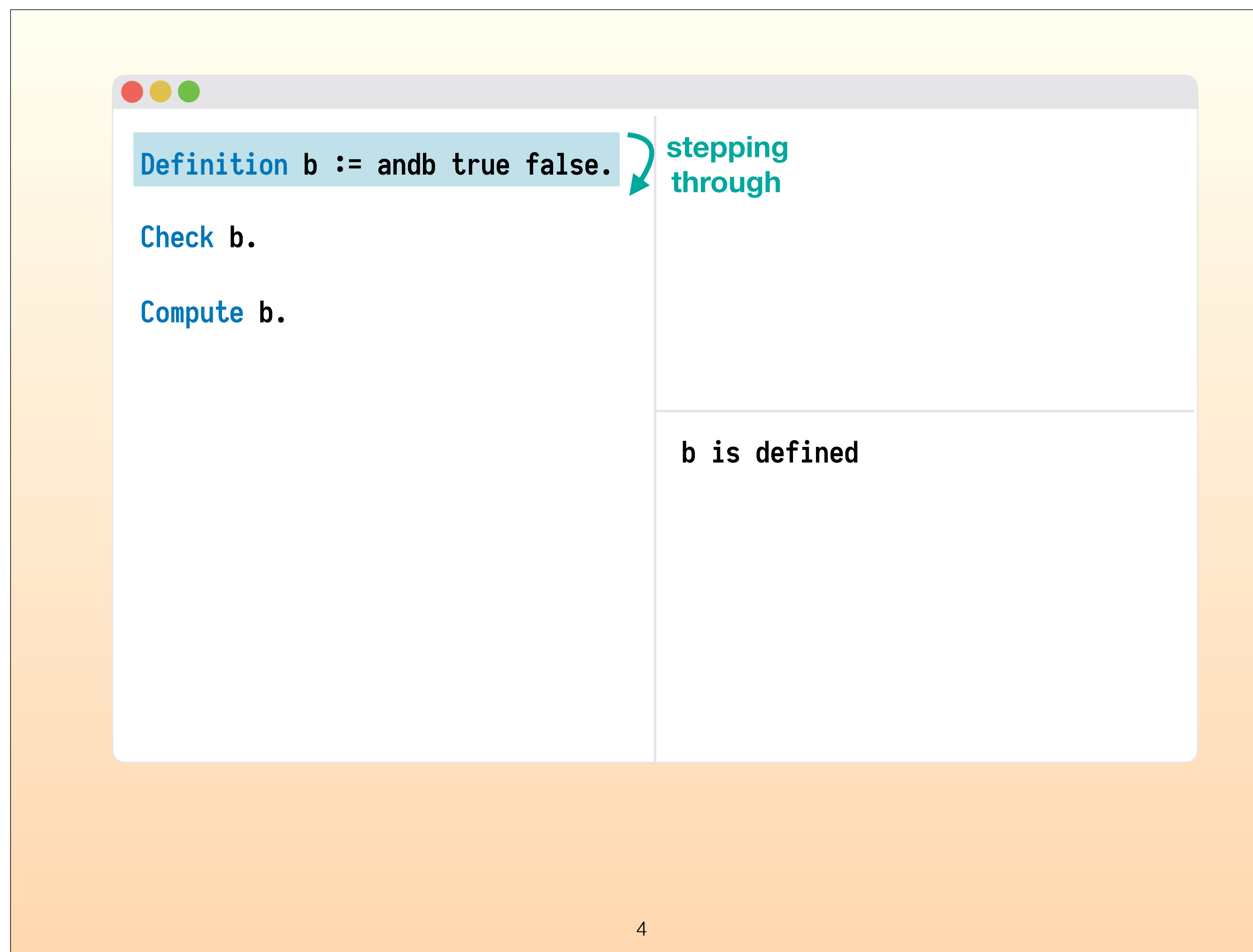
CompCert uses Clight as the one of the first intermediate languages in their compiler pipeline.

<click> That being said, we will not deal with those parts today. We will look at the interaction between Coq and C, and use Clight as a step in between for those. Not only that, I will try to hide the noise that is inherent to Clight in the code excerpts I will show you today.

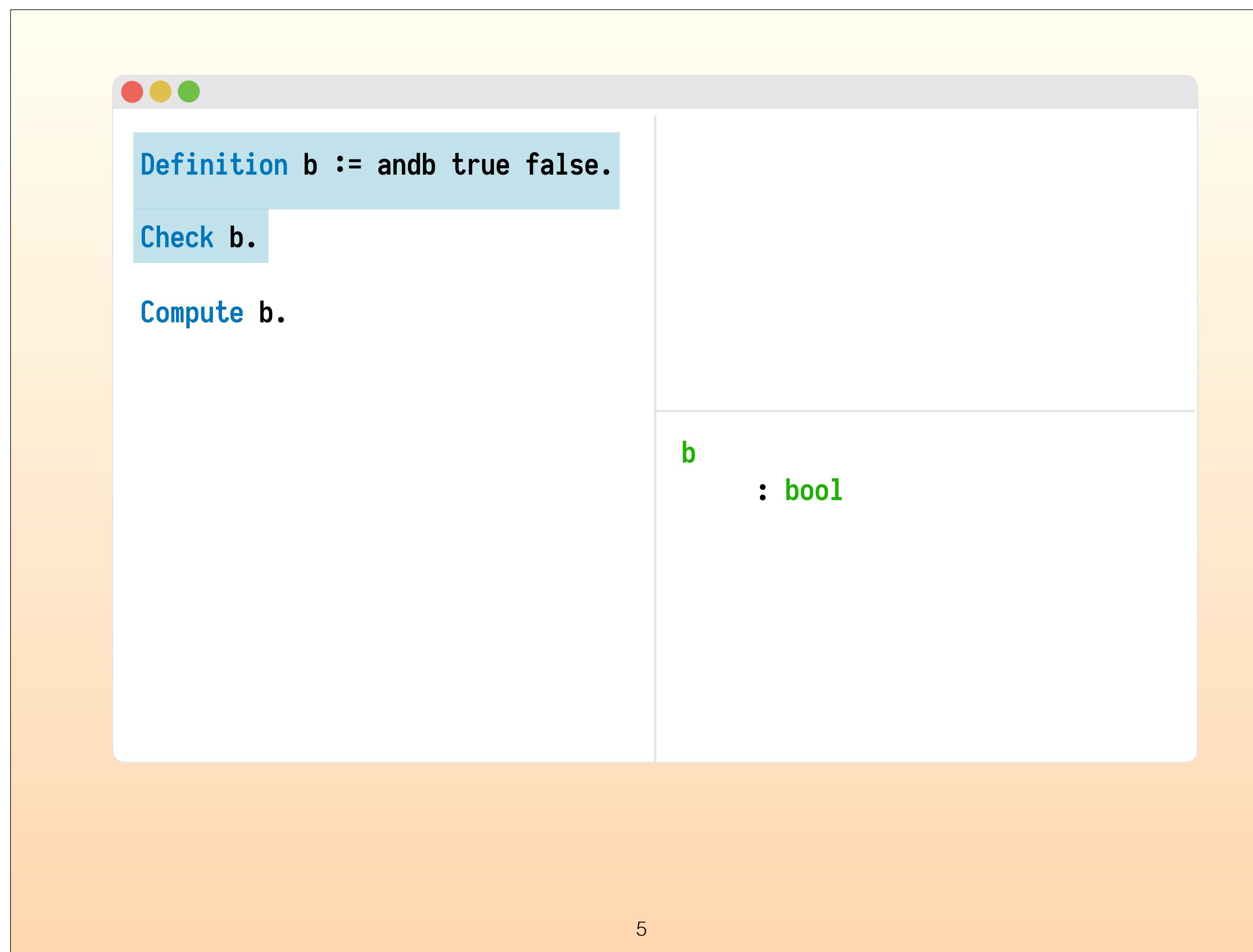


So how does a user compile their Coq program with CertiCoq?

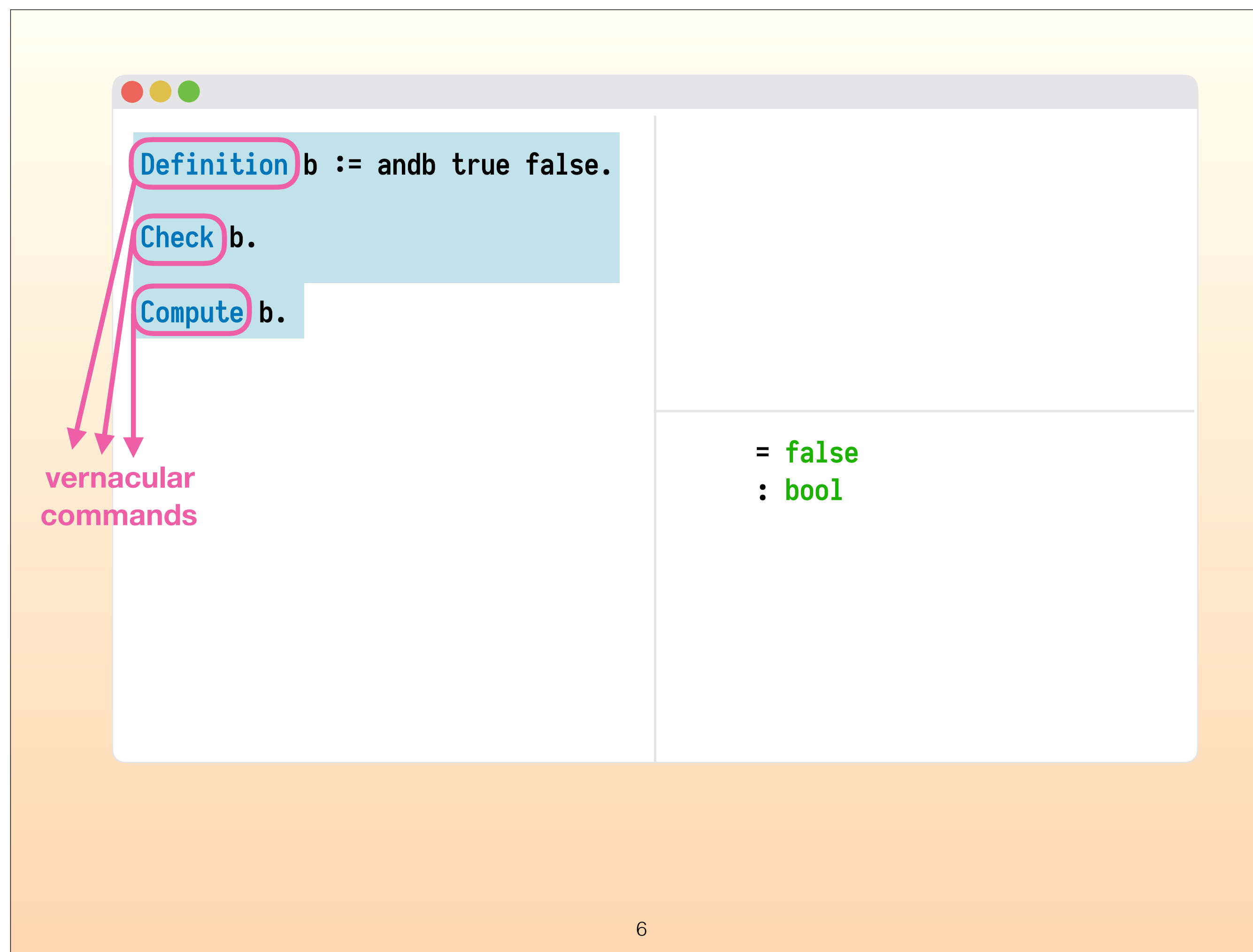
A Coq programming session looks like this, we have a list of commands that we step through one by one.



We can create new definitions and functions with these. We get a response from the Coq environment in the bottom right window pane.

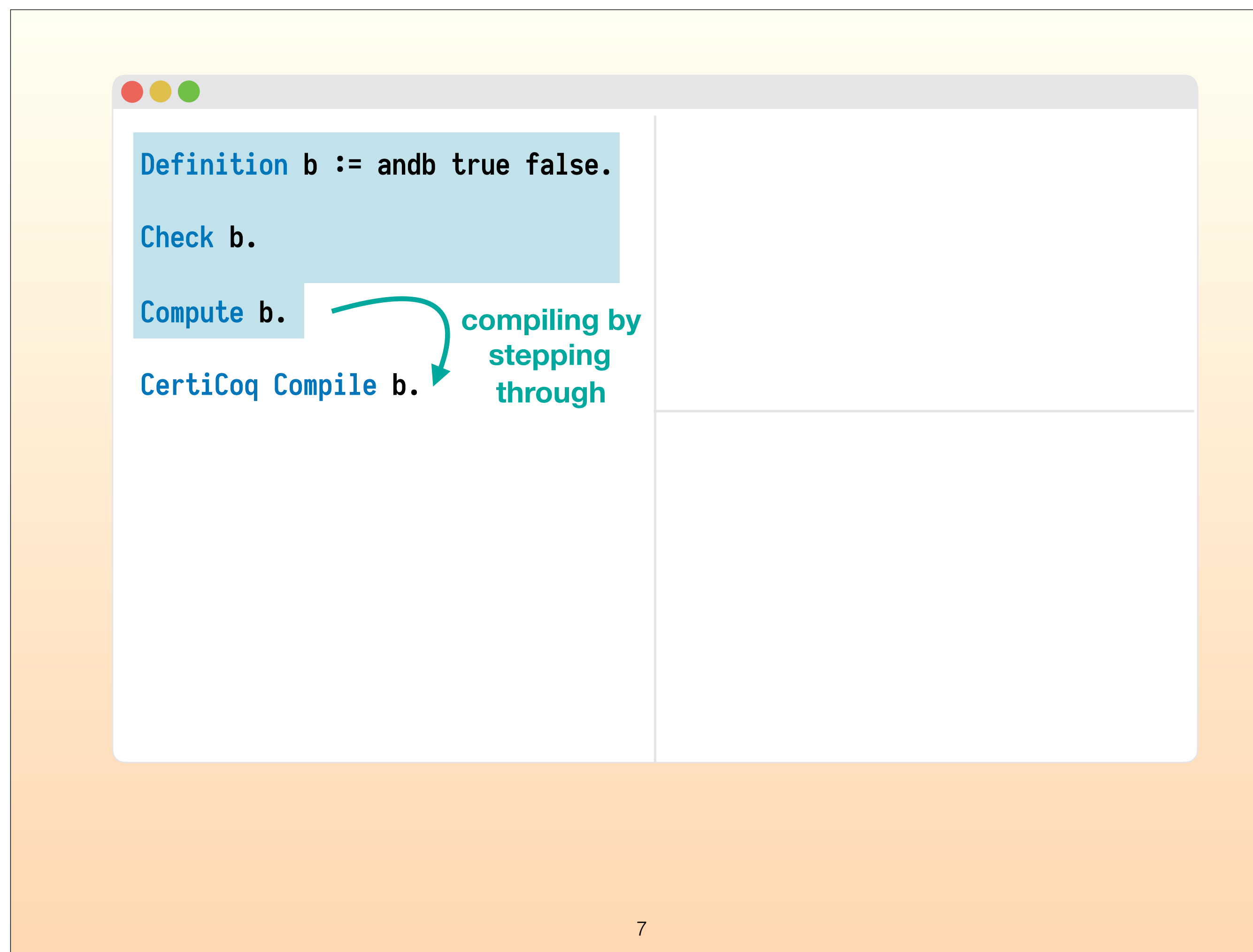


We can ask the types of expressions and definitions, and get a result on the bottom right.



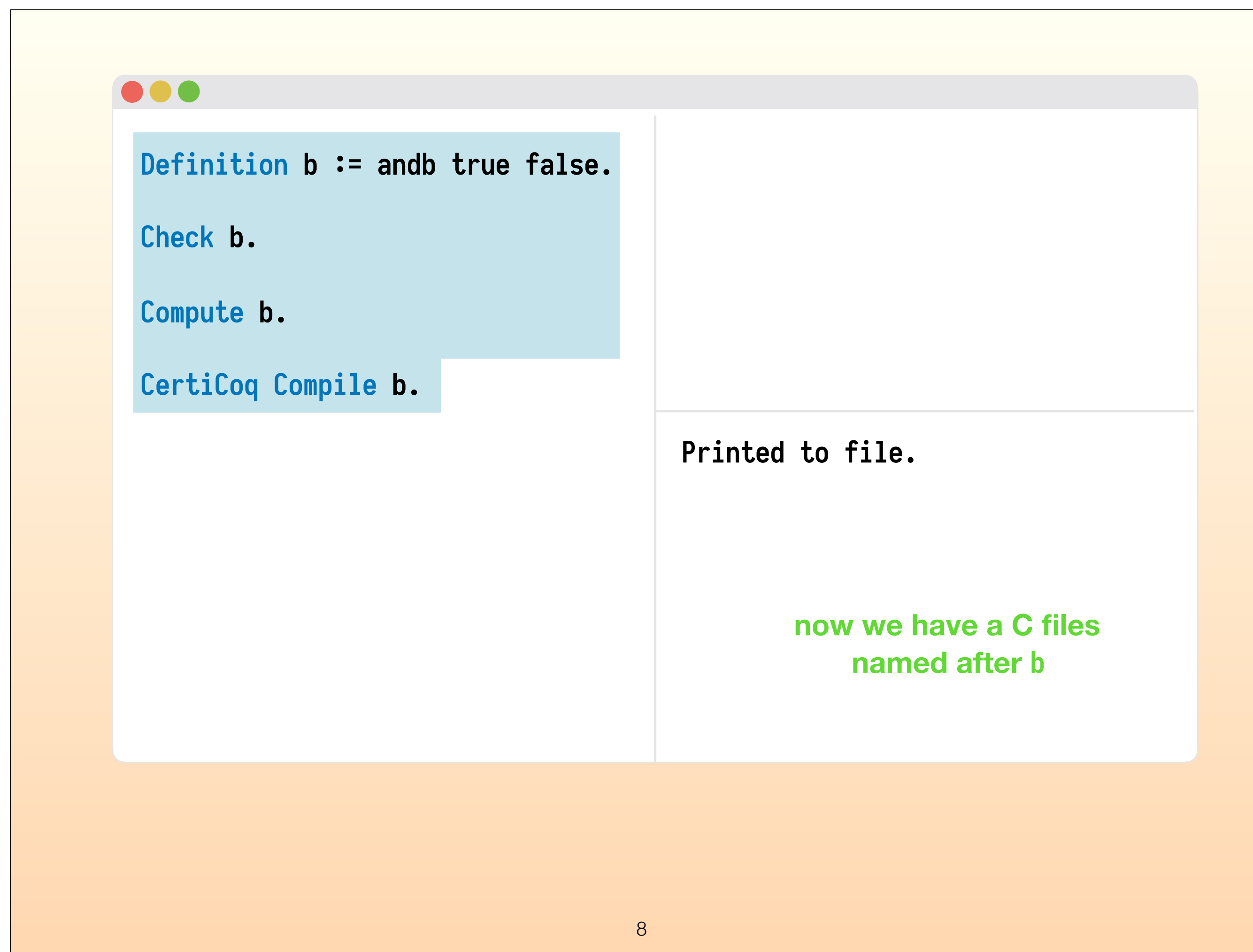
We can evaluate the results of expressions.

<click> Through what we call the "vernacular commands", we turn a Coq programming session into a conversation between the programmer and the environment.

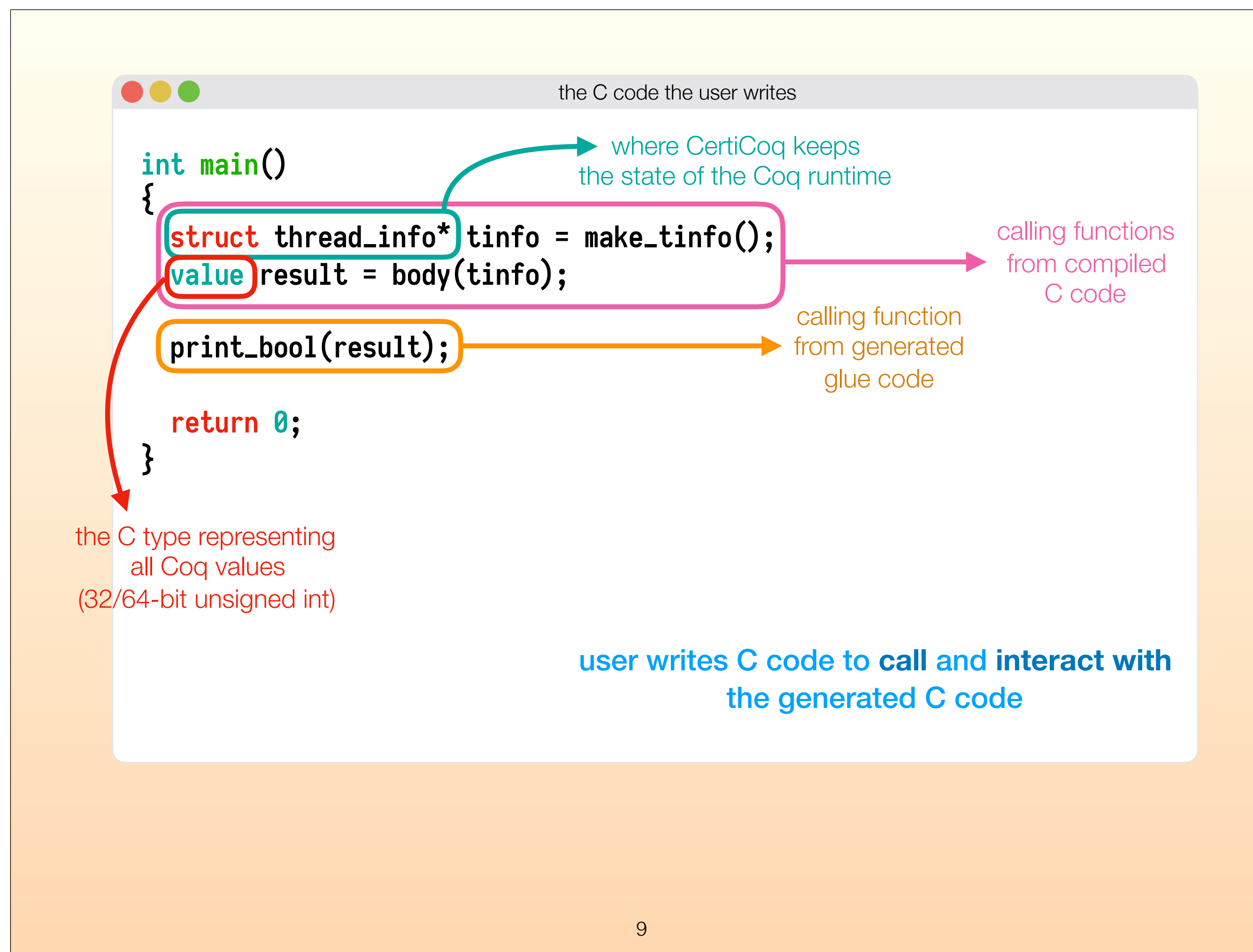


Compiling a Coq definition works just the same way. Unlike other compilers, you don't compile the full Coq file. You compile a single definition at a time. This definition can be simple expression, or a big function, or a tuple of multiple functions, that is up to you.

<click> We step through...



And this creates a new C file in the same directory.



Then what does the user do with that C file? They cannot just compile and run it, because it doesn't have a "main" function. But it can be used as a library.

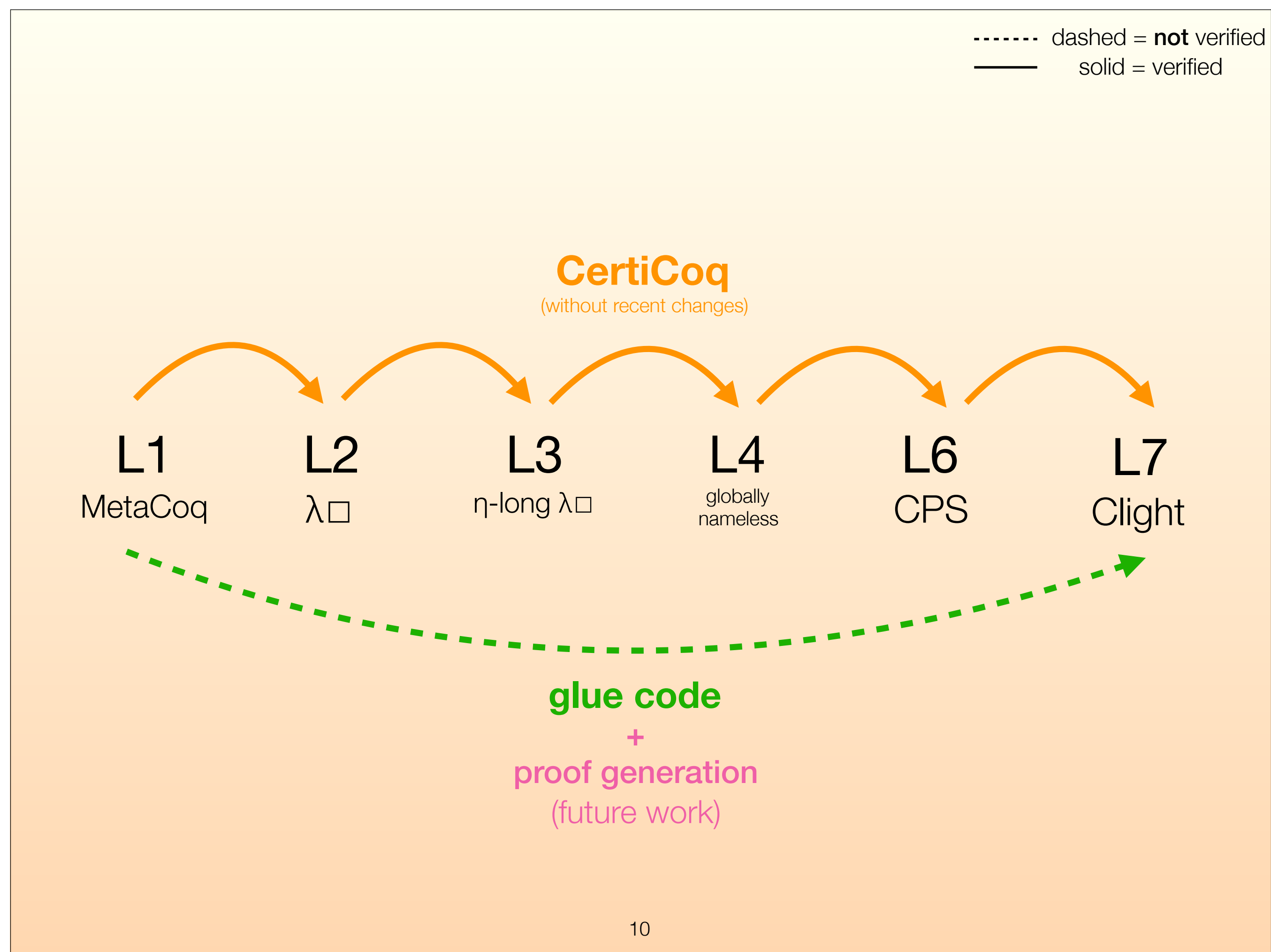
So if the user wants to run it and use the result, they have to write some C code to do that.
<click> The first two lines in the main function here are to set up the Coq runtime and run the initial expression.

<click> Specifically we set up the part of memory to be used to allocate memory for Coq values, and to return Coq values, We do that in this C struct type called thread_info. Any function that needs to allocate new memory on CertiCoq's heap will have to deal with the thread_info.

<click> The type of all CertiCoq expressions is "value", which is an alias for a 32 or 64-bit unsigned integer. In reality, this can be a pointer or an actual integer, both of which fit this size.

<click> Once we have the C representation of the result of the program, we can print it using a print function in the generated glue code.

So this is the simplest example of a C file that uses a glue code function. Users of CertiCoq currently have to write some C code file like this to do something meaningful with the compiler output. We will see more complex examples of this later in the talk.



The CertiCoq compiler consists of many different phases, each defined in Coq. It starts from the MetaCoq description of the term that is compiled, that is, a syntax tree of a Coq program in Coq itself, also often called a **reified program**. After many phases, it eventually generates a Clight syntax tree. However, most information about inductive types are erased fairly early in this pipeline.

<click> For that reason, I worked on a glue code generator, which is an extra code generator that takes the MetaCoq description of a term and generates helper functions in C for the types involved.
<click> One direction we always keep in mind when we do that, is the verification of these functions. We are going to use the Verified Software Toolchain to generate specifications and proofs for these helper functions. This part isn't finished yet, we are still exploring the right way to do that, I'll talk about some ideas on this towards the end of this talk.

example Coq function for the big picture

```
Fixpoint simplify (r : rgx) : rgx :=
  match r with
  | star epsilon            => epsilon
  | star (star r')          => star (simplify r')
  | or epsilon (star r')    => star (simplify r')
  | or empty r'             => simplify r'
  | and r1 r2               => and (simplify r1) (simplify r2)
  | or r1 r2                => or (simplify r1) (simplify r2)
  | star r'                 => star (simplify r')
  | _                       => r
  end.
```

1 inspecting a term's constructor

2 inspecting a constructor's arguments

3 creating new constructor values

4 calling existing functions

5 defining new functions

question: How can we do the same things with Coq values on the C side?

Here's some example Coq code to simplify a regular expression into an equivalent regular expression, we'll look at this code and identify what kind of expressive capabilities we use when we define Coq programs, to get the big picture.

Ideally, whatever we can do in Coq with values of this data type, we should be able to do the same things in C; we want the same expressiveness in C.
The glue code that we generate should accommodate the same kind of actions, but in C, using the C representations of Coq values. And what *can* we do in Coq? Let's look at this example Coq function and identify different parts.

<click> We can inspect what constructor a value is created with.
<click> We can inspect the arguments carried by those constructors.
<click> We can create new values using constructors.
<click> We can call existing functions by passing them arguments.
<click> We can define new functions.

<click> So now we'll look at these 5 expressive capabilities...
<click> and see how we can do the same things with Coq values, ... but on the C side.

How can we do the same things with Coq values on the C side?

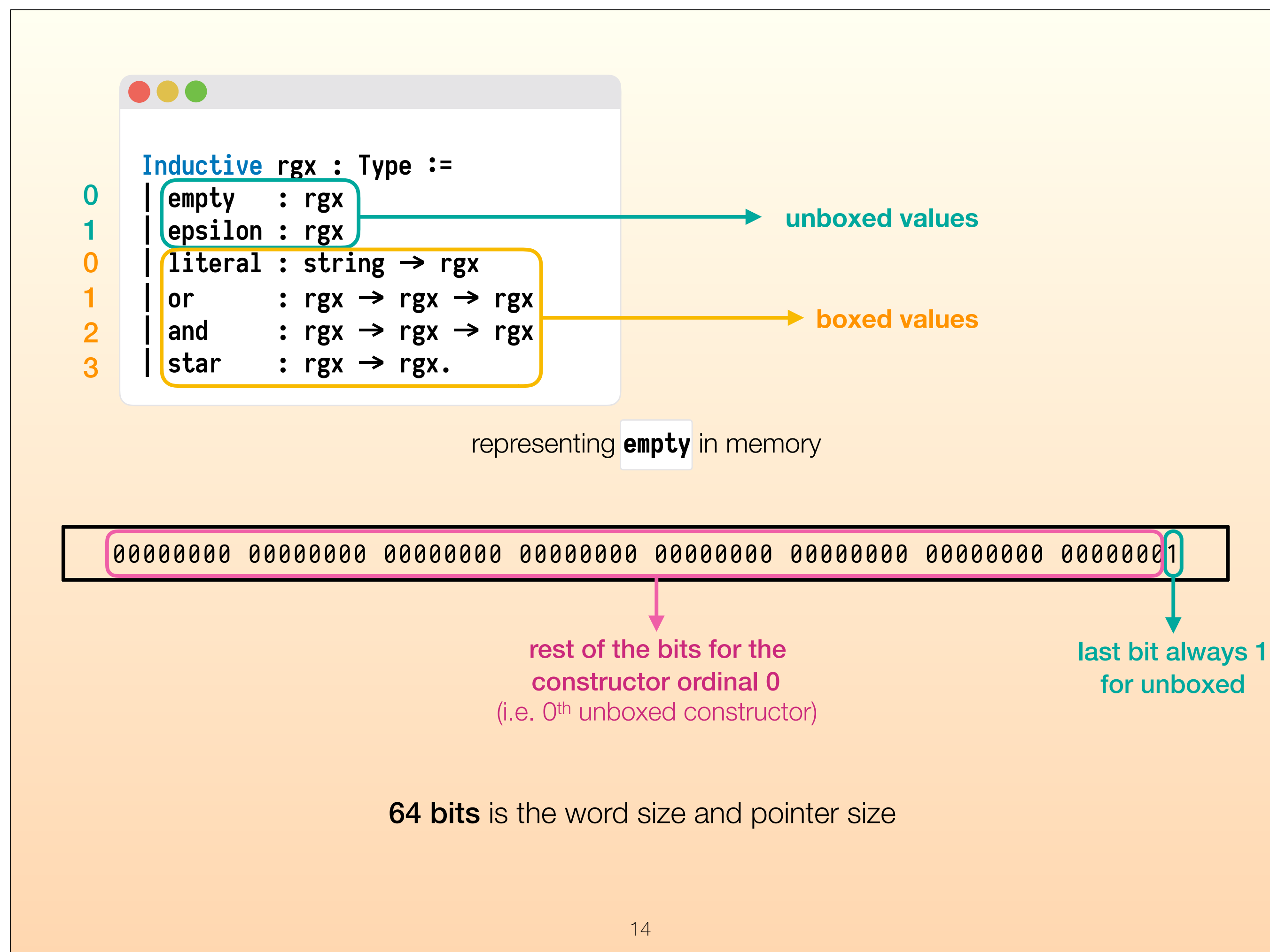
Coq	C
1 inspecting a term's constructor	constructor tag getter
2 inspecting a constructor's arguments	constructor argument structs
3 creating new constructor values	constructor functions
4 calling existing functions	closure caller
5 defining new functions	closure currier

We will generate 5 basic kinds of glue code, each corresponding to an expressive capability of Coq.
<click> The first one is constructor tag getter functions, which will tell us which constructor a value belongs to.
<click> The second one is a way to get the collection of arguments a constructor has taken, ideally in a type that outlines how many.
<click> To create new values, we'll generate a few kinds of functions to put them in different areas in memory. We also have to be mindful of how the garbage collector deals with this.
<click> The other kinds of values in CertiCoq are closures. We need to be able to call existing closures...
<click> ... and create new closures that would fit a given Coq type.

How can we do the same things with Coq values on the C side?

Coq	C
1 inspecting a term's constructor	constructor tag getter
2 inspecting a constructor's arguments	constructor argument structs
3 creating new constructor values	constructor functions
4 calling existing functions	closure caller
5 defining new functions	closure currier

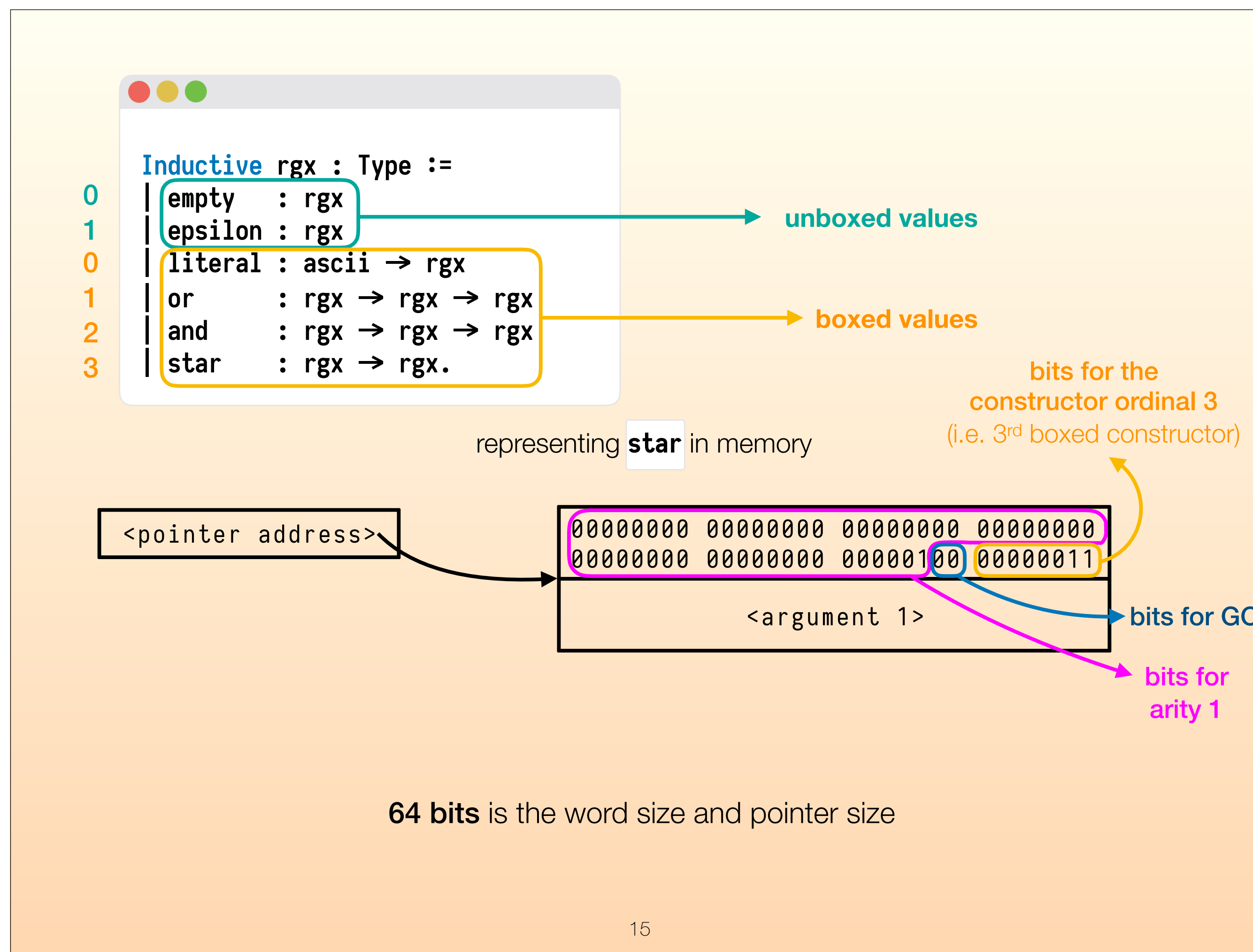
Let's start with the first three, but first we need to understand how CertiCoq represents constructors in memory. So here's some background information.



Here's how we would define the inductive type of regular expressions that we used in the simplify function earlier.

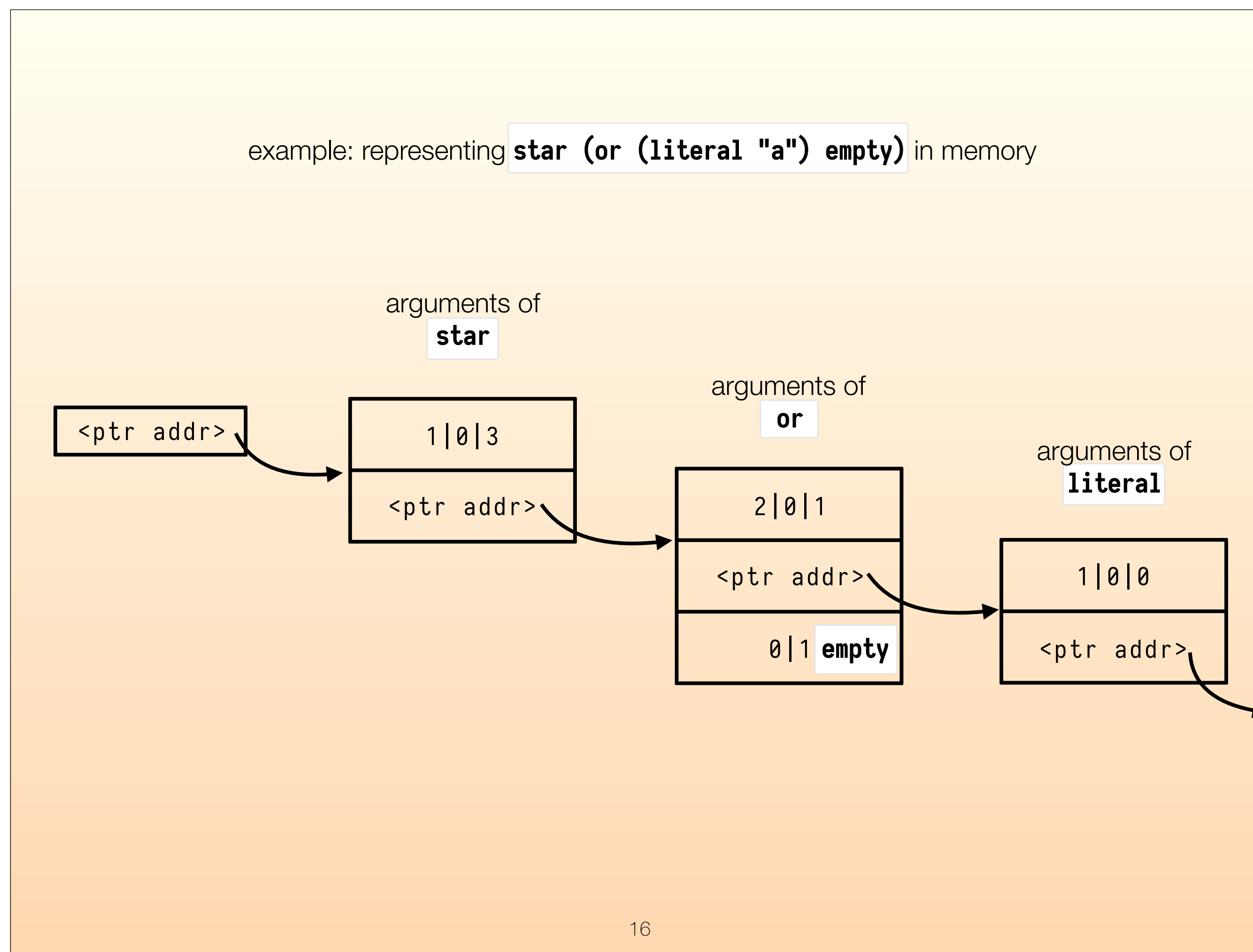
CertiCoq groups constructors of inductive data types into two:
<click> unboxed
<click> and boxed ones.
Boxedness means these values are represented as a pointer to a heap object. In the way CertiCoq (and also OCaml) represent types, nullary constructors, that is, constructors that take 0 arguments, are unboxed. The rest are boxed.

What we have in this slide is the first nullary constructor called "empty".
Now, we will represent this constructor as a single 64-bit integer.
<click> CertiCoq pointer addresses are word aligned, they always end with 0, so to denote non-pointers, we set the last bit to 1.
<click> We use the remaining bits on the left to denote the constructor ordinal. empty is the 0th unboxed constructor, so we just put 0. So in decimal numbers, the integer 1 would represent empty.



Now let's look at a boxed constructor, namely "star". Boxed means the value is represented as a pointer to a heap object. And that object consists of a word for each argument, and a header right before where the pointer points. (and this is important, it will come up often) This header consists of
<click> the constructor ordinal, notice that this is the 3rd *boxed* constructor in the type if you count from 0,
<click> it has two bits for the garbage collector,
<click> and the remaining bits on the left are used to denote the arity of the constructor, that is, how many arguments this constructor takes.

Now let's see how these would be combined.



So have an example regular expression here. Let's see how this would be represented in memory.

The outermost constructor "star" is boxed, so it will be a pointer to a heap object.
<click> This heap object will contain the header for star (written in a different notation here for conciseness), and the argument for it, which is created with the "or" constructor, so that is boxed as well.
<click> The heap object for the "or" constructor takes two arguments, the first one is created with the "literal" constructor, which is also boxed, so that will be a pointer. But the second argument is created with the "empty" constructor, which is unboxed, so it is an integer by itself.
<click> The heap object of the "literal" constructor takes one argument, which points to a string but we won't go deeper than that.

Now we know how Coq constructors are represented in memory, let's see what kind of glue code we generate to deal with them.

① inspecting a term's constructor

generated glue code

```
unsigned int get_unboxed_ordinal(value v)
{
  return v >> 1;
}

unsigned int get_boxed_ordinal(value v)
{
  return *((value *) v - 1) & 255;
}
```

generated glue code

```
unsigned int get_rgx_tag(value v)
{
  if (is_ptr(v)) {
    switch (get_boxed_ordinal(v)) {
      case 0: return 2;
      case 1: return 3;
      case 2: return 4;
      case 3: return 5;
    }
  } else {
    switch (get_unboxed_ordinal(v)) {
      case 0: return 0;
      case 1: return 1;
    }
  }
}
```


To inspect a term's constructor, we generate a few kinds of functions. Two to get the constructor ordinals, both for boxed and unboxed constructors. However, remember that these functions only get the order among just boxed constructors, or just unboxed constructors. Ideally, we would want to isolate our users from this, they shouldn't have to think about that level of detail.

Therefore, we also generate a "get tag" function for each type, as seen on the right for the regular expression type. This function gets the right ordinal, boxed ordinal for the boxed constructor, unboxed ordinal for the unboxed constructor.

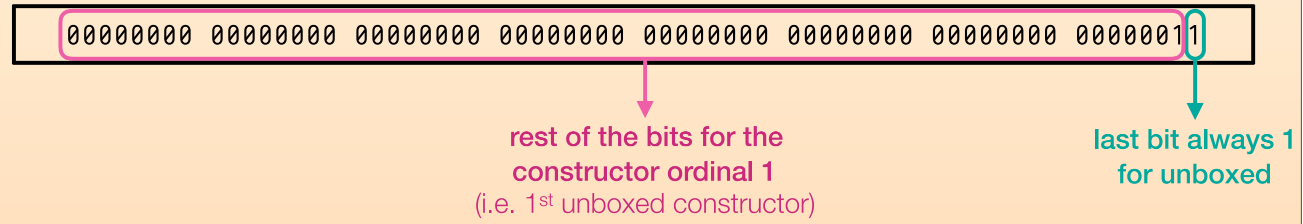
It then maps it to the tag number, starting from 0 for the first constructor, enumerating all constructors of the data type. If a user desires a higher level of abstraction, they can create an enum type in C that contains all the constructors, which would fit perfectly on this unsigned integer. We couldn't generate that because Clight does *not* support enum types.

① inspecting a term's constructor

generated glue code

```
unsigned int get_unboxed_ordinal(value v)
{
  return v >> 1;
}
```

representing **`epsilon`** in memory



64 bits is the word size and pointer size

And if you're curious about how we obtain these ordinals, here it is. For unboxed constructor, we just need to discard the last bit, so we right shift.

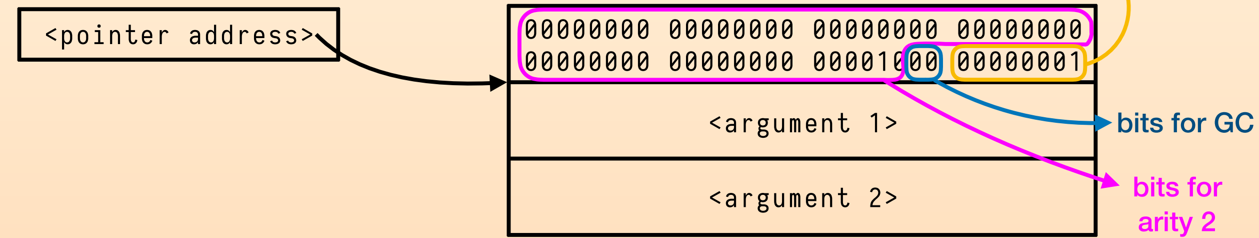
1 inspecting a term's constructor

generated glue code

```
unsigned int get_boxed_ordinal(value v)
{
  return *((value *) v - 1) & 255;
}
```

bits for the
constructor ordinal 1
(i.e. 1st boxed constructor)

representing `or` in memory



64 bits is the word size and pointer size

To get the ordinal from a boxed constructor, we need to follow the pointer, go back one word to find the header, and then discard everything but the last 8 bits.

② inspecting a constructor's arguments

generated glue code

```
struct empty_args {};

struct epsilon_args {};

struct literal_args {
  value literal_arg_0;
};

struct or_args {
  value or_arg_0;
  value or_arg_1;
};

struct and_args {
  value and_arg_0;
  value and_arg_1;
};

struct star_args {
  value star_arg_0;
};
```

generated glue code

```
struct empty_args *get_empty_args(value v)
{
  return (struct empty_args *) 0;
}

...

struct or_args *get_or_args(value v)
{
  return (struct or_args *) v;
}

...
```


But it's not enough to learn what kind of constructor something is, we also want to do something with the arguments of that constructor. Now, we know boxed values are represented as a pointer to a heap object, so we should try to give a type to that heap object. We realized that if we forget about the boxed constructor header, the rest of heap object has the same memory layout as the structs we see on the left. The idea is to have a struct for each constructor, and the struct should have a value for each argument the constructor takes.

Then the getter function for these arguments is just a type cast.
We also have similar functions for unboxed constructors, as you can see in top right, that just return a null pointer. This is just for the sake of completeness, it will not actually be used.

② inspecting a constructor's arguments

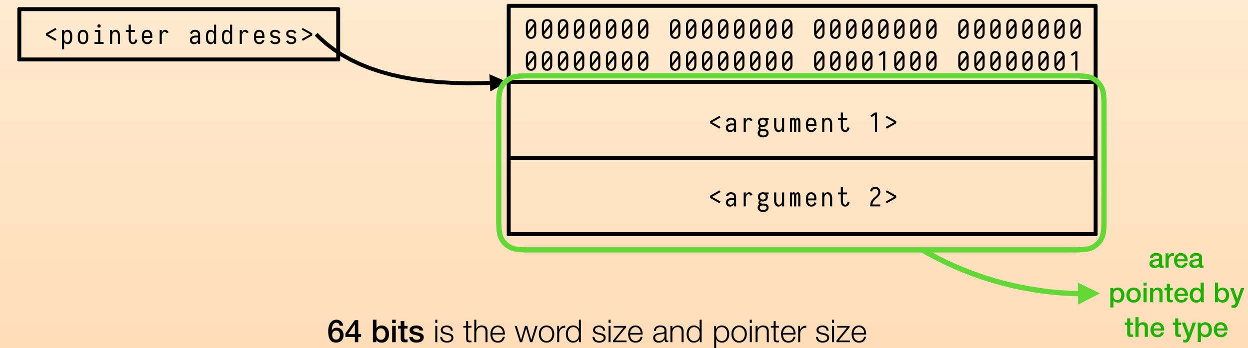
generated glue code

```c
struct or_args {
  value or_arg_0;
  value or_arg_1;
};
```

generated glue code

```c
struct or_args *get_or_args(value v)
{
  return (struct or_args *) v;
}
```

representing `or` in memory



The args struct doesn't include the header, because there is no right C type for the header. The header is technically a 64 bit integer, but then unboxed and boxed headers would have similar types while in reality they are entirely different.

Since the C user would **not** interact with the headers directly but only through the glue code functions, we did not include the header in the struct. We want to isolate the users from these differences through functions that give them the necessary abstractions.

3 creating new constructor values

C heap

generated glue code

```c
value make_rgx_empty(void)
{
  return 1;
}

value make_rgx_epsilon(void)
{
  return 3;
}
```

generated glue code

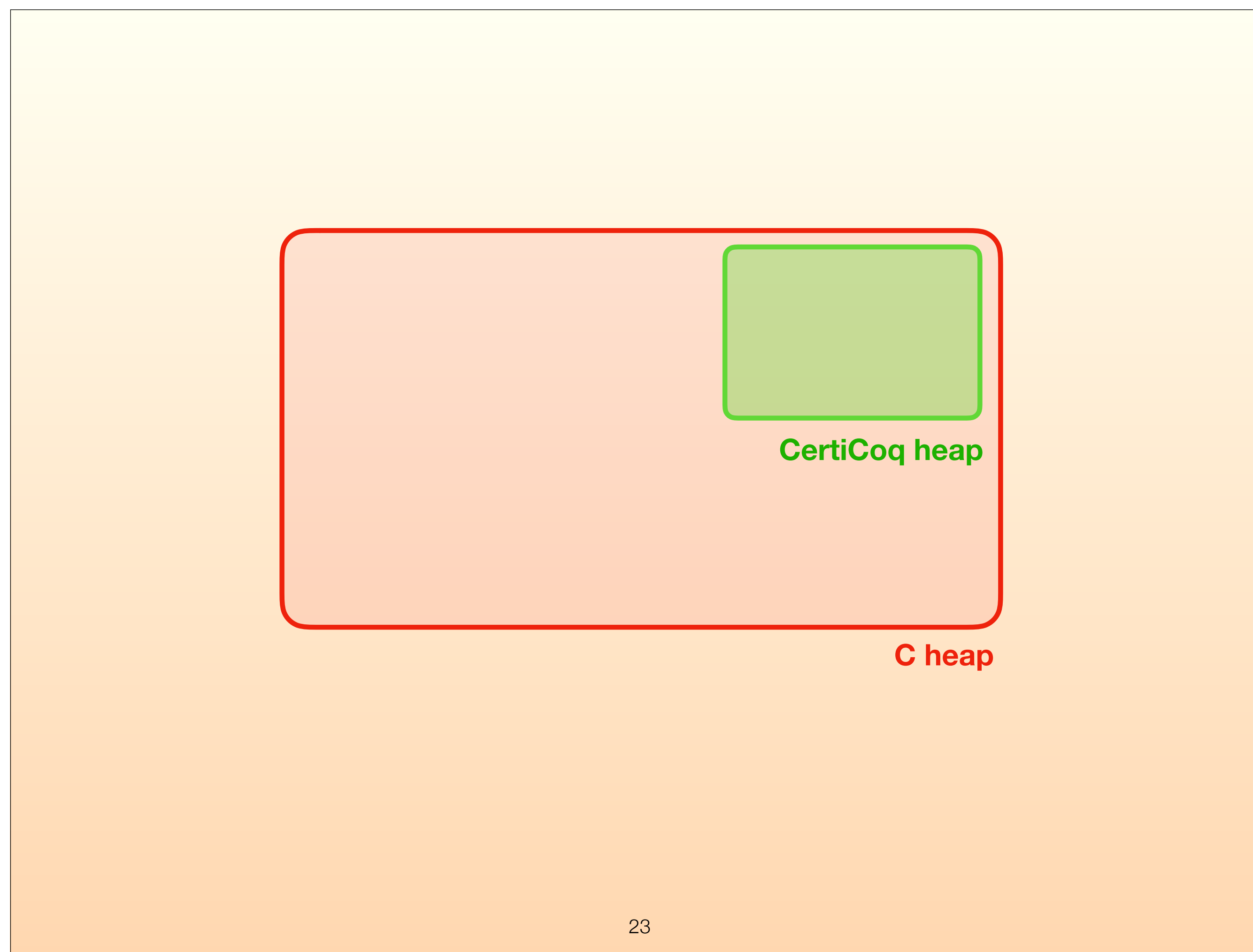
```c
value make_rgx_literal
      (value arg0, value *argv)
{
  *(argv + 0) = 1024;
  *(argv + 1) = arg0;
  return argv + 1;
}

value make_rgx_or
      (value arg0, value arg1,
       value *argv)
{
  *(argv + 0) = 2049;
  *(argv + 1) = arg0;
  *(argv + 2) = arg1;
  return argv + 1;
}

...
```


So now that we know how to use the existing constructor values, let's look at how to create new ones from the C side.

We generate constructor functions to do that.
The first kind is for unboxed constructors, as seen on the left. We only have to return the right integer header for this one.

The second kind is for boxed constructors, as seen on the right. We take a value as input for each argument, and as the last argument, we take the memory address to put the new value. We need this because the boxed value will be a pointer after all. At the given pointer address we put the header, all arguments, and then return a pointer to the first argument.

But something to notice is that, the memory address passed to the constructor should point to memory that's actually available! It's up to the user to allocate the right size of memory to pass into the function. The user also has to remember to free this memory afterwards.



In other words, this constructor that we have just seen can be used to create values in the C-land, the C heap. This is desirable sometimes, but the user has to do the memory management themselves.

So for that reason, we have another variety of boxed constructor functions that automatically allocates memory on the part of heap controlled by CertiCoq!

③ creating new constructor values

CertiCoq heap

generated glue code

```
value alloc_make_rgx_literal(struct thread_info *tinfo, value arg0)
{
  value *argv = tinfo->alloc;
  *(argv + 0) = 1024;
  *(argv + 1) = arg0;
  tinfo->alloc += 2;
  return argv + 1;
}

value alloc_make_rgx_or(struct thread_info *tinfo, value arg0, value arg1)
{
  value *argv = tinfo->alloc;
  *(argv + 0) = 2049;
  *(argv + 1) = arg0;
  *(argv + 2) = arg1;
  tinfo->alloc += 2;
  return argv + 1;
}

...
```


Here's what it would look like to create values in the CertiCoq-land.

<click> In the thread info, we have address of the next available memory in CertiCoq heap. We create the new value there, and update the info afterwards.

However, this function is still work in progress. First of all, we need to check if we got to the end of the CertiCoq controlled heap. The other concern is that the value we create here is subject to the garbage collector, and we have no way of reporting to GC not to free this memory just yet because we're using it. The work to alleviate these concerns is ongoing, in collaboration with Aquinas Hobor and his students, and Kathrin Stark, who is a postdoc here.

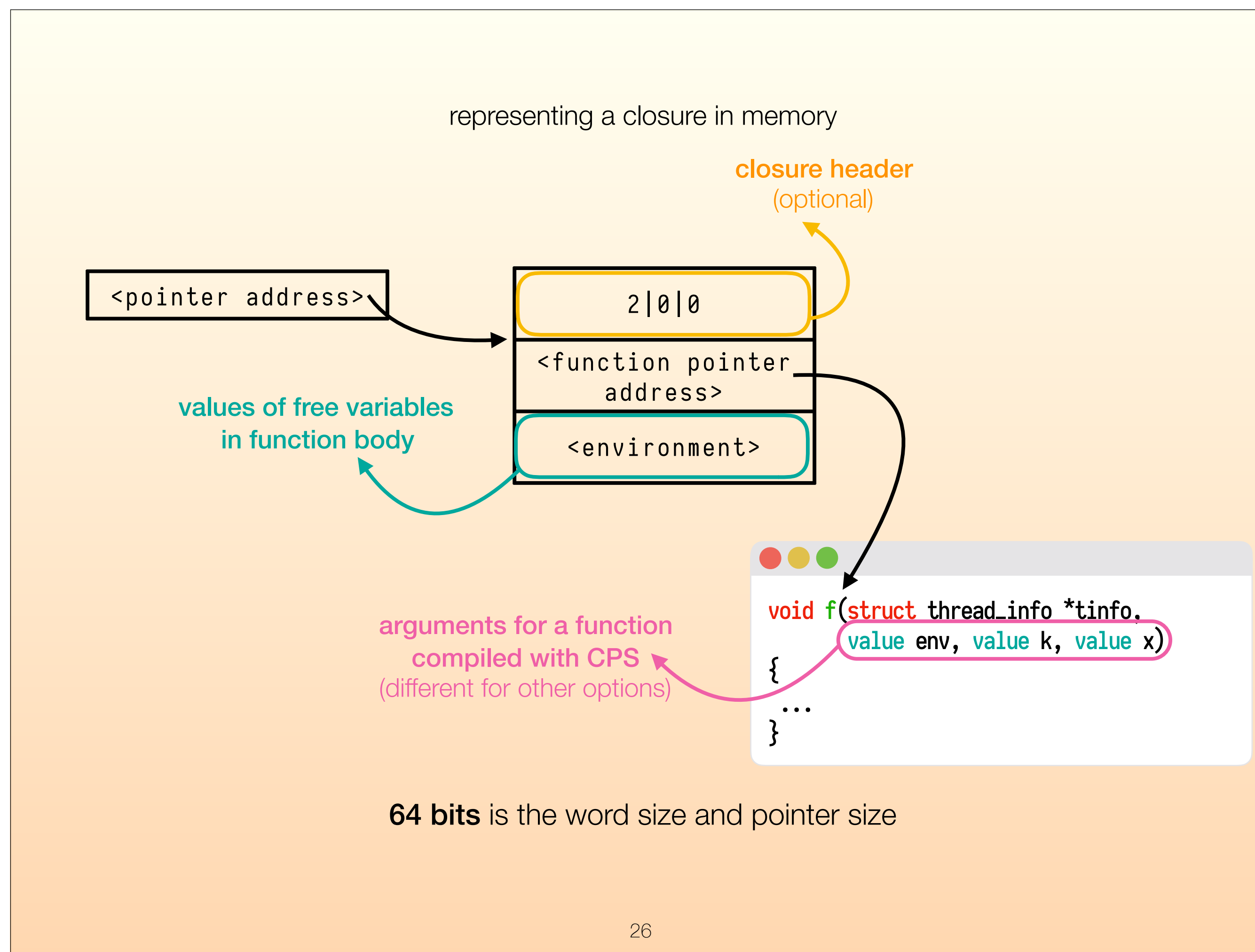
How can we do the same things with Coq values on the C side?

Coq	C
1 inspecting a term's constructor	constructor tag getter
2 inspecting a constructor's arguments	constructor argument structs
3 creating new constructor values	constructor functions
4 calling existing functions	closure caller
5 defining new functions	closure currier

Now we know how to deal with constructors, but since we're dealing with a functional language, dealing with functions is a lot more interesting to us.

These last two are a bit more complex because of how closures are represented in Coq, and because our compiler options can change closures representations. We want to isolate users from these. That is what I mean by ergonomics, the glue code should be easy to use for the human programmer, and that means not having to know all the details about memory representations, because they can be very specialized.

Let's look at the last two expressive capabilities, but for these we need to know how CertiCoq represents functions in memory.



A function is kept in memory as a closure, which points a heap object,
<click> first word of which points to an actual C function with certain arguments that Coq knows how to call. In this example we see what those arguments look like for the default compiler options. For continuation-passing style, this function takes in an environment, a continuation closure, and the actual argument to this closure.

<click> The second word of the heap object is a list of values for the free variables in the function. I am not planning to explain in depth how CPS and closures work here, but in short, when we call the function f, we need to pass this environment in, a continuation that states what to do after.

<click> and finally we have a closure header, but that is optional since we never inspect it.

④ calling existing functions

C glue code generated for function calls

```
void halt(struct thread_info *tinfo, value env, value arg)
{
  tinfo->args[1] = arg;
  return;
}

value const halt_clo[2] = { &halt, 1 };

value call(struct thread_info *tinfo, value clo, value arg)
{
  value *f    = *((value *) clo + 0);
  value *envi = *((value *) clo + 1);
  ((void (*)(struct thread_info *, value, value, value))
     f)(tinfo, envi, halt_clo, arg);
  return tinfo->args[1];
}
```


Here's what it looks like to call a closure from the C side. We have a call function that takes a closure and an argument to call it with. Remember that a closure is a heap object, so we extract the function and the environment from that object.

Then we call that function with that environment, a "halt" closure, which just says put whatever result you have at the end at a place I can find it, and the actual argument to the function. We go get that result and return it.

You might wonder why we have to generate this function...

4 calling existing functions

compiled with C args = 0

C glue code generated for function calls

```
void halt(struct thread_info *tinfo)
{
  return;
}

value const halt_clo[2] = { &halt, 1 };

value call(struct thread_info *tinfo, value clo, value arg)
{
  value *f    = *((value *) clo + 0);
  value *envi = *((value *) clo + 1);
  tinfo->args[0] = envi;
  tinfo->args[0] = halt_clo;
  tinfo->args[0] = arg;
 ((void (*)(struct thread_info *)) f)(tinfo);
  return tinfo->args[1];
}
```


However, this function can take many different forms according to the compiler options. Here's what it looks like if we use the compiler setting that says CertiCoq C functions should take 0 arguments other than the thread info. This has some performance implications that Matthew Weaver is looking into.

And...

4 calling existing functions

compiled with ANF

C glue code generated for function calls

```
void halt(struct thread_info *tinfo, value env, value arg)
{
  tinfo->args[1] = arg;
  return;
}

value const halt_clo[2] = { &halt, 1 };

value call(struct thread_info *tinfo, value clo, value arg)
{
  value *f    = *((value *) clo + 0);
  value *envi = *((value *) clo + 1);
  ((void (*)(struct thread_info *, value, value)) f)(tinfo, envi, arg);
  return tinfo->args[1];
}
```


Here's what it looks like if you choose to use the A-normal form backend, instead of the continuation-passing style one. The new backend is a feature that Zoe Paraskevopoulou is working on, so glue code support for their closures is still experimental.

However, our glue code generator can take these options into account and can mold glue functions in different shapes to fit the options.
I should note that the generation of the call functions is joint work with Kathrin Stark.

5 defining new functions

user's Coq code

```
Class RegexFFI :=
  { test : rgx → string → bool
  ; exec : rgx → string → option string
  }.

Definition prog `{RegexFFI} : bool :=
  test (star (literal "a")) "aaa".

CertiCoq FFI RegexFFI.
CertiCoq Compile prog.
```

collection of foreign functions we want to use in a program

using the type class as an argument

a new vernacular command for generating closures

And finally, the most important part of any foreign function interface, defining new functions for Coq in C.
This is the most complex part so far, because of a few reasons.

One of the reasons is that the way we pass C functions to Coq programs is actually just argument passing. In other languages this is often done at a module level. CertiCoq cannot handle Coq modules yet so we do it on a term level here.

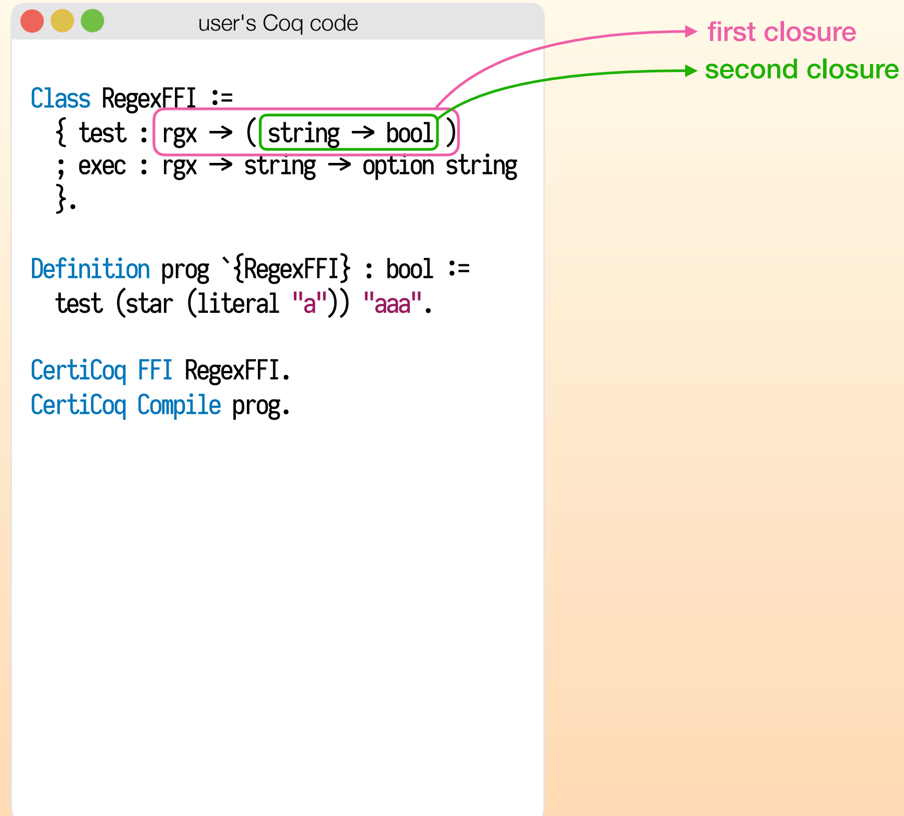
<click> Since we don't have modules to bundle Coq functions together, we use Coq's type classes, which are just dependently typed records. If you are not familiar with type classes, they are very similar to interfaces in Java or traits in Scala and Rust.

The user is supposed to bundle together the types of the functions they're going to implement in C in a new type class. And any function that uses any of these functions is supposed to require this type class to be implemented, which in reality is just a new argument to that function.

<click> For example, in the "prog" example, the regular expression FFI type class is an *argument*, we when we run "prog" on the C side we will have to pass the collection of these functions. So how will we generate those functions?

<click> We have a new vernacular command for that that takes in the type class we defined above. Here's what it does:

5 defining new functions



For each function in the type class, it generates the necessary closures. Let's do the first one as an example.
Notice that this function takes two arguments in a curried way.
<click> So this is actually a function that returns a function
<click> and a second function that returns the whole result. We will have to generate two closures for this one. And as you have functions that take in more arguments, this becomes a lot more complex. So our new vernacular command fixes that.

⑤ defining new functions

`test : rgx → string → bool`

C glue code generated for functions

```
extern value test(struct thread_info *, value, value);
```

→ function to be filled by the user

```
void test_2nd_fn(struct thread_info *tinfo,
                 value env, value k, value arg1)
{
  ... // extract arg0 from env
  value result = test(tinfo, arg0, arg1);
  ... // "return" the result of a call test
}
```

→ function for the second closure

```
void test_1st_fn(struct thread_info *tinfo,
                 value env, value k, value arg0)
{
  ... // add arg0 to the env
  ... // "return" the second closure
}
```

→ function for the first closure

```
value test_clo[2] = { &test_1st_fn, 1 };
```

→ first closure

The new vernacular command, generates a file like this.
<click> It generates the first closure, which points to a function,
<click> which takes the arguments required by our compiler options. It adds the first argument to the environment, which should be used by the second closure later. The second closure is created dynamically,
<click> and it points to the second function, which extracts the earlier arguments from the environment, and calls...
<click> the external function "test". Notice that the type of the test function is a lot easier to grasp than the other ones. It takes the thread info, which is a given at this point, then two values, one for the regular expression and one for the string. Notice that these are the C representations of Coq values. But as I said, "test" is an external function that the user is supposed to implement...

5 defining new functions

test : rgx → string → bool

C code written by the user

```
#include <pcre.h>

char *string_value_to_string(value s); // elided
char *regex_value_to_pcre_string(value r); // elided

value test(struct thread_info * tinfo, value r, value s)
{
  char *rs = regex_value_to_pcre_string(r);
  char *matched = string_value_to_string(s);

  ...
  pcre *re = pcre_compile(...);
  int rc = pcre_exec(...);

  free(rs);
  free(matched);

  return (rc ≥ 0) ? make_bool_true() : make_bool_false();
}
```

user written functions

And here's what it would look like to implement "test".

<click> We would depend on two functions, one to convert the C representation of a Coq string to a C string, and one to convert a regular expression into a C string containing the regular expression syntax we use for the PCRE library in C, a commonly used regular expression library in C.

<click> We can then call those functions to get C strings, which we can use with PCRE.

5 defining new functions

user's Coq code

```
Class RegexFFI :=
  { test : rgx → string → bool
  ; exec : rgx → string → option string
  }.

Definition prog `{RegexFFI} : bool :=
  test (star (literal "a")) "aaa".

CertiCoq FFI RegexFFI.
CertiCoq Compile prog.
```

user's C code

```
int main()
{
  struct thread_info* tinfo =
    make_tinfo();
  value prog = body(tinfo);

  value regex_ffi =
    alloc_make_RegexFFI_Build_RegexFFI(
        tinfo,
        test_clo,
        exec_clo);

  value v = call(tinfo, prog, regex_ffi);

  print_bool(v);

  return 0;
}
```

constructing the collection of closures

passing the foreign functions into the program

Once we define the "test" function, we will now have a test closure fully defined, thanks to the glue code we generated for the type class. But we still need to create an implementation of this type class.

<click> which is what we do here. Type classes are records and records are inductive types, so we will use the constructor function for this type class, which will take the closures we generated earlier. Now we have bundled the foreign functions together.

<click> Once we do that, we can pass that bundle into our Coq program, this will run the actual program and give us a result, in this case a boolean value that we can print.

By the way, we haven't talked much about the print functions today but my glue code generator can generate these print functions in C for most Coq types, including recursive, polymorphic or parametrized ones.

So, this FFI is nice and dandy, but there's something we haven't discussed yet! A lot of C functions are effectful! What do we then?

You might have seen this comic from xkcd, it has been around for while.
it'd be even more accurate to say the same for Coq,
<click> because Coq really doesn't have any effectful functions built into the language.

C functions, however, don't just do some computation, they can also print stuff to the screen, change or read files, launch missiles etc. A big reason we want a C FFI in the first place is to achieve the same in Coq in a controlled environment. We want to have our closure generator to handle this case as well. But we don't want to do them as "side effects", like in OCaml. We want a monadic effect type.

Now, let's look at an example that defines this monadic type and a simple effectful program.

5 defining new functions

user's Coq code

```
Class IO_Types : Type :=
  { IO : Type → Type }.

Class IO_Impl `{IO_Types} : Type :=
  { io_ret : forall (A : Type), A → IO A
  ; io_bind : forall (A B : Type),
              IO A → (A → IO B) → IO B
  }.

Class StringFFI `{IO_Impl} :=
  { print_string : string → IO unit
  ; scan_string : IO string
  }.

Definition prog `{StringFFI} : IO unit :=
  print_string "What's your name?" ;;
  name ← scan_string ;;
  print_string ("Hello, " ++ name ++ "!").

CertiCoq FFI IO_Impl, StringFFI.
CertiCoq Compile prog.
```

the description of IO
imagine:

```
Definition IO A : World → A * World.
```

but implemented in C
so that it is opaque in Coq.

collection of effectful functions
we want to use in a program

using the type class as an argument

the new vernacular command
for generating closures

This is what the entire program looks like. Some of this should be defined in a library but the code here is self-contained, with the exception of some notations.

This program is supposed to ask the user in terminal what their name is, get the terminal input, and then say hello using their name, we can see that in the prog function at the end. Now let's see what the types of these things are.

<click> IO is how we define effectful actions. You might be familiar with the IO type from Haskell. These are *not* side effects, effects don't happen until we execute an IO action. And execution will only happen when we say execute on the C side.
We don't define IO in Coq, we leave it as an argument to everything and define it in C later. But the C definition corresponds to what you see here on the right, we define it as if it's a state monad, but that's not visible from the Coq side. This is to keep IO actions opaque to Coq! If we defined it as a function in pure Coq, then the users would be able to pry into these effectful actions and cause side effects. We don't want that.
The second type class you see here, the one that defined io_ret and io_bind, is for describing how monadic actions are composed. Return and bind functions suffice to define a monad, but I won't get deeper than that here. We have to implement these functions in C as well, because the IO type is opaque to Coq.

<click> Now we can define the collection of effectful functions we want to use. Here we have one to print a Coq string in terminal, and take a Coq string from the terminal.
<click> Same as before, our program takes this FFI collection as an input.
<click> When we run the new vernacular command at the end, the command takes IO into account, and makes sure that the effects don't run until the action is executed. It ensures that we don't have side effects.

5 defining new functions

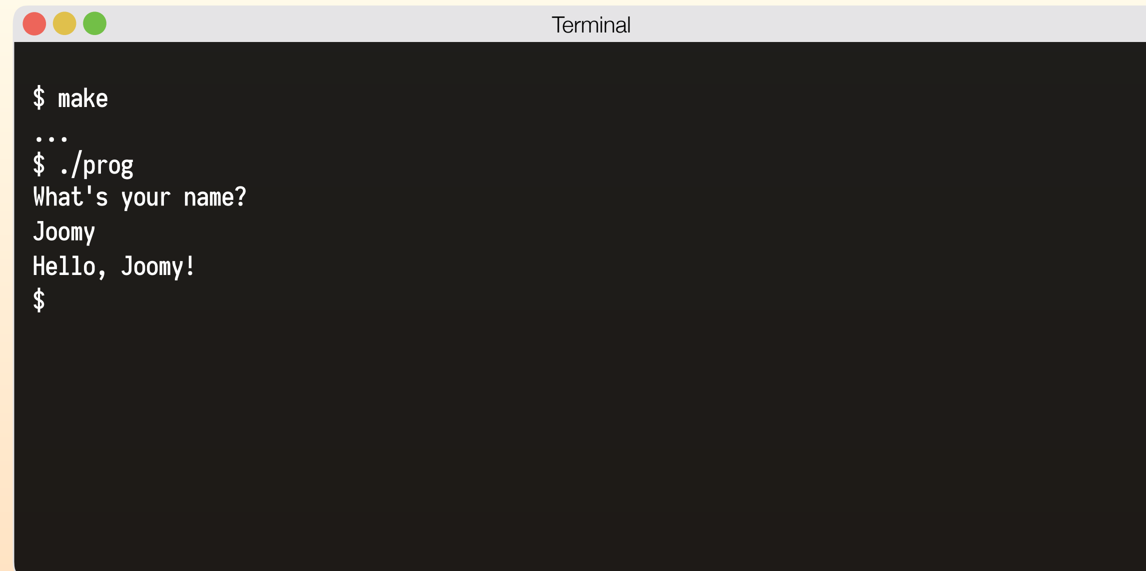
```
print_string : string → IO unit
scan_string : IO string
```

C code written by the user

```c
value print_string(struct thread_info * tinfo, value s)
{
  char *s = string_value_to_string(s);
  puts(s); free(s);
  return make_unit_tt();
}

value scan_string(struct thread_info * tinfo)
{
  char *input = (char *) malloc(100 * sizeof(char));
  scanf("%s", input);
  return string_to_string_value(tinfo, input);
}

int main()
{
  ...
  value string_ffi =
    alloc_make_CertiCoq_Benchmarks_io_io_StringFFI_Build_StringFFI(
        tinfo, print_string_clo, scan_string_clo);
  ...
}
```


Here is how the user would define the string FFI functions in C. The print function, as one would expect, takes a Coq string in, converts it to a C string, and prints it. Then it returns the C representation of a Coq unit. This should be familiar to OCaml users as well. Notice that the user is isolated from the monadic nature of IO as much as possible here, the abstraction is handled by the generated code.

⑤ defining new functions



Once we complete all the functions and compile all the C files, we can have an executable that indeed, does what we want.
<click> ... <click> ...

Okay, now onto verification.

⋆ verification of glue functions

VST specs in Coq

```
Definition function_name_spec :=
  DECLARE _function_name
    WITH ...
    PRE [ ... ]
      PROP  ( ... )
      LOCAL ( ... )
      SEP   ( ... )
    POST [ ... ]
      PROP  ( ... )
      LOCAL ( ... )
      SEP   ( ... ).
```

PROP	Coq propositions
LOCAL	variable bindings and addresses
SEP	spatial assertions in separation logic

(for the old version of VST)

We want to write specifications for our glue functions in separation logic, which is a modern version of Hoare logic. Similar to Hoare logic we have pre and post conditions for imperative programs, as you can see in the PRE and POST notation here in the code. What's special about separation logic is that it lets us reason about different parts of the memory separately.

The way VST embeds separation logic requires our specifications to have three parts, one for Coq propositions, one for variable bindings and addresses, and one for spatial assertions in separation logic.

Here's what the simplest VST specification would look like for some of the glue code functions we have.

verification of glue functions - future work!

generated C glue code

```
value make_rgx_empty(void)
{
  return 1;
}

value make_rgx_epsilon(void)
{
  return 3;
}
```

generating VST specs in Coq

```
Definition make_rgx_empty_spec :=
  DECLARE _make_rgx_empty
    WITH _ : unit
    PRE [ ]
      PROP () LOCAL () SEP ()
    POST [ tulong ]
      PROP  ()
      LOCAL (temp ret_temp v)
      SEP   (!!(v = Vlong (Int64.repr 1))).

Definition make_rgx_epsilon_spec :=
  DECLARE _make_rgx_epsilon
    WITH _ : unit
    PRE [ ]
      PROP () LOCAL () SEP ()
    POST [ tulong ]
      PROP  ()
      LOCAL (temp ret_temp v)
      SEP   (!!(v = Vlong (Int64.repr 3))).
```


I'm just gonna talk about constructor functions for now. Constructor functions for unboxed constructors are relatively easy because they don't write anything in memory, they merely return an integer. Here in the specifications we say that the return value of the function is equal to 1 for empty and 3 for epsilon, which are the unboxed constructor headers we saw earlier, they were in binary there but they're in decimal here.

⋆ verification of glue functions - future work!

generated C glue code

```c
value make_rgx_literal
      (value arg0, value *argv)
{
  *(argv + 0) = 1024;
  *(argv + 1) = arg0;
  return argv + 1;
}
```

generating VST specs in Coq

```
Definition make_rgx_literal_spec :=
  DECLARE _make_rgx_literal
    WITH arg0 : val, p : val,
         s : string
    PRE [ _arg0 OF tulong ,
          _argv OF tptr tulong]
      PROP  ()
      LOCAL (temp _arg0 arg0 ;
             temp _argv p)
      SEP   (coq_string_rep s arg0);
             data_at_ Tsh (tarray tulong 2) p)
    POST [ tulong ]
     EX v : val,
      PROP  ()
      LOCAL (temp ret_temp (offset_val 8 p))
      SEP
        (data_at Tsh (tarray tulong 2)
            [Vlong (Int64.repr 1024); v] p ;
         coq_string_rep s v).
```

argument is a valid Coq string

available memory

pointer to one word after

When we get to the boxed constructor functions, things get more interesting.
Now we take an argument, arg0, that represents a Coq string.
<click> A precondition for our function should be that arg0 should be a value that represents a Coq string.
And the second argument, namely argv ...
<click> should be a pointer to some available memory, that's the other precondition that we have.
<click> At the end, we should return a pointer to one word after the initial pointer address, having the first argument pointing to the Coq string, and having placed the header 1024 in the right location.

⋆ verification of glue functions - future work!

generating VST specs in Coq

```
Fixpoint coq_rgx_rep (r : rgx) (x : val) : mpred :=
  match r, x with
  | empty  , Vlong i ⇒ !! (x = Vlong (Int64.repr 1))
  | epsilon, Vlong i ⇒ !! (x = Vlong (Int64.repr 3))
  | literal s, Vptr b z ⇒
      data_at Tsh tuint (Vlong (Int64.repr 1024)) (offset_val 8 x)
    * coq_string_rep s (Vptr b z)
  | or r1 r2, Vptr b z ⇒
      data_at Tsh tuint (Vlong (Int64.repr 2049)) (offset_val (-8) x)
    * coq_rgx_rep r1 x
    * coq_rgx_rep r2 (offset_val 8 x)
  | and r1 r2, Vptr b z ⇒
      data_at Tsh tuint (Vlong (Int64.repr 2050)) (offset_val (-8) x)
    * coq_rgx_rep r1 x
    * coq_rgx_rep r2 (offset_val 8 x)
  | star r', Vptr b z ⇒
      data_at Tsh tuint (Vlong (Int64.repr 1027)) (offset_val (-8) x)
    * coq_rgx_rep r' x
  | _ , _ ⇒ !! False
  end.
```


One thing that we want to do eventually is to automatically generate spatial predicates for all types, this will provide a higher level of abstraction when the users are dealing with proofs about their C programs manipulating Coq values. What we have on the screen is the simplest form of what that would look like, for tree-like structures. In practice, CertiCoq data is a directed acyclic graph rather than a tree, but this should give you an idea for now.

Clearly, there's a long way to go about VST specifications of glue code. We still have to deal with closures, the garbage collector, and the bigger picture on the proof side of all this. Kathrin, other collaborators and I are already discussing these issues so we'll have a better story on those later.

Comparisons with related work

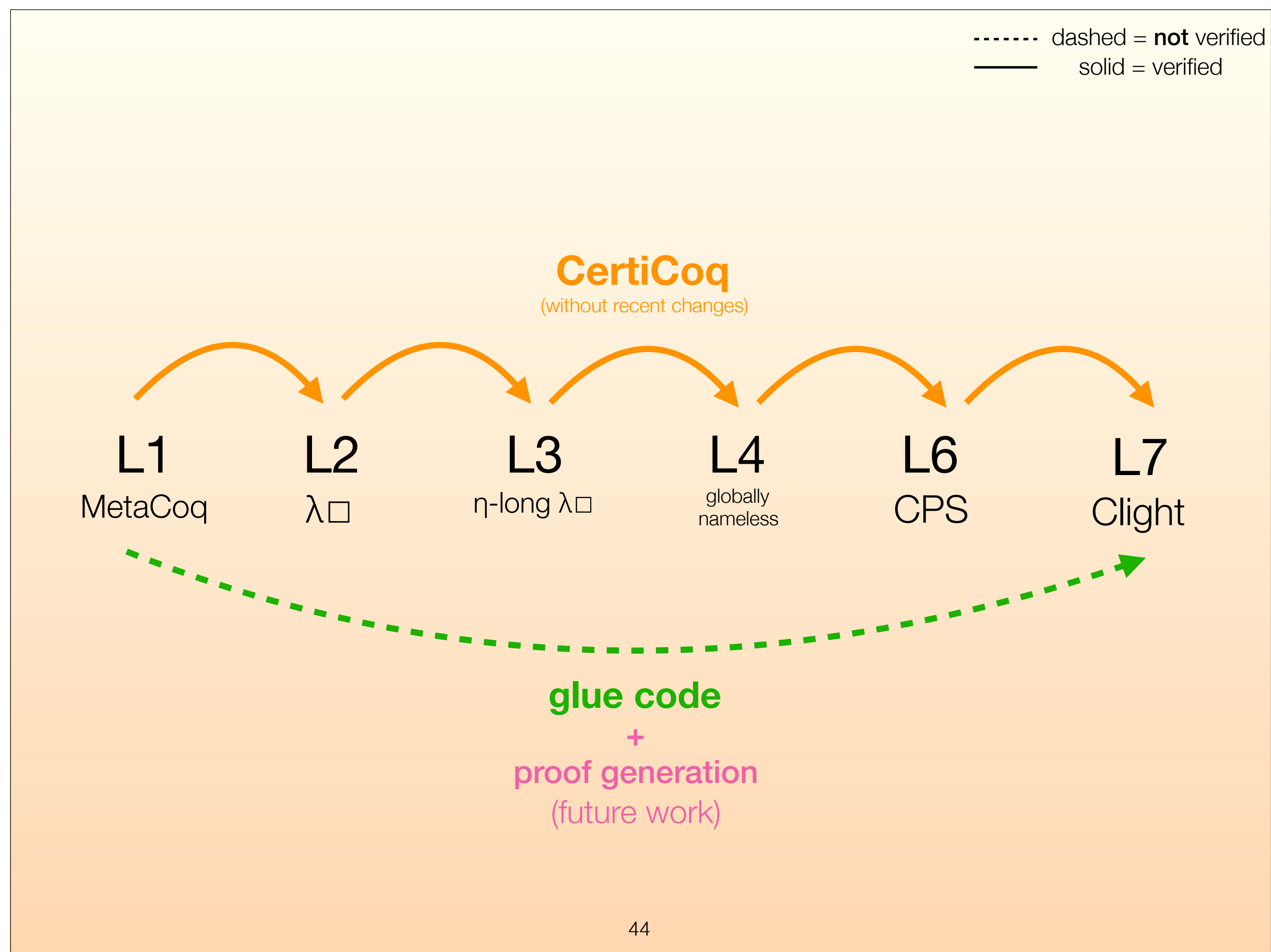
- *No-longer-foreign: Teaching an ML compiler to speak C "natively".*
  Matthias Blume. 2001.
  - encoding C types in ML types, writing imperative C code in ML
  - glue code generator that **takes** C types and **outputs** ML
- *Checking type safety of foreign function calls.*
  Michael Furr, Jeffrey S. Foster. 2005.
  - a **multilingual type system**: accounting for both OCaml and C types, and the GC at the same time
  - dataflow analysis for checking if C FFI programs fit these types
- *Œuf: Minimizing the Coq Extraction TCB.*
  Mullen, Pernsteiner, Wilcox, Tatlock, Grossman. 2018
  - very **limited subset** of Gallina
  - exposes correctness theorem to their limited glue code

In my reading list I had 3 FFI papers, one about Standard ML, one about OCaml, and one about Coq. The first two were trying to solve different problems, the last one was more relevant.

<click> The first paper by Matthias Blume was trying to allow users to write most of their imperative code in ML, the human programmer would only write their structs and unions in C and write and see the rest in Standard ML. We don't want that because we have the separation logic tools for C that we want to take advantage of, the programmer should eventually be able to reason about their C programs using VST.

<click> The second paper by Furr and Foster is closer to our goal here. They worked on a multilingual type system that captured how OCaml values are represented in memory (and OCaml representation is the same as CertiCoq representation), and used dataflow analysis to check it. They have certain shortcomings like polymorphic functions, but overall it's a cool idea. That being said, for CertiCoq we want to take a different direction. First, we can express the same things using separation logic, and we can automate their checking with tactics in the future. And those separation logic proofs are more expressive than just type checking, we can give more accurate specifications about what the FFI functions do.

<click> The last paper is from another verified compiler project from Coq to C, called Oeuf, ... pardon my French. Their system can compile a much smaller subset of Gallina, with no user defined types, no dependent types, no fixpoints, just eliminators. They also don't have any effect system like I showed you today. They claim that they expose their correctness theorem to their glue code, but I'm not clear about what they mean by that, and I looked at their repo too but couldn't exactly figure that out.



To summarize, I worked on the foreign function interface for Coq, which means a way for Coq to call C functions and C to call Coq functions. For that goal, I added a glue code generator that takes Coq and generates C code.

There is a bunch of other stuff that I didn't have time to show today, such as print functions, or the polymorphic mutable hash table data structure I implemented using this FFI.

The next step in this project is generating proofs for this glue code using VST's Verifiable C, which allows us to prove specifications in separation logic about C programs.

<click> Thanks for listening, I'm happy to take questions.