An application: foreign function bindings

```c
int puts(const char *s);
```
C in two minutes

**object types**

numeric types

*int, char, float, ...*

pointers

*int *, char *, int **, ...*

structures and unions

```c
struct t { int x, char y };
```

arrays

```c
int x[3] = { 1, 2, 3};
```

**function types**

built from object types

*n arguments, one return type*

```c
int(const char *);
char *(char *, char *);
```

**Operations**

address, sizeof, read, write, ...

**Operations**

call
Talking to C: two challenges

Conversions between value representations

Long_val, Val_long, caml_copy_double, ...

Interactions with the garbage collector

Protect locals & parameters against disappearance & destruction

```c

value puts_stub(value s)
{
    CAMLparam1(s);
    const char *p = String_val(s);
    int n = puts(p);
    CAMLreturn(Val_int(n));
}
```
Representing types
Representing object types

### C object types

```
type ::= int char type *
...```

### Representing C object types

```
type _ typ =
    Int : int typ
| Char : char typ
| Ptr : 'a typ \rightarrow 'a ptr typ
| ...
| View : ('a \rightarrow 'b)
    * ('b \rightarrow 'a)
    * 'a typ \rightarrow 'b typ
| ...
```

```ml
let string : string typ =
    View (ptr_of_string, string_of_ptr, PTR Char)
```
Operations on object types

Low-level operations

val read : 'a typ → address → 'a
val write : 'a typ → 'a → address → unit
val sizeof : 'a typ → int

let read : type a. a typ → address → a =
  fun typ addr → match typ with
  | Int → read_int address
  | Char → read_char address
  | ...  

Higher-level operations

type 'a ptr (* = 'a typ * address *)
val (!@) : 'a ptr → 'a
val (@+) : 'a ptr → int → 'a ptr
Representing function types

C function types

ftype ::= type(type, type, ..., type)

Representing C function types

(type _ fn = Returns: 'a typ \to 'a fn
    | Function: 'a typ * 'b fn \to ('a \to 'b) fn

let (@\to) a b = Function (a,b) and returning v = Returns v

Example

Ptr Char @\to Int @\to returning Int

represents

int(char *, int)
Operations on function types

\[
\text{val foreign : string } \rightarrow (\text{'a } \rightarrow \text{'b}) \text{ fn } \rightarrow (\text{'a } \rightarrow \text{'b})
\]

Example

\[
\text{let puts = foreign "puts" (string } \circ \rightarrow \text{ returning int)}
\]

\text{produces}

\[
\text{val puts : string } \rightarrow \text{ int}
\]
Anatomy of a binding

```
let puts = foreign "puts" (string @→ returning int)
    puts "Hello, world"
```

1. resolve the name
2. create a buffer with enough space
3. convert and write arguments
4. apply function
5. read results
Anatomy of a binding

```
let puts = foreign "puts" (string @→ returning int)  
    puts "Hello, world"
```

1. resolve the name  "puts" ↦ 0x7f0d1eebcf60
2. create a buffer with enough space
3. convert and write arguments
4. apply function
5. read results
Anatomy of a binding

```
let puts = foreign "puts" (string @→ returning int)
   puts "Hello, world"
```

1. resolve the name  "puts" ↦ 0x7f0d1eebcf60
2. create a buffer with enough space
3. convert and write arguments
4. apply function
5. read results
Anatomy of a binding

```ocaml
let puts = foreign "puts" (string @→ returning int) puts "Hello, world"
```

1. resolve the name "puts" ↝ 0x7f0d1eebcf60

2. create a buffer with enough space

3. convert and write arguments

4. apply function

5. read results
Anatomy of a binding

```ocaml
let puts = foreign "puts" (string @→ returning int)

puts "Hello, world"
```

1. resolve the name  "puts" ↝ 0x7f0d1eebcf60
2. create a buffer with enough space
3. convert and write arguments
4. apply function
5. read results
Anatomy of a binding

let puts = foreign "puts" (string @→ returning int)
  puts "Hello, world"

1. resolve the name  "puts" ~→ 0x7f0d1eeb6f0
2. create a buffer with enough space
3. convert and write arguments
4. apply function
5. read results        puts "Hello, world" ~→ 13
More type interpretations
Drawbacks of dynamism

No type safety

Name lookup may fail dynamically

Interpretive overhead

Can’t use standard tools (nm, objdump, ldd, …)
module Bindings(F: FOREIGN) = struct
  open F
  let puts = foreign "puts" (string @→ returning int)
end

value puts_stub(value s) {
  char *p = Ptr_val(s);
  int n = puts(p);
  return Val_int(n);
}

external puts_stub : address → int = "puts_stub"

let foreign nm fn = match nm, fn with
  | "puts", Function (View ...
Staged binding

module Bindings(F: FOREIGN) = struct
  open F
  let puts = foreign "puts" (string @→ returning int)
end

value puts_stub(value s) {
  char *p = Ptr_val(s);
  int n = puts(p);
  return Val_int(n);
}

external puts_stub : address → int = "puts_stub"

let foreign nm fn = match nm, fn with
  | "puts", Function (View ...
module type FOREIGN = sig
  type _ result
  val foreign: string \to ('a \to 'b) \to ('a \to 'b) result
end

Example

module Bindings(F: FOREIGN) = struct
  open F
  let puts = foreign "puts" (string \to returning int)
end
Staged binding: recovering the dynamic interpretation

module Foreign_dynamic = struct
  type 'a result = 'a
  let foreign = foreign (* i.e. implementation above *)
end

Example

Bindings(Foreign_dynamic)

produces

sig
  val puts : string \to int
end
Staged binding: generating C

val generate_C : string → 'a fn → unit

module Foreign_generate_C = struct
  type 'a result = unit
  let foreign = generate_C
end

Example

Bindings(Foreign_generate_C)

  outputs

  value puts_stub(value s) {
    char *p = Ptr_val(s);
    int n = puts(p);
    return Val_int(n);
  }
Staged binding: generating ML

```ocaml
val generate_ML : string \to \ 'a fn \to unit

module Foreign_generate_ML = struct
  type \ 'a result = unit
  let foreign = generate_ML
end
```

Example

```
Bindings(Foreign_generate_ML)
```

```
outputs

external puts_stub : address \to int = "puts_stub"

let foreign nm fn = match nm, fn with
  | "puts", Function (View (_, froms, Ptr Char)) \to
    fun s \to puts_stub (froms s)
  | "puts", fn \to fail "type mismatch"
  | name, _ \to fail "unexpected name"
```
module Bindings(F: FOREIGN) = struct
    open F
    let puts = foreign "puts" (string @→ returning int)
end

module Generated_ML : FOREIGN with type 'a result = 'a
    = (* code generated on previous slide *)

 Bindings(Generated_ML)

Type safe linking via type refinement!
concurrency

remote calls

function pointers

void (*)(int, float);

more object types

struct s x[3];

determining object layout

struct t {
  int x, y;
};

inverted bindings
Next time: overloading

\[
\text{val} \ (=) : \ \{E:E\text{EQ}\} \to E.t \to E.t \to \text{bool}
\]