

ppx_interpret: selective interpretation

August 15, 2017

What We Want to Achieve

1. Interpreting a mixed OCaml/C program with arbitrary outside code in C. OCaml has the `%external` keyword and C macros for this, so it is well-defined. Complications: function pointers, callbacks, custom blocks.
2. Interpreting just one module of an OCaml program and running others in bytecode (Dynlink?) or native code (dlopen?).
3. As a special case of this, running the Standard Library full speed always.
4. Calling the debugger from any build system. We envisage this to be done by making the debugger take a command line just like the final build step.

Key Tasks

It is important to distinguish various things when we talk about making the interpreter run real programs using arbitrary libraries:

1. Getting the interpreter linked with the parts of the program not in plain OCaml – so it may call the outside functions. This may be static or dynamic linking.
2. Getting the `%external` definitions compiled in so that OCaml can actually call those outside functions.
3. Actually calling a function (converting interpreter's data structure to OCaml heap value, reading back any resultant heap value into the interpreter's data structure).
4. Ensuring that modules (both C and OCaml) are run module-initialisation in the same order when interpreted as in normal operation for semantic consistency.

We can already convert `Tinyocaml.t` values to and from heap values in the OCaml heap. Since whatever scheme we do will involve the interpreter and the target program being linked in to the same heap, this is sufficient. No custom blocks, functions yet.

Solution: ppx_interpret

Can we implement this whole thing using PPX? A new `PPX_interpret` is created. This converts each OCaml source file to an interpreted version with exactly the same interface.

- No changes to build system
- Everything just works
- Interpret one module or many (Annotate `[@interpret]` for example)
- Module initialisation problem goes away too...

- Can annotate just a single function instead of a whole file, to interpret just that function.

Claim: the PPX system is the perfect way to embed our interpreter (and thus debugger) into the OCaml build process so that it will be *accessible* as we require.

The `ppx_interpret` extension takes a given implementation `.ml` file, and produces another version, which follows the same interface `.mli` but which interprets its contents. For example, consider the file `b.ml`:

```
(* The marker to say this file should be interpreted *)
[%interpret]

(* An external function implemented in C. It will call back into OCaml too. *)
external c_function : int -> int = "c_function"

(* A recursive function, which also uses something from another module *)
let rec double x =
  if x < 100 then double (x * 2) else A.double x

(* A simple function, using the C function *)
let trip x =
  let double x = x * 2 in
  (* Use Callback.register, so c_function can call back to double. *)
  let () = Callback.register "double" double in
  c_function x * 3

(* Here, a function which calls something in this module *)
let f x = double x
```

This is processed by `ppx_interpret` to yield, for example, for just the `double` function (which calls `A.double`):

```
let double x =
  let module A =
    struct
      let double env =
        function
        | x::[] ->
          let heap_x = Tinyexternal.to_ocaml_value x in
          let result = A.double heap_x in
          Tinyexternal.of_ocaml_value env result "int"
        | _ -> failwith "A.double: arity"
      end in
    let open Tinyocaml in
      let tiny_x = Tinyexternal.of_ocaml_value [] x {|int|} in
      let (_,program) =
        Tinyocamlrw.of_string
        {|let rec double x = if x < 100 then double (x * 2) else A.double x in double x|}
      in
      let env =
        [EnvBinding (false, (ref [((PatVar "x"), tiny_x)]))] @
        ([EnvBinding
          (false,
            (ref [((PatVar "A.double"), (mk "A.double" A.double))]))]
        @
        [EnvBinding
          (false, (ref [((PatVar "c_function"), c_function_builtin))])])
      in
      let tiny_result =
        Eval.eval_until_value true false (env @ (!Eval.lib)) program in
      (Tinyexternal.to_ocaml_value tiny_result : int)
```