Making ocamli Work for Real Programs

July 13, 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.

Achieved So Far

1. We can 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.

Standard Library Tasks

Polymorphic Comparison, Hashing, Serialization

We need to do our own polymorphic comparison, which must act exactly the same as that of the OCaml runtime. Two approaches: (a) Write our own, which would traverse the `Tinyocaml.t` datatype (b) Convert the data structures to be compared into the OCaml heap representation and call `compare_val` directly.

New Strategy 13th Jul 2017

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)
- Just remember to link in ocamli-support.o for the C parts of ocamli.
- Module initialisation problem goes away too...
- Can annotate just a single function instead of a whole file!

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.

Example

Consider a .ml file

```
external f : int -> int = "f"

let _ = A.some_module_initialisation ()

let g a = f a (* A normal function which calls the external *)
```

The PPX_interpret code will replace this with code to interpret the module initialisation and shims for all the other functions. For example:

```
external f : int -> int = "f"

(* Read .interp files left behind by runs of PPX_interpret on the other modules *)
let _ =
   OCamli.setup_environment ()

let call_A.some_module_initialisation () = A.some_module_initialisation ()

(* Because we read the .interp files, we know what A.some_module_initialisation is here *)
let _ =
   Ocamli.eval "A.some_module_initialisation ()"

let call_f x = f x

let g a =
  to_ocaml_heap_value (string_of_ocaml_heap_value (call_f a))
```

Do we even need these .interp files? Because everything goes through the interface. Maybe just types? We can get these from source files, though? Or .cmt files?