Inline BOM

Inline BOM programmers to embed BOM code in PML programs. We use inline BOM to build some of the following mechanisms:

- primitive operations, "e.g., operations over 32-bit ints"
- foreign calls
- low-level IO support
- synchronization / concurrent data structures, "i.e., implementing CML"
- process schedulers

One might rightly ask why we encode these mechanisms in a low-level IR like BOM. Our primary concern is state. The above mechanisms make heavy use of stateful operations, which for various reasons, are unsafe in PML. Another important concern is data layout. Because PML is high-level, we wish to keep byte layouts of objects abstract. But to implement concurrent data structures, we frequently need to access byte layouts for synchronization operations. By encoding these mechanisms in BOM, we can keep PML pure and more amenable to aggressive optimizations, e.g., CML specialization.

Features

Our inline BOM mechanism consists of a few extensions to PML. The programmer can access each of these mechanisms through special syntax that we describe below.

Inline BOM declarations

The following syntax allows us to make BOM declarations. Primcode declarations occur at the top level of modules. BOM declarations can be types, C function prototypes, or HLOps definitions.

```
_primcode (
BOMDecl ...
)
```

Example: BOM type declaration

We can define the state field of single-touch futures as follows. The state can be either a non-pointer flag, or a pointer to either the value of the future or the continuation of a blocked thread.

```
type 'a thunk = unit -> 'a

_primcode (
  (*
    a future_state word contains one of the following values:
    * EMPT_Y_F
    * STOLEN_F
    * EVAL_F
    * FULL value
    * WAITING cont
  *)
  typedef future1_state = any;
)
```

Example: C function prototype

The following snippet contains a prototype for a printer function, which is defined in some C file.

```
_primcode(
  extern void M_Print(void*)
)
```

Example: HLOps

The following HLOp prints its parameter string and a trailing new line.

```
_primcode(
  define inline @print-ln(msg : string / exh : exh) : () =
do ccall M_Print(msg)
do ccall M_Print("\n")
return ()
)
```

Types

We can make type declarations that are visible to PML and BOM code. In PML, the type t is an abstract type, but in BOM, it has the type bomTy.

```
type (*a, ...) t = _prim(bomTy)
```
Example

We can define the type of single-toucher futures as follows. The future structure is just a mutable pair of the state and the thunk.

type future1 = _prim(!future1_state, thunk)

Exporting BOM definitions

We can bind BOM functions and HLOps to PML identifiers. Because the translation from BOM types to PML types is undefined, we require the programmer to ascribe the bound variable a PML type.

val f : ty = _prim(bomId)

Example

_primcode(_
define @f(_: unit / exh : exh) : unit = ...;
)
val f : unit -> unit = _prim(@f)

Importing PML datatypes

BOM code can refer to datatypes and constructors defined in PML.

Example

datatype t = X of int | Y

_primcode(_
define @mk-x(x : int / exh : exh) : t =
return(X(X
)
)

Importing PML declarations

BOM code can refer to PML variables that are bound at the type level.

let f : bomTy = pmlvar(pmlId)

Example

let app : fun(_ / exh -> unit), List.list / exh = pmlvar(List.app)

Loading HLOps at compile time

In many cases, we wish to create PML syntax that expands into several HLOps. For example, the pval binding form expands into a few HLOps from the single-toucher-future module. But because the compiler is responsible for doing this expansion, we do not not have a static handle on the HLOps. Instead we use a dynamic lookup at compile time. The following function in the HLOpEnv module does the job:

(* locate a HLOp by path name, e.g., Future1.@touch *)
val findDefByPath : string list -> hlop_def option

Example

To look up the Future1 touch operation, we say:

case HLOpEnv.findDefByPath ("Future1", "touch")
of SOME def => (* got it *)

Todos

- Make HLOps safe for PML optimizations that rename variables.
- Let inline BOM code refer to nullary constructors by name rather than by the enumeration index.