Adding static staging to OCaml

Olivier Nicole, Dima Szamozvancev, Leo White, Ningning Xie, **Jeremy Yallop**

IFIP WG 2.11, Delft, April 2023

Quotes

(from MetaML)

Quote

Construct code representing `e`:

$$\ll \texttt{e} \gg$$

$\ll \texttt{e} \gg$ has type `t expr`
when `e` has type `t`

$\ll \texttt{e} \gg$ is at level $n - 1$
when `e` is at level n

Splice

Evaluate `e` and splice the result:

`$(e)`

`$(e)` has type `t`
when `e` has type `t expr`

`$(e)` is at level n
when `e` is at level $n - 1$

Idea: `mpower n` builds code for $\lambda x.x^n$.

```
(* int -> int expr -> int expr *)
macro rec mpow n x =
  if n = 0 then ≪ 1 ≫
  else ≪ $x * $(mpow (n - 1) x)≫

(* int -> (int -> int) expr *)
macro mpower n = ≪ fun x → $(mpow n ≪x≫) ≫
```

Missing features: run & CSP

Run

Convert a representation of `e` to `e`:

$$\texttt{run}\ \ll \texttt{e} \gg \rightsquigarrow \texttt{e}$$

`run e` has type `t`
when `e` has type `t expr`

$\ll \texttt{e} \gg$ is at level n
when `e` is at level n

CSP

Store a heap reference within code:

$$\ll \texttt{x} \gg$$

$\ll \texttt{x} \gg$ has type `t expr`
when `x` has type `t`

$\ll \texttt{x} \gg$ is at level n
when `x` is at level n

We don't have either of these!

Phases

(from Racket)

let bindings are at level 0. **macro** bindings are at level -1.

```
let square x = x * x

macro rec mpow n x =
 if n = 0 then ≪ 1 ≫
 else if n mod 2 = 0 then ≪ square $(mpow (n / 2) x) ≫
 else ≪ $x * $(mpow (n - 1) x)≫

macro mpower n = ≪ fun x → $(mpow n ≪x≫) ≫

let pow5 = $(mpower 5)
```

We can quote **let**-bound square in mpow and splice **macro**-bound mpower in pow5.

Imports (two types)

level 1 import

Make `M`'s bindings available at level 0:

open `M`

Can quote `M.f` in **macro** bindings.
Can call `M.f` in **let** bindings.

level −1 import

Make `M`'s bindings available at level −1:

open `~M`

Can call `M.f` in **macro** bindings.
Can splice `M.f` in **let** bindings.

Safe to discard compile-time heap & erase compile-time bindings before run time.

```
let square x = x * x

macro rec mpow n x =
  if n = 0 then ≪ 1 ≫
  else if n mod 2 = 0 then ≪ square $(mpow (n / 2) x) ≫
  else ≪ $x * $(mpow (n - 1) x)≫

macro mpower n = ≪ fun x → $(mpow n ≪x≫) ≫

let pow5 = $(mpower 5)
```

Safe to discard compile-time heap & erase compile-time bindings before run time.

```
let square x = x * x

macro rec mpow n x =
  if n = 0 then ≪ 1 ≫
  else if n mod 2 = 0 then ≪ square $(mpow (n / 2) x) ≫
  else ≪ $x * $(mpow (n - 1) x)≫

macro mpower n = ≪ fun x → $(mpow n ≪x≫) ≫

let pow5 = fun x → x * square (square (x * 1))
```

Modules

(from OCaml)

```
module type MPOWER = sig macro mpower : int → int expr end

module MPower : MPOWER = struct
  let square x = x * x

  macro rec mpow n x =
    if n = 0 then ≪ 1 ≫
    else if n mod 2 = 0 then ≪ square $(mpow (n / 2) x) ≫
    else ≪ $x * $(mpow (n - 1) x)≫

  macro mpower n = ≪ fun x → $(mpow n ≪x≫) ≫
end

let pow5 = $(MPower.mpower 5) (* square not in scope! *)
```

```
module type MPOWER = sig macro mpower : int → int expr end

module Pow5(M: MPOWER) = struct
  let pow5 = $(M.mpower 5)  (* When is M.mpower available? *)
end
```

Thank you