-XStaticPointers

Lightweight remutable computations

Mathieu Boespflug, Facundo Domínguez, Simon Peyton Jones (MS Research)
Example: remote calculator (I)

```
data Arith = Plus Double Double  
    | Mult Double Double  
    | Neg Double

instance Serializable ArithOp
```

```
server :: Process ()
server = forever $ do
    expect >>= say . show . \case of
        Plus x y -> x + y
        Mult x y -> x * y
        Neg x -> negate x

let'sDoSomeMath :: ProcessId -> Process ()
let'sDoSomeMath there = do
    send there $ Plus 10 2
    send there $ Mult (2^10) (3^10)
    send there $ Neg 1
```
I'mASupervisor :: ProcessId -> Process ()
I'mASupervisor there = forever $ do
  monitor there
  MonitorRef _ _ <- expect
  pid <- restartServer
  I'mASupervisor pid

server :: Process ()
server = forever $ do
  expect >>= say . show . case of
    Plus x y -> x + y
    Mult x y -> x * y
    Neg x -> negate x

data Arith = Plus Double Double |
            Mult Double Double |
            Neg Double

instance Serializable ArithOp
Supervision hierarchies

“Reports to”
Calculation type is ...

★ … indirect;
★ … error prone;
★ … difficult to extend;
★ … antimodular.
Example: Remote calculator (II)

```haskell
plus, mult :: Int -> Int -> Process ()
plus x y = say $ show $ x + y
mult x y = say $ show $ x * y

neg :: Int -> Process ()
neg x = say $ show $ negate x

let'sDoSomeMath :: NodeId -> Process ()
let'sDoSomeMath there = do
  spawn there $ plus 10 2
  spawn there $ mult (2^10) (3^10)
  spawn there $ neg 1
```
Spawning functions is ...

★ ... direct;
★ ... error proof;
★ ... easy to extend;
★ ... modular.
send :: Serializable a => NodeId -> a -> Process ()
expect :: Serializable a => Process a
spawn :: NodeId -> Closure (Process ()) -> Process ()
Serialization

data Static a

data Closure a =
    Closure (Static a) -- code pointer
    Env

type Env = ByteString

instance Serializable (Static a)
instance Serializable (Closure a)
Serialization

data Static a

data Closure a =
  Closure (Static (Env -> a))
  Env

type Env = ByteString

instance Serializable (Static a)
instance Serializable (Closure a)
Winging it with Template Haskell

```
plus, mult :: (Int, Int) -> Process ()
plus (x, y) = say $ show $ x + y
mult (x, y) = say $ show $ x * y

neg :: Int -> Process ()
neg x = say $ show $ negate x

remotable ['plus', 'mult', 'neg']
```

```
let'sDoSomeMath :: NodeId -> Process ()
let'sDoSomeMath there = do
  spawn there $ $(mkClosure 'plus) (10, 2)
  spawn there $ $(mkClosure 'mult) (2^10, 3^10)
  spawn there $ $(mkClosure 'neg) 1
```
What’s missing ...

★ TH splices clunky, bad error messages.
★ Unnatural uncurrying of remutable functions.
★ Need to declare upfront what is remutable.
★ Can’t remote arbitrary expressions.
★ Remote tables management anti-modular.
Example: Remote calculator (IV)

```haskell
plus, mult :: Int -> Int -> Process ()
plus x y = say $ show $ x + y
mult x y = say $ show $ x * y

neg :: Int -> Process ()
neg x = say $ show $ negate x

let'sDoSomeMath :: NodeId -> Process ()
let'sDoSomeMath there = do
  spawn there $ closure $ static (plus 10 2)
  spawn there $ closure $ static (mult (2^10) (3^10))
  spawn there $ closure $ static (neg 1)
```
-XStaticPointers

★ New syntactic form:

\[ e ::= \ldots \mid \text{static } e_1 \]

★ Meaning: “label of \( e_1 \)”.

★ Restrictions: \( e_1 \) must be \( closed \).

\[
\Delta \vdash e :: a \\
\Delta; \Gamma \vdash \text{static } e :: \text{Static } a
\]
Desugaring static pointers

\[ C[\text{static } e] \downarrow f :: a; f = e; C[s] \]

where
★ C is any context;
★ s is a \textit{value} of type Static a;
★ f is a fresh \textit{(unique)} name.
Static Pointers are ...

★ ... guaranteed to be unique;
★ ... free;
★ ... modular;
★ ... safe.
RPC calls

type Dict c = c => Dict

spawn :: NodeId -> Closure (Process ()) -> Process ()
send :: Serializable a => NodeId -> a -> Process ()
expect :: Serializable a => Process a

call :: Static (Dict (Serializable a))
    -> NodeId
    -> Closure (Process a)
    -> Process a
Let's implement (simplified) call ... 

call :: Static (Dict (Serializable a))
    -> NodeId
    -> Closure (Process a)
    -> Process a

call sdict there cf = do
    here <- getSelfPid
    spawn there (closure cf `bindCP` sendBack dict here)
    x <- expect
    return x

bindCP :: Closure (Process a)
    -> Closure (a -> Process b)
    -> Closure (Process b)
Let's implement bindCP ...

bindCP :: Closure (Process a)
    -> Closure (a -> Process b)
    -> Closure (Process b)
bindCP cm ck =
    static (>>=) `closureApply` cm `closureApply` ck
Let’s implement bindCP ...

\[
\text{bindCP} :: \forall a\ b. \text{Closure (Process}\ a) \\
\quad \rightarrow \text{Closure (a -> Process}\ b) \\
\quad \rightarrow \text{Closure (Process}\ b)
\]

\[
\text{bindCP} = \Lambda a. \Lambda b. \lambda cm. \lambda ck. \\
\quad \text{static (} \gg=_{\text{Process}} a\ b) \ `\text{closureApply} ` \ cm \ `\text{closureApply} ` \ ck
\]
Typing rule for polymorphic expr.

★ New syntactic form:
\[ e ::= \ldots \mid \text{static } e_1 \]

★ Meaning: “label of \( e_1 \)”.
★ Restrictions: \( e_1 \) must be \textit{term closed}.

\[
\frac{
\Delta; \text{types}(\Gamma) \vdash e :: a
}{
\Delta; \Gamma \vdash \text{static } e :: \text{Static } a
} 
\]
From here to there

<table>
<thead>
<tr>
<th>GHC Extension</th>
<th>implemented</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>distributed-static</code></td>
<td>implemented</td>
</tr>
<tr>
<td>(combinator library)</td>
<td></td>
</tr>
<tr>
<td><code>distributed-process</code></td>
<td>implemented</td>
</tr>
<tr>
<td>(Cloud Haskell)</td>
<td></td>
</tr>
</tbody>
</table>

★ Next up: **simplify, review, merge.**
★ Ideally: resistance to *malicious segfaults.*
Serialization

data Static a

data Closure a =
    Closure (Static (Env -> a))
    Env

type Env = ByteString

instance Serializable (Static a)
instance Serializable (Closure a)
Related work

★ Serialisation support in the RTS (Eden, Jost Berthold).
★ Serialisation as a library (Packman, Jost Berthold).
★ Hdph closures (Maier, Stewart, Trinder).
Conclusion

★ Static pointers: very general notion of “value with a name”.
★ Polymorphism is just as important in a distributed setting.
★ Much smaller language extension than native support for runtime reflection.
★ Beware of unimplemented sidenotes…
Thank you!