OCamlによるEmbedding by Unembedding

類家 健永（東北大学電気情報物理工学科）松田 一孝（東北大学大学院情報科学研究科）

目的

- 埋め込み領域特化言語の実装方式であるEmbedding by Unembedding（EbU）[Matsuda+ 23]の実現に必要な言語機能を探る
  - 元の実装はHaskellの型族，高階多相，GADTを駆使
  - 代替の機能を用いつつ、同等のユーザビリティを達成
- Case-Studyを通して有用性を評価

背景：Embedding by Unembedding

- 埋め込み領域特化言語の実装方式のひとつ
- HOASを用いて高度なセマンティクスを埋め込むフレームワーク

変数バインディングを
ホスト言語の関数で表現する

$[\![\Gamma \vdash e : A]\!]$ がホスト言語の $[\![\Gamma]\!]$ から $[\![A]\!]$ への関数では表現が難しい
高度なセマンティクスの例：
- 可逆計算
- 増分計算
- 双方向変換

EbUは**lift処理**で、ギャップを埋める

結果

OCaml版とHaskell版でほぼ同等のユーザビリティを維持しつつ、EbUの実現に成功

OCaml
```
let (let_) = fun e1 e2 ->
  let argSpec = XCons (XZ, XCons (XS XZ, XNil))
  in toF argSpec (liftCore {f = fun x -> fromF argSpec letSem x}) e1 e2
```

Haskell
```
let_ =
  let argSpec = LZ :. (LS LZ) :. End
  in EbU.lift argSpec letSem
```

OCaml版とHaskell版のlift処理の用法の差異の原因

Haskell
```
lift :: forall sem ss r. Variables sem => Dim ss
     -> (forall env. FuncTerm sem env ss r) -> FuncU sem ss r
lift ns f =                          -- arg_specでは高階多相の表現が難しい！
  let h :: forall env. Env (SemRep sem env) ss -> sem env r
      h =  fromF f
  in toF ns (lift' @sem h)
```

仮に高階型変数があれば，対応する型は
`'env. (...,'env,...,'env 'F,...) arg_spec -> 'env 'F`となる
functorを用いることで表現は可能だが、ユーザビリティを大きく損う

OCaml版EbUを用いたIncremental λ-Calculus (ILC) [Cai+ 14] の埋め込み例

ILCとは？

関数の返り値と、差分の計算を同時に行う
例：
$f\ x = x^2$を定義すると
$df\ x\ dx = 2 * x * dx + dx * dx$も自動的に定義

Step1. 意味領域の定義

$\Gamma \vdash e : A$

OCaml
```
type ('x, 'xs) ilcSem =
     ('xs H.hlist -> 'x FromFst.t)
   * ('xs H.hlist -> 'xs D.hlist -> 'x FromSnd.t)
```

$\Gamma \vdash e : A$ の意味領域$([\![\Gamma]\!] \to [\![A]\!]) \times ([\![\Gamma]\!] \to \Delta[\![\Gamma]\!] \to \Delta[\![A]\!])$ に対応
つまり、項eの意味は通常の関数と差分関数の組

Step2. ILCの意味関数の定義

OCaml
```
let letSem = fun (f, df) (g, dg) ->
      (fun theta ->
         let FFst v = f theta in
         let FFst w = g (HCons(FFst v,theta)) in
         FFst w),
      (fun theta dtheta ->
         let FFst v = f theta in
         let FSnd dv = df theta dtheta in
         dg (HCons(FFst v, theta)) (HCons(FSnd dv,dtheta)))
let mulSem = ...
let fstSem = ...
```

Step3. syntaxの定義

埋め込む言語のHOAS（finally-tagless style [Carette+ 09]）を定義

OCaml
```
module type ILCSYN = sig
  type 'a expr

  val ( let_ ) : 'a expr -> ('a expr -> 'b expr) -> 'b expr
  val mul : (int * int) expr -> (int * int) expr -> (int * int) expr
  val fst : (('a * 'b) * ('da * 'db)) expr -> ('a * 'da) expr
end
```

Step4. 意味モジュールの定義

OCaml
```
module HILC_sem ILCSYN with type 'a expr = 'a ILC.VarILC.envi = struct
  type 'a expr = 'a envi

  let (let_) = fun e1 e2 ->
    let argSpec = XCons (XZ, XCons (XS XZ, XNil))
    in toF argSpec (liftCore {f = fun x -> fromF argSpec letSem x}) e1
  let mul = fun e1 e2 ->
    let argSpec = XCons (XZ, XCons (XZ, XNil))
    in toF argSpec (liftCore {f = fun x -> fromF argSpec mulSem x}) e1 e2
  let fst = fun e ->
    let argSpec = XCons (XZ, XNil)
    in toF argSpec (liftCore {f = fun x -> fromF argSpec fstSem x}) e
...
end
```

lift処理

lift処理の実装

lift処理の内訳

※$[a_1,...,a_n] \simeq a_1 * (... * (a_n * unit)...)$

```
letSem : ('a,'env) ilcSem -> ('b, ('a * 'env )) ilcSem ->
                                          ('b, 'env) ilcSem
      |fromF
letSem' : ('env,[([],'a) sig2;(['a],'b) sig2]) SemRep.hlist ->
                                          ('b,'env) ilcSem
      |liftCore
let_' : ([([],'a) sig2 * (['a],'b) sig2]) HoasRep.hlist ->
                                                  'b expr
      |toF
let_ : 'a expr -> ('a expr -> 'b expr) -> 'b expr
```

toF、fromFの実装

OCamlでは代わりにGADT(arg_spec)を導入

Haskell
```
toF :: Dim ss -> (Env (HoasRep sem) ss -> EnvI sem r) -> FuncU sem ss r
toF End f       = f ENil
toF (n :. ns) f = ¥k -> toF ns (f . ECons (toHRep n k))
```

Haskell
```
fromF :: FuncTerm sem env ss r -> Env (SemRep sem env) ss -> sem env r
fromF f ENil              = f
fromF f (ECons (SR x) xs) = fromF (f x) xs
```

OCaml
```
type (_,_,_,_,_,_) arg_spec =
  | XNil  : ('dummy, 'env, 'rr, 'rr envi, ('rr,'env) sem, unit) arg_spec
  | XCons : ('xs, 'env, 'app, 'x envi, 'f) wapp_cspec *
            ('dummy, 'env, 'rr, 'toF, 'fromF, 'ss) arg_spec ->
            ('app * 'dummy, 'env, 'rr, 'f -> 'toF, ('x,'app) sem ->
                        'fromF, (('xs,'x) sig2 * 'ss)) arg_spec
```

liftCore関数の実装

Haskell
```
liftCore :: ...
liftCore f ks = ...
  where ...
        mkXs :: TEnv env -> TEnv as'
             -> TEnv (Append as' env)
             -> Env (EnvI sem) as'
        mkXs _ ENil _ = ENil
        mkXs p (ECons _ as) te@(ECons _ te')
          = let x = EnvI $ ¥e' ->
                      weakenMany te e' var
            in ECons x (mkXs p as te')
```

型レベルのアペンド

移植

OCaml
```
type (_,_,_) wapp =
  | AppNil  : (unit, 'b, 'b) wapp
  | AppStep : ('x , 'y , 'r) wapp ->
                  ('a * 'x, 'y, 'a * 'r) wapp
```

Haskell
```
type family Append as bs where
  Append '[] bs = bs
  Append (a ': as) bs = a ': Append as bs
```

提供

OCaml
```
let liftCore : ...
  = fun ff ks ->
  { runEnvI = fun (type env) (e : env TEnv.hlist) -> ... in
    let rec mkXs : type env ys ys_env. env TEnv.hlist
             -> ys TEnv.hlist
             -> (ys, env, ys_env) wapp
             -> ys_env TEnv.hlist -> ys HListEnvI.hlist =
      fun p ys wit te ->
        match ys, wit, te with
          | HNil, _, _ -> HNil
          | (HCons(_,ys')),AppStep(wit'),HCons(_,te') ->
            let x = { runEnvI = fun e' ->
                                weakenMany te e' H.var} in
            HCons(x,mkXs p ys' wit' te') in ...
  }
```

Appendはtype familyで定義されているがOCamlではGADTで表現
`(x , 'y , 'r) wapp`は‘r = 'x ++ 'yを表す