Distributive Disjoint Polymorphism for Compositional Programming

Xuan Bi[1]; **Ningning Xie**[1]; Bruno C. d. S. Oliveira[1]; Tom Schrijvers[2]

[1] The University of Hong Kong , [2] KU Leuven

Compositional Programming

- Simple compositional design techniques:
  - shallow embeddings of DSLs
  - finally tagless
  - object algebras
- The F_i^+ calculus improves on existing techniques by supporting highly modular and compositional designs.
- We compare shallow embeddings of parallel prefix circuits [1]:
  - The finally tagless encoding [2];
  - SEDEL encoding [3], a source language built on top of F_i^+ .

	$\lambda_{,,}$ [8]	λ_i [4]	$\lambda_\wedge^\vee$ [9]	λ_i^+ [7]	F_i [5]	F_i^+
Disjointness	○	●	○	●	●	●
Unrestricted intersections	●	○	●	●	○	●
BCD subtyping	○	○	●	●	○	●
Polymorphism	○	○	○	○	●	●
Coherence	○	◑	○	●	◑	●
Bottom type	○	○	●	○	○	●

Summary of intersection calculi

F_i^+ Language Features

- Intersection types
  If `e :: A`, and `e :: B`, then we have `e :: A & B`.
  ```
  Int & Bool
  (Int → Int) & (Int → Bool)
  ```
- In many languages and calculi, intersection types do not increase the expressiveness of terms.
  ```
  Int & Bool  -- uninhabited
  ```
- In F_i^+, the merge operator increases the expressiveness of terms.
  ```
  (1 ,, True) :: Int & Bool
  (1 ,, True) :: Int  -- reduces to 1
  (1 ,, True) :: Bool -- reduces to True
  (1 ,, 2)    :: Int  -- ambiguous
  ```
- Disjointness [4]
  In `(e1 : A ,, e2 : B)`, we have `A * B`
  ```
  (1 ,, True) :: Int & Bool -- valid as Int * Bool
  (1 ,, 2)    :: Int & Int  -- invalid
  ```
- Disjoint Polymorphism [5]
  $$\Lambda\alpha.\Lambda\beta * \alpha.\lambda(x : a).\ \lambda(y : a).\ (x \,,, y)$$
- Distributive subtyping (BCD-style subtyping [6]):
  ```
  (Int → Int) & (Int → Bool) <: Int → (Int & Bool)
          {l: Int} & {l:Bool} <: {l: Int & Bool)
  (∀α*Int.Int) & (∀α*Int.Bool) <: ∀α*Int.(Int & Bool)
  ```

Coherence

- Intersections elaborate to pairs
  ```
  1 ,, 2              ⇝ (1, 2)
  1 :: Int & Int      ⇝ (1, 1)
  (1 ,, True) :: Int  ⇝ fst (1, True)
  (1 ,, True) :: Bool ⇝ snd (1, True)
  ```
- The coherence issue
  ```
  (1 : Int & Int) :: Int ⇝ fst (1, 1)
                         ⇝ snd (1, 1)
  ```
- Contextual equivalence
  $$\texttt{fst (1, 1)} \cong \texttt{snd (1, 1)}$$
- The canonicity relation for F_i^+
  - Heterogeneous logical relation
  - Predicativity
- Formalization of coherence lemmas in Coq

Take-Home Message

- F_i^+ is a type-safe and coherent calculus.
- F_i^+ has disjoint intersection types, BCD subtyping and parametric polymorphism.
- F_i^+ improves the state-of-art of compositional designs.

Finally tagless encoding

```
class Circuit c where
  identity :: Int -> c;         fan   :: Int -> c
  beside   :: c -> c -> c;      above :: c -> c -> c
  stretch  :: [Int] -> c -> c

data Width = W { width :: Int }
instance Circuit Width where
  identity n = W n
  fan n = W n
  beside c1 c2 = W (width c1 + width c2)
  above c1 c2 = c1
  stretch ws c = W (sum ws)

data Depth = D { depth :: Int }
instance Circuit Depth where
  identity n = D 0
  fan n = D 1
  beside c1 c2 = D (max (depth c1) (depth c2))
  above c1 c2 = D (depth c1 + depth c2)
  stretch ws c = c

{-------- Interpreting multiple ways --------------}
type DCircuit = forall c. Circuit c => c
brentKung :: DCircuit =
   above (beside (fan 2) (fan 2))(above (stretch [2, 2] (fan 2))
     (beside (beside (identity 1) (fan 2)) (identity 1)))
e1 :: Width = brentKung
e2 :: Depth = brentKung

{-------- Composition of embeddings ---------------}
instance (Circuit c1, Circuit c2) => Circuit (c1, c2) where
  identity n = (identity n, identity n)
  fan n = (fan n, fan n)
  beside c1 c2 = (beside (fst c1)(fst c2), beside (snd c1) (snd c2))
  above c1 c2 = (above (fst c1) (fst c2), above (snd c1) (snd c2))
  stretch ws c = (stretch ws (fst c), stretch ws (snd c))
e3 :: (Width, Depth) = brentKung

{-------- Composition of dependent interpretations --------------}
data WellSized = WS { wS :: Bool, ox :: Width }
instance Circuit WellSized where
  identity n = WS True (identity n)
  fan n = WS True (fan n)
  beside c1 c2 = WS (wS c1 && wS c2) (beside (ox c1) (ox c2))
  above c1 c2 = WS (wS c1 && wS c2 && width (ox c1) == width (ox c2))
                   (above (ox c1) (ox c2))
  stretch ws c = WS (wS c && length ws==width (ox c))
                    (stretch ws (ox c))
e4 :: WellSized = brentKung
```

SEDEL encoding

```
type Circuit[C] = {
  identity : Int -> C,  fan : Int -> C,
  beside : C -> C -> C, above : C -> C -> C,
  stretch : List[Int] -> C -> C };

type Width = { width : Int };
language1 : Circuit[Width] = {
  identity (n : Int) = { width = n },
  fan (n : Int) = { width = n },
  beside (c1 : Width) (c2 : Width) = { width = c1.width + c2.width },
  above (c1 : Width) (c2 : Width) = { width = c1.width },
  stretch (ws : List[Int]) (c : Width) = { width = sum ws } };

type Depth = { depth : Int };
language2 : Circuit[Depth] = {
  identity (n : Int) = { depth = 0 },
  fan (n : Int) = { depth = 1 },
  beside (c1 : Depth) (c2 : Depth) = { depth = max c1.depth c2.depth},
  above (c1 : Depth) (c2 : Depth) = { depth = c1.depth + c2.depth},
  stretch (ws : List[Int]) (c : Depth) = { depth = c.depth } };

{-------- Interpreting multiple ways --------------}
type DCircuit = { accept : forall C. Circuit[C] -> C };
brentKung : DCircuit = { accept C l = l.above (l.beside (l.fan 2)
  (l.fan 2)) (l.above (l.stretch (cons 2 (cons 2 nil)) (l.fan 2))
     (l.beside (l.beside (l.identity 1) (l.fan 2)) (l.identity 1))))};
e1 = brentKung.accept Width language1;
e2 = brentKung.accept Depth language2;

{-------- Composition of embeddings ---------------}
language3 : Circuit[Width & Depth] = language1 ,, language2;
e3 = brentKung.accept (Width & Depth) language3;

{-------- Composition of dependent interpretations --------------}
type WellSized = { wS : Bool };
language4 = {
  identity (n : Int) = { wS = true },
  fan (n : Int) = { wS = true },
  above (c1 : WellSized & Width) (c2 : WellSized & Width)
    = { wS = c1.wS && c2.wS && c1.width == c2.width },
  beside (c1 : WellSized) (c2 : WellSized) = { wS = c1.wS && c2.wS },
  stretch (ws : List[Int]) (c : WellSized & Width)
    = { wS = c.wS && length ws == c.width } };
e4 = brentkung.accept (WellSized & Width) (language1 ,, language4)
```

References

1. Gibbons, J., Wu, N.: Folding domain-specific languages: deep and shallow embeddings (functional pearl). ICFP 2014
2. Carette, J., Kiselyov, O., Shan, C.C.: Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. JFP 19(05), 509 (2009)
3. Bi, X., Oliveira, B.C.d.S.: Typed first-class traits. ECOOP 2018
4. Oliveira, B.C.d.S., Shi, Z., Alpuim, J.: Disjoint intersection types. ICFP 2016
5. Alpuim, J., Oliveira, B.C.d.S., Shi, Z.: Disjoint polymorphism. ESOP 2017
6. Barendregt, H., Coppo, M., Dezani-Ciancaglini, M.: A filter lambda model and the completeness of type assignment. The journal of symbolic logic 48(04), 931–940 (1983)
7. Bi, X., Oliveira, B.C.d.S., Schrijvers, T.: The essence of nested composition. ECOOP 2018
8. Dunfield,J.:Elaborating intersection and union types. JFP 24(2-3), 133–165 (2014)
9. Blaauwbroek, L.: On the interaction between unrestricted union and intersection types and computational effects. Master's thesis, Technical University Eindhoven (2017)