Compositional Programming

Weixin Zhang[1,2], Yaozhu Sun[2], and Bruno C. d. S. Oliveira[2]

1.University of Bristol

2.The University of Hong Kong

ECOOP 2021

Motivation

OOP

```
abstract class Exp {
  def eval: Int
}
class Lit(n: Int) extends Exp {
  def eval = n
}
class Add(e1: Exp, e2: Exp) extends Exp {
  def eval = e1.eval + e2.eval
}

class Mul(e1: Exp, e2: Exp) extends Exp {
  def eval = e1.eval * e2.eval
}
```

FP

```
data Exp where
  Lit :: Int -> Exp
  Add :: Exp -> Exp -> Exp

eval :: Exp -> Int
eval (Lit n) = n
eval (Add e1 e2) = eval e1 + eval e
print :: Exp -> String
print (Lit n) = show n
print (Add e1 e2) =
  if eval e2 == 0 -- dependency on eval
  then print e1
  else "(" ++ print e1 ++ "+" ++ print e2 ")
```

- Conventional object-oriented programming and functional programming suffer from the **Expression Problem** [Wadler 1998]
- Dealing with dependencies modularly poses extra challenges
- Existing design patterns partly address these problems
  - E.g. Object Algebras [Oliveira & Cook 2012], Polymorphic Embedding [Hofer et al. 2008], Cake pattern [Odersky & Zenger 2005], Finally Tagless [Carette et al. 2009], Datatypes a la carte [Swierstra 2008]
  - Lack of proper mechanisms for modular dependencies and compositions
  - Heavily parameterized and boilerplate code

Contributions

- **Compositional Programming**: A new statically-typed modular programming style
  - Solving the Expression Problem and dealing with modular programs with complex dependencies
- **CP**: A language design for Compositional Programming
  - Elaborated to F_i^+ [Bi et al., 2019], a recent calculus that supports *disjoint intersection types* [Oliveira et al. 2016], *disjoint polymorphism* [Alpuim et al. 2017] and *nested composition* [Bi et al. 2018]
  - We proved that the elaboration is type-safe and coherent
- Attribute Grammars in **CP**
  - Inspired by Rendel et al. [2014]'s encoding but without explicit definitions of composition operators
- Polymorphic contexts
  - Allowing for modular contexts in modular components
- Implementation, case studies, and examples

Solving the Expression Problem: Operation Extensions

```
type ExpSig<Exp> = {
  Lit : Int -> Exp;
  Add : Exp -> Exp -> Exp;
};
type Eval = { eval : Int };
evalNum = trait implements ExpSig<Eval> => {
  (Lit     n).eval = n;
  (Add e1 e2).eval = e1.eval + e2.eval;
};
type Print = { print : String };
printNum = trait implements ExpSig<Print> => {
  (Lit     n).print = n.toString;
  (Add e1 e2).print = "(" ++ e1.print ++ "+" ++ e2.print ++ ")";
};
expAdd Exp = trait [self : ExpSig<Exp>] => {
  test = new Add (new Lit 4) (new Lit 8);
};
e = new evalNum ,, printNum ,, expAdd @(Eval&Print);
e.test.print ++ " is " ++ e.test.eval.toString  --> "(4+8) is 12"
```

Compositional interfaces

First-class traits

Method patterns

Self-type annotations

Nested trait composition

Solving the Expression Problem: Variant Extensions

```
type MulSig<Exp> extends ExpSig<Exp> = {
  Mul : Exp -> Exp -> Exp;
};
```

```
evalMul = trait implements MulSig<Eval> inherits evalNum => {
  (Mul e1 e2).eval = e1.eval * e2.eval;
};
printMul = trait implements MulSig<Print> inherits printNum => {
  (Mul e1 e2).print = "(" ++ e1.print ++ "*" ++ e2.print ++ ")";
};
```

Inheritance

```
expMul Exp = trait [self : MulSig<Exp>] inherits expAdd @Exp => {
  override test = new Mul super.test (new Lit 4);
};
```

Overriding

```
e' = new evalMul ,, printMul ,, expMul @(Eval&Print);
e'.test.print ++ " is " ++ e'.test.eval.toString  --> "((4+8)*4) is 48"
```

Dependencies and S-attributed Grammars

- **CP** can deal with programs with complex dependencies *modularly*
  - **Child dependencies**: attributes depend on other synthesized attributes of the children

```
printInh = trait implements ExpSig<Eval&Print> inherits evalNum => {
  (Lit     n).print = n.toString;
  (Add e1 e2).print = if e2.eval == 0 then e1.print
                      else "(" ++ e1.print ++ "+" ++ e2.print ++ ")";
};
```

Strong dependency

```
printChild = trait implements ExpSig<Eval % Print> => {
  (Lit     n).print = n.toString;
  (Add e1 e2).print = if e2.eval == 0 then e1.print
                      else "(" ++ e1.print ++ "+" ++ e2.print ++ ")";
};
new printChild ,, expAdd @Print                       -- Type Error!
new printChild ,, evalNum ,, expAdd @(Print&Eval)  -- OK!
```

Weak dependency

Dependencies and S-attributed Grammars

- **Self dependencies**: attributes depend on other synthesized attributes of the self-reference

```
printSelf = trait implements ExpSig<Eval % Print> => {
  (Lit     n              ).print = n.toString;
  (Add e1 e2 [self:Eval]).print = if self.eval == 0 then "0"
                                  else "(" ++ e1.print ++ "+" ++ e2.print ++ ")";
};
```

- **Mutual dependencies**: two attributes are inter-defined

```
type PrintAux = { printAux : String };
printMutual = trait implements ExpSig<PrintAux % Print> => {
  (Lit     n).print = n.toString;
  (Add e1 e2).print = e1.printAux ++ "+" ++ e2.printAux;
};
printAux = trait implements ExpSig<Print % PrintAux> => {
  (Lit     n [self:Print]).printAux = self.print;
  (Add e1 e2 [self:Print]).printAux = "(" ++ self.print ++ ")";
};
```

Context Evolution

- **Problem**: different modular components may require different contexts

```
type Eval = { eval : EnvN -> EnvF -> Int };
evalNum = trait implements ExpSig<Eval> => {
  (Lit     n).eval (envN:EnvN) (envF:EnvF) = n;
  (Add e1 e2).eval (envN:EnvN) (envF:EnvF) = e1.eval envN envF + e2.eval envN envF;
};
evalVar = trait implements VarSig<Eval> => {
  (Let s e1 e2).eval (envN:EnvN) (envF:EnvF) =
    e2.eval (insert @Int s (e1.eval envN envF) envN) envF;
  (Var       s).eval (envN:EnvN) (envF:EnvF) = lookup @Int s envN;
};
evalFunc = trait implements FuncSig<Eval> => {
  (LetF s f e).eval (envN:EnvN) (envF:EnvF) = e.eval envN (insert @Func s f envF);
  (AppF  s  e).eval (envN:EnvN) (envF:EnvF) = (lookup @Func s envF) (e.eval envN envF);
};
```

- **Highly non-modular**: existing code has to be modified when a new context is needed
- **Not encapsulating contexts**: contexts are fully exposed even if not directly used

Polymorphic Contexts

- Allowing **modular** & **encapsulated** contexts

```
type Eval Context = { eval : Context -> Int };
evalNum Context = trait implements ExpSig<Eval Context> => {
  (Lit     n).eval (ctx:Context) = lookup @Int "foobar" ctx;  -- Type Error!
  (Add e1 e2).eval (ctx:Context) = e1.eval ctx + e2.eval ctx;
};
```

Disjoint polymorphism

```
type CtxN = { envN : EnvN };
evalVar (Context * CtxN) = trait implements VarSig<Eval (CtxN&Context)> => {
  (Let s e1 e2).eval (ctx:CtxN&Context) =
    e2.eval ({ envN = insert @Int s (e1.eval ctx) ctx.envN } ,, ctx:Context);
  (Var       s).eval (ctx:CtxN&Context) = lookup @Int s ctx.envN;
};
type CtxF = { envF : EnvF };
evalFunc (Context * CtxF) = trait implements FuncSig<Eval (CtxF&Context)> => {
  (LetF s f e).eval (ctx:CtxF&Context) =
    e.eval ({ envF = insert @Func s f ctx.envF } ,, ctx:Context);
  (AppF  s  e).eval (ctx:CtxF&Context) = (lookup @Func s ctx.envF) (e.eval ctx);
};
```

Polymorphic Contexts

- Composing the components with different contexts **modularly**

```
evalNum Context = trait implements ExpSig<Eval Context> =>
evalVar (Context * CtxN) = trait implements VarSig<Eval (CtxN&Context)> =>
evalFunc (Context * CtxF) = trait implements FuncSig<Eval (CtxF&Context)> =>
```

```
expPoly Exp = trait [self : ExpSig<Exp>&VarSig<Exp>&FuncSig<Exp>] => {
  test = new LetF "f" (\(x:Int) -> x * x)
                 (new Let "x" (new Lit 9) (new AppF "f" (new Var "x")));
};
```

```
e = new evalNum @(CtxN&CtxF) ,, evalVar @CtxF ,, evalFunc @CtxN ,,
        expPoly @(Eval (CtxN&CtxF));
```

```
e.test.eval { envN = empty @Int, envF = empty @Func }  --> 81
```

Formal Syntax

Program	P	$::=$	$D; P \mid E$
Declarations	D	$::=$	$M \mid \textbf{type } X\langle\overline{\alpha}\rangle \textbf{ extends } A = B$
Term declarations	M	$::=$	$x = E \mid (L\ \overline{x : A}\ [\textbf{self} : B]).\ell = E$
Types	A, B	$::=$	$\textbf{Int} \mid \alpha \mid \top \mid \bot \mid A \to B \mid \forall(\alpha * A).B \mid A \,\&\, B \mid \{\ell : A\}$
		$\mid$	$\textbf{Trait}[A, B] \mid X\langle\overline{S}\rangle$
Sorts	S	$::=$	$A \mid A \,\%\, B$
Expressions	E	$::=$	$i \mid x \mid \top \mid \lambda x.E \mid E_1\ E_2 \mid \Lambda(\alpha * A).E \mid E\ @A \mid E_1 ,,\ E_2 \mid \{\overline{M}\} \mid E.\ell$
		$\mid$	$E : A \mid \textbf{let } x : A = E_1 \textbf{ in } E_2 \mid \textbf{open } E_1 \textbf{ in } E_2 \mid \textbf{new } E \mid E_1 \hat{}\, E_2$
		$\mid$	$\textbf{trait}[\textbf{self} : A] \textbf{ implements } B \textbf{ inherits } E_1 \Rightarrow E_2$

Source CP program

Elaboration

Declarations, Sorts & Trait-related constructs

Target F_i^+ expression

Types	τ	$::=$	$\textbf{Int} \mid \alpha \mid \top \mid \bot \mid \tau_1 \to \tau_2 \mid \forall(\alpha * \tau_1).\tau_2 \mid \tau_1 \,\&\, \tau_2 \mid \{\ell : \tau\}$
Expressions	e	$::=$	$i \mid x \mid \top \mid \lambda x.e \mid e_1\ e_2 \mid \Lambda(\alpha * \tau).e \mid e\ \tau \mid e_1 ,,\ e_2 \mid \{\ell = e\} \mid e.\ell$
		$\mid$	$\textbf{let } x : \tau = e_1 \textbf{ in } e_2$

Elaborating Compositional Interfaces and Sorts

- The elaboration builds on ideas from generalized Object Algebras [Oliveira et al. 2013] and the denotational model of inheritance [Cook and Palsberg, 1989]

```
type ExpSig<Exp> = {
  Lit : Int -> Exp;
  Add : Exp -> Exp -> Exp;
};
```

```
type MulSig<Exp> extends ExpSig<Exp> = {
  Mul : Exp -> Exp -> Exp;
};
```

↓

```
type ExpSig Exp OExp =
  { Lit : Int -> Trait[Exp,OExp] } &
  { Add : Exp -> Exp -> Trait[Exp,OExp] }
```

```
type MulSig Exp OExp =
  { Lit : Int -> Trait[Exp,OExp] } &
  { Add : Exp -> Exp -> Trait[Exp,OExp] } &
  { Mul : Exp -> Exp -> Trait[Exp,OExp] };
```

Elaborating Traits

```
evalNum = trait implements ExpSig<Eval> => {
  (Lit     n).eval = n;
  (Add e1 e2).eval = e1.eval + e2.eval;
};
```

↓

```
evalNum = trait [self: Top] implements ExpSig Eval Eval => open self in
  { Lit = \(n: Int) -> trait => { eval = n } } ,,
  { Add = \(e1: Eval) -> \(e2: Eval) -> trait => { eval = e1.eval + e2.eval} };
```

↓

```
let evalNum = \(self: Top) ->
  { Lit = \(n: Int) -> \(self: Top) -> { eval = n } } ,,
  { Add = \(e1: Eval) -> \(e2: Eval) -> \(self: Top) -> { eval = e1.eval + e2.eval } }
in ...
```

Elaborating Child Dependencies

```
printChild = trait implements ExpSig<Eval % Print> => {
  (Lit     n).print = n.toString;
  (Add e1 e2).print = if e2.eval == 0 then e1.print
                      else "(" ++ e1.print ++ "+" ++ e2.print ++ ")";
};
```

```
printChild = trait [self: Top] implements ExpSig (Eval&Print) Print => open self in
  { Lit (n: Int) = trait => { print = n.toString } } ,,
  { Add (e1: Eval&Print) (e2: Eval&Print) = trait =>
    { print = if e2.eval == 0 then e1.print
              else "(" ++ e1.print ++ "+" ++ e2.print ++ ")" } };
```

Elaborating Self-type Annotations

```
expAdd Exp = trait [self : ExpSig<Exp>] => {
  test = new Add (new Lit 4) (new Lit 8);
};
```

↓

```
expAdd = /\Exp. trait [self: ExpSig Exp Exp] => open self in {
  test = new Add (new Lit 4) (new Lit 8);
};
```

↓

```
let expAdd =
  /\Exp. \(self : { Lit : Int -> Exp -> Exp } & { Add : Exp -> Exp -> Exp -> Exp }) ->
    let Add = self.Add
    in let Lit = self.Lit
    in { test = letrec self : Exp = Add (letrec self : Exp = Lit 4 self in self)
                                        (letrec self : Exp = Lit 8 self in self)
                                        self
                in self }
in ...
```

Elaborating Inheritance and Overriding

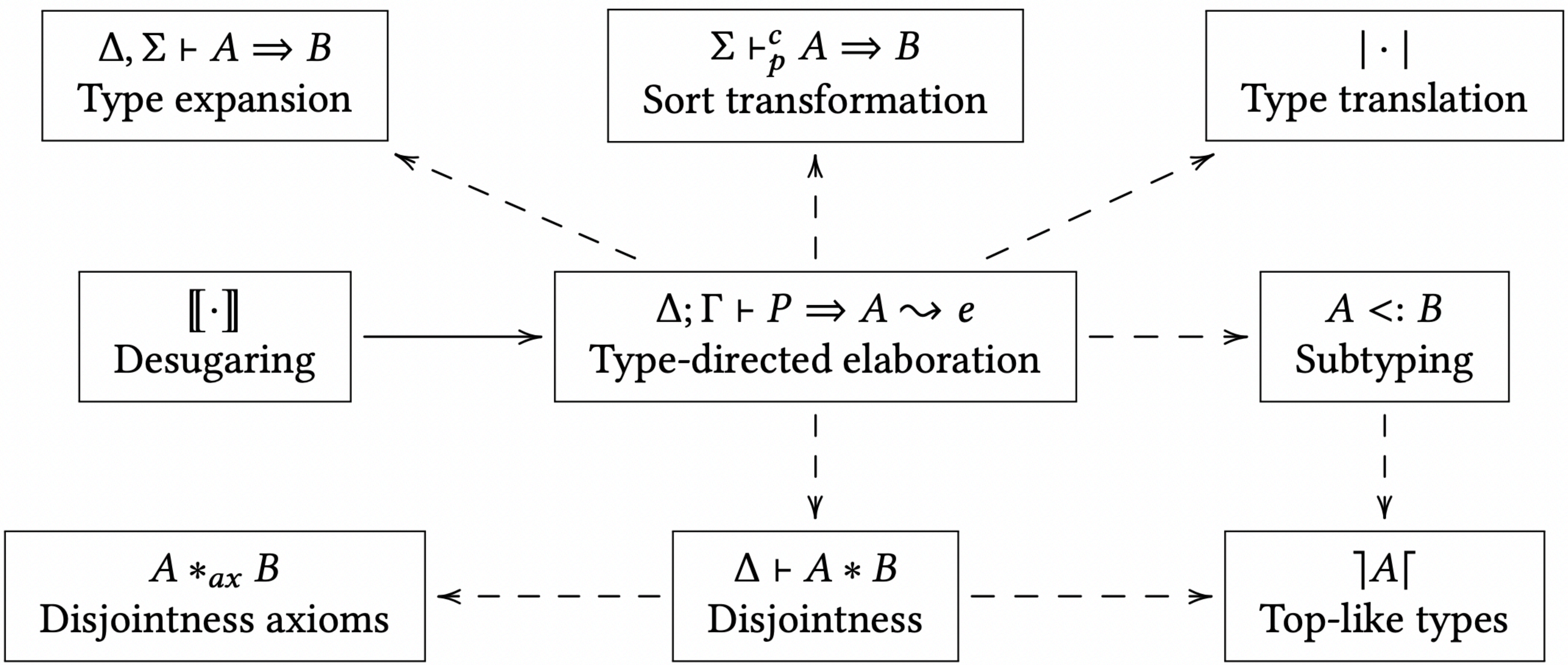
```
expMul Exp = trait [self : MulSig<Exp>] inherits expAdd @Exp => {
  override test = new Mul super.test (new Lit 4);
};
```

↓

```
let expMul = /\ Exp. \(self : { Lit : Int -> Exp -> Exp } &
                              { Add : Exp -> Exp -> Exp -> Exp } &
                              { Mul : Exp -> Exp -> Exp -> Exp }) ->
  let super = (expAdd Exp) self
  in (super : Top) ,,
    let Add = self.Add
    in let Lit = self.Lit
    in let Mul = self.Mul
    in { test = letrec self : Exp = Mul super.test
                                        (letrec self : Exp = Lit 4 self in self)
                                        self
              in self }
in ...
```

Elaboration Overview

Term contexts	Γ	$::=$	$\bullet \mid \Gamma, x : A$
Type contexts	Δ	$::=$	$\bullet \mid \Delta, \alpha * A \mid \Delta, X\langle\overline{\alpha}, \overline{\beta}\rangle \mapsto A$
Sort contexts	Σ	$::=$	$\bullet \mid \Sigma, \alpha \mapsto \beta$



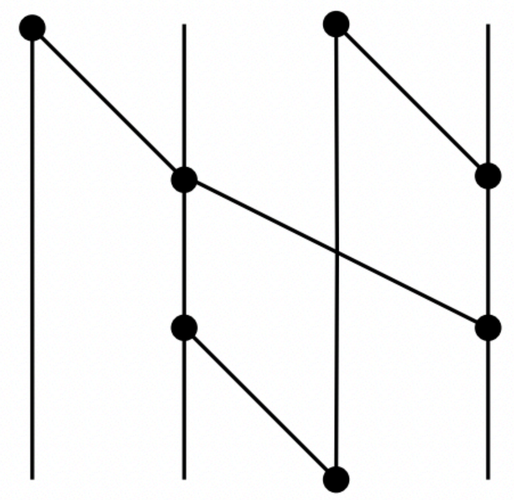
Metatheory

- We have proved that the elaboration of **CP** into F_i^+ is type-safe and coherent
- Type-safety theorem
  - *If* $\Delta; \Gamma \vdash P \Rightarrow A \rightsquigarrow e$ *then* $|\Delta|; |\Gamma| \vdash e \Rightarrow |A|$
- Coherence theorem
  - Each well-typed CP program has a unique elaboration

Case Studies: Scans

- A DSL for parallel prefix circuits [Hinze 2004]

```
type CircuitSig<Circuit> = {
  Identity : Int -> Circuit;
  Fan      : Int -> Circuit;
  Above    : Circuit -> Circuit -> Circuit;
  Beside   : Circuit -> Circuit -> Circuit;
  Stretch  : (List Int) -> Circuit -> Circuit;
};
```



  - Interpretations: **width**, **depth**, **wellSized** and **layout** (depending on **width**)
  - Variant extension: **RStretch**
- Most compact and modular w.r.t existing implementations

Language	Haskell [Gibbons & Wu, 2014]	Scala [Zhang & Oliveira, 2019]	F_i^+ [Bi et al., 2019]	**CP**
SLOC	87	129	72	70

Case Studies: Mini Interpreter

- A mini interpreter for an expression language (~700 SLOC)
  - Including numeric and boolean literals, arithmetic expressions, logical expressions, comparisons, branches, variable bindings, function closures …
  - Sublanguages are **separately** defined as **features** that can be arbitrarily combined to form *a* **product line of interpreters**
- Examine the ability to model non-trivial dependencies and multi-sorted languages

	Operation			
Dependency	eval	print	print(aux)	log
Child dependencies		✓		
Self dependencies		✓		✓
Mutual dependencies			✓	
Inherited attributes	✓			

```
type CmpSig<Boolean,Numeric> = {
  Eq  : Numeric -> Numeric -> Boolean;
  Cmp : Numeric -> Numeric -> Numeric;
  -- other constructors are omitted
};
```

Case Studies: C0 Compiler

- An educational one-pass compiler
  - A subset of C compiled to Java bytecode
  - Originally written in Java with semantics hardcoded in the parser, thus is non-modular
- Rendel et al. [2014] modularized C0 using generalized Object Algebras
- Comparison

Java (Aarhus University)	SLOC	**Scala** (Rendel et al. [2014])	SLOC	**CP**	SLOC
Entangled Compiler	235	Generic	140	Maybe Algebra	12
(Tokenizer excluded)		Trees, Signatures and Combinators	558	Compositional Interfaces	32
		Composition and Assembly	101		
		Attribute Interfaces	32	Attribute Interfaces	8
		Algebra Implementations	191	Trait Implementations	216
Bytecode (Reformatted)	25	Bytecode Prelude	25	Bytecode Prelude	25
Main	14	Main	5	Main Example	21
Total	274	**Total**	1,052	**Total**	314

Future Work

- There is a lot of room for making **CP** more expressive and practical
    - Recursive types and type constructors
    - Mutable states
    - Type inference

Conclusion

- We have presented key concepts of **Compositional Programming** and a language design called **CP**
  - Offering an alternative style to FP and OOP
  - Allowing programs with *non-trivial dependencies* to be modularized in a natural way
  - Applicability demonstrated by various examples and case studies
- Artifact is available at

  https://github.com/wxzh/CP

Thank you!