香 港 大 學

THE UNIVERSITY OF HONG KONG

Shallow EDSLs and Object-Oriented Programming: Beyond Simple Compositionality

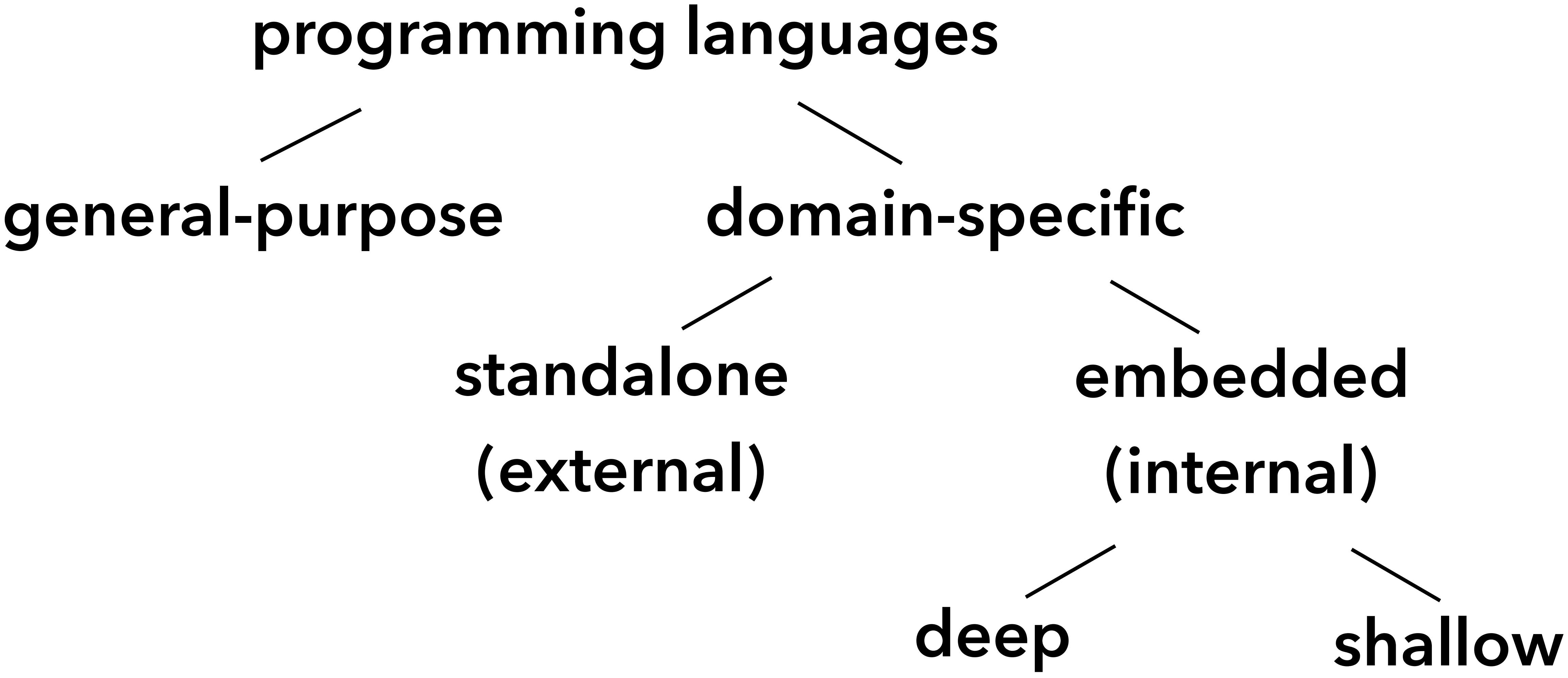
---

Weixin Zhang and Bruno C. d. S. Oliveira

<Programming> 2019

April 3, 2019

Background



programming languages

general-purpose

domain-specific

standalone (external)

embedded (internal)

deep

shallow

Shallow vs. deep embeddings

- Shallow embeddings
  - Semantics first
  - Compositional
  - No AST
  - Easy to add new language constructs
  - Hard to add new interpretations

- Deep embeddings
  - Syntax first
  - Non-compositional
  - Have an AST
  - Easy to add new interpretations
  - Hard to add new language constructs

Contribution

- Shallow embeddings and OOP are closely related
  - Both essence is **procedural abstraction** [Reynolds,1978]

Gibbons & Wu, 2015

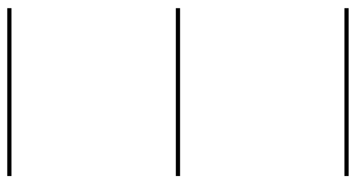
Cook, 2009

- OOP mechanisms, **subtyping**, **inheritance** and **type-refinement** increase the modularity of shallow EDSLs
  - Enable multiple (possibly dependent) interpretations

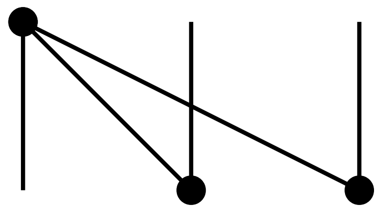
SCANS: a DSL for parallel prefix circuits

- Grammar: ⟨*circuit*⟩ ::= ‘id’ ⟨*positive-number*⟩
  | ‘fan’ ⟨*positive-number*⟩
  | ⟨*circuit*⟩ ‘beside’ ⟨*circuit*⟩
  | ⟨*circuit*⟩ ‘above’ ⟨*circuit*⟩
  | ‘stretch’ ⟨*positive-numbers*⟩ ⟨*circuit*⟩
  | ‘(’ ⟨*circuit*⟩ ‘)’

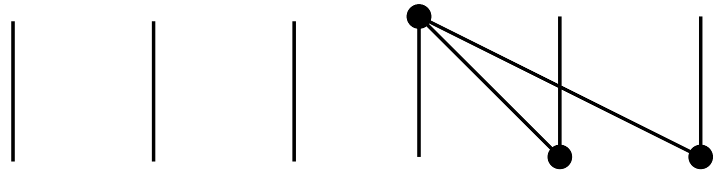
id 3



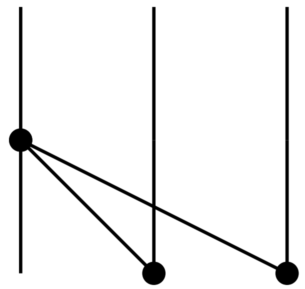
fan 3



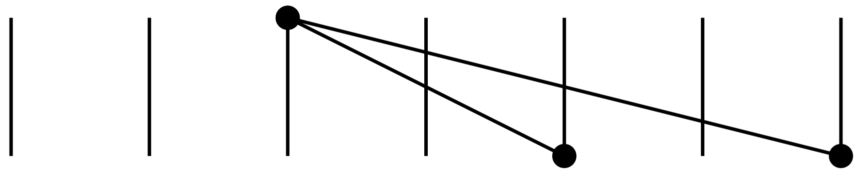
id 3 beside fan 3

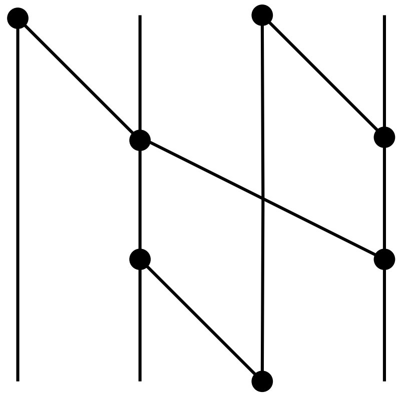


id 3 above fan 3



stretch 3 2 3 fan 3





(fan 2 beside fan 2)
above
(stretch 2 2 fan 2)
above
(id 1 beside fan 2 beside id 1)

Embedding Scans in Haskell

- A shallow implementation should conform to the following signatures

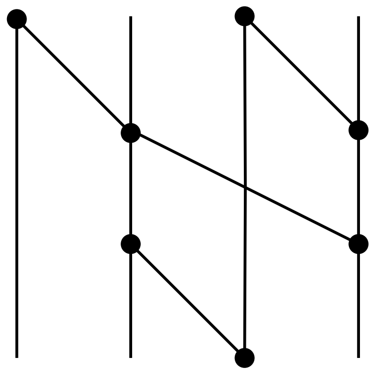
```
type Circuit = ...
id          :: Int → Circuit
fan         :: Int → Circuit
beside      :: Circuit → Circuit → Circuit
above       :: Circuit → Circuit → Circuit
stretch     :: [Int] → Circuit → Circuit
```

semantic domain

procedural abstraction

- E.g. an interpretation calculating the **width**

```
type Circuit = Int
id n           = n
fan n          = n
beside c1 c2   = c1 + c2
above c1 c2    = c1
stretch ns c   = sum ns
```



```
> ((fan 2 ‘beside‘ fan 2) ‘above‘
  | stretch [2,2] (fan 2) ‘above‘
  | (id 1 ‘beside‘ fan 2 ‘beside‘ id 1))
4
```

Towards OOP

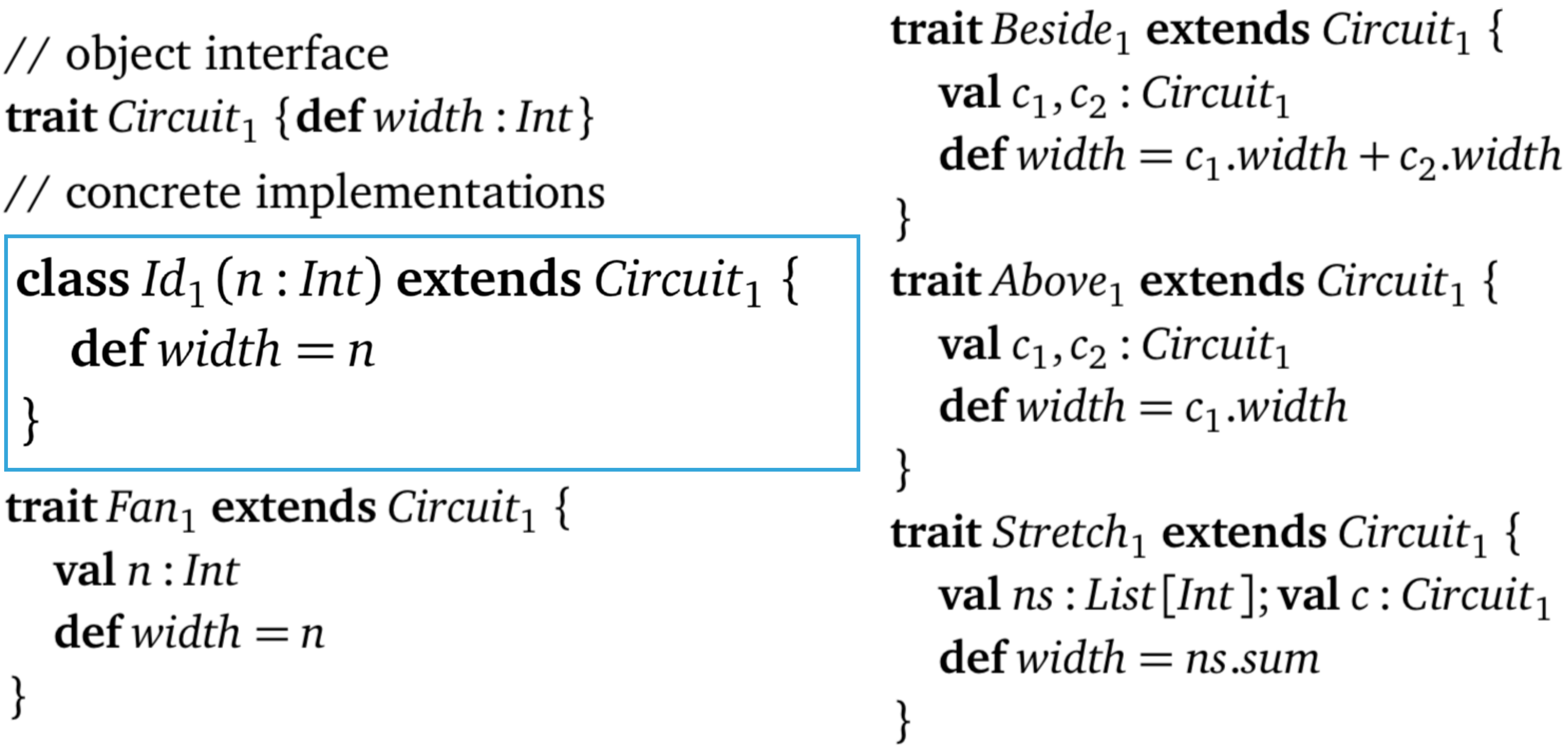
- An *isomorphic* encoding of **width**

$$\begin{aligned}
&\textbf{type}\ \mathit{Circuit} &&= \mathit{Int}\\
&\mathit{id}\ n &&= n\\
&\mathit{fan}\ n &&= n\\
&\mathit{beside}\ c_1\ c_2 &&= c_1 + c_2\\
&\mathit{above}\ c_1\ c_2 &&= c_1\\
&\mathit{stretch}\ \mathit{ns}\ c &&= \mathit{sum}\ \mathit{ns}
\end{aligned}$$

$$\begin{aligned}
&\textbf{newtype}\ \mathit{Circuit}_1 &&= \mathit{Circuit}_1\ \{\mathit{width}_1 :: \mathit{Int}\}\\
&\mathit{id}_1\ n &&= \mathit{Circuit}_1\ \{\mathit{width}_1 = n\}\\
&\mathit{fan}_1\ n &&= \mathit{Circuit}_1\ \{\mathit{width}_1 = n\}\\
&\mathit{beside}_1\ c_1\ c_2 &&= \mathit{Circuit}_1\ \{\mathit{width}_1 = \mathit{width}_1\ c_1 + \mathit{width}_1\ c_2\}\\
&\mathit{above}_1\ c_1\ c_2 &&= \mathit{Circuit}_1\ \{\mathit{width}_1 = \mathit{width}_1\ c_1\}\\
&\mathit{stretch}_1\ \mathit{ns}\ c &&= \mathit{Circuit}_1\ \{\mathit{width}_1 = \mathit{sum}\ \mathit{ns}\}
\end{aligned}$$

Embedding SCANS in OOP

- It is easy to port the definition into an OOP language like Scala



```
// object interface
trait Circuit1 {def width : Int}
// concrete implementations
class Id1 (n : Int) extends Circuit1 {
  def width = n
}
trait Fan1 extends Circuit1 {
  val n : Int
  def width = n
}
trait Beside1 extends Circuit1 {
  val c1, c2 : Circuit1
  def width = c1.width + c2.width
}
trait Above1 extends Circuit1 {
  val c1, c2 : Circuit1
  def width = c1.width
}
trait Stretch1 extends Circuit1 {
  val ns : List[Int]; val c : Circuit1
  def width = ns.sum
}
```

Smart constructors

- Smart constructors are needed for building a circuit object conveniently

$$
\begin{array}{lll}
\textbf{def}\ \mathit{id}(x:\mathit{Int}) & = \textbf{new}\ \mathit{Id}_1 & \{\textbf{val}\ n = x\} \\
\textbf{def}\ \mathit{fan}(x:\mathit{Int}) & = \textbf{new}\ \mathit{Fan}_1 & \{\textbf{val}\ n = x\} \\
\textbf{def}\ \mathit{beside}(x:\mathit{Circuit}_1, y:\mathit{Circuit}_1) & = \textbf{new}\ \mathit{Beside}_1 & \{\textbf{val}\ c_1 = x; \textbf{val}\ c_2 = y\} \\
\textbf{def}\ \mathit{above}(x:\mathit{Circuit}_1, y:\mathit{Circuit}_1) & = \textbf{new}\ \mathit{Above}_1 & \{\textbf{val}\ c_1 = x; \textbf{val}\ c_2 = y\} \\
\textbf{def}\ \mathit{stretch}(x:\mathit{Circuit}_1, \mathit{xs}:\mathit{Int}*) & = \textbf{new}\ \mathit{Stretch}_1 & \{\textbf{val}\ \mathit{ns} = \mathit{xs}.\mathit{toList}; \textbf{val}\ c = x\}
\end{array}
$$

- Constructing the example circuit again

$$
\begin{array}{l}
\textbf{val}\ \mathit{circuit} = \mathit{above}(\mathit{beside}(\mathit{fan}(2), \mathit{fan}(2)), \\
\qquad\qquad\qquad\qquad \mathit{above}(\mathit{stretch}(\mathit{fan}(2), 2, 2), \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{beside}(\mathit{beside}(\mathit{id}(1), \mathit{fan}(2)), \mathit{id}(1))))
\end{array}
$$

$> \mathit{circuit}.\mathit{width}$

4

Multiple interpretations in Haskell

- Often claimed as a limitation of shallow embedding
- Typical workaround is to use tuples
  - e.g. additionally supporting **depth** for SCANS

$$\begin{aligned}
&\mathbf{type}\ \mathit{Circuit}_2 &&= (\mathit{Int}, \mathit{Int}) \\
&\mathit{id}_2\ n &&= (n, 0) \\
&\mathit{fan}_2\ n &&= (n, 1) \\
&\mathit{above}_2\ c_1\ c_2 &&= (\mathit{width}\ c_1, \mathit{depth}\ c_1 + \mathit{depth}\ c_2) \\
&\mathit{beside}_2\ c_1\ c_2 &&= (\mathit{width}\ c_1 + \mathit{width}\ c_2, \mathit{depth}\ c_1\ \text{`}\mathit{max}\text{`}\ \mathit{depth}\ c_2) \\
&\mathit{stretch}_2\ \mathit{ns}\ c &&= (\mathit{sum}\ \mathit{ns}, \mathit{depth}\ c)
\end{aligned}$$

$$\begin{aligned}
\mathit{width} &= \mathit{fst} \\
\mathit{depth} &= \mathit{snd}
\end{aligned}$$

- However, this implementation is *non-modular*

Multiple interpretations in Scala

- Multiple interpretations can be modular with Scala

Subtyping

trait $Circuit_2$ **extends** $Circuit_1$ $\{$**def** $depth : Int\}$ // extended semantic domain
trait Id_2 **extends** Id_1 **with** $Circuit_2$ $\{$**def** $depth = 0\}$
trait Fan_2 **extends** Fan_1 **with** $Circuit_2$ $\{$**def** $depth = 1\}$
trait $Above_2$ **extends** $Above_1$ **with** $Circuit_2$ $\{$
  override val $c_1, c_2 : Circuit_2$ // type-refinement that allows depth invocations
  def $depth = c_1.depth + c_2.depth$
$\}$

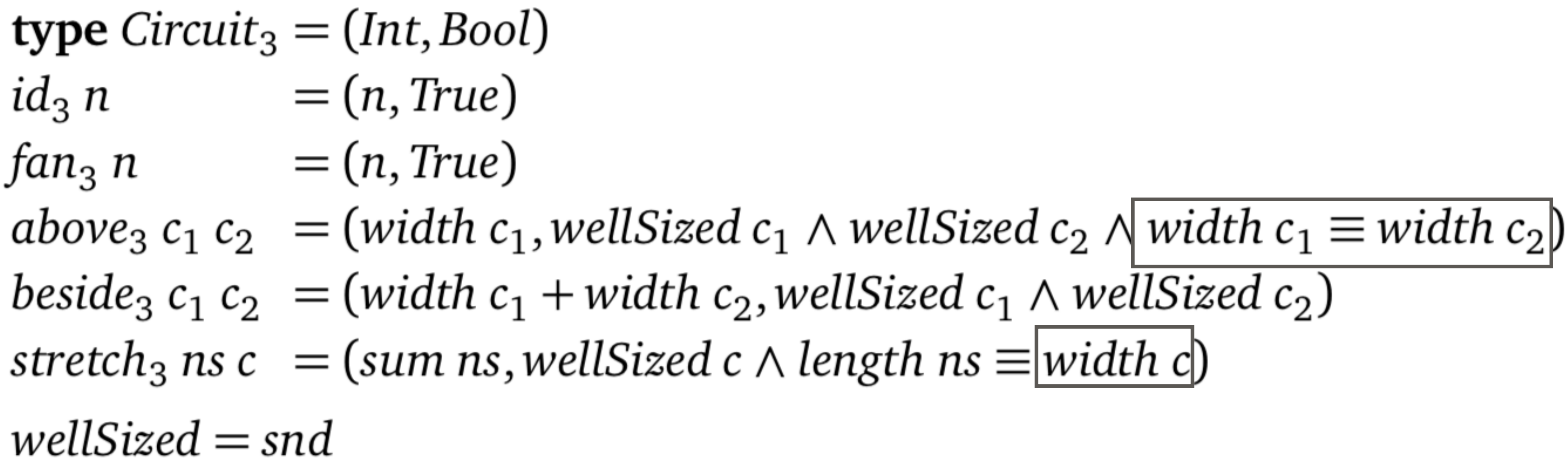
Type-refinement

Inheritance

trait $Beside_2$ **extends** $Beside_1$ **with** $Circuit_2$ $\{$
  override val $c_1, c_2 : Circuit_2$ // type-refinement that allows depth invocations
  def $depth = Math.max\,(c_1.depth, c_2.depth)$
$\}$
trait $Stretch_2$ **extends** $Stretch_1$ **with** $Circuit_2$ $\{$
  override val $c : Circuit_2$ // type-refinement that allows depth invocations
  def $depth = c.depth$
$\}$

Dependent interpretations in Haskell

- An interpretation depends not *only* on itself but also on *other* interpretations
  - E.g. **wellSized**, which depends on **width**



$$\begin{aligned}
\textbf{type}\ Circuit_3 &= (Int, Bool)\\
id_3\ n &= (n, True)\\
fan_3\ n &= (n, True)\\
above_3\ c_1\ c_2 &= (width\ c_1, wellSized\ c_1 \wedge wellSized\ c_2 \wedge \boxed{width\ c_1 \equiv width\ c_2})\\
beside_3\ c_1\ c_2 &= (width\ c_1 + width\ c_2, wellSized\ c_1 \wedge wellSized\ c_2)\\
stretch_3\ ns\ c &= (sum\ ns, wellSized\ c \wedge length\ ns \equiv \boxed{width\ c})
\end{aligned}$$

$$wellSized = snd$$

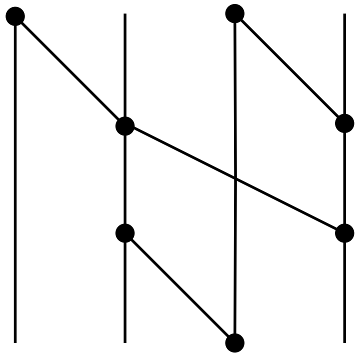
Dependent interpretations in Scala

- Again, modular dependent interpretations are unproblematic in Scala

trait $Circuit_3$ **extends** $Circuit_1$ $\{$**def** $wellSized : Boolean\}$ // extended semantic domain
trait Id_3 **extends** Id_1 **with** $Circuit_3$ $\{$**def** $wellSized = true\}$
trait Fan_3 **extends** Fan_1 **with** $Circuit_3$ $\{$**def** $wellSized = true\}$
trait $Above_3$ **extends** $Above_1$ **with** $Circuit_3$ $\{$
    override val $c_1, c_2 : Circuit_3$
    def $wellSized =$
        $c_1.wellSized \wedge c_2.wellSized \wedge \boxed{c_1.width \equiv c_2.width}$ // width dependency
$\}$
trait $Beside_3$ **extends** $Beside_1$ **with** $Circuit_3$ $\{$
    override val $c_1, c_2 : Circuit_3$
    def $wellSized = c_1.wellSized \wedge c_2.wellSized$
$\}$
trait $Stretch_3$ **extends** $Stretch_1$ **with** $Circuit_3$ $\{$
    override val $c : Circuit_3$
    def $wellSized = c.wellSized \wedge \boxed{ns.length \equiv c.width}$ // width dependency
$\}$

Context-sensitive interpretations in Haskell

- An interpretation relies on some context
  - e.g. **layout**



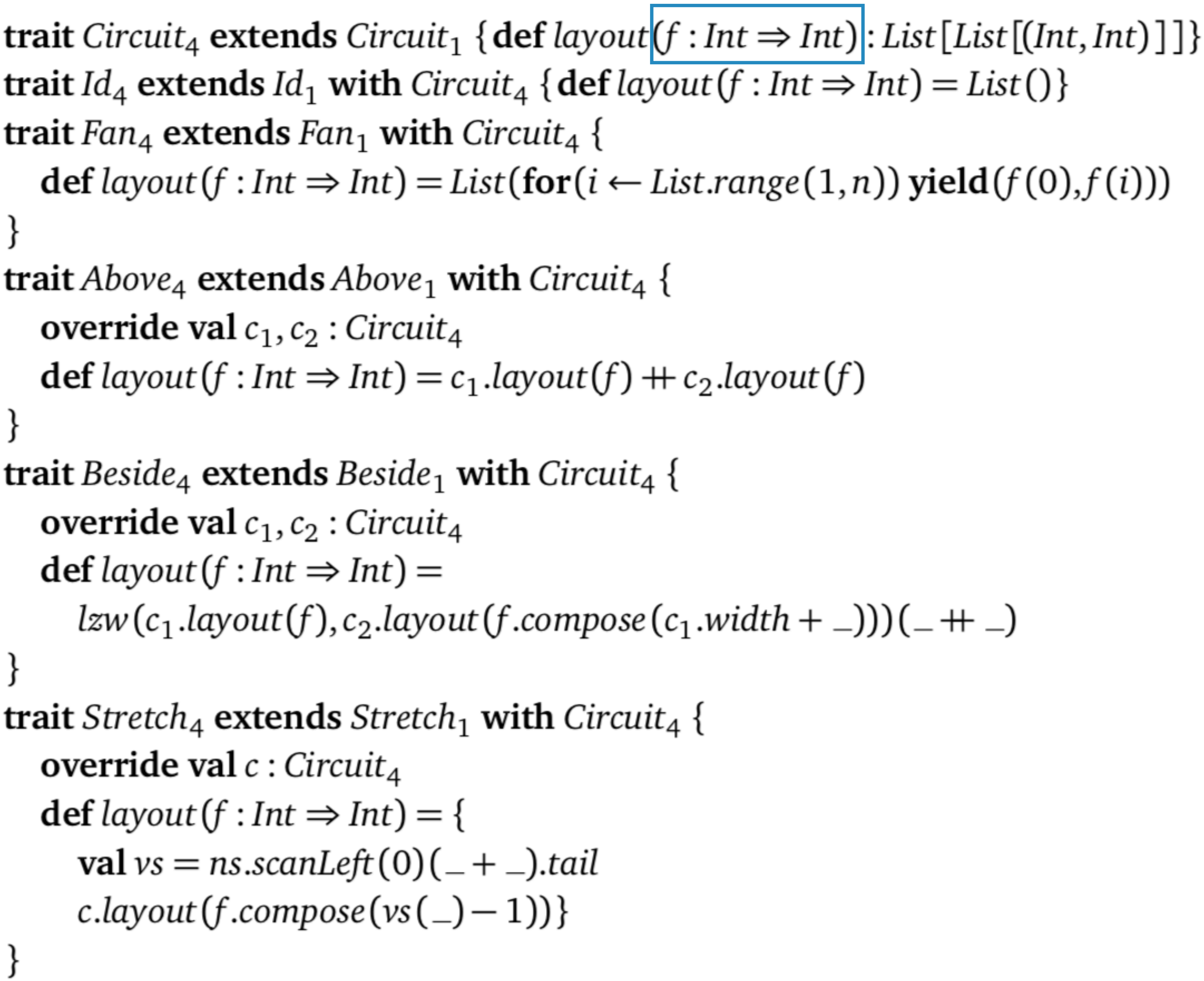
$[[(0,1),(2,3)],[(1,3)],[(1,2)]]$

accumulating parameter

$$\begin{aligned}
\textbf{type}\ \mathit{Circuit}_4 &= (\mathit{Int}, \boxed{(\mathit{Int} \to \mathit{Int})} \to [[(\mathit{Int},\mathit{Int})]])\\
\mathit{id}_4\ n &= (n, \lambda f \to [\,])\\
\mathit{fan}_4\ n &= (n, \lambda f \to [[(f\ 0, f\ j) \mid j \leftarrow [1\,..\,n-1]]])\\
\mathit{above}_4\ c_1\ c_2 &= (\mathit{width}\ c_1, \lambda f \to \mathit{layout}\ c_1\ f \mathbin{+\!\!+} \mathit{layout}\ c_2\ f)\\
\mathit{beside}_4\ c_1\ c_2 &= (\mathit{width}\ c_1 + \mathit{width}\ c_2,\\
&\qquad \lambda f \to \mathit{lzw}\ (\mathbin{+\!\!+})\ (\mathit{layout}\ c_1\ f)\ (\mathit{layout}\ c_2\ (f \circ (\mathit{width}\ c_1 +))))\\
\mathit{stretch}_4\ \mathit{ns}\ c &= (\mathit{sum}\ \mathit{ns}, \lambda f \to \mathit{layout}\ c\ (f \circ \mathit{pred} \circ (\mathit{scanl1}\ (+)\ \mathit{ns}!!)))
\end{aligned}$$

$$\mathit{layout} = \mathit{snd}$$

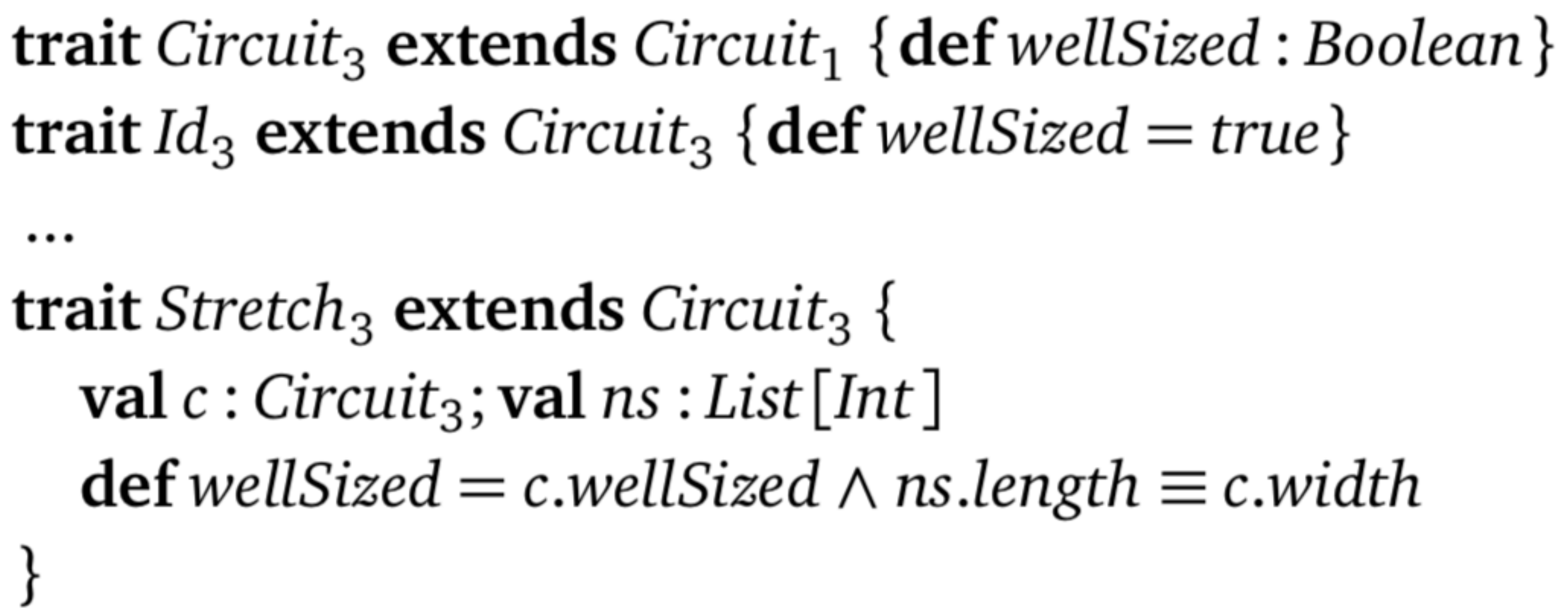
Context-sensitive interpretations in Scala



```
trait Circuit4 extends Circuit1 {def layout(f : Int ⇒ Int) : List[List[(Int, Int)]]}
trait Id4 extends Id1 with Circuit4 {def layout(f : Int ⇒ Int) = List()}
trait Fan4 extends Fan1 with Circuit4 {
  def layout(f : Int ⇒ Int) = List(for (i ← List.range(1, n)) yield (f(0), f(i)))
}
trait Above4 extends Above1 with Circuit4 {
  override val c1, c2 : Circuit4
  def layout(f : Int ⇒ Int) = c1.layout(f) ++ c2.layout(f)
}
trait Beside4 extends Beside1 with Circuit4 {
  override val c1, c2 : Circuit4
  def layout(f : Int ⇒ Int) =
    lzw(c1.layout(f), c2.layout(f.compose(c1.width + _)))(_ ++ _)
}
trait Stretch4 extends Stretch1 with Circuit4 {
  override val c : Circuit4
  def layout(f : Int ⇒ Int) = {
    val vs = ns.scanLeft(0)(_ + _).tail
    c.layout(f.compose(vs(_) − 1))}
}
```

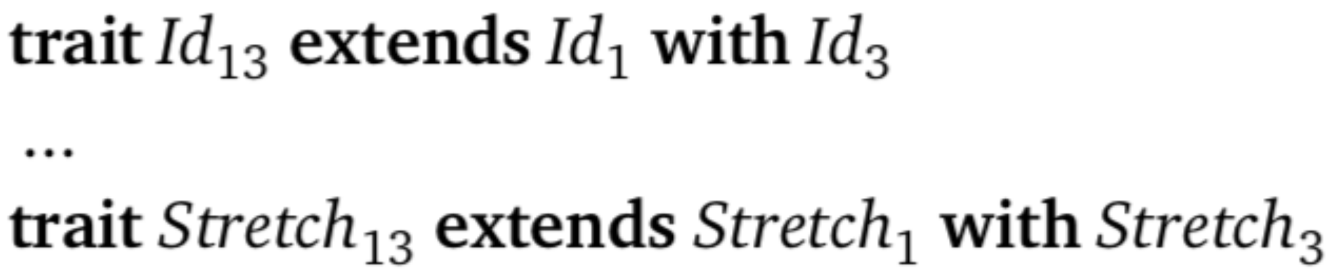
An alternative encoding of modular interpretations

- Allow non-linear extensions and loose dependencies
  - e.g. **wellSized**



trait $\mathit{Circuit}_3$ **extends** $\mathit{Circuit}_1$ {**def** *wellSized* : *Boolean*}
trait Id_3 **extends** $\mathit{Circuit}_3$ {**def** *wellSized* $=$ *true*}
...
trait $\mathit{Stretch}_3$ **extends** $\mathit{Circuit}_3$ {
  val *c* : $\mathit{Circuit}_3$; **val** *ns* : *List*[*Int*]
  def *wellSized* $= c.\mathit{wellSized} \land ns.\mathit{length} \equiv c.\mathit{width}$
}

- Require an extra step for combining **wellSized** and **width**



trait Id_{13} **extends** Id_1 **with** Id_3
...
trait $\mathit{Stretch}_{13}$ **extends** $\mathit{Stretch}_1$ **with** $\mathit{Stretch}_3$

Adding language constructs

- Extend SCANS with right stretches

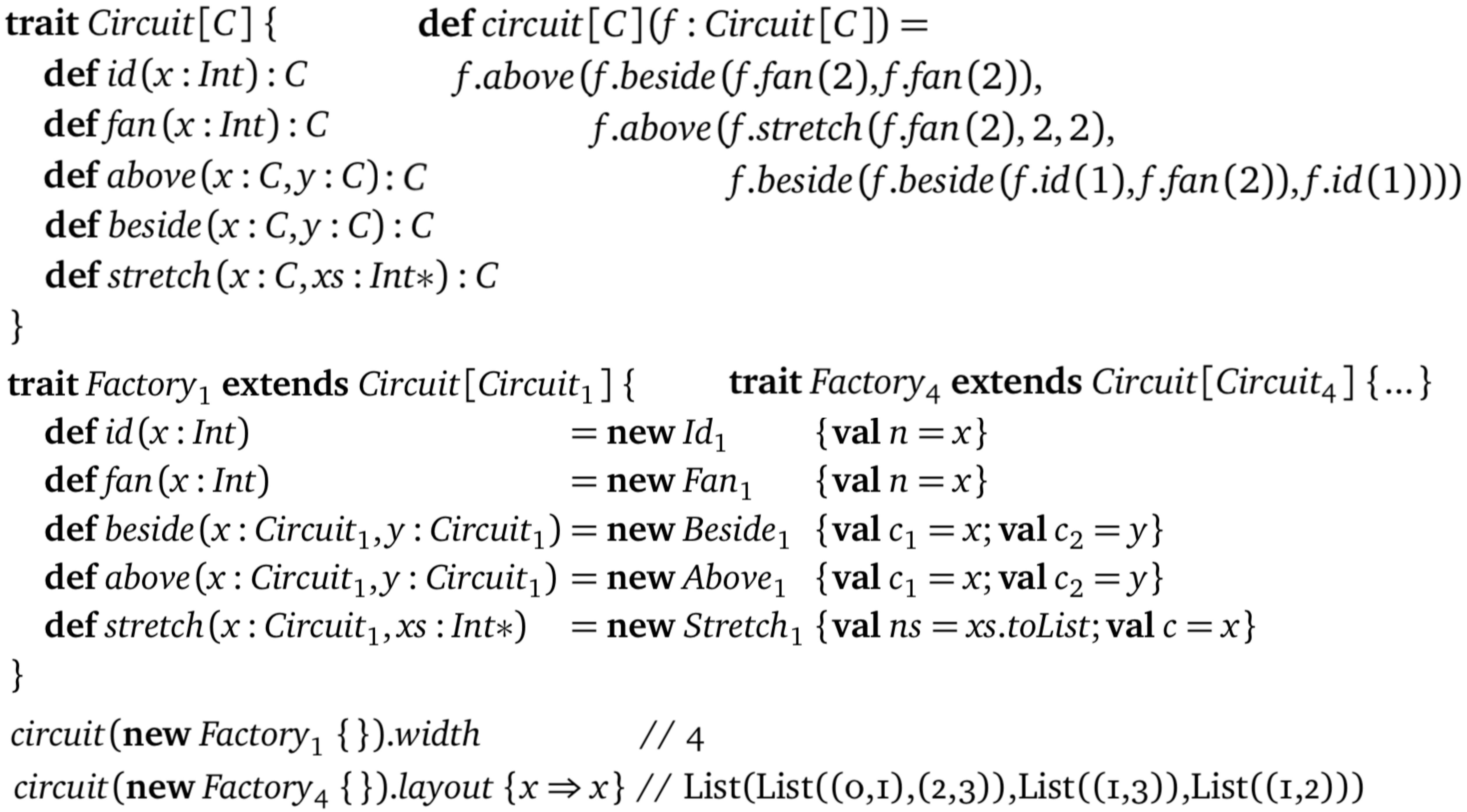
```
rstretch      :: [Int] → Circuit4 → Circuit4
rstretch ns c = stretch4 (1 : init ns) c ‘beside4‘ id4 (last ns − 1)
```

```
def rstretch(ns : List[Int], c : Circuit4) =
  stretch(1 :: ns.init, beside(c, id(ns.last − 1)))
```

```
trait RStretch extends Stretch4 {
  override def layout(f : Int ⇒ Int) = {
    val vs = ns.scanLeft(ns.last − 1)(_ + _).init
    c.layout(f.compose(vs(_)))}
}
```

Modular terms

- Object Algebras [Oliveira & Cook, 2012] come to the rescue



```
trait Circuit[C] {
  def id(x : Int) : C
  def fan(x : Int) : C
  def above(x : C, y : C) : C
  def beside(x : C, y : C) : C
  def stretch(x : C, xs : Int*) : C
}
```

```
def circuit[C](f : Circuit[C]) =
  f.above(f.beside(f.fan(2), f.fan(2)),
          f.above(f.stretch(f.fan(2), 2, 2),
                  f.beside(f.beside(f.id(1), f.fan(2)), f.id(1))))
```

```
trait Factory1 extends Circuit[Circuit1] {
  def id(x : Int)                            = new Id1      {val n = x}
  def fan(x : Int)                           = new Fan1     {val n = x}
  def beside(x : Circuit1, y : Circuit1)     = new Beside1  {val c1 = x; val c2 = y}
  def above(x : Circuit1, y : Circuit1)      = new Above1   {val c1 = x; val c2 = y}
  def stretch(x : Circuit1, xs : Int*)       = new Stretch1 {val ns = xs.toList; val c = x}
}
```

```
trait Factory4 extends Circuit[Circuit4] {...}
```

```
circuit(new Factory1 {}).width              // 4
circuit(new Factory4 {}).layout {x => x}    // List(List((0,1),(2,3)),List((1,3)),List((1,2)))
```

Modular terms, extended

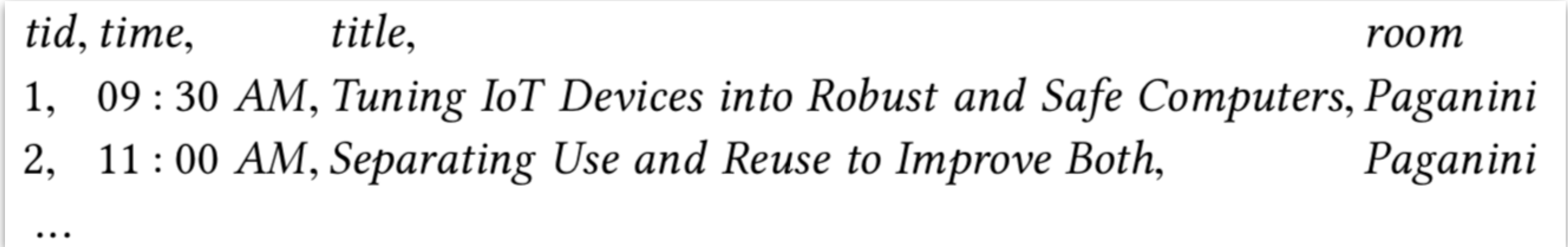
$$
\begin{array}{l}
\mathbf{trait}\ \mathit{ExtendedCircuit}[C]\ \mathbf{extends}\ \mathit{Circuit}[C]\ \{\\
\quad \mathbf{def}\ \mathit{rstretch}(x : C, \mathit{xs} : \mathit{Int}{*}) : C\\
\}
\end{array}
$$

$$
\begin{array}{l}
\mathbf{trait}\ \mathit{ExtendedFactory}_4\ \mathbf{extends}\ \mathit{ExtendedCircuit}[\mathit{Circuit}_4]\ \mathbf{with}\ \mathit{Factory}_4\ \{\\
\quad \mathbf{def}\ \mathit{rstretch}(x : \mathit{Circuit}_4, \mathit{xs} : \mathit{Int}{*}) = \mathbf{new}\ \mathit{RStretch}\ \{\mathbf{val}\ c = x; \mathbf{val}\ \mathit{ns} = \mathit{xs}.\mathit{toList}\}\\
\}
\end{array}
$$

$$
\mathbf{def}\ \mathit{circuit}_2[C](f : \mathit{ExtendedCircuit}[C]) = f.\mathit{rstretch}(\mathit{circuit}(f), 2, 2, 2, 2)
$$

Case study

- We refactored an external SQL query processor [Rompf & Amin, 2015] to make it more *modular*, *shallow*, and *embedded*



tid,	*time*,	*title*,	*room*
1,	09 : 30 *AM*,	*Tuning IoT Devices into Robust and Safe Computers*,	*Paganini*
2,	11 : 00 *AM*,	*Separating Use and Reuse to Improve Both*,	*Paganini*
…			

talks.csv

select ∗ **from** *talks.csv*

select *room*, *title* **from** *talks.csv*
where *time* = `'09:00 AM'`

select ∗
from(**select** *time*, *room*, *title* **as** $title_1$ **from** *talks.csv*)
join (**select** *time*, *room*, *title* **as** $title_2$ **from** *talks.csv*)
where $title_1 <> title_2$

def q_0 = *FROM* ("talks.csv")

def q_1 = q_0 *WHERE* *'time* === "09:00 AM"
SELECT (*'room*, *'title*)

def q_2 =
q_0 *SELECT* (*'time*, *'room*, *'title AS* $'title_1$) *JOIN*
(q_0 *SELECT* (*'time*, *'room*, *'title AS* $'title_2$)) *WHERE*
$'title_1 <> 'title_2$

A relational algebra interpreter

- Under the surface syntax, a relational algebra expression is constructed

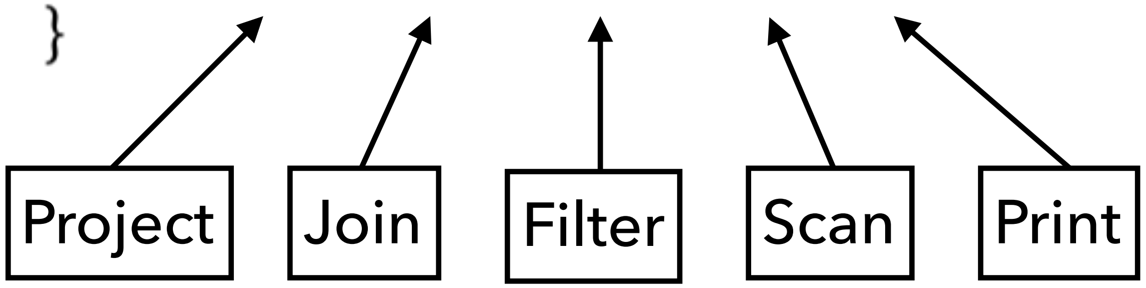
```
FROM("talks.csv")
WHERE 'time === "09:00 AM"
SELECT('room, 'title)
```

→

```
Project(Schema("room", "title"),
        Filter(Eq(Field("time"), Value("09:00 AM")),
               Scan("talks.csv")))
```

- Each relational algebra operator implements the following interface

```
trait Operator {
  def resultSchema : Schema
  def execOp(yld : Record ⇒ Unit) : Unit
}
```



Project Join Filter Scan Print

```
trait Join extends Operator {
  val op1, op2 : Operator
  def resultSchema =
    op1.resultSchema ++ op2.resultSchema
  def execOp(yld : Record ⇒ Unit) =
    op1.execOp {rec1 ⇒
      op2.execOp {rec2 ⇒
        val keys = rec1.schema intersect rec2.schema
        if(rec1(keys) ≡ rec2(keys))
          yld(Record(rec1.fields ++ rec2.fields,
            rec1.schema ++ rec2.schema))
    }}
}
```

From interpreter to compiler

- The interpreter is simple but slow
- Turning a slow interpreter into a fast compiler while keeping the simplicity – **staging** (LMS [Rompf & Odersky, 2010])
  - Actions on records are delayed to the generated code

$$\mathbf{def}\ execOp(yld : Record \Rightarrow Unit) : Unit$$

$$\Downarrow$$

$$\mathbf{def}\ execOp(yld : Record \Rightarrow \boxed{Rep[Unit]}) : \boxed{Rep[Unit]}$$

- Two backends are supported (Scala and C), modularly

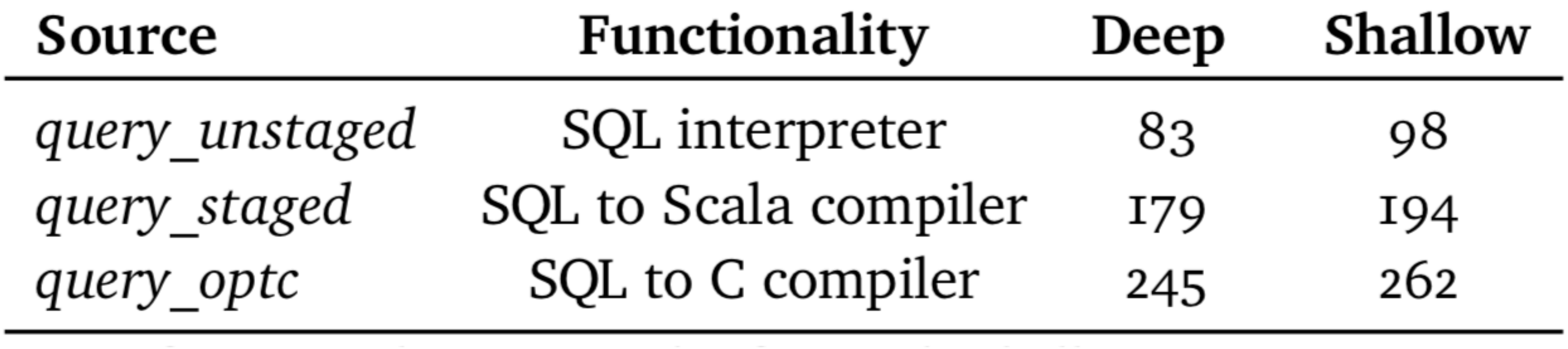
Syntax extensions

- Add aggregations (**group by**) and hash joins

```
trait Group extends Operator {
  val keys, agg : Schema; val op : Operator
  def resultSchema = keys ⧺ agg
  def execOp(yld : Record ⇒ Unit) {...}
}
trait HashJoin extends Join {
  override def execOp(yld : Record ⇒ Unit) = {
    val keys = op1.resultSchema intersect op2.resultSchema
    val hm = new HashMapBuffer(keys, op1.resultSchema)
    op1.execOp {rec1 ⇒
      hm(rec1(keys)) += rec1.fields}
    op2.execOp {rec2 ⇒
      hm(rec2(keys)) foreach {rec1 ⇒
        yld(Record(rec1.fields ⧺ rec2.fields, rec1.schema ⧺ rec2.schema))}}}
}
```

Evaluation

- The **same** code is generated, thus performance is similar
- The modularity comes with a few more lines of code



Source	Functionality	Deep	Shallow
query_unstaged	SQL interpreter	83	98
query_staged	SQL to Scala compiler	179	194
query_optc	SQL to C compiler	245	262

More in the paper

Shallow EDSLs and Object-Oriented Programming

Beyond Simple Compositionality

Weixin Zhang[a] and Bruno C. d. S. Oliveira[a]

a The University of Hong Kong, Hong Kong, China

Abstract

Context. Embedded Domain-Specific Languages (EDSLs) are a common and widely used approach to DSLs in various languages, including Haskell and Scala. There are two main implementation techniques for EDSLs: *shallow embeddings* and *deep embeddings*.

Inquiry. Shallow embeddings are quite simple, but they have been criticized in the past for being quite limited in terms of modularity and reuse. In particular, it is often argued that supporting multiple DSL interpretations in shallow embeddings is difficult.

Approach. This paper argues that shallow EDSLs and Object-Oriented Programming (OOP) are closely related. Gibbons and Wu already discussed the relationship between shallow EDSLs and procedural abstraction, while Cook discussed the connection between procedural abstraction and OOP. We make the transitive step in this paper by connecting shallow EDSLs directly to OOP via procedural abstraction. The knowledge about this relationship enables us to improve on implementation techniques for EDSLs.

Knowledge. This paper argues that common OOP mechanisms (including *inheritance*, *subtyping*, and *type-refinement*) increase the modularity and reuse of shallow EDSLs when compared to classical procedural abstraction by enabling a simple way to express *multiple, possibly dependent, interpretations*.

Grounding. We make our arguments by using Gibbons and Wu's examples, where procedural abstraction is used in Haskell to model a simple shallow EDSL. We recode that EDSL in Scala and with an improved OO-inspired Haskell encoding. We further illustrate our approach with a case study on refactoring a deep external SQL DSL implementation to make it more modular, shallow, and embedded.

Importance. This work is important for two reasons. Firstly, from an intellectual point of view, this work establishes the connection between shallow embeddings and OOP, which enables a better understanding of both concepts. Secondly, this work illustrates programming techniques that can be used to improve the modularity and reuse of shallow EDSLs.

ACM CCS 2012

- **Software and its engineering → Language features; Domain specific languages;**

Keywords embedded domain-specific languages, shallow embedding, object-oriented programming

The Art, Science, and Engineering of Programming

Perspective The Art of Programming

Area of Submission Domain-Specific Languages, Modularity and Separation of Concerns

- An OOP inspired Haskell encoding of modular (dependent) interpretations

```
class Circuit c where
  id      :: Int → c
  fan     :: Int → c
  above   :: c → c → c
  beside  :: c → c → c
  stretch :: [Int] → c → c

class a ≺ b where
  prj :: a → b
instance a ≺ a where
  prj x = x
instance (a, b) ≺ a where
  prj = fst
instance (b ≺ c) ⇒ (a, b) ≺ c where
  prj = prj ∘ snd
```

Conclusion

Thank you!

- OOP and shallow embeddings are closely related
  - The essence of both is *procedural abstraction*
- OOP abstractions bring extra modularity to shallow embeddings
  - Subtyping, inheritance and type-refinement
- Combine extensible interpreters with Object Algebras for greater good
  - Modular multiple (possibly dependent) interpretations and terms
- Shallow embeddings can be performant with *staging*
- The motivation to employ deep embeddings becomes weaker
  - Mostly reduced to the need for AST transformations

https://github.com/wxzh/shallow-dsl