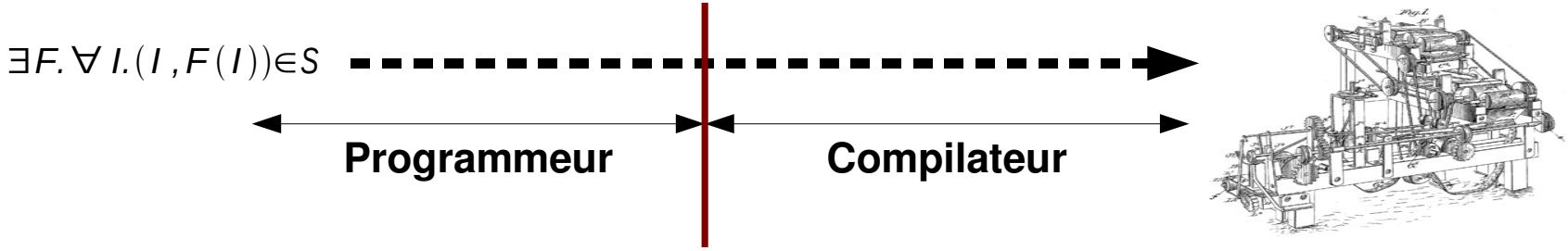
Des performances dans les nuages avec la virtualisation des langages

Yann Régis-Gianas
yrg@pps.jussieu.fr
Université Paris-Diderot – PPS – INRIA πr^2
EPITA – Séminaire Performance et Généricité du LRDE

15 février 2012

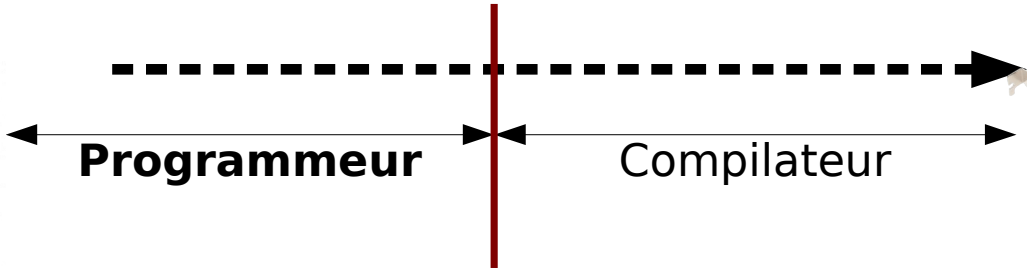
Spécification du problème **Résolution effective par la machine**



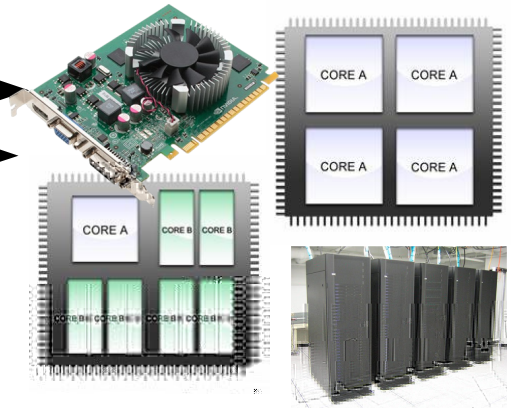
Langage de programmation

Spécification du problème

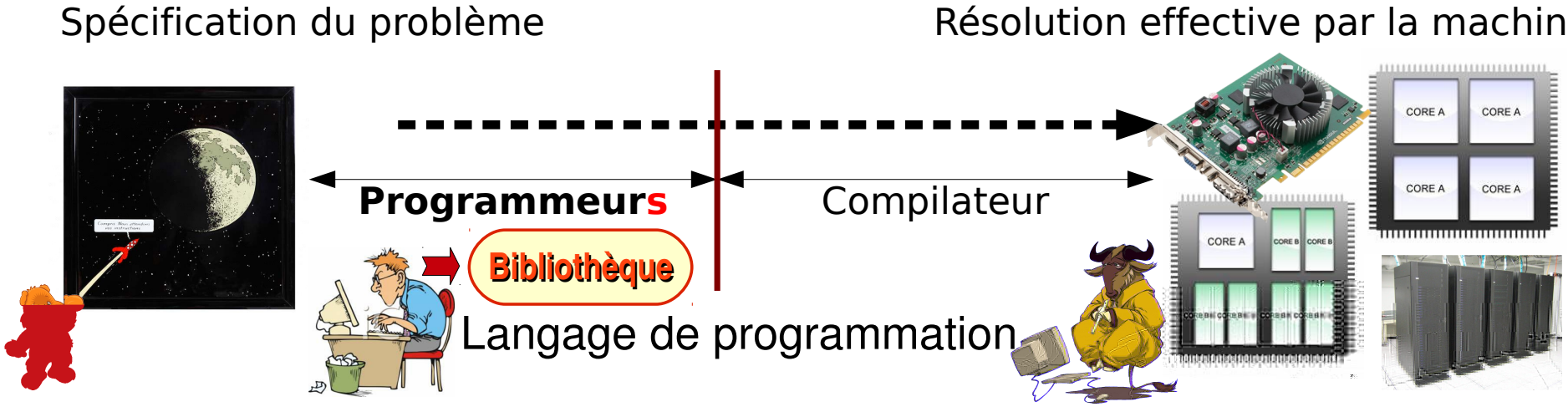
Résolution effective par la machine



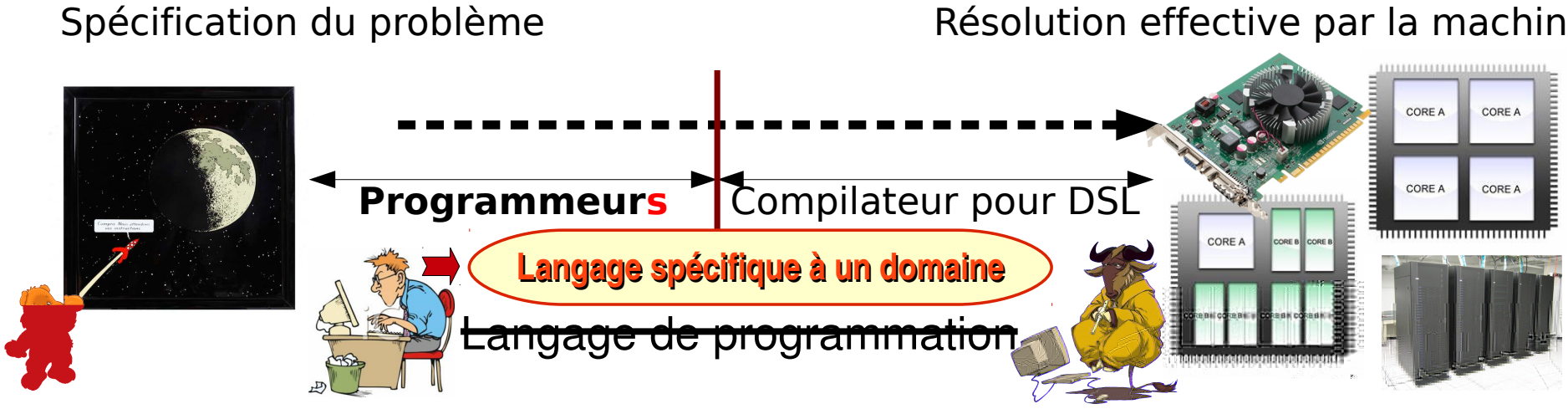
Langage de programmation



- En pratique, les problèmes sont **complexes.**
- Les architectures de calculs sont de plus en plus **hétérogènes.**

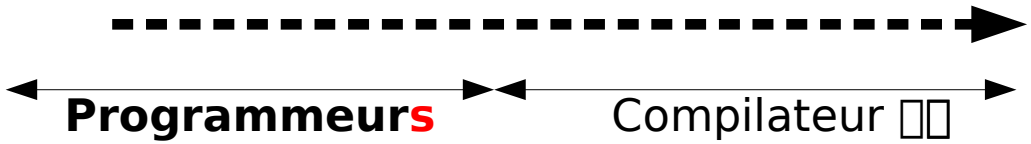


- Les problèmes sont **décomposés**.
- Les solutions se construisent à l'aide d'**abstractions**.
- Des experts fournissent des **bibliothèques** qui les réalisent.



- Définir un **langage dédié à ces abstractions** permet :
  - d'exprimer des idiomes du domaine ;
  - de rendre possible des optimisations spécifiques ;
  - de réimplémenter les passes de compilation habituelles.

Spécification du problème Résolution effective par la machine



- La **virtualisation de langage** consiste à :
  A. **embarquer** un langage spécifique dans un langage généraliste ;
  B. **réutiliser** et **étendre** les passes de son compilateur.

Ce qu'est cet exposé

Une explication des techniques utilisées en Scala
pour rendre possible les points A et B précédents
dans le *framework* Delite
qui permet l'écriture de DSLs virtualisés en Scala.

Ce que n’est pas cet exposé

DISCLAIMER

Je n’ai pas contribué à ces travaux.

Carnet de vol

Le passage à l'échelle avec SCALA

L'embarquement polymorphe d'un langage dans un autre

La modularisation d'un compilateur

Le framework DELITE

Conclusion

Apprenons à programmer en SCALA !

Scala : un langage suisse

`http://www.scala-lang.org`

- Scala est un langage généraliste développé à l'École Polytechnique de Lausanne par Martin Odersky et son équipe.
- Il est compilé vers la JVM (il existe aussi une version expérimentale vers .Net).
- Sa distribution inclut un compilateur, un serveur de compilation, une boucle interactive, un système d'automatisation de la compilation et de gestion de paquets, une bibliothèque standard riche et extensible.

Scala : un langage d'ordre supérieur

- En Scala, toutes les valeurs sont des objets.
- Toutes les constructions du langage se traduisent en appels de méthode.
- En particulier, les fonctions sont des objets :

```
trait Function1 [-T, +R] {
  def apply (x : T) : R
}
```

- Un appel de fonction "`f (a)`" est un sucre syntaxique pour "`f.apply (a)`"
- Le langage fournit des primitives de programmation par continuations (délimitées) ainsi qu'une bibliothèque d'acteurs (des co-routines).

Un compteur d'opérations en SCALA

```
object CountOperations {
  import scala.collection.mutable.HashMap
  val counter = HashMap.empty[String, Int]
  def reset (label : String) { counter (label) = 0 }
  def increment (label : String) { counter (label) += 1 }
  def bench [A] (f : => A) = {
    for ((label, _) ← counter) reset (label)
    f
    counter.toList
  }
}
```

- Le mot-clé `object` définit un objet singleton.
- La construction "`for (x ← e) a`" est un sucre syntaxique pour : "`e.foreach (x => a)`", qui s'écrit aussi "`e foreach (x => a)`".

  (Le point est optionnel.)

SCALA : un langage à types riches

- Les types des objets et des méthodes sont très précis et généraux.
- Plus précisément, SCALA propose :
  - (i) une relation de sous-typage nominal et structurel ;
  - (ii) du polymorphisme borné et d'ordre supérieur ;
  - (iii) une forme restreinte de types dépendants de valeur.
- SCALA est muni d'un algorithme d'inférence locale de type.
- SCALA permet de définir des arguments implicites inférés par le contexte de typage.

Une relation de sous-typage nominal et structurel

```
class a
class b extends a { def foo = 1 }
class c { def foo = 1 }
class d extends a

object test {
   def f (x : a) = 0
   def g (x : Any { def foo : Int }) = x.foo
   def h (x : a { def foo : Int }) = x.foo
}
```

Du polymorphisme borné et d'ordre supérieur

```
case class ListNode[+T](h: T, t: ListNode[T]) {
  def head: T = h
  def tail: ListNode[T] = t
  def prepend[U >: T](elem: U): ListNode[U] = ListNode(elem, this)
}

object LowerBoundTest {
  val empty: ListNode[Null] = ListNode(null, null)
  val strList: ListNode[String] = empty.prepend("hello")
                                       .prepend("world")
  val anyList: ListNode[Any] = strList.prepend(12345)
}

class Domains {
  type Point
  type Vec
  type Domain[_] <: Traversable[_]
  type Site = Domain[Vec]
  type Box = Domain[Point]
}
```

Types virtuels et *self type*

```
abstract class SubjectObserver {

  type O <: Observer

  type S <: Subject

  abstract class Subject {
    self:S =>

    var observers : List[O] = Nil

    def register (new_observers : O*) = observers ++= new_observers

    def notifyObservers = observers.foreach (_.onNotify (this))
  }

  abstract class Observer { def onNotify (s : S) }
}
```

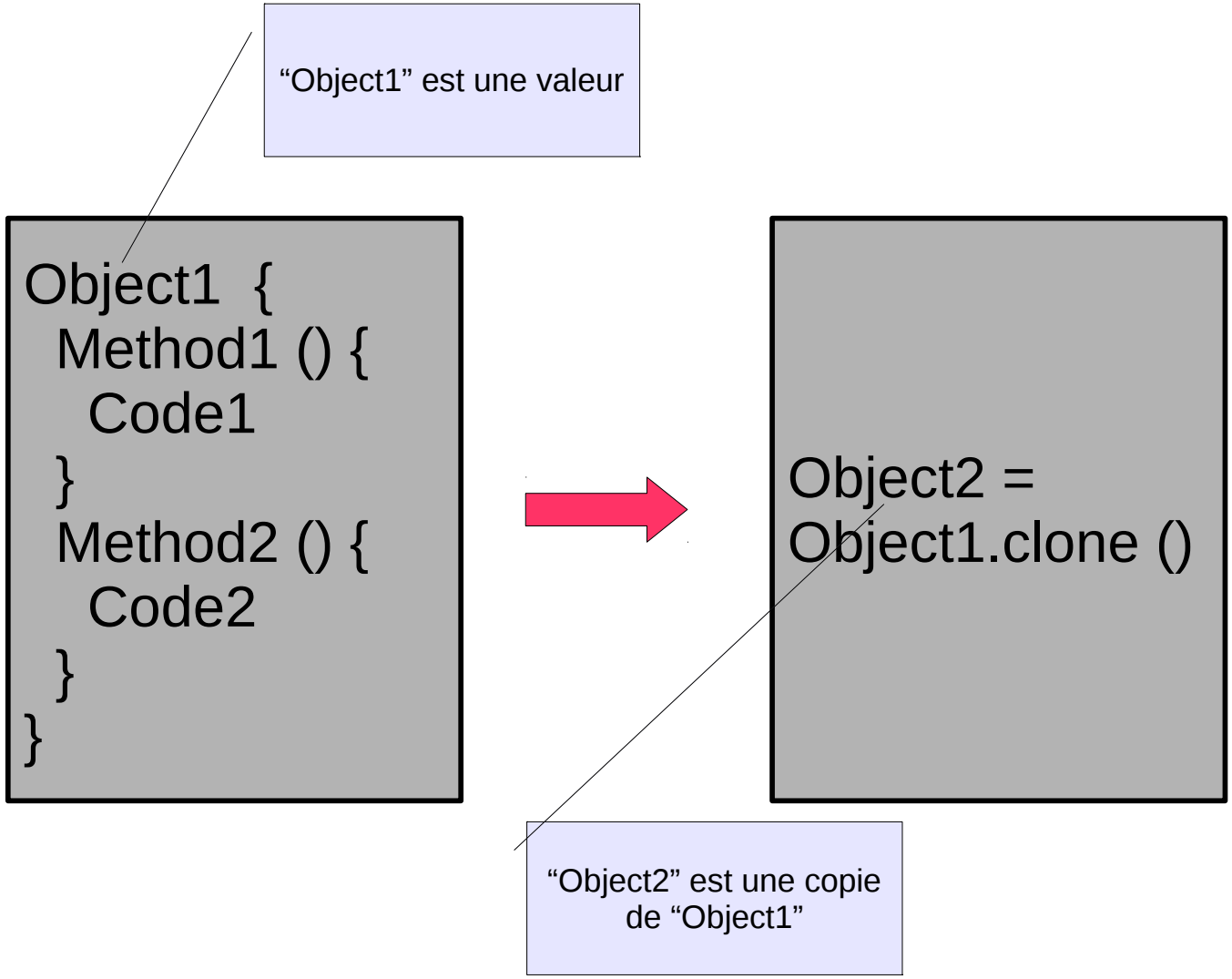
Une forme restreinte de types dépendants

```
class Graph {
  class Node {
    var connectedNodes: List[Node] = Nil
    def connectTo(node: Node) {
      if (connectedNodes.find(node.equals).isEmpty) {
        connectedNodes = node :: connectedNodes
      }
    }
  }
  var nodes: List[Node] = Nil
  def newNode: Node = {
    val res = new Node
    nodes = res :: nodes
    res
  }
}

val g = new Graph
val n1 = g.newNode
val n2 = g.newNode
n1.connectTo(n2)
val h: Graph = new Graph
val n3: h.Node = h.newNode
n1.connectTo(n3) // Rejeté!
```

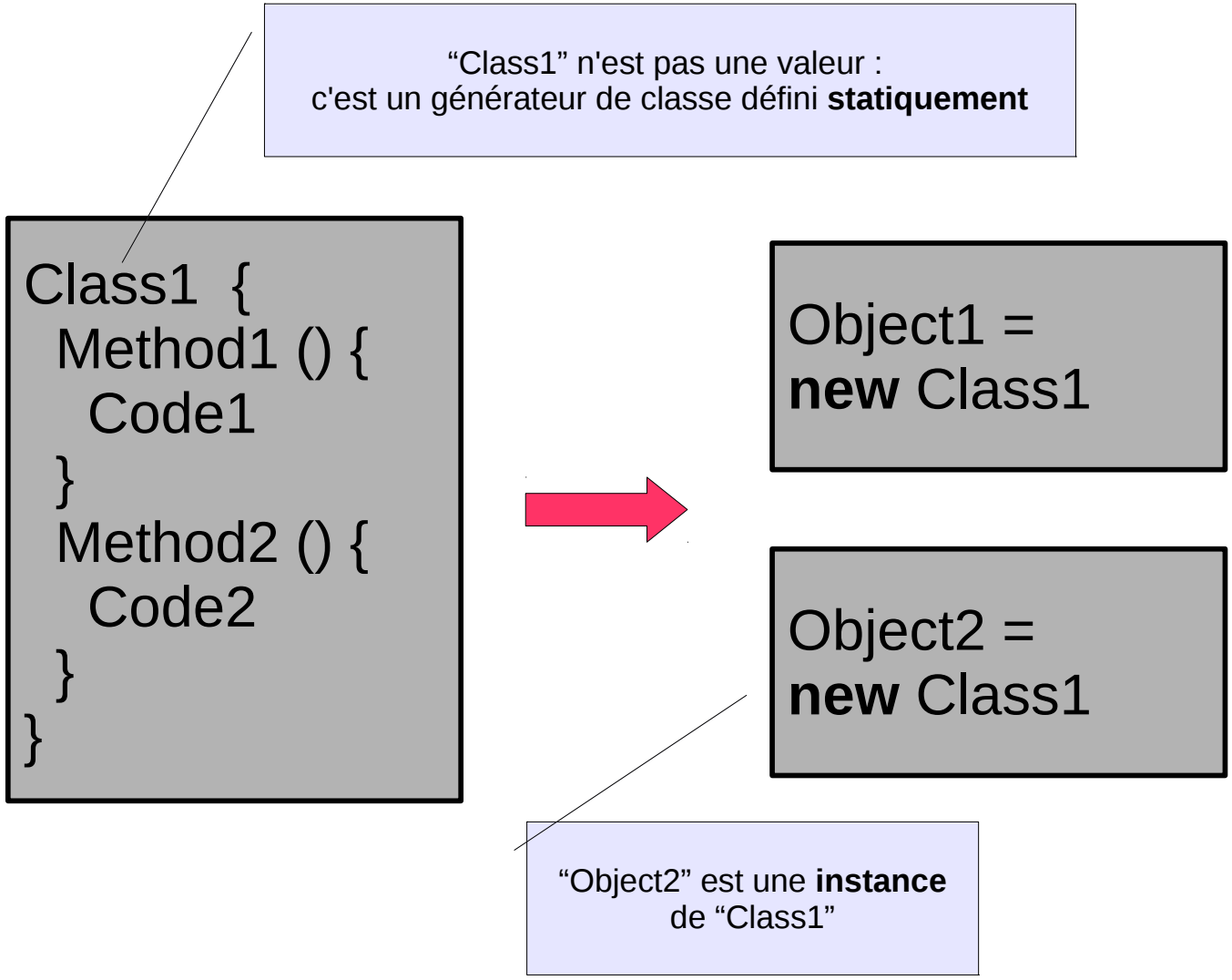
Des arguments implicites inférés à l'aide des types

```
 abstract class SemiGroup[A] {
   def add(x: A, y: A): A
 }

abstract class Monoid[A] extends SemiGroup[A] {
   def unit: A
}

object ImplicitTest extends Application {
  implicit object StringMonoid extends Monoid[String] {
    def add(x: String, y: String): String = x concat y
    def unit: String = ""
  }
  implicit object IntMonoid extends Monoid[Int] {
    def add(x: Int, y: Int): Int = x + y
    def unit: Int = 0
  }
  def sum[A](xs: List[A])(implicit m: Monoid[A]): A =
    if (xs.isEmpty) m.unit
    else m.add(xs.head, sum(xs.tail))

    println (sum (List(1, 2, 3)))
    println (sum (List("a", "b", "c")))
}
```

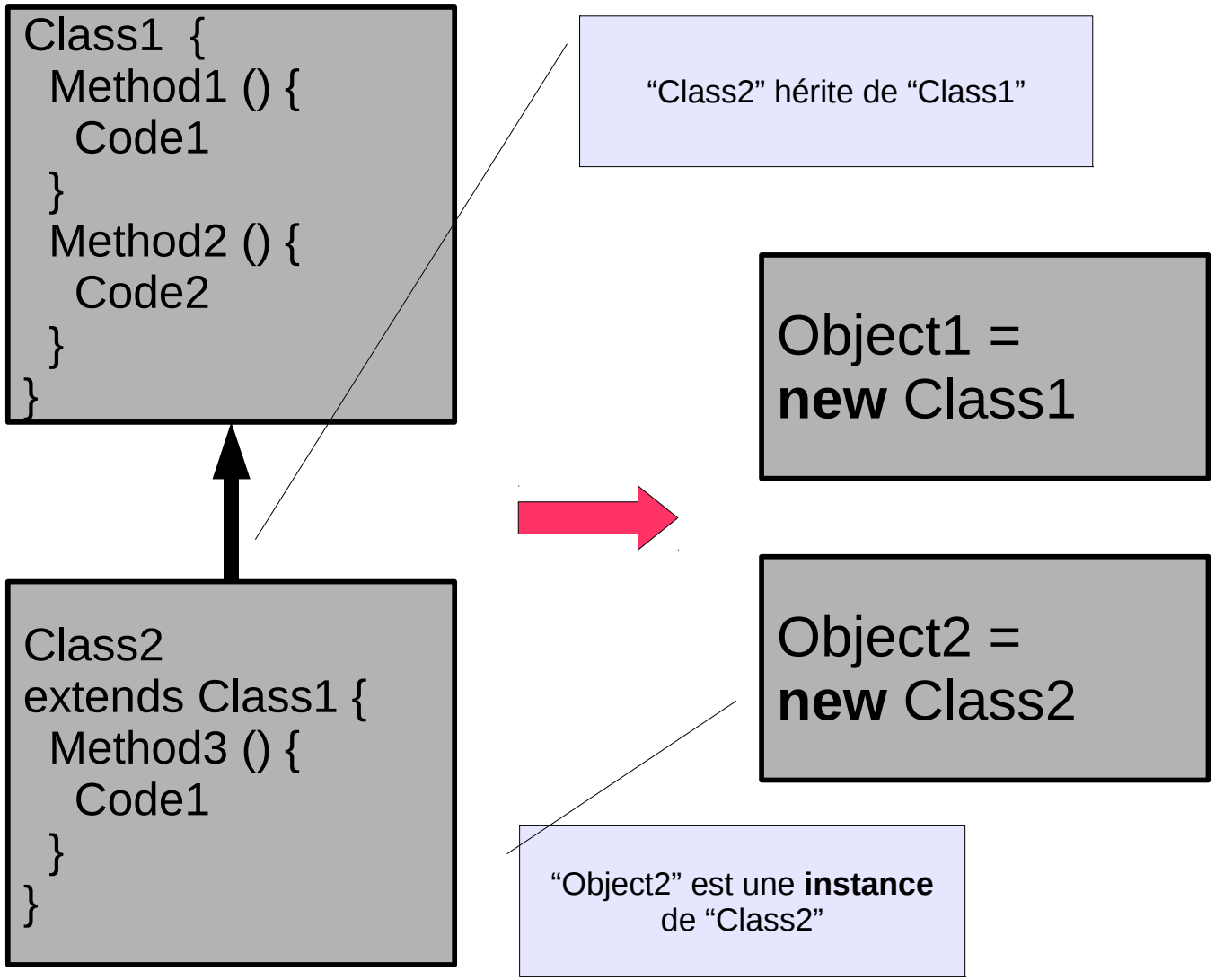
SCALA : un langage pour la modularité



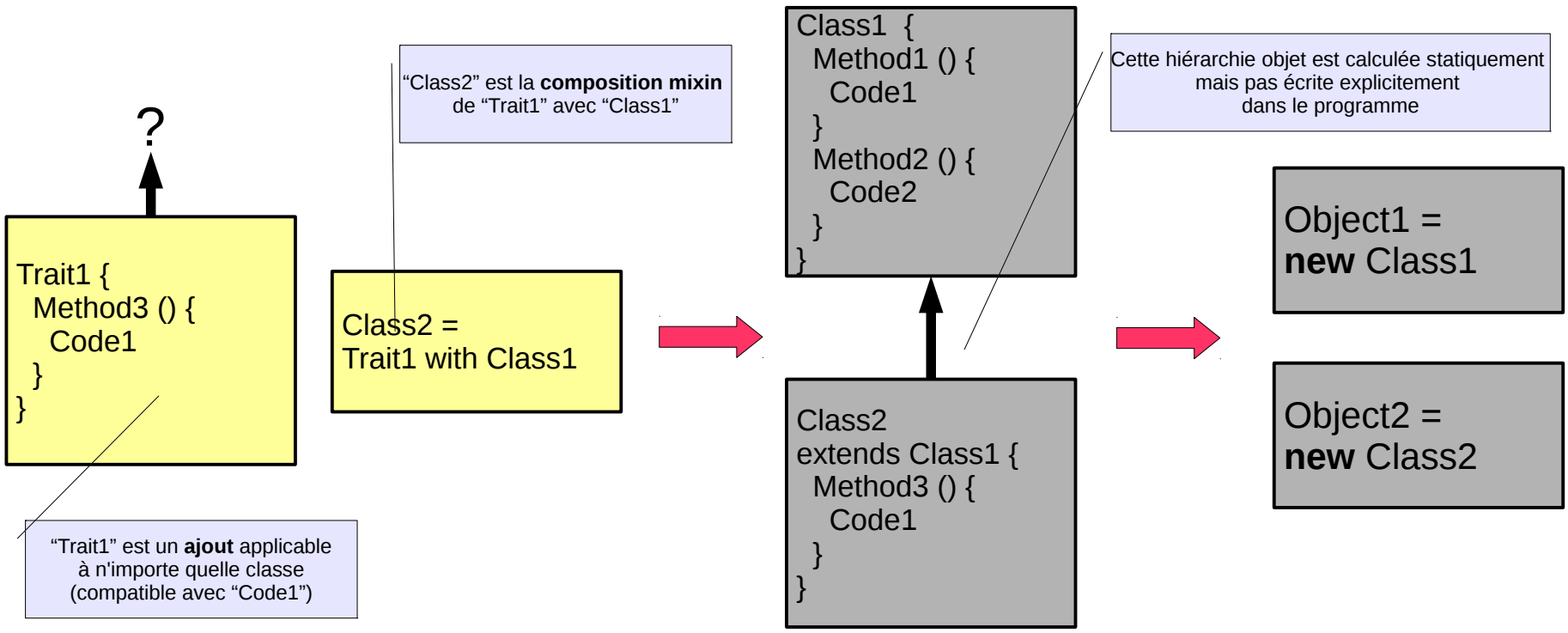
SCALA : un langage pour la modularité



SCALA : un langage pour la modularité



SCALA : un langage pour la modularité



Construisons une bibliothèque de multiplications de matrices en SCALA !

Multiplication de matrices

Definition

Soit une matrice A de taille $p \times q$ et une matrice B de taille $q \times r$, la matrice C produit de A et B, $C = AB$ est définie par :

$$C(i,j) = \sum_{k=1}^{q} A(i,k).B(k,j)$$

Un client pour notre bibliothèque

```
trait Client { this : MatrixLib =>

  val a = matrix (Seq (1, 2, 3, 4),
                  Seq (5, 6, 7, 8))

  val b = matrix (Seq (7, 8),
                  Seq (9, 0),
                  Seq (1, 2),
                  Seq (3, 4))

  val c = matrix (Seq (7, 8),
                  Seq (0, 1))

  def d = a * (b * c)
}
```

Réaction de l'auteur de la bibliothèque

Les raisons du drame

“$a * (b * c)$” nécessite 32 multiplications.

“$(a * b) * c$” nécessite 24 multiplications.

Carnet de vol

Le passage à l'échelle avec SCALA

L'embarquement polymorphe d'un langage dans un autre

La modularisation d'un compilateur

Le framework DELITE

Conclusion

Prenons cette optimisation en charge !

Du calcul à la **représentation du calcul**

Changeons la promesse faite à notre utilisateur :

La bibliothèque `MatrixLib`
ne produit pas des matrices
mais des calculs s'évaluant en des matrices.

Changement dans l'interface de la bibliothèque

```
trait Base {
  type Rep[T]
  val eval[T] (r : Rep[T]) : T
}

trait MatrixLib extends Base {
  def matrix (rows : Seq[Double]*) : Rep[Matrix]
  def nrows (m : Rep[Matrix]) : Int
  def ncols (m : Rep[Matrix]) : Int
  def infix_* (a : Rep[Matrix], b : Rep[Matrix]) : Rep[Matrix]
}
```

Changement pour le client

```
trait Client { this : MatrixLib =>
  val a = ...
  val b = ...
  val c = ...
  def d = eval (a * (b * c))
}
```

Opportunités d'optimisation

Notre bibliothèque est aisément adaptable à cette interface :

```
trait ConcreteMatrixLib extends MatrixLib {
  type Rep[+T] = T
  def nrows (m : Matrix) = m.nrows
  def ncols (m : Matrix) = m.ncols
  def matrix (rows : Seq[Double]*) : Matrix = new Matrix (rows)
  def infix_* (a : Matrix, b : Matrix) : Matrix = a * b
  def eval[T] (x : T) = x
}
```

. . . mais évidemment on peut aller bien plus loin !


OPPORTUNITY
Did you know that Chinese uses the same character for 'Problem' and 'Experience Points'?

Représentation des calculs par des ASTs

```
trait Expressions {
  abstract class Exp[T]
  def eval[T] (e : Exp[T]) : T
}

trait BaseExp extends Base with Expressions {
  type Rep[T] = Exp[T]
}
```

Représentation des calculs par des ASTs

```
trait SymbolicMatrixLib extends MatrixLib with BaseExp {
  case class Const (x : Matrix) extends Exp[Matrix]
  case class Times (lhs : Exp[Matrix], rhs : Exp[Matrix]) extends Exp[Matrix]

  def matrix (rows : Seq [Double]*) : Exp[Matrix] = Const (new Matrix (rows))

  def infix_* (a : Exp[Matrix], b : Exp[Matrix]) = Times (a, b)

  def dim (e : Exp[Matrix]) : (Int, Int) =
    e match {
      case Const (m) => (m.nrows, m.ncols)
      case Times (a, b) => ((dim (a))._1, (dim (b))._2)
    }

  def nrows (m : Exp[Matrix]) = (dim (m))._1
  def ncols (m : Exp[Matrix]) = (dim (m))._2

  override def eval[T] (e : Exp[T]) : T = e match {
    case Const (m) => m
    case Times (a, b) => eval (a) * eval (b)
  }
}
```

Retour sur la multiplication de matrices

Definition

Soit une matrice A de taille $p \times q$ et une matrice B de taille $q \times r$, la matrice C produit de A et B, $C = AB$ est la matrice de taille $p \times r$ définie par :

$$C(i,j) = \sum_{k=1}^{q} A(i,k).B(k,j)$$

- La complexité de cette opération est $\Theta(pqr)$.
- La multiplication est associative : A (B C) = (A B) C

... mais le nombre d'opérations n'est pas le même !

Un bon parenthèsage, ça compte énormément

Soit A, une matrice de taille $p \times q$, B de taille $q \times r$ et C de taille $r \times s$ alors :

- pour calculer $(AB)C$, il faut $pqr + prs$ opérations ;
- pour calculer $A(BC)$, il faut $qrs + pqs$ opérations.

Par exemple, si $p = 5$, $q = 4$, $r = 6$ et $s = 2$ alors :

- $pqr + prs = 180$,
- $qrs + pqs = 88$.

Mémoization d'une fonction récursive

```
class Memoize1[-T, +R] (f: T => R) extends (T => R) {
  import scala.collection.immutable
  private[this] var vals = Map.empty[T, R]
  def apply(x: T): R = {
    if (vals.contains(x)) vals(x) else {
      val y = f(x); vals = vals + ((x, y)); y }
  }
}

object Memoize1 {
  def apply[T, R](f: T => R) = new Memoize1(f)
  def Y[T, R](f: (T, T => R) => R) = {
    var yf: T => R = null
    yf = Memoize1(f(_, yf(_)))
    yf
  }

  def tabulate [T, R] (f: (T, T => R) => R, seq: Seq[T]) : T => R = {
    val ft = Memoize1.Y (f)
    seq foreach ft
    ft
  }
}
```

Parenthésage optimal en SCALA

```
trait SymbolicMatrixLibOpt extends SymbolicMatrixLib {
  import Memoize1._
  def parenthesis (matrixes : Array[Matrix]) : Exp[Matrix] = {
    val n = matrixes.size - 1
    def p (k : Int) = if (k < n) (matrixes (k + 1)).nrows else (matrixes (k)).ncols
    def min (p : (Int, Int), q: (Int, Int)) = if (p._2 < q._2) p else q

    def m (s : (Int, Int), mr : ((Int, Int)) => (Int, Int)) : (Int, Int) = {
      val i = s._1; val j = s._2
      def cost (k : Int) = mr (i, k)._2 + mr (k + 1, j)._2 + p (i - 1) * p (k) * p (j)
      if (i == j) (i, 0) else (for { k ← i to j - 1 } yield (k, cost (k))) reduceLeft min
    }

    val mt = tabulate (m, for (l ← 1 to n; i ← 1 to n - l + 1) yield (i, i + l - 1))

    def best (i : Int, j : Int) : Exp[Matrix] =
      if (i == j) Const (matrixes (i)) else Times (best (i, mt (i, j)._1), best (mt (i, j)._1 + 1, j))
    best (0, n)
  }

  def collect (m : Exp[Matrix]) : Array[Matrix] =
    m match {
      case Const (m) => Array (m)
      case Times (a, b) => collect (a) ++ collect (b)
    }

  def optimize (m : Exp[Matrix]) : Exp[Matrix] =
    parenthesis (collect (m))

  override def eval[T] (e : Exp[T]) : T =
    super.eval (optimize (e))

}
```

Checkpoint

- Nous venons d'embarquer un langage de multiplication de matrices dans Scala en utilisant le polymorphisme d'ordre supérieur, c'est-à-dire en paramétrant les calculs vis-à-vis d'un type `Rep[T]` des calculs s'évaluant en des valeurs de type `T`.

- La **virtualisation de langage** consiste à :
  A. ~~**embarquer** un langage spécifique dans un langage généraliste~~ ;
  B. **réutiliser** et **étendre** les passes de son compilateur.

Carnet de vol

Comment étendre un langage et ses transformations ?

Le problème des expressions

"The Expression Problem is a new name for an old problem. The goal is ***to define a datatype by cases****, where one can* ***add new cases to the datatype*** *and* ***new functions over the datatype****, without recompiling existing code, and while retaining static type safety (e.g., no casts). For the concrete example, we take expressions as the data type, begin with one case (constants) and one function (evaluators), then add one more construct (plus) and one more function (conversion to a string)."*

– The Expression Problem, Philip Wadler, 12 November 1998

[. . .] "whether a language can solve the Expression Problem is a salient indicator of its capacity for expression."

– The Expression Problem, Philip Wadler, 12 November 1998

Le problème d'un côté . . .

Dans un langage avec des types algébriques et du pattern-matching [1] :

```
module Exp = struct
  type t = Const of int
  let eval = function Const x → x
end

module ExpWithPlus = struct
  type t = Exp of Exp.t | Plus of t × t
  let rec eval = function
    | Plus (lhs, rhs) → eval lhs + eval rhs
    | Exp e → Exp.eval e
end
```

```
module ExpWithShow = struct
  let show = function
    | Const x → string_of_int x
end

module ExpWithShowWithPlus =
struct
  type t = ExpWithPlus.t
  let eval = ExpWithPlus.eval
  let rec show = function
    | ExpWithPlus.Exp t →
      ExpWithShow.show t
    | ExpWithPlus.Plus (a, b) →
      show a ^ "+" show b
end
```

1. Il existe de jolies solutions en O'Caml *via* variants polymorphes par exemple.

Le problème d'un autre côté ...

Dans un langage orienté objet traditionnel :

```
abstract class Exp {
  def eval : Int
}
class Const (n : Int) extends Exp {
  def eval = n
}
class Plus (lhs : Exp, rhs : Exp) extends Exp {
  def eval = lhs.eval + rhs.eval
}
class ConstWithShow (n : Int) extends Const (n : Int) {
  def show = n.toString
}
class PlusWithShow (lhs : Exp, rhs : Exp) extends Exp {
  def show = lhs.show + "+" + rhs.show
}
```

Ce programme est rejeté !

Le *design pattern* `Visitor`

```
object Base {
  abstract class Visitor {
    def visitConst (v : Const) : Unit
  }
  abstract class Exp {
    def accept (e : Visitor) : Unit
  }
  class Const (val x : Int) extends Exp {
    override def accept (v : Visitor) = v.visitConst (this)
  }
  class EvalVisitor extends Visitor {
    var value : Int = 0
    override def visitConst (v: Const) { value = v.x }
  }
}
```

Le *design pattern* `Visitor` a aussi une limite

```
object WithPlus {
  abstract class VisitorWithPlus extends Visitor {
    def visitPlus (v: Plus) : Unit
  }
  class Plus (lhs : Exp, rhs : Exp) extends Exp {
    override def accept (v: VisitorWithPlus) = v.visitPlus (this)
  }
// error: class Plus needs to be abstract,
// since method accept in class Exp of type
// (e: Base.Visitor)Unit is not defined

// error: method accept overrides nothing
// override def accept (v: VisitorWithPlus) = v.visitPlus (this)

// class VisitorEvalWithPlus extends VisitorEval
// or VisitorWithPlus ?
}
```

Solution par pattern `matching`

```
trait Base {
  abstract class Exp
  case class Const (x: Int) extends Exp
  def eval (e : Exp) : Int = e match {
    case Const (x) => x
  }
}
trait BasePlus extends Base {
  case class Plus (lhs : Exp, rhs : Exp) extends Exp
  override def eval (e : Exp) : Int = e match {
    case Plus (l, r) => eval (l) + eval (r)
    case _ => super.eval (e)
  }
}
trait Show extends Base {
  def show (e : Exp) : String = e match {
    case Const (x) => x.toString
  }
}
trait ShowPlus extends Show with BasePlus {
  override def show (e : Exp) = e match {
    case Plus (l, r) => show (l) + "+" + show (r)
    case _ => super.show (e)
  }
}
```

Solution par pattern `matching`

Un inconvénient de la solution précédente :

> Il ne faut pas oublier de redéfinir `show` dans `ShowPlus`.

Solution par type virtuel orientée donnée

```
trait Base {
  type VExp <: Exp
  trait Exp { def eval : Int }
  class Num (v: Int) extends Exp { def eval = v }
}
trait BasePlus extends Base {
  class Plus (l : VExp, r : VExp) extends Exp {
    def eval = l.eval + r.eval
  }
}
trait Show extends Base {
  type VExp <: Exp
  trait Exp extends super.Exp { def show : String }
  class Num (v: Int) extends super.Num (v) with Exp {
    def show = v.toString
  }
}
trait ShowPlus extends BasePlus with Show {
  class Plus (l: VExp, r: VExp) extends super.Plus (l, r) with Exp {
    def show = l.show + r.show
  }
}
```

Solution par type virtuel orientée opération (1/2)

```
trait Base {
  type VVisitor <: Visitor
  trait Visitor { def visitConst (v : Const) : Unit }
  abstract class Exp { def accept (e : VVisitor) : Unit }
  class Const (val x : Int) extends Exp {
    override def accept (v : VVisitor) = v.visitConst (this)
  }
  class EvalVisitor extends Visitor { this: VVisitor =>
    var value : Int = 0
    def apply (t: Exp) = { t.accept (this); value }
    override def visitConst (v: Const) { value = v.x }
  }
}
```

Solution par type virtuel orientée opération (2/2)

```
trait Show extends Base {
  class ShowVisitor extends super.Visitor { this: VVisitor =>
    var value: String = _
    def apply (t: Exp) = { t.accept (this); value }
    override def visitConst (v: Const) { value = v.x.toString }
  }
}

trait PlusBase extends Base {
  type VVisitor <: Visitor
  trait Visitor extends super.Visitor { def visitPlus (v: Plus) : Unit }
  class Plus (val lhs : Exp, val rhs : Exp) extends Exp {
    override def accept (v: VVisitor) = v.visitPlus (this)
  }
 class EvalVisitor extends super.EvalVisitor with Visitor { this: VVisitor =>
   override def visitPlus (v: Plus) { value = apply (v.lhs) + apply (v.rhs) }
 }
}

trait ShowPlus extends Show with PlusBase {
  class ShowVisitor extends super.ShowVisitor { this: VVisitor =>
    def visitPlus (v: Plus) { value = apply (v.lhs) + apply (v.rhs) }
  }
}
```

Utilisons ces techniques pour étendre notre DSL virtualisé !

Multiplication de matrices par un scalaire

```
trait MatrixLib2 extends MatrixLib {
  implicit def unit (d : Double) : Rep[Double]
  def infix_× (a : Rep[Double], b : Rep[Double]) : Rep[Double]
  def infix_⊗ (a : Rep[Double], b : Rep[Matrix]) : Rep[Matrix]
}
```


OPPORTUNITY
Did you know that Chinese uses the same character for 'Problem' and 'Experience Points'?

Optimisation de la multiplication par un scalaire

- De nouveau, des optimisations spécifiques au domaine permettent de factoriser des calculs :

$$(kA)(k'B) = (kk')(AB)$$

Définition de matrices

```
trait MatrixLib3 extends MatrixLib2 {
  def define (nrows : Int, ncols : Int, f : (Int, Int) => Rep[Double])
      : Rep[Matrix]
  def get (m : Rep[Matrix], row : Int, col : Int) : Rep[Double]
}
```

Un utilisateur malin

"Supposons que je connaisse une matrice A, à un moment t et que je dois calculer $A * B_i$ pour une liste de matrices (B_i) qui m'est donnée par la suite, est-ce que je ne pourrais pas me créer une version **spécialisée** de mon produit ?"

Tu y es presque, petit malin !

```
trait Client3 extends Client { this : MatrixLib3 =>
  def specialize_product (a : Rep[Matrix]) : Rep[Matrix] => Rep[Matrix] = {
    val m = eval (a)
    (b : Rep[Matrix]) =>
      define (m.nrows, nrows (b),
        (i, j) => {
          var zero : Rep[Double] = 0
          for (k ← Range (0, m.ncols))
            zero = zero + m (i, k) × get (b, k, j)
          zero
        })
  }

  def f = eval ((c * c) * (c * a))
  def g = {
    val prod = specialize_product (c)
    eval (prod (c) * prod (a))
  }
}
```

Il faudrait pouvoir évaluer sous les λ.

De quoi avons-nous besoin...

```
trait MatrixLib3 extends MatrixLib2 {
  def define (nrows : Int, ncols : Int,
              f : (Int, Int) => Rep[Double]) : Rep[Matrix]
  def get (m : Rep[Matrix], row : Int, col : Int) : Rep[Double]
  // Représentation des fonctions.
  def lam[A, B] (f : Rep[A] => Rep[B]) : Rep [A => B]
  def app[A, B] (f : Rep[A => B], x : Rep[A]) : Rep[B]
}
```

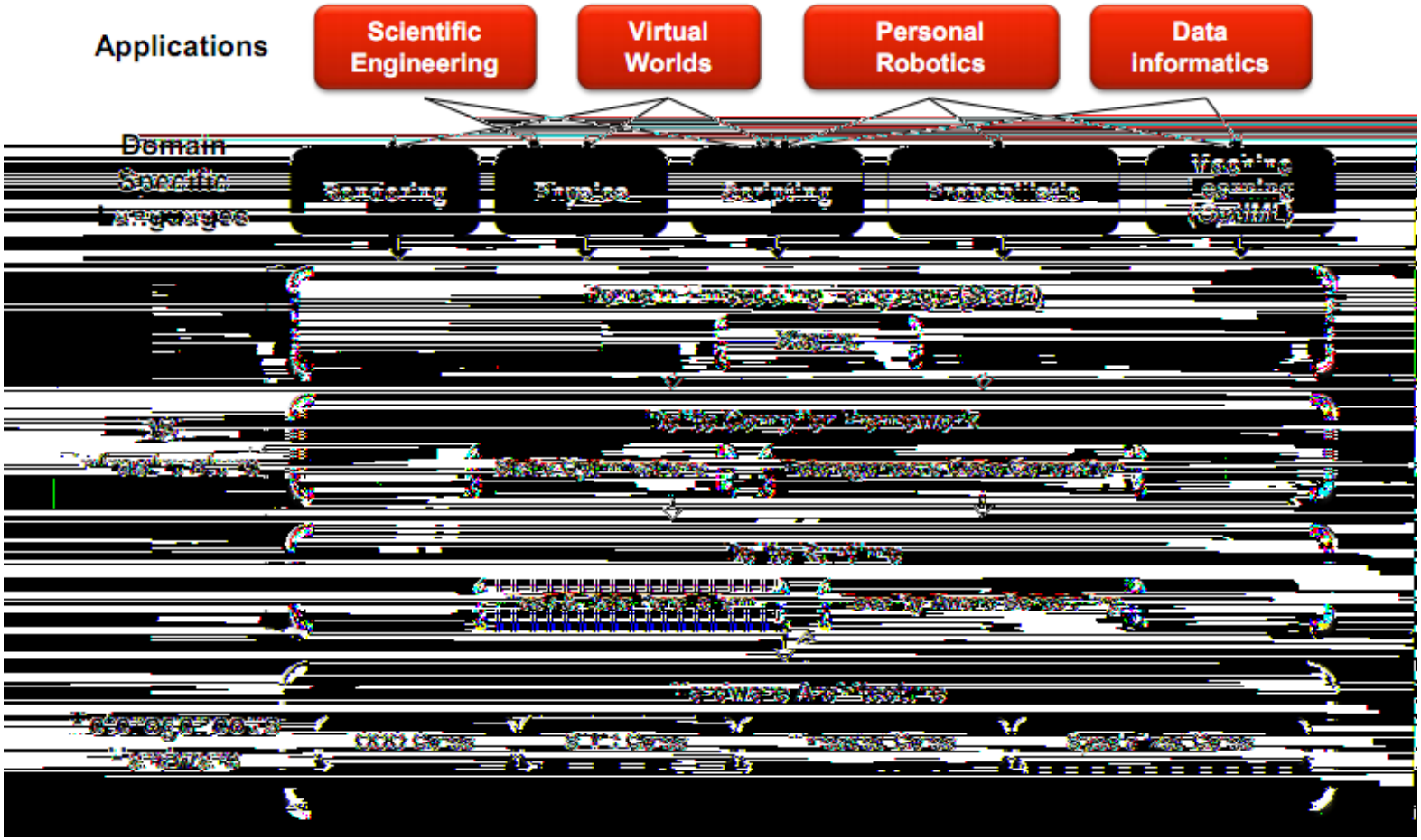
Carnet de vol

Le framework DELITE

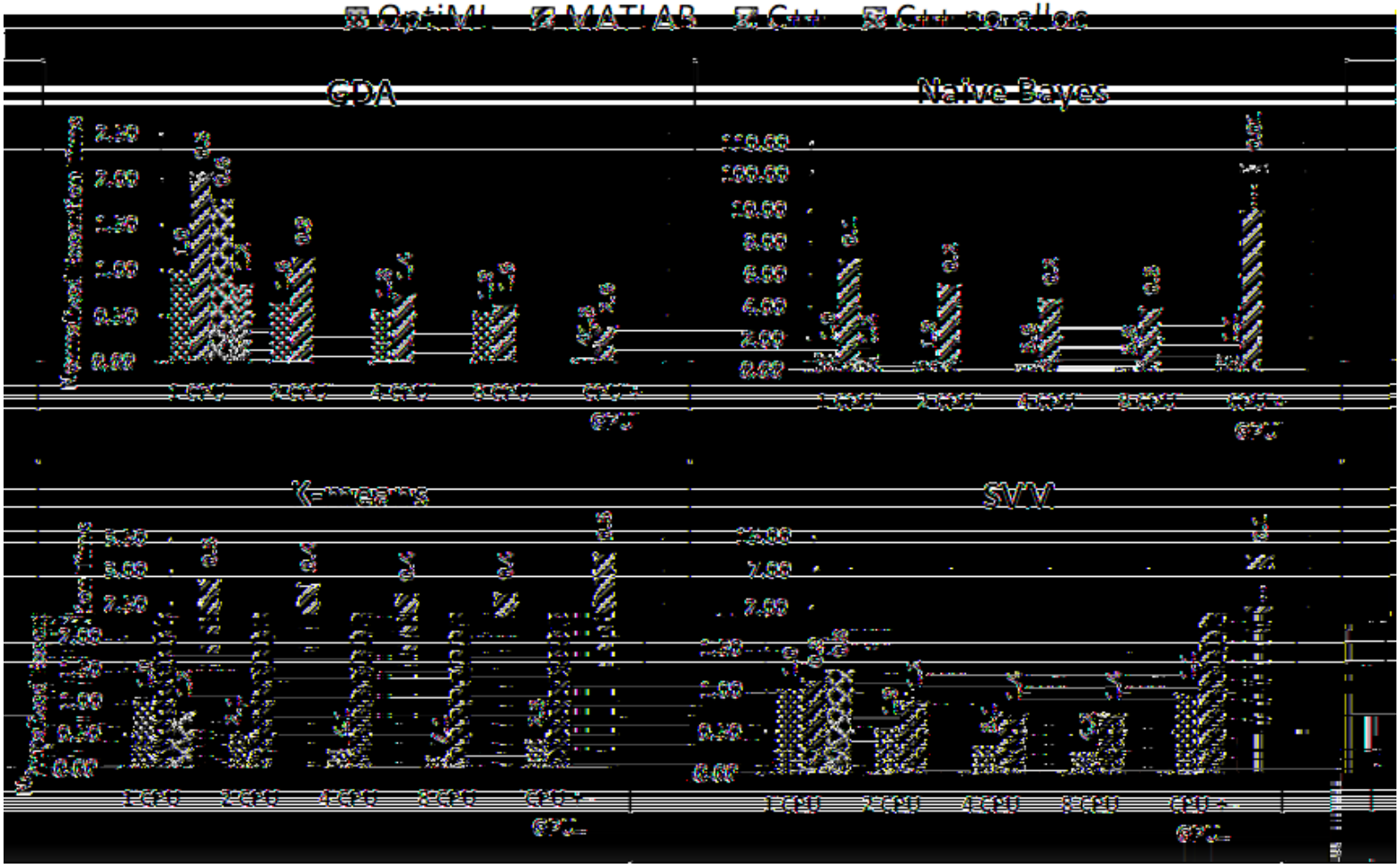
`http://stanford-ppl.github.com/Delite/getting_started.html`

- Il est développé par l'équipe PPL de Stanford et par l'EPFL.
- C'est une bibliothèque pour écrire des DSLs qui produisent du code à l'exécution vers des architectures hétérogènes.
- OPTIML est un DSL développé pour les algorithmes de machine learning, une sorte de MATHLAB boosté.

Architecture de Delite



Résultats préliminaires



Carnet de vol

Le passage à l'échelle avec SCALA

L'embarquement polymorphe d'un langage dans un autre

La modularisation d'un compilateur

Le framework DELITE

Conclusion

Idées à ramener à la maison

À l'aide des techniques d'embarquement polymorphe et de programmation modulaire, un DSL peut être virtualisé de façon sûre dans un langage hôte et profiter du compilateur de ce dernier tout en intégrant ses propres optimisations spécifiques.