Introduction to Multi-stage Programming with Dotty

Kenichi Suzuki

Visional Incubation, Inc.
yamory

Goals for Today

- Understanding what MSP (multi-stage programming) is
- Learn the basics of MSP, and how to use it in Dotty
- Understanding what tagless-final is
- Learn how to create interpreters (as code generators) for DSL using tagless-final

本日はMSPの基礎とその使い方、tagless-finalと組み合わせたインタプリタの作成について見ていきます

Introduction

Multi-stage Programming

- Multi-stage programming (MSP) is a paradigm for developing generic software, designed to address a number of problems with dynamic code generation. [1,2]

[1] W. Taha. Multi-Stage Programming: Its Theory and Applications. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999.

[2] W. Taha. A gentle introduction to multi-stage programming. In Domain-Specific Program Generation, Springer LNCS 3016, pages 30--50, 2003.

MSP はコード生成とその実行をサポートするプログラミングパラダイムです

Benefits and Problem of Code Generation

- Benefits
  - Maintainability and productivity
  - Performance
- Problems
  - Meta-programming is highly error prone, if done by hand
  - Generated code is too hard to debug

コード生成はメンテナンス性や生産性、パフォーマンスの向上できるメリットがあります
一方で、コード生成を直接手で記述すると、どこかで間違えたときにデバッグが難しくなります

Advantages of MSP

- MSP languages ensure followings:
  - Any generator only produces syntactically well-formed
  - Any generated program is also well-typed
  - Inadvertent name capture is not possible
- MSP provide good balance of abstraction and high performance
  - It helps programmers leverage program specialization to optimize evaluation costs
  - Abstraction without Guilt/Reglet

MSPは生成されたコードの構文や型安全を保証します
MSPは抽象とパフォーマンスを両立します

Basics

General-purpose vs Special-purpose

- Classic example:

```
def power(a: BigInt, x: Int): BigInt = x match {
  case 0 => 1
  case _ => a * power(a, x-1)
}
```

古典的な例でMSPの使い方をみていきましょう
これは、べき乗関数を素直に記述したサンプルプログラムです

General-purpose vs Special-purpose

- Classic example:

```
def power(a: BigInt, x: Int): BigInt = x match {
  case 0 ⇒ 1
  case _ ⇒ a * power(a, x-1)
}


power(2, 10)   // 2^10 ⇒ 1024
```

底 (base) を2、べき指数 (exponent) が10のべき (power) を得るには **power**(2,10) とします

General-purpose vs Special-purpose

- General-purpose programs can be easier to implement and more reusable, but run more slowly than special-purpose programs

General-purpose program:

```
def power(a: BigInt, x: Int): BigInt =
 x match {
  case 0 => 1
  case _ => a * power(a, x-1)
}
```

Special-purpose program:

```
def power10(a: BigInt) =
a * a * a * a * a * a * a * a *
a * a
```

No recursive calls

左が汎用的な再帰で記述したpower関数、右がある数xの10乗に特化したpower関数です
汎用的に記述したプログラムは再利用性が高いですが、特化されたプログラムに比べて遅くなります

General-purpose vs Special-purpose

- General-purpose programs can be easier to implement and more reusable, but run more slowly than special-purpose programs

General-purpose program:

```
def power(a: BigInt, x: Int): BigInt =
 x match {
  case 0 => 1
  case _ => a * power(a, x-1)
}
```

Special-purpose program:

```
def power1000(a: BigInt) =
      a * a * a * a * … * a
```

1000 a's

Again, no recursive calls

再帰がないため、1000乗の場合は効果はより顕著になります

Specializing

- How can we convert from a general-purpose program to a special-purpose program?

General-purpose program:

```
def power(a: BigInt, x: Int): BigInt =
 x match {
  case 0 => 1
  case _ => a * power(a, x-1)
}
```

?

Special-purpose program:

```
def power1000(a: BigInt) =
      a * a * a * a * … * a
```

汎用プログラムから特化されたプログラムにどのように変換できるでしょうか？

Use Staging Facilities

- Dotty supports MSP
- Here we use Dotty 0.27.0-RC1

```
$ sbt new lampepfl/dotty-staging.g8
```

```scala
import scala.quoted._
import scala.quoted.staging._

def main(args: Array[String]) = {
  given Toolbox = Toolbox.make(getClass.getClassLoader)
  …
}
```

sbt new でステージングを使うためのプロジェクトを生成します
Quotationを扱うためにToolboxを与えておきます

Basic Three Constructs of MSP

- Brackets
- Escape
- Run

MSPにおける3つの基本的な構成子は ブラケット、エスケープ、Runです

Brackets (a.k.a Quotations)

- Brackets '{ … } can be inserted around any (normal) expression, changing its type and delaying its evaluation

```
def x:        Int =     1 + 2
```

任意の式をブラケットで囲むことで、その式の評価を遅らせることができます

Brackets (a.k.a Quotations)

- Brackets '{ … } can be inserted around any (normal) expression, changing its type and delaying its evaluation

```
def x: Expr[Int] = '{ 1 + 2 }
```

任意の式をブラケットで囲むことで、その式の評価を遅らせることができます

Brackets (a.k.a Quotations)

- Brackets '{ … } can be inserted around any (normal) expression, changing its type and delaying its evaluation

```
def x:       Int =     1 + 2
```

Stage 0: Current

```
def x: Expr[Int] = '{ 1 + 2 }
```

Stage 1: Future

This calculation was delayed. That is, this calculation was sent to the future (stage 1).

任意の式をブラケットで囲むことで、その式の評価を遅らせることができます

Brackets (a.k.a Quotations)

- NOTE: Hereafter, `` `using ctx: QuoteContext` `` is omitted

```scala
def x:        Int =     1 + 2
```

```scala
def x(using ctx: QuoteContext): Expr[Int] = '{ 1 + 2 }
```

実際には暗黙のQuoteContextが必要ですが、簡単のため、本発表ではQuoteContext の宣言は省略します

Brackets (a.k.a Quotations)

- NOTE: Hereafter, `` `using ctx: QuoteContext` `` is omitted

```
def x:       Int =    1 + 2
```

```
def x: Expr[Int] = '{ 1 + 2 }
```

実際には暗黙のQuoteContextが必要ですが、簡単のため、本発表ではQuoteContext の宣言は省略します

Escape (a.k.a. Splice)

- Used for combining smaller fragments of code into larger ones

```
def x: Expr[Int] = '{ 1 + 2 }
```

```
def xx: Expr[Int] = '{ $x + $x }
```

エスケープ、あるいはスプライスは、コード片をコードの中で展開し、
他のコード片とつなぎ合わせられるようにします

Escape (a.k.a. Splice)

- Used for combining smaller fragments of code into larger ones

```
def x: Expr[Int] = '{ 1 + 2 }
```

```
def xx: Expr[Int] = '{ $x + $x }
```

エスケープ、あるいはスプライスは、コード片をコードの中で展開し、
他のコード片とつなぎ合わせられるようにします

Escape (a.k.a. Splice)

- Used for combining smaller fragments of code into larger ones

```
def x: Expr[Int] = '{ 1 + 2 }
```

```
def xx: Expr[Int] = '{ $x + $x }
```

```
'{ (1+2) + (1+2) }
```

エスケープを適用すると、未来のステージにあった計算が、現在のステージの計算として扱われます

Escape (a.k.a. Splice)

- Used for combining smaller fragments of code into larger ones

```
def x: Expr[Int] = ‘{ 1 + 2 }
```

```
def xx: Expr[Int] = ‘{ $x + $x }
```

```
‘{ (1+2) + (1+2) }
```

Stage 0

エスケープを適用すると、未来のステージにあった計算が、現在のステージの計算として扱われます

Escape (a.k.a. Splice)

- Used for combining smaller fragments of code into larger ones

```
def x: Expr[Int] = '{ 1 + 2 }
```

```
def xx: Expr[Int] = '{ $x + $x }
```

```
'{ (1+2) + (1+2) }
```

Stage 1

Stage 0

エスケープを適用すると、未来のステージにあった計算が、現在のステージの計算として扱われます

Bracket and Escape

- Bracket and Escape are dual

$$\$\{'\{\ e\ \}\} = e$$

$$'\{\$e\} = e$$

BracketとEscapeの操作は双対（dual）です

Run

- Used to compile and execute the dynamically generated code

```
def x: Expr[Int] = '{ 1 + 2 }

run(x)  // 1 + 2 ⇒ 3
```

run は構築されたコードを実行（評価）します

Staging Level

- The level of a term is:
  - the number of surrounding brackets - the number of surrounding escapes
  - e.g.)

2-1 =1

```
'{ ${'{ 1 + 2 }} * ${'{ 1 + 2 }} }
```

ある項のステージレベルは、「周囲のブラケットの数– 周囲のエスケープの数」で求められます

Specializing

- How can we convert from a general-purpose program to a special-purpose program?
  - We can use MSP constructs

General-purpose program:

```
def power(a: BigInt, x: Int): BigInt =
 x match {
  case 0 ⇒ 1
  case _ ⇒ a * power(a, x-1)
}
```

?

Special-purpose program:

```
def power1000(a: BigInt) =
      a * a * a * a * … * a
```

MSPの構成子を使って、汎用プログラムにステージングの注釈を付与します

Specializing with MSP

[3] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In Proceedings of the Symposium on Partial Evaluation and Semantic-Based Program Manipulation (PEPM), pages 203–217, Amsterdam, 1997. ACM Press.

- Staging = Conventional Program + Staging Annotation [3]

General-purpose program with staging annotations:

```
def powerCode(a: Expr[BigInt], x: Int):
Expr[BigInt] =
 x match {
  case 0 ⇒ ‘{ 1 }
  case _ ⇒ ‘{ $a * ${ powerCode(a, x-1) }}
}
```

power 関数にステージ注釈を付けました
この関数の戻り型 (return type) は Expr になり、べき乗を計算するコードが生成されるようになります

Specializing with MSP

[3] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In Proceedings of the Symposium on Partial Evaluation and Semantic-Based Program Manipulation (PEPM), pages 203–217, Amsterdam, 1997. ACM Press.

- Staging = Conventional Program + Staging Annotation [3]

General-purpose program with staging annotations:

```
def powerCode(a: Expr[BigInt], x: Int):
Expr[BigInt] =
 x match {
  case 0 ⇒ '{ 1 }
  case _ ⇒ '{ $a * ${ powerCode(a, x-1) }}
}
```

Generator to generate a specialized program:

```
def stagedPower(x: Int): BigInt ⇒ BigInt = {
  def code = '{
     (a: BigInt) ⇒ ${ powerCode('a, x) }
  }

  run(code)
}
```

さらに、ある数に特化したべき乗を計算するプログラムを生成するジェネレータを作成します

Specializing with MSP

[3] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In Proceedings of the Symposium on Partial Evaluation and Semantic-Based Program Manipulation (PEPM), pages 203–217, Amsterdam, 1997. ACM Press.

- Staging = Conventional Program + Staging Annotation [3]

General-purpose program with staging annotations:

```
def powerCode(a: Expr[BigInt], x: Int):
Expr[BigInt] =
 x match {
  case 0 ⇒ '{ 1 }
  case _ ⇒ '{ $a * ${ powerCode(a, x-1) }}
}
```

Generator to generate a specialized program:

```
def stagedPower(x: Int): BigInt ⇒ BigInt = {
  def code = '{
     (a: BigInt) ⇒ ${ powerCode('a, x) }
  }

  run(code)
}
```

```
val power1000 = stagedPower(1000)
```

このジェネレータから、ある数nに対する1000乗を計算することに特化したプログラムを生成します

Specializing with MSP

[3] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In Proceedings of the Symposium on Partial Evaluation and Semantic-Based Program Manipulation (PEPM), pages 203–217, Amsterdam, 1997. ACM Press.

- Staging = Conventional Program + Staging Annotation [3]

General-purpose program with staging annotations:

```
def powerCode(a: Expr[BigInt], x: Int):
Expr[BigInt] =
  x match {
   case 0 ⇒ ‘{ 1 }
   case _ ⇒ ‘{ $a * ${ powerCode(a, x-1) }}
}
```

Generator to generate a specialized program:

```
def stagedPower(x: Int): BigInt ⇒ BigInt = {
  def code = '{
     (a: BigInt) ⇒ ${ powerCode('a, x) }
  }

  run(code)
}
```

```
val power1000 = stagedPower(1000)
power1000(2)  // ⇒1.0715086071862673E301
```

このジェネレータから、ある数nに対する1000乗を計算することに特化したプログラムを生成します

Specializing with MSP

General-purpose program with staging annotations:

```
def powerCode(a: Expr[BigInt], x: Int):
Expr[BigInt] =
 x match {
  case 0 ⇒ ‘{ 1 }
  case _ ⇒ ‘{ $a * ${ powerCode(a, x-1) }}
}
```

Generator to generate a specialized program:

```
def stagedPower(x: Int): BigInt ⇒ BigInt = {
  def code = '{
     (a: BigInt) ⇒ ${ powerCode('a, x) }
  }

  run(code)
}
```

```
val power1000 = stagedPower(1000)
// ((a: scala.BigInt) ⇒ a.*(a.*(a.*(a.*(a.*(a.*(a.*(a.*(a.*(a.*(a.*(a.*(…

power1000(2)  // 2 * 2 * 2 * … * 2 ⇒ 1.0715086071862673E301
```

特化したプログラムpower1000は、再帰呼び出しのないパフォーマンスのよいプログラムのため、
実際に計算すると素早く結果が返ってきます

Staged Interpreter (Translator)

Unstaged Interpreter

- A typical problem with writing interpreters is the performance overhead required when execute programs

```
enum Exp {
  case IntLit(x: Int)
  case Add(e1: Exp, e2: Exp)
  case Mul(e1: Exp, e2: Exp)
  …
}
```

```
def eval(e: Exp): Int = e match {
  case IntLit(x)    => x
  case Add(e1, e2) => eval(e1) + eval(e2)
  case Mul(e1, e2) => eval(e1) * eval(e2)
  ...
}
```

インタプリタを素朴に作ると、タグで構成されたユーザプログラムを解釈するためにパターンマッチと再帰を多用することになります

Unstaged Interpreter

- I want to write a user program according to DSL
- On the other hand, is it possible to convert the user program into a form that can be processed directly by Scala instead of the interpreter?

```scala
enum Exp {
  case IntLit(x: Int)
  case Add(e1: Exp, e2: Exp)
  case Mul(e1: Exp, e2: Exp)
  …
}
```

```scala
Add(IntLit(1),
  Mul(IntLit(2), IntLit(3)))
```

No

```scala
def eval(e: Exp): Int = e match {
  case IntLit(x)    => x
  case Add(e1, e2) => eval(e1) + eval(e2)
  case Mul(e1, e2) => eval(e1) * eval(e2)
  ...
```

Yes

```
1 + (2 * 3)  ⇒ 7
```

ユーザプログラムはDSLに従ったまま、そのプログラムの最終的な計算を、インタプリタではなく、Scala上で直接解釈させるにはどうしたらよいでしょうか？

Staged Interpreter

- We can also use the MSP constructs for the interpreter

```
enum Exp {
  case IntLit(x: Int)
  case Add(e1: Exp, e2: Exp)
  case Mul(e1: Exp, e2: Exp)
  …
}
```

```
Add(IntLit(1),
  Mul(IntLit(2), IntLit(3)))
```

Yes

```
def eval(e: Exp): Int = e match {
  case IntLit(x)   ⇒ ‘{ x }
  case Add(e1, e2) ⇒ ‘{ ${eval(e1)} + ${eval(e2)} }
  case Mul(e1, e2) ⇒ ‘{ ${eval(e1)} * ${eval(e2)} }
  …
```

```
1 + (2 * 3)
  ⇒ 7
```

インタプリタにステージングアノテーションを付与することで、
ユーザプログラムを変換することができます

Staged Interpreter

- We can also use the MSP constructs for the interpreter

```
enum Exp {
  case IntLit(x: Int)
  case Add(e1: Exp, e2: Exp)
  case Mul(e1: Exp, e2: Exp)
  …
}
```

```
def eval(e: Exp): Int = e match {
  case IntLit(x)    ⇒ '{ x }
  case Add(e1, e2) ⇒ '{ ${eval(e1)} + ${eval(e2)} }
  case Mul(e1, e2) ⇒ '{ ${eval(e1)} * ${eval(e2)} }
```

A staged interpreter is a translator

```
Add(IntLit(1),
  Mul(IntLit(2), IntLit(3)))
```

```
1 + (2 * 3)
  ⇒ 7
```

ステージ化されたインタプリタは変換器とみなせます

Tagless-final

What is tagless-final?

- The tagless-final approach[4] is an approach of embedding domain-specific languages (DSLs) in a typed functional language
  - It is also called "final embedding", "tagless", "typed-final" or "tagless encoding"
  - Doing a tagless-final embedding is literally writing a denotational semantics for the DSL
- The tagless-final approach has collected and polished a number of techniques for representing typed higher-order languages in a typed metalanguage, along with type-preserving interpretation, compilation and partial evaluation

[4] J. Carette, O. Kiselyov, and C. Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. J. Funct. Program., 19(5):509–543, 2009.

tagless-final は埋込みDSLを構築する手法の1つです
この手法は埋込みで高階言語を表現するための数々のテクニックを集めて体系化したものです

Advantages of tagless-final

- Typing is done at a meta-language
  - No need to write typing algorithms
- We can use HOAS (higher-order abstract syntax) to represent object language binders using meta language binders
  - Meta-language ensures hygiene
- We don't have to make a parser
  - An object term is represented as a meta-language term

Tagless-finalによる埋込みでは、型付けはメタ言語のそれに帰着します
HOAS（高階抽象構文）を使うことで、メタ言語によってスコープ安全が保証されます

Object Language and Metalanguage

- Object language
  - A language we want to represent
  - DSL
- Metalanguage
  - A language used to describe an object language

対象言語は我々が表現したい言語（DSL）です
メタ言語は対象言語の埋め込む先となる、ScalaやHaskell等の言語です

Syntax and typing rules

The object language

Syntax

$$e ::= n \mid e + e$$

Typing rules

$$\frac{n \text{ is an integer}}{n : \mathbb{Z}} \qquad \frac{e_1 : \mathbb{Z} \qquad e_2 : \mathbb{Z}}{e_1 + e_2 : \mathbb{Z}}$$

Symantics

```scala
trait Symantics[Repr[_]] {
  def int(n: Int): Repr[Int]
  def add(e1: Repr[Int], e2: Repr[Int]): Repr[Int]
}
```

加算と整数リテラルのみの単純な言語を、
どのように tagless-finalでエンコーディングするか見ていきましょう

Syntax and typing rules

The object language

Syntax

$$e ::= \boxed{n} \mid \boxed{e + e}$$

Typing rules

$$\frac{n \text{ is an integer}}{n : \mathbb{Z}} \qquad \frac{e_1 : \mathbb{Z} \qquad e_2 : \mathbb{Z}}{e_1 + e_2 : \mathbb{Z}}$$

Symantics

```
trait Symantics[Repr[_]] {
  def int(n: Int): Repr[Int]
  def add(e1: Repr[Int], e2: Repr[Int]): Repr[Int]
}
```

対象言語の構成子と関数名を合わせます

Syntax and typing rules

The object language

Wrap the types in the representation type (Repr)

Syntax

$$e ::= n \mid e + e$$

Typing rules

$$\frac{n \text{ is an integer}}{n : \mathbb{Z}} \qquad \frac{e_1 : \mathbb{Z} \qquad e_2 : \mathbb{Z}}{e_1 + e_2 : \mathbb{Z}}$$

Symantics

```
trait Symantics[Repr[_]] {
  def int(n: Int): Repr[Int]
  def add(e1: Repr[Int], e2: Repr[Int]): Repr[Int]
}
```

次に、関数のシグニチャを型付け規則と一致させます
このとき、表現型 (Repr) で型を包みます

Syntax and typing rules

The object language

Associate the function signatures with the typing rules

Syntax

$$e ::= n \mid e + e$$

Typing rules

$$\frac{n \text{ is an integer}}{n : \boxed{\mathbb{Z}}} \qquad \frac{e_1 : \boxed{\mathbb{Z}} \qquad e_2 : \boxed{\mathbb{Z}}}{e_1 + e_2 : \boxed{\mathbb{Z}}}$$

Symantics

```
trait Symantics[Repr[_]] {
  def int(n: Int): Repr[Int]
  def add(e1: Repr[Int], e2: Repr[Int]): Repr[Int]
}
```

次に、関数のシグニチャを型付け規則と一致させます
このとき、表現型 (Repr) で型を包みます

Syntax and typing rules

The object language

Syntax

$$e ::= n \mid e + e$$

Typing rules

$$\frac{n \text{ is an integer}}{n : \mathbb{Z}} \qquad \frac{e_1 : \mathbb{Z} \qquad e_2 : \mathbb{Z}}{e_1 + e_2 : \mathbb{Z}}$$

Symantics

```
trait Symantics[Repr[_]] {
  def int(n: Int): Repr[Int]
  def add(e1: Repr[Int], e2: Repr[Int]): Repr[Int]
}
```

ここまでで対象言語の構文・型付け規則が表現されました
これらの表現に使われるインタフェースはSymanticsと呼ばれます

Interpreters

```scala
trait Symantics[Repr[_]] {
  def int(n: Int): Repr[Int]
  def add(e1: Repr[Int], e2: Repr[Int]): Repr[Int]
}
```

```scala
final case class R[A](unR: A)

object RInterpreter extends Symantics[R] {
  def int(n: Int): R[Int] = R(n)
  def add(a: R[Int], b: R[Int]): R[Int] =
    R(a.unR + b.unR)
}
def eval[A](e: R[A]): A = e.unR
```

Types are preserved

Evaluation function

Symanticsインターフェースに対する実装を意味論として与えることでインタプリタを作ります
インタプリタはSymanticsに従っているため、型を保存します

User program

- Terms are represented by a combination of functions

```
object example {
  import Interpreters._
  import Interpreters.RInterpreter._

  val program: R[Int] = add(int(1), int(2))
  eval(program) // ⇒ 3
}
```

ユーザプログラムは、関数の組み合わせで記述します
組み合わせてできた式は、型付け規則の表現（Repr）に従います

Higher-order abstract syntax

- Techniques for expressing variable bindings in the metalanguage abstraction

$$\dfrac{\begin{matrix}[x : \tau_1] \\ \vdots \\ e : \tau_2\end{matrix}}{\lambda x.e : \tau_1 \to \tau_2} \qquad \dfrac{e_1 : \tau_1 \to \tau_2 \qquad e_2 : \tau_1}{e_1\ e_2 : \tau_2}$$

```
trait LamSym[Repr[_]] {
  def lam[A, B](f: Repr[A] ⇒ Repr[B]): Repr[A ⇒ B]
  def app[A, B](e1: Repr[A ⇒ B], e2: Repr[A]): Repr[B]
}
```

変数束縛はメタ言語のそれにお任せします

Higher-order abstract syntax

- Techniques for expressing variable bindings in the metalanguage abstraction

$$\frac{\begin{array}{c}\boxed{\begin{array}{c}[x : \tau_1] \\ \vdots \\ e : \tau_2\end{array}}\end{array}}{\lambda x.e : \tau_1 \to \tau_2} \qquad \frac{e_1 : \tau_1 \to \tau_2 \qquad e_2 : \tau_1}{e_1\ e_2 : \tau_2}$$

```
trait LamSym[Repr[_]] {
  def lam[A, B](f: Repr[A] ⇒ Repr[B]): Repr[A ⇒ B]
  def app[A, B](e1: Repr[A ⇒ B], e2: Repr[A]): Repr[B]
}
```

束縛子と関数ボディを眺めると、矢印の関係が見えてきます

Higher-order abstract syntax

- Implementing interpreters are type puzzles :)

```scala
val interpret = new LamSym[R] {
    def lam[A, B](f: R[A] ⇒ R[B]): R[A ⇒ B] =
      R((x: A) ⇒ f(R(x)).unR)

    def app[A, B](e1: R[A ⇒ B], e2: R[A]): R[B] =
      R(e1.unR(e2.unR))
  }
```

インタプリタは型合わせゲーム

Higher-order abstract syntax

- Implementing interpreters are type puzzles :)

```
val interpret = new LamSym[R] {
    def lam[A, B](f: R[A] ⇒ R[B]): R[A ⇒ B] =
      R((x: A) ⇒ f(R(x)).unR)

    def app[A, B](e1: R[A ⇒ B], e2: R[A]): R[B] =
      R(e1.unR(e2.unR))
  }
```

インタプリタは型合わせゲーム

Higher-order abstract syntax

- Implementing interpreters are type puzzles :)

```
val interpret = new LamSym[R] {
    def lam[A, B](f: R[A] ⇒ R[B]): R[A ⇒ B] =
      R((x: A) ⇒ f(R(x)).unR)

    def app[A, B](e1: R[A ⇒ B], e2: R[A]): R[B] =
      R(e1.unR(e2.unR))
  }
```

インタプリタは型合わせゲーム

Higher-order abstract syntax

- Implementing interpreters are type puzzles :)

```scala
val interpret = new LamSym[R] {
   def lam[A, B](f: R[A] ⇒ R[B]): R[A ⇒ B] =
     R((x: A) ⇒ f(R(x)).unR)

   def app[A, B](e1: R[A ⇒ B], e2: R[A]): R[B] =
     R(e1.unR(e2.unR))
 }
```

インタプリタは型合わせゲーム

Higher-order abstract syntax

- Implementing interpreters are type puzzles :)

```scala
val interpret = new LamSym[R] {
  def lam[A, B](f: R[A] ⇒ R[B]): R[A ⇒ B] =
    R((x: A) ⇒ f(R(x)).unR)

  def app[A, B](e1: R[A ⇒ B], e2: R[A]): R[B] =
    R(e1.unR(e2.unR))
}
```

インタプリタは型合わせゲーム

Compose language components

```scala
trait BaseSym[Repr[_]] {
  def int(n: Int): Repr[Int]
  def add(e1: Repr[Int], e2: Repr[Int]): Repr[Int]
}

trait LamSym[Repr[_]] {
  def lam[A, B](f: Repr[A] ⇒ Repr[B]): Repr[A ⇒ B]
  def app[A, B](e1: Repr[A ⇒ B], e2: Repr[A]): Repr[B]
}

trait FullSym[Repr[_]] extends BaseSym[Repr] with LamSym[Repr]
```

言語コンポーネントを合成できます

Staged Tagless-final Interpreter

Optimizer + Code Generator

- How to optimize terms and finally generate code while represent the object language with the tagless-final style?

tagless-finalで対象言語を表現しつつ、最適化のための項の書き換えや、
コード生成を行うにはどうすればよいでしょうか

Optimizer + Code Generator

- How to optimize terms and finally generate code while represent the object language with the tagless-final style?
- Tagless-final style allow multiple interpretations for a language

tagless-finalスタイルでは、1つの言語に複数の解釈を与えられます

Optimizer + Code Generator

- How to optimize terms and finally generate code while represent the object language with the tagless-final style?
- Tagless-final style allow multiple interpretations for a language
- It is possible to combine different interpretations [5]

[5] Suzuki, K., Kiselyov, O., Kameyama, Y.: Finally, safely-extensible and efficient language-integrated query. In: Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, 20–22 January 2016, pp. 37–48 (2016)

それらの解釈を合成することができます

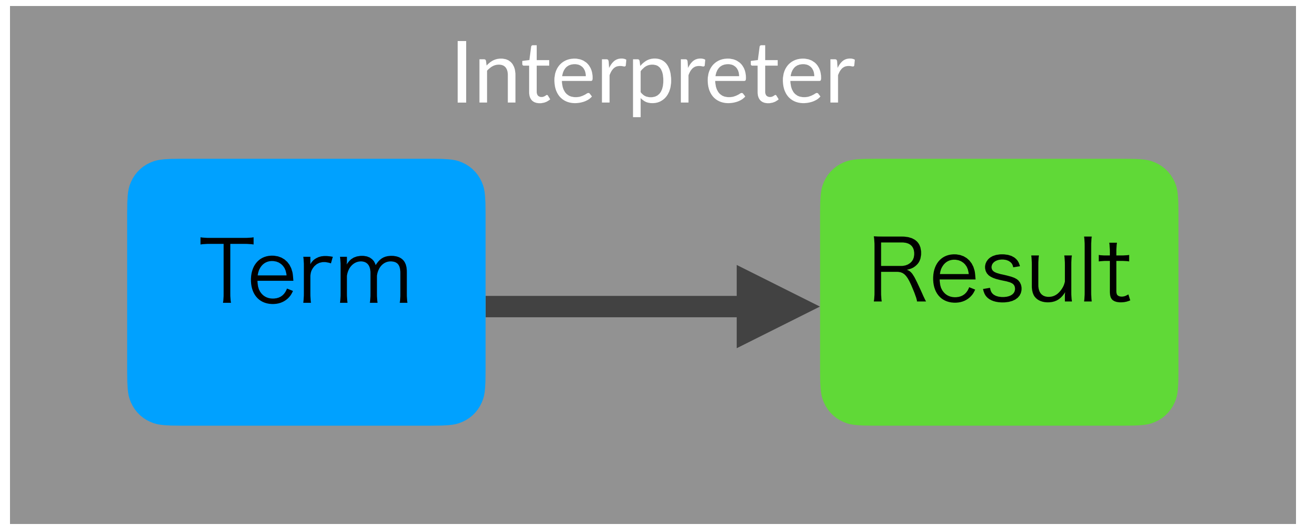
Final-style program transformations

- Make an interpreter as a module functor
- Compose those module functors
- Give a framework to reify and reflect the representation in each interpreter

インタプリタをモジュールファンクタとして構成し、それらのモジュールを合成します
そして、それぞれのインタプリタの表現を変換する枠組みを与えます

Final-style program transformations

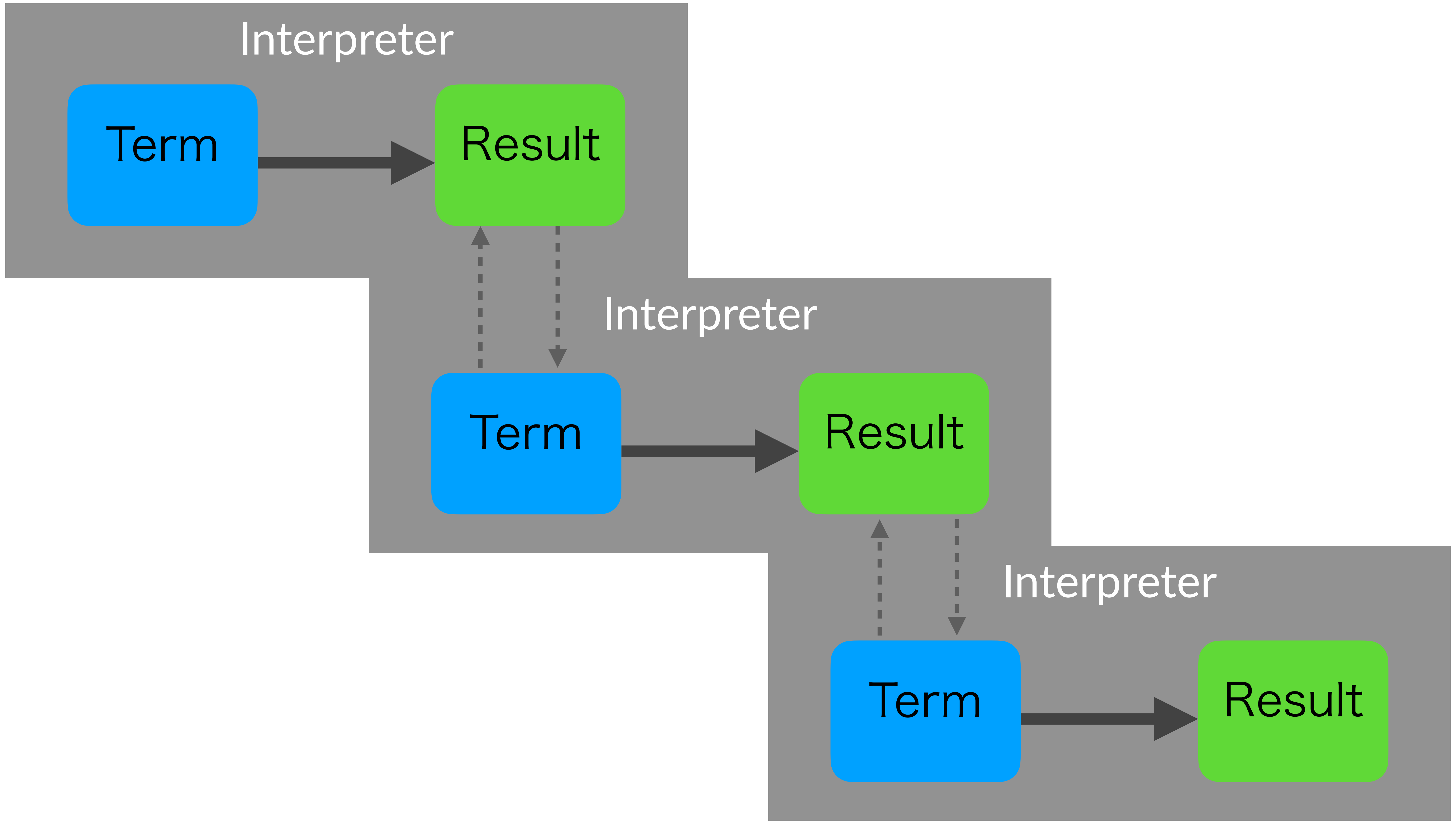
- An interpreter



ただのインタプリタ

Final-style program transformations

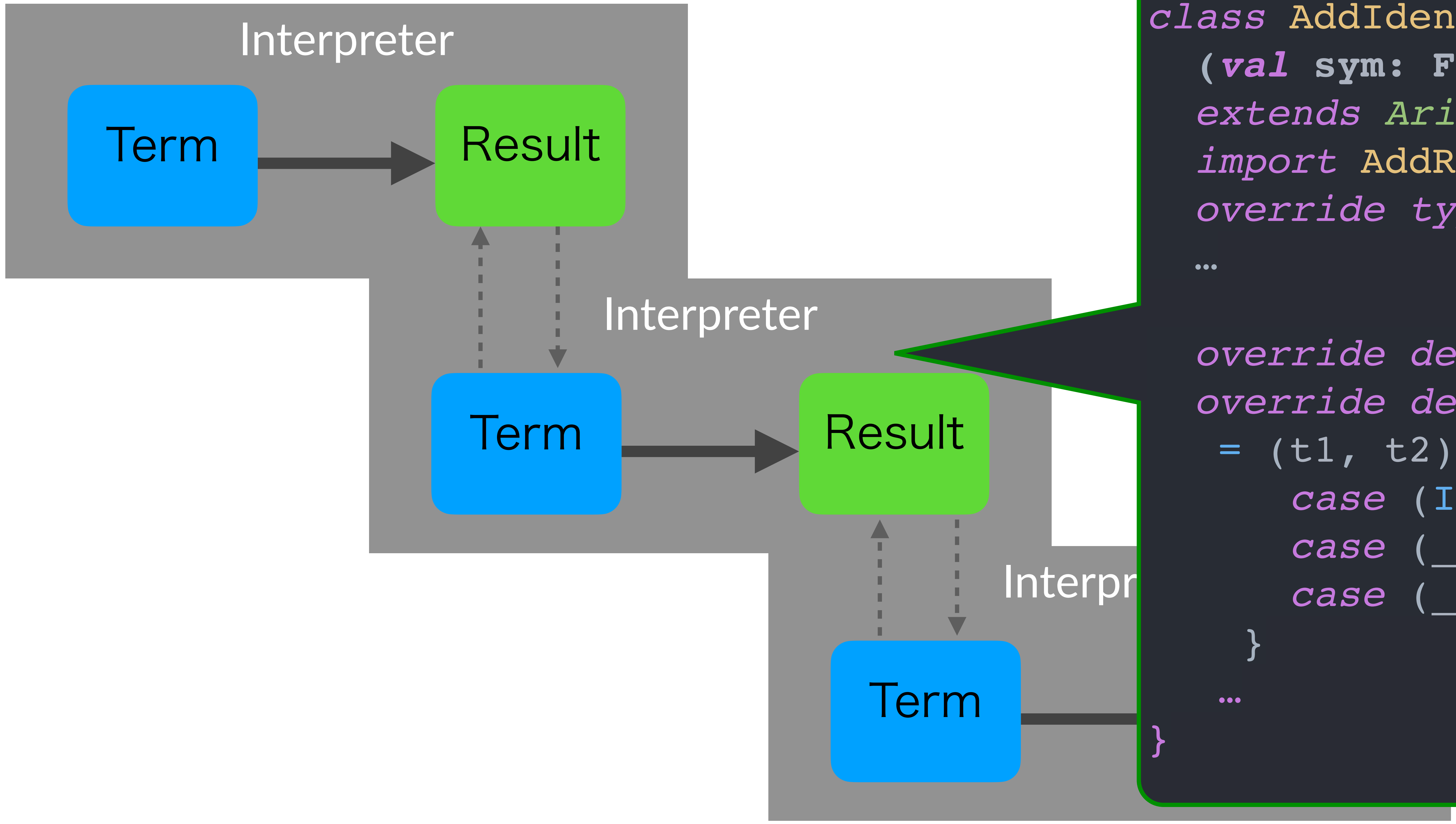
- Composed interpreters are also an interpreter



インタプリタをモジュールファンクタとすることで、複数のインタプリタを合成します

Final-style program transformations

- Composed interpreters are also an interpreter

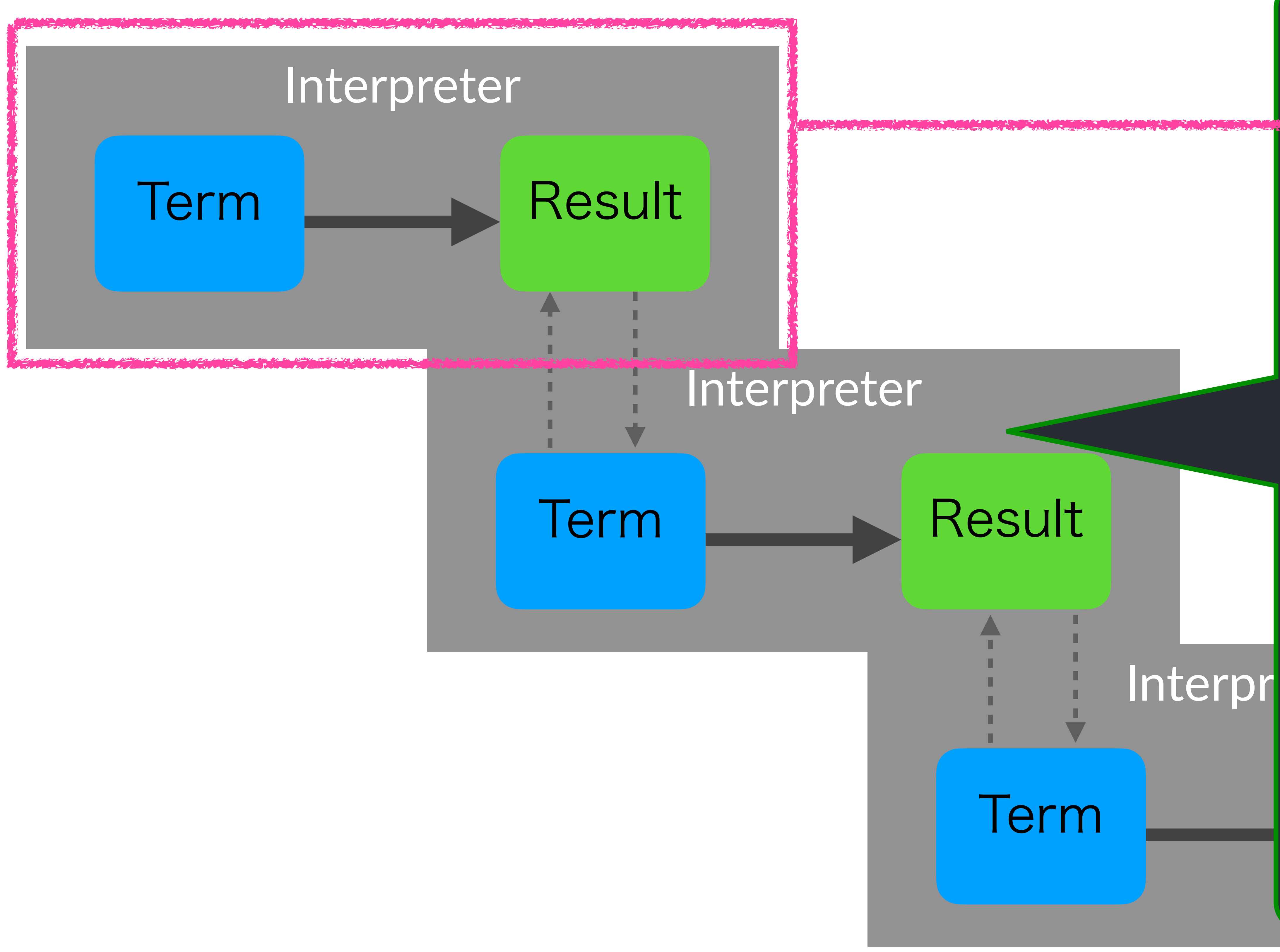



```scala
class AddIdentOptim[FS <: ArithSym]
  (val sym: FS {type Repr[T] = AddRR.From[T]})
  extends ArithSym {
  import AddRR._
  override type Repr[T] = To[T]
  …

  override def int(n: Int): To[Int] = IntLit(n)
  override def add(t1: To[Int], t2: To[Int]): To[Int]
   = (t1, t2) match {
      case (IntLit(0), _)          => t2
      case (_,         IntLit(0)) => t1
      case (_,         _)          => Add(t1, t2)
    }
   …
}
```

インタプリタをモジュールファンクタとすることで、複数のインタプリタを合成します

Final-style program transformations

- Composed interpreters are also an interpreter

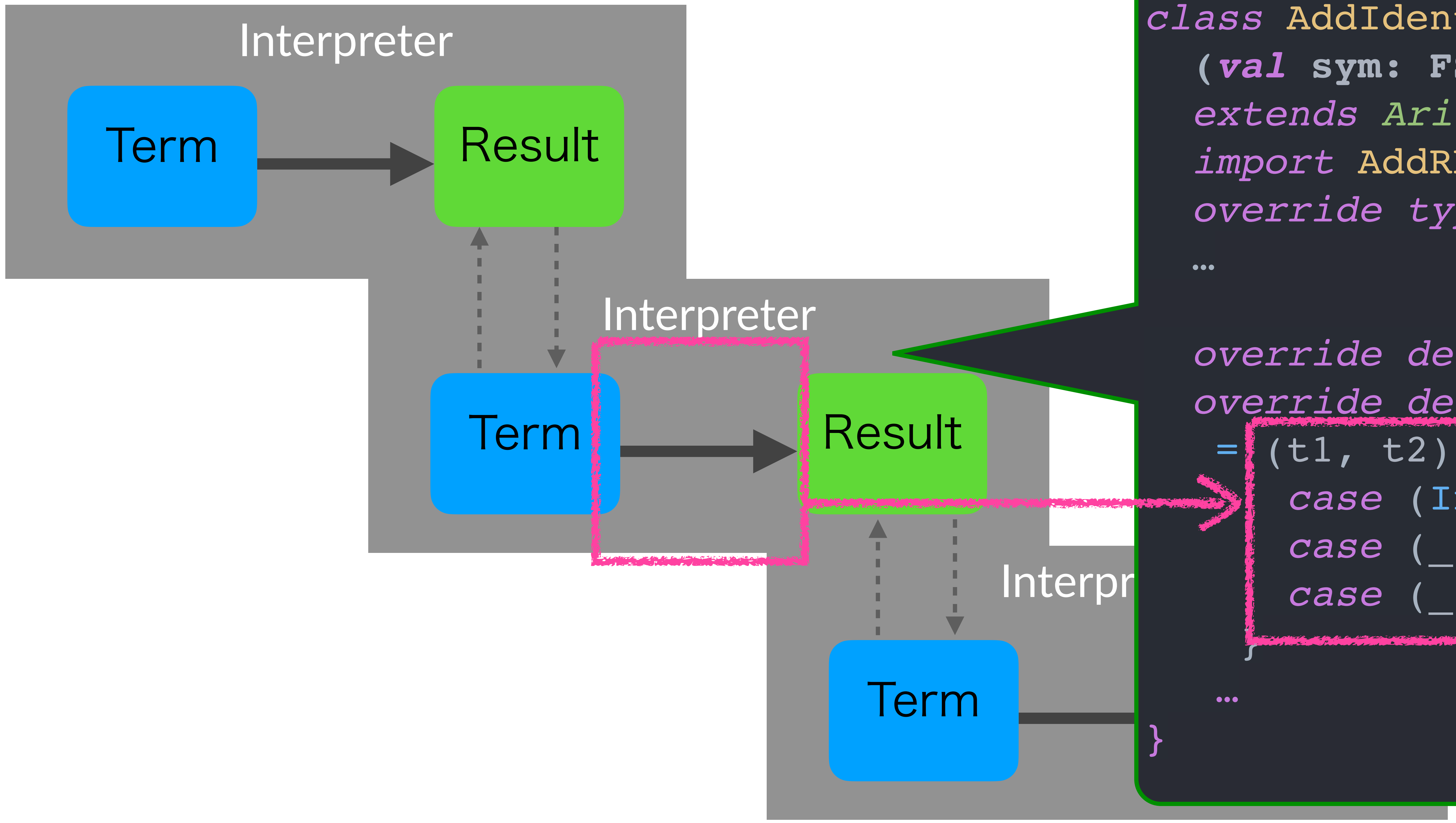



```scala
class AddIdentOptim[FS <: ArithSym]
  (val sym: FS {type Repr[T] = AddRR.From[T]})
  extends ArithSym {
  import AddRR._
  override type Repr[T] = To[T]
  …

  override def int(n: Int): To[Int] = IntLit(n)
  override def add(t1: To[Int], t2: To[Int]): To[Int]
   = (t1, t2) match {
      case (IntLit(0),  _)          => t2
      case (_,          IntLit(0))  => t1
      case (_,          _)          => Add(t1, t2)
    }
   …
}
```

インタプリタをモジュールファンクタとすることで、複数のインタプリタを合成します

Final-style program transformations

- Composed interpreters are also an interpreter

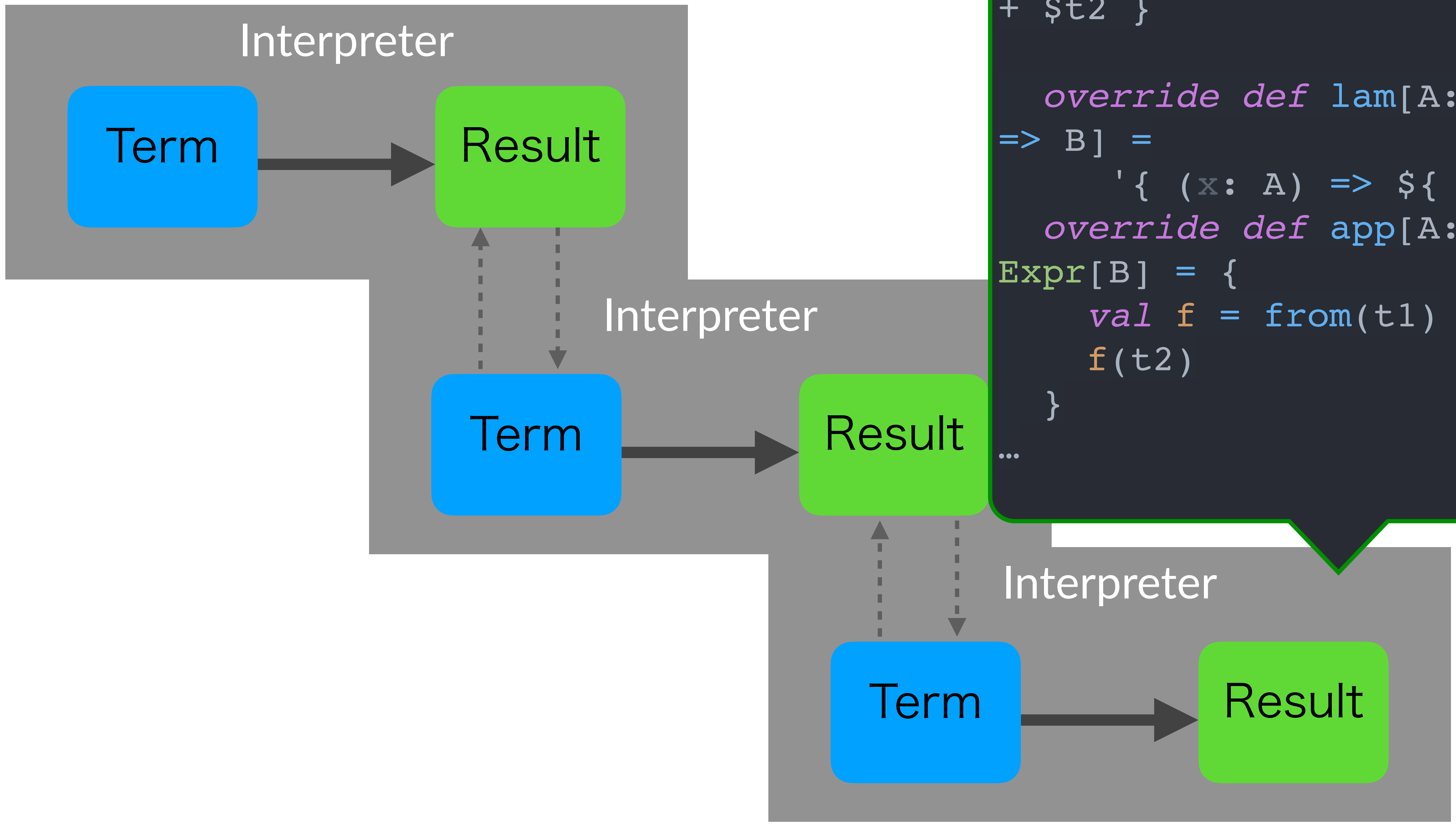



```
class AddIdentOptim[FS <: ArithSym]
  (val sym: FS {type Repr[T] = AddRR.From[T]})
  extends ArithSym {
  import AddRR._
  override type Repr[T] = To[T]
  …

  override def int(n: Int): To[Int] = IntLit(n)
  override def add(t1: To[Int], t2: To[Int]): To[Int]
   = (t1, t2) match {
    case (IntLit(0), _)          => t2
    case (_,         IntLit(0)) => t1
    case (_,         _)          => Add(t1, t2)
   }
   …
}
```

インタプリタをモジュールファンクタとすることで、複数のインタプリタを合成します

Final-style program transformations

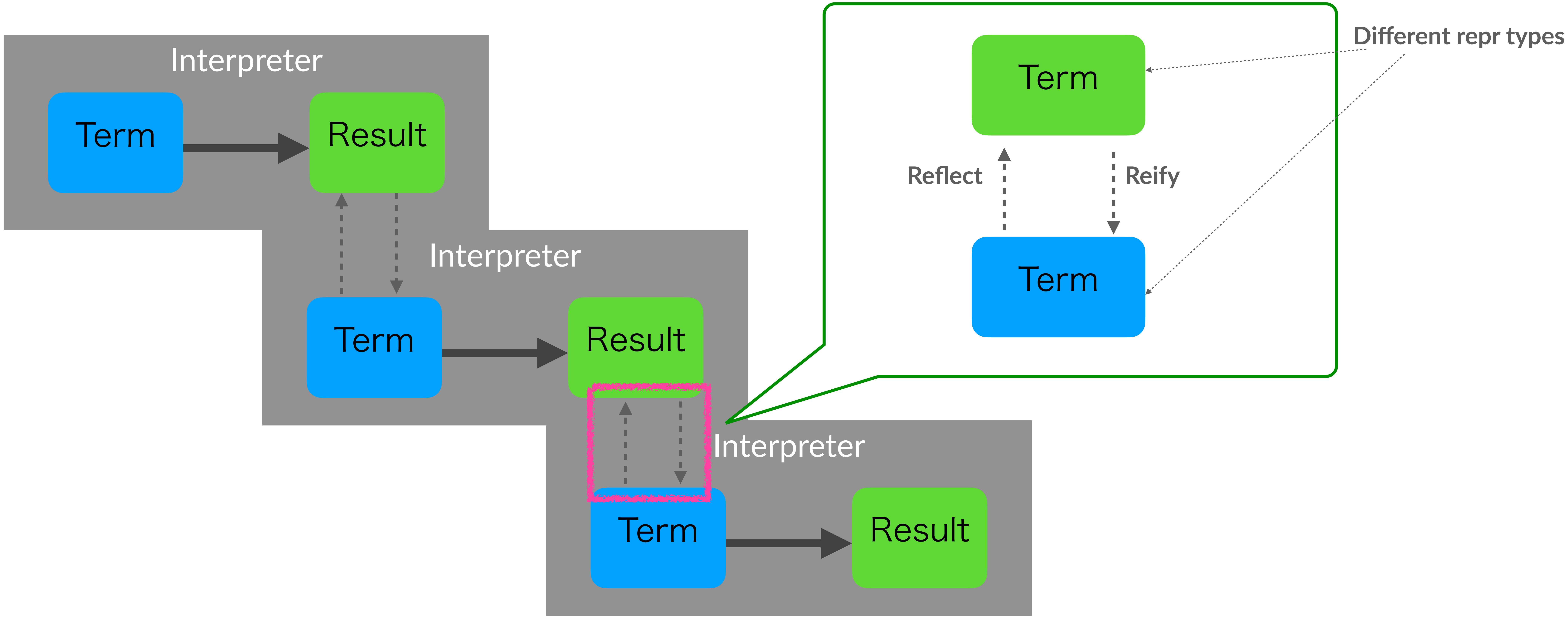
```
class Translator(using ctx: QuoteContext) extends FullSym {
  override type Repr[T] = Expr[T]

  override def int(n: Int): Expr[Int] = Expr(n)
  override def add(t1: Expr[Int], t2: Expr[Int]): Expr[Int] = '{ $t1
+ $t2 }

  override def lam[A: Type, B: Type](f: Expr[A] => Expr[B]): Expr[A
=> B] =
     '{ (x: A) => ${ f('x) }}
  override def app[A: Type, B: Type](t1: Expr[A => B], t2: Expr[A]):
Expr[B] = {
    val f = from(t1)
    f(t2)
  }
…
```



コード生成器をインタプリタとすることにより、最終的に最適化されたコードを生成することができます
このときの表現型はステージングで用いられる Expr 型です

Final-style program transformations

- Composed interpreters are also an interpreter



これらのインタプリタ間の表現型は異なるため、表現の行き来ができるようなフレームワークを与えます

Reflect-reify framework

```scala
trait RR {
  import cats.~>

  type From[_]
  type To[_] // Term

  def fwd: From ~> To // reflection
  def bwd: To ~> From // reification

  def map[A, B](f: From[A] ⇒ From[B]): To[A] ⇒ To[B] =
    (t: To[A]) ⇒ fwd(f(bwd(t)))

  def map2[A, B, C](f: (From[A], From[B]) ⇒ From[C]): (To[A], To[B]) ⇒ To[C] =
    (t1: To[A], t2: To[B]) ⇒ fwd(f(bwd(t1), bwd(t2)))

}
```

RRフレームワーク
自然変換を用いています

Example program and its results

- Identity optimization

```
val e1 = add(int(1), int(0))
    // 1 + 0
    // 1
val e2 = mul(int(2), int(10))
    // 2.*(10)
val e3 = add(mul(int(1), int(10)), int(0))
    // (1*10) + 0
    // 10 + 0
    // 10
```

単位元の除去により最適化した式が生成されました

Thank you for your kind attention

yamory

https://yamory.io