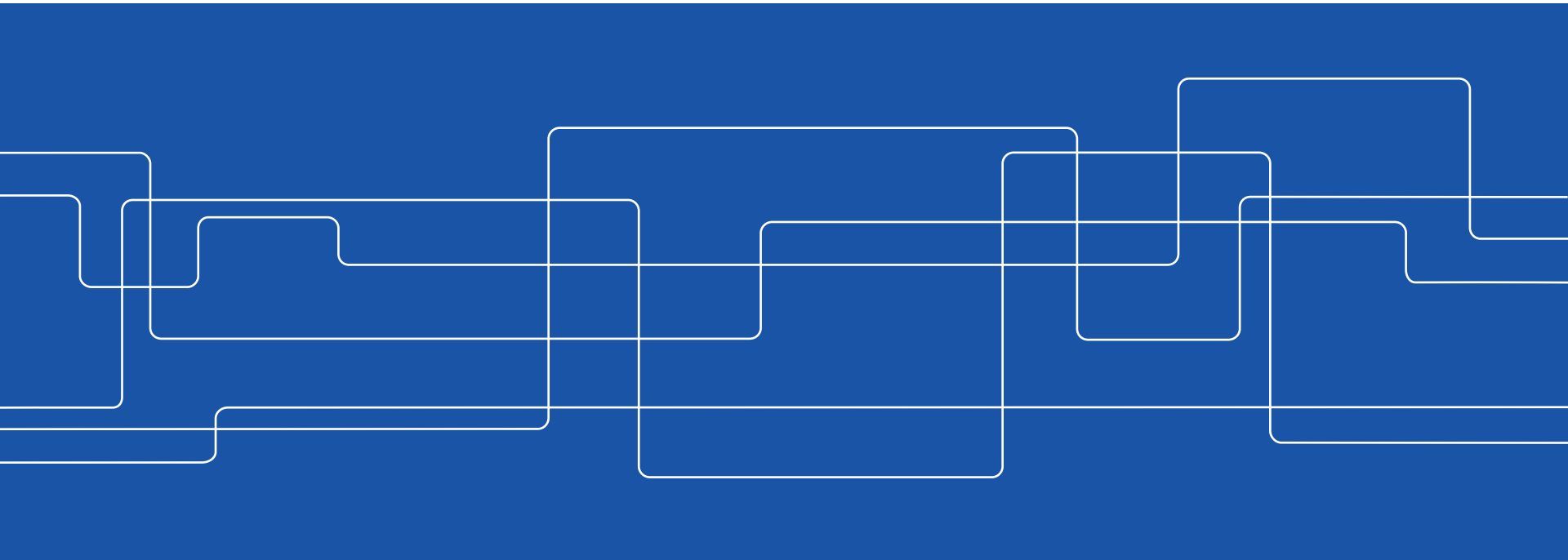
Rust code generation in Scala

Master Thesis

Klas Segeljakt



Outline

Introduction

- Problem
- Approach

Background

- The Rust programming language
- Domain specific languages
- The Scala programming language
- Related work

Summary & Questions

References

Problem

Modern general purpose distributed systems, e.g., Spark and Flink, run on the JVM [1]

Problem

Modern general purpose distributed systems, e.g., Spark and Flink, run on the JVM [1]

Byte code is 10x - 1000x slower than compiled optimized code [2]

Problem

Modern general purpose distributed systems, e.g., Spark and Flink, run on the JVM [1]

Byte code is 10x - 1000x slower than compiled optimized code [2]

Laptop with compiled code can outperform 128 node Spark cluster at PageRank [3]

Problem

Modern general purpose distributed systems, e.g., Spark and Flink, run on the JVM [1]

Byte code is 10x - 1000x slower than compiled optimized code [2]

Laptop with compiled code can outperform 128 node Spark cluster at PageRank [3]

=> Need better performance to support new programming models (ML tensors, graph analytics)

Approach

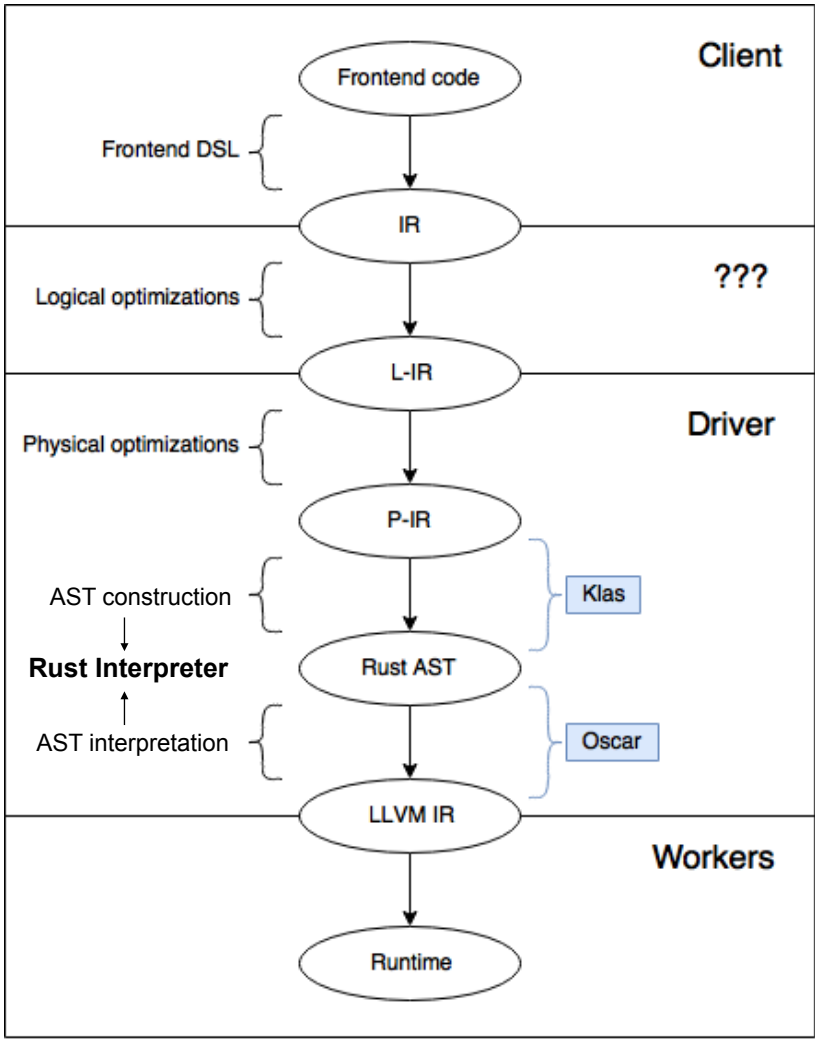
Build an interpreter in Scala for Rust

Given an IR, generate a Rust AST

IR design is not yet decided

Backtrace runtime errors

Future work: Translating UDFs



Desirable properties [4,5]

Reuse of infrastructure - Interpreter reuses the Scala language

Desirable properties [4,5]

Reuse of infrastructure - Interpreter reuses the Scala language

Pluggable semantics - Constructs can be interpreted in multiple ways

Desirable properties [4,5]

Reuse of infrastructure - Interpreter reuses the Scala language

Pluggable semantics - Constructs can be interpreted in multiple ways

Static safety - Static checking for syntactic and semantic errors

Desirable properties [4,5]

Reuse of infrastructure - Interpreter reuses the Scala language

Pluggable semantics - Constructs can be interpreted in multiple ways

Static safety - Static checking for syntactic and semantic errors

Runtime safety - Strict guarantees about generated program's behaviour

Desirable properties [4,5]

Reuse of infrastructure - Interpreter reuses the Scala language

Pluggable semantics - Constructs can be interpreted in multiple ways

Static safety - Static checking for syntactic and semantic errors

Runtime safety - Strict guarantees about generated program's behaviour

Extensibility - Should be easy to add new constructs / interpretations

Desirable properties [4,5]

Reuse of infrastructure - Interpreter reuses the Scala language

Pluggable semantics - Constructs can be interpreted in multiple ways

Static safety - Static checking for syntactic and semantic errors

Runtime safety - Strict guarantees about generated program's behaviour

Extensibility - Should be easy to add new constructs / interpretations

Expressiveness - Express constructs in a natural way to Rust programmers

Desirable properties [4,5]

Reuse of infrastructure - Interpreter reuses the Scala language
Pluggable semantics - Constructs can be interpreted in multiple ways
Static safety - Static checking for syntactic and semantic errors
Runtime safety - Strict guarantees about generated program's behaviour
Extensibility - Should be easy to add new constructs / interpretations
Expressiveness - Express constructs in a natural way to Rust programmers

Composability - Can be integrated with other interpreters (C,C++)

Desirable properties [4,5]

Reuse of infrastructure - Interpreter reuses the Scala language
Pluggable semantics - Constructs can be interpreted in multiple ways
Static safety - Static checking for syntactic and semantic errors
Runtime safety - Strict guarantees about generated program's behaviour
Extensibility - Should be easy to add new constructs / interpretations
Expressiveness - Express constructs in a natural way to Rust programmers

Composability - Can be integrated with other interpreters (C,C++)
Performance - Should use domain knowledge to produce optimal code

The Rust Programming Language [6]

Relatively new (released 2010)

Large active community

Zero-cost abstractions

C: Control/Performance > Safety
Java: Control/Performance < Safety
Rust: Control/Performance & Safety

Rust's features

Variables	References
Tuples	Mutability/Immutability
Functions	Trait objects
Scopes	Closures
Primitives	Modules
Type casting	Modifiers
Type inference	Attributes
Control flow	Type aliasing
Vectors	Associated types
Ownership	Dynamic sized types
Lifetimes	Operator overloading
Structs	Macros
Enums	Raw pointers
Pattern matching	**Unsafe operations**
Traits	Big standard library
Generics	Error handling

Rust's features

Variables	References
Tuples	Mutability/Immutability
Functions	Trait objects
Scopes	Closures
Primitives	Modules
Type casting	Modifiers
Type inference	Attributes
Control flow	Type aliasing
Vectors	Associated types
Ownership	Dynamic sized types
Lifetimes	Operator overloading
Structs	Macros
Enums	Raw pointers
Pattern matching	**Unsafe operations**
Traits	Big standard library
Generics	Error handling

Missing:
Classes

Rust's memory model

Manage memory by *ownership* rules at compile time

=> No run-time costs

=> No explicit allocations/deallocations

Rust's memory model

Manage memory by *ownership* rules at compile time

=> No run-time costs
=> No explicit allocations/deallocations

Ownership:

1. Each value has a variable that's called its *owner.*
2. There can only be one owner at a time.
3. When the owner goes out of scope, the value is dropped.

=> Prevents *double free* errors and *data races*
=> Enables more aggressive optimizations

Ownership

```
fn main() {
  let mut data = Vec::new();
  data.push(1);
  data.push(2);
  foo(data);
  data.push(9);
}

fn foo(data: Vec<i32>) {
  println!("{:?}", data);
}
```

Ownership

```
fn main() {
  let mut data = Vec::new();
  data.push(1);
  data.push(2);
  foo(data);
  data.push(9);
}

fn foo(data: Vec<i32>) {
  println!("{:?}", data);
}
```

Ownership

```
fn main() {
  let mut data = Vec::new();
  data.push(1);
  data.push(2);
  foo(data);
  data.push(9);
}

fn foo(data: Vec<i32>) {
  println!("{:?}", data);
}
```

Ownership

```
fn main() {
  let mut data = Vec::new();
  data.push(1);
  data.push(2);
  foo(data);
  data.push(9);
}

fn foo(data: Vec<i32>) {
  println!("{:?}", data);
}
```

Ownership

```
fn main() {
  let mut data = Vec::new();
  data.push(1);
  data.push(2);
  foo(data);
  data.push(9);
}

fn foo(data: Vec<i32>) {
  println!("{:?}", data);
}
```

Ownership

```
fn main() {
  let mut data = Vec::new();
  data.push(1);
  data.push(2);
  foo(data);
  data.push(9);
}

fn foo(data: Vec<i32>) {
  println!("{:?}", data);
}
```

Ownership

```
fn main() {
  let mut data = Vec::new(); // <-+ data lifetime starts here
  data.push(1);              //   |
  data.push(2);              //   |
  foo(data); // move         // <-+ foo takes ownership of data
  data.push(9);
}

fn foo(data: Vec<i32>) {
  println!("{:?}", data);
}
```

Ownership

```
fn main() {
  let mut data = Vec::new(); // <-+ data lifetime starts here
  data.push(1);              //   |
  data.push(2);              //   |
  foo(data); // move         // <-+ foo takes ownership of data
  data.push(9);
}

fn foo(data: Vec<i32>) {  // <-+
  println!("{:?}", data); //   |
}                         // <-+ data lifetime ends here (drop)
```

Ownership

```
fn main() {
  let mut data = Vec::new(); // <-+ data lifetime starts here
  data.push(1);              //   |
  data.push(2);              //   |
  foo(data); // move         // <-+ foo takes ownership of data
  data.push(9);
}

fn foo(data: Vec<i32>) {   // <-+
  println!("{:?}", data); //   |
}                          // <-+ data lifetime ends here (drop)
```

Ownership

```
fn main() {
  let mut data = Vec::new(); // <-+ data lifetime starts here
  data.push(1);              //   |
  data.push(2);              //   |
  foo(data); // move         // <-+ foo takes ownership of data
  data.push(9); // ERROR
} ^^^^ value used here after move

fn foo(data: Vec<i32>) {  // <-+
  println!("{:?}", data); //   |
}                         // <-+ data lifetime ends here (drop)
```

Borrowing

A variable can temporarily borrow ownership of a value

The variable gets a mutable or immutable reference to the value

A value can have either:

- N immutable references and 0 mutable
- 0 immutable references and 1 mutable

Borrowing

```
fn main() {
  let mut data = Vec::new();
  data.push(1);
  data.push(2);
  foo(data);
  data.push(9);
}

fn foo(temp: Vec<i32>) {
  //..
}
```

Borrowing

```
fn main() {
  let mut data = Vec::new();
  data.push(1);
  data.push(2);
  foo(&data);
  data.push(9);
}

fn foo(temp: &Vec<i32>) {
  //..
}
```

Borrowing

```
fn main() {
  let mut data = Vec::new(); // <-+ data lifetime starts here
  data.push(1);              //   |
  data.push(2);              //   |
  foo(&data);                //  ...
  data.push(9);
}

fn foo(temp: &Vec<i32>) {
  //..
}
```

Borrowing

```
fn main() {
  let mut data = Vec::new(); // <-+ data lifetime starts here
  data.push(1);              //   |
  data.push(2);              //   |
  foo(&data); // borrow      //  ...
  data.push(9);
}

fn foo(temp: &Vec<i32>) {
  //..
}
```

Borrowing

```
fn main() {
  let mut data = Vec::new(); // <-+ data lifetime starts here
  data.push(1);              //   |
  data.push(2);              //   |
  foo(&data); // borrow      //  ...
  data.push(9);
}

fn foo(temp: &Vec<i32>) { // <-+
  //..                    //   |
}                         // <-+ borrow ends here
```

Borrowing

```
fn main() {
  let mut data = Vec::new(); // <-+ data lifetime starts here
  data.push(1);              //   |
  data.push(2);              //   |
  foo(&data); // borrow      //   |
  data.push(9);              //   |
}                            // <-+ data lifetime ends here

fn foo(temp: &Vec<i32>) { // <-+
  //..                    //   |
}                         // <-+ borrow ends here
```

Borrowing

```
fn main() {
  let mut data = Vec::new(); // <-+ data lifetime starts here
  data.push(1);              //   |
  data.push(2);              //   |
  foo(&data); // borrow      //   |
  data.push(9); // OK        //   |
}                            // <-+ data lifetime ends here

fn foo(temp: &Vec<i32>) { // <-+
  //..                    //   |
}                         // <-+ borrow ends here
```

Lifetimes

Lexical lifetimes

- Current method for ownership (Rust 1.25)
- Lifetime ends when owner goes out of scope
- Based on abstract syntax tree (AST)

Lifetimes

Lexical lifetimes

- Current method for ownership (Rust 1.25)
- Lifetime ends when owner goes out of scope
- Based on abstract syntax tree (AST)

Non-lexical lifetimes

- Future method for ownership, less restrictive
- Lifetime ends when owner will no longer be used
- Based on control flow graph (CFG) liveness analysis

Lexical lifetimes

```
fn main() {
  let mut data = Vec::new();
  data.push(1);
  data.push(2);
  foo(&data);
  data.push(9);
}
```

Lexical lifetimes

```
fn main() {
  let mut data = Vec::new();   // <-+ data lifetime starts here
  data.push(1);                //   |
  data.push(2);                //   |
  foo(&data);          // borrow //   |
  data.push(9);                //   |
}                              // <-+ data lifetime ends here
```

Lexical lifetimes

```
fn main() {
  let mut data = Vec::new();   // <-+ data lifetime starts here
  data.push(1);                //   |
  data.push(2);                //   |
  let temp = &data; // borrow  //   |
  foo(temp);                   //   |
  data.push(9);                //   |
}                              // <-+ data lifetime ends here
```

Lexical lifetimes

```
fn main() {
  let mut data = Vec::new();   // <-+ data lifetime starts here
  data.push(1);                //   |
  data.push(2);                //   |
  let temp = &data; // borrow  // <-+-+ temp lifetime starts here
  foo(temp);                   //   | |
  data.push(9);                //   | |
}                              // <-+-+ both lifetimes end here
```

Lexical lifetimes

```
fn main() {
  let mut data = Vec::new();   // <-+ data lifetime starts here
  data.push(1);                //   |
  data.push(2);                //   |
  let temp = &data; // borrow  // <-+-+ temp lifetime starts here
  foo(temp);                   //   | |
  data.push(9);                //   | |
}                              // <-+-+ both lifetimes end here
```

Lexical lifetimes

```
fn main() {
  let mut data = Vec::new();   // <-+ data lifetime starts here
  data.push(1);                //   |
  data.push(2);                //   |
  let temp = &data; // borrow  // <-+-+ temp lifetime starts here
  foo(temp);                   //   | |
  data.push(9); // ERROR       //   | |
} ^^^^ cannot borrow 'data'    // <-+-+ both lifetimes end here
  as mutable because it is
  also borrowed as immutable
```

Lexical lifetimes

```
fn main() {
  let mut data = Vec::new();   // <-+ data lifetime starts here
  data.push(1);                //   |
  data.push(2);                //   |
  let temp = &data; // borrow  // <-+-+ temp lifetime starts here
  foo(temp);                   //   | |
  data.push(9); // ERROR       //  xxxxx
} ^^^^ cannot borrow 'data'    // <-+-+ both lifetimes end here
  as mutable because it is
  also borrowed as immutable
```

Lexical lifetimes

```
fn main() {
  let mut data = Vec::new();   // <-+ data lifetime starts here
  data.push(1);                //   |
  data.push(2);                //   |
  let temp = &data; // borrow  // <-+-+ temp lifetime starts here
  foo(temp); // copy           //   | |
  data.push(9); // ERROR       //   | |
}                              // <-+-+ both lifetimes end here
```

Lexical lifetimes

```
fn main() {
  let mut data = Vec::new();
  data.push(1);
  data.push(2);
  let temp = &data;
  foo(temp);
  data.push(9); // ERROR
}
```

Lexical lifetimes

```
fn main() {
  let mut data = Vec::new();
  data.push(1);
  data.push(2);
  {
    let temp = &data;
    foo(temp);
  }
  data.push(9); // OK
}
```

Lexical lifetimes

```
fn main() {
  let mut data = Vec::new(); // <-+ data lifetime starts here
  data.push(1);              //   |
  data.push(2);              //   |
  {                          //   |
    let temp = &data;        // <-+-+ temp lifetime starts here
    foo(temp);               //   | |
  }                          // <-+-+ temp lifetime ends here
  data.push(9); // OK        //   |
}                            // <-+ data lifetime ends here
```

Non-lexical lifetimes

```
fn main() {
  let mut data = Vec::new();
  data.push(1);
  data.push(2);
  let temp = &data;
  foo(temp);
  data.push(9);
}
```

Non-lexical lifetimes

```
fn main() {
  let mut data = Vec::new(); // <-+ data lifetime starts here
  data.push(1);              //   |
  data.push(2);              //   |
  let temp = &data;          // <-+-+ temp lifetime starts here
  foo(temp);                 // <-+-+ temp lifetime ends here
  data.push(9); // OK        //   |
}                            // <-+ data lifetime ends here
```

Aliasing

Ownership restricts aliasing

Aliasing it is still possible:

- Use *smart pointers* and *interior mutability*: Rc<RefCell<>>
- Use *unsafe operations*

Aliasing

Ownership restricts aliasing

Aliasing it is still possible:

- Use *smart pointers* and *interior mutability*: Rc<RefCell<>>
- Use *unsafe operations*

Unsafe operations allow dereferencing raw pointers (C-style pointers)

Raw pointers can point to the same data

Tools like RustBelt can verify soundness of unsafe operations

Rust's syntax

Expression statements

```
let result = if x > 0 { // OK
  x
} else {
  0
};
```

Rust's syntax

Expression statements

```
let result = if x > 0 { // OK
  x
} else {
  0
};
```

```
let result = loop { // OK
  // ...
  break x
};
```

Rust's syntax

Expression statements

```
let result = if x > 0 { // OK
  x
} else {
  0
};
```

```
let result = loop { // OK
  // ...
  break x
};
```

Declaration statements

```
let x = 3; // OK
let x = let y = 3; // ERROR
```

let declarations

Rust's syntax

Expression statements

```
let result = if x > 0 { // OK
  x
} else {
  0
};
```

```
let result = loop { // OK
  // ...
  break x
};
```

Declaration statements

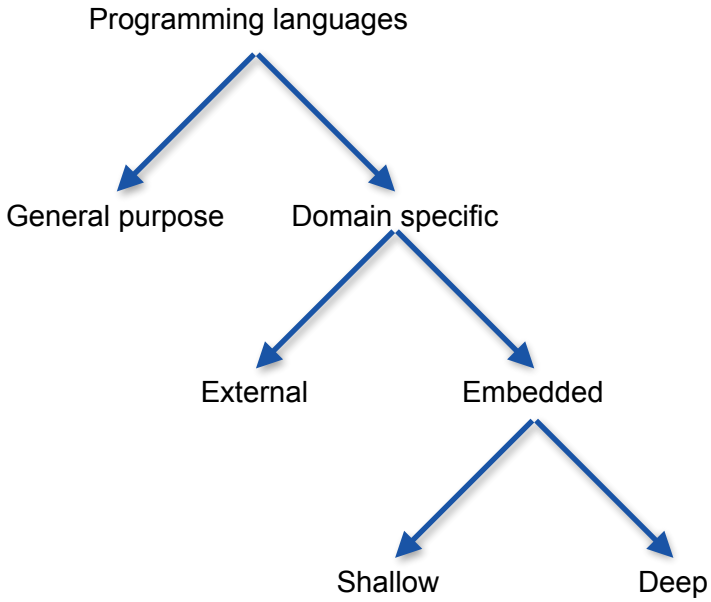
```
let x = 3; // OK
let x = let y = 3; // ERROR
```

let declarations

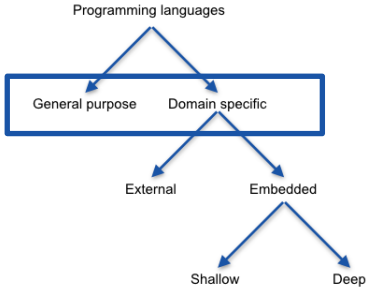
```
let x = struct MyStruct {
  // ..
}; // ERROR
```

item declarations
(struct, fn, enum)

Domain Specific Languages

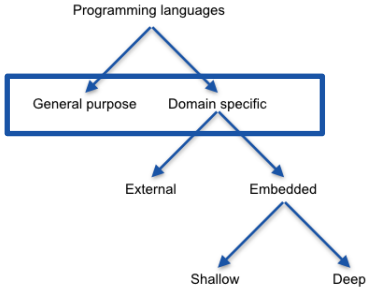


Programming Languages [7]



General Purpose Language (GPL) **Domain Specific Language (DSL)**

Programming Languages [7]



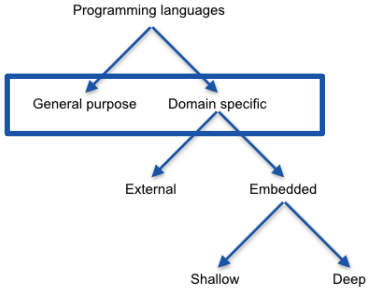
General Purpose Language (GPL)

- Turing complete

Domain Specific Language (DSL)

- Usually not Turing complete

Programming Languages [7]



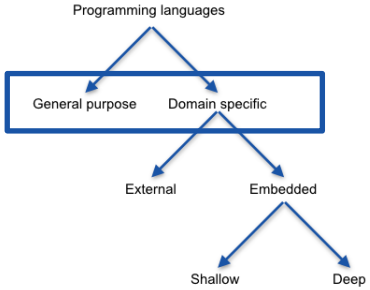
General Purpose Language (GPL)

- Turing complete
- Designed for a wide variety of problem domains

Domain Specific Language (DSL)

- Usually not Turing complete
- Designed for a specific problem domain

Programming Languages [7]



General Purpose Language (GPL)

- Turing complete
- Designed for a wide variety of problem domains

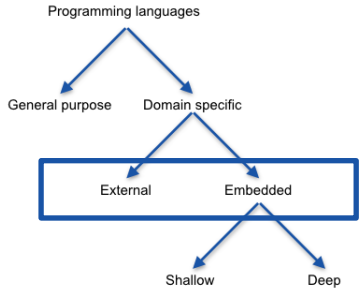
- Java, Scala, C, C++, Rust, Python, …

Domain Specific Language (DSL)

- Usually not Turing complete
- Designed for a specific problem domain

- Regex, CSS, VHDL, SQL, OpenGL, Shell, LaTeX …

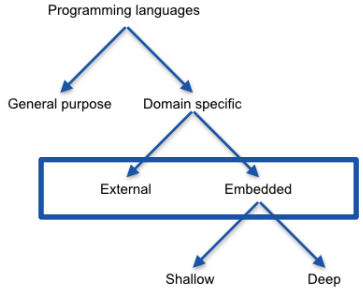
External and Embedded DSLs [8]



External DSLs

Embedded DSLs

External and Embedded DSLs [8]



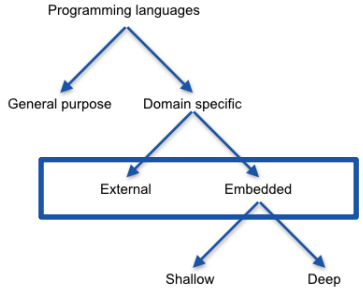
External DSLs

External compiler, editor, debugger

Embedded DSLs

Embedded into a host GPL as a library/framework

External and Embedded DSLs [8]



External DSLs

External compiler, editor, debugger

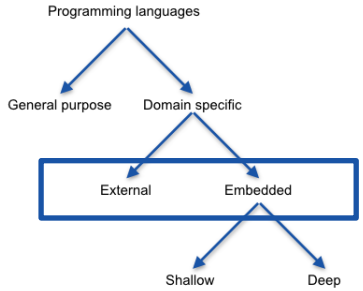
- Bigger work effort

Embedded DSLs

Embedded into a host GPL as a library/framework

+ Smaller work effort

External and Embedded DSLs [8]



External DSLs

External compiler, editor, debugger

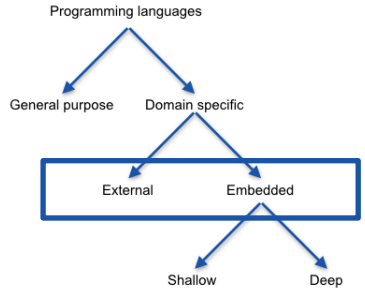
- Bigger work effort
- Difficult to combine with other DSLs

Embedded DSLs

Embedded into a host GPL as a library/framework

+ Smaller work effort
+ Can be combined with other DSLs

External and Embedded DSLs [8]



External DSLs

External compiler, editor, debugger

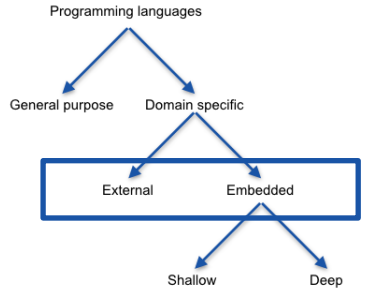
- Bigger work effort
- Difficult to combine with other DSLs
- Requires learning a new syntax

Embedded DSLs

Embedded into a host GPL as a library/framework

+ Smaller work effort
+ Can be combined with other DSLs
+ Reuses syntax of host GPL

External and Embedded DSLs [8]



External DSLs

External compiler, editor, debugger

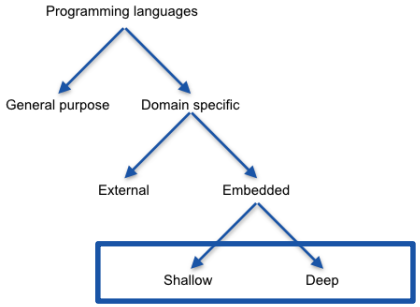
- Bigger work effort
- Difficult to combine with other DSLs
- Requires learning a new syntax
+ No limits

Embedded DSLs

Embedded into a host GPL as a library/framework

+ Smaller work effort
+ Can be combined with other DSLs
+ Reuses syntax of host GPL
- Limited by the host GPL

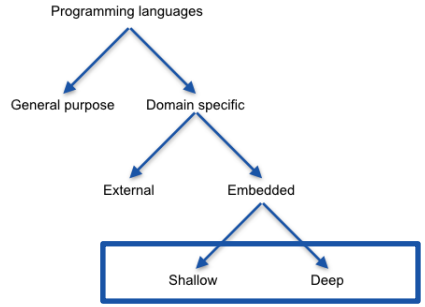
Shallow and Deep Embedding [9]



Shallow Embedding

Deep Embedding

Shallow and Deep Embedding [9]



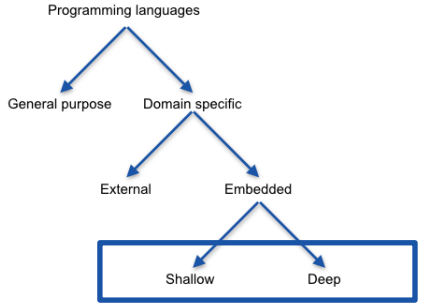
Shallow Embedding

Translate constructs to semantics, interpretation is fixed

Deep Embedding

Translate constructs to Abstract Syntax Tree (AST), multiple interpretations

Shallow and Deep Embedding [9]



Shallow Embedding

Translate constructs to semantics, interpretation is fixed

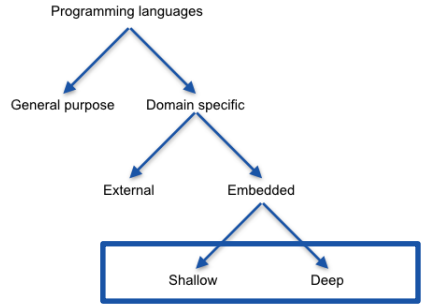
\+ Smaller work effort

Deep Embedding

Translate constructs to Abstract Syntax Tree (AST), multiple interpretations

\- Bigger work effort

Shallow and Deep Embedding [9]



Shallow Embedding

Translate constructs to semantics, interpretation is fixed

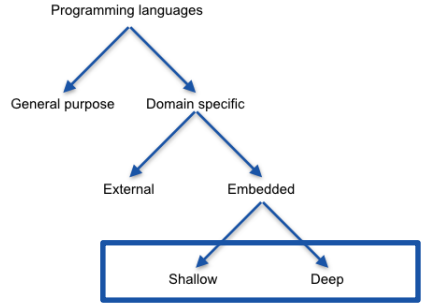
+ Smaller work effort
+ Flexible in adding new constructs

Deep Embedding

Translate constructs to Abstract Syntax Tree (AST), multiple interpretations

- Bigger work effort
- Inflexible in adding new constructs

Shallow and Deep Embedding [9]



Shallow Embedding

Translate constructs to semantics, interpretation is fixed

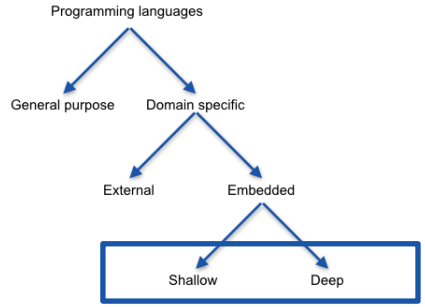
+ Smaller work effort
+ Flexible in adding new constructs
- Inflexible in adding new interpretations

Deep Embedding

Translate constructs to Abstract Syntax Tree (AST), multiple interpretations

- Bigger work effort
- Inflexible in adding new constructs
+ Flexible in adding new interpretations

Shallow and Deep Embedding [9]



Shallow Embedding

Translate constructs to semantics, interpretation is fixed

+ Smaller work effort
+ Flexible in adding new constructs
- Inflexible in adding new interpretations
- Less control

Deep Embedding

Translate constructs to Abstract Syntax Tree (AST), multiple interpretations

- Bigger work effort
- Inflexible in adding new constructs
+ Flexible in adding new interpretations
+ More control

Deep Embedding

Example DSL:

Constructs: add, lit
Interpretations: eval(), emit()

Deep Embedding

Example DSL:

Constructs: add, lit
Interpretations: eval(), emit()

Example program:

add(add(lit(1),lit(5)),lit(4))

Deep Embedding

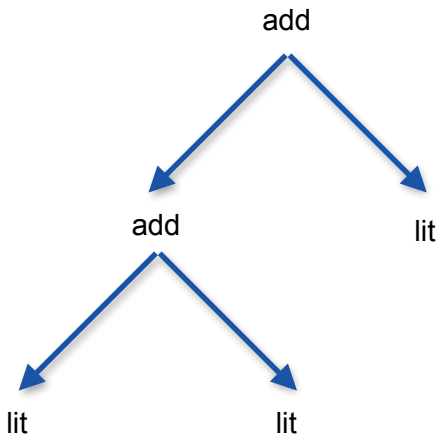
Example DSL:

Constructs: add, lit
Interpretations: eval(), emit()

Example program:

add(add(lit(1),lit(5)),lit(4))

AST



Deep Embedding

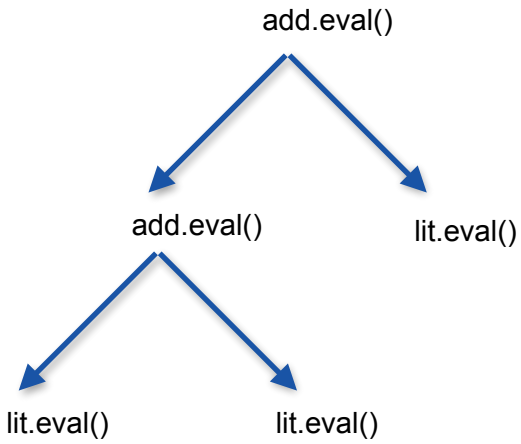
Example DSL:

Constructs: add, lit
Interpretations: eval(), emit()

Example program:

add(add(lit(1),lit(5)),lit(4))

AST



Deep Embedding

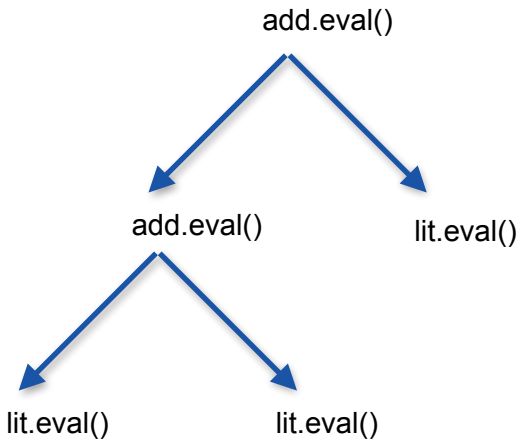
Example DSL:

Constructs: add, lit
Interpretations: eval(), emit()

Example program:

add(add(lit(1),lit(5)),lit(4)) => 10

AST



Deep Embedding

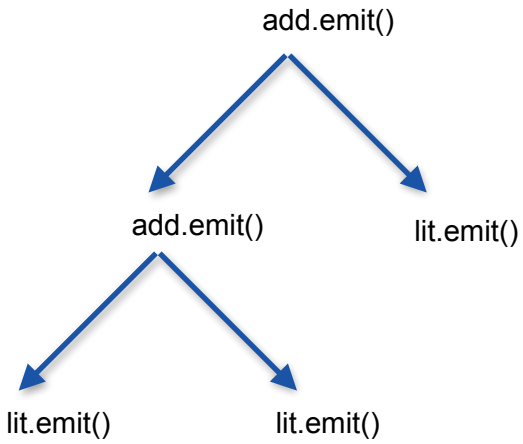
Example DSL:

Constructs: add, lit
Interpretations: eval(), emit()

Example program:

add(add(lit(1),lit(5)),lit(4))

AST

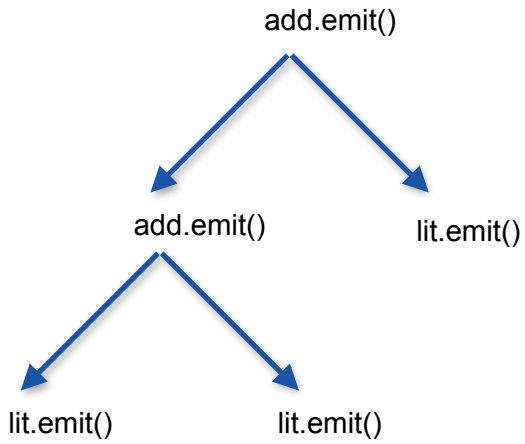


Deep Embedding

Example DSL:

Constructs: add, lit
Interpretations: eval(), emit()

AST



Example program:

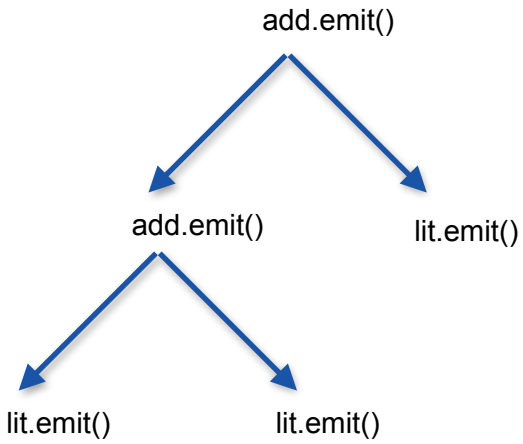
add(add(lit(1),lit(5)),lit(4)) => “(1 + 5) + 4)”

Expression problem [10]

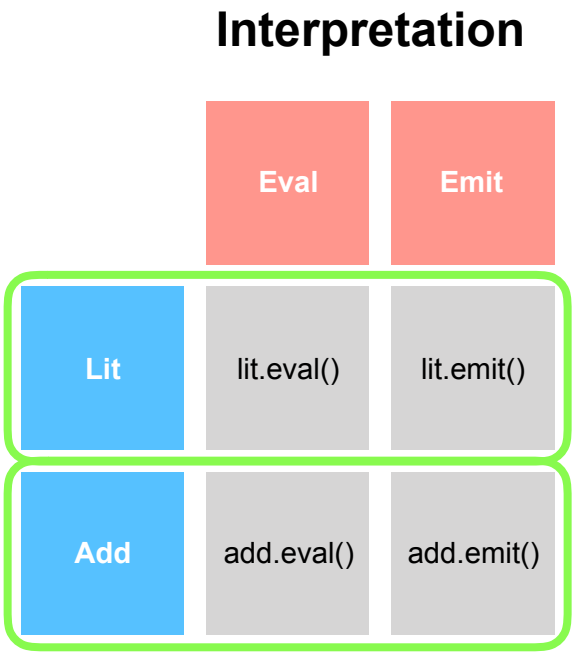
Constructs

Interpretation

	Eval	Emit
Lit	lit.eval()	lit.emit()
Add	add.eval()	add.emit()



Expression Problem: Object oriented solution



Adding new interpretations requires changing all constructs

Expression Problem: Object oriented solution

Interpretation

Constructs	Eval	Emit
Lit	lit.eval()	lit.emit()
Add	add.eval()	add.emit()

Adding new interpretations requires changing all constructs

```
/* -------- Module 1: Interface -------------- */
trait Interp {
  fn eval(&self) -> i32;
  fn emit(&self) -> String;
}
```

Expression Problem: Object oriented solution

Interpretation

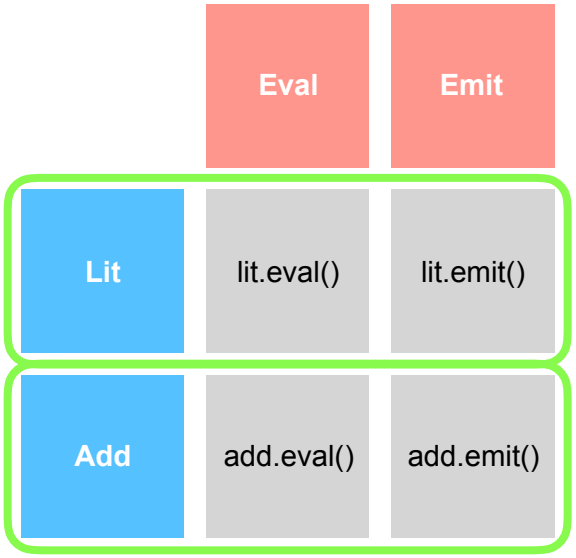
Constructs	Eval	Emit
Lit	lit.eval()	lit.emit()
Add	add.eval()	add.emit()

Adding new interpretations requires changing all constructs

```
/* -------- Module 1: Interface -------------- */
trait Interp {
  fn eval(&self) -> i32;
  fn emit(&self) -> String;
}
/* -------- Module 2: Lit -------------------- */
struct Lit {
  val: i32,
}

/* -------- Module 3: Add -------------------- */
struct Add<L:Interp,R:Interp> {
  lhs: L,
  rhs: R,
}
```

Expression Problem: Object oriented solution

Interpretation



Constructs

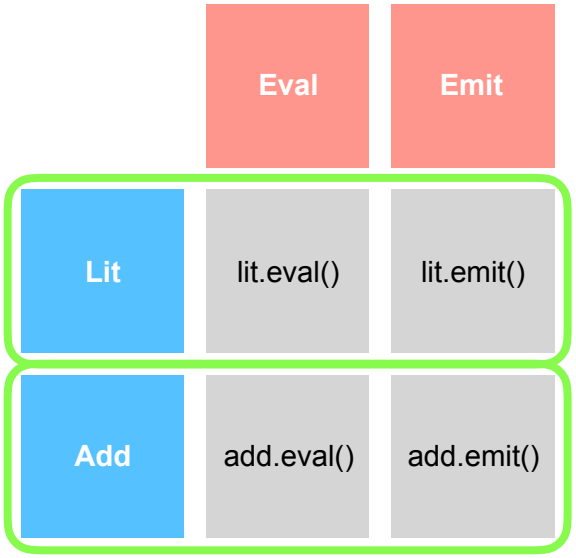
Adding new interpretations requires changing all constructs

```
/* -------- Module 1: Interface -------------- */
trait Interp {
  fn eval(&self) -> i32;
  fn emit(&self) -> String;
}
/* -------- Module 2: Lit -------------------- */
struct Lit {
  val: i32,
}
impl Interp for Lit {









}
/* -------- Module 3: Add -------------------- */
struct Add<L:Interp,R:Interp> {
  lhs: L,
  rhs: R,
}
impl<L:Interp,R:Interp> Interp for Add<L,R> {









}
```

Expression Problem: Object oriented solution

Interpretation

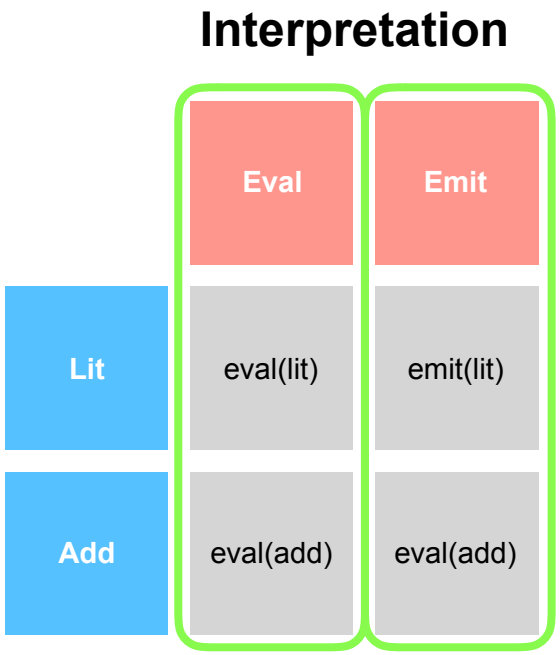
Constructs



Adding new interpretations requires changing all constructs

```
/* -------- Module 1: Interface -------------- */
trait Interp {
  fn eval(&self) -> i32;
  fn emit(&self) -> String;
}
/* -------- Module 2: Lit -------------------- */
struct Lit {
  val: i32,
}
impl Interp for Lit {
  fn eval(&self) -> i32 {
    self.val
  }
  fn emit(&self) -> String {
    self.val.to_string()
  }
}
/* -------- Module 3: Add -------------------- */
struct Add<L:Interp,R:Interp> {
  lhs: L,
  rhs: R,
}
impl<L:Interp,R:Interp> Interp for Add<L,R> {
  fn eval(&self) -> i32 {
    self.lhs.eval() + self.rhs.eval()
  }
  fn emit(&self) -> String {
    format!("{} + {}",
            self.lhs.emit(),
            self.rhs.emit())
  }
}
```

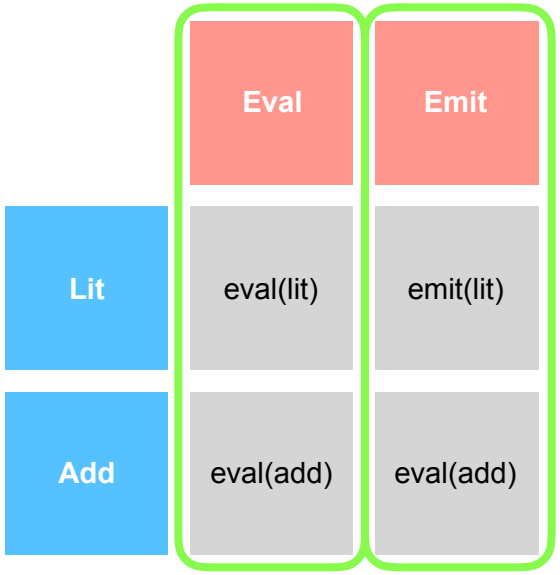
Expression Problem: Functional solution



Adding new constructs requires changing all interpretations

Expression Problem: Functional solution

Interpretation



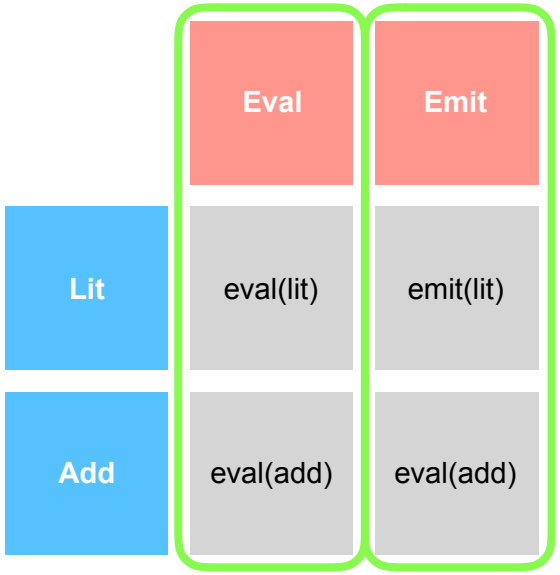
Constructs

Adding new constructs requires changing all interpretations

```
/* -------- Module 1: Types ------------------ */
enum Expr {
  Lit(Lit),
  Add(Add),
}
struct Lit {
  val: i32,
}
struct Add {
  lhs: Box<Expr>,
  rhs: Box<Expr>,
}
```

Expression Problem: Functional solution

Interpretation



Adding new constructs requires changing all interpretations

```
/* -------- Module 1: Types ------------------ */
enum Expr {
  Lit(Lit),
  Add(Add),
}
struct Lit {
  val: i32,
}
struct Add {
  lhs: Box<Expr>,
  rhs: Box<Expr>,
}
/* -------- Module 2: Eval ------------------- */
fn eval(node: Expr) -> i32 {




}
/* -------- Module 3: Emit ------------------- */
fn emit(node: Expr) -> String {




}
```

Expression Problem: Functional solution

Interpretation

Constructs	Eval	Emit
Lit	eval(lit)	emit(lit)
Add	eval(add)	eval(add)

Adding new constructs requires changing all interpretations

```
/* -------- Module 1: Types ------------------ */
enum Expr {
  Lit(Lit),
  Add(Add),
}
struct Lit {
  val: i32,
}
struct Add {
  lhs: Box<Expr>,
  rhs: Box<Expr>,
}
/* -------- Module 2: Eval ------------------- */
fn eval(node: Expr) -> i32 {
  match node {
    Expr::Lit(node) => node.val,
    Expr::Add(node) => eval(*node.lhs)
                     + eval(*node.rhs),
  }
}
/* -------- Module 3: Emit ------------------- */
fn emit(node: Expr) -> String {
  match node {
    Expr::Lit(node) => node.val.to_string(),
    Expr::Add(node) => format!("{} + {}",
                               emit(*node.lhs),
                               emit(*node.rhs)),
  }
}
```

Metaprogramming [11]

Programming technique which treats programs as data

A metaprogram can read, generate, analyse, or modify itself or other programs during runtime

Multistage programming (MSP)

A type of generative metaprogramming

Divide program into stages

Program in stage *i* generates program for stage *i+1*

Language Virtualization [5]

Embedded DSLs are decoupled from the GPL logic

DSL is unaware of GPL's control flow

Language virtualization: Give DSL more control by overloading constructs

Language Virtualization [5]

Embedded DSLs are decoupled from the GPL logic

DSL is unaware of GPL's control flow

Language virtualization: Give DSL more control by overloading constructs

```
while(a) {
  b
}
```

Language Virtualization [5]

Embedded DSLs are decoupled from the GPL logic

DSL is unaware of GPL's control flow

Language virtualization: Give DSL more control by overloading constructs

```
while(a) {
  b
}
```

```
_while(a,b)
```

https://github.com/TiarkRompf/scala-virtualized

The Scala Programming Language [12]

Released 2004

Scala = Scalable Language

Portable and high level

Less boilerplate than Java

Multiple parameter lists & By-name parameters

```
object MultiParameterLists {

  def While(cond: => Boolean)(body: => Unit): Unit = {
    if(cond) {
      body
      While(cond)(body)
    }
  }

}
```

Multiple parameter lists & By-name parameters

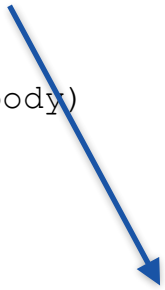
```
object MultiParameterLists {

  def While(cond: => Boolean)(body: => Unit): Unit = {
    if(cond) {
      body
      While(cond)(body)
    }
  }

}
```

Multiple parameter lists & By-name parameters

```
object MultiParameterLists {

  def While(cond: => Boolean)(body: => Unit): Unit = {
    if(cond) {
      body
      While(cond)(body)
    }
  }

}
```

Multiple parameter lists & By-name parameters

```
object MultiParameterLists {

  def While(cond: => Boolean)(body: => Unit): Unit = {
    if(cond) {
      body
      While(cond)(body)
    }
  }

}
```



Re-evaluate cond whenever it is used

Multiple parameter lists & By-name parameters

```
object MultiParameterLists {

  def While(cond: => Boolean)(body: => Unit): Unit = {
    if(cond) {
      body
      While(cond)(body)
    }
  }

}
```

Multiple parameter lists & By-name parameters

```
object MultiParameterLists {

  def While(cond: => Boolean)(body: => Unit): Unit = {
    if(cond) {
      body
      While(cond)(body)
    }
  }

  def main(args: Array[String]): Unit = {

    var i = 0

    While(i < 3) {
      println("Hello");
      i += 1
    }

  }
}
```

Implicit parameters

```
object ExplicitParameters {

  import scala.collection.mutable.ListBuffer

  def Let(value: Int, s: ListBuffer[Int]): Int = {
    s += value
    value
  }









}
```

Implicit parameters

```
object ExplicitParameters {

  import scala.collection.mutable.ListBuffer

  def Let(value: Int, s: ListBuffer[Int]): Int = {
    s += value
    value
  }

  def main(args: Array[String]): Unit = {

    val seq = new ListBuffer[Int]
    val x = Let(3, seq)
    val y = Let(3, seq)
    val z = Let(x+y, seq)
    println(seq) // Prints: ListBuffer(3, 3, 6)

  }
}
```

Implicit parameters

```
object ExplicitParameters {

  import scala.collection.mutable.ListBuffer

  def Let(value: Int, s: ListBuffer[Int]): Int = {
    s += value
    value
  }

  def main(args: Array[String]): Unit = {

    val seq = new ListBuffer[Int]
    val x = Let(3, seq)
    val y = Let(3, seq)
    val z = Let(x+y, seq)
    println(seq) // Prints: ListBuffer(3, 3, 6)

  }
}
```

Implicit parameters

```
object ImplicitParameters {

  import scala.collection.mutable.ListBuffer

  def Let(value: Int)(implicit s: ListBuffer[Int]): Int = {
    s += value
    value
  }

  def main(args: Array[String]): Unit = {

    implicit val seq = new ListBuffer[Int]
    val x = Let(3)
    val y = Let(3)
    val z = Let(x+y)
    println(seq) // Prints: ListBuffer(3, 3, 6)

  }
}
```

Implicit parameters

```
object ImplicitParameters {

  import scala.collection.mutable.ListBuffer

  def Let(value: Int)(implicit s: ListBuffer[Int]): Int = {
    s += value
    value
  }

  def main(args: Array[String]): Unit = {

    implicit val seq = new ListBuffer[Int]
    val x = Let(3)
    val y = Let(3)
    val z = Let(x+y)
    println(seq) // Prints: ListBuffer(3, 3, 6)

  }
}
```

Global AST object

```
object AST {
  // API
  def add(…) = {
    ...
  }
  ... // Data
}

object Test {

  def Let(value: Int): Int = {
    AST.add(...)
    value
  }

  def main(args: Array[String]): Unit = {
    val x = Let(3)
    val y = Let(3)
    val z = Let(x+y)
  }
)
```

Implicit classes

```
object ImplicitClasses {

  implicit class ImplicitInt(i: Int) {
    def print() {
      println(i)
    }
  }
  implicit class ImplicitString(i: String) {
    def print() {
      println(i)
    }
  }









}
```

Title Text

Implicit classes

```
object ImplicitClasses {

  implicit class ImplicitInt(i: Int) {
    def print() {
      println(i)
    }
  }
  implicit class ImplicitString(i: String) {
    def print() {
      println(i)
    }
  }

  def main(args: Array[String]): Unit = {

    3.print()       // Prints: 3
    "Hello".print() // Prints: Hello

  }
}
```

Other features

Macros - Static code generation, analysis, and typechecking

```
import scala.reflect.macros.Context
import scala.language.experimental.macros // Enable
```

Compiler Plugins - Complete control over compilation

```
import scala.tools.nsc
```

```
<plugin>
   <name>CDA-compiler-plugin</name>
   <classname>com.CDA.CompilerPlugin</classname>
</plugin>
```

Related work

Lightweight Modular Staging
Delite
Flare
Weld
Voodoo
ScyllaDB

Lightweight Modular Staging (LMS) [2]

Scala Multistage Programming library

Matrix - Matrix type in generator code
Rep[Matrix] - Matrix type in generated code, **Rep** overloads Matrix to emit code

Matrix is abstract, could be generated as Sparse or Dense

Generates an IR which is then optimized

- Common subexpression elimination
- Dead code elimination
- Partial evaluation
- Domain specific optimizations

Delite [13]

Compiler framework and runtime for high-performance DSLs

Built on top of LMS

Framework provides:

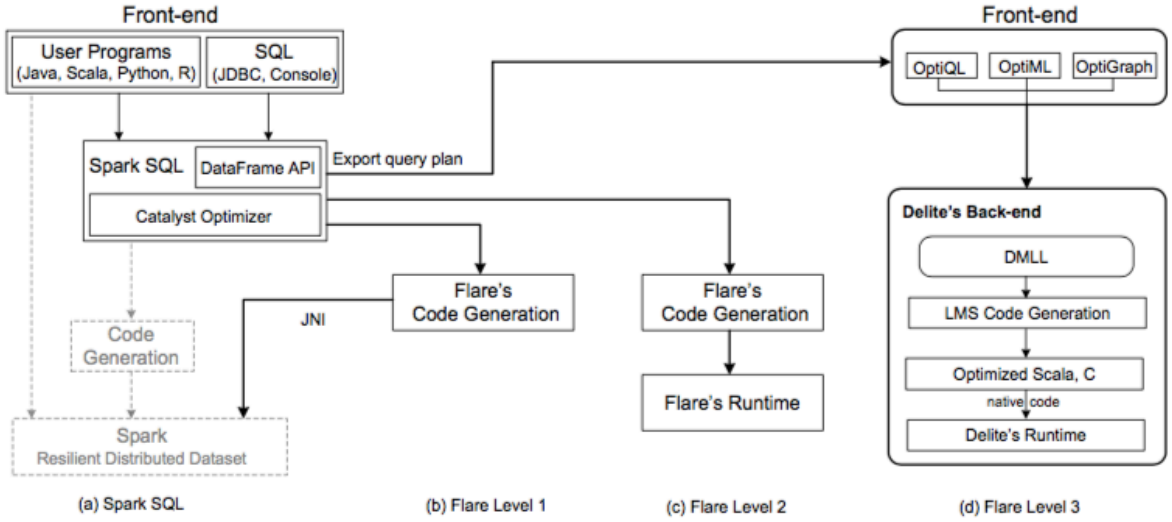
- Generic IR: General DSL language constructs
- Parallel IR: Parallel constructs (Map, Reduce, Zip, MultiLoop)
- Domain-specific IR: Custom construct extensions

Runtime:

- Builds execution graph
- Generates code for CPU and GPU
- Schedules work among machines

Flare [3]

Bypasses inefficient abstraction layers of Spark
Level 1. Generate code for stages in queries with LMS
Level 2. Generate code for entire queries with LMS
Level 3. Translate queries to Delite, and generate code for UDFs by injecting LMS



Weld [14]

Problem: Successive calls to different libraries, e.g., NumPy and Tensorflow, require materialization of intermediate results

=> Big data movement cost

Solution: Bridge gap between libraries through common IR, and lazily-evaluated runtime API

1. Libraries submit their IR code to the Weld runtime
2. Weld dynamically compiles, optimizes, and executes code fragments

Voodoo [15]

High performance external DSL

Applied as a backend to MonetDB

Vector oriented: Data is stored in vector-format for high parallelism

Assigns resources for each code segment based on its Extent and Intent

- Extent: Degree of parallelism
- Intent: Degree of sequential iterations per parallel work unit

Title Text

ScyllaDB [16]

C++ rewrite of Cassandra

Better memory utilization - dynamically sized row cache

Summary

Rust

- Support checking for non-lexical lifetimes?
- Support statement expressions & statement declarations

DSL

- Probably use deep embedding DSL
- Borrow ideas from LMS

Scala

- Singleton AST or not?
- Use Scala Virtualized?

References

[1] Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S. and Tzoumas, K. (2015) 'Apache flink: Stream and batch processing in a single engine', Bulletin of the IEEE Computer Society Technical Committee on Data Engineering. IEEE Computer Society, 36(4).

[2] https://vjovanov.github.io/dsldi-summer-school/materials/tuesday/07-summer-school.pdf (https://www.youtube.com/watch?v=16A1yemmx-w&t=261s)

[3] Essertel, G. M., Tahboub, R. Y., Decker, J. M., Brown, K. J., Olukotun, K. and Rompf, T. (2017) 'Flare: Native compilation for heterogeneous workloads in apache spark', *arXiv preprint arXiv:1703.08219*.

[4] Hofer, C., Ostermann, K., Rendel, T. and Moors, A. (2008) 'Polymorphic embedding of dsls', in Proceedings of the 7th international conference on generative programming and component engineering. ACM, pp. 137–148.

[5] Moors, A., Rompf, T., Haller, P. and Odersky, M. (2012) 'Scala-virtualized', in Proceedings of the acm sigplan 2012 workshop on partial evaluation and program manipulation. ACM, pp. 117–120.

[6] The rust programming language (2018). The Rust Community. Available at: https://doc.rust-lang. org/book/second-edition/ (Accessed: 31 January 2018).

[7] Brady. E, (2015). University of St Andrews, [online] Available at: https://www.idris-lang.org/SSGEP/lecture2.pdf [Accessed 2018-2-19].

[8] Gibbons, J. and Wu, N. (2014) 'Folding domain-specific languages: Deep and shallow embeddings (functional pearl)', in ACM sigplan notices. ACM (9), pp. 339–347.

[9] Svenningsson, J. and Axelsson, E. (2012) 'Combining deep and shallow embedding for edsl', in International symposium on trends in functional programming. Springer, pp. 21–36.

References

[10] Manzyuk, O. (2014) From object algebras to finally tagless interpreters. Available at: https:// oleksandrmanzyuk.wordpress.com/ 2014/06/18/from-object-algebras-to-finally-tagless-interpreters-2/ (Accessed: 19 February 2018).

[11] Rompf, T. and Odersky, M. (2010) 'Lightweight modular staging: A pragmatic approach to runtime code generation and compiled dsls', in *Acm sigplan notices*. ACM (2), pp. 127–136.

[12] Swartz, J. (2014) Learning scala: Practical functional programming for the jvm. " O'Reilly Media, Inc.".

[13] Brown, K. J., Sujeeth, A. K., Lee, H. J., Rompf, T., Chafi, H., Odersky, M. and Olukotun, K. (2011) 'A heterogeneous parallel framework for domain-specific languages', in Parallel architectures and compilation techniques (pact), 2011 international conference on. IEEE, pp. 89–100.

[14] Palkar, S., Thomas, J., Narayanan, D., Shanbhag, A., Palamuttam, R., Pirk, H., Schwarzkopf, M., Amarasinghe, S., Madden, S. and Zaharia, M. (2017) 'Weld: Rethinking the interface between data-intensive applications', *arXiv preprint arXiv:1709.06416*.

[15] Pirk, H., Moll, O., Zaharia, M. and Madden, S. (2016) 'Voodoo-a vector algebra for portable database performance on modern hardware', *Proceedings of the VLDB Endowment*. VLDB Endowment, 9(14), pp. 1707–1718.

[16] Mahgoub, A., Ganesh, S., Meyer, F., Grama, A. and Chaterji, S. (2017) 'Suitability of nosql systems—cassandra and scylladb—for iot workloads', in Communication systems and networks (comsnets), 2017 9th international conference on. IEEE, pp. 476–479.