Implement SQL parser and type system in Databend

Databend Cloud is an affordable cloud **data warehouse** developed in Rust. In this exploration, we will dive into Databend's internal, specifically examining its SQL parser and type system that are meticulously crafted for Rust.

by **@AndyLok**

Who am I

骆迪安

https://github.com/andylokandy
https://twitter.com/AndylokandyLok

Databend Planner Team
Distributed transaction

Rust contributor
Idris-lang contributor

Databend

https://databend.rs

Feature-Rich

Instant Elasticity

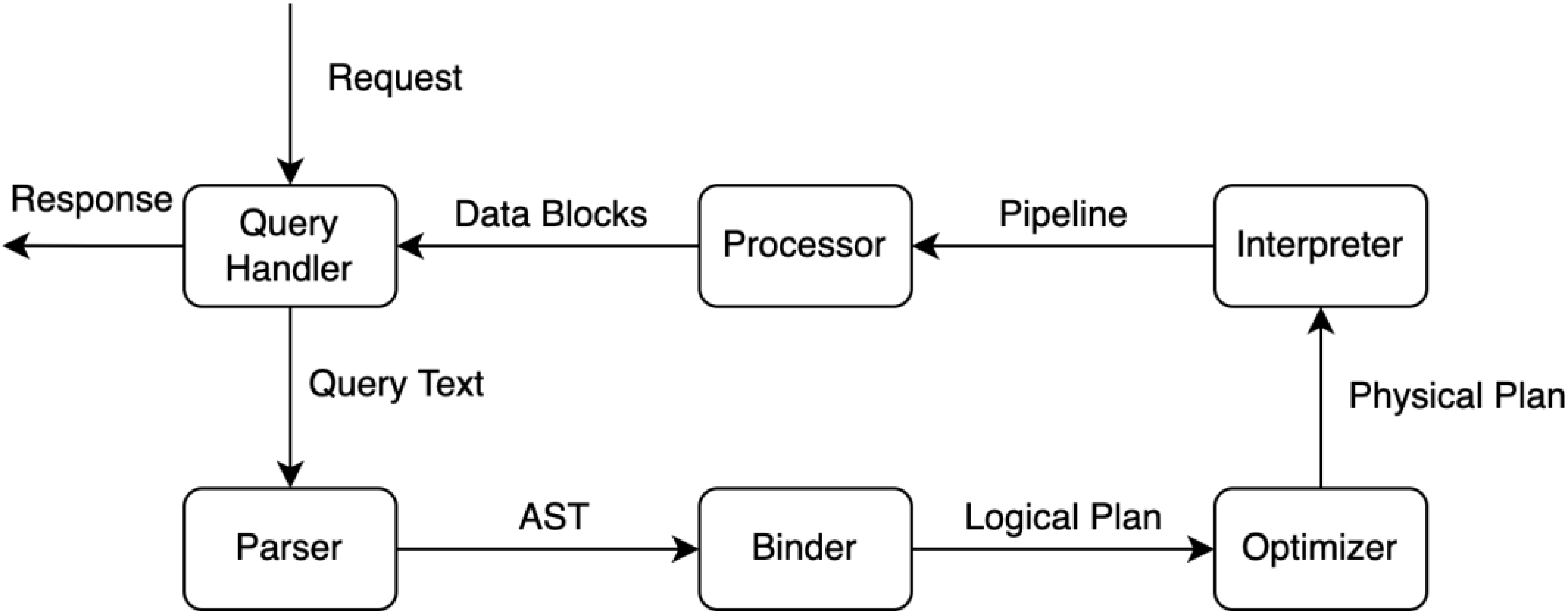
Support for Semi-Structured Data

MySQL/ClickHouse Compatible

Low Cost

Easy To Use

Cloud Native (S3, Azure Blob, Google Cloud Storage, Alibaba Cloud OSS, etc)



Request
Response
Query Handler
Data Blocks
Processor
Pipeline
Interpreter
Query Text
Physical Plan
Parser
AST
Binder
Logical Plan
Optimizer

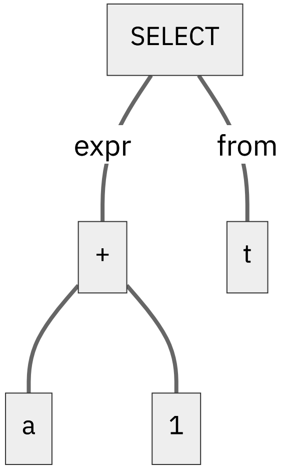
SQL Parser

Check for syntax violation and construct **Abstract Syntax Tree (AST)**

Input

```
SELECT a + 1 FROM t
```

Output



Tokenize

Convert **String** to **Token Stream**

Parse

Convert **Token Stream** to **AST**

Tokenize

Recognise **tokens** by regular expression

Regular expression for each token kind

```
Ident = [_a-zA-Z][_$a-zA-Z0-9]*
Number = [0-9]+
Plus = \+
SELECT = SELECT
FROM = FROM
```

Tokenize

Recognise **tokens** by regular expression

Input

```
SELECT a + 1 FROM t
```

Output

```
[
    Keyword(SELECT),
    Ident(a),
    BinOp(+),
    Number(1),
    Keyword(FROM),
    Ident(t),
]
```

Parse

Define **gramma rule** and construct **AST**

Backus–Naur form (BNF) grammars

```
<select_statement> ::= SELECT <expr> FROM <ident>
<expr> ::= <number>
         | <ident>
         | <expr> <op> <expr>
<op> ::= + | - | * | /
```

Parse

Define **gramma rule** and construct **AST**

Input

```
[
    Keyword(SELECT),
    Ident(a),
    BinOp(+),
    Number(1),
    Keyword(FROM),
    Ident(t),
]
```

Output

```
SelectStatement {
    projection: Expr::BinaryOp {
        op: Op::Plus,
        args: [
            Expr::ColumnRef("a"),
            Expr::Constant(Scalar::Int(1))
        ]
    }
    from: "t",
}
```

Choosing SQL Parser

sqlparser-rs

ANTLR4

LALRPOP

nom

Tokenizer

https://github.com/maciejhirsz/logos

```
#[derive(Logos)]
pub enum TokenKind {
    #[regex(r"[ \t\r\n\f]+", logos::skip)]
    Whitespace,

    #[regex(r#"[_a-zA-Z][_$a-zA-Z0-9]*"#)]
    Ident,
    #[regex(r"[0-9]+")]
    Number,

    #[token("+")]
    Plus,

    #[token("SELECT", ignore(ascii_case))]
    SELECT,
    #[token("FROM", ignore(ascii_case))]
    FROM,
}
```

Tokenizer

https://github.com/maciejhirsz/logos

Input

```sql
SELECT a + 1 FROM t
```

Output

```
[
    Token { kind: TokenKind::Select, text: "SELECT", span: 0..6 },
    Token { kind: TokenKind::Ident, text: "a", span: 7..8 },
    Token { kind: TokenKind::Plus, text: "+", span: 9..10 },
    Token { kind: TokenKind::Number, text: "1", span: 11..12 },
    Token { kind: TokenKind::From, text: "from", span: 13..17 },
    Token { kind: TokenKind::Ident, text: "t", span: 18..19 },
]
```

Tokenizer

https://github.com/maciejhirsz/logos

Input

```
SELECT a + 1 FROM t
```

Output

```
[
    Token { kind: TokenKind::Select, text: "SELECT", span: 0..6 },
    Token { kind: TokenKind::Ident, text: "a", span: 7..8 },
    Token { kind: TokenKind::Plus, text: "+", span: 9..10 },
    Token { kind: TokenKind::Number, text: "1", span: 11..12 },
    Token { kind: TokenKind::From, text: "from", span: 13..17 },
    Token { kind: TokenKind::Ident, text: "t", span: 18..19 },
]
```

Span

```
SELECT  a    +     1     FROM   t
0..6    7..8 9..10 11..12 13..17 18..19
```

Error Report

Pretty print the error report thanks to the **span** information

```
error:
  --> SQL:1:19
   |
1  | create table a (c varch)
   | ------          - ^^^^^ expected `BOOLEAN`, `BOOL`, `UINT8`, `TINYINT`, `UINT16`, `SMALLINT`, or 33 more ...
   | |               |
   | |               while parsing `<column name> <type> [DEFAULT <default value>] [COMMENT '<comment>']`
   | while parsing `CREATE TABLE [IF NOT EXISTS] [<database>.]<table> [<source>] [<table_options>]`
```

Parser

https://github.com/rust-bakery/nom

Terminal

Recognize only one **token** from **token stream**

Combinator

Combine **terminals** and other small parsers into a larger parser

Terminal

Recognize only one **token** from **token stream**

Recongnize a token that has the exactly **text**

```
fn match_text(text: &str)
    -> impl FnMut(&[Token]) -> IResult<&[Token], Token>
{
    satisfy(|token: &Token| token.text == text)(i)
}
```

Recongnize a token that is of the **token kind**

```
fn match_token(kind: TokenKind)
    -> impl FnMut(&[Token]) -> IResult<&[Token], Token>
{
    satisfy(|token: &Token| token.kind == kind)(i)
}
```

Combinator

Combine **terminals** and other small parsers into a larger parser

tuple(a, b, c)

alt(a, b, c)

many0(a)

many1(a)

opt(a)

BNF

Formally defined **gramma rule**

```
<select_statement> ::=
    SELECT <expr> FROM <ident>
```

Code

Practical Rust code using **nom**

```
tuple((
    match_token(SELECT),
    expr,
    match_token(FROM),
    match_token(Ident),
))
```

BNF

Formally defined **gramma rule**

```
<select_statement> ::=
    SELECT <expr> [FROM <ident>]
```

Code

Practical Rust code using **nom**

```
tuple((
    match_token(SELECT),
    expr,
    opt(tuple((
        match_token(FROM),
        match_token(Ident),
    )))
))
```

BNF

Formally defined **gramma rule**

```
<select_statement> ::=
    SELECT <expr>+ [FROM <ident>]
```

Code

Practical Rust code using **nom**

```
tuple((
    match_token(SELECT),
    many1(expr),
    opt(tuple((
        match_token(FROM),
        match_token(Ident),
    )))
))
```

BNF

Formally defined **gramma rule**

```
<select_statement> ::=
    SELECT (<expr> AS <ident> | <expr>)+
    [FROM <ident>]
```

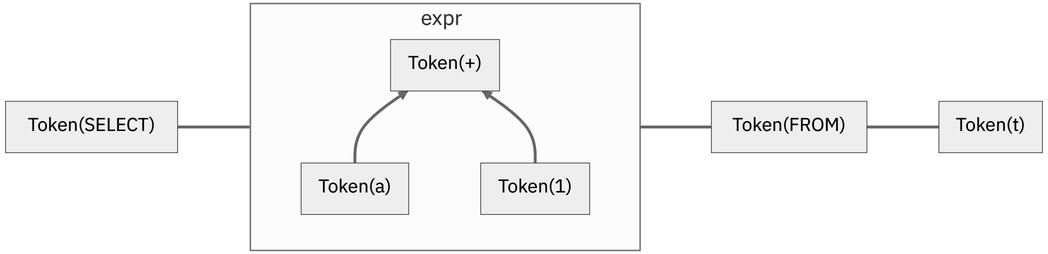
Code

Practical Rust code using **nom**

```
tuple((
    match_token(SELECT),
    many1(alt((
        tuple((
            expr,
            match_token(AS),
            expr,
        )),
        expr,
    ))),
    opt(tuple((
        match_token(FROM),
        match_token(Ident),
    )))
))
```

Parse Tree

The **parse tree** of `SELECT a + 1`



AST

The **AST** of `SELECT a + 1`

```
SelectStatement {
    projection: Expr::BinaryOp {
        op: Op::Plus,
        args: [
            Expr::ColumnRef("a"),
            Expr::Constant(Scalar::Int(1))
        ]
    }
    from: "t",
}
```

map( )

Use **map( )** to convert **Parse Tree** into **AST**

```
fn select_statement(input: &[Token])
    -> IResult<&[Token], SelectStatement>
{
    map(
        tuple((
            match_token(SELECT),
            many1(alt((
                tuple((
                    expr,
                    match_token(AS),
                    expr,
                )),
                expr,
            ))),
            opt(tuple((
                match_token(FROM),
                match_token(Ident),
            )))
        )),
        |(_, projections, _, opt_from)| SelectStatement {
            projections: projections
                .map(|(expr, _, alias)| {
                    Projection::Aliased(expr, alias)
                })
                .collect(),
            from: opt_from.map(|(_, from)| from),
        }
    )(input)
}
```

nom-rule

https://github.com/andylokandy/nom-rule
Simplify nom parser using **BNF**-like gramma

Syntax definition using **nom-rule**

```
nom_rule! {
    SELECT
    ~ (#expr ~ AS ~ Ident | #expr)+
    ~ (FROM ~ Ident)?
}
```

Generated **nom** parser

```
tuple((
    match_token(SELECT),
    many1(alt((
        tuple((
            expr,
            match_token(AS),
            expr,
        )),
        expr,
    ))),
    opt(tuple((
        match_token(FROM),
        match_token(Ident),
    )))
))
```

nom-rule

https://github.com/andylokandy/nom-rule
Simplify nom parser using **BNF**-like gramma

Update the example using **nom-rule**

```
fn select_statement(input: &[Token])
    -> IResult<&[Token], SelectStatement>
{
    map(
        nom_rule! {
            SELECT
            ~ (#expr ~ AS ~ Ident | #expr)+
            ~ (FROM ~ Ident)?
        },
        |(_, projections, _, opt_from)| SelectStatement {
            projections: projections
                .map(|(expr, _, alias)| {
                    Projection::Aliased(expr, alias)
                })
                .collect(),
            from: opt_from.map(|(_, from)| from),
        }
    )(input)
}
```

nom-rule

https://github.com/andylokandy/nom-rule
Simplify nom parser using **BNF**-like gramma

nom-rule cheatsheet

nom-rule	Translated
TOKEN	match_token(TOKEN)
"+"	match_text("+")
a ~ b ~ c	tuple((a, b, c))
(...)*	many0(...)
(...)+	many1(...)
(...)?	opt(...)
a | b | c	alt((a, b, c))

Left Recursion

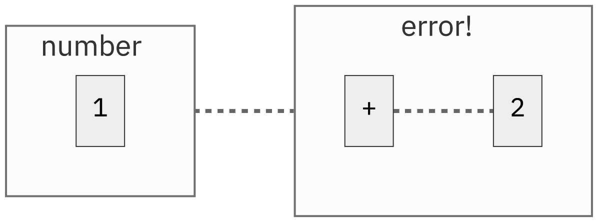
A classical **problem** every parser will meet when parsing expression

Try to define the syntax for expression like `1 + 2`

```
<expr> ::= <number>
         | <expr> + <expr>
```

---

oops! The second rule will **never apply**



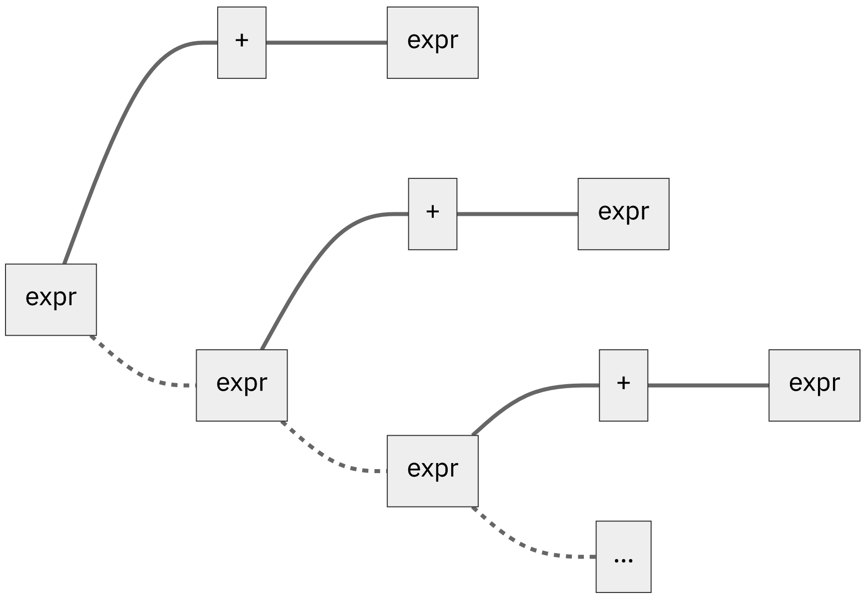
Left Recursion

A classical **problem** every parser will meet when parsing expression

Try again! **Reverse the order**!

```
<expr> ::= <expr> + <expr>
         | <number>
```

but not gonna work... it'll **never stop**



Pratt Parser

https://github.com/segeljakt/pratt/
A decent solution to the problem of **Left Recursioin**

Instead of parsing to a tree, parse the flatten elements

```
<expr_element> ::= + | <number>
```

Pratt Parser

https://github.com/segeljakt/pratt/
A decent solution to the problem of **Left Recursioin**

Construct the AST using **Pratt Parser**

```rust
use pratt::PrattParser;

impl<I: Iterator<Item = ExprElement> PrattParser<I> for ExprParser {
    type Output = Expr;

    fn query(&mut self, elem: &ExprElement> -> Affix {
        match elem {
            ExprElement::Plus =>
                Affix::Infix(Precedence(20), Associativity::Left),
            ExprElement::Number(_) => Affix::Nilfix,
        }
    }

    fn primary(&mut self, elem: ExprElement) -> Expr {
        match elem {
            ExprElement::Number(n) => Expr::Number(n),
            _ => unreachable!(),
        }
    }

    fn infix(
        &mut self,
        lhs: Expr,
        elem: ExprElement,
        rhs: Expr,
    ) -> Expr {
        match elem {
            ExprElement::Plus(n) => Expr::Plus(lhs, rhs),
            _ => unreachable!(),
        }
    }
}
```

Type Check

Validate the **sematic** of the SQL

1. Given an AST

```
1 + 'foo'
```

2. **Desugar** AST into function calls

```
plus(1, 'foo')
```

3. Infer type of **literals**

```
1 :: Int8
'foo' :: String
```

Type Check

Validate the **sematic** of the SQL

4. So we desire to find a function overload tha matches

```
plus(Int8, String)
```

---

5. Query function overloads for `plus()`

```
plus(Int8, Int8) -> Int8
plus(Int16, Int16) -> Int16
plus(Timestamp, Timestamp) -> Timestamp
plus(Date, Date) -> Date
```

---

6. However, no matching overload is found, thus typechecker reports a **type error**

```
1 + 'foo'
  ^ function `plus` has no overload for parameters `(Int8, String)`

  available overloads:
    plus(Int8, Int8) -> Int8
    plus(Int16, Int16) -> Int16
    plus(Timestamp, Timestamp) -> Timestamp
    plus(Date, Date) -> Date
```

Type Judgement

Formal rules to **prove type** of a expression

1. Rule to assume the type of **boolean** literal

```
⊢ TRUE : Boolean
```

---

2. Also, the same for **numbers** and **string**

```
⊢ 1 : Int8
⊢ "foo" : String
```

---

3. Rule to prove the type of **function call**

```
Γ ⊢ e1 : Int8      Γ ⊢ e2 : Int8
--------------------------------
    Γ ⊢ plus(e1, e2) : Int8
```

```
*Γ : Type Environment
```

Type Environment

The meaning of the mystery `Γ`

What is the type of variable `a` ?

```sql
SELECT 1 + a
```

---

a. determined by querying the **table metadata**

```sql
SELECT 1 + a from t
```

---

b. determined by type checking the **subquery**

```sql
SELECT 1 + a from (
    SELECT number as a from numbers(100)
)
```

Subtype

1. Given an AST

```
1 + 256
```

---

2. **Desugar** AST to funtion call

```
plus(Int8, Int16)
```

---

3. oops! There is no matching overloads

```
plus(Int8, Int8) -> Int8
plus(Int16, Int16) -> Int16
```

Subtype

4. Because `Int8` is subtype of `Int16`, `Int8` can be cast to `Int16`

```
Int8 <: Int16
```

---

5. Update the type rule of function call to accept **subtype**

```
Γ ⊢ e1 : T1      Γ ⊢ e2 : T2      T1 <: In16      T2 <: Int16
-----------------------------------------------------------
                 Γ ⊢ plus(e1, e2) : Int16
```

---

6. Finally we prove

```
⊢ plus(1, 256) : Int16
```

Generic

1. Type of **array**

```
⊢ [1, 2, 3] : Array<Int8>
```

---

2. oops! It's impossible to enumerate all possible types

```
get(Array<Int8>, Int64) -> Int8
get(Array<Array<Int8>>, Int64) -> Array<Int8>
get(Array<Array<Array<Int8>>>, Int64) -> Array<Array<Int8>>
...
```

---

3. Use **generic** in the function signature

```
get<T>(Array<T>, Int64) -> T
```

Generic

4. Assume we are checking this expression

```
get([1, 2, 3], 0)
```

---

5. Type check its arguments first

```
get(Array<Int8>, Int8)
```

---

6. We have got the **substitution** after **unification**, then apply the substitution to the signature

```
{T=Int8}
// then apply the substitution to the signature
get<Int8>(Array<Int8>, Int64) -> Int8
```

---

7. Finally we prove

```
⊢ get([1, 2, 3], 0) : Int8
```

Evaluation

Typechecker produces expression that is ready to be evaluated

```
pub enum Expr {
    Constant {
        scalar: Scalar,
        data_type: DataType,
    },
    ColumnRef {
        id: String,
        data_type: DataType,
    },
    Cast {
        is_try: bool,
        expr: Box<Expr>,
        dest_type: DataType,
    },
    FunctionCall {
        name: String,
        args: Vec<Expr>,
        return_type: DataType,
        eval: Box<Fn([Value]) -> Value>,
    },
}
```

Evaluation

Expression of `1 + a`

```
Expr::FunctionCall {
    name: "plus",
    args: [
        Expr::Constant {
            scalar: Scalar::Int8(1),
            data_type: DataType::int8,
        },
        Expr::ColumnRef {
            id: "a",
            data_type: DataType::Int8,
        },
    ],
    return_type: DataType::Int8,
    eval: Box<Fn([Value]) -> Value>,
}
```

Evaluation

The place where actual work takes place

```
fn plus_eval(args: &[Value]) -> Value {
    ...
}
```

Vectorized input which can either be a **column** or **scalar** (all values are same)

```
enum Value {
    Scalar(Scalar),
    Column(Column),
}

enum Scalar {
    Int8(i8),
    Int16(i16),
    Boolean(bool),
    ...
}

enum Column {
    Int8(Vec<i8>),
    Int16(Vec<i16>),
    Boolean(Vec<bool>),
    ...
}
```

Evaluation

A working example for `plus(Int8, Int8) -> Int8`

```
fn plus_eval(args: &[Value]) -> Value {
    match (&args[0], &args[1]) {
        (Value::Scalar(Scalar::Int8(lhs)), Value::Scalar(Scalar::Int8(rhs))) => {
            Value::Scalar(Scalar::Int8(lhs + rhs))
        },
        (Value::Column(Column::Int8(lhs)), Value::Scalar(Scalar::Int8(rhs))) => {
            let result: Vec<i8> = lhs
                .iter()
                .map(|lhs| *lhs + rhs)
                .collect();
            Value::Column(Column::Int8(result))
        },
        (Value::Scalar(Scalar::Int8(lhs)), Value::Column(Column::Int8(rhs))) => {
            let result: Vec<i8> = rhs
                .iter()
                .map(|rhs| lhs + *rhs)
                .collect();
            Value::Column(Column::Int8(result))
        },
        (Value::Column(Column::Int8(lhs)), Value::Column(Column::Int8(rhs))) => {
            let result: Vec<i8> = lhs
                .iter()
                .zip(rhs.iter())
                .map(|(lhs, rhs)| *lhs + *rhs)
                .collect();
            Value::Column(Column::Int8(result))
        },
        _ => unreachable!()
    }
}
```

Evaluation

Extract the pattern for **vectorization**

```rust
fn register_2_arg_int8(&mut self, name: String, eval: impl Fn(i8, i8) -> i8) {
    self.register_function(Function {
        signature: FunctionSignature {
            name: "plus",
            arg: [DataType::Int8, DataType::Int8],
            return_type: DataType::Int8,
        },
        eval: |args: &[Value]| -> Value {
            match (&args[0], &args[1]) {
                (Value::Scalar(Scalar::Int8(lhs)), Value::Scalar(Scalar::Int8(rhs))) => {
                    Value::Scalar(Scalar::Int8(eval(lhs, rhs)))
                },
                (Value::Column(Column::Int8(lhs)), Value::Scalar(Scalar::Int8(rhs))) => {
                    let result: Vec<i8> = lhs
                        .iter()
                        .map(eval(*lhs, rhs))
                        .collect();
                    Value::Column(Column::Int8(result))
                },
                (Value::Scalar(Scalar::Int8(lhs)), Value::Column(Column::Int8(rhs))) => {
                    let result: Vec<i8> = rhs
                        .iter()
                        .map(eval(lhs, *rhs))
                        .collect();
                    Value::Column(Column::Int8(result))
                },
                (Value::Column(Column::Int8(lhs)), Value::Column(Column::Int8(rhs))) => {
                    let result: Vec<i8> = lhs
                        .iter()
                        .zip(rhs.iter())
                        .map(|(lhs, rhs)| eval(*lhs, *rhs))
                        .collect();
                    Value::Column(Column::Int8(result))
                },
                _ => unreachable!()
            }
        }
    });
}
```

Evaluation

As a result, it's easy to register overload for any **vectorized** input

```
registry.register_2_arg_int8("plus", |lhs: i8, rhs: i8| lhs + rhs);
registry.register_2_arg_int8("minus", |lhs: i8, rhs: i8| lhs - rhs);
```

But still many **duplicated types** to register

```
registry.register_2_arg_int16("plus", |lhs: i16, rhs: i16| lhs + rhs);
registry.register_2_arg_int16("minus", |lhs: i16, rhs: i16| lhs - rhs);
registry.register_2_arg_int32("plus", |lhs: i32, rhs: i32| lhs + rhs);
registry.register_2_arg_int32("minus", |lhs: i32, rhs: i32| lhs - rhs);
registry.register_2_arg_int64("plus", |lhs: i64, rhs: i64| lhs + rhs);
registry.register_2_arg_int64("minus", |lhs: i64, rhs: i64| lhs - rhs);
```

Evaluation

```rust
// Marker type for `Int8`
struct Int8Type;

// Define all methods needed to process data of the type
trait ValueType {
    type Scalar;

    fn data_type() -> DataType;
    fn downcast_scalar(Scalar) -> Self::Scalar;
    fn upcast_scalar(Self::Scalar) -> Scalar;
    fn iter_column(Column) -> impl Iterator<Item = Self::Scalar>;
    fn collect_iter(impl Iterator<Item = Self::Scalar>) -> Column;
}
```

```rust
impl ValueType for Int8Type {
    type Scalar = i8;

    fn data_type() -> DataType {
        DataType::Int8
    }
    fn downcast_scalar(scalar: Scalar) -> Self::Scalar {
        match scalar {
            Scalar::Int8(val) => val,
            _ => unreachable!(),
        }
    }
    fn upcast_scalar(scalar: Self::Scalar) -> Scalar {
        Scalar::Int8(scalar)
    }
    fn iter_column(col: Column) -> impl Iterator<Item = Self::Scalar> {
        match col {
            Column::Int8(col) => col.iter().cloned(),
            _ => unreachable!(),
        }
    }
    fn collect_iter(iter: impl Iterator<Item = Self::Scalar>) -> Column {
        let col = iter.collect();
        Column::Int8(col)
    }
}
```

Evaluation

Extract the data type using **marker type**

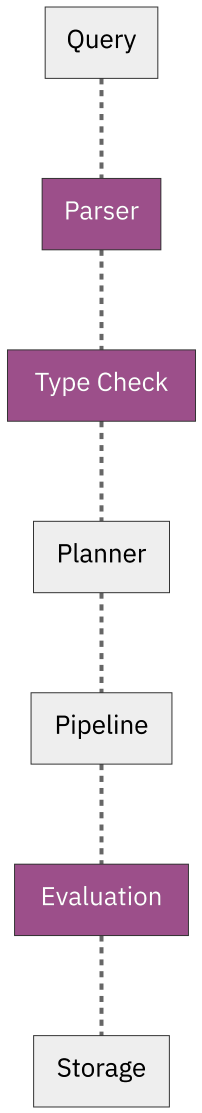
```
fn register_2_arg<I1: ValueType, I2: ValueType, Output: ValueType>(
    &mut self,
    name: String,
    eval: impl Fn(I1::Scalar, I2::Scalar) -> Output::Scalar
) {
    self.register_function(Function {
        signature: FunctionSignature {
            name: "plus",
            arg: [I1::data_type(), I2::data_type()],
            return_type: Output::data_type(),
        },
        eval: |args: &[Value]| -> Value {
            match (&args[0], &args[1]) {
                (Value::Scalar(lhs), Value::Scalar(rhs)) => {
                    let lhs: I1::Scalar = I1::downcast_scalar(lhs);
                    let rhs: I2::Scalar = I2::downcast_scalar(rhs);
                    let res: Output::Scalar = eval(lhs, rhs);
                    Output::upcast_scalar(O::upcast_scalar(res))
                },
                ...
            }
        }
    });
}
```

Evaluation

As a result, its easy to register overload for input of **any data type**

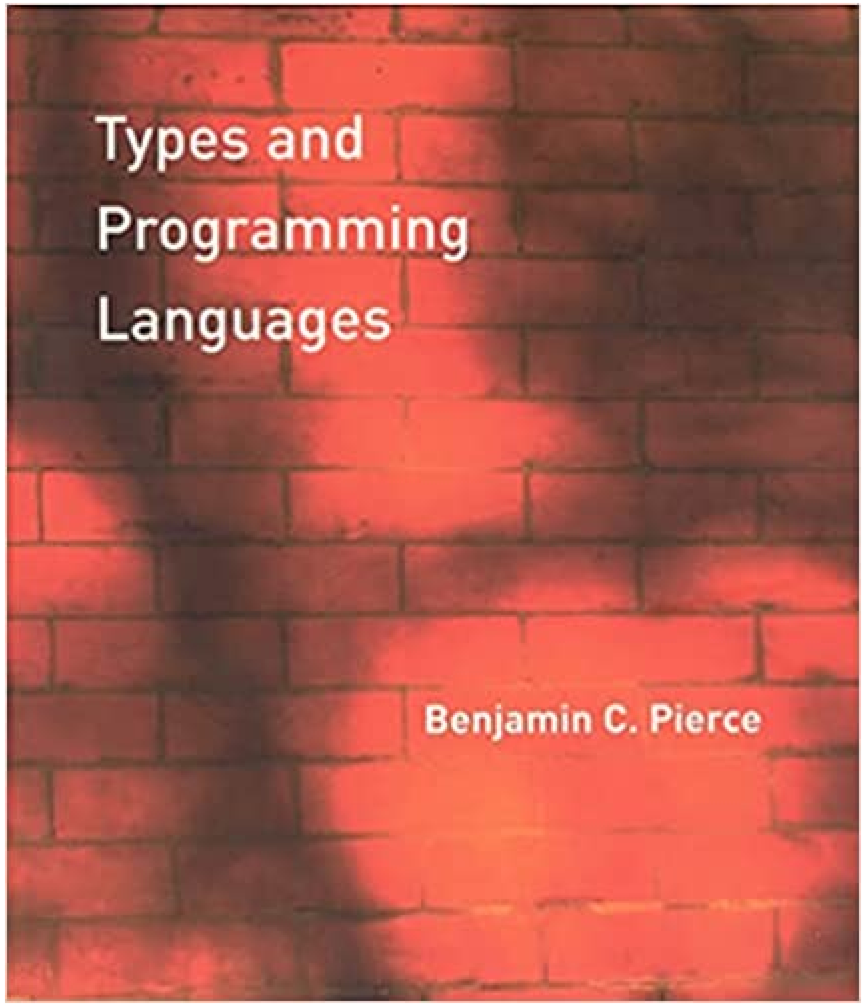
```
registry.register_2_arg::<Int8Type, Int8Type, Int8Type>(
    "plus", |lhs: i8, rhs: i8| lhs + rhs
);
registry.register_2_arg::<Int8Type, Int8Type, Int8Type>(
    "minus", |lhs: i8, rhs: i8| lhs - rhs
);
```

Conclusion



Query
Parser
Type Check
Planner
Pipeline
Evaluation
Storage

Conclusion



Types and Programming Languages