Compiler for CP

Utkarsh Dhandhania
UID - 3035550705
What is CP?
A Programming Language

• Based on Compositional Programming paradigm
• Elaborates to $F_i^+$ calculus, a smaller programming language
• Has high degree of Modularity and Extensibility
• Has a LaTeX-like Embedded DSL for document authoring, called ExT
Writing Code in CP

Algebraic Expressions - Creating a new data type

**CP**

```
1 type ExpSig<Exp> = {
2     Lit : Int -> Exp;
3     Add : Exp -> Exp -> Exp;
4 }

5 type Eval = {
6     eval : Int;
7 }

8 eval = trait implements ExpSig<Eval> => {
9     (Lit n).eval = n;
10     (Add a b).eval = a.eval + b.eval;
11 }
```

**Haskell**

```
1 newtype Exp = Lit Int
2       | Add Exp Exp

3 class Eval a where
4     eval :: a -> Int

5 instance Eval Exp where
6     eval (Lit n) = n
7     eval (Add a b) = (eval a) + (eval b)
```
Writing Code in CP

Algebraic Expressions - Creating an interface with method `eval`

```
1 type ExpSig<Exp> = {
2     Lit : Int -> Exp;
3     Add : Exp -> Exp -> Exp;
4 };;

5 type Eval = {
6     eval : Int;
7 };;
8
9 eval = trait implements ExpSig<Eval> => {
10     (Lit n).eval = n;
11     (Add a b).eval = a.eval + b.eval;
12 };;
```

```
1 newtype Exp = Lit Int
2 |
3 | Add Exp Exp
4 |
5 |
6 class Eval a where
7     eval :: a -> Int
8 |
9 instance Eval Exp where
10     eval (Lit n) = n
11     eval (Add a b) = (eval a) + (eval b)
12 |
```

CP

Haskell
Writing Code in CP

Algebraic Expressions - instantiating the interface Eval

```
1 type ExpSig<Exp> = {
2    Lit : Int -> Exp;
3    Add : Exp -> Exp -> Exp;
4 };;
5
6 type Eval = {
7    eval : Int;
8 };;
9
10 eval = trait implements ExpSig<Eval> => {
11    (Lit n).eval = n;
12    (Add a b).eval = a.eval + b.eval;
13 };;
```

CP

```
1 newtype Exp = Lit Int
2 | Add Exp Exp
3
4 class Eval a where
5    eval :: a -> Int
6
7 instance Eval Exp where
8    eval (Lit n) = n
9    eval (Add a b) = (eval a) + (eval b)
```

Haskell
Writing Code in CP

Extensibility - Adding new method disp

class Disp a where
    disp :: a -> String

instance Disp Exp where
    disp (Lit n) = show n
    disp (Add a b) = ""++(disp a)++""++(disp b)++""

Writing Code in CP

Extensibility - Adding new data variant Mul

type ExpSig'<Exp> = ExpSig<Exp> & {
  Mul : Exp -> Exp -> Exp;
};

eval' = trait implements ExpSig'<Eval> inherits eval => {
  (Mul a b).eval = a.eval * b.eval;
};

disp' = trait implements ExpSig'<Disp> inherits disp => {
  (Mul a b).disp = " " ++ a.disp ++ " * " ++ b.disp ++ " ";
};

newtype Exp = Lit Int
  | Add Exp Exp
  | Mul Exp Exp

class Eval a where
  eval :: a -> Int

instance Eval Exp where
  eval (Lit n) = n
  eval (Add a b) = (eval a) + (eval b)
  eval (Mul a b) = (eval a) * (eval b)

class Disp a where
  disp :: a -> String

instance Disp Exp where
  disp (Lit n) = show n
  disp (Add a b) = "++(disp a)++"++(disp b)++"
  disp (Mul a b) = "++(disp a)++"++(disp b)++""
Yaren, in earlier times Makwa/Moqua, is a district of the Pacific nation of Nauru. It is the de facto capital of Nauru and is coextensive with Yaren Constituency.\Cite(ref1)\Cite(ref2)

The district was created in 1968. Its original name, Makwa
Problem Statement

- Interpreted Language
- Interpreter written in PureScript
- Not fast enough
- A faster way to execute CP code is needed
Objective

Source Code in CP

Parser

Abstract Syntax Tree

Intermediate Code Generator

Intermediate Code in $F_i^+$

Compile

Output in JavaScript

Goal 1

Create a faster Lexer and Parser to generate an Abstract Syntax Tree from Source Code.

Goal 2

Create a compiler to translate the Merge Operator, which is an integral part of $F_i^+$, to JavaScript
Objective

Goal 1
Create a faster Lexer and Parser to generate an Abstract Syntax Tree from Source Code.

Goal 2
Create a compiler to translate the Merge Operator, which is an integral part of $F_i^+$, to JavaScript.
Lexer and Parser

How is an Abstract Syntax Tree created from Source Code?

Source Code

\[ x = 1 + 2 \]

Tokens

- "x"
- "=
- "1"
- "+"
- "2"

Parser

Parse Tree

- Statement
  - Expression
    - x
    - =
    - 1
    - "+"
    - 2

Visitor

Abstract Syntax

- Assign
  - Sum
    - 1
    - 2
Methodology - Lexer and Parser

Using a Parser Generator - ANTLR

Language Grammar

stmt: id "=" expr ;
expr: Num
    | expr "+" expr
    | expr "-" expr ;
Num : [0-9]+ ;
Methodology – Lexer

How the Lexer was made?

• In the Lexer Grammar file for ANTLR, all the different kinds of tokens in CP are defined.

• The language of generated parser has been specified as JavaScript

• ANTLR supports lexical modes. This is allows us to easily parse the embedded document language, ExT.
Methodology – Lexer

How the Lexer was made?

- In the Lexer Grammar file for ANTLR, all the different kinds of tokens in CP are defined.

- The language of generated parser has been specified as JavaScript.

- ANTLR supports lexical modes. This is allows us to easily parse the embedded document language, ExT.

```csharp
lexer grammar CPLexer;

options {
  language = JavaScript;
}

/* LITERALS */

IntLit : [0-9]+ (',', [0-9]+)? (\'(e' | 'E') ('+' | '-'?) [0-9]+)?
  | (\'0x' | '0X') [0-9a-fA-F]+
  | (\'0o' | '0O') [0-7]+
  ;

StringLit : (\'\"\" (\r\n\r\n) | \r\n\.)\"\" \"\"
  ;

/* KEYWORDS */

Type : 'type'
  ;
```
Methodology – Lexer

How the Lexer was made?

- In the Lexer Grammar file for ANTLR, all the different kinds of tokens in CP are defined.

- The language of generated parser has been specified as JavaScript

- ANTLR supports lexical modes. This is allows us to easily parse the embedded document language, ExT.
Methodology – Parser

How the Parser was made?

- In the Parser Grammar file, the syntax of CP is defined.
- Along with the target language JavaScript, the Lexer Grammar also needs to be specified.
- The grammar files are used by ANTLR to generate a parser in the specified language.
Methodology – Parser

How the Parser was made?

- In the Parser Grammar file, the syntax of CP is defined.
- Along with the target language JavaScript, the Lexer Grammar also needs to be specified.
- The grammar files are used by ANTLR to generate a parser in the specified language.
Methodology – Visitor

How the Visitor was made?

• ANTLR generates a template Visitor Class, which has pre-defined functions like `visitExpression()`, `visitOpexpr()`, etc. to perform a Depth-First Traversal of the Parse Tree.

• These functions can be overwritten to perform a traversal and transform the Parse Tree into an Abstract Syntax Tree.
Methodology – Visitor

How the Visitor was made?

• ANTLR generates a template Visitor Class, which has pre-defined functions like `visitExpression()`, `visitOpexpr()`, etc. to perform a Depth-First Traversal of the Parse Tree.

• These functions can be overwritten to perform a traversal and transform the Parse Tree into an Abstract Syntax Tree.
Results – Parser

Comparison between Old and New Parser

- Tested on over 10,000 lines of code.
- 93% less time on average
Objective

Create a faster Lexer and Parser to generate an Abstract Syntax Tree from Source Code.

Goal 1
Create a compiler to translate the Merge Operator, which is an integral part of $F_i^+$, to JavaScript.

Goal 2
Compiling Merge Operator

What is the Merge Operator?

37 : Int
False : Bool

Merge

(37, False) : Int&Bool

Intersection Type
Compiling Merge Operator
Type Coercion of merged values

Example 1
not (37, False) → not False → True

Example 2
1 + (37, 22) → 38 OR 23

To avoid this ambiguity,
Only Disjoint Types are allowed to be merged
## Compiling Merge Operator

What makes 2 types "Disjoint"?

Consider the types – Int | Boolean | String | A → B | A & B

<table>
<thead>
<tr>
<th>Type 1</th>
<th>Type 2</th>
<th>Disjointness Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primitive (A)</td>
<td>Primitive (B)</td>
<td>A ≠ B</td>
</tr>
<tr>
<td>Primitive</td>
<td>Function (A → B)</td>
<td>Always Disjoint</td>
</tr>
<tr>
<td>Function (A₁ → B₁)</td>
<td>Function (A₂ → B₂)</td>
<td>The return types B₁ and B₂ are disjoint</td>
</tr>
<tr>
<td>Intersection (A &amp; B)</td>
<td>Any (C)</td>
<td>(A and C are disjoint) &amp;&amp; (B and C are disjoint)</td>
</tr>
</tbody>
</table>
Methodology – Compiling Merge Operator

How to represent values in JavaScript so that they can be merged and coerced?

JavaScript Objects!

• JavaScript Objects are essentially a Hash Map from strings to values any type.

• Objects can be merge using the "Object.assign()" function.

• Type coercion is possible by accessing object attributes.

• A way to convert types into unique and reproducible strings is needed

```javascript
let obj1 = {
  "Int" : 1,
  "String" : "Hello"
};

let obj2 = {
  "Bool" : false
};

// merge
let obj = Object.assign({}, obj1, obj2);

// type coercion to Int
let objInt = obj.Int
```
Results – Compiling Merge Operator

Primitive Types to String

- Int → "Int"
- Bool → "Bool"

Function with Primitive Return Type to String

\[ A_1 \rightarrow A_2 \rightarrow A_3 \rightarrow \text{String} \rightarrow \text{fun_fun_fun_String} \]
Results – Compiling Merge Operator
Splittable types to String
Results – Compiling Merge Operator

CP → JavaScript

```javascript
var var_4 = {
  "splitBegin_fun_Double_fun_Int_splitEnd": function (x) {
    var var_1 = {
      "splitBegin_fun_Double_fun_Int_splitEnd": function (y) {
        var var_0 = Object.assign({}, x, y);
        return var_0;
      }
    }
    var var_2 = {
      Double: 2.0
    }
    var var_3 = var_1.splitBegin_fun_Double_fun_Int_splitEnd(var_2);
    return var_3;
  }
}
var var_5 = {
  Int: 1.0
};
var var_6 = var_4.splitBegin_fun_Double_fun_Int_splitEnd(var_5);
console.log(var_6);
```
Results – Compiling Merge Operator

Execution Time – Interpreter vs Compiler

- CP code compiled into JavaScript takes around 95% less time to execute.
- The time difference between the compiler and interpreter increases with the size of the program.
Key Takeaways

- CP is a programming language that elaborates to $F_{i}^+$
- The project involves developing two components of a compiler for CP –
  1. A Parser to create an AST from CP source code
  2. A compiler which translates Merge Operator in $F_{i}^+$ to JavaScript
- The Parser has been implemented using ANTLR, and takes 93% less time
- The compiler for Merge Operator is part of a bigger compiler. The compiled code executes in 95% less time than the interpreter.
Q & A?