Final Year Project Report

Compiler for CP

Utkarsh Dhandhania
UID : 3035550705

Supervised by:
Dr. Bruno C. d. S. Oliveira

Date : April 18, 2022
Abstract

CP is a new programming language based on the Compositional Programming. CP allows writing highly modular code. It allows extensibility in multiple dimensions without modifying existing code. CP also has a document-authoring language called ExT embedded in it. ExT, by virtue of being embedded in CP, is also extensible. New features and language constructs can be modularly added to ExT. In its current form, CP is an interpreted language with its interpreter written in PureScript. It is possible to write and execute programs in CP. However, the execution times can be undesirably long. This project aims to solve this problem, and does that in two ways. The first of them is creating a new and faster parser for CP. The current parser, which is a part of the CP interpreter, is built using the parser combinator library of PureScript. Parser combinators are a great way to prototype parsers, but they can be inefficient and slow. In this a project, ANTLR, a parser generator has been used to develop a new parser. The new parser generated is written in JavaScript, which allows it to be easily integrated and deployed with the existing in-browser interpreter. Experiments show that the new parser is upto 99% less time than the existing one. The second contribution this project makes to speed up CP is by designing and implementing a compilation technique for the merge operator. CP elaborates to Fi+ as an intermediate language during interpretation (and also compilation). The merge operator is an important component of the Fi+ calculus and facilitates the modularity of CP. The compiler for merge operator will be part of the complete compiler for Fi+, and therefore contribute to making CP faster.
Acknowledgement

I would like to thank Dr Bruno Oliveira for introducing me to Functional Programming and Programming Languages research and giving me really cool projects to work on and learn more. I would also like to thank Yaozhu Sun for his constant guidance and help as I learnt about different aspects of CP, the interpreter, and many other things. My peers and friends like Harsh, Manya and Shrivatsa have been a constant source of support and motivation. Thanks to them for that. I would also like to thank my parents and my sister for being there for me at my best and worst.
# Table of Contents

1. Introduction  
   1.1. Background  
   1.2. Motivation  
   1.3. Objective & Scope  
2. Methodology  
   2.1. Lexer and Parser for CP  
   2.2. Compiler for Merge Operator  
3. Results  
   3.1. Lexer, Parser, and Visitor  
   3.2. Merge Operator  
4. Limitations  
5. Conclusion  
6. Bibliography
List of Figures

Figure 1 - A simple CP program to add 2 numbers 1
Figure 2 - Non-parameterised terms / constants 2
Figure 3 - Parameterised terms / functions 2
Figure 4 - A Compositional Interface defining a List of Int 2
Figure 5 - A trait to evaluate the sum of a list using the sort Sum 2
Figure 6 - A trait create an list object from the interface ListSig 3
Figure 7 - traits sum and list merged and the method sum is called 3
Figure 8 - An Example of ExT document 4
Figure 9 - Values created using merge operator 4
Figure 10 - Using “merged” values in expressions 4
Figure 11 – Example of ambiguity upon merging non-disjoint types 5
Figure 12 - Different stages of a compiler 6
Figure 13 - A lexer grammar for arithmetic expressions 10
Figure 14 - Example of 'body' being used differently in HTML 11
Figure 15 - parser grammar for arithmetic expressions. 12
Figure 16 - Parse Tree to AST 12
Figure 17 - (1,,False) in JavaScript 13
Figure 18 - Merging using Object.assign() 14
Figure 19 - Lexer Grammar for CP 16
Figure 20 - Parser Grammar for CP 17
Figure 21 - Flattening of (Int&(Bool&(Int -> String))) 19

List of Abbreviations

AST – Abstract Syntax Tree
ANTLR – ANother Tool for Language Recognition
ES – EcmaScript

List of Tables

Table 1 – A comparison of parsing times taken by PureScript’s parser combinator and ANTLR 18
1. Introduction

CP[1][2] is a new programming language which implements Compositional Programming[2], a programming paradigm which allows the programmer to write highly modular programs. Compositional Programming solves the Expression Problem[3], i.e. it allows extensibility in multiple dimensions. This means that for a data structure, new constructs and new operations can be added modularly, without modifying preexisting code. CP elaborates to $F_{i+}[4]$, a smaller programming language, which is a subset of CP. CP also has an embedded domain-specific language for document-authoring called ExT[1].

1.1. Background

This section gives a brief description of the necessary topics and concepts to understand the project. It begins with a brief description of the two languages CP and ExT. This is followed by an explanation of what the Merge Operator[5] is.

The CP Programming Language

Some basic features of the CP language are discussed below, followed by an example of a CP program which employs the features described. This is not an exhaustive list of CP features, but given an idea of what programming in CP is like.

Program Structure. The structure of a typical CP program is simple. It is a semicolon-separated sequence of definitions, followed by an expression that evaluates to create the output of the program. A very primitive program for adding 2 number is shown in Figure 1. The definitions can either be terms or types, which can optionally be parameterised.

```plaintext
a = 1;
b = 33;
a + b                    -- Evaluates to 34
```

Figure 1 - A simple CP program to add 2 numbers

Terms. Non-parameterised terms behave like constants, and parameterised terms behave like functions, similar to any functional programming language like. This is demonstrated in Figures 2 and 3 respectively.
Types. Like terms, types can also be parameterised. A notable feature of CP is that types can also be parameterised using *sorts*. These sorts are used to create Compositional Interfaces as shown in Figure 4.

![Diagram of Compositional Interfaces](image)

**Compositional Interfaces.** Compositional Interfaces are like OOP interfaces, but they also allow constructor signatures. These can be used to create objects or data structures using an interface, without a concrete implementation of a class or associated methods. Compositional Interfaces are parameterised with a *sort* which is used to define operations for the data structures represented by the compositional interface.

**Compositional Traits.** Traits are basically a group of methods. In CP, compositional traits are used to define operations on data structures defined by compositional interfaces as shown in Figure 5. In Figure 5, methods to evaluate the sum of the List are defined. By using traits along with self-

```plaintext
type ListSig<T> = {
    Cons : Int -> T;
    Nil : T;
};
```

**Figure 4 - A Compositional Interface defining a List of Int**

```plaintext
type Sum = {
    sum : Int;
};
sum = trait implements ListSig<Sum> => {
    (Cons n ns).sum = n + ns.sum;
    (Nil).sum = 0;
};
```

**Figure 5 - A trait to evaluate the sum of a list using the sort Sum**
type annotations, objects of data structures can be created from Compositional Interfaces as shown in Figure 6, where a list object [5,2,8] is created from the interface ListSig.

```haskell
list = trait [self: ListSig<Sum>] => {
    l = (Cons 5 (Cons 2 (Cons 8 Nil)));
};
```

Figure 6 - A trait create an list object from the interface ListSig

**Merge Operator.** A notable feature of compositional programming is the merge operator. It can used to combine values (primitive values, traits, functions, etc.) of more than in type. An `Int` and a `Bool` value can be merge to create a value of type `Int&Bool`. It can also be used to combine traits which are dependent on one another. This combination of traits if demonstrated in Figure 7 where the traits defined in Figures 5 and 6 are merged and the method `sum` is called.

```
(new sum, list).l.sum
```

Figure 7 - traits `sum` and `list` merged and the method `sum` is called

**ExT, the embedded document-authoring language**

ExT is the document authoring language embedded in CP[1]. It has a syntax similar to that of LaTeX. ExT is composed of commands of the format `\Cmd` followed by `{…}, (…), […]`, or any combination of them. `{…}` is used pass values of *record type* into the ExT commands. (…) is used to pass values of any type supported by CP to ExT commands. Finally, […] passes ExT code into an ExT command. CP expressions can also be used in ExT using the syntax `\(…\)` with a CP expression between the parentheses. Under the surface, ExT is syntactic sugar for CP code. For example - `\Href("www.hku.hk")[HKU]` desugars to `(Href "www.hku.hk" (Str "HKU"))`. ExT commands and libraries are defined in CP. This means that the ExT language can be easily extended. New language features and new target languages can be modularly added without modifying existing code. A typical ExT document is shown in Figure 8.
Merge Operator

Merge Operator is a key component of CP, as well as the $Fi^+$ which it elaborates to[2][4]. It combines values to create new values which have more than one type. This allows the creation of values like in Figure 9. The types of these “merged” values are called Intersection Types, for example - Int & String.

\[
x: \text{Int} \& \text{String} = 1 ,, "I am a String"
y: \text{Float} \& \text{Bool} = 3.14 ,, \text{False}
\]

Figure 9- Values created using merge operator

The values thus created can be used as either of its constituent types in expressions. An example of this behaviour is demonstrated in Figure 10.

\[
2 + x \quad // \text{evaluates to 3}
not \ y \quad // \text{evaluates to True}
\]

Figure 10 - Using “merged” values in expressions

In the Fi+ calculus, only disjoint types are allowed to be merged[4]. Two types are disjoint when they have no common supertypes. This restriction prevents ambiguity when using the merged values in expressions[5]. An example of such an ambiguity is demonstrated in Figure 11. Types of values $x$ (Int&Bool) and $y$ (Int&String) have a common supertype, Int. Therefore, if they are
merged and the new composed value \((x,,y)\) is added to 0, of type Int, an ambiguity occurs. This is because the value \((x,,y)\) has two \(Ints\) and cannot be converted into a unique Int value.

The Merge Operator cannot only be used to combine values of types like \(Int\), \(Boolean\), and \(String\). It can also be used to merge functions and records[5]. A function which takes input of \(n\) parameters of type \(A_i\) for \(i = 1, 2, \ldots n\), and returns a value of type \(B\), is of type \((A_1 \rightarrow A_2 \rightarrow \ldots \rightarrow A_n \rightarrow B)\). For example, a function which takes 2 \(Ints\) as parameters, and returns a \(String\) is of type \(Int \rightarrow Int \rightarrow String\). Thus, functions can also be combined using the merge operator, only if they are disjoint. In case of functions, the disjointness condition works like this. Let there be two functions – \(f : (A_1 \rightarrow A_2 \rightarrow \ldots \rightarrow A_n \rightarrow B)\), and \(g : (C_1 \rightarrow C_2 \rightarrow \ldots \rightarrow C_m \rightarrow D)\). The functions \(f\) and \(g\) are disjoint if their return types \(B\) and \(D\) are disjoint OR their number of parameter are unequal i.e. \(n \neq m\)[5].

Records are a mapping from a label to a value. Their type is composed of two components – the label and the type of the corresponding value[4]. Two records are called disjoint if either the label is different, or the types of the values is disjoint. For example \{\(a : Int\&Bool\}\} and \{\(b : Int\&Bool\}\} are disjoint, but \{\(a : Int\&Bool\}\} and \{\(a : Int\}\} are not.

It is important to note that the merge operation (as well as type intersection) is commutative and associative. The property of commutativity implies that \((1,,False) : Int\&Bool\) is equivalent to \((False,,1) : Bool\&Int\). In addition, associativity implies that values like \((1,,(False,,"Hello")) : Int&(Bool&String)\) and \(((1,,False),,"Hello") : (Int&Bool)&String\) are equivalent.

---

**Compilation**

Compilation is the process of translating a computer program in a *source language* into a *target language*[6]. This translation usually produces code that executes faster than an interpreter processing the source code. The program that performs this translation is called a *compiler*. A compiler also checks the source code for syntactic and semantic errors. The input to a compiler is a stream of characters. It is processed in various stages and transformed into code in a target language.
that can be executed on a computer [6]. A simplified structure of these stages is shown in the Fig. 12.

Figure 12 - Different stages of a compiler

*Lexical Analyser and Syntax Analyser:* These stages translate the input character stream into an *Abstract Syntax Tree* or *AST*. It contains just the essential information about the procedures the code
represents. The Lexical analyser translates character stream into a sequence of tokens of the programming language[6]. Common examples of tokens are if, for, int, and variables like i, num, etc. The Syntax Analyser converts the sequence of tokens into a syntax tree representing the program[6]. This syntax tree is further processed to create the AST. These stages also check the code for lexical and syntax errors. These two stages can be combined into a single program, or be separate Lexer and Parser.

**Semantic Analyser.** A Semantic Analyser checks the AST for semantic errors. Examples of semantic errors include using values of improper places, accessing a value before declaration [6].

**Intermediate Code Generator.** In the next step of compilation, the semantically verified AST is translated into a simpler AST of a language that smaller, and easier to translate into code in target language code[6].

**Code Generator.** The intermediate code generated is then used to generate code in target language that can be executed on a computer[6].

The final code generated is often machine-level executable code. However, this is not necessary. The generated code can be in higher-level languages. PureScript[11] and TypeScript[12] are good examples, both of which compile to JavaScript[13].

### 1.2. Motivation

Any programming language, other than machine code, cannot be directly executed on a computer. It either needs to be compiled into another language that can be executed on a computer, or executed using an interpreter, which runs the code without compiling it. Currently, CP, along with ExT, currently exists as an interpreted programming language. Its interpreter is written in PureScript.

Many of the stages in the interpreter are similar to what a compiler would look like. The first stage is a combined lexer-parser. It is built using the PureScript’s parser combinator library called parsing. The AST generated is then translated into $F_i^+$, before being interpreted. In its current form, CP works but is not fast enough, so its practical utility remains limited despite the modularity it allows. There are two bottlenecks observed in the speed of CP.

**Parser Combinator.** Parser combinators are a convenient method to create and prototype parsers. When executed using node on an Apple M1 machine, CP’s parser takes around 0.86 milliseconds – on an average – to convert one line of code into Abstract Syntax Tree. This extrapolates to more than 8 seconds for just parsing 10,000 SLOC. This is inefficient and slow

**Interpreter.** Interpreters are generally slower as compared to compiled code. CP interpreter is written in PureScript, which compiles to JavaScript. JavaScript itself is an interpreted language[13].
The result is that we have an interpreter running on another interpreter. This compounds the already relatively low speed of interpreters. Fixing these issues will make CP code execution much faster, which shall make a stronger case for the usability and utility of a programming language like CP.

1.3. Objective & Scope

The objectives of the project revolves around improving the execution speed of CP. There are two primary goals of the project –

• Creating a new lexer and parser for CP
• Creating a compiler for the Merge Operator

Creating a new lexer and parser for CP. A new lexer and parser for CP has been built as part of this project. This part of the project can be further broken down into three parts. The first is creating the lexer, which shall tokenise the input stream of characters into a sequence of tokens. The next is creating a parser, which creates a parse tree from the sequence of characters. The final step is creating a program to translate the generated parse tree to extract the relevant information from the parse tree to create an Abstract Syntax Tree

Creating a compiler for Merge Operator. The Merge Operator is an important part of Fi+ which CP elaborates to. A compiler for the Merge Operator will be helpful and necessary for the development of a complete CP language compiler. For creating a compiler for Merge Operator, a compilation technique needs to be devised first. Then, the compilation technique needs to be implemented in PureScript so that it can be integrated into the complete CP compiler.
2. Methodology

2.1. Lexer and Parser for CP

The first stage of interpreting (as well as compiling) is creating an Abstract Syntax Tree from Source Code. This is done in 3 steps. The first step is a lexer. A lexer takes a stream of characters and converts them into a sequence of tokens. The next step is a parser. The parser checks the sequence of tokens for syntax errors and converts it into a parse tree based on the grammar of the language. The third and the final stage is called a visitor. The parse tree generated contains a significant amount of extraneous information like parentheses, semicolons etc. A visitor performs a depth-first traversal of a tree. In this case, it will traverse the parse tree, removes unnecessary information, and converts it into an Abstract Syntax Tree. The AST does not directly correspond with the grammar of the language like a parse tree, but maintains all the necessary information required for computation. The lexer, parser, and visitor as described above will be created with the help of ANTLR[7]. ANTLR is a Parser Generator. What a parser generator does is that it takes the grammar of a language as input, and generates a lexer, a parser, and optionally a generic visitor, which can be overwritten to further process the parse tree generated by the parser. ANTLR can generate parsers in a variety of languages. In this project, the language of choice if JavaScript so that it can be easily integrated into the existing CP interpreter, which is written in a language that compiles to JavaScript[11].

---

**Lexer**

To create the lexer, all possible tokens in CP need to be defined in ANTLR grammar. CP has a variety of tokens which include literals, comments, keywords, different kinds of identifiers, and symbols [1]. All these different kinds of tokens are discussed below.

*Whitespaces.* Whitespaces include the characters ' ', ‘\n’, ‘\r’, and ‘\t’. These are used to separate the other token and do not provide any special meaning to the program. They are ignored when the sequence of tokens is generated by the lexer.

*Comments.* CP has two types of comments – single-line and multi-line. Single-line comments begin with a double hyphen ('--'), and end at a new line. Multi-line comments begin and end with '{ - ' and ' - }' respectively.

*Literals.* CP literals comprise integer and floating-point numeric values like 3.14, and 202, boolean values like true and false, and string values like "hello".
Keywords. These are words that possess a special meaning. A common example of keywords are type names like Int and Double, and also words which specify what instruction computer should perform like if and else.

Identifiers. Identifiers are the names given to terms like constants, functions, and types. In CP, term identifiers begin with a lower-case letter, whereas type identifiers begin with an upper-case letter. In addition, CP has an underscore (‘_’) identifier, which can be used a wildcard in pattern-matching.

Symbols. Like all programming languages, CP also has symbols as part of its grammar. These include but are not limited to ‘+', '−', ';', '{'.

To create a lexical analyser for this list of tokens using ANTLR, these tokens need to specified in ANTLR grammar. ANTLR takes this grammar as input and creates a lexer in the desired target language, which in case of this project is JavaScript. The generated lexer can be imported in other JavaScript programs to be used. It also generates errors when lexical analysis fails to recognise characters in the input code. Regex expressions are used to describe the different tokens, and each token is assigned a name. A basic token vocabulary to express arithmetic expressions is show int Fig 13.

```
lexer grammar Exp;

Add : '+' ;
Sub  : '−' ;
Mul  : '*' ;
Div  : '/' ;
Num  : [0-9]+ (\.' [0-9]+)?
```

Figure 13 - A lexer grammar for arithmetic expressions

A single lexical grammar is not always enough to represent a whole language. Same tokens can have different meanings in different contexts. A simple example of this is HTML. Consider the code in Fig. 14. When used inside angle brackets (<...>), the token body specifies which HTML tag is
being used. However, outside angle brackets, body behaves like ordinary text. This is a much simpler version of the problem that is faced in case of CP, which has another language ExT, embedded in it. ExT is written using commands, much like LaTeX. Like HTML tags, these commands are a mini-grammar of their own. Moreover, CP expression can used as arguments in ExT commands. This allows general purpose computation provided by CP to be used to write ExT documents. However, these multiple layers of nested grammars make lexical analysis complicated. To handle languages like these, ANTLR has a feature called Lexical Modes. Each lexical mode has its different set of tokens, with certain tokens triggering changes in modes. This provides a form of context-based lexical analysis necessary for CP.

```html
<body>
    This is body.
</body>
```

Figure 14 - Example of 'body' being used differently in HTML

---

Parser

Just like the lexer, to create the parser, the grammar of the language needs to be specified in ANTLR grammar format and parser is generated. In the parser grammar, how different tokens generated from lexical analysis interact with one another and create different structures of the programming language like function definitions, arrays, and algebraic expressions to name a few. These different structures, often mutually recursive, build together to form a top-level structure. Some of these structures are discussed below[1].

**Program.** As mentioned in Section 1.1, a CP program is definitions of followed by an expression.

**Type Definition.** A type definition is defined by the keyword type followed by the name of the type, sorts, type parameters, = sign, and finally ending with the the actual definition of the type and a semicolon.

**Term Definition.** The term definition begins with the name of the defined term. It is followed by term parameters, type parameters, an optional type annotation, '=' sign, the expression for the term definition and Semicolon.

A parser grammar for arithmetic expression is shown as an example in Fig. 15.
Visitor

The parse tree created by the parser is large and has many unnecessary nodes. This makes it difficult to work with. Therefore, it needs to be translated into a smaller tree called AST before further processing. Along with the lexer and parser, ANTLR also generates a template visitor class containing functions that perform a depth-first traversal of the parse tree. These functions need to be overwritten to process the tree, which in this case means to create an AST. An example of how parser tree translates into an AST and how they differ is shown in Fig. 16.

```
parser grammar Exp;

exp : Num 
    | exp (Div | Mul) exp 
    | exp (Add | Sub) exp 
    ;
```

Figure 15 - parser grammar for arithmetic expressions

Figure 16 - Parse Tree to AST
2.2. Compiler for Merge Operator

Compilation is just a form of language translation. Before a sentence X in language A can be translated to language B, it needs to be known what sequence of words in language B will convey the same meaning as X. Similarly, To create a compiler for Merge Operator, the first step is to devise a way to implement the functionality of Merge Operator in JavaScript. The subsequent step is to create a compiler that performs that translation.

Compilation Technique

Disjoint Intersection Types and the merge operator have been discussed in Chapter 1.1. The problem that needs to be solved is how to translate values of disjoint intersection types and the merge operation into existing programming languages (in case of this project, JavaScript). It can be divided into 2 parts –

Representation of values of disjoint intersection types. A representation of values of disjoint intersection types is needed that supports accessing the constituent values by their type. For example, if there is a value \( a = (1, \text{isZero}) \) of type \((\text{Int} \& (\text{Int} \rightarrow \text{Bool}))\), there needs to be a way to access the constituent values just by knowing their type. This is to facilitate coercion into supertypes when needed. A way to accomplish this in JavaScript is using objects. JavaScript objects are essentially hash maps from a String to any other type. A value \( x = (1, \text{false}) \) can be represented as shown in Fig. 17. This is representation is straightforward when only primitive types like Int and Bool are used. However, when values like functions and function returning merged values are taken into consideration, a method to create unique names for types needs to be developed.

```javascript
var x = {
    Int : 1;
    Bool : false;
};
```

Figure 17 - (1,False) in JavaScript
Merging values. Once there is a straightforward way to represent and store values is designed in JavaScript, there needs to be a way to merge two values. Assuming the two values to be merged are disjoint, merging two objects in JavaScript is straightforward. There exists a function called Object.assign(), which does exactly that. An example of how it can be used is shown Fig. 18.

```javascript
var a = {
    Int : 1;
    Bool : false;
};

var b = {
    String : "Hello";
};

var c = Object.assign({}, a, b);

var c_ = {
    Int : 1;
    Bool : false;
    String : "Hello";
};

// c and c_ are equivalent
```

Figure 18 - Merging using Object.assign()

Compiler Implementation
The program to translate disjoint intersection type values and merge operations from CP to JavaScript is written in PureScript. It will be part of the bigger compiler complete CP compiler written in PureScript. The existing interpreter is written in PureScript, so the compiler created can easily be integrated into the existing code. The interpreter has multiple layers of processing CP code
already implemented which can be shared by the compiler. These include type checking, type inference, the various steps required to translate CP to Fi+, and other utility functions that can be helpful in the development of the compiler.
3. Results

3.1. Lexer, Parser, and Visitor

A newer system to generate an AST from CP source code has been built using ANTLR. The first step in creating a parser using ANTLR is to write the grammar of CP in Backus-Naur form in ANTLR’s syntax. The lexer and parser grammar can be separated into two files. ANTLR generated the lexer, parser, and a generic visitor in JavaScript from the grammar files.

**Lexer.** For the lexer grammar, all the different kinds of tokens are defined using regular expressions as shown in Figure 19. The generated lexer tokenises the source code into these tokens, giving preference to tokens of longer length to resolve ambiguities. An interesting challenge that was faced was that there is ExT, the document-authoring embedded DSL. This was a problem because the two languages has different syntax, and therefore different set of tokens. To tackle problems like these, ANTLR has a feature called Lexical Modes, which allows having languages inside languages. In CP, ExT documents begin and end with a backtick(‘), which were defined as the trigger token to switch to document mode and back.

```plaintext
lexer grammar CPLexer;

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

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

Type
  :   'type'
  ;

...
...
```

Figure 19 – Lexer Grammar for CP
In the parser grammar file, the associated lexer grammar is specified. Then using the tokens defined in the lexer grammar, the syntax of CP is defined as shown in Figure 20. The lexer and parser grammars are processed by ANTLR, to produce the lexer and parser, and a generic visitor.

Visitor. The generated visitor is a generic program which does nothing but just traverses the parse tree generated by the parser. This generic visitor is a template that is meant to be overwritten to create a visitor that can do something useful. In this project, this visitor has been overwritten to create an Abstract Syntax Tree which matches the one that is generated by the pre-existing CP interpreter written in PureScript. PureScript constructors are imported into JavaScript visitor file to create an AST which can be smoothly integrated without much modification to the existing interpreter.

Speedup. The goal of creating a new parser was to improve the parsing time of the CP interpreter, which was successfully achieved. The new parser created using ANTLR is, on an average, 93% faster than the older parser based on PureScript's parser combinator library. The comparisons are made on multiple CP and ExT files, both big and small. Generally, the speedup in larger files is much more significant than smaller files. In files exceeding 250 SLOC, the parsing times reduced
by around 99%. The speedup achieved is huge. Table 1 shows the parsing times for PureScript’s parser combinator and ANTLR parsers for programs of different lengths.

<table>
<thead>
<tr>
<th>SLOC</th>
<th>Time using parser combinator (ms)</th>
<th>Time using ANTLR (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>26</td>
<td>13.53</td>
</tr>
<tr>
<td>2</td>
<td>79</td>
<td>133.88</td>
</tr>
<tr>
<td>3</td>
<td>148</td>
<td>157.77</td>
</tr>
<tr>
<td>4</td>
<td>251</td>
<td>225.76</td>
</tr>
<tr>
<td>5</td>
<td>521</td>
<td>598.60</td>
</tr>
</tbody>
</table>

Table 1 – A comparison of parsing times taken by PureScript’s parser combinator and ANTLR

### 3.2 Merge Operator

The primary problem to solve was to generate unique name for types, so that values of different types like Boolean, Integer, String, (Integer -> Boolean), (Double -> String -> String) etc. can be combined together in a JavaScript Object. Generating unique names for primitive types, function types, and record types is straightforward as described below.

**Primitive Types.** To convert primitive types to a String, their names can be directly used in the string format. For example – "Int" for `Int`.

**Functions.** To convert functions to Strings, only the return type of the function is relevant. This is because only disjoint types can be merged and disjointness depends only on the return types of the functions. Therefore, function types can be converted into String by prefixing "fun_" `n` times, where `n` is the number of parameters of the function. For example – the type `(String -> (String -> Boolean) -> Boolean) -> Int)` would map to "fun_fun_fun_Int".

The method to generate unique names is inspired by the Prefix Notation of writing mathematical expressions. For example – `(5 + (7 - 2))` would be written as `+ 5 - 7 2` in prefix notation. The intuition is to similarly map the type `(Int & Boolean)` to "and_Int_Bool". However, applying this strategy directly on the merge operator does not work, because of its commutative and associative properties. The solutions to these problems are discussed below.

Due to commutativity, the types `Int & Boolean` and `Boolean & Int` would generate different names – "and_Int_Bool" and "and_Bool_Int" – while being the same type. This problem can be solved by defining an ordering between types.
Primitive Types. Primitive types like Int, Bool, and Double are ordered alphabetically.

Functions. Functions are assumed to be greater than Primitive Types. In general, all functions with (n+1) parameters are greater than all functions of n parameters irrespective of their parameter and return types. However, if two functions have an equal number of parameters, their ordering is decided by their return types.

Similarly because of associativity, equivalent types Bool&(Int&String) and (Bool&Int)&String would map to different strings "and_Bool_and_Int_String" and "and_and_Bool_Int_String". To solve this problem, the subtree formed by type intersections can be flattened into a list as shown in Fig. 21. Then, the types in the list can be sorted according to the ordering described earlier in this subsection. The final string can generated by starting and ending with keywords with the types in the sorted list between them. For example, the type (Int&(Bool&(Int -> String))) shall convert to "andBegin_Bool_Int_fun_String_andEnd".

![Figure 21 - Flattening of (Int&(Bool&(Int -> String)))](image-url)
4. Limitations

JavaScript exists in two variants – CommonJS[8], and EcmaScript (ES) [9] and interoperability between them remains limited. CommonJS files can be easily imported into ES files. However, the converse is not true. ANTLR-generated parser is in ES format, whereas PureScript compiles to CommonJS. The in-browser interpreter of CP used WebPack[10] for bundling, which makes integration between CommonJS and ES easy. However, the REPL interface of CP interpreter cannot use WebPack so the integration with the ANTLR parser is difficult. The current implementation of merge operator supports primitive types, functions, and records. It generates unique and reproducible names for any combination of these. However, it does not yet support parameterised types and polymorphism. The current implementation of the merge operator needs to be extended to support these features of Fi+.

5. Conclusion

The modularity and extensibility in CP is valuable and can be useful in software development. Software that can be modified to add new features and be maintained for a long time is always desirable. CP achieves that goal and ExT is an excellent example of that. The existing implementation of CP as an interpreted language is inefficient. This project makes contributions that solves the problem, at least in part. A new parser for CP has been developed using ANTLR. This has replaced its old parser which is implemented in PureScript using parser combinators. It was slow. The new parser is, on an average, 93% faster, and can be upto 99% faster on longer samples on code. This has resulted in CP being observably faster when executed in its in-browser interpreter. Moreover, a compilation technique for the merge operator has also been designed and a compiler based on that has been developed. Though the current implementation is limited in its scope, it is promising. Future work could consider extending it to other Fi+ features, as well as developing a complete compiler for all the different features of Fi+.
6. Bibliography


