Ștefan - Alexandru Pițur

Fully In-Place Updates in Functional Programming

Computer Science Tripos – Part II

Robinson College

May 13, 2024

Declaration

I, Ștefan - Alexandru Pițur of Robinson College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose. In preparation of this dissertation I did not use text from AI-assisted platforms generating natural language answers to user queries, including but not limited to ChatGPT. I am content for my dissertation to be made available to the students and staff of the University.

Signed Ștefan - Alexandru Pițur

Date May 13, 2024

Proforma

Candidate Number:	**2398E**
Project Title:	**Fully In-Place Updates in Functional Programming**
Examination:	**Computer Science Tripos – Part II, 2024**
Word Count:	**10496**[1]
Line Count:	**20444**[2]
Project Originator:	Dr Jeremy Yallop
Supervisor:	Dr Jeremy Yallop

Original Aims of the Project

Functional programming languages often perform copying when modifying an object. Copying is usually suboptimal because the initial object's memory might not be immediately reused, imposing a performance penalty on metrics such as peak memory usage and execution time. Therefore, memory reusage would be best for enhancing the program's performance. The project's core aim was to implement a functional programming language to identify at compile time and take advantage of in-place updates where safe, allowing programmers to write in a functional style while offering imperative execution that doesn't change observable behaviour.

Work Completed

I have met all the success criteria presented in the project's proposal. Furthermore, I have implemented one of the extensions, namely adding polymorphism to the language.

Therefore, FipML supports the full in-place execution of functions. These functions must comply with a second-order calculus presented in Lorenzen, Leijen, and Swierstra 2023, the foundation of my project. FipML compiles to OCaml native code to ensure efficient cross-platform code execution. I have conducted experiments on data structures and algorithms, such as red-black trees, quicksort, mergesort and mapping a function onto a generic tree. The results display the superiority of FipML over its non-optimised counterparts by having lower memory usage and faster execution times. (to be revised after running the experiments)

[1]Computed using Overleaf's buit-in counter.

[2]Computed using `git ls-files | xargs wc -l` from which remove those .exe executables

Special Difficulties

None.

Acknowledgements

I want to express sincere gratitude to my dissertation supervisor, Dr. Jeremy Yallop. His guidance, expertise, and unwavering support were crucial in completing this work. I am incredibly fortunate to have had the opportunity to learn from such a dedicated mentor who introduced me to a field that quickly turned into a passion for studying typing systems and compilers.

Throughout my academic journey, I received invaluable advice, encouragement, and insightful feedback from my Director of Studies, Dr. Richard W Sharp, to whom I am also indebted. Dr. Sharp's commitment to excellence has inspired me greatly, encouraging me to push the boundaries of my studies and strive for academic excellence.

To my beloved parents, sister, and grandparents, whose love, encouragement, and sacrifices have been the cornerstone of my success, I am eternally grateful. Your unwavering support and belief in me have been a source of strength and motivation, and I dedicate this achievement to you.

Last but certainly not least, I want to extend my heartfelt thanks to an extraordinary person who has been a constant source of inspiration, love, and encouragement. Your belief in me has fueled my determination, and your unwavering support has been the guiding light on this journey. Thank you for always being there for me.

Contents

Chapter 1

Introduction

1.1 Motivation

Functional programming has many attributes that make it popular among Computer Science enthusiasts and professionals. Unfortunately, whenever we code, we have a choice to make. We can either maintain a purely functional coding style or aim for efficiency by using in-place updates, therefore abandoning the pursuit of data immutability and referential transparency. To understand this problem better, it is best to look at a commonly encountered example, namely mapping a function over a list of elements.

```
type 'a list = Nil | Cons of 'a

let rec map_f (xs : int list) (f : int -> int) : int list =
    match xs with
    | Nil -> Nil
    | Cons (x, xs) -> Cons(f x, map_f xs f)
```

In OCaml and other functional programming languages, each object is immutable. Therefore, applying the `map_f` function must allocate memory for each new list element, requiring a garbage collection system to clear the heap during code execution. However, assuming that the list passed as an argument is unique (i.e. `xs` is the only reference pointing to the initial list), it is clear that we could do better by reusing the heap memory in-place, which is precisely what one would do in an imperative language.

FipML, the subject of this dissertation, is a functional programming language that can determine at compile time whether it is safe to execute a function fully in-place and apply this optimisation where fit.

To understand how FipML would optimise a function written in a functional style, it is best to see how the compiler reasons about `map_f`. When `xs` is unique, FipML analyses the destructive `match` expression:

- The `Nil` constructor in the first pattern takes no arguments. Therefore, the compiler has not previously allocated any memory to represent the atom. Hence, we may freely return a constant or another atom, as they do not require any additional memory allocation.

- The `Cons` constructor takes two arguments. FipML infers that to represent such value, it has previously allocated a memory block of size 2. Therefore, when the pattern row returns `Cons (f x, map_xs f)`, the compiler observes that it can reuse the memory of the matched constructor since the return value also requires a memory block of size 2 to store its contents.

Since we can reuse the memory for each branch of the `map_f` function, the compiler transforms the function into an imperative one with in-place updates:

```
let rec map_f! (xs : int list) (f : int -> int) : int list =
    match xs with
    | Nil -> Nil
    | Cons { hd = x, tl = xs} -> xs.hd<-(f x); xs.tl<-(map_f xs f); xs
```

1.2 Previous work and current research

The idea of introducing a calculus that makes precise where a compiler can achieve fully in-place (**fip**) updates is not new. Aspinall and Hofmann 2002 first introduce the concept of a non-linear type scheme that tracks the sharing properties of data structure and explicitly models an abstract resource $\diamond$, encoding the heap-allocated memory size of an object after a destructive usage. In this dissertation, we will refer to this resource as reuse credit because, in an in-place updating system, we want to reuse the memory when creating new constructors rather than allocating fresh heap memory. Furthermore, Aspinall and Hofmann 2002 allow for splitting and merging of reuse credits $\diamond$ to allow for more flexible memory usage - i.e. from a reuse credit of size 2, one can create two new constructors each of size 1.

More recent research by Lorenzen, Leijen, and Swierstra 2023 further develops these ideas and formulates a key result: reuse of allocated values is safe as long as the values are uniquely referenced. Lorenzen et al. used this result to create a linear calculus (FIP^S) under which a compiler can execute a broad class of programs without requiring any (de)allocation while maintaining the stack space constant, hence **fip** compliant. One of the key differences to the work of Aspinall and Hofmann 2002 is that FIP^S does not allow for splitting or merging of reuse credits. This difference is essential when considering a compiler's memory allocation pattern since an object's fields are stored in adjacent memory addresses. Lorenzen et al. have implemented this calculus in Koka (v2.4.2), using reference counting to determine at run-time whether it is safe to apply the optimisation on a function application or revert to the default non-optimised version based on owned variables sharing property.

Tracking variable ownership through a typing system rather than via dynamic reference counting is an active research area, as it enables expressivity and control over data and memory ownership. One functional programming language that exhibits such a uniqueness type system is Clean, which represents a source of inspiration regarding tracking variables sharing properties for new languages such as Rust and FipML or for future enhancements of existing systems such as OCaml (Lorenzen, White, et al. 2024).

Another alternative Lorenzen, Leijen, and Swierstra 2023 suggest but do not explore further is to statically determine if a function can be optimised by deploying a uniqueness type system to track whether a variable is unique or shared. I have decided to explore this approach in FipML, contributing to the field.

Chapter 2

Preparation

2.1 Starting point

I had limited exposure to OCaml during the Foundations of Computer Science course from Part IA. It is worth noting that during the development process, I used libraries such as Jane Street's Core and the Lambda module from the OCaml compiler distribution. Apart from that, I have meticulously constructed the entire codebase from the ground up. However, I have incorporated and expanded upon the concepts from the work I cite to provide a solid theoretical foundation for the project.

2.2 Requirement analysis

As outlined in Chapter 1, my approach, derived from the work of Lorenzen, Leijen, and Swierstra 2023, requires FipML to:

- statically determine at compile time whether fully in-place execution of specially annotated functions is possible,
- automatically apply the fully in-place memory re-usage optimisation where fit without resorting to the programmer to make use of reference, therefore preserving data immutability and referential transparency,
- deploy a uniqueness typing system that allows the programmer to model resource management,
- be flexible enough that the language can express and optimise interesting algorithms and data structures, such as Red-black trees, merge- and quicksort, and mapping a function over an algebraic data structure.

FipML is the first functional programming language to combine these features.

2.3 Background knowledge

This section pinpoints the tools, algorithms, and theories I studied to complete FipML's requirements. Section §2.3.1 begins with a short presentation of Ocamllex and Menhir, essential lexical analysis and parsing tools. Section §2.3.2 introduces the main concepts of type inference in a Damas-Hindley-Milner type system, fundamental to FipML's type checking. Following, section §2.3.3 presents the supporting citations and ideas for implementing a uniqueness typing system. Having presented the uniqueness typing concepts, section §2.3.4 informally introduces Lorenzen, Leijen, and Swierstra 2023 calculus FIP^S, forming the core theoretical foundation of this dissertation. Finally, section §2.3.5 presents Lambda, OCaml intermediate representation which FipML targets to provide code execution. This section also includes a short discussion about issues I have encountered compared to the project proposal's plan.

2.3.1 Ocamllex and Menhir

Ocamllex is a library widely used to generate an OCaml lexer from a declarative specification. In general, a lexer is responsible for splitting a text, the input code in the case of a compiler, into a stream of tokens. Following this stage, a parser uses the tokens to form a parsed abstract syntax tree (AST). Ocamllex creates a lexer from a file with `.mll` extension containing a set of regular expressions. The generated lexer matches these regular expressions in the order they appear, producing tokens. Here is an example lexer:

```
let digit = ['0'-'9']
let int_regex = '-'? digit+
rule token = parse
    | "+" { ADD }
    | "*" { MUL }
    | int_regex { INT (int_of_string (Lexing.lexeme lexbuf)) }
    | _ { raise (LexerError ("Unidentified token")) }
```

In the snippet above, we observe how to define regular expressions (`digit`, `int_regex`). Next, we define the parsing rule `token`, which matches the input buffer against `+`, `*` and `int_regex` in this exact order. If they all fail to match the input buffer, the lexer throws an error. Lastly, we see how we can execute arbitrary OCaml code on regex match (e.g., if the input buffer is `+`, then we create the `ADD` token).

FipML uses Menhir (Régis-Gianas 2016) for generating an OCaml parser. Defining a grammar within a .mly file is achieved by specifying its production rules and tokens and their associativity and precedence. Menhir uses these specifications to automatically generate an LR(1) parser following the algorithms studied in the Compiler Construction course from Part IB. I chose Mehir because it explains grammar conflicts more precisely than competing tools, such as Ocamlyacc. The example below shows how to specify tokens, their associativity and precedence, and production rules. It also displays how to execute arbitrary OCaml code after applying any production rule. Here is an example parser:

```
%token<int>INT
%token ADD MUL
%left ADD
%left MUL
e:
| i=INT { Integer i$ }
| e1=e; ADD; e2=e { Add(e1, e2) }
| e1=e; MUL; e2=e { Mul(e1, e2) }
```

The example parser firstly defines the tokens (`<int>INT`, `ADD` and `MUL`). Then we set `ADD` to lower precedence than `MUL` and make them both left associative. Finally, we present production rules. For input $1 + 2$, we generate an abstract syntax tree of the form `Add( Int 1, Int 2)`. Similarly, $3 * 4$ is encoded as `Mul( Int 3, Int 4)`.

Using Ocamllex and Menhir proved easy as they worked well together. Minsky, Madhavapeddy, and Hickey 2013 show more examples of their interoperability and how to connect them, resources I have used during the development's initial stage.

2.3.2 Damas-Hindley-Milner type inference

Compared to a dynamically typed programming language, a strongly typed one can prevent execution errors at compile time rather than runtime (Milner 1978). Unfortunately, specifying the types of all the variables and expressions can be difficult for a programmer as it adds to the language's verbosity.

Type inference is an algorithm that aims to provide all the advantages of a strongly typed language while requiring minimal input from the programmer, achieving the best of both worlds. Moreover, it is at the core of most functional programming languages, such as OCaml and Haskell. There are multiple implementations of type inference, but I focused on Algorithm W (Milner 1978), which guarantees that, if successful, it returns the most general type for any term t under a typing context Γ.

To ensure the correctness of FipML programs, I implemented type inference by modifying the constraint-generation rules for a classical Hindley-Milner type system (Pierce 2002) to fit the language syntax. Later, I have added support for let-polymorphism, which involved changes to the basic monomorphic type inference algorithm, in particular to the `Let` and `Var` rules, having to support two critical operations: generalisation and instantiation (Milner 1978)(Pierce 2002). Section §3.4 will present FipML's typing rules. Therefore, FipML can correctly typecheck and accept programs such as :

```
let id (x : 'a) : 'a = x in
print_string (id "string");
print_int (id 0);
```

2.3.3 Uniqueness typing system

As section §1.2 mentions, FipML leverages a uniqueness typing system to determine at compile-time whether a function can be executed fully in-place. I added such a typing system to FipML by modifying and extending the uniqueness rules presented in the work of De Vries, Plasmeijer, and Abrahamson 2008 to fit the language syntax. de Vries et al. 2008 suggest annotating partial types (i.e. `int`, `bool`, `custom_type`) with uniqueness attributes (i.e. `shared`, `unique`) through the introduction of kinds, which we can regard as being "the type of types".

Say we have function `f : int @ shared -> bool @ shared` that returns `true` if and only if the supplied argument is positive. If we have a naive uniqueness typing system, an application of `f` would require its parameter to be always `shared`. Therefore, the programmer would be prohibited from passing a `unique` integer, making the type system inflexible. This issue can be resolved if we allow `unique` values to be passed where `shared` are required. Hence, subtyping $unique \leq shared$ would need to be introduced, a concept formalised in an earlier work of De Vries, Plasmeijer, and Abrahamson 2007. However, supporting subtyping relation complicates the type system when considering further extensions. Therefore, de Vries et al. 2008 suggest the introduction of uniqueness polymorphism to avoid subtyping at all. Uniqueness polymorphism allows the programmer to rewrite the example function as `f : int @ 'u -> bool @ shared`, solving the problem of the naive typing system.

Section §3.5 will further explore these concepts where we present how uniqueness attributes have been added to FipML's type system and how the type inference algorithm has been modified to accommodate for this.

2.3.4 Calculus for fully in-place functions

The novelty of FipML is that it provides automatic, fully in-place executions of functions that are safe to do without any run-time overhead. The calculus FIP^S under which a function is optimisable is defined by Lorenzen, Leijen, and Swierstra 2023, representing the core theoretical foundation of the dissertation. The paper presents a carefully designed calculus under which function arguments ownership and constructors sizes are tracked, making precise whether a function is **fip** compliant or not at compile time. I modified the FIP^S calculus to support FipML operational semantics. Section §3.6 presents the modified calculus in greater detail, as well as the implementation details behind the static checker.

2.3.5 Lambda - OCaml's backend

In the project proposal, I suggested using Flambda or Malfunction (Dolan 2016) to target FipML intermediate representation to OCaml since they provide a way of generating efficient native code. Both proved to be difficult to work with. By analysing Malfunction's codebase, I discovered that Malfunction cannot read and set mutable fields of memory blocks, critical functionalities for reusing allocated memory. After further research, the project supervisor suggested I use Lambda, one of OCaml's IRs, which proved successful. Due to the lack of widely available resources and examples, connecting FipML to OCaml's backend proved not to be as straightforward as initially pictured. The implementation details of converting FipML's final IR to Lambda are presented in section §3.7.

2.4 Software development practices

I used Dune (Street 2024) as the default build system, which has greatly aided in creating FipML's codebase. Most importantly, Dune helped integrate libraries and packages as dependencies without much manual configuration and overhead. Additionally, Dune incorporates testing into the build process, allowing me to effortlessly execute large test suites §2.4.3.

2.4.1 Development plan

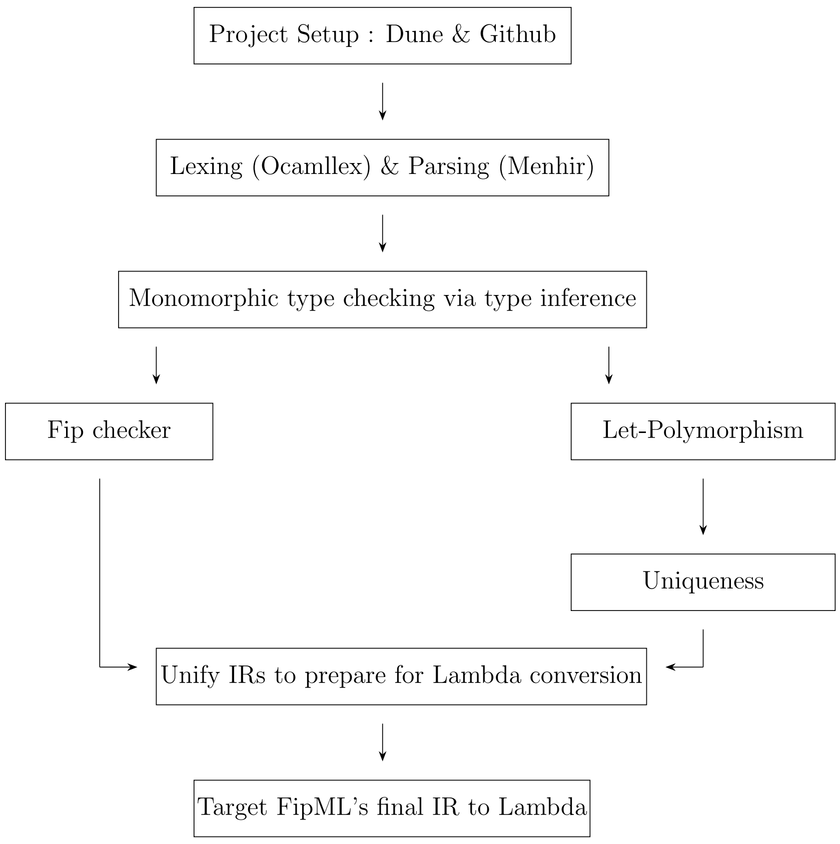

Project Setup : Dune & Github
Lexing (Ocamllex) & Parsing (Menhir)
Monomorphic type checking via type inference
Fip checker
Let-Polymorphism
Uniqueness
Unify IRs to prepare for Lambda conversion
Target FipML's final IR to Lambda

Figure 2.1: Development plan

I divided the project into achievable smaller components, as in Figure 2.1, resembling the key points previously described in section §2.3. Each node in the figure describes one deliverable, with arrows representing dependencies between key components. The development path differed slightly from the one initially outlined in the project proposal. I had to complete one of the extensions (namely adding polymorphism) before implementing the uniqueness typing system. By having a clear overall picture of the project, I managed to sequentially approach each task without being stuck for long.

2.4.2 Version control

I used GitHub as the primary version control platform and deployed industry-standard workflow. Firstly, I would create an issue by highlighting the problem or enhancing it.

Next, I add a new branch to the repository, enabling me to work on multiple features concurrently when necessary. Finally, I would merge each branch into `master` and close it, keeping the repository clean. Furthermore, I ensured a clear history of the development process by providing over 250 detailed commit messages across 17 issues.

2.4.3 Testing

Testing has been a crucial part of the development of FipML, with each component tested in both isolation and integration through "expect tests", using customised pretty-printers for each intermediate representation (IR) of the code inside the compiler. Tests have assured the correctness of each transformation and the entire workflow. Since development's early days, I enabled continuous integration testing through the command `dune runtest`. Below, the reader may find some testing metrics highlighting the degree to which I tested the implementation code.

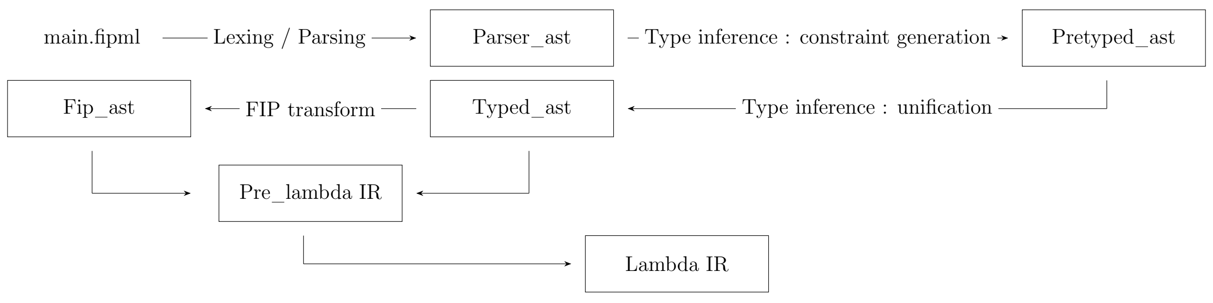
Tests	**Test code lines**	**Test / implementation lines ratio**
40	5304	0.25

Chapter 3

Implementation

This chapter presents the implementation details, typing rules, and calculus residing at the core of FipML. Section §3.1 provides an overview of FipML's pipeline, introducing all the intermediate representations used within the compiler and the execution flow, aiding the reader in navigating the remaining sections. Next, §3.2 introduces FipML's syntax, followed by §3.3 to explore how I implemented the grammar using Ocamllex and Menhir, therefore turning a file with `.fipml` extension into the `Parser_ast` intermediate representation. Section §3.3 introduces FipML's initial typing system supporting Damar-Hindley-Milner polymorphism and explains how typechecking has been implemented. In §3.5, I extend the previously defined type system with uniqueness attributes, therefore modifying the typechecking algorithm. Section §3.6 presents the calculus behind the static checker that enables FipML to perform an automatic transformation to allow in-place execution of annotated functions. Next, §3.7 describes how FipML's frontend is connected with OCaml's backend using Lambda IR as an entry point, enabling code execution. Lastly, §3.8 describes FipML's repository overview.

3.1 FipML pipeline overview



3.2 FipML syntax

I chose a simplistic syntax that is straightforward to analyse but still expressive enough to implement exciting programs. This section presents FipML's syntax, describing how users can define custom types and functions. The syntax is heavily influenced by the work of Lorenzen, Leijen, and Swierstra 2023, with a couple of additions by me, namely `if`-statements, binary and unary operators.

$$
\begin{array}{llll}
v & ::= & & \text{value} \\
 & \mid & u & \text{(unit)} \\
 & \mid & n & \text{(integer)} \\
 & \mid & b & \text{(boolean)} \\
 & \mid & x,\ y & \text{(variable)} \\
 & \mid & C^k\ (v_1,\ \ldots,\ v_k) & \text{(constructor)}
\end{array}
\qquad
\begin{array}{llll}
p & ::= & & \text{pattern} \\
 & \mid & \_ & \text{(underscore)} \\
 & \mid & x & \text{(variable)} \\
 & \mid & C^k\ (p_1,\ \ldots, p_k) & \text{(constructor)}
\end{array}
$$

$$
\begin{array}{lllr}
e & ::= & & \text{expression} \\
 & \mid & v & \text{(unboxed singleton)} \\
 & \mid & (v,\ \ldots,\ v) & \text{(unboxed tuple)} \\
 & \mid & \mathit{unary\_op}\ e & \text{(unary)} \\
 & \mid & e\ \mathit{binary\_op}\ e & \text{(binary)} \\
 & \mid & \textbf{if}\ e\ \textbf{then}\ \{e\}\ \textbf{endif} & \text{(if)} \\
 & \mid & \textbf{if}\ e\ \textbf{then}\ \{e\}\ \textbf{else}\ \{e\}\ \textbf{endif} & \text{(if - else)} \\
 & \mid & e\ \overline{v} & \text{(application)} \\
 & \mid & f\ \overline{v} & \text{(call)} \\
 & \mid & \textbf{let}\ \overline{x} = e\ \textbf{in}\ e & \text{(let)} \\
 & \mid & \textbf{match}\ x\ \textbf{with}\ \overline{\mid p\ \rightarrow\ \{e\}} & \text{(match)} \\
 & \mid & \textbf{drop}\ x;\ e & \text{(drop)} \\
 & \mid & \textbf{free}\ n;\ e & \text{(free)} \\
 & \mid & \textbf{inst}\ n;\ e & \text{(inst)} \\
 & \mid & \textbf{weak}\ n;\ e & \text{(weak)}
\end{array}
$$

$\mathit{unary\_op}$ ::= `-` | `!` $\mathit{binary\_op}$ ::= `+` | `-` | `*` | `%` | `<` | `>` | `<=` | `>=` | `==` | `!=` | `&&` | `||`

I have explicitly omitted type and function definitions from the grammar because I believe it is best to understand them by looking at examples, as their syntax resembles that of OCaml.

```
type simple_custom_type =
| Atom
| SimpleConstructor of int @ unique * bool @ shared
```

Listing 1: Non-polymorphic custom type definition

Because FipML has a uniqueness typing system at its core, the programmer must annotate each partial type with a uniqueness attribute. The available uniqueness attributes are `shared`, `unique` and polymorphic in uniqueness. For example, in Listing 1, `simple_custom_type` has two constructors, `Atom`, and `SimpleConstructor`, which takes as argument a unique integer and a shared boolean.

Similarly, we may define a polymorphic custom type in Listing 2. Such a definition takes precisely three types of polymorphic parameters, delimited in the definition by the usage of @. The first set of polymorphic parameters represents partial types, the second set represents uniqueness attributes, and the last set allows for the programmer to pass fully initialised types (partial type annotated with uniqueness attribute).

```
type ('t @ 'u1, 'u2 @ 'a) poly_custom_type =
| PolyAtom
| PolyConstructor of 't @ 'u1 * 'a * simple_custom_type @ 'u2

(* Therefore, the following expression has type
       (int ; shared, unique ; (bool @ unique)) poly_custom_type *)
PolyConstructor (int @ shared, bool @ unique, Atom @ unique)
```

Listing 2: Polymorphic custom type definition

From now on, as a syntactic convention, we will use $'t_\alpha$ for partial types, $'u_\beta$ for uniqueness attributes and $'a_\gamma$ for fully initialised types. Section §3.4 presents an in-depth discussion about FipML's types.

Respecting the syntax introduced so far, it is easy to define custom function definitions:

```
(and) fun my_f (x : 't @ shared) ^(y : simple_custom_type @ 'u) : 'a {
    ...
}
```

Listing 3: Custom function definition

Note that FipML requires each function signature to be explicitly typed by the programmer. Listing 3 is designed to illustrate the syntax. We note the usage of polymorphic parameters ($'t$ for partial type, $'u$ for uniqueness attribute and $'a$ for fully initialised type) in the correct positions and the capabilities of the language to define mutually recursive functions by using the `and` keyword. Furthermore, observe that `y` is annotated with `^`, marking it as *borrowed*. We will explore this concept further in section §3.6.

3.3 Lexing and parsing

Parsing is split into two parts (lexing and actual parsing). Lexing, the process of converting the input text into a stream of tokens, is achieved using Ocamllex. The full implementation of the lexer can be found in Appendix A.1. The order in which the regular expressions appear in the `token` function required attention. To illustrate this challenge, we should look at an example.

Say the input buffer contains the text `"||"`. By having the following *incorrect* implementation of the `token` function:

```
rule token = parse
...
| "|" -> { BAR }
| "||" -> { OR }
```

the lexer would produces `BAR BAR` rather than `OR`.

File `src/frontend/parsing/parser.mly` provides a declarative syntax for specifying the grammar Menhir uses to generate an LR(1) parser. The full implementation of the parser is too large to be included in the dissertation, and the reader is encouraged to explore the accompanying code by using section §3.8 to quickly identify the file. Appendix A.2 presents a short snippet, following the syntax given in §3.2.

We combine the lexer and parser to provide a working flow of converting the input file into the `Parser_ast` intermediate representation.

```
let parse_source_code_with_error (lexbuf : Lexing.lexbuf) :
    Parser_ast.program Or_error.t =
  try Ok (Parser.program Lexer.token lexbuf) with
  | Lexer.LexerError lexer_error ->
      let lexer_error_string =
        lexer_error ^ " - " ^ Ast.Ast_types.string_of_loc lexbuf.lex_curr_p
      in
      Or_error.of_exn (LexerException lexer_error_string)
  | Parser.Error ->
      let parsing_error_string =
        "Syntax Error - " ^ Ast.Ast_types.string_of_loc lexbuf.lex_curr_p
      in
      Or_error.of_exn (ParserException parsing_error_string)
```

The above method takes the input file as a `Lexing.lexbuf` buffer and produces a `Parser_ast.program`. We note the usage of the `Result` monad from `Core` to accurately display any errors that might occur during parsing, aiding the programmer in repairing the syntax issue faster.

I created pretty-printers for debugging and testing since complicated programs may have a complex `Parser_ast` intermediate representation. Furthermore, to validate the parsing process, I have added tests to the repository in `test/frontend/lexer` and `test/frontend/parser`.

3.4 Typechecking programs

Typechecking is one of FipML's core components. FipML deploys a Damas-Hindley-Milner type system for its partial types. Partial types are fully initialised types but without the uniqueness attribute. For example, both `custom_type` and `int` are valid partial types. This section will discuss the available partial types in FipML and explain how the compiler typechecks programs.

I have divided the discussion about typechecking programs into three distinct components: typechecking custom type definitions §3.4.1, performing type inference on arbitrary expressions while allowing let-polymorphism §3.4.2 and typechecking custom function definitions §3.4.3. While typechecking the program, FipML's internal representation of the source code will be transformed from `Parser_ast` into `Typed_ast`, now with each node of the abstract syntax tree being annotated with a partial type.

3.4.1 Typechecking custom type definitions

We start by presenting the available partial types, defined in `ast_types.ml` :

```
type type_expr =
  | TEUnit of loc
  | TEInt of loc
  | TEBool of loc

  (* Means of encoding polymorphic type variables by their name *)
  | TEPoly of loc * string

  (* Custom type definitions may be parameterised by TEPolys. *)
  | TECustom of loc * type_expr list * Type_name.t
  | TEArrow of loc * type_expr * type_expr
  | TETuple of loc * type_expr list
```

Typechecking custom type definitions is relatively complicated, as we should allow recursive algebraic data types. The code responsible for this subsection is available in `src/frontend/typing/typecheck_type_defns.ml`. Here is the signature of the relevant function:

```
val typecheck_type_defns :
  Parsing.Parser_ast.type_defn list ->
  (Type_defns_env.types_env
  * Type_defns_env.constructors_env
  * Typed_ast.type_defn list)
  Or_error.t
(** Typechecks custom type definitions, returns if successful *)
```

The function responsible for typechecking custom type definitions takes as argument a list of `Parser_ast.type_defn` and returns a tuple containing three important data structures:

- `types_env` is a list containing an entry for each user-defined type. Here is an example `types_env`:

```
[
  ([TEPoly (_, "'a"); TEPoly (_, "'b")],
    Type_name.of_string "complex_type");
  ([], Type_name.of_string "simple_type")
]
```

  Here, `types_env` tells us that the programmer has defined only two custom types, a polymorphic one (`complex_type`, where $'a$ and $'b$ are polymorphic type variables) and a monomorphic one (`simple_type`). Exposing polymorphic type variables in such a manner is, by design, aiding in typechecking the program where such user-defined custom type appears.

- `constructors_env` is a list containing an entry for each constructor part of the user-defined types. Here is an example:

  ```
  [
    (
      Type_name.of_string "simple_type",    // constructor's type
      "SimpleAtom", // constructor name
      [] // constructor arguments
    );
    (
      Type_name.of_string "complex_type",
      "ComplexConstructor",
      [TECustom (_, [], "simple_type"];
       TECustom (_, ["'b"; "'a"], "complex_type")
    )
  ]
  ```

  `constructors_env` accumulates all the user-defined constructors, keeping track of their type names and the required arguments' types. In the above example, we see that the program only has two custom constructors: `SimpleAtom` of type `simple_type`, taking no arguments and `ComplexConstructor` which requires two arguments of types `simple_type` and `('b, 'a) complex_type`.

- `Typed_ast.type_defn list` accumulates all the type definitions converted to the `Typed_ast` intermediate representation.

The algorithm for typechecking custom type-definitions iterates over them in the order in which they appear in `Parser_ast`. We may outline the algorithm when evaluating one type-definition step by step:

1. Check that the polymorphic type variables are unique. We should reject programs such as

   ```
   type ('a, 'a, 'b) incorrect_type = ...
   ```

2. Assert that the programmer did not previously define another type sharing the same name as the one we are analysing by checking inside `types_env`.

3. Add entry of the current type to `types_env` so we allow recursive type definitions, such as

   ```
   type 'a list = Nil | Cons of 'a * 'a list
   ```

4. For each constructor within the current type definition, assert that no other constructor sharing the same name is already in `constructors_env`.

5. Typecheck each constructor's argument. To understand this step, it is best to look at an example:

```
type 'a custom_type = C of 'a * (int -> 'a) * bool custom_type
```

This step first verifies that $'a$ is well-defined (and it is because it is a bounded type variable from the custom type definition). Next, it checks each component of the `int ->`$'a$ type. Finally, the process asserts that `bool custom_type` is also well-defined. This is done by checking that `custom_type` belongs in the `types_env` and is applied with the correct number of polymorphic types (in this example, just `bool`). Recursively, the process also checks that all applied polymorphic types are well-defined.

3.4.2 Type inference

This section presents the typing rules and implementation of let-polymorphism type inference without accounting for uniqueness attributes yet.

The type inference algorithm has two steps: *constraint generation* and *unification*. A *constraint* is an equality equation between two type expressions (t_α, t_β), potentially including type variables. When typechecking an expression, the algorithm records constraints about the types of the subexpressions instead of trying to assert them on the spot.

The constraint generation step of the algorithm creates a set of equations between types. For example, the set of constraints $\{(t_0, int), (t_1, t_0)\}$ encode that t_0 needs to have type int, and that t_1 should be equal to t_0. These equations have to simultaneously hold for the program to be well-typed. The constraint typing relation $\Gamma \vdash e : t \dashv C$ is defined below, and the reader should interpret it as "term e has type t under the typing context assumptions Γ given the constraint set C is satisfiable".

I derived FipML's constraint typing relations by extending and modifying those of a regular Hindley-Milner type system (Pierce 2002), with particular care for enabling let-polymorphism (Milner 1978). Appendix B contains the full definition of the constraint typing relations, while below I present a subset of them:

$$\frac{x : \forall \alpha_1, ..., \alpha_n.t \in \Gamma}{\Gamma \vdash x : \text{inst}(\forall \alpha_1, ..., \alpha_n.t) \dashv \emptyset}\ \text{Variable}$$

$$\frac{\Gamma \dashv e_1 : t_1 \dashv C_1 \qquad \text{generalise}(C_1, \Gamma, \overline{x} : t_1) \vdash e_2 : t_2 \dashv C_2}{\text{let } \overline{x} = e_1 \text{ in } e_2 : t_2 \dashv C_1 \cup C_2}\ \text{Let}$$

$$\frac{\Gamma \vdash e_1 : t_1 \dashv C_1 \qquad \Gamma \vdash e_2 : t_2 \dashv C_2 \qquad \Gamma \vdash e_2 : t_3 \dashv C_3 \qquad t = \text{fresh}()}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t \dashv \{(t_1, \text{bool}), (t, t_3), (t_2, t_3)\} \cup C_1 \cup C_2 \cup C_3}\ \text{IfElse}$$

The `Variable` and `Let` rule contain methods to enable let-polymorphism to which we will come back later in this section. The `IfElse` rule is best for understanding the general idea behind constraint generation. The algorithm first generates constraints for each expression e_1, e_2 and e_3. Generally, an `if`-statement is well-typed if the condition expression has type `bool` and if both branches have the same type. This constraint is encoded in $\{(t_1, \text{bool}), (t, t3), (t_2, t_3)\}\}$, where t is the type of the entire `if`-statement.

Next, we combine the previous constraints with C_1, C_2 and C_3, yielding the final set of constraints for the entire expression.

FipML does not allow variable shadowing. Hence, the following code is not valid because it defines `x` twice in the same scope:

```
let x = 1 in let x = 2 in x
```

Therefore, each variable appears precisely once inside the typing context Γ.

Unification is the process of solving the set of previously generated constraints C. The method works by calculating a substitution σ from a type variable to a CHANGE base type (or another type variable) that unifies all the constraints - i.e., $\sigma t_\alpha = \sigma t_\beta, \forall (t_\alpha, t_\beta) \in C$.

The `generalise` and `inst` methods enable let-polymorphism. Benjamin Pierce 2002 (Pierce 2002) describes an efficient way to typecheck let-expressions such as $\Gamma \vdash \text{let } x = e_1 \text{ in } e_2$. In the constraint generation implementation for `Let` expressions, I have a section of code corresponding to each of the following steps:

1. Compute t_1, C_1 by running type inference on e_1.

2. Run unification to find the substitution σ satisfying all constraints in C_1 and apply σ to both t_1 and Γ to determine the principal type T_1 of e_1.

3. Generalise any type variables in T_1. If $X_1, ..., X_n$ are the type variables remaining in T_1, we encode the principal type scheme of e_1 as $\forall X_1, ..., X_n.T_1$. In this step, we have to be careful not to generalise type variables that are also free in Γ, as previous constraints might have made assumptions about these type variables already.

4. Extend the typing context Γ to record the type scheme of the bounded variable x.

5. Perform type inference on the body of the `let` expression, e_2.

   ```
   generate_constraints types_env constructors_env functions_env
       extended_typing_context expr ~verbose
   >>= fun (_, expr_type, expr_constrs, expr_substs, pretyped_expr) ->
   unify_substs expr_substs var_substs >>= fun expr_substs ->
   unify_substs expr_substs var_pretyped_substs >>= fun expr_substs ->
   ```

6. Each time the variable x is used inside e_2, we instantiate the type scheme $\forall X_1, ..., X_n$ of x with fresh type variables $(Y_1, ..., Y_n)$ and use $[X_1/Y_1, ..., X_n/Y_n]T_1$ as the type of x.

For implementing type inference via constraint-based typing as previously defined, I defined the following interface, where `ty` is the type of any expression evaluated during constraint generation.

```
type ty =
  | TyVar of string
  | TyUnit
```

```
  | TyInt
  | TyBool
  | TyPoly of string list * ty
  | TyCustom of ty list * Type_name.t
  | TyArrow of ty * ty
  | TyTuple of ty list

type subst = string * ty
type constr = ty * ty
type typing_context = ty Type_context_env.typing_context

(** Create fresh type variable TyVar *)
val fresh : unit -> ty
```

Note that `TyVar` encodes a type variable parameterised by its identifier/name. `TyPoly` is a wrapper placed around another `ty` used to describe the type scheme of a term. The reader should observe the correspondence between the `fresh` function in the code above and its presence in the typing rules, as it provides an easy way to create new type variables. `subst` is encoding an entry in the solution substitution σ and `constr` encodes a constrain between two `tys`.

Below, we can see the signatures of the `generalise` and `inst` methods.

```
val generalise : typing_context -> Var_name.t -> ty -> typing_context
val instantiate : Var_name.t -> typing_context -> ty
```

The `generalise` method generalises variables that are free in the given `ty` and also not bounded by any previous assumptions made in the `typing_context`. For example, for the given expression $x : t_1 \vdash e : (t_1, t_2)\,\text{custom\_type}$, the method should the expression into $x : t_1 \vdash e : \forall t_2.(t_1, t_2)\,\text{custom\_type}$. On the other hand, `instantiate` substitutes all polymorphic type variables with fresh types. Therefore, expression $x : t_1 \vdash e : \forall t_2.(t_1, t_2)\,\text{custom\_type}$ would be transformed into $x : t_1 \vdash e : (t_1, t')\,\text{custom\_type}$, where t' is a fresh variable.

Having all these methods in place, implementing the typing relations previously defined for generating constraints is not challenging, but it involves lots of code (1071 lines). Therefore, analysing the entire file would be inefficient. Hence, we should look at some code that generates constraints for let-expressions. We may outline the regions of code corresponding with each of Pierce's algorithm.

Here is the function signature for the `generate_constraints` method

```
val generate_constraints : Type_defns_env.types_env ->
  Type_defns_env.constructors_env -> Functions_env.functions_env ->
  typing_context -> Parsing.Parser_ast.expr ->
  verbose:bool ->
  (typing_context * ty * constr list * subst list * Pretyped_ast.expr) Or_error.t
```

which upon examination, we observe it returns a couple of data structures:

- `typing_context` of the current analysed expression,
- `ty` type of the current expression,
- `constr list` aggregating all the constraints generated by recursive calls on sub-expressions of the current expression,
- `subst list` is a set of partial substitutions that have been created during step 2 of Pierce's algorithm,
- `Pretyped_ast` is an intermediate representation where each expression is annotated with the `ty` type computed during constraint generation. This `ty` acts as a placeholder for the actual FipML type, which is not yet known. The sole purpose of this intermediate representation is to provide the compiler with an easy way of updating the type of every expression within the program after type inference finishes, and we have a solution satisfying all constraints.

Furthermore, having the `verbose` parameter in the `generate_constraints` function allowed me to easily debug this complex procedure and create expect-tests to verify its functionality.

Next, we look at the unification step. The procedure analyses each constraint sequentially, trying to unify one equality expression at a time. The implementation is standard as described in Algorithm W (Milner 1978). For example, unifying a type variable `TyVar` with any other `ty` we perform the following routine:

```
let rec unify (constraints : constr list) (substs : subst list) =
...
    | (TyVar type_var, ty) :: constraints when not (occurs type_var ty) ->
        let ts = ty_subst [ (type_var, ty) ] in
        unify
          (List.map constraints ~f:(fun (ty1, ty2) -> (ts ty1, ts ty2)))
          ((type_var, ty) :: List.map substs ~f:(fun (ty1, ty2) -> (ty1, ts ty2)))
```

The routine first checks that the `TyVar` does not occur in `ty` (occurs check), as this would lead to infinite loops or non-termination. Then, the algorithm creates a substitution from the `TyVar` to `ty` and applies the substitution to all remaining constraints and the solution of the previous constraints.

Having defined the algorithm for constraints generation and unification, we may connect them to complete type inference. We first generate the typing constraints for a given expression via `generate_constraints`, which we later solve using `unify`. The substitution solution is applied to the `Pretyped_ast` created during constraint generation, yielding the final `Typed_ast` intermediate representation.

3.4.3 Typechecking custom function definitions

We start by analysing the signature of the function responsible for typechecking function definitions.

```
val typecheck_functions_defns :
  Type_defns_env.types_env ->
  Type_defns_env.constructors_env ->
  Parsing.Parser_ast.function_defn list ->
  (functions_env * Typed_ast.function_defn list) Or_error.
(** Typechecks function definitions, returns if successful *)
```

The function takes three arguments `types_env`, `constructors_env` in order to allow functions to use custom-defined types and `Parser_ast.function_defn` list and returns two important data structures:

- `functions_env` is a list containing an entry for each user-defined function. One such entry has the type

  ```
  type function_env_entry =
    | FunctionEnvEntry of
        int
        * fip option
        * Function_name.t
        * type_expr list
        * borrowed option list
        * type_expr
  ```

  encoding the function's mutually recursive group identifier, **fip** annotation status - further explored in section §3.6, the name of the function, types of the required arguments, the borrowed status of the arguments and the return type of the function.

  This data structure will be essential to typechecking the entire program, especially in type inference, where we allow function calls and have to check that the arguments are of the correct type and that the function's return type is consistent with the application's context.

- `Typed_ast.function_defn` list accumulates all the user-defined function definitions converted to the `Typed_ast` intermediate representation.

We may outline the algorithm for typechecking custom function definitions as follows:

1. Verify that the function's signature types are well-defined against `types_env`. A type is well-defined if it is a basic type (`int`, `bool`, `unit`), a custom-defined type or a tuple/function of well-defined types.

2. Extend `functions_env` to include an entry for each function from the mutually recursive group of the current one. This is done only once when we analyse the first function in the group. We can do this because we assume that all functions within the mutually recursive group will typecheck, and an error will be thrown if any of them do not.

3. Instantiate all the types for the function's signature. FipML supports prenex polymorphism rather than first-class. Therefore, this procedure creates a mapping from type parameters to a concrete replacement types `ty`. An example is best for understanding the necessity of this critical step.

   Say we have a custom-defined function with the following signature :

   ```
   fun map_f (xs : 'a list) (f : 'a -> 'b) : 'b list = { ... }
   ```

   Then we will create a mapping $\{a \mapsto \text{TyVar}(t_0);\ b \mapsto \text{TyVar}(t_1)\}$, modifying the function's signature into :

   ```
   fun map_f (xs : TyVar(t0) list) (f : TyVar(t0) -> TyVar(t1))
       : TyVar(t1) list = { ... }
   ```

4. Create the initial typing context Γ of the function's body. The typing context will have an entry for each function argument and their associated type `ty`.

   Following on the previous example, this initial typing context would be $\{xs \mapsto \text{TyVar}(t_0)\ list;\ f \mapsto \text{TyVar}(t_0) \rightarrow \text{TyVar}(t_1)\}$.

   ```
   let function_typing_context : typing_context =
     List.map function_params
       ~f:(fun
           (Ast.Ast_types.TParam (
             function_param_type,
             function_param_var, _)) ->
         Type_context_env.TypingContextEntry
           ( function_param_var,
             convert_ast_type_to_ty
                 function_param_type
                 type_scheme_assoc_list ))
   ```

   Subsection §3.4.2 presented the type inference algorithm and typing rules. We note the usage of the `type_scheme_assoc_list` (the mapping defined in step 3) parameter in the `convert_ast_type_to` function as being used in converting the polymorphic argument's type into a `ty` type.

5. Perform type inference on the function's body using Γ. Variable `function_typing_context` is responsible for encoding the typing context Γ.

   ```
   type_infer types_env constructors_env extended_functions_env
     function_typing_context function_body
   ```

6. Check that type inference returns successfully and verify that the type of the function's body matches the return type. The compiler converts back from the types `ty` used by the type inference algorithm into the regular `ast_type` to check for equality.

3.4.4 Typechecking the entire program

We may now typecheck entire FipML programs. First, we check custom type definitions, computing `types_env` and `constructors_env`. These are later passed to typecheck custom function definitions, synthesising `functions_env`. With everything in place, we can run type inference on the main expression. Connecting these stages, we successfully transform `Parser_ast` into `Typed_ast`, assuring our programs are well-typed.

3.5 Uniqueness typing

FipML deploys a uniqueness typing system to determine whether a function can be executed fully in-place at compile-time. This section explains how I implemented FipML's uniqueness type system based on the work of De Vries, Plasmeijer, and Abrahamson 2008. It also covers the implementation details of how I adjusted the existing type inference algorithm to facilitate uniqueness inference.

Previous work of De Vries, Plasmeijer, and Abrahamson 2008 shows a methodology of extending any existing typing system to track uniqueness attributes of various variables and expressions. I adapted de Vries et al.'s 2008 uniqueness rules to fit the requirements of FipML.

Variables annotated as `unique` have the guarantee that they are used precisely once throughout their definition scope. They represent resources consumed upon use, such as file handles. On the other hand, `shared` variables allow multiple references to the same underlying resource.

One of the key points of de Vries et al. 2008 work is to treat unique attributes as types through the introduction of kinds. Therefore, from now on, a FipML type is regarded as a partial type $\mathcal{T}$ (`int`, `bool`, `custom_type`, `int@unique` $\rightarrow$ `int@unique`) annotated with a uniqueness attribute $\mathcal{U}$ (`unique`, `shared`, or polymorphic in uniqueness). Therefore, FipML's fully initialised types `ast_type` will now have the following form:

```
(* Types of expressions in FipML *)
type poly = Poly of loc * string
type uniqueness = Unique of loc | Shared of loc | PolyUnique of loc * poly

type typ =
  | TEUnit of loc
  | TEInt of loc
  | TEBool of loc
  | TEPoly of loc * poly
  | TECustom of loc * typ list * uniqueness list * type_expr list * Type_name.t
  | TEArrow of loc * type_expr * type_expr
  | TETuple of loc * type_expr list

and type_expr = TAttr of loc * typ * uniqueness | TPoly of poly
```

Note the separation of `Poly` constructor such that I may use a common structure for polymorphic type variables at all levels, partial types (`TEPoly`), uniqueness attributes (`PolyUnique`) or fully initialised types (`TPoly`).

The uniqueness typing rules present in FipML rely on a "sharing analysis" to annotate variable uses with a uniqueness attribute ($\odot$ for unique usage and $\otimes$ for shared usage). The algorithm behind the sharing analysis is straightforward: count how often each variable is used within an expression. When defining a variable through a function definition, `match`- or `let`-expression, the uniqueness type system checks that the variable's uniqueness attribute complies with the result of the sharing analysis on the variable's scope. To understand sharing analysis better, we should look at an example:

```
fun dup (x : 't @ unique) : ('t @ shared * 't @ shared) @ 'u = {
    (x, x)
}
```

In the above snippet, while the programmer has annotated `x` as `unique` in the signature of the function `dup`, the uniqueness type system will reject this program because the sharing analysis on the function's body discovers that `x` has been used more than once, hence used as `shared`.

I implemented sharing analysis by modelling variables' usages in an expression as a map from variable name to number of usages.

```
module SharingAnalysisMap : Map.S with type Key.t = Var_name.t
```

Sequential expressions should add the counts of all variables

```
let join_sharing_analysis_map
    sharing_analysis_map1 sharing_analysis_map2 =
  Map.merge_skewed sharing_analysis_map1 sharing_analysis_map2
    ~combine:(fun ~key:_ v1 v2 -> v1 + v2)
```

while branching expressions should take the maximum usage count

```
let join_max_sharing_analysis_map
    sharing_analysis_map1 sharing_analysis_map2 =
  Map.merge_skewed sharing_analysis_map1 sharing_analysis_map2
    ~combine:(fun ~key:_ v1 v2 -> Int.max v1 v2)
```

The reader can find the full implementation of generating the sharing analysis map of an expression in Appendix C.1. The following code snippet should give main idea:

```
let rec get_sharing_analysis (parsed_expr : Parser_ast.expr)
  : int SharingAnalysisMap.t =
  match parsed_expr with
  ...
  | IfElse (_, expr_cond, expr_then, expr_else) ->
      join_sharing_analysis_map
```

```
        (join_max_sharing_analysis_map
           (get_sharing_analysis expr_then)
           (get_sharing_analysis expr_else))
        (get_sharing_analysis expr_cond)
  ...
```

Now that the sharing analysis has annotated each variable with a usage attribute, we describe the uniqueness typing rules at the core of FipML. In the implementation, I have modified the type inference algorithm §3.4.2 to perform uniqueness inference as well. Note that the uniqueness rules do not generate constraints regarding partial types $\mathcal{T}$, as it assumes that the type inference algorithm already records them. Following de Vries et al. 2008, we use s, t for partial types, u, v for uniqueness attributes and a, b for fully initialised types. For the complete rules, refer to Appendix C.3. Below, we present a subset of the uniqueness typing rules of FipML.

$$\frac{}{\Gamma, x : t^u \vdash x^{\odot} : t^u \dashv \emptyset}\ \text{Var}^{\odot} \qquad \frac{}{\Gamma, x : t^{\times} \vdash x^{\otimes} : t^{\times} \dashv \emptyset}\ \text{Var}^{\otimes}$$

$$\frac{\Gamma \vdash e_1 : t_1^{u_1} \dashv C_1 \qquad \Gamma, x : t_1^{u_1} \vdash e_2 : a \dashv C_2 \qquad u_2 = \text{sharing\_analysis}(x, e_2)}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : a \dashv \{(u_1, u_2)\} \cup C_1 \cup C_2}\ \text{Let}$$

$$\frac{\Gamma \vdash e_1 : \text{bool}^{u_1} \dashv C_1 \qquad \Gamma \vdash e_2 : t^{u_2} \dashv C_2 \qquad \Gamma \vdash e_3 : t^{u_3} \dashv C_3 \qquad u = \text{fresh\_unique}()}{\Gamma \vdash \text{if}\, e_1 \,\text{then}\, e_2 \,\text{else}\, e_3 : t^u \dashv \{(u, u_2), (u_2, u_3)\} \cup C_1 \cup C_2 \cup C_3}\ \text{IfElse}$$

Notice how a variable has two distinct rules depending on its usage attribute. It is apparent in the `Var`$^{\odot}$ rule that we annotate uniquely used variables as polymorphic in uniqueness rather than as `unique`. Using t^u instead of $t^{\odot}$, the type system allows passing `unique` values where `shared` ones are required. On the other hand, rule `Var`$^{\otimes}$ always annotates `shared` variable as `shared` - $t^{\times}$.

The `Let` rule is very similar to the one for principal types, except we add a uniqueness constraint to verify that the sharing analysis of variable `x` computed from e_2 complies with the uniqueness attribute of the expression e_1. For example, the type system rejects programs such as

```
fun f : unit @ 'u -> int @ unique
let x = f() in (x, x)
```

because while `x` is `unique`, it is used in a `shared` way in the body of the `let`-expression. The `If` rule records uniqueness constraints similarly to the constraint typing relation from §3.4.

Lastly, let-polymorphism is not applied to uniqueness attributes because the sharing property of a variable is static and does not vary with the context of usage. Not allowing let-polymorphism for uniqueness attributes is apparent in the `Let`, `Var`$^{\odot}$ and `Var`$^{\otimes}$ rules, where we do not make use of the `generalise` and `inst` methods as we did in §3.4.

For the implementation, I modified the internal types used by the type inference algorithm (and `Pretyped_ast` IR) to accommodate uniqueness attributes.

```
type ty_unique = TyVarUnique of string | TyShared | TyUnique

type ty =
  | TyVar of string
  | TyUnit
  | TyInt
  | TyBool
  | TyPoly of string list * ty
  | TyCustom of ty list * ty_unique list * ty_attr list * Type_name.t
  | TyArrow of ty_attr * ty_attr
  | TyTuple of ty_attr list

and ty_attr = ty * ty_unique
...
type subst_unique = string * ty_unique
type constr_unique = ty_unique * ty_unique
type typing_context = ty_attr Type_context_env.typing_context
```

Because of the introduction of kinds, we now have a mix of partial types, uniqueness attributes, and fully initialised types, which I decided to track individually. The reader should observe how FipML generates constraints for partial types and uniqueness attributes separately, keeping them apart. Furthermore, the constraint generation algorithm does not create constraints on fully initialised types (`ty_attr`) but rather splits them into a `constr` for `ty` and `constr_unique` for `ty_unique`. The signature of the `generate_constraints` method is

```
val generate_constraints : Type_defns_env.types_env ->
  Type_defns_env.constructors_env -> Functions_env.functions_env ->
  typing_context -> Parsing.Parser_ast.expr -> verbose:bool ->
  ( typing_context * ty_attr * constr list * constr_unique list
    * subst list * Pretyped_ast.expr ) Or_error.t
```

The function returns:

- the `typing_context` of the given expression,
- the type of the expression as `ty_attr`,
- the generated constraints regarding partial types as `constr list`,
- the generated constraints regarding uniqueness attributes as `constr_unique list`,
- a set of substitutions of partial types, generated by Pierce's algorithm on `let`-expressions, as described in 3.4,
- the `Pretyped_ast` intermediate representation of the expression.

After modifying the constraint generation method to account for uniqueness attributes, I also updated the unification algorithm to perform unification on the uniqueness constraints. Furthermore, `ty` constraints might intrinsically generate further constraints on uniqueness attributes, as this is apparent in the definition of `ty` types such as the `TyCustom`, `TyArrow` and `TyTuple` constructors, as they take as arguments fully initialised types `ty_attr` (containing uniqueness properties). Appendix C.2 presents the code for unifying uniqueness constraints.

With methods in place for generating constraints on partial types `ty` and uniqueness attributes `ty_unique` and for unifying these constraints (`unify` and `unify_unique`), we may now connect them to finalise type inference for FipML's complete typing system:

1. Generate constraints on partial types and uniqueness attributes.
2. Unify the partial type constraints, possibly yielding extra uniqueness attribute constraints.
3. Unify any substitutions made during Pierce's algorithm with the resulting substitutions from step 2. This unification may yield further uniqueness constraints.
4. Combine all the generated uniqueness attribute constraints and unify them.
5. Construct the `Typed_ast` intermediate representation from the generated `Pretyped_ast` by applying the substitutions for partial types and uniqueness attributes.

The current section concludes all the implementation details of the type system presented in FipML, completing the pipeline that transforms the `Parser_ast` IR into the fully-typed `Typed_ast`. From now on, the compiler knows that the input program is well-typed.

3.6 Fully in-place conditions - fip and fbip

The previous sections of this chapter have presented the functionality requirements necessary for the compiler to determine whether performing in-place optimisations to specific functions is safe. This section presents the calculus and notions that make precise whether FipML can optimise a function. The concepts presented are inspired by the work of Lorenzen et al. 2023 (Lorenzen, Leijen, and Swierstra 2023) in their FIP^S calculus.

We say a function is fully in-place (**fip**) if the compiler can optimise it in a way that does not require any (de)allocation and which execution uses bounded stack space. FipML performs this static check on functions annotated with the **fip** keyword. Analogously, we may define a function to be functional but in-place, a concept introduced by Reinking et al. 2021 as a function similar to `fip` but which allows deallocation and arbitrary stack space usage. FipML performs a static check abiding by this calculus on functions annotated with the **fbip** keyword.

Any `fip` or `fbip` function may take two types of arguments:

- *owned* - the function exclusively controls the argument. Hence, the function is free to modify the underlying memory of the argument, potentially using it destructively.
- *borrowed* - the function does not have ownership of the underlying memory of the argument and hence cannot modify it.

Lorenzen et al. 2023 FIP^S calculus has the following properties:

- The calculus is second-order, meaning that we may pass functions as arguments but cannot return one as the result of any computation.
- Borrowed parameters cannot be used in a destructive manner, passed as owned arguments to function calls or returned as the result of a computation.
- Arbitrary lambda expressions are not allowed. Hence, any function defined has to be top-level.
- Tuples are unboxed.
- Atoms (constructors without arguments) do not require memory allocation.
- Destructive match-expressions precisely consume matched variables and create proportional reuse credits based on underlying block memory size.
- Reuse credits must be used linearly (precisely once), and splitting or merging such credits is not allowed.
- Owned variables must be used linearly.

Destructive match-expressions imply that the underlying memory of the matched variable may be reused for subsequent constructor allocations. An example is best for understanding this property, where we import the destructive match syntax from Lorenzen et al. 2023:

```
match! x with
| Cons (x, xs) -> ...
```

Here, the destructively matched constructor `Cons(x, xs)` creates owned variables `x` and `xs`, as well as a reuse credit $\diamond_2$. I have informally introduced this reuse credit $\diamond_k$ resource in section §2.3.4, representing a memory block of size k that may be reused for further construction allocation.

Lorenzen et al. 2023 also support bounded allocation in both their calculi for FIP^S and FBIP by using special keywords `fip(`n`)` and `fbip(`n`)`. For functions annotated with these keywords, FipML allows the function to potentially allocate n blocks of memory of any size, each specified by the programmer via the `inst(`k`)` instruction. Alternatively, if the programmer decides not to use k allocation credits $\diamond$, they should specify it by the `weak(`k`)` instruction. Allocation credits become apparent in the FIP^{ML} and FBIP^{ML} calculi, but an example is best for understanding this idea:

```
fip(1) fun append (x : 't @ 'u1) (xs : ('t ; ; ) list @ unique)
  : ('t ; ; ) list @ 'u2 = {
    inst (2); Cons (x, xs)
}
```

The above FipML code prepends an element to a list. Notice how we transform the only allocation credit $\diamond$ into a reuse credit $\diamond_2$ by using the `inst 2` instruction. We then linearly consume the reuse credit by allocating a constructor of size 2, namely `Cons (x, xs)`. In this example, it is also worth noting how the function linearly uses the owned environment $\Gamma = \{$`x, xs`$\}$, abiding by the properties of Lorenzen et al.'s 2023.

Figure 3.1 presents FipML's FIP^{ML} calculus responsible for performing the `fip` check, where $\Delta \,|\, \Gamma \,|\, \rho \vdash e$ means that given the borrowed Δ, owned Γ and reuse credit ρ environments, e is well-formed under FIP^{ML}. The calculus is adapted from Lorenzen et al. 2023 by making the reuse credit environment ρ explicit and extending to support `if`-statements and binary and unary expressions.

In the FIP^{ML} calculus, each variable appears once in the Δ or Γ environments, an essential constraint due to the linearity of the underlying calculus. Note that the `Integer`, `Boolean`, `Unit`, and `Atom` rules do not create any owned context or reuse credit. The `Var` rule makes explicit the linearity of the owned context Γ, as it requires `x` to be the only element of the environment. Similarly, the `Tuple` rule aggregates the Γs and ρs of the underlying values, where $\Gamma_1, ..., \Gamma_n$ implies the union of the owned contexts. The `FunCall` rule describes that only calls to `fip` functions are allowed inside FIP^{ML}, where we can pass other top-level functions or borrowed parameters as arguments, as well as owned values. Furthermore, the rule also keeps track of allocation credits $\diamond$ of the called function and the functions passed as arguments. The `FunApp` rule allows for borrowed calls of functions passed as arguments, where all the parameters of a borrowed call have to be owned. The `Let` rule is slightly more complicated, as it allows temporarily borrowing the owned context Γ_2 of the e_2 expression. This borrowing is sounds because we *consume* the variables of Γ_2 in the body of e_2, so we are free to treat them as borrowed (and hence do not destructively use Γ_2) in e_1. The `BorrowedMatch` and `DestructiveMatch` are similar, except that the latter consumes the matched variable from Γ and creates further owned variables and reuse credit proportional to the matched constructors arity. The `IfElse` rule is trivial, except that we have to check that both branches' owned and reuse credit environments are equal. `UnOp` and `BinaryOp` are trivially following from `Tuple`. Next, `Inst` and `Weak` describe the rules for handling allocation credits $\diamond$ where the first consumes an allocation credit to create a reuse credit $\diamond_k$, and the second is explicitly saying it will not use k such allocation credits. Lastly, `Defbase` and `Deffun` describe how to `fip` check function definitions from the top-level environment Σ rather than arbitrary expression.

FIP^{ML} calculus also describes a condition making `fip` annotated functions execute using bounded stack space. The bounded stack space condition requires two more conditions in the `FunCall` rule:

- Σ contains only functions defined before or part of the mutually recursive group of the current function, noted $\overline{f}$.

- Any recursive calls between the mutually recursive group of $\overline{f}$ should be tail-recursive.

  Formally, $\forall f \in \overline{f}, f(\overline{y}; \overline{x}) = \mathcal{T}[\overline{f}]$:

$$\begin{aligned}\mathcal{T}[\overline{f}] \quad ::= {} & e_0 \mid f_i(e_0; e_0) \mid \text{let}\, \overline{x} = e_0 \,\text{in}\, \mathcal{T}[\overline{f}] \mid \text{match}\, e_0 \,\text{with}\, \{\overline{p_i \mapsto \mathcal{T}_i[\overline{f}]}\} \\ & \mid \text{match!}\, e_0 \,\text{with}\, \{\overline{p_i \mapsto \mathcal{T}_i[\overline{f}]}\} \mid \text{drop}\, x;\ \mathcal{T}[\overline{f}] \mid \text{free}(k);\ \mathcal{T}[\overline{f}] \mid \text{weak}(k);\ \mathcal{T}[\overline{f}] \\ & \mid \text{inst}(k);\ \mathcal{T}[\overline{f}] \mid \text{if}\, e_0 \,\text{then}\, \mathcal{T}[\overline{f}] \mid \text{if}\, e_0 \,\text{then}\, \mathcal{T}[\overline{f}] \,\text{else}\, \mathcal{T}[\overline{f}]\end{aligned}$$

  where $\overline{f} \cap \text{fv}(e_0) = \emptyset$ and $f_i \in \overline{f}$.

A `match` expression can be destructive or non-destructive depending on the matched variable. Since $\Delta \cap \Gamma = \emptyset$, the compiler can always accurately determine whether the input `match` expression is destructive or borrowed. Therefore, if the matched variable is in the owned environment Γ, FipML treats that `match` expression as destructive. Otherwise, the compiler treats it as borrowed.

Figure 3.2 presents the FBIP^{ML} calculus allowing arbitrary stack usage and deallocation. The calculus is an extension of FIP^{ML} with `drop` and `free` instructions, as well as not imposing any tail-recursive constraint on the `FbipFunCall` rule. It is important to note that the `FbipFunCall` rule allows for passing both `fip` and `fbip` functions as arguments since FIP^{ML} is a subset of FBIP^{ML}.

Having derived the previously presented calculus, the implementation of the `fip` and `fbip` static checkers is straightforward. Since each element appears once in the owned and borrowed environments, I modelled Δ and Γ as sets. Next, I implemented the reuse credit environment ρ as a map from the size of the reuse credit to the count of how many there are available. For modelling allocation credits $\diamond$, the reuse credit map encodes them as having the key -1, therefore being distinctive from any reuse credits the programmer might generate using the `inst` rule. The reader can find the interfaces responsible for managing Δ, Γ and ρ in Appendix D.

Determining the borrowed set is easy and can be inferred from the function's signature. On the other hand, the owned and reuse credit environments have to be synthesised in a bottom-up approach in order from the functions body. The `Let` rule is particularly challenging, the strategy deployed being that we first check e_2 and then use the synthesised Γ_2 and ρ_2 environments before checking e_1. Also, when merging the owned and reuse credit environments, we must check for linearity. Nevertheless, environments must be checked for equality when analysing multiple branches, as in the `IfElse` rule, showcased in an example in Appendix D.3.

Note that the `fip` and `fbip` static checkers transform a `Typed_ast` function into one in the `Fip_ast` intermediate representation. This transformation is critical when targeting FipML's code to Lambda §3.7, so it is easy to distinguish if the programmer used a destructive match and what variable's underlying memory block the compiler can reuse. This conversion is apparent when analysing the signature of the `fip` and `fbip` static checking methods:

$$
\begin{array}{lll}
\Gamma & ::= \varnothing \mid \Gamma, x & \text{(owned environment)} \\
\Delta & ::= \varnothing \mid \Delta, y & \text{(borrowed environment)} \\
\rho & ::= \varnothing \mid \rho, \diamond_k \mid \rho, \diamond & \text{(reuse credit environment)} \\
\Sigma & ::= \varnothing \mid \Sigma, f(\overline{y};\ \overline{x}) = e & \text{(recursive top-level functions with borrowed parameters } \overline{y}\text{)}
\end{array}
$$

$$
\frac{n \in \mathbb{N}}{\Delta \,|\, \varnothing \,|\, \varnothing \vdash n}\ \text{Integer}
\qquad
\frac{b \in \mathbb{B}}{\Delta \,|\, \varnothing \,|\, \varnothing \vdash b}\ \text{Boolean}
\qquad
\frac{}{\Delta \,|\, \varnothing \,|\, \varnothing \vdash ()}\ \text{Unit}
$$

$$
\frac{}{\Delta \,|\, x \,|\, \varnothing \vdash x}\ \text{Var}
\qquad
\frac{}{\Delta \,|\, \varnothing \,|\, \varnothing \vdash C^0}\ \text{Atom}
$$

$$
\frac{\Delta \,|\, \Gamma_i \,|\, \rho_i \vdash v_i}{\Delta \,|\, \Gamma_1, ..., \Gamma_n \,|\, \rho_1, ..., \rho_n \vdash (v_1, ..., v_n)}\ \text{Tuple}
$$

$$
\frac{\overline{y_1} \in \Delta \qquad \overline{y_2} \in \operatorname{dom}(\Sigma) \qquad \Delta \,|\, \Gamma \,|\, \rho \vdash e \qquad f = \operatorname{fip}(n) \qquad \overline{y_2} = \operatorname{fip}(\overline{n})}{\Delta \,|\, \Gamma \,|\, \rho, \overline{\diamond}^{n+\overline{n}} \vdash f(\overline{y_1}, \overline{y_2}; e)}\ \text{FunCall}
$$

$$
\frac{y \in \Delta \qquad \Delta \,|\, \Gamma \,|\, \rho \vdash e}{\Delta \,|\, \Gamma \,|\, \rho \vdash ye}\ \text{FunApp}
$$

$$
\frac{\Delta, \Gamma_2 \,|\, \Gamma_1 \,|\, \rho_1 \vdash e_1 \qquad \Delta \,|\, \Gamma_2, \overline{x} \,|\, \rho_2 \vdash e_2 \qquad \overline{x} \notin \Delta, \Gamma_2}{\Delta \,|\, \Gamma_1, \Gamma_2 \,|\, \rho_1, \rho_2 \vdash \text{let}\, \overline{x} = e_1 \,\text{in}\, e_2}\ \text{Let}
$$

$$
\frac{y \in \Delta \qquad \Delta\, \overline{x_i} \,|\, \Gamma \,|\, \rho \vdash e_i \qquad \overline{x_i} \notin \Delta, \Gamma}{\Delta \,|\, \Gamma \,|\, \rho \vdash \text{match}\, y \,\text{with}\, \{C_i\, \overline{x_i} \mapsto e_i\}}\ \text{Borrowed Match}
$$

$$
\frac{\Delta \,|\, \Gamma, \overline{x_i} \,|\, \rho, \diamond_k \vdash e_i \qquad k = |\overline{x_i}| \qquad \overline{x_i} \notin \Delta, \Gamma}{\Delta \,|\, \Gamma, x \,|\, \rho \vdash e_i \vdash \text{match!}\, x \,\text{with}\, \{C_i\, \overline{x_i} \mapsto e_i\}}\ \text{Destructive Match}
$$

$$
\frac{\Delta \,|\, \Gamma_1 \,|\, \rho_1 \vdash e_1 \qquad \Delta \,|\, \Gamma_2 \,|\, \rho_2 \vdash e_2}{\Delta \,|\, \Gamma_1, \Gamma_2 \,|\, \rho_1, \rho_2 \vdash \text{if}\, e_1 \,\text{then}\, e_2}\ \text{If}
$$

$$
\frac{\Delta \,|\, \Gamma_1 \,|\, \rho_1 \vdash e_1 \qquad \Delta \,|\, \Gamma_2 \,|\, \rho_2 \vdash e_2 \qquad \Delta \,|\, \Gamma_2 \,|\, \rho_2 \vdash e_3}{\Delta \,|\, \Gamma_1, \Gamma_2 \,|\, \rho_1, \rho_2 \vdash \text{if}\, e_1 \,\text{then}\, e_2 \,\text{else}\, e_3}\ \text{IfElse}
$$

$$
\frac{\Delta \,|\, \Gamma \,|\, \rho \vdash e}{\Delta \,|\, \Gamma \,|\, \rho \vdash \text{unary\_op}(e)}\ \text{UnOp}
\qquad
\frac{\Delta \,|\, \Gamma_1 \,|\, \rho_1 \vdash e_1 \qquad \Delta \,|\, \Gamma_2 \,|\, \rho_2 \vdash e_2}{\Delta \,|\, \Gamma_1, \Gamma_2 \,|\, \rho_1, \rho_2 \vdash \text{binary\_op}(e_1, e_2)}\ \text{BinaryOp}
$$

$$
\frac{\Delta \,|\, \Gamma \,|\, \rho, \diamond_k \vdash e \qquad k \geq 1}{\Delta \,|\, \Gamma \,|\, \rho, \diamond \vdash \text{inst}\, k;\ e}\ \text{Inst}
\qquad
\frac{\Delta \,|\, \Gamma \,|\, \rho, \overline{\diamond}^{(n-k)} \vdash e \qquad 1 \leq k \leq n}{\Delta \,|\, \Gamma \,|\, \rho, \overline{\diamond}^{n} \vdash \text{weak}\, k;\ e}\ \text{Weak}
$$

$$
\frac{}{\Vdash \varnothing}\ \text{Defbase}
\qquad
\frac{\Vdash \Sigma' \qquad \overline{y} \,|\, \overline{x} \,|\, \overline{\diamond}^{n} \vdash e \qquad f = \operatorname{fip}(n)}{\Vdash \Sigma',\ f(\overline{y}; \overline{x}) = e}\ \text{Deffun}
$$

Figure 3.1: FIP^{ML} calculus

$$\frac{\Delta \,|\, \Gamma \,|\, \rho \vdash e}{\Delta \,|\, \Gamma, x \,|\, \rho \vdash \text{drop}(x);\ e}\ \text{Drop} \qquad \frac{\Delta \,|\, \Gamma \,|\, \rho \vdash e \qquad k \geq 1}{\Delta \,|\, \Gamma \,|\, \rho, \diamond_k \vdash \text{free}(k);\ e}\ \text{Free}$$

$$\frac{\overline{y_1} \in \Delta \qquad \overline{y_2} \in \text{dom}(\Sigma) \qquad \Delta \,|\, \Gamma \,|\, \rho \vdash e \qquad f = \text{fbip(n)} \qquad \overline{y_2} = \text{fip}(\overline{n}) \vee \text{fbip}(\overline{n})}{\Delta \,|\, \Gamma \,|\, \rho, \overline{\diamond}^{n+\overline{n}} \vdash f(\overline{y_1}, \overline{y_2}; e)}\ \text{FbipFunCall}$$

Figure 3.2: FBIP^{ML} calculus

```
(* src/frontend/typing/static_fip.mli *)
val fip : Typed_ast.function_defn -> Functions_env.functions_env ->
  Fip_ast.function_defn Or_error.t

(* src/frontend/typing/static_fbip.mli *)
val fbip : Typed_ast.function_defn -> Functions_env.functions_env ->
  Fip_ast.function_defn Or_error.t
```

After an annotated function is typechecked, the compiler performs the `fip`/`fbip` static check and transformation. If the static check succeeds, the now generated `Fip_ast` is accumulated, and an entry for the `fip`-ed function is added to the `functions_env` so that the programmer may explicitly call the optimised function. Here is an example:

```
fip fun map_f (xs : int list @ unique) ^(f : int @ shared -> int @ shared)
  : int list @ 'u = {
  match xs with
  | Nil -> { Nil }
  | Cons (x, xs) -> { Cons (^f(x), map_f xs f) }
  endmatch
}
fun f : int @ 'u1 -> int @ 'u2 ...
let xs = Cons (1, Cons (2, Nil)) in
map_f! (xs, f)
```

where `map_f` is a function abiding by FIP^{ML} calculus, which is executed fully in-place when explicitly called on in the last line `map_f!` using the `!` notation.

This section concludes all FipML's internals, determining whether a `fip`/`fbip` annotated function can be fully in-place. The optimisations are made explicit in the `Fip_ast` intermediate representation, used when targeting FipML code to OCaml, as we will see in the next section §3.7.

3.7 Targeting Lambda

This section describes the process of translating FipML's final intermediate representations (`Typed_ast` and `Fip_ast`) to OCaml's Lambda IR to produce executable code. The Lambda intermediate representation comprises all the necessary features for FipML to be compiled into native code with in-place optimisations. These features are the ability to allocate mutable blocks of memory, change the tag of a memory block, and update fields of memory blocks in-place.

The implementation uses an intermediate representation between the `Typed_ast` and the `Fip_ast` intermediate representations, and `Lambda` in order to avoid code duplication due to increased similarities between the two source IRs but with slightly different types and constructors.

One of the key features of the `Pre_lambda_ast` is that it provides clear differentiation between destructive and borrowed match expressions. This differentiation becomes handy when tracking which variables have been used destructively, so the translation generates code that reuses the memory blocks on `fip` optimised functions. The intermediate representation is also not annotated with types since we already know that the program is well-typed as Lambda requires. Conversion from `Typed_ast` and `Fip_ast` to `Pre_lambda_ast` is straightforward.

Before targeting `Pre_lambda_ast` to `Lambda`, there is one more step to be performed. Lambda has no direct equivalent for translating `match` expressions. Instead, match expressions are converted into a set of switches based on the tag of each constructor. It is best to look at an example:

```
(* my_list.ml *)
type my_list = Nil of unit | (::) of int * my_list
let rec zip = function
| Nil (), Nil () -> Nil ()
| x :: xs, y :: ys -> (x + y) :: zip (xs, ys)
| _ -> failwith "Length mismatch"

==============
ocamlopt -dlambda my_list.ml
(* This is a pretty-printing instance of Lambda's IR for my_list.ml *)
(seq
  (letrec
    (zip/270
       (function (param/278, param/279)
         (catch
           (switch* param/278
            case tag 0:
             (switch* param/279 case tag 0: [0: 0]
                                case tag 1: (exit 1))
            case tag 1:
             (switch* param/279
              case tag 0: (exit 1)
              case tag 1:
               (makeblock 1 (int,*)
                 (+ (field 0 param/278) (field 0 param/279))
                 (apply zip/270
                   (makeblock 0 (field 1 param/278) (field 1 param/279))))))
          with (1) (apply (field 1 (global Stdlib!)) "Length mismatch"))))
    (setfield_ptr(root-init) 0 (global Helper!) zip/270))
  0)
```

In the example above, we have the Lambda representation of a recursive function named `zip`, composed of a `match` expression. Note that `Nil` has constructor tag 0, and `(::)` has constructor tag 1. The first case of the first switch assures that both matched variables are indeed `Nil`s, raising `exit 1` if the second one is not a `Nil`. The outer `catch` expression catches the `exit` code, which handles the case where no patterns are matched and therefore executes `failwith "Length mismatch"`. Similarly, the second case of the first switch asserts based on the matched variables tags if they are both of `(::)`, in which case it extracts `x`, `xs`, `y` and `ys` using the `Pfield` directive, makes a recursive call to `zip` and finally allocates a new block storing the newly created `(::)` object.

Converting a `match`-expression into a series of switch statements is a common problem faced by functional programming languages. To perform this translation, the I implemented Philip Walder's algorithm for efficient pattern-matching compilation (Peyton Jones 1987), applying the algorithm directly to the `Pre_lambda_ast` match expressions.

After performing the pattern-matching compilation, we are all prepared to target `Pre_lambda_ast` to `Lambda`. Creating the mapping from FipML's final IR to Lambda has been challenging due to the lack of widely available resources and examples other than the indispensable help from the supervisor, who provided examples of generating lambda code for particular programs. In the implementation, I tracked which variables have been destructively used (since this is easy to see in the `Pre_typed_ast` IR) by using a map from reuse credit size to a list of `Var_name.ts` encoding consumed variables. The map is necessary so the converter reuses already allocated memory blocks.

On memory block re-usage conversion, we potentially have to set the block's tag to the tag of the constructor we want to store. The `Pccal` primitive `caml_obj_set_tag` updates the block's tag, which forced me to use OCaml 4.12 over the new 5.1 available distribution. Here is an example that illustrates this requirement:

```
type t = C1 of int | C2 of int
let x = C1 42 in
match x with C1 y -> C2 (y + 1)
...
```

In the destructive `match`-expression above, while both constructors have size two, `C1` has the tag 0 and `C2` has the tag 1. Therefore, when reusing `C1`'s block memory to store `C2`'s data, we also have to update the memory block's header.

Once the Lambda IR was generated for the FipML program, the I used OCaml's `Asmgen.compile_implementation` function to generate a native code executable. Alternatively, I could have targeted the code to native bytecode, but native bytecode is considerably slower than native code.

One last caveat, a FipML program may only return an integer. I had to write Lambda code to display anything in the terminal and decided to simplify the codebase and only allow for integers to be printed. The code associated with the Lambda function for printing an integer can be found in `src/frontend/target/print_int_lambda.ml`.

With everything explained in this chapter in place, FipML's pipeline is complete, managing to turn `.fipml` code into a native executable with guaranteed performance.

3.8 Repository overview

```
  fipml
└─ src
    ├─ experiments
    │   ├─ msort
    │   ├─ qsort
    │   ├─ rbtree
    │   └─ tmap
    ├─ frontend
    │   ├─ ast
    │   ├─ parsing
    │   ├─ target
    │   └─ typing
    ├─ main .1 test
    └─ frontend
        ├─ fip
        ├─ lexer
        ├─ parsing
        └─ typing
            ├─ function_defns
            ├─ type_defns
            └─ type_infer
```

FipML's entire repository is split into two parts: implementation and testing. In the `test/` directory, one can find tests for each transformation in FipML's pipeline. In the `src/` directory, a developer will find:

- the `experiments/` directory, with one separate directory for each benchmark as explained in §4.2,
- the `frontend/` directory containing all the intermediate representations, transformations and algorithms discussed during this chapter. Notably, the `fip/` subdirectory includes the code for the static `fip/fbip` check as described in section §3.6. Similarly, subdirectory `typing/` contains all the code responsible for typechecking FipML programs §3.4 and §3.5.

Lastly, `src/main.ml` is the entry point of our compiler. In that file, a programmer may specify where his `.fipml` file is, and executing `main.ml` will create the native code for the input program.

Chapter 4

Evaluation

4.1 Success criteria

The project meets all the success criteria presented in the project proposal (Appendix E):

- Design a grammar and implement a lexer and parser for the FipML §3.2.
- Implement the static analyses to determine opportunities for in-place updates §3.6.
- Implement a compiler targeting OCaml's back-end that takes advantage of the in-place update opportunities §3.7.
- Evaluate the performance of the generated code with and without in-place update optimisations §4.2.

FipML is a functional programming language that deploys a polymorphic uniqueness typing system, allowing the compiler to optimise functions where fully in-place execution is safe automatically. Lastly, FipML code is translated to native executables using OCaml's Lambda intermediate representation to provide code execution. Evidence of FipML's memory reusage is presented in §4.2, where I have tested the performance of my language against other functional languages on various data structures and algorithms.

4.2 FipML benchmarks

This section presents FipML's performance against other functional programming languages on well-designed algorithms and data structures. I used the work of Lorenzen, Leijen, and Swierstra 2023 as inspiration for designing the experiments. Running the experiments records execution speed and peak memory usage on Red-black trees, Quicksort, Mergesort and mapping a function onto a generic tree. I have picked these data structures and algorithms because they involve intensive memory allocation by a non-in-place optimised programming language, as modifications to the underlying structure require fresh heap memory. The execution metrics are averaged over five runs on a Mac Book Pro 2021, 10-Core M1 Max, 64GB RAM machine to remove the variance due to the pre-existing load of the underlying machine. I have tested each benchmark in the following variants:

Experiment	Koka	OCaml	Non-fip FipML	FipML
msort time	344.5 s	612.32 s	614.97 s	404.47 s
qsort time	81 s	821.71 s	841.96 s	428.39 s
tmap time	75.03 s	59.98 s	80.84 s	61.07 s
rbtree time	14.08	13.55	40.47	13.25

Figure 4.1: Experiments execution time

Execution metric	Koka	OCaml	Non-fip FipML	FipML
msort mem	40.20 MB	275.78 MB	263.51 MB	75.65 MB
qsort mem	11.07 MB	20.70 MB	20.43 MB	14.78 MB
tmap mem	28770.67 MB	28009.98 MB	39379.29 MB	21203.45 MB
rbtree mem	5.89 MB	67.95 MB	110.78 MB	73.07 MB

Figure 4.2: Experiments peak memory usage

non-fip optimised FipML, fip optimised FipML, fip optimised Koka and plain OCaml. Each benchmark consists of:

- mergesort - *msort* sorts a descending list of N=500000 elements with values from N to 1. This procedure is executed K=10,000 times in a row. After each round, it records the smallest element of the sorted list. Once all the K rounds are completed, the experiment sums the previously recorded values (which should total 10,000) to ensure the correctness of the underlying execution.

- quicksort - *qsort* sorts a descending list of N=100000 elements with values from N to 1. This procedure is executed five times in a row. Finally, the experiment computes the sum of the smallest element of each list after every round (which should total 10000) to ensure correctness.

- *tmap* - applies a function over a tree containing $N=2^30$ nodes. This procedure is executed only once. Finally, the experiment computes the sum of all elements in the tree to check for correctness.

- *rbtree* - performs N=100000 balanced red-black tree insertions, after which a fold operation is applied to compute the sum of all the elements in the data structure.

With these experiment settings described, using `/usr/bin/time -l` and xCode's Instruments profiler, I have gathered the metrics about execution time (Figure 4.1) and peak memory usage (Figure 4.2).

Comparing the OCaml execution metrics with the optimised FipML ones, we see that on average FipML:

- takes 66% of the running time of OCaml no *msort* and uses 27% of the memory,

- takes 53% of the running time of OCaml no *qsort* and uses 72% of the memory,

- takes 101% of the running time of OCaml no *tmap* and uses 75% of the memory - OCaml is marginally faster but uses more memory,

- takes 97% of the running time of OCaml no *rbtree* and uses 107% of the memory.

Therefore, FipML is, on average, more performant on the experiments suite, consuming more memory than OCaml only on the *rbtree* experiment.

Fip optimisations significantly improved both memory usage and execution time, comparing non-fip FipML and optimised-FipML on every single benchmark. On the two sorting benchmarks, execution time and memory usage were significantly better than OCaml, especially for the *msort* benchmark where FipML used between 23% and 31% less memory. Lastly, on *tmap* and *rbtree*, non-optimised FipML was notably worse than OCaml, but the `fip` optimisations brought it into a similar range.

On the other hand, Koka appears to be more performant, with better execution speed and peak memory usage than both OCaml and optimised FipML on all benchmarks, except for *tmap*. This could potentially be explained as Koka having a dynamic reference count that might free memory when it is no longer needed. By deallocating memory, Koka could have lower peak memory usage and faster execution due to not frequently resorting to garbage collection to save heap memory. Comparing fip-optimised FipML against Koka, we observe that:

- on *msort*, Koka takes 85% of the running time of FipML, consuming only 53% of the memory,
- on *qsort*, Koka takes 20% of the running time of FipML, consuming only 75% of the memory,
- on *tmap*, Koka takes 122% of the running time of FipML, and consumes 135% of the memory, and
- on *rbtree*, Koka takes 106% of the running time of FipML, consuming only a mere 8% of the memory.

Chapter 5

Conclusion

In the dissertation, I presented FipML, the first functional programming language that automatically performs in-place updates by leveraging a uniqueness typing system. Furthermore, I understood OCaml's internals better, having to use Lambda, one of its intermediate representations.

5.1 Lessons learnt

Throughout the dissertation, I learned how to create a compiler from scratch. I have deepened my knowledge of type systems, having studied and implemented one for the first time.

5.2 Future work

Throughout the project, I discovered several areas where FipML could be extended, some of them inspired by the extensions mentioned in the project proposal that I could not approach given the strict timeframe:

- The sharing analysis heuristic is a bit too simplistic. For example, the uniqueness type system should allow a unique variable to be used as shared as many times before it is used for the first time as unique. This is currently not implemented in FipML, but it would be interesting to explore improving the sharing analysis heuristic.
- One extension I did not get to implement is to compare the relative strength of the `fip` static check to the dynamic reference counting approach used in Koka. As one would expect, a type system does not accept as many programs as a dynamic checker would.
- Enhancing FipML to perform Tail Modulo Constructor optimisation (Bour, Clément, and Scherer 2021) could lead to even better performance gains.
- Currently, FipML's syntax is rigid if using polymorphic custom types, as the programmer has to present the polymorphic type variables in a pre-defined order. The

syntax could be improved to resemble OCaml's by potentially doing unification on the polymorphic type variables such that it can automatically infer which type variable encodes a partial type, uniqueness attribute, or fully initialised type.

Bibliography

Aspinall, David and Martin Hofmann (2002). “Another type system for in-place update”. In: *European Symposium on Programming*. Springer, pp. 36–52.

Bour, Frédéric, Basile Clément, and Gabriel Scherer (2021). “Tail modulo cons”. In: *arXiv preprint arXiv:2102.09823*.

De Vries, Edsko, Rinus Plasmeijer, and David M Abrahamson (2007). “Uniqueness typing redefined”. In: *Implementation and Application of Functional Languages: 18th International Symposium, IFL 2006, Budapest, Hungary, September 4-6, 2006, Revised Selected Papers 18*. Springer, pp. 181–198.

— (2008). “Uniqueness typing simplified”. In: *Implementation and Application of Functional Languages: 19th International Workshop, IFL 2007, Freiburg, Germany, September 27-29, 2007. Revised Selected Papers 19*. Springer, pp. 201–218.

Dolan, Stephen (2016). “Malfunctional programming”. In: *ML Workshop*.

Lorenzen, Anton, Daan Leijen, and Wouter Swierstra (2023). “FP^2: Fully in-Place Functional Programming”. In: *Proceedings of the ACM on Programming Languages* 7.ICFP, pp. 275–304.

Lorenzen, Anton, Leo White, et al. (2024). “Oxidizing OCaml with Modal Memory Management”.

Milner, Robin (1978). “A theory of type polymorphism in programming”. In: *Journal of Computer and System Sciences* 17.3, pp. 348–375. ISSN: 0022-0000. DOI: `https://doi.org/10.1016/0022-0000(78)90014-4`. URL: `https://www.sciencedirect.com/science/article/pii/0022000078900144`.

Minsky, Yaron, Anil Madhavapeddy, and Jason Hickey (2013). “Real World OCaml: Functional programming for the masses”. In: ” O’Reilly Media, Inc.”. Chap. Parsing with OCamllex and Menhir, pp. 311–324.

OCaml (2024). *The Flambda backend project for OCaml.* Accessed on 2024-05-8. URL: `https://github.com/ocaml-flambda/flambda-backend`.

Peyton Jones, Simon L (1987). *The implementation of functional programming languages (Prentice-Hall international series in computer science).* Prentice-Hall, Inc. Chap. Efficient compilation of pattern-matching, pp. 78–103.

Pierce, Benjamin C (2002). “Types and programming languages”. In: MIT press. Chap. Constraint-Based Typing, pp. 317–338.

Régis-Gianas, François Pottier Yann (2016). *Menhir Reference Manual.*

Reinking, Alex et al. (2021). “Perceus: Garbage free reference counting with reuse”. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pp. 96–111.

Street, Jane (2024). *Dune: A Composable Build System.* Accessed on 2024-04-29. URL: `https://github.com/ocaml/dune`.

University, Radboud (2024). *Clean Wiki.* Accessed on 2024-05-11. URL: `https://wiki.clean.cs.ru.nl/Clean`.

Appendix A

Ocamllex and Menhir

A.1 Lexing

```
{
  open Lexing
  open Parser

  exception LexerError of string

  let next_line lexbuf =
    let pos = lexbuf.lex_curr_p in
    lexbuf.lex_curr_p <-
      { pos with pos_bol = lexbuf.lex_curr_pos;
                 pos_lnum = pos.pos_lnum + 1
      }
}

(** Helper regexes for the Lexer *)
let digit = ['0'-'9']
let lowercase = ['a'-'z']
let uppercase = ['A'-'Z']
let letter = lowercase | uppercase

(** Regexes for finding tokens *)
let integer_regex_expression = '-'? digit+
let lid_regex_expression = lowercase (letter | digit | '_')*
let uid_regex_expression = uppercase (letter | digit | '_')*
let whitespace_regex_expression = [' ' '\t']+
let newline_regex_expression = '\r' | '\n' | "\r\n"


rule token = parse
  | whitespace_regex_expression { token lexbuf }
  | "@" { AT }
  | "(" { LPAREN }
  | ")" { RPAREN }
  | "{" { LCURLY }
  | "}" { RCURLY }
  | "," { COMMA }
  | ":" { COLON }
  | ";" { SEMICOLON }
  | "+" { ADD }
  | "-" { SUB }
  | "*" { MUL }
  | "/" { DIV }
  | "%" { MOD }
  | "!" { NOT }
  | "^" { BORROWED }
  | "=" { ASSIGN }
  | "<" { LT }
  | ">" { GT }
  | "|" { BAR }
  | "_" { UNDERSCORE }
  | "<=" { LEQ }
  | ">=" { GEQ }
  | "==" { EQ }
  | "!=" { NEQ }
  | "&&" { AND }
  | "||" { OR }
  | "->" { ARROW }
  | "()" { UNIT }
  | "of" { OF }
  | "if" { IF }
  | "then" { THEN }
  | "else" { ELSE }
  | "endif" { ENDIF }
  | "true" { TRUE }
  | "false" { FALSE }
  | "let" { LET }
  | "fun" { FUN }
  | "and" { ANDFUN }
  | "in" { IN }
  | "type" { TYPE }
  | "match" { MATCH }
  | "endmatch" { ENDMATCH }
  | "with" { WITH }
  | "'" lid_regex_expression { POLY (Lexing.lexeme lexbuf) }
  | "int" { TYPE_INT }
  | "bool" { TYPE_BOOL }
  | "unit" { TYPE_UNIT }
  | "unique" { UNIQUE }
  | "shared" { SHARED }
  | "fip" { FIP }
  | "fbip" { FBIP }
  | "drop" { DROP }
  | "free" { FREE }
  | "weak" { WEAK }
  | "inst" { INST }
  | integer_regex_expression { INT (int_of_string (Lexing.lexeme lexbuf)) }
  | lid_regex_expression { LID (Lexing.lexeme lexbuf) }
  | uid_regex_expression { UID (Lexing.lexeme lexbuf) }
  | "/*" { multi_line_comment lexbuf }
  | "//" { single_line_comment lexbuf }
  | newline_regex_expression { next_line lexbuf; token lexbuf }
  | eof { EOF }
  | _ { raise (LexerError ("Unidentified token")) }

and multi_line_comment = parse
  | "*/" { token lexbuf }
  | newline_regex_expression { next_line lexbuf; multi_line_comment lexbuf }
  | eof { raise (LexerError("Unexpected End of FILE (EOF) - Finish comment")) }
  | _ { multi_line_comment lexbuf }

and single_line_comment = parse
  | newline_regex_expression { next_line lexbuf; token lexbuf }
  | eof { EOF }
  | _ { single_line_comment lexbuf }
```

A.2 Parsing

```
%{
  (* When compiling the .mly file, this code is put at the very top. *)
  open Ast.Ast_types
  open Parser_ast

  let mutually_recursive_group_id = ref 0
  let incr_mutually_recursive_group_id () =
    mutually_recursive_group_id := !mutually_recursive_group_id + 1
  let get_mutually_recursive_group_id () = !mutually_recursive_group_id
%}
```

```
%token UNIT TRUE FALSE LPAREN RPAREN
%token SUB NOT
%token ADD SUB MUL DIV MOD LT GT LEQ GEQ EQ NEQ AND OR
%token COMMA LET ASSIGN
%token<int> INT
%token<string> LID UID
...
/* Precedence and associativity */
%nonassoc LT GT LEQ GEQ EQ NEQ IN
%left OR
%left AND
%right NOT
%left ADD SUB
%left MUL DIV MOD
...
/* Starting non-terminal, endpoint for calling the parser */
%start <program> program
...
%%
program:
| type_defns=list(type_defn); function_defns=list(function_defn);
      main_expr_option=option(block_expr); EOF {
    TProg($startpos, type_defns, function_defns, main_expr_option)
  }
...
/* Value Production Rules */
value:
| UNIT { Unit($startpos) }
| n=INT { Integer($startpos, n) }
| TRUE { Boolean($startpos, true) }
| FALSE { Boolean($startpos, false) }
| var_name=LID { Variable($startpos, Var_name.of_string var_name) }
| constructor_name=UID {
    Constructor($startpos, Constructor_name.of_string constructor_name, [])
  }
| constructor_name=UID; LPAREN;
    constructor_args=separated_nonempty_list(COMMA, value); RPAREN {
    Constructor($startpos,
        Constructor_name.of_string constructor_name,
        constructor_args
    )
  }
```

```
%inline unary_op:
| SUB { UnOpNeg }
| NOT { UnOpNot }

%inline binary_op:
| ADD { BinOpPlus } | SUB { BinOpMinus } | MUL { BinOpMult } | DIV { BinOpDiv }
| MOD { BinOpMod } | LT { BinOpLt } | GT { BinOpGt } | LEQ { BinOpLeq }
| GEQ { BinOpGeq } | EQ { BinOpEq } | NEQ { BinOpNeq } | AND { BinOpAnd }
| OR { BinOpOr }

expr:
/* Unboxed */
| value=value { UnboxedSingleton($startpos, value) }
| LPAREN; values=separated_nonempty_list(COMMA, value); RPAREN {
    UnboxedTuple($startpos, values)
  }

/* Convoluted expressions */
| unary_op=unary_op; expr=expr { UnOp($startpos, unary_op, expr) }
| expr_left=expr; binary_op=binary_op; expr_right=expr {
    BinaryOp($startpos, binary_op, expr_left, expr_right)
  }

| LET; var_name=LID; ASSIGN; var_expr=expr; IN; var_scope=expr {
    Let($startpos, [Var_name.of_string var_name], var_expr, var_scope)
  }
| LET; LPAREN;
    var_names=separated_nonempty_list(COMMA, LID);
  RPAREN; ASSIGN; var_expr=expr; IN; var_scope=expr {
    let vars =
        List.map (fun var_name -> Var_name.of_string var_name) var_names
    in
    Let($startpos, vars, var_expr, var_scope)
  }
...
function_defn:
| mut_rec=option(ANDFUN); FIP; FUN; fun_name=LID;
    fun_params=nonempty_list(function_param);
    COLON; return_type=type_expr; ASSIGN;
    fun_body=block_expr {
    (match mut_rec with
    | None ->  incr_mutually_recursive_group_id ()
    | _ -> ());
    TFun($startpos, get_mutually_recursive_group_id (),
        Some (Fip 0), Function_name.of_string fun_name,
        fun_params, fun_body, return_type)
  }
...
```

Appendix B

Constraint typing rules

$$\frac{n \in \mathbb{Z}}{\Gamma \vdash n : \text{int} \dashv \emptyset}\ \text{Integer} \qquad \frac{b \in \mathbb{B}}{\Gamma \vdash b : \text{bool} \dashv \emptyset}\ \text{Boolean}$$

$$\frac{x : \forall \alpha_1, ..., \alpha_n.t \in \Gamma}{\Gamma \vdash x : \text{inst}(\forall \alpha_1, ..., \alpha_n.t) \dashv \emptyset}\ \text{Variable}$$

$$\frac{\Gamma \vdash v_i : t_i \dashv C_i \qquad t'_i = \text{inst\_arg\_type}_i(C^k) \qquad t' = \text{inst}(\text{type}(C^k)) \qquad t = \text{fresh}()}{\Gamma \vdash C^k(v_1, ..., v_k) : t \dashv \{(t, t'), (t_1, t'_1), ..., (t_k, t'_k)\} \cup C_1 \cup ... \cup C_k}\ \text{Constructor}$$

$$\frac{\Gamma \vdash v : t \dashv C}{\Gamma \vdash (v) : t \dashv C}\ \text{UnboxedSingleton}$$

$$\frac{\Gamma \vdash v_i : t_i \dashv C_i \qquad t = \text{fresh}()}{\Gamma \vdash (v_1, ..., v_k) : t \dashv \{(t, (t_1, ..., t_k))\} \cup C_1 \cup ... \cup C_k}\ \text{UnboxedTuple}$$

$$\frac{\Gamma \dashv e_1 : t_1 \dashv C_1 \qquad \text{generalise}(C_1, \Gamma, \overline{x} : t_1) \vdash e_2 : t_2 \dashv C_2}{\text{let } \overline{x} = e_1 \text{ in } e_2 : t_2 \dashv C_1 \cup C_2}\ \text{Let}$$

$$\frac{\text{inst}(\wedge f) : t_1 \to ... \to t_k \to t_{k+1} \in \Gamma \qquad \Gamma \vdash v_i : t'_i \dashv C_i \qquad t = \text{fresh}()}{\Gamma \vdash \wedge f(v_1, ..., v_k) : t \dashv \{(t, t_{k+1}), (t_1, t'_1), ..., (t_k, t'_k)\} \cup C_1 \cup ... \cup C_k}\ \text{FunApp}$$

$$\frac{\text{inst}(f) : t_1 \to ... \to t_k \to t_{k+1} \in functions\_env \qquad \Gamma \vdash v_i : t'_i \dashv C_i \qquad t = \text{fresh}()}{\Gamma \vdash f(v_1, ..., v_k) : t \dashv \{(t, t_{k+1}), (t_1, t'_1), ..., (t_k, t'_k)\} \cup C_1 \cup ... \cup C_k}\ \text{FunCall}$$

$$\frac{\Gamma \vdash e_1 : t_1 \dashv C_1 \qquad \Gamma \vdash e_2 : t_2 \dashv C_2 \qquad t = \text{fresh}()}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 : t \dashv \{(t_1, \text{bool}), (t, t_2), (t_2, \text{unit})\} \cup C_1 \cup C_2}\ \text{If}$$

$$\frac{\Gamma \vdash e_1 : t_1 \dashv C_1 \qquad \Gamma \vdash e_2 : t_2 \dashv C_2 \qquad \Gamma \vdash e_2 : t_3 \dashv C_3 \qquad t = \text{fresh}()}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t \dashv \{(t_1, \text{bool}), (t, t_3), (t_2, t_3)\} \cup C_1 \cup C_2 \cup C_3}\ \text{IfElse}$$

$$\frac{\Gamma \vdash e : t \dashv C}{\Gamma \vdash -e : t \dashv \{(t, \text{int})\} \cup C}\ \text{UnOpNeg}$$

$$\frac{\Gamma \vdash e : t \dashv C}{\Gamma \vdash !e : t \dashv \{(t, \text{bool})\} \cup C}\ \text{UnOpNot}$$

$$\frac{\Gamma \vdash e_1 : t_1 \dashv C_1 \qquad \Gamma \vdash e_2 : t_2 \dashv C_2}{\Gamma \vdash e_1 + e_2 : \text{int} \dashv \{(t_1, \text{int}), (t_2, \text{int})\} \cup C_1 \cup C_2} \text{BinOpPlus}$$

$$\frac{\Gamma \vdash e_1 : t_1 \dashv C_1 \qquad \Gamma \vdash e_2 : t_2 \dashv C_2}{\Gamma \vdash e_1 < e_2 : \text{bool} \dashv \{(t_1, \text{int}), (t_2, \text{int})\} \cup C_1 \cup C_2} \text{BinOpLt}$$

$$\frac{\Gamma \vdash e_1 : t_1 \dashv C_1 \qquad \Gamma \vdash e_2 : t_2 \dashv C_2}{\Gamma \vdash e_1 == e_2 : \text{bool} \dashv \{(t_1, t_2)\} \cup C_1 \cup C_2} \text{BinOpEq}$$

$$\frac{\Gamma \vdash e_1 : t_1 \dashv C_1 \qquad \Gamma \vdash e_2 : t_2 \dashv C_2}{\Gamma \vdash e_1 \;\&\&\; e_2 : \text{bool} \dashv \{(t_1, \text{bool}), (t_2, \text{bool})\} \cup C_1 \cup C_2} \text{BinOpAnd}$$

$$\frac{\Gamma \vdash e : t \dashv C}{\Gamma, x : t' \vdash \text{drop } x;\; e : t \dashv C} \text{Drop} \qquad \frac{\Gamma \vdash e : t \dashv C \qquad k \in \mathbb{N}}{\Gamma \vdash \text{free } k;\; e : t \dashv C} \text{Free}$$

$$\frac{\Gamma \vdash e : t \dashv C \qquad k \in \mathbb{N}}{\Gamma \vdash \text{weak } k;\; e : t \dashv C} \text{Weak} \qquad \frac{\Gamma \vdash e : t \dashv C \qquad k \in \mathbb{N}}{\Gamma \vdash \text{inst } k;\; e : t \dashv C} \text{Inst}$$

$$\frac{\Gamma_i, t_i, C_i = \text{expand}(p_i) \qquad \Gamma, \Gamma_i \vdash e_i : t_i' \dashv C_i' \qquad t = \text{fresh}()}{\Gamma, x : t_0 \vdash \text{match } x \text{ with } \{\overline{p_i \mapsto e_i}^k\} : t \dashv \bigcup_{i=1}^k (\{(t_0, t_i), (t, t_i')\} \cup C_i \cup C_i')} \text{Match}$$

$$\text{expand}(\_) = \emptyset, t, \emptyset \quad \text{where } t = \text{fresh}()$$

$$\text{expand}(x) = \{x : t\}, t, \emptyset \quad \text{where } t = \text{fresh}()$$

$$\begin{aligned}
&\text{expand}(C^k\, p_1 \ldots p_k) = \Gamma, t, C \quad \text{where} \\
&\qquad \Gamma_i, t_i, C_i = \text{expand}(p_i) \\
&\qquad \forall \alpha_1 \ldots \alpha_n . t_0 = \text{type}(C^k) \\
&\qquad t = [\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n] t_0 \quad \text{with } \beta_i = \text{fresh}() \\
&\qquad \Gamma = \Gamma_1, \ldots, \Gamma_k \\
&\qquad C = C_1 \cup \cdots \cup C_k \cup \{(t_i, \text{arg}_i(t)) \mid 1 \leq i \leq k\}
\end{aligned}$$

Appendix C

Uniqueness type system

C.1 Sharing analysis

```
let join_sharing_analysis_map (sharing_analysis_map1 : int SharingAnalysisMap.t)
    (sharing_analysis_map2 : int SharingAnalysisMap.t) :
    int SharingAnalysisMap.t =
  Map.merge_skewed sharing_analysis_map1 sharing_analysis_map2
    ~combine:(fun ~key:_ v1 v2 -> v1 + v2)

let join_max_sharing_analysis_map
    (sharing_analysis_map1 : int SharingAnalysisMap.t)
    (sharing_analysis_map2 : int SharingAnalysisMap.t) :
    int SharingAnalysisMap.t =
  Map.merge_skewed sharing_analysis_map1 sharing_analysis_map2
    ~combine:(fun ~key:_ v1 v2 -> Int.max v1 v2)

let rec get_sharing_analysis (parsed_expr : Parser_ast.expr) :
    int SharingAnalysisMap.t =
  match parsed_expr with
  | UnboxedSingleton (_, value) -> get_sharing_analysis_value value
  | UnboxedTuple (_, values) -> get_sharing_analysis_values values
  | Let (_, vars, vars_expr, expr) ->
      let vars_expr_sharing_analysis_map = get_sharing_analysis vars_expr in
      let expr_sharing_analysis_map = get_sharing_analysis expr in
      let joined_sharing_analysis_map =
        join_sharing_analysis_map vars_expr_sharing_analysis_map
          expr_sharing_analysis_map
      in
      List.fold vars ~init:joined_sharing_analysis_map
        ~f:(fun acc_sharing_analysis_map var ->
          Map.remove acc_sharing_analysis_map var)
  | FunApp (_, _, values) | FunCall (_, _, values) ->
      get_sharing_analysis_values values
  | If (_, expr_cond, expr_then) ->
      join_sharing_analysis_map
        (get_sharing_analysis expr_cond)
        (get_sharing_analysis expr_then)
  | IfElse (_, expr_cond, expr_then, expr_else) ->
      join_sharing_analysis_map
        (join_max_sharing_analysis_map
           (get_sharing_analysis expr_then)
           (get_sharing_analysis expr_else))
        (get_sharing_analysis expr_cond)
  | Match (_, matched_var, pattern_exprs) ->
      let aggr_sharing_analysis_map =
        List.fold pattern_exprs ~init:SharingAnalysisMap.empty
          ~f:(fun
               acc_sharing_analysis_map
               (Parser_ast.MPattern (_, matched_expr, expr))
             ->
             let matched_vars = Parser_ast.get_matched_expr_vars matched_expr in
             let expr_sharing_analysis_map = get_sharing_analysis expr in
             let extended_sharing_analysis_map =
               join_max_sharing_analysis_map acc_sharing_analysis_map
                 expr_sharing_analysis_map
             in
             List.fold matched_vars ~init:extended_sharing_analysis_map
               ~f:(fun acc_sharing_analysis_map matched_var ->
                 Map.remove acc_sharing_analysis_map matched_var))
      in
      Map.update aggr_sharing_analysis_map matched_var ~f:(fun key_option ->
          match key_option with None -> 1 | Some key -> key + 1)
  | UnOp (_, _, expr) -> get_sharing_analysis expr
  | BinaryOp (_, _, expr1, expr2) ->
      join_sharing_analysis_map
        (get_sharing_analysis expr1)
        (get_sharing_analysis expr2)
  | Drop (_, drop_var, expr) ->
      Map.update (get_sharing_analysis expr) drop_var ~f:(fun key_option ->
          match key_option with None -> 1 | Some key -> key + 1)
  | Free (_, _, expr) | Weak (_, _, expr) | Inst (_, _, expr) ->
      get_sharing_analysis expr

and get_sharing_analysis_value (parsed_value : Parser_ast.value) :
    int SharingAnalysisMap.t =
  match parsed_value with
  | Unit _ | Integer _ | Boolean _ -> SharingAnalysisMap.empty
  | Variable (_, var_name) -> SharingAnalysisMap.singleton var_name 1
  | Constructor (_, _, values) -> get_sharing_analysis_values values

and get_sharing_analysis_values (parsed_values : Parser_ast.value list) :
    int SharingAnalysisMap.t =
  List.fold parsed_values ~init:SharingAnalysisMap.empty
    ~f:(fun acc_sharing_analysis_map value ->
      join_sharing_analysis_map
        (get_sharing_analysis_value value)
        acc_sharing_analysis_map)
```

C.2 Unification for base type constraints

This section includes the full code for unifying base-type constraints. This is slightly more complicated, as unifying two constraints might generate further uniqueness constraints, as we can see from the comments inserted in the code.

```
let unify (constraints : constr list) :
    (subst list * constr_unique list) Or_error.t =
  let rec unify (constraints : constr list)
      (extra_unique_constraints : constr_unique list) (substs : subst list) =
    match constraints with
    | [] -> Ok (substs, extra_unique_constraints)
    | (t1, t2) :: constraints when ty_equal t1 t2 ->
        unify constraints extra_unique_constraints substs
    | (TyVar type_var, t) :: constraints when not (occurs type_var t) ->
        let ts = ty_subst [ (type_var, t) ] [] in
        unify
          (List.map constraints ~f:(fun (ty1, ty2) -> (ts ty1, ts ty2)))
          extra_unique_constraints
          ((type_var, t) :: List.map substs ~f:(fun (ty1, ty2) -> (ty1, ts ty2)))
    | (t, TyVar type_var) :: constraints when not (occurs type_var t) ->
        let ts = ty_subst [ (type_var, t) ] [] in
        unify
          (List.map constraints ~f:(fun (ty1, ty2) -> (ts ty1, ts ty2)))
          extra_unique_constraints
          ((type_var, t) :: List.map substs ~f:(fun (ty1, ty2) -> (ty1, ts ty2)))
    | ( TyArrow ((ty11, ty_unique11), (ty12, ty_unique12)),
        TyArrow ((ty21, ty_unique21), (ty22, ty_unique22)) )
      :: constraints ->

    (* When unifying function types, we have to be careful that their
       arguments' uniqueness attributes also unify. This is handled by
       recording extra uniqueness constraints which will be all later
       solved by the unify_unique method. This idea is further generalised
       for TyTuple and TyCustom. *)
        unify
          ((ty11, ty21) :: (ty12, ty22) :: constraints)
          ((ty_unique11, ty_unique21) :: (ty_unique12, ty_unique22)
         :: extra_unique_constraints)
          substs
    | (TyTuple ty_attrs1, TyTuple ty_attrs2) :: constraints
      when List.length ty_attrs1 = List.length ty_attrs2 ->
        let ty_constraints, ty_unique_constraints =
          List.fold2_exn ty_attrs1 ty_attrs2 ~init:([], [])
            ~f:(fun
                (acc_ty_constraints, acc_ty_unique_constraints)
                (ty1, ty_unique1)
                (ty2, ty_unique2)
              ->
              ( (ty1, ty2) :: acc_ty_constraints,
                (ty_unique1, ty_unique2) :: acc_ty_unique_constraints ))
        in
        unify
          (ty_constraints @ constraints)
          (ty_unique_constraints @ extra_unique_constraints)
          substs
    | ( TyCustom (tys1, ty_uniques1, ty_attrs1, type_name1),
        TyCustom (tys2, ty_uniques2, ty_attrs2, type_name2) )
      :: constraints
      when Type_name.( = ) type_name1 type_name2
           && List.length tys1 = List.length tys2
           && List.length ty_attrs1 = List.length ty_attrs2 ->
        let tys_constraints =
          List.map2_exn tys1 tys2 ~f:(fun ty1 ty2 -> (ty1, ty2))
        in
        let ty_unique_constraints_from_ty_uniques =
          List.map2_exn ty_uniques1 ty_uniques2 ~f:(fun ty_unique1 ty_unique2 ->
              (ty_unique1, ty_unique2))
        in
        let ty_constraints_from_ty_attrs, ty_unique_constraints_from_ty_attrs =
          List.fold2_exn ty_attrs1 ty_attrs2 ~init:([], [])
            ~f:(fun
                (acc_ty_constraints, acc_ty_unique_constraints)
                (ty1, ty_unique1)
                (ty2, ty_unique2)
              ->
              ( (ty1, ty2) :: acc_ty_constraints,
              (ty_unique1, ty_unique2) :: acc_ty_unique_constraints ))
        in
        unify
          (tys_constraints @ ty_constraints_from_ty_attrs @ constraints)
          (ty_unique_constraints_from_ty_uniques
         @ ty_unique_constraints_from_ty_attrs @ extra_unique_constraints)
          substs
    | (ty1, ty2) :: _ ->
        let error_string = Fmt.str "Unable to unify types %s and %s@." in
        let string_ty1 = Pprint_type_infer.string_of_ty ty1 in
        let string_ty2 = Pprint_type_infer.string_of_ty ty2 in
        Or_error.of_exn (UnableToUnify (error_string string_ty1 string_ty2))
  in
  unify constraints [] []
```

C.3 Uniqueness typing rules

$$\frac{}{\Gamma, x : t^u \vdash x^{\odot} : t^u \dashv \emptyset}\text{Var }^{\odot} \qquad \frac{}{\Gamma, x : t^{\times} \vdash x^{\otimes} : t^{\times} \dashv \emptyset}\text{Var }^{\otimes}$$

$$\frac{}{\Gamma \vdash n : \text{int}^u \dashv \emptyset}\text{Int} \qquad \frac{}{\Gamma \vdash b : \text{bool}^u \dashv \emptyset}\text{Bool} \qquad \frac{}{\Gamma \vdash u : \text{unit}^u \dashv \emptyset}\text{Unit}$$

$$\frac{\Gamma \vdash v_i : t_i^{u_i} \dashv C_i \qquad t = \text{type}(C^k) \qquad u = \text{fresh\_unique}() \qquad u_i' = \text{inst}(\text{arg\_sharing}_i(C^k))}{\Gamma \vdash C^k\,(v_1, ..., v_k) : t^u \dashv \{(u_i, u_i') \mid 1 \leq i \leq k\} \cup \bigcup_{i=1}^{k} C_i}\text{Constr}$$

$$\frac{\Gamma, v_i : a_i \vdash v_i : a_i \dashv C_i}{\Gamma, v_1 : a_1, ..., v_n : a_n \vdash (v_1, ..., v_n) : (a_1, ..., a_n)^u \dashv \bigcup_{i=0}^{n} C_i}\text{UnboxedTuple}$$

$$\frac{\Gamma \vdash e : s^u \dashv C}{\Gamma \vdash \text{unary\_op }(e) : t^v \dashv C}\text{UnOp} \qquad \frac{\Gamma \vdash e_1 : s^{u_1} \dashv C_1 \qquad \Gamma \vdash e_2 : s^{u_2} \dashv C_2}{\Gamma \vdash \text{binary\_op }(e_1, e_2) : t^v \dashv C_1 \cup C_2}\text{BinaryOp}$$

$$\frac{\Gamma \vdash e_1 : t_1^{u_1} \dashv C_1 \qquad \Gamma, x : t_1^{u_1} \vdash e_2 : a \dashv C_2 \qquad u_2 = \text{sharing\_analysis}(x, e_2)}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : a \dashv \{(u_1, u_2)\} \cup C_1 \cup C_2}\text{Let}$$

$$\frac{\Gamma \vdash e_1 : \text{bool}^{u_1} \dashv C_1 \qquad \Gamma \vdash e_2 : t^{u_2} \dashv C_2 \qquad u = \text{fresh\_unique}()}{\Gamma \vdash \text{if}\, e_1\, \text{then}\, e_2 : t^u \dashv \{(u, u_2)\} \cup C_1 \cup C_2}\text{If}$$

$$\frac{\Gamma \vdash e_1 : \text{bool}^{u_1} \dashv C_1 \qquad \Gamma \vdash e_2 : t^{u_2} \dashv C_2 \qquad \Gamma \vdash e_3 : t^{u_3} \dashv C_3 \qquad u = \text{fresh\_unique}()}{\Gamma \vdash \text{if}\, e_1\, \text{then}\, e_2\, \text{else}\, e_3 : t^u \dashv \{(u, u_2), (u_2, u_3)\} \cup C_1 \cup C_2 \cup C_3}\text{IfElse}$$

$$\frac{\Gamma \vdash e : a \dashv C}{\Gamma, x : t^u \vdash \text{drop } x;\ \ e : a \dashv \{(u, \texttt{unique})\} \cup C}\text{Drop}$$

$$\frac{\Gamma \vdash e : a \dashv C \qquad k \in \mathbb{N}}{\Gamma \vdash \text{free } k;\ \ e : a \dashv C}\text{Free}$$

$$\frac{\Gamma \vdash e : a \dashv C \qquad k \in \mathbb{N}}{\Gamma \vdash \text{weak } k;\ \ e : a \dashv C}\text{Weak} \qquad \frac{\Gamma \vdash e : a \dashv C \qquad k \in \mathbb{N}}{\Gamma \vdash \text{inst } k;\ \ e : a \dashv C}\text{Inst}$$

Appendix D

Implementing the fip and fbip environments

D.1 Modelling the owned context Γ as a set

Similar endpoints may be defined for modelling the borrowed context Δ.

```
(* Snippet from src/frontend/typing/owned_context.mli *)
 module OwnedSet : Set.S with type Elt.t = Var_name.t
(** Model the context handling owned variables which have
        non-primitive types as a set. *)

val extend_owned_set :
  element:Var_name.t ->
  element_type_expr:type_expr ->
  owned_set:OwnedSet.t ->
  OwnedSet.t Or_error.t
(** Given an [element] and an owned context [owned_set], extend
        the context or throw error if duplicates exist. *)

val extend_owned_set_by_list :
  elements:Var_name.t list ->
  elements_type_exprs:type_expr list ->
  owned_set:OwnedSet.t ->
  OwnedSet.t Or_error.t
(** Given a bunch of [elements] and an owned context [owned_set],
      extend the context with the elements or throw error if duplicates exist. *)

val combine_owned_sets :
  owned_set1:OwnedSet.t -> owned_set2:OwnedSet.t -> OwnedSet.t Or_error.t
(** Performs union of the two sets, throws error if their
        intersection is not the empty set. *)
```

```
val remove_element_from_owned_set :
  element:Var_name.t ->
  element_type_expr:type_expr ->
  owned_set:OwnedSet.t ->
  OwnedSet.t Or_error.t
(** Remove [element] from [owned_set], returns new set or throws
        error if the element was not in the set to beginning with. *)

val assert_owned_sets_are_equal :
  owned_set1:OwnedSet.t -> owned_set2:OwnedSet.t -> unit Or_error.t

val pprint_owned_set : Format.formatter -> indent:string -> OwnedSet.t -> unit
```

D.2 Modelling the reuse credit context ρ as a map

```
module ReuseMap : Map.S with type Key.t = int
(** Model the context handling reuse credits as a map from reuse
        credit size to how many there are available *)

type reuse_map_entry = int * Var_name.t list

val allocation_credit_size : unit -> int
(** Constant function that returns a unique identifier for the
        key of allocation credits *)

val extend_reuse_map :
  reuse_size:int ->
  reuse_var:Var_name.t ->
  reuse_map:reuse_map_entry ReuseMap.t ->
  reuse_map_entry ReuseMap.t
(** Extends the map of reuse credits to increase the [reuse_size]
        availability by 1. *)

val consume_reuse_map :
  reuse_size:int ->
  reuse_map:reuse_map_entry ReuseMap.t ->
  (Var_name.t * reuse_map_entry ReuseMap.t) Or_error.t
(** Consume one [reuse_size] credit by decreasing its availability by 1. *)

val combine_reuse_maps :
  reuse_map1:reuse_map_entry ReuseMap.t ->
  reuse_map2:reuse_map_entry ReuseMap.t ->
  reuse_map_entry ReuseMap.t
```

```
(** Combines two ReuseMaps into one. *)

val assert_reuse_maps_are_equal :
  reuse_map1:reuse_map_entry ReuseMap.t ->
  reuse_map2:reuse_map_entry ReuseMap.t ->
  unit Or_error.t

val pprint_reuse_map :
  Format.formatter -> indent:string -> reuse_map_entry ReuseMap.t -> unit
```

D.3 Static fip/fbip checker for IfElse statements

```
...
| IfElse (loc, _, cond_expr, then_expr, else_expr) ->
    (* Synthesise owned and reuse credit environments for cond_expr *)
      fip_rules_check_expr cond_expr borrowed_set functions_env
      >>= fun fip_cond_expr ->
      let _, cond_owned_set, cond_reuse_map =
        Fip_ast.get_fip_contexts_from_expr fip_cond_expr
      in

    (* Synthesise owned and reuse credit environments for then_expr *)
      fip_rules_check_expr then_expr borrowed_set functions_env
      >>= fun fip_then_expr ->
      let _, then_owned_set, then_reuse_map =
        Fip_ast.get_fip_contexts_from_expr fip_then_expr
      in

    (* Synthesise owned and reuse credit environments for else_expr *)
      fip_rules_check_expr else_expr borrowed_set functions_env
      >>= fun fip_else_expr ->
      let _, else_owned_set, else_reuse_map =
        Fip_ast.get_fip_contexts_from_expr fip_else_expr
      in

    (* Assert that the owned sets of both branches are equal. *)
      Or_error.ok_exn
        (assert_owned_sets_are_equal ~owned_set1:then_owned_set
           ~owned_set2:else_owned_set);

    (* Assert that the reuse credit maps of both branches are equal. *)
      Or_error.ok_exn
        (assert_reuse_maps_are_equal ~reuse_map1:then_reuse_map
          ~reuse_map2:else_reuse_map);

    (* Combine the reuse credit maps of the cond_expr and the branches  *)
      let combined_reuse_map =
        combine_reuse_maps ~reuse_map1:cond_reuse_map ~reuse_map2:then_reuse_map
      in

    (* Combine the owned sets of the cond_expr and branches
          while checking to preserve linearity. *)
      let combined_owned_set =
        Or_error.ok_exn
          (combine_owned_sets ~owned_set1:cond_owned_set
             ~owned_set2:then_owned_set)
      in

    (* Construct Fip_ast intermediate representation. *)
      Ok
        (Fip_ast.IfElse
           ( loc,
             borrowed_set,
             combined_owned_set,
             combined_reuse_map,
             fip_cond_expr,
             fip_then_expr,
             fip_else_expr ))
...
```

Appendix E

Project proposal

Introduction and description

Functional programming has lots of attributes that make it so popular among us, Computer Science enthusiasts. Unfortunately, whenever we are coding, we have a choice to make. We can either maintain a purely functional coding style, or aim for efficiency by using in-place updates, therefore abandoning the pursuit of data immutability and referential transparency. This problem is identified and tackled in a research paper(Lorenzen, Leijen, and Swierstra 2023), forming the foundation of my project.

My project will implement the ideas presented in the paper, aiming to validate the hypothesis of achieving optimised purely functional programming under specific conditions. Therefore, the end result of my project will be a programming language able to solve the previously mentioned problem, targeting OCaml's back-end, Flambda (OCaml 2024). Hence, we may divide the project into two distinct parts: core and extensions.

The core of the project is focused on creating a programming language which enables functions to be executed fully in-place, where that is possible. The cited paper argues that this is achievable given that all of a function's owned parameters are unique and not shared. Furthermore, it performs this by dynamically checking using Perceus reference counting. The ingenuity of this project rests in being able to perform the check statically and executing the function fully in-place if the check is passed. In order to achieve this, we firstly need to build the compiler's front-end, mainly the lexer and parser, followed by connecting with OCaml's back-end in order to be able to execute arbitrary code. The details of how to implement the static check are not explored by the supporting research paper, but it suggests using a uniqueness type system, similar to the one present in Clean(University 2024).

After building the core of the project, we will be able to design experiments for evaluating language's performance. It would be insightful to compare the in-place execution against its non- in-place variant from OCaml, Haskell, Scala and even C on relevant metrics, such as execution time, peak memory usage, number of allocations and more, using profilers. These experiments will consists of recording the specified execution metrics on different algorithms and data structures, such as Red-Black Tree, Finger Tree, Quicksort, Mergesort and mapping a function onto a generic tree.

For extensions, I plan on re-implementing the dynamic checking using Perceus reference counting, as done in the research paper. By doing so, we may evaluate the performance of the execution using the static check against the dynamic one on the designed experiments and expect to observe that the former approach is faster due to the overhead on the latter. Furthermore, this allows us to investigate if the static analysis validates as many functions to be executed in-place as the dynamic one does. Other extensions that I consider implementing include introducing OCaml's equivalent of references in the new language, Tail Recursion Modulo Cons optimisation (or even more generalised Tail Modulo Constructor), as well as enabling polymorphism for the typing system. It is worth mentioning that I do not expect to implement all of the extensions, due to the time limit imposed by the project, but will choose nearer the time which to focus on.

Starting point

As mentioned, my project expands on the findings of a research paper(Lorenzen, Leijen, and Swierstra 2023), providing an implementation of its theoretical details. Furthermore, open source projects such as Flambda(OCaml 2024) and Malfunction(Dolan 2016) could be used when building the compiler targeting the OCaml's backend.

Nevertheless, concepts presented in courses such as **Compiler Construction** and **Optimising Compilers** will be used as sources of information.

Resources required

Throughout the project, I will be using my personal computer (Mac Book Pro 2021, 10-Core M1 Max, 64GB RAM), for both code development and dissertation writing. In case of any unexpected failures, I plan on using another personal computer, without losing any of the current progress.

I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure.

The project will always be saved in Cloud using the Git technology in a private repository on GitHub. Furthermore, copies of the project will periodically be saved offline, on USB drives. A similar approach will be taken with respect to the dissertation paper, continuously uploading it into GitHub, Google Drive and Overleaf, as well as regularly creating offline copies.

Success criteria

The project will be a success if the resulting functional programming language is able to correctly identify and run functions that can be executed fully in-place, using the static checking method, under specific conditions such as the ones hypothesised by the supporting research paper.

Apart from this, the project should be able provide information about well defined execution metrics and report significant findings when comparing the fully in-place execution against the non- in-place variants from other functional languages, but not limited only to.

Therefore, we may list the key-points of a successful core project into the following:

- Design a grammar and implement a lexer and a parser for the new simple functional programming language.
- Implement the static analyses to determine opportunities for in-place updates.
- Implement a compiler targeting Ocaml's back-end that takes advantage of the in-place update opportunities.
- Evaluate the performance of the generated code with and without in-place update optimisations.

Similarly, we may define key-points to be achieved by the proposed project extensions. As previously mentioned, it is hard to predict upfront how many of these extensions will be attempted due to the time constraint imposed by the project, but I can confirm that I will prioritise the first one over the others:

- Implement the dynamic analyses checker using Perceus reference counting.
  - Evaluate performance of the static checker against the dynamic one.
  - Provide insight into the relative strength of the static check compared the dynamic one.
- Implement the OCaml equivalent of references.
- Implement Tail Recursion Modulo Cons optimisation. Alternatively, Tail Modulo Constructor optimisation could be attempted as well.
- Enable polymorphism for the typing system.

Timetable

1. **Michaelmas weeks 1–2 (3rd October - 16th October)**

   Planned work

   - Setup GitHub repository, configure OCaml and IDEs for local development.
   - Further research into the supporting paper, gathering a deeper understanding of the semantics and ideologies presented.
   - Deciding on a language grammar and further research into pre-requisites for building the lexer and the parser.
   - Research on Flambda and Malfunction and how will my compiler be able to make calls to OCaml's back-end.

Milestones

- Submit draft and full proposal.
- Having a fully configured initial development environment.

Deadlines

- 6th October - Full Draft Proposal submission
- 16th October - Final Proposal submission

2. **Michaelmas weeks 3–4 (17th October - 30th October)**

   Planned work

   - Settle on a language grammar.
   - Build the lexer for the compiler.

   Milestones

   - Have a fully functional lexer which is able to tokenize the project's new programming language.

3. **Michaelmas weeks 5–6 (31st October - 13th November)**

   Planned work

   - Fix any of the emerging issues with the lexer.
   - Build the parser for the compiler.
   - Further investigate how to connect with OCaml's back-end, Flambda.

   Milestones

   - Have a fully functional parser, therefore completing the compiler's front-end.

4. **Michaelmas weeks 7–8 (14th November - 27th November)**

   Planned work

   - Further research into ways of implementing the static analysis for determining whether a function can be executed in-place.
   - Initial steps of connecting the emerging programming language with OCaml's back-end.

   Milestones

   - Better understanding of possible approaches to performing the static check, along an initial implementation of some of the researched ideas.
   - Partial targeting to OCaml, enabling simple code to be executed.

5. **Michaelmas week 9 – Christmas Vacation (28st October - 11th December)**

   Planned work

   - Start implementing ideas found as part of the research.
   - Connect compiler's front-end with OCaml's back-end for arbitrary code execution.

   Milestones

   - Finalise the core part of the compiler.
   - Create initial experiments and record execution metrics.

6. **Christmas Vacation (12th December - 15th January)**

   Planned work

   - Further research into the static check.
   - Finalise the static check implementation.
   - Fully enabling programs to be executed via OCaml, including optimised in-place execution for functions that pass the static check.
   - Implement part of the evaluation experiments, record execution metrics and prepare intermediate report for comparison with other programming languages.

   Milestones

   - Finalise the core part of the compiler.
   - Create initial experiments and record execution metrics.

7. **Lent weeks 1–2 (16th January - 29th January)**

   Planned work

   - Fix any of the emerging issues with the static check.
   - Finalise implementing all the experiments and perform recordings as planned.

   Milestones

   - Finalise the core part of the project.

8. **Lent weeks 3–4 (30th January - 12th February)**

   Planned work

   - Start researching ways of approaching the first extension, implementing the dynamic check presented in the support paper using Perceus reference counting.
   - Write and finish the Progress Report.
   - Prepare for Progress Report Presentations.

   Milestones

- Have a deeper understanding of the requirements and design of the first extension and be prepared for implementation.

Deadlines

- 2nd February - Progress Report submission

9. **Lent weeks 5–6 (13th February - 26th February)**

   Planned work

   - Finalise first extension implementation, have the dynamic checker ready and connected with OCaml's back-end.
   - Perform experiments on the new functionality, comparing execution metrics against static checking version and other programming languages.

   Milestones

   - Finish the first extension.
   - Gather new execution metrics and prepare initial reports for *Evaluation*.
   - Begin working on the dissertation, starting with the *Introduction* chapter and producing an initial version of it.

10. **Lent weeks 7–8 (27th February - 11th March)**

    Planned work

    - Decide on the second extension to be implemented.
    - Research into the second extension.
    - Initial implementation of the second extension.
    - Begin writing the *Preparation* chapter of the dissertation.

    Milestones

    - Finish the second extension.
    - Finalising the *Introduction* chapter, in light of feedback.

11. **Lent week 9 - Easter Vacation (12th March - 25th March)**

    Planned work

    - Fix any of the emerging issues with the second extension.
    - Modify the *Preparation* chapter, in light of received feedback.
    - Begin writing the *Implementation* chapter.

    Milestones

    - Have a perfectly functioning second extension.
    - Completely finalise the *Preparation* chapter.

12. **Easter Vacation (26th March - 22nd April)**

    Planned work

    - Further implementation on remaining extensions.
    - Full completion of the *Implementation*, *Evaluation* and *Conclusions* chapters, incorporating any received feedback.

    Milestones

    - Finalise the *Implementation*, *Evaluation* and *Conclusions* chapters of the dissertation.

13. **Easter weeks 1–2 (23th April - 6th May)**

    Planned work

    - Fixing any last minute emerging problems with any extensions.
    - Integrating the changes into the correct chapters of the dissertation.
    - Prepare further documents required to be appended to the dissertation.
    - Perform any last changes to any of the chapters, in light of new feedback.

    Milestones

    - Finalise a penultimate version of the dissertation and have it ready for submission.

14. **Easter weeks 3 (7th May - 13th May)**

    Planned work

    - Perform any last changes by incorporating any feedback left.
    - Further proof reading followed by on-time submission.

    Deadlines

    - 10th May - Dissertation Deadline (electronic)
    - 10th May - Source Code Deadline (electronic copies)