Pirouette Compiler: Design and Implementation of a Compiler for Higher-Order Functional Choreographies

by

Yining Zhang

May 2025

A project report submitted to the
faculty of the Graduate School of
the University at Buffalo, The State University of New York
in partial fulfillment of the requirements for the
degree of

Master of Science
Department of Computer Science

Abstract

This report presents the development of a compiler infrastructure for Pirouette using OCaml. It will be a practical tool that enables programmers to write and execute choreographic programs; and also, a foundational framework for researching and developing optimization techniques specifically tailored to choreographic programming. The compiler is backend-agnostic and can be configured to use multiple communication channel backends. This infrastructure validates the theoretical underpinnings of choreographic programming and also serves as a testbed for exploring how the unique characteristics of choreographies.

Table of Contents

List of Equations

List of Listings

1 | Introduction

1.1 Choreographic Programming

Modern software development is growing more complex, and multi-core processors are now more common. To handle this complexity and tap into modern hardware, developers are turning to concurrent programming paradigms. Concurrency lets different parts of a program run at the same time, improving performance and responsiveness. Many ways exist to structure concurrent systems. One popular approach is message passing. Alongside traditional multi-threading, message-passing architectures, where independent components communicate by sending messages rather than sharing memory directly, offer a way to structure complex interactions. [9]

Concurrent programming is powerful, but it can also introduce challenges for developers, particularly concerning message ordering and deadlocks. Keeping the right order of messages between concurrent entities, such as actors, is crucial. Message order violations or bad interleavings may come from unexpected delays or execution overlaps. Such problems can lead to incorrect states or even system failures. Deadlocks are another big problem. They happen when parts of a program are blocked waiting for each other. Identifying these concurrency issues is notoriously difficult. Their non-deterministic nature means they often manifest only under specific, rare timing conditions, making them hard to reproduce consistently. The inherent complexity and elusive nature of these bugs typically require considerable time and effort to detect, diagnose, and resolve. And there is a lack of specialized frameworks to help. [6, 9]

Choreographic programming, represents a emerging paradigm in developing concurrent and distributed systems, moving from writing the program for each participant individually towards specifying the system's interactions from a unified, global perspective [1]. In the choreographic paradigm, the programmer writes a *choreography*, which serves as a blueprint detailing the sequence of communications and interactions among all participating roles (or *endpoints*) [3].

This choreography raises the level of abstraction and offers an objective view of the entire system's communication procedure.

A choreography undergoes a compilation process called *Endpoint Projection (EPP)*, which transforms the choreography into individual implementations deployed across distributed nodes. When endpoint projection maintains soundness, the resulting system inherently guarantees deadlock freedom — a fundamental property ensuring that every message sent will be matched with a corresponding receive operation[1]. Beyond this core benefit, choreographies facilitate comprehensive whole-program analyses that can systematically identify and eliminate various potential bugs, and simultaneously creating many opportunities for verification and performance optimization.

1.2 The Pirouette Language

Pirouette [4] is the first language for choreographic programming which is higher-order, functional, and statically typed. Pirouette is defined generically over a local language of messages, and lifts guarantees about the message type system to its own. The language's type soundness and the properties of endpoint projection guarantee *deadlock freedom*, which is also formally proven.

The communication model in Pirouette centers around message passing between participants. The message passing operator `~>` combines sending and receiving into a single construct, ensuring by design that communication endpoints are properly matched.

A distinguishing feature of Pirouette is its support for higher-order programming. Functions in Pirouette can accept both local values and entire choreographies as arguments. This enables powerful abstraction capabilities not available in previous choreographic languages.

To demonstrate the Pirouette language, consider a simple ping-pong protocol that parameterizes an interaction between `Alice` and `Bob`. `Alice` sends an integer to `Bob`; `Bob` applies a function to this integer and sends the result back to `Alice`, who then prints it. We can implement this protocol using a higher-order Pirouette function, `make_pingpong`, which takes `Bob`'s processing function (`handler`) as an argument:

```
make_pingpong : (Bob.int -> Bob.int) -> (Alice.int -> Alice.unit);

make_pingpong handler :=
  let do_pingpong Alice.value :=
    [Alice] Alice.value ~> Bob.received;
    let Bob.result := handler Bob.received; in
    let Alice.response := [Bob] Bob.result ~> Alice; in
    Alice.print_int Alice.response;
  in do_pingpong
;

main :=
  let Alice.choice := Alice.read_line Alice.(); in
  if Alice.(choice = "double") then
    Alice[DOUBLE] ~> Bob;
    make_pingpong (fun Bob.x -> Bob.(x * 2)) Alice.5
  else
    Alice[ADDONE] ~> Bob;
    make_pingpong (fun Bob.x -> Bob.(x + 1)) Alice.5
;
```

Listing 1: Pirouette Program for Ping-Pong Interaction.

The type signature `(Bob.int -> Bob.int) -> (Alice.int -> Alice.unit)` shows Pirouette's use of located types. It specifies that `make_pingpong` accepts a function (`handler`) that takes an integer located at `Bob` (`Bob.int`) and returns another integer at `Bob`. The `make_pingpong` function itself returns a new function (the actual choreography `do_pingpong`) which expects an integer located at `Alice` (`Alice.int`) and ultimately results in a unit value located at `Alice` (`Alice.unit`), signifying the completion of Alice's role (printing the result). This use of a function argument (`handler`) showcases the higher-order nature of Pirouette, allowing parts of the protocol's logic to be abstracted and passed in as data.

Inside the do_pingpong function body:

1. `[Alice] Alice.value ~> Bob.received` denotes a communication step. The location in brackets (`[Alice]`) indicates the sender. `Alice.value` is the value being sent (bound to the input of `_`). `Bob.received` signifies that the value is received by `Bob` and bound to the local variable received in `Bob`'s scope.

2. `let Bob.result := handler Bob.received; in` represents a local computation occurring only at `Bob`. `Bob` applies the handler function (passed into `make_pingpong`) to the received value `Bob.received`, binding the output to `Bob.result`.

3. `let Alice.response := [Bob] Bob.result ~> Alice; in` shows the return communication. `Bob` sends the computed `Bob.result` to `Alice`, who receives it as `Alice.response`. The `let ... := ... ~> ...; in` syntax combines the send, receive, and binding into a single choreographic step.

4. `Alice.print_int Alice.response;` is another local computation, this time performed by `Alice` using the standard library function `print_int` on her received value `Alice.response`.

```
1 main :=
2   let Alice.choice := Alice.read_line Alice.(); in
3   if Alice.(choice = "double") then
4     Alice[DOUBLE] ~> Bob;
5     make_pingpong (fun Bob.x -> Bob.(x * 2)) Alice.5
6   else
7     Alice[ADDONE] ~> Bob;
8     make_pingpong (fun Bob.x -> Bob.(x + 1)) Alice.5
9 ;
```

Listing 2: Main function of the ping-pong choreography with protocol variations.

Pirouette's endpoint projection transforms these choreographies into separate programs (NetIR) for each participant. For the ping-pong example, the projection produces the following NetIR code for `Alice`:

```
make_pingpong handler :=
  let do_pingpong value :=
    ret value ~> Bob;
    let response := <~ Bob in
    ret (print_int (ret response))
  in
  do_pingpong
;

main :=
  let choice := ret read_line ret () in
  if ret (choice = "double") then
    choose DOUBLE for Bob;
    make_pingpong (fun _ -> ()) ret 5
  else
    choose ADDONE for Bob;
    make_pingpong (fun _ -> ()) ret 5
;
```

Listing 3: `Alice`'s endpoint code of the ping-pong choreography.

And for `Bob`, the projection yields:

```
make_pingpong handler :=
  let do_pingpong _ :=
    let received := <~ Alice in
    let result := handler received in
    ret result ~> Alice
  in
  do_pingpong;

main :=
  match choice from Alice with
  | DOUBLE -> make_pingpong (fun x -> x * 2) ()
  | ADDONE -> make_pingpong (fun x -> x + 1) ()
;
```

Listing 4: `Bob`'s endpoint code of the ping-pong choreography.

The type system ensures that these communications are well-formed and that distributed control flow is coordinated correctly. This projection demonstrates how the original choreography is accurately projected into programs for each node. At `Alice`'s endpoint, we see explicit send operations (`~>`) and receive operations (`<~`). Similarly, at `Bob`'s endpoint, we see the comple-

mentary receive from `Alice` and send back to `Alice`, along with the handler function that processes the received value.

When a conditional expression evaluates a condition at one location, other locations involved in either branch must be informed of the decision, ensuring all participants follow the same execution path. The control flow branching in the main function is coordinated by the `choose` construct at `Alice`'s side and the corresponding `match choice` at `Bob`'s side.

By unifying functional and choreographic programming, Pirouette offers an expressive yet safe approach to distributed programming. Programmers can leverage familiar and convenient functional programming concepts while gaining the coordination benefits of choreographic programming. This makes Pirouette particularly suitable for implementing complex distributed protocols where correctness guarantees are essential.

2 | Implementation

The compilation pipeline of Pirouette consists of the following phases:

1. **Lexical and Syntactic Analysis:** The compiler's front-end employs a lexer and parser to analyze the source code and construct a Choreography Abstract Syntax Tree (AST), which represents local expressions elevated to the choreographic level.

2. **Location Analysis:** The compiler traverses the complete AST structure to identify and catalog all distinct locations specified in the choreographic program.

3. **Endpoint Projection:** For each identified location, the compiler performs endpoint projection, transforming choreographic constructs into location-specific implementations in an intermediate representation called NetIR.

4. **Code Generation:** The compiler translates each node's NetIR into OCaml code, preserving the communication patterns and computational logic specified in the original choreography.

5. **Target Compilation:** The generated OCaml code is subsequently processed by the OCaml compiler, producing executable binaries for each network endpoint.

2.1 Hierarchically Composed ASTs

The implementation of choreographic programming in Pirouette requires an Abstract Syntax Tree (AST) architecture that reflects the dual nature of the paradigm: global choreographies and their projection to local behaviors. In the compiler, three interconnected AST structures are defined — Local, Choreo, and NetIR — forming a complete representation for distributed computations.

The Local AST serves as the foundation, defining an expression-based language with standard constructs such as values (integers, strings, booleans), operators, pattern matching, and basic data structures (products and sums). This local language represents the computational capabilities of individual nodes in the distributed system.

$$
\begin{aligned}
\tau_L &::= \text{unit} \mid \text{int} \mid \text{string} \mid \text{bool} \\
&\quad \mid \tau_L \times \tau_L \mid \tau_L + \tau_L \\
p_L &::= \_ \mid v \mid x \mid (p_L, p_L) \mid \text{left}(p_L) \mid \text{right}(p_L) \\
e_L &::= () \mid v \mid x \mid op_1(e_L) \mid op_2(e_L, e_L) \\
&\quad \mid \text{let } x = e_L \text{ in } e_L \\
&\quad \mid (e_L, e_L) \mid \text{fst}(e_L) \mid \text{snd}(e_L) \\
&\quad \mid \text{left}(e_L) \mid \text{right}(e_L) \\
&\quad \mid \text{match } e_L \text{ with } \{p_L \Rightarrow e_L\}^+ \\
v &::= \mathit{integer} \mid \mathit{string} \mid \mathit{boolean} \\
op_1 &::= - \\
op_2 &::= + \mid - \mid * \mid / \mid = \mid \leq \mid \geq \mid \neq \mid > \mid < \mid \&\& \mid \|
\end{aligned}
$$

Equation 1: Abstract syntax of the Local language.

Building upon this foundation, the Choreo AST encapsulates the global choreographic specifications by introducing location-aware constructs. It extends the type system with TLoc to represent location-specific types and introduces specialized expressions like LocExpr to execute local computations at specific locations (corresponding to the $\ell.e$ syntax), Send to represent communication between locations ($\ell 1.e\ \ell 2.x$; C), and Sync for synchronization events. Notably, the Let construct corresponds to the let $\ell.x\ C1$ in $C2$ syntax, binding values computed at specific locations for use in subsequent choreographies.

$$
\begin{aligned}
\tau_C ::= \ & \text{unit} \mid \ell.\tau_L \mid \tau_C \mapsto \tau_C \\
& \mid \tau_C \times \tau_C \mid \tau_C + \tau_C \\
p_C ::= \ & \_ \mid x \mid \ell.p_L \mid (p_C, p_C) \\
& \mid \text{left}(p_C) \mid \text{right}(p_C) \\
e_C ::= \ & () \mid x \mid \ell.e_L \\
& \mid \ell_1.e_C \rightsquigarrow \ell_2 \\
& \mid \text{if } e_C \text{ then } e_C \text{ else } e_C \\
& \mid \text{let } \overline{s} \text{ in } e_C \\
& \mid \overline{p_C} \mapsto e_C \mid e_C(e_C) \\
& \mid (e_C, e_C) \mid \text{fst}(e_C) \mid \text{snd}(e_C) \\
& \mid \text{left}(e_C) \mid \text{right}(e_C) \\
& \mid \text{match } e_C \text{ with } \{p_C \Rightarrow e_C\}^+ \\
s_C ::= \ & p_C : \tau_C \\
& \mid \overline{p_C} := e_C
\end{aligned}
$$

Equation 2: Abstract syntax of the Choreo language.

By Endpoint Projection, every node runs programs in a local expression-based language. So the third component, NetIR AST, is also built upon the Local AST. It represents the translation of global choreographies into location-specific behaviors. NetIR introduces communication primitives like Send and Recv that replace the unified Send from the choreography, and distributed choice mechanisms (ChooseFor and AllowChoice) that implement the projection of conditional behaviors across multiple participants.

$$
\begin{aligned}
\tau_E ::= {} & \text{unit} \mid \tau_L \mid \tau_E \mapsto \tau_E \\
& \mid \tau_E \times \tau_E \mid \tau_E + \tau_E \\
e_E ::= {} & () \mid x \mid \text{ret}(e_L) \\
& \mid \text{if } e_E \text{ then } e_E \text{ else } e_E \\
& \mid \text{let } \overline{s_E} \text{ in } e_E \\
& \mid \text{send}_\ell\ e_E \mid \text{recv}_\ell() \\
& \mid \text{choose}_\ell\ L; e_E \mid \text{allow}_\ell\ \{L : e_E\}^+ \\
& \mid \overline{p_L} \mapsto e_E \mid e_E(e_E) \\
& \mid (e_E, e_E) \mid \text{fst}(e_E) \mid \text{snd}(e_E) \\
& \mid \text{left}(e_E) \mid \text{right}(e_E) \\
& \mid \text{match } e_E \text{ with } \{p_L \Rightarrow e_E\}^+ \\
s_E ::= {} & p_L : \tau_E \\
& \mid \overline{p_L} := e_E
\end{aligned}
$$

Equation 3: Abstract syntax of the Net IR.

This tiered AST architecture enables the complete representation and transformation pipeline from high-level choreographic specifications to executable distributed programs, maintaining the semantic correspondence between the global view of the computation and its local realizations at each endpoint.

In compiler construction, Abstract Syntax Trees (ASTs) typically undergo multiple transformations across compilation phases, with each pass potentially requiring a different representation to accommodate phase-specific metadata. Typically we need to define distinct AST types for each phase, resulting in significant code duplication and challenging maintenance, e.g., we need to write pretty-printinghelper functions for each AST definition. We leverage OCaml's Generalized Algebraic Data Types (GADTs) and module functors to address this challenge elegantly. Our approach parameterizes AST node definitions with a type variable representing metadata, allowing the core structure to remain invariant while metadata evolves across compilation phases. This architecture enables the compiler to maintain a single definition of AST structure

while attaching different metadata to nodes at each phase, rather than constructing entirely new AST types.

The implementation begins with a parameterized definition of AST node types. As shown in the following excerpt, each node carries metadata of type `'m`:

```
type 'm expr =
  | Unit of 'm
  | Val of 'm value * 'm
  | Var of 'm var_id * 'm
  | UnOp of 'm un_op * 'm expr * 'm
  | BinOp of 'm expr * 'm bin_op * 'm expr * 'm
  | Let of 'm var_id * 'm expr * 'm expr * 'm
  | Pair of 'm expr * 'm expr * 'm
  | Match of 'm expr * ('m pattern * 'm expr) list * 'm
```

Listing 5: Parameterized AST definition with metadata type variable `'m`

The implementation relies on a functor-based design that instantiates AST types with specific metadata through a parameterized module. This is achieved through the `With` functor, which expects a module providing a metadata type:

```
module With (Info : sig
    type t
  end) =
struct
  type nonrec value = Info.t M.value
  type nonrec expr = Info.t M.expr

  let get_info_expr : expr -> Info.t = function
    | Unit i -> i
    | Val (_, i) -> i
    | Var (_, i) -> i
    | UnOp (_, _, i) -> i
    | BinOp (_, _, _, i) -> i
    | Let (_, _, _, i) -> i
    (* ... other cases ... *)
  ;;

  let set_info_expr : Info.t -> expr -> expr =
    fun i -> function
    | Unit _ -> Unit i
    | Val (v, _) -> Val (v, i)
    | Var (x, _) -> Var (x, i)
    | UnOp (op, e, _) -> UnOp (op, e, i)
    | BinOp (e1, op, e2, _) -> BinOp (e1, op, e2, i)
    (* ... other cases ... *)
  ;;
end
```

Listing 6: The `With` functor that instantiates AST nodes with specific metadata types.

When transitioning between compiler phases, a new AST type can be instantiated with phase-specific metadata. For example, during parsing, we might attach source location information:

```
module Pos_info = struct
  type t =
    { fname : string
    ; start : int * int (* line, column *)
    ; stop : int * int (* line, column *)
    }
end

module Local = Ast_core.Local.With (Pos_info)
```

Listing 7: Instantiation of the AST with source position information for the parsing phase.

This instantiation yields a new AST module with nodes carrying position information. Later phases can introduce different metadata types without altering the AST structure. For instance, a typing phase might replace position information with type annotations:

```
module Type_info = struct
  type t = ty option (* Option containing inferred type *)
end

module Typed_ast = Ast_core.Local.With (Type_info)
```

Listing 8: Instantiation of the AST with type information for the typing phase.

Type safety is preserved across transformations through the functor's type constraints and the OCaml type system's static guarantees. The accessor and modifier functions such as and ensure that metadata is handled consistently for each node type:

```
let transform_expr (e : Local.expr) : Typed_ast.expr =
  match e with
  | Local.Var (x, _) ->
      let x' = transform_var_id x in
      Typed_ast.Var (x', Some inferred_type)
  | Local.BinOp (e1, op, e2, _) ->
      let e1' = transform_expr e1 in
      let e2' = transform_expr e2 in
      let op' = transform_binop op in
      Typed_ast.BinOp (e1', op', e2', Some result_type)
  (* ... other cases ... *)
```

Listing 9: Transformation function that converts AST nodes with different metadata types.

Our design yields substantial code reuse benefits, as traversal algorithms, pattern matching structures, and transformation logic can be written once and reused across different compilation phases. Beyond reducing code duplication, this approach enhances compiler robustness by limiting the surface for potential inconsistencies between AST representations.

Interestingly, after completing our implementation, we discovered that a similar approach was explored in the development of the Catala compiler[2], a DSL for implementing algorithms defined in law. The authors of that work also leverage GADTs to achieve a unified AST representation, though with some architectural differences. Their design centers around a parameterized

GADT `'kind Generic.expr` that represents all possible AST nodes across compilation passes, using phantom types as guards to characterize different families of terms. Their approach offers several notable advantages that align with our goals: it distinguishes intermediate ASTs at the type level for stronger compile-time guarantees, reduces boilerplate through systematic code reuse, and allows for comfortable AST manipulation through OCaml's natural pattern matching. While our implementation focuses on attaching different metadata to nodes through functors, their design emphasizes using phantom types to control which terms are allowed in different compilation phases. Both approaches demonstrate how OCaml's advanced type system features can be leveraged to create maintainable compiler architectures that balance expressiveness with type safety.

2.2 Parsing

2.2.1 Lexical Analyzer Generation

A lexical analyzer, often referred to as a scanner or tokenizer in compiler design, is the first phase of the compilation process responsible for converting a sequence of characters into meaningful lexical units called tokens. The process of lexical analysis divides the input text into these tokens according to a set of rules that define the language's lexical structure.

Traditionally, lexical analyzers were hand-coded, requiring developers to write complex character-by-character processing routines that could identify patterns in the input stream. Lexical analyzer generators [7] are specialized tools to automate this process, offering a more systematic and maintainable approach to building scanners. These generators accept high-level specifications of lexical patterns, typically expressed as regular expressions, and produce executable code that implements a lexical analyzer conforming to those specifications.

The generator transforms these abstract pattern definitions into efficient finite automata that can recognize the specified patterns in the input stream. When a pattern is recognized in the input text, the generated analyzer can execute associated semantic actions, which might include recording token information, manipulating symbol tables, or performing other context-specific operations.

The core principle behind lexical analyzer generators is the formal language theory concept that regular expressions can be systematically converted to deterministic finite automata (DFAs), which serve as efficient recognition engines for the patterns defined by those expressions.

The generator approach creates a clear separation between the specification of lexical structure (what patterns to recognize) and the implementation details of the recognition process, allowing language designers and compiler writers to focus on the language definition rather than low-level character processing algorithms.

The lexer definition in the compiler uses OCamllex[5], a lexical analyzer generator for the OCaml programming language. The lexer transforms source code into tokens through specifications consisting of regular expressions with associated semantic actions. The implementation comprises three main components: a header section containing auxiliary OCaml code for position tracking and error handling, pattern definitions that establish the vocabulary of recognizable lexemes, and rule definitions that specify the state transitions of the underlying finite automaton. The primary rule, read, handles most token recognition by matching patterns for language constructs and emitting appropriate token constructors. The lexer employs specialized rules for complex lexical structures including string literals (`read_string`) and both single-line and multi-line comments (`read_single_line_comment` and `read_multi_line_comment`). Error handling is integrated throughout the implementation, with explicit exception raising for unexpected characters and unterminated constructs. Position information is maintained to enable precise error reporting.

2.2.2 Parser Generator

An LR(1) parser generator transforms high-level grammar specifications into efficient parsers capable of processing input text according to the rules of a formal language. The “LR” designation indicates that the input is scanned from Left to right to produce a rightmost derivation, while the “(1)” signifies that parsing decisions require looking ahead at most one token. These parsers operate as deterministic pushdown automata, processing input tokens sequentially and constructing derivation trees in a bottom-up fashion. The performance characteristics of LR(1) parsers are highly predictable: assuming constant-time semantic actions, they achieve linear time complexity with respect to input size.[13]

At the core of LR(1) parsing lies the concept of item-sets, which characterize the states of the parser. Each state represents a collection of "LR(1) items" — essentially productions with a marker indicating how much of the production has been recognized, along with lookahead information. The parser generator constructs action and goto tables from these states, which together form the parsing automaton. The action table determines what the parser should do upon encountering each possible input token in each state (shift, reduce, accept, or error), while the goto table dictates which state to transition to after a reduction. This table-driven approach enables efficient parsing decisions without backtracking.[14]

Our parser is implemented using Menhir[10], a modern LR(1) parser generator for OCaml. Menhir transforms high-level grammar specifications, decorated with semantic actions, into deterministic pushdown automata parsers with predictable performance characteristics. Assuming constant-time semantic actions, our parser achieves linear time complexity with respect to input size.

Menhir's table-based LR(1) parsing approach ensures our implementation delivers deterministic and efficient parsing. The parser processes input tokens sequentially, building derivation trees bottom-up while using semantic actions to construct corresponding AST nodes that capture both syntactic structure and source locations.

The semantic actions in our implementation are fragments of OCaml code that execute during parsing to construct abstract syntax tree (AST) nodes. We maintain separate AST hierarchies for choreographic and local expressions, reflecting the dual nature of our language which must represent both communication patterns between participants and local computations. Position tracking is systematically integrated throughout the parser via the `gen_pos` function, which records precise source location information from the lexer for each syntactic construct. After parsing, a new AST module is generated with location metainformation attached to each AST node.

2.3 Endpoint Projection

At its core, the implementation transforms a global choreographic description into location-specific programs, maintaining the distributed semantics while ensuring each participant only sees their relevant part of the interaction.

The implementation centers around a recursive traversal of the choreographic syntax tree, where each construct is projected differently based on the target location. For local expressions (`LocExpr`), only the intended location receives the actual expression; all others receive an empty unit value. This follows the theoretical principle that local computations should only affect their respective participants.

Communication primitives represent the heart of choreographies. When projecting a `Send` operation between locations `loc1` and `loc2` , the implementation produces three distinct outcomes: the source location loc1 receives a `Send` operation, the target location `loc2` receives a corresponding Recv operation, and all other locations receive nothing. This directly corresponds to the theoretical notion that in a communication $\ell_1.e \rightsquigarrow \ell_2.x$, each location plays a specific role.

The `merge_net_expr` and `merge_net_stmt` functions implement the theoretical merging operator ($\sqcup$) that combines the control flow from different branches when a location is not the decision maker. For example, when projecting an If expression, the implementation first attempts to merge the projections of both branches. If merging succeeds, the location receives the merged program; otherwise, it receives a conditional statement that mirrors the original choreography's structure.

Synchronization primitives, represented by `Sync` in the choreographic language, are projected into `ChooseFor` and `AllowChoice` operations, which implement the theoretical choice propagation mechanism. The initiating location gets a `ChooseFor` operation, while the receiving location gets an `AllowChoice` operation with appropriate handlers for each choice label.

By systematically implementing these projection rules, the `NetGen` module transforms the unified choreographic view into a collection of communicating processes (`NetIR`) that collectively implement the specified behavior.

2.4 Code Generation

2.4.1 Code generation using Metaprogramming

Currently, our target language for code generation is OCaml, because our ASTs are very close to OCaml. We can build the compiler more quickly for further uses.

Metaprogramming[8, 12, 15] — the practice of writing programs that generate or manipulate other programs — represents a powerful paradigm in OCaml. In OCaml, metaprogramming enables developers to construct code using abstract syntax tree (AST) manipulations rather than string-based generation, offering greater safety and expressivity. The OCaml ecosystem has had various metaprogramming frameworks as well as Parsetree, a library for working with OCaml's abstract syntax tree, their APIs for representing ASTs are either outdated or unstable across OCaml versions. Ppxlib has emerged as the definitive solution to this challenge, providing a stable and complete framework for OCaml metaprogramming.

The core of our approach utilizes Ppxlib's `Ast_builder.Default` module, which provides functions for constructing various AST elements. For example, `eint`, `estring`, and `ebool` create expression nodes for literals, while `evar` constructs variable references. Our implementation uses these builders extensively in functions like `emit_local_pexp`, which transforms a Pirouette expression (`Local.expr`) into OCaml's expression:

```
let rec emit_local_pexp (expr : 'a Local.expr) =
  match expr with
  | Unit _ -> Ast_builder.Default.eunit ~loc
  | Val (Int (i, _), _) -> Ast_builder.Default.eint ~loc i
  | Val (String (s, _), _) -> Ast_builder.Default.estring ~loc s
  /* ... */
```

Listing 10: Function to transform Pirouette expressions into OCaml AST expressions.

Ppxlib's quotation syntax significantly simplifies AST construction. Rather than manually building complex expressions, quotations like `[%expr ...]` and `[%pat? ...]` allow us to write OCaml code that is automatically converted to AST nodes. This is particularly valuable for complex patterns, as seen in our pattern expression translation:

```
| Pair (p1, p2, _) -> [%pat? [%p emit_local_ppat p1], [%p emit_local_ppat p2]]
| Left (p, _) -> [%pat? Either.Left [%p emit_local_ppat p]]
```

Listing 11: Pattern expression translation using Ppxlib's quotation syntax.

After constructing the generated AST, we simply call Ppxlib's pretty printing function to get the generated OCaml code.

2.4.2 Backend-Agnostic Messaging Architecture

The compiler uses a backend-agnostic approach to generating code for concurrent/distributed nodes, where the core code generation functionality is parameterized by the messaging backend. This architecture leverages OCaml's first-class modules to abstract away the details of message passing, allowing the same code generator to target different execution environments without modification.

The module interface in `Msg_intf.ml` encapsulates the essential operations required for communication:

```
1 module type M = sig
2   val emit_net_send : src:string -> dst:string -> expression -> expression
3   val emit_net_recv : src:string -> dst:string -> expression
4 end
```

Listing 12: Message passing interface that abstracts send and receive operations.

This interface captures the minimal set of operations needed for message passing: sending a value from one location to another, and receiving a value from a specific source. It separates concerns between the core code generation logic and the specific details of message passing implementations.

By parameterizing the code generator with a first-class module implementing the messaging interface, we can generate code for different execution environments without changing the generator itself. This is a snippet in the `_` function, which handles the translation of network expressions:

```
and emit_net_pexp ~(self_id : string) (module Msg : Msg_intf) (exp : 'a
Net.expr) =
  match exp with
  (* ... other cases ... *)
  | Send (e, LocId (dst, _), _) ->
    let val_id = Id.gen "val_" in
    [%expr
      let [%p Ast_builder.Default.pvar ~loc val_id] =
        [%e emit_net_pexp ~self_id (module Msg) e]
      in
      [%e
        Msg.emit_net_send
          ~src:self_id
          ~dst
          [%expr Marshal.to_string [%e Ast_builder.Default.evar ~loc val_id]
[]]]]
  | Recv (LocId (src, _), _) ->
    [%expr Marshal.from_string [%e Msg.emit_net_recv ~src ~dst:self_id] 0]
  (* ... more cases ... *)
```

Listing 13: Code generation for NetIR expressions using a parameterized messaging module.

The implementation delegates to the provided messaging module for the actual code generation of send and receive operations, while handling the marshalling of data uniformly. This separation allows the core logic to remain focused on the transformation of the intermediate representation, while the message passing details are encapsulated in the backend module.

For concrete implementations, we can create modules that implement the messaging interface for specific backends. The example provided demonstrates a shared memory implementation using OCaml's domains and channels:

```
module Msg_chan_intf : Msg_intf.M = struct
  let emit_net_send ~src ~dst pexp =
    ignore dst;
    [%expr
      Domainslib.Chan.send
        [%e Ast_builder.Default.evar ~loc (spf "chan_%s_%s" src dst)]
        [%e pexp]]
  ;;

  let emit_net_recv ~src ~dst =
    [%expr
      Domainslib.Chan.recv [%e Ast_builder.Default.evar ~loc (spf "chan_%s_%s"
src dst)]]
  ;;
end
```

Listing 14: Implementation of the shared memory messaging interface.

This implementation generates code that uses channel-based communication between domains, with channels named according to the source and destination. The implementation details are completely hidden from the core code generator, which only needs to know about the abstract send and receive operations.

The toplevel code generation function then handles the creation of the overall program structure, including the initialization of the communication infrastructure and the spawning of domains. This architectural separation allows for different backend implementations to have different toplevel structures without affecting the core code generation logic. Under such an architecture, adding a message passing backend becomes relatively trivial; it only requires implementing the corresponding message interface and generating the top-level.

2.4.3 The Shared-Memory Backend

The shared memory backend implementation presented in our architecture leverages Multicore OCaml's domain system to provide efficient parallel execution with structured communication channels. Multicore OCaml extends the OCaml runtime with true parallel execution capabilities through its domain system, where each domain runs as a separate system thread within the same address space.[11]

In our implementation, we utilize domains to represent the distinct locations in a distributed system, but within a single process. The `emit_toplevel_domain` function generates code that

creates a domain for each location identifier, spawns the appropriate code within each domain, establishes communication channels between domains, and manages domain lifecycle:

```
let emit_toplevel_domain
      out_chan
      (loc_ids : string list)
      (net_stmtblock_l : 'a Net.stmt_block list)
  = ...
```

Listing 15: Signature of function that generates code to spawn Domains for each location.

Each domain is created using `.spawn`, which takes a function to execute within the new Domain. This is evident in the `_` function:

```
List.map2
  (fun loc_id net_stmts ->
     Ast_builder.Default.value_binding
       ~loc
       ~pat:(Ast_builder.Default.pvar ~loc (spf "domain_%s" loc_id))
       ~expr:[%expr Domain.spawn (fun _ -> [%e emit_net_toplevel loc_id
net_stmts])])
  loc_ids
  net_stmtblock_l
```

Listing 16: Code generation for spawning domains, mapping each location identifier to a Domain.

This code generates bindings for domains, where each domain executes the network statements associated with a particular location identifier. The important observation is that domains are first-class values in OCaml - they can be passed around, stored in data structures, and joined when needed.

I would add this paragraph after discussing the domain creation with `Domain.spawn` and before moving to the channel communication section:

A technical challenge in our implementation lies in transforming the list-based network IR representation into a suitable function for `.spawn`. Since `.spawn` requires a function of type `unit -> 'a`, but our NetIR represents code as a list of statements, we need to restructure the

code. The `emit_net_toplevel` function recursively processes network statements, converting them into OCaml let-bindings while searching for the main expression:

```
let main_expr = ref (Ast_builder.Default.eunit ~loc) in
let rec emit_net_toplevel loc_id stmts =
  match stmts with
  | [] -> !main_expr
  | stmt :: stmts ->
    (match Emit_core.emit_net_binding ~self_id:loc_id (module Msg_chan_intf)
stmt with
     | exception Emit_core.Main_expr e ->
       main_expr := e;
       emit_net_toplevel loc_id stmts
     | binding ->
       Ast_builder.Default.pexp_let
         ~loc
         Recursive
         [ binding ]
         (emit_net_toplevel loc_id stmts))
```

Listing 17: Implementation of function that wrapping NetIR statements for Domain execution.

This code detects assignments to the special variable “main” through an exception mechanism ( ), capturing the assigned expression as the return value while wrapping all other statements as let-bindings around it. The result is a properly structured function suitable for execution within a new domain, with all statements appearing as let-bindings surrounding the main expression.

For communication between domains, we employ the `.Chan` module, which provides typed channels for safe inter-domain communication. The channel mechanism offers a structured approach to exchanging data between domains without explicit locks. Our implementation creates a bidirectional channel for each pair of communicating locations:

```
let emit_chan_defs loc_ids =
  let loc_pairs =
    List.concat_map
      (fun a -> List.filter_map (fun b -> if a <> b then Some (a, b) else None) loc_ids)
      loc_ids
  in
  List.map
    (fun (a, b) ->
       Ast_builder.Default.value_binding
         ~loc
         ~pat:(Ast_builder.Default.pvar ~loc (spf "chan_%s_%s" a b))
         ~expr:[%expr Domainslib.Chan.make_bounded 0])
    loc_pairs
```

Listing 18: Function that generates code to create bounded communication channels.

We use bounded channels with capacity 0, creating synchronous communication points between domains. This design choice ensures that communication between domains is synchronized - a sending domain blocks until the receiving domain is ready to receive the message, which closely models the semantics of distributed message passing.

The implementation of the messaging interface for domains is straightforward, delegating to Domainslib's channel operations:

```
module Msg_chan_intf : Msg_intf.M = struct
  let emit_net_send ~src ~dst pexp =
    ignore dst;
    [%expr
      Domainslib.Chan.send
        [%e Ast_builder.Default.evar ~loc (spf "chan_%s_%s" src dst)]
        [%e pexp]]
  ;;

  let emit_net_recv ~src ~dst =
    [%expr
      Domainslib.Chan.recv [%e Ast_builder.Default.evar ~loc (spf "chan_%s_%s" src dst)]]
  ;;
end
```

Listing 19: Implementation of the messaging interface for Domains.

This code generates expressions that send and receive data through the named channels. The channel names follow a convention that encodes both the source and destination.

Domain termination is handled through the `.join` primitive, which blocks until a domain finishes execution. Our implementation generates code that joins all domains sequentially:

```
let rec emit_domain_join_seq = function
  | [] -> assert false
  | [ loc_id ] ->
    [%expr Domain.join [%e Ast_builder.Default.evar ~loc (spf "domain_%s"
loc_id)]]
  | loc_id :: loc_ids ->
    Ast_builder.Default.pexp_sequence
      ~loc
      [%expr Domain.join [%e Ast_builder.Default.evar ~loc (spf "domain_%s"
loc_id)]]
      (emit_domain_join_seq loc_ids)
```

Listing 20: Function that generates code to join all domains sequentially.

This approach ensures proper cleanup and prevents the program from terminating before all domains complete their work. Since domains share the same address space, memory allocated by one domain could potentially be accessed by another domain. However, our communication model uses channels exclusively, avoiding direct sharing of mutable state between domains and mitigating data races.

The generated code maintains a clean separation between the concurrent execution structure (domains) and the communication mechanism (channels), while adhering to the message-passing paradigm. This aligns with Multicore OCaml's design philosophy, which encourages explicit communication rather than shared memory synchronization. The result is a backend implementation that provides true parallelism with structured communication, serving as an effective testbed for distributed systems while remaining within a single process.

Glossary

***AST* – Abstract Syntax Tree**: TODO 7

***EPP* – Endpoint Projection**: TODO 2

Bibliography

[1] Marco Carbone and Fabrizio Montesi. 2013. *Deadlock-freedom-by-design: Multiparty Asynchronous Global Programming*. Association for Computing Machinery (ACM), New York, NY, USA Rome, Italy.

[2] Louis Gesbert and Denis Merigoux. 2023. *Modern DSL compiler architecture in OCaml: our experience with Catala*.

[3] Saverio Giallorenzo, Fabrizio Montesi, Marco Peressotti, David Richter, Guido Salvaneschi, and Pascal Weisenburger. 2021. *Multiparty Languages: The Choreographic and Multitier Cases*. Leibniz International Proceedings in Informatics (LIPIcs), Dagstuhl, Germany.

[4] Andrew K. Hirsch and Deepak Garg. 2022. Pirouette: Higher-Order Typed Functional Choreographies. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 1–27.

[5] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2025. Lexer and parser generators (ocamllex, ocamlyacc). *The OCaml Manual*.

[6] Carmen Torres Lopez, Stefan Marr, Hanspeter Mössenböck, and Elisa Gonzalez Boix. 2017. A Study of Concurrency Bugs and Advanced Development Support for Actor-based Programs. *arXiv* (2017), 1706.

[7] E. Lesk Michael and E. Schmidt Eric. 1990. *Lex—a lexical analyzer generator*.

[8] Kiselyov Oleg. 2024. *MetaOCaml: Ten Years Later - System Description*.

[9] Godefroid Patrice and Nagappan Nachiappan. 2008. *Concurrency at Microsoft – An Exploratory Survey*.

[10] François Pottier and Yann Régis-Gianas. 2024. *Menhir Reference Manual*. INRIA.

[11] KC Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman, and Anil Madhavapeddy. 2020. Retrofitting parallelism onto OCaml. *Proceedings of the ACM on Programming Languages* 4, ICFP (2020), 1–30.

[12] Joshua B. Smith. 2007. Camlp4. *Practical OCaml*, 411–429.

[13] C. Wetherell and A. Shannon. 1981. LR—Automatic Parser Generator and LR(1) Parser. *IEEE Transactions on Software Engineering* 3 (1981), 274–278.

[14] David A. Workman and John B. Higdon. 1978. *The design of a parser generator.* Association for Computing Machinery (ACM), New York, NY, USA.

[15] Hongbo Zhang. 2013. *Fan: compile-time metaprogramming for OCaml.*