Pura: A Statically-Typed Functional Language for Reliable UI Development

Architectural Design and Formal Implementation of a Hindley-Milner Compiler

Special Project on Information Media I

202513228 Atharva Timsina
Advisor: Prof. Hisashi Nakai

January 2026

Source code available at: `https://github.com/Axarva/pura-compiler`

Abstract

This report presents the design and implementation of Pura, a statically-typed functional programming language designed to demonstrate how first-principle implementation of strict immutability and explicit side-effect tracking can effectively eliminate architectural unpredictability in web development. The compiler, implemented in Haskell, uses a Hindley-Milner type inference engine to ensure type safety without requiring exhaustive manual annotations. This document details the language specification, the multi-pass compiler architecture, and the formal implementation of the inference system, concluding with a demonstration of the language's capabilities through a functional counter application and a presentation framework.

Contents

1 Introduction

Contemporary web development is predominantly characterized by imperative paradigms. While JavaScript and its associated frameworks are powerful, they suffer from a fundamental lack of architectural predictability. The prevalence of shared mutable state frequently results in inconsistent application behavior where side effects are non-transparent.

While established functional languages like **Elm** address these issues and declarative libraries like **React** improve UI composition, this project was driven by a deeper academic ambition: to understand and implement the internal mechanics of compiler construction. The primary motivation was to understand how raw character streams are transformed into a verified executable logical structure. **Pura** serves as a research platform to explore the balance between strict functional safety and the requirements of modern UI behaviors. Furthermore, unlike React, which allows arbitrary side effects within components, Pura aims to enforce a purity contract, ensuring that no side effect occurs without explicit type-level permission.

1.1 Design Philosophy

Pura is engineered based on three core technical pillars:

- **Purity and Immutability:** Data structures are persistent and immutable. Once a value is bound to an identifier, it remains constant, eliminating synchronization bugs.
- **Explicit Effect Tracking:** Interactions with the external environment (e.g I/O) are restricted operations. They must be declared via a `REQUIRES` clause.
- **Mathematical Rigor via HM Inference:** Pura utilizes a Hindley-Milner type system [1, 2] to provide a formal proof of type safety without requiring exhaustive manual annotations.

1.2 Project Scope

Initially planned as a domain-specific language for financial modeling, the project scope was refined to target web-based User Interface (UI) development. This change allowed for a direct evaluation of *The Elm Architecture* (TEA) [6], a pattern prevalent in functional user-interface design, in addition to compiler development. This transition also eliminated the overhead of studying financial concepts before beginning the project. By switching to UI development and targeting the web, the project was able to sidestep aforementioned issues and also validate that the language's architecture is capable of supporting practical, interactive web applications.

2 Language Specification

A language specification provides the formal definition of valid programs. It dictates both the syntax—the specific sequence of characters that form valid statements—and the structural hierarchy—the logical organization of components. The specification ensures that the lexer and parser can transform arbitrary text into a verifiable logical structure. Sections 2.1 and 2.2 discuss Pura's grammar and the conceptual abstract syntax tree of any Pura program.

2.1 Formal Grammar (Backus-Naur Form)

Listing 1 defines the syntax of the Pura language.

```
<program> ::= <top_level_declaration>*
<top_level_declaration> ::= <type_declaration> | <function_definition>


<type_declaration> ::= <identifier> ":" <type>

<function_definition> ::= "let" <identifier> "=" <parameter>* <body> <
    requires_clause>?

<parameter> ::= <identifier> "=>"

<body> ::= <block> | <expr>

<requires_clause> ::= "REQUIRES" <effect> ( "," <effect> )*

<type> ::= <basic_type> ( "->" <type> )?

<basic_type> ::= "Int" | "String" | "Bool" | "Unit" | "Msg"| <list_type>
               | <html_type> | <attr_type> | "(" <type> ")"

<list_type>      ::= "List" <basic_type>
<html_type>      ::= "Html" <basic_type>
<attr_type>      ::= "Attribute" <basic_type>

<expr> ::= <or_expr>
<or_expr> ::= <and_expr> ( "||" <and_expr> )*
<and_expr> ::= <comparison_expr> ( "&&" <comparison_expr> )*
<comparison_expr> ::= <additive_expr> ( <comp_op> <additive_expr> )*
<comp_op> ::= "==" | "!=" | "<" | ">" | "<=" | ">="
<additive_expr> ::= <concat_expr> ( ( "+" | "-" ) <concat_expr> )*
<concat_expr> ::= <multiplicative_expr> ( "++" <multiplicative_expr> )*
<multiplicative_expr> ::= <unary_expr> ( ( "*" | "/" ) <unary_expr> )*
<unary_expr> ::= "!" <unary_expr> | <application>
<application> ::= <atom>+

<atom> ::= <literal> | <variable> | <if_expr> | <let_expr> | <block>
         | <do_block> | "(" <expr> ")" | "(" <bin_op> ")"


<if_expr>  ::= "if" <expr> "then" <expr> "else" <expr>
<let_expr> ::= "let" <identifier> "=" <expr> "in" <expr>
<block>    ::= "{" ( <expr> ";" )* "}"
<do_block> ::= "do" "{" <expr>* "}"
<literal> ::= <int_literal> | <string_literal> | <bool_literal> | <list_literal>
    | <unit_literal>

<unit_literal>   ::= "()"
<list_literal>   ::= "[" ( <expr> ("," <expr>)* )? "]"
<bool_literal>   ::= "True" | "False"
<bin_op>         ::= "+" | "-" | "*" | "/" | "==" | "!=" | "<" | ">" | "<=" |
    ">=" | "&&" | "||"

<effect> ::= "ConsoleWrite" | "FileIO" | "Network" | "BrowserPrompt"
```

Listing 1: Complete Pura Backus-Naur Form (BNF) Grammar

2.2 High-Level Program Structure

A Pura program is represented internally as a hierarchical tree structure. The program conceptually flows from top-level declarations (Type and Function) down to atomic literals and variables.

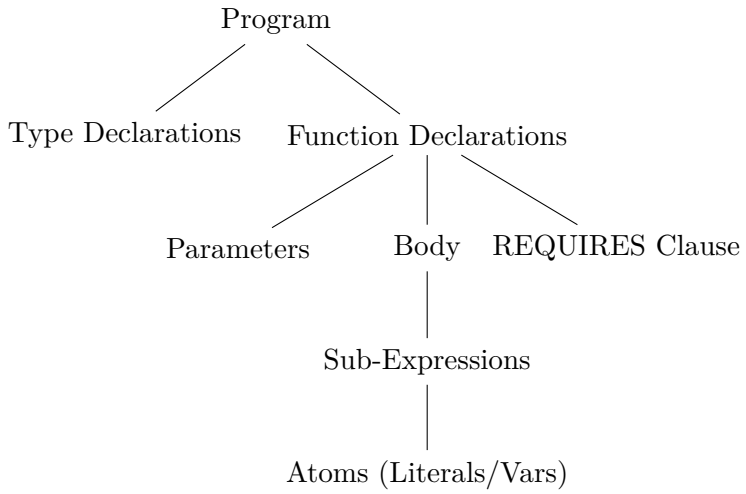

Figure 1: Conceptual Abstract Syntax Tree (AST) Hierarchy

The compiler processes these declarations sequentially. Each function is parsed into a specific `Function` record in the Haskell source. Although drawing out the entire tree for a large program is impractical, the recursive nature of the BNF ensures that every program can be reduced to the fundamental structure presented in Figure 1. Notable is the lack of mutable variable declarations, which is common in functional programming languages, where functions are treated as first-class citizens.

3 Compiler Architecture

The Pura compiler follows a standard multi-pass architecture. The source code undergoes a series of transformations, converting a raw string of characters into structured data (Abstract Syntax Tree), which is then analyzed for semantic validity before code generation. This section details the frontend of the compiler: the Lexer and the Parser.

3.1 Lexical Analysis (Lexer.hs)

Lexical analysis, or tokenization, is the first phase of compilation. Its primary role is to group the input character stream into meaningful sequences called *tokens* (e.g., keywords, identifiers, literals), discarding whitespace and comments.

The Pura compiler utilizes a handwritten lexer rather than a generated one to maintain full control over metadata. Each `Token` is defined not just by its type, but by its coordinate:

```
data Token = Token { tokType :: TokenType, tokLine :: Int, tokCol :: Int }
```

The lexer handles complex tokens such as curried arrows (`=>`), string literals with escape sequences, and multi-line comments. By maintaining position data, the compiler provides high-fidelity error messages that pinpoint the exact location of a syntax or type error.

3.2 Parsing Strategy (Parser.hs)

Parsing is the process of analyzing a given stream of tokens to determine its grammatical structure with respect to the given formal grammar. The result is an Abstract Syntax Tree (AST), a tree

representation of the syntactic structure of the source code.

The Pura compiler's parser is built using Haskell's Megaparsec library, following a recursive descent (with backtracking) strategy [4]. To handle binary operator precedence without ambiguity, Pura implements a "precedence cascade," where functions call each other in order of increasing priority:

1. `parseOr` (Priority 1: `||`)
2. `parseAnd` (Priority 2: `&&`)
3. `parseComparison` (Priority 3: `==`, `!=`, `<`, `>`)
4. `parseAdditive` (Priority 4: `+`, `-`)
5. `parseConcat` (Priority 5: `++`)
6. `parseMultiplicative` (Priority 6: `*`, `/`)
7. `parseUnary` (Priority 7: `!`)
8. `parseApplication` (Priority 8: Function Application)
9. `parseAtom` (Priority 9: Literals, Parens, Blocks)

A priority of 1 refers to the lowest priority, and a priority of 9 is the highest priority in the Pura language currently.

4 Technical Implementation of Type Inference

Type inference is the logic that allows the compiler to deduce the types of expressions without explicit annotations. The Pura compiler implements the Hindley-Milner (HM) type system [1, 2], which provides a formal guarantee of type safety while supporting parametric polymorphism.

4.1 Core Data Structures

The implementation relies on specific data structures defined in `Types.hs` to represent the type universe.

4.1.1 Types and Schemes

Pura distinguishes between monomorphic types (`Type`) and polymorphic schemes (`Scheme`). The implementation of these structures in the Pura compiler are detailed in Listing 2.

```haskell
-- Represents concrete types
data Type
  = TInt | TBool | TString       -- Primitives
  | TArr Type Type                -- Functions (T1 -> T2)
  | TVar String                   -- Type Variables (e.g., "a")
  | THtml Type                    -- UI Nodes (Html Msg)
  ...

-- Represents a type with quantified variables (forall a. a -> a)
data Scheme = Forall [String] Type
```

Listing 2: Data structure definitions for type checking

4.1.2 The Environment

The `TypeEnv` corresponds to the typing environment Γ in formal literature. It maps variable names to their type schemes, and is implemented as follows:

```
type TypeEnv = Map.Map String Scheme
```

4.2 The Inference Monad

To manage the stateful generation of fresh type variables and propagate type errors, the compiler employs a custom monad transformer stack defined in `Inference.hs` as shown in Listing 3. This architecture follows established functional compiler design patterns [3], utilizing a transformer stack to isolate state management from the core logic:

```
type Infer a = ExceptT String (StateT InferState Identity) a
```

Listing 3: Definition of the Monad Transformer Stack for the Infer monad

The inclusion of `Identity` as the base monad is sufficient for a pure compiler, as it allows the stack to be evaluated into a pure value. By explicitly defining the stack (as opposed to `ExceptT String (State InferState a)`), the implementation ensures that the fresh-variable counter persists through error boundaries while keeping the engine extensible (i.e, the Identity monad can be swapped out for another monad such as IO for debugging convenience).

4.3 Substitutions and the Substitutable Typeclass

A **substitution** (S) is a mapping from type variables to types. Mathematically, it can be thought of as a transformation function S such that $S(T)$ represents the result of applying the mapping corresponding to S throughout the structure of type T.

The core solver (`unify`, discussed in section 4.4.3) relies on the `Substitutable` typeclass to handle the recursive application of substitutions to types across the environment, adapting the implementation strategy described by Diehl [3], detailed in Listing 4.

```
type Subst = Map.Map String Type

class Substitutable a where
  apply :: Subst -> a -> a   -- Apply substitution S to structure a
  ftv   :: a -> Set.Set String -- Return Free Type Variables in a
```

Listing 4: Substitutable typeclass implementation in the Pura compiler

All major data structures related to Pura's type environment (`Type`, `Scheme`, `TypeEnv`) implement this class.

- `apply`: Propagates a substitution down the tree. For example, applying $\{a \mapsto \text{Int}\}$ to the list type $[a]$ results in [Int].
- `ftv`: Returns the set of unbound variables. This is crucial for determining which variables can be generalized (made polymorphic).

4.4 Auxiliary Inference Algorithms

The inference engine is built upon three fundamental operations: generalization, instantiation and unification.

4.4.1 Generalization

Generalization converts a `Type` into a `Scheme`. It identifies variables that are free in the type but not constrained by the environment Γ.

$$\text{gen}(\Gamma, \tau) = \forall \vec{\alpha}.\tau \quad \text{where } \vec{\alpha} = \text{ftv}(\tau) \setminus \text{ftv}(\Gamma)$$

In the Pura compiler, generalization is implemented as the `generalize` function with the following type signature:

```
generalize :: TypeEnv -> Type -> Scheme
```

4.4.2 Instantiation

Instantiation converts a `Scheme` back into a `Type` by replacing quantified variables with fresh ones. This function depends on the `Infer` monad to generate unique names.

$$\text{inst}(\forall \vec{\alpha}.\tau) = \tau[\vec{\alpha} := \vec{\beta}] \quad \text{where } \vec{\beta} \text{ are fresh}$$

In the Pura compiler, instantiation is implemented as the `instantiate` function with the following type signature:

```
instantiate :: Scheme -> Infer Type
```

4.4.3 Unification

Unification serves as the mechanism for constraint resolution. Formally, the algorithm seeks a substitution S such that two provided types become syntactically identical ($S(\tau_1) \equiv S(\tau_2)$). In the compiler, this logic is encapsulated as a pure function as shown in Listing 5:

```
unify :: Type -> Type -> Either String Subst
unify (TVar a) t = bind a t              -- Bind a to t if unifying type
                                         -- variable a and static type t
unify (TArr t1 t2) (TArr t3 t4) = ... -- Recursive unification
...
unify t1 t2 = Left "Type mismatch"     -- Throw error if all checks fail
```

Listing 5: Unification function implementation in the compiler

4.5 Formal Typing Judgments and Implementation

The main driver of the inference process is the `inferExpr` function. It traverses the AST, generating constraints and solving them via `unify`.

Below are the three primary judgments of the HM system alongside their Haskell implementation in the Pura compiler.

4.5.1 [Var] Variable Access

$$\frac{x : \forall \vec{\alpha}.\tau \in \Gamma \quad \vec{\beta} \text{ are fresh}}{\Gamma \vdash x : \tau[\vec{\alpha} := \vec{\beta}]}$$

The compiler looks up the variable in the environment and instantiates it, as shown in Listing 6.

```
Var name -> case Map.lookup name env of
  Just scheme -> do
    t <- instantiate scheme  -- Replace quantified vars with fresh ones
    return (Map.empty, t)
```

Listing 6: Var Judgement Implementation in the Compiler

4.5.2 [App] Function Application

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2}$$

We infer the function and argument, then `unify` the function's input type with the argument's type. This is visualized in Figure 2, and implemented as detailed in Listing 7.

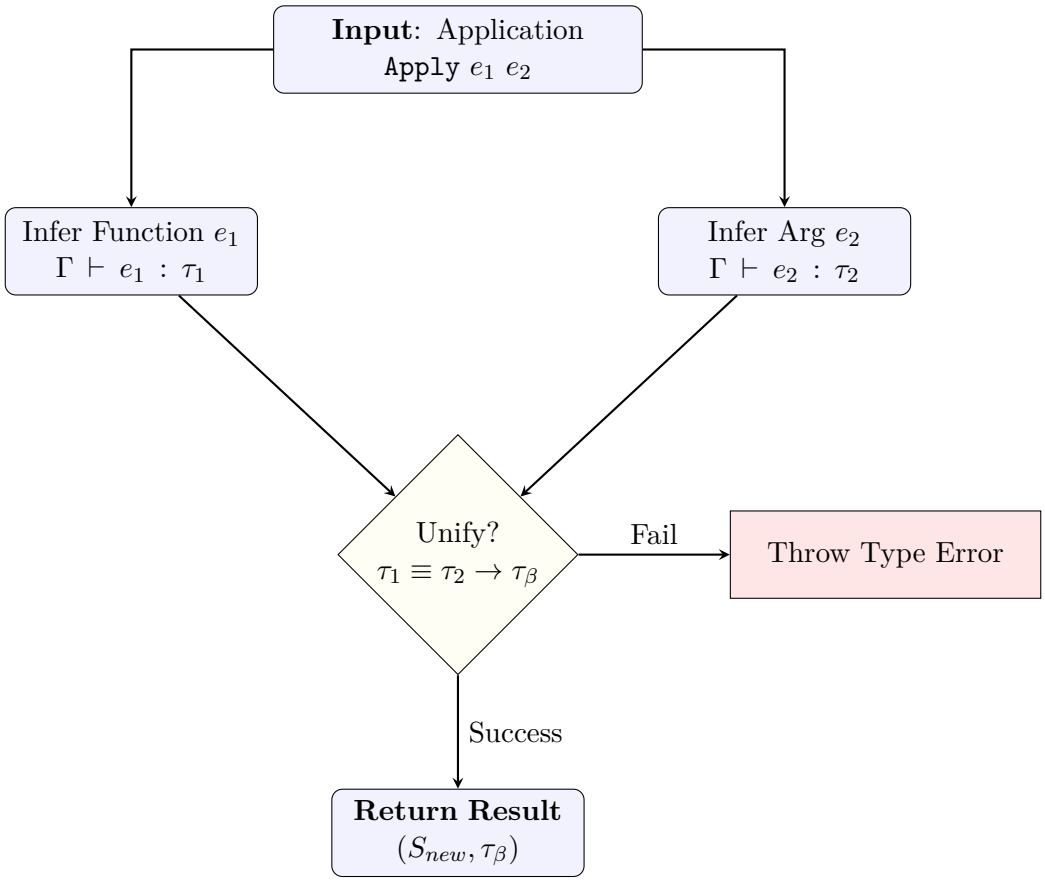

Figure 2: Visualizing the Recursive Step: Type Inference for Function Application

```haskell
Apply e1 e2 -> do
  tv <- freshTVar
  (s1, t1) <- inferExpr globalEnv env e1
  (s2, t2) <- inferExpr globalEnv (apply s1 env) e2
  -- Unify t1 with (t2 -> tv)
  s3 <- liftEither $ unify (apply s2 t1) (TArr t2 tv)
  return (s3 `compose` s2 `compose` s1, apply s3 tv)
```

Listing 7: App Judgement Implementation in the Compiler

4.5.3 [Let] Polymorphism Rule

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \text{gen}(\Gamma, \tau_1) \vdash e_2 : \tau_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2}$$

Crucially, the type of the value is **generalized** before being added to the environment, handled through the code provided in Listing 8.

```
Let name val body -> do
  (s1, t1) <- inferExpr globalEnv env val
  let env' = apply s1 env
  let scheme = generalize env' t1  -- Generalize to allow polymorphism
  let env'' = Map.insert name scheme env'
  (s2, t2) <- inferExpr globalEnv env'' body
  return (s2 `compose` s1, t2)
```

Listing 8: Let Judgement Implementation in the Compiler

Through the described data structures for type information, the inference monad, and the auxiliary inference algorithms assisting the core unification algorithm, any Pura program can be rigorously checked for type safety.

5 Effect and Code Generation

Following the successful verification of types, the compiler proceeds to the synthesis phase. This stage involves two critical steps: a final semantic pass to verify the safety of side effects, and the generation of executable JavaScript code that interfaces with the runtime environment.

5.1 Semantic Verification of Side Effects

The effect system is implemented as a dedicated semantic analysis pass in `Permissions.hs`. Unlike the type checker, which verifies data consistency, this pass verifies *behavioral* safety.

The compiler performs a traversal of the AST to identify all atomic operations classified as side effects (e.g., `ConsoleWrite`, `Network`). It then enforces an invariant: any function body containing a side-effectful operation must explicitly declare that effect in its `REQUIRES` clause. If an undeclared effect is detected, the compiler halts with a static error before code generation begins. This design enforces the principle of least privilege, ensuring that a function cannot perform I/O or network requests invisibly—a common source of non-deterministic bugs in JavaScript development.

5.2 Compilation Target: JavaScript

The generator (`CodeGen.hs`) handles currying and uses Immediately Invoked Function Expressions (IIFEs) in the output JavaScript code to preserve Pura's expression-based semantics in JavaScript.

In addition, to handle two separate use cases for Pura programs (i.e execution as scripts in the command line, and TEA-based applications for the web), the code generator also includes a simple runtime in every output JavaScript file. This runtime detects if the functions `model`, `view` and `update` are defined in the input Pura file, and based on the result, provide an environment for the program to run on the web through an HTML file or to be executed by a runtime environment such as Node.js.

6 Implementation Constraints

While Pura successfully demonstrates a functional compiler architecture, its current implementation as an academic project entails certain engineering trade-offs. This section outlines the structural and performance limitations of the current system compared to production-grade languages like Elm or Haskell.

6.1 Runtime Efficiency and Re-rendering

While Pura adopts the declarative structure of TEA, it lacks the runtime optimizations found in established functional UI languages. As detailed by Czaplicki [7], efficient Functional Reactive Programming (FRP) systems rely on dependency graphs or memoization to ensure that only the necessary

components are recomputed when inputs change. In contrast, the current implementation of Pura performs a full re-render of the web page DOM on every model update. While this simplifies the compilation process to a direct JavaScript translation, it results in computational inefficiency compared to languages such as Elm.

6.2 Messaging Architecture

One critical implementation constraint is the handling of state updates, specifically in applications following The Elm Architecture. In a mature system like Elm, `Msg` is a user-defined union type providing modular isolation [6]. Due to complexities in implementing custom constructor type-checking, Pura currently utilizes a simple string-based system (`Html String`). Consequently, Pura lacks component-local message encapsulation. Since messages are strings passed to a global `update` function, every event is visible to the parent dispatcher. This may be changed in the future.

6.3 Effect System Limitations

While Pura enforces effect safety, the current implementation operates primarily as a static semantic verifier rather than a fully algebraic effect system. In research languages like Koka [8], effects are treated as algebraic operations that can be intercepted and handled by the runtime (e.g., swapping a `Network` effect for a mock implementation during tests). Pura currently lacks this handler capability, treating effects solely as static permissions to be checked during compilation. Consequently, Pura cannot yet support advanced control flow patterns like effect resumption or cooperative concurrency.

6.4 Syntactic and Structural Limitations

1. **Anonymous Lambdas**: Lambdas (e.g., `x => x + 1`) are restricted to the RHS of bindings to simplify the HM implementation. Listing 9 shows a comparison of what is desirable compared to what is currently possible in Pura.

```
-- Current Pura Code (Must bind locally)
let inc = x => x + 1 in map inc [1, 2, 3]

-- Hypothetical Future Pura Code (First-class lambdas)
map (x => x + 1) [1, 2, 3]

```

Listing 9: Comparison of Lambda Support

2. **Control Flow**: Pura lacks a `case` statement, necessitating nested `if-then-else` blocks.
3. **Standard Library**: The environment lacks high-level primitives and advanced data structures.

7 Demonstration

To satisfy the requirement of demonstrating the compiler's execution state and practical capability, this section presents the compilation artifacts of a verification program and the deployment of a full-scale application.

7.1 Compiler Execution Verification

To verify the correctness of the compilation pipeline, we utilized a standard counter application `counter.pura`. This program tests the essential features of the language: state management, event handling (via `update`), and DOM rendering (via `view`). The source code for `counter.pura` can be found in Listing 10.

```
initialModel : Int
let initialModel = 0

update : String -> Int -> Int
let update = msg => model => {
  if msg == "0" then {
   model + 1
  } else {
    if msg == "1" then {
      model - 1
    } else model
   }
}

view : Int -> Html Msg
let view = model => {
  div [] [
    h1 [] [text "Pura Counter"],
    p [] [text ("Count: " ++ (toString model))],
    button [htmlOnClick "0"] [text "+"],
    button [htmlOnClick "1"] [text "-"]
  ]
}
```

Listing 10: Source Code: examples/counter.pura

The log presented in Listing 11 demonstrates the compiler successfully processing the source file, performing type inference, and generating the target JavaScript.

```
$ examples git:(main) stack exec pura-compiler-exe counter.pura
Compiling: counter.pura
--- Parsing Successful ---
--- Type Checking Successful ---
--- Effect Checking Successful ---
--- Generating Code ---
----------------------
Successfully wrote generated code to counter.pura.js
$ examples git:(main)
```

Listing 11: Terminal Output during Compilation

The resulting JavaScript file was loaded into a web browser environment. Figure 3 illustrates the running application, where the state ‘model‘ is correctly updated and reflected in the DOM upon user interaction.

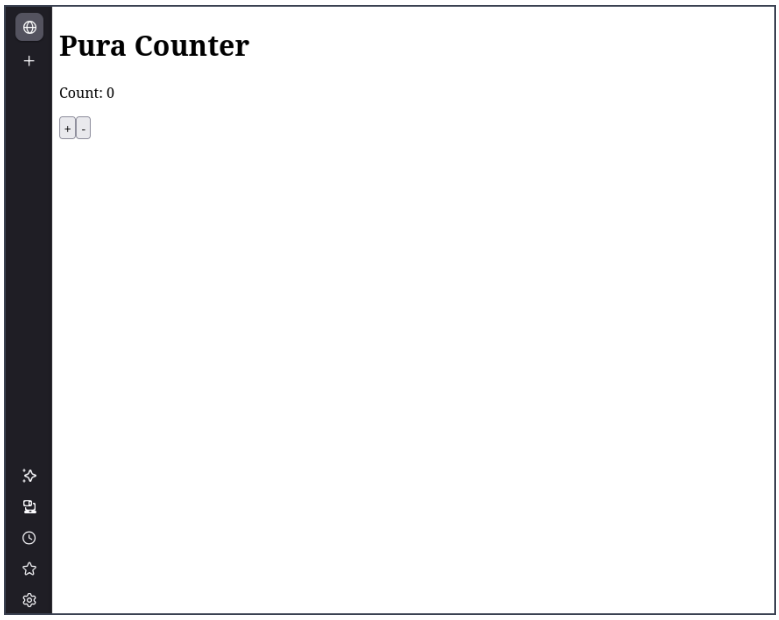

Figure 3: The compiled counter.pura running in a browser environment.

7.2 Practical Application: Presentation Framework

To evaluate Pura's capability in a complex, state-driven context, the final project presentation was developed entirely within the language itself. This application leverages the TEA architecture to manage navigation state and DOM updates, and has a simple TODO list application built into it as well.

The successful compilation and deployment of this project serves as a comprehensive test for the compiler, validating its handling of basic web inputs such as button clicks and static data structures, and also serves as a milestone in Pura's development. The application is accessible online at: `https://axarva.me/pura-compiler`. Figure 4 shows the front page displayed on a modern browser when the website is visited.

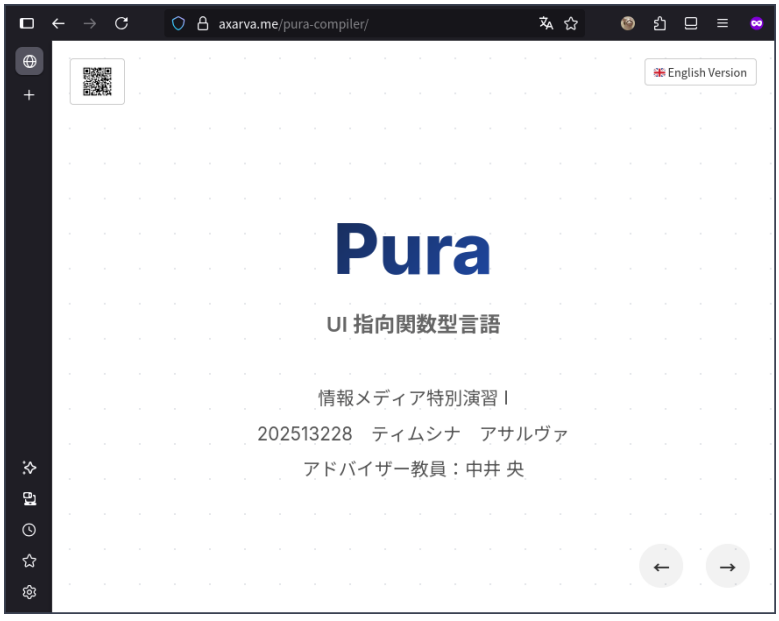

Figure 4: The project's final presentation running in Mozilla Firefox.

8 Conclusion and Future Work

Pura successfully implements a statically-typed functional language that enforces strict immutability and side-effect safety. The project realized its primary goal of constructing a verified compiler pipeline from first principles, validating that a custom-built compiler enforcing these constraints is capable of supporting complex, interactive web applications. Through the development of the verifying examples and the presentation framework, the compiler has proven its viability as a research platform.

Future work includes:

- **Effect System Expansion**: Transitioning to full algebraic effects with handlers to support user-defined control flow, or a better alternative.
- **Standard Library**: Adding basic UI and logic primitives expected in a usable programming language.
- **Wasm Backend**: Targeting WebAssembly, if feasible.
- **Language Expressivity**: Implementing first-class anonymous lambdas and encapsulated messages.
- **Pattern Matching**: Replacing nested conditionals with a full `case` system.

References

[1] R. Milner, "A theory of type polymorphism in programming," *Journal of Computer and System Sciences*, vol. 17, no. 3, pp. 348–375, Dec. 1978, doi: 10.1016/0022-0000(78)90014-4.

[2] L. Damas and R. Milner, "Principal type-schemes for functional programs," in *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Language*, pp. 207–212, Jan. 1982, doi: 10.1145/582153.582176.

[3] S. Diehl, "Write You a Haskell," *Stephen Diehl's Personal Website*, 2014–Present. [Online]. Available: `http://dev.stephendiehl.com/fun/`

[4] A. W. Appel, *Modern Compiler implementation in ML*, Cambridge University Press, 1997, pp. 38–86. doi: 10.1017/cbo9780511811449.

[5] A. Gill and K. Ross, "The mtl package: Monad classes for transformers," *Hackage*, 2001–Present. [Online]. Available: `https://hackage.haskell.org/package/mtl`

[6] E. Czaplicki, "The Elm Architecture," *Elm Language Guide*, 2024. [Online]. Available: `https://guide.elm-lang.org/architecture/`

[7] E. Czaplicki, "Elm: Concurrent FRP for Functional GUIs," Senior thesis, Harvard University, Cambridge, MA, 2012.

[8] D. Leijen, "Koka: Programming with Row Polymorphic Effect Types," *Electronic Proceedings in Theoretical Computer Science*, vol. 153, pp. 100–126, Jun. 2014, doi: 10.4204/eptcs.153.8.