

Leveraging Metaprogramming in Functional Programming: An Inventory Management System Case Study

May 1, 2025

Contents

1	Introduction	2
2	Problem Statement and Project Goals	2
3	Functional Programming and Metaprogramming	3
3.1	Functional Programming Principles	3
3.2	Introduction to Metaprogramming	6
3.3	Metaprogramming in Functional Languages	7
4	Technical Deep Dive: Metaprogramming in Haskell	7
4.1	Compile-Time Metaprogramming with Template Haskell	7
4.1.1	Introduction to Template Haskell	8
4.1.2	Compile-Time Database Structure Generation	8
4.2	Runtime Metaprogramming	12
4.2.1	Dynamic CRUD Operations	12
4.2.2	User-Definable Function Pipelines	15
4.2.3	Server Module ('app/Server.hs' - Conceptual)	16
5	How to Build and Run	17
5.1	Prerequisites	17
5.2	Building the Project	18
5.3	Running the Server	18
5.4	Running the Frontend	19
6	Demonstrations	19
6.1	Compile-Time Metaprogramming: Template Haskell Code Generation	19
6.2	Runtime Metaprogramming: Dynamic CRUD Dispatch	20
6.3	Runtime Function Composition: Pipeline Construction and Execution	20
7	Team and Contributions	22
8	Challenges and Learnings	23
8.1	Challenges Encountered	24
8.2	Learnings Gained	25
9	Future Work	26
10	Conclusion	27

1 Introduction

The programming-language ecosystem continually advances, driving demand for abstractions that yield expressive, maintainable, and efficient code. **Functional programming**—emphasizing immutability, pure functions, and composition—provides a solid foundation for complex system development. **Metaprogramming** augments this paradigm by enabling code to introspect, generate, or transform other code, thereby automating repetitive patterns, minimizing boilerplate, and maximizing adaptability.

This project explores the convergence of these methodologies via an **Inventory Management System** implemented in **Haskell**. As a statically typed functional language with **Template Haskell**, Haskell offers a compelling platform for both **compile-time** and **runtime** metaprogramming. We demonstrate how to leverage **Template Haskell** for automated code generation during compilation and how to implement dynamic behavior at runtime—facilitating on-the-fly adaptation to user inputs without recompilation.

Inventory management exemplifies standard **CRUD** operations—adding, viewing, updating, and deleting items—while serving as a realistic use case for our metaprogramming techniques. At **compile-time**, **Template Haskell** generates repetitive data-structure and boilerplate code from concise specifications. At **runtime**, a user-definable **function pipeline** enables flexible request handling, leveraging **higher-order functions** for pipeline composition, dispatch logic, and dynamic extension.

Our architecture underscores the pivotal role of **higher-order functions** in metaprogramming: treating functions as data allows code to assemble, modify, and execute function pipelines dynamically. This document details the theoretical underpinnings of **functional programming** and **metaprogramming**, outlines the system architecture, presents Haskell code examples for both compile-time and runtime techniques, analyzes the **function-pipeline** mechanism, and reflects on development challenges, lessons learned, and future extension opportunities.

2 Problem Statement and Project Goals

This project directly addresses the following problem statement, which served as the guiding principle for our development efforts:

Leveraging Meta-Programming in Functional Programming

TYPE: (implementation)

This project aims to explore the intersection of meta-programming and functional programming.

- Analyze specific functional programming features that facilitate meta-programming, such as first-class functions, closures, and function composition. Examine how these features can be used for meta-programming purposes.
- Develop a suite of programs in a functional programming language that demonstrate both compile-time and runtime meta-programming techniques.

Based on this problem statement, the primary goals of our project were clearly defined to ensure a focused and impactful implementation:

- **Deep Understanding of Metaprogramming in FP:**
 - **Theoretical Foundation:** Research core **metaprogramming**, **code-as-data**, and **evaluation stages**.
 - **Haskell Features:** Examine **type system**, **lazy evaluation**, and **purity**.

- **Analysis of Functional Features for Metaprogramming:**
 - **First-Class Functions:** Show passing and composing **functions**.
 - **Function Composition:** Use `.` and `$` to streamline pipelines.
- **Implementation of Compile-Time Metaprogramming:**
 - **Template Haskell Basics:** Use `Q` monad and `Dec/Exp` AST nodes.
 - **Automated Code Generation:** Write TH splices (`$(...)`), derive **CRUD** interfaces.
- **Implementation of Runtime Metaprogramming:**
 - **Dynamic Dispatch:** Define `InventoryRequest` and `processRequest`.
 - **Function Pipeline:** Build registry of **user functions** and compose at **execution time**.
- **Building a Demonstrative Application:**
 - **Backend Server:** Implement **HTTP API** and concurrency-safe **state**.
 - **Frontend Interface:** Provide minimal **UI** for CRUD and pipelines.
- **Differentiating Compile-Time and Runtime Metaprogramming:**
 - **Trade-offs:** Compare **performance**, **safety**, and **flexibility**.
 - **Use Cases:** Recommend when to use **compile-time** vs. **runtime**.
- **Reflection and Documentation:**
 - **Challenges:** Note issues with **AST manipulation** and **concurrency**.
 - **Future Directions:** Outline extensions like **type-level metaprogramming**.

By diligently pursuing these goals through the development of the Inventory Management System, we aimed to not only fulfill the project requirements but also to gain valuable hands-on experience with advanced programming techniques and contribute to our understanding of the power and flexibility that metaprogramming, when combined with functional programming principles, can offer.

3 Functional Programming and Metaprogramming

3.1 Functional Programming Principles

Functional programming (**FP**) redefines computation through the lens of mathematical function evaluation rather than mutable state manipulation. This paradigm introduces several **core principles** essential to our project’s metaprogramming foundation:

- **Pure Functions:**
 - Always return the same output for identical inputs.
 - Produce no **side effects**—no modification of external state or I/O within the function.
 - Enhance testability, reasoning, and enable predictable behavior, critical for **code analysis** in metaprogramming.

```

al run PureFunctionsCurrying
Available functions: add, multiply, subtract, divide
Enter functions to apply (space-separated):
add multiply subtract divide
Enter first input number:
12
Enter second input number:
5
Results: [17,60,7,2]
Continue? (y/n)
n
Goodbye.

```

Figure 1: Illustration of Pure Functions and Currying in Haskell

- **First-Class Functions:**

- Functions can be assigned to variables, passed, returned, or stored in structures.
- Underpins **dynamic function composition**, user-defined pipelines, and metaprogram-driven function manipulation.

```

tanish@TanishDELL:/mnt/c/Users/offic/OneDrive/Desktop/Work/ProgrammingShowcase$ cabal run HigherOrderFunctions
Just 7
Just [2,3,4]
tanish@TanishDELL:/mnt/c/Users/offic/OneDrive/Desktop/Work/ProgrammingShowcase$ cabal run FirstClassFunctions
Just 15
concatDynamic failed

```

Figure 2: Higher-Order and First-Class Function Concepts

- **Higher-Order Functions (HOFs):**

- Accept or return other functions.
- Enable abstraction patterns such as `map`, `filter`, and `fold`.
- Serve as the foundation of our **pipeline executor**, processing lists of functions dynamically.

- **Function Composition:**

- Build new behavior via chaining: if f and g , then $h(x) = f(g(x))$.
- Supports modular design and code reuse—a key mechanism in **automated code synthesis**.

```
al run FunctionComposition

Choose an option:
1. Square, then increment and then double the number
2. Apply uppercase+reverse to a string
3. Compose functions dynamically
Type 'exit' to quit.
1
> Enter a number:
12
290
```

(a) Function Composition - Example 1

```
Choose an option:
1. Square, then increment and then double the number
2. Apply uppercase+reverse to a string
3. Compose functions dynamically
Type 'exit' to quit.
2
> Enter a string:
Hello World!!!
!!!DLROW OLLEH
```

(b) Function Composition - Example 2

```
Choose an option:
1. Square, then increment and then double the number
2. Apply uppercase+reverse to a string
3. Compose functions dynamically
Type 'exit' to quit.
3
> Available functions: double, square, increment
Enter names to compose (space-separated):
double square
Enter a number:
10
Result: 400

Choose an option:
1. Square, then increment and then double the number
2. Apply uppercase+reverse to a string
3. Compose functions dynamically
Type 'exit' to quit.
exit
```

(c) Function Composition - Example 3

Figure 3: Visual comparison of different stages in function composition

• Declarative Style:

- Emphasizes **what** to compute over **how** to compute it.
- Facilitates cleaner abstraction and **more analyzable code**—aiding metaprogramming tooling.

```
run MonadLazyEvaluation

Select an option:
1. Safe Chain Division
2. Lazy Infinite List (take N)
3. Apply Dynamic Integer Operations
4. Apply Dynamic String Operations
5. Lazy Conditional Evaluation
6. Exit
1
Choice: Enter initial number:
12
Enter divisors separated by space:
2 3 2
Result: 1
```

(a) Part 1

```
Select an option:
1. Safe Chain Division
2. Lazy Infinite List (take N)
3. Apply Dynamic Integer Operations
4. Apply Dynamic String Operations
5. Lazy Conditional Evaluation
6. Exit
2
Choice: How many elements to take?
10
[1,2,3,4,5,6,7,8,9,10]
```

(b) Part 2

```
Select an option:
1. Safe Chain Division
2. Lazy Infinite List (take N)
3. Apply Dynamic Integer Operations
4. Apply Dynamic String Operations
5. Lazy Conditional Evaluation
6. Exit
Choice: Invalid choice. Please select again.

Select an option:
1. Safe Chain Division
2. Lazy Infinite List (take N)
3. Apply Dynamic Integer Operations
4. Apply Dynamic String Operations
5. Lazy Conditional Evaluation
6. Exit
4
Choice: Available string operations: reverse, uppercase, append-exclamation
Enter operations separated by space (applied left to right):
reverse uppercase append-exclamation
Enter the string to operate on:
Hello World!!!
Result: !!!DLROW OLLEH!
```

(c) Part 3

Figure 4: Monads and Lazy Evaluation (Parts 1–3)

Haskell enforces these principles through a **strong static type system**, **monadic effect handling**, and native support for **HOFs and immutability**, making it ideal for metaprogramming tasks at both compile-time and runtime.

3.2 Introduction to Metaprogramming

Metaprogramming enables code to manipulate or generate other code—essentially treating **programs as data**. Its major applications include:

- **Reducing Boilerplate:**
 - Automates repetitive code like typeclass instances or serialization logic.
- **Embedding Domain-Specific Languages (DSLs):**
 - Creates concise, domain-aligned abstractions inside a host language.
- **Code Analysis and Transformation:**
 - Enables static checks, refactoring, or enforcing conventions via **AST manipulation**.
- **Optimization:**
 - Moves computations to compile time, reducing runtime overhead.
- **Reflection and Introspection:**
 - Inspects code at runtime to support adaptive behavior.

Two key forms of metaprogramming:

- **Compile-Time Metaprogramming:**
 - Operates during compilation on ASTs (e.g., using **Template Haskell**).
 - Benefits:
 - * **Performance gains** from precomputed logic.
 - * **Code reuse** via automated generation.
 - * **Static error checking** of generated code.
- **Runtime Metaprogramming:**
 - Executes metaprograms during program execution (e.g., **eval**, dynamic function loading).
 - Benefits:
 - * **Runtime flexibility**, adapting to inputs or configs.
 - * **Extensibility** without recompilation.
 - * **Dynamic behavior** driven by user-defined constructs.

Despite its power, metaprogramming increases **complexity**, reduces **readability**, and complicates **debugging**. Judicious use is required.

3.3 Metaprogramming in Functional Languages

Functional languages offer distinct metaprogramming advantages due to their abstraction mechanisms and well-defined semantics:

- **Code as Data (Homoiconicity):**
 - Languages like Lisp allow seamless AST manipulation.
 - Haskell achieves similar goals through **Template Haskell**'s AST APIs.
- **HOFs and Function Composition:**
 - Facilitate logic abstraction and transformation.
 - Enable compositional metaprograms that operate over dynamic logic flows (e.g., function pipelines).
- **Strong Type Systems:**
 - Type-driven code generation (e.g., typeclass derivation) ensures **type-safe metaprogramming**.
 - Errors in metaprograms or generated code are caught **statically**.
- **Purity and Immutability:**
 - Guarantee **predictability** and easier reasoning during code transformation.
- **Well-Defined Semantics:**
 - Simplifies static analysis and automated reasoning about code behavior.

This project leverages Haskell's capabilities to demonstrate both **compile-time** (via Template Haskell) and **runtime** metaprogramming (via dynamic function composition). The integration of both paradigms in a single system exemplifies the expressive power and practicality of metaprogramming in a functional language.

4 Technical Deep Dive: Metaprogramming in Haskell

This section delves into the specific techniques and implementations used in our project to demonstrate compile-time and runtime metaprogramming within the Haskell language. We focus on Template Haskell for compile-time code generation and a combination of data-driven dispatch and higher-order functions for runtime code manipulation and execution.

4.1 Compile-Time Metaprogramming with Template Haskell

Compile-time metaprogramming in our project is primarily implemented using Template Haskell (TH). TH is a powerful extension to GHC that allows developers to write Haskell code that generates or modifies other Haskell code during the compilation process. This is particularly useful for automating repetitive coding tasks and performing computations or transformations based on static information available at compile time.

4.1.1 Introduction to Template Haskell

Template Haskell operates in a distinct phase of the compilation pipeline. When the compiler encounters a Template Haskell splice, it pauses the normal compilation of the surrounding code, executes the TH computation defined within the splice, and then resumes compilation with the result of the TH computation (which is a piece of Haskell code represented as an Abstract Syntax Tree - AST) inserted into the original source code.

Key concepts in Template Haskell include:

- **Quotes and Splices:**

- **Quotes:** TH provides syntax for "quoting" Haskell code, lifting it from its concrete syntax into its AST representation. This is done using square brackets: `'[...]'` for expressions ('Q Exp'), `'[p...]'` for patterns ('Q Pat'), `'[t...]'` for types ('Q Type'), and `'[d...]'` for declarations ('Q [Dec]'). For example, `'[1 + 1]'` represents the AST for the expression `1 + 1`.
- **Splices:** Splices are the inverse of quotes; they execute a computation in the 'Q' monad and insert the resulting AST into the code. Splices are denoted by a dollar sign followed by parentheses: `'(expr)'`

- **Q Monad:** Template Haskell computations run within the 'Q' monad. This monad provides access to the compiler's internal state and environment. It allows TH code to perform actions like:

- Looking up the names of types, functions, or variables (`'lookupTypeName'`, `'lookupValueName'`).
- Reifying declarations: inspecting the structure and type information of existing declarations (`'reify'`). This is crucial for writing generic TH functions that operate on different data types.
- Emitting warnings or errors during compilation (`'reportWarning'`, `'reportError'`).
- Generating fresh names to avoid naming conflicts (`'newName'`).

- **Abstract Syntax Tree (AST):** Template Haskell manipulates Haskell code as ASTs. The `'Language.Haskell.Syntax'` module defines the datatypes that represent the different syntactic constructs of Haskell (expressions `'Exp'`, patterns `'Pat'`, types `'Type'`, declarations `'Dec'`). Writing TH code often involves constructing or pattern matching on these AST datatypes.

Template Haskell is a powerful tool for reducing boilerplate, implementing domain-specific languages (DSLs), and performing static analysis or code transformations based on type information or other compile-time knowledge. It allows developers to write code that is more abstract and generates repetitive parts automatically, leading to more concise and potentially more maintainable codebases.

4.1.2 Compile-Time Database Structure Generation

In our Inventory Management System, a primary application of compile-time metaprogramming is the automated generation of code related to our database structure. Instead of manually writing the Haskell data types that mirror the structure of our inventory items and the necessary instances for interacting with data formats (like JSON for the frontend API) or a database, we use Template Haskell to define and generate this code based on a concise, high-level specification.

The core idea is to define the structure of an inventory item (its fields and their types) once in a simple format. A Template Haskell function then reads this definition at compile time and generates the full Haskell ‘data’ type declaration, including deriving instances for relevant type classes like ‘Show’, ‘Eq’, ‘Generic’ (for use with generic programming libraries), and ‘FromJSON’/‘ToJSON’ (for serialization/deserialization to/from JSON).

Consider a simplified example where we want to define an ‘InventoryItem’ with fields such as ‘itemId’ (an integer identifier), ‘name’ (a string), ‘quantity’ (an integer representing stock count), and ‘price’ (a double).

First, we define the structure in a simple list of pairs, where each pair is ‘(fieldName, field-TypeString)’. This definition is placed in a module that will be processed by Template Haskell, perhaps in a dedicated ‘templates/’ directory.

```
-- templates/THGen.hs
module THGen where

import Language.Haskell.TH

-- Define the structure of the InventoryItem data type
inventoryFields :: [(String, String)]
inventoryFields =
  [ ("itemId", "Int")
  , ("name", "String")
  , ("quantity", "Int")
  , ("price", "Double")
  ]

-- This list serves as the input for our Template Haskell function.
-- It's a simple, declarative way to specify the desired fields.
```

Listing 1: Conceptual Base Data Structure Definition (templates/THGen.hs)

Next, we write the Template Haskell function that takes this list of fields and generates the Haskell ‘data’ declaration. This function operates in the ‘Q’ monad and constructs the AST for the data type.

```
{-# LANGUAGE TemplateHaskell #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE MultiParamClasses #-}
{-# LANGUAGE DeriveGeneric #-}

-- Continue in templates/THGen.hs
import GHC.Generics
import Data.Aeson -- Assuming Aeson is used for JSON

mkInventoryItemType :: [(String, String)] → Q [Dec]
mkInventoryItemType fields = do
  let typeName = mkName "InventoryItem" -- Name of the data type
      conName = mkName "InventoryItem" -- Name of the data constructor
      -- Create strict fields for the record syntax data type
      vars = [ (mkName fieldName, ConT (mkName fieldType))
              | (fieldName, fieldType) <- fields ]
      recC = RecC conName [(mkName fieldName, Bang NoSourceUnpackedness
                          SourceStrict, fieldType)
                          | (fieldName, fieldType) <- vars]
```

```

-- Define the data type declaration, including deriving common instances
dataType = DataD [] typeName [] Nothing [recC] [DerivClause Nothing [ConT ''
    Show, ConT ''Eq, ConT ''Generic, ConT ''FromJSON, ConT ''ToJSON]]

-- Report a message during compilation to indicate TH execution
reportSplice ("Template Haskell: Generating InventoryItem data type...")

-- Return a list containing the generated data type declaration
return [dataType]

-- In the same file, or a separate one, you might splice this:
-- $(mkInventoryItemType inventoryFields)

```

Listing 2: Conceptual Template Haskell Function (templates/THGen.hs)

Finally, in the module where you want to use the ‘InventoryItem’ type (e.g., ‘app/Inventory.hs’), you use a Template Haskell splice to execute the ‘mkInventoryItemType’ function and insert the generated code into your program.

```

{-# LANGUAGE TemplateHaskell #-}
{-# LANGUAGE DeriveGeneric #-} -- Needed if Generic is not derived by TH

module Inventory (
    InventoryItem(..) -- Export the data type and its fields
) where

import THGen (mkInventoryItemType, inventoryFields) -- Import the TH function and
    field definition
import Data.Aeson -- Assuming you derive FromJSON/ToJSON
import GHC.Generics -- Needed for Generic derivation

-- Splice the data type declaration generated by Template Haskell
-- When this module is compiled, the compiler runs mkInventoryItemType
-- with inventoryFields and inserts the resulting data declaration here.
$(mkInventoryItemType inventoryFields)

-- Potentially splice other generated code here, e.g., database mapping functions
-- $(mkDatabaseConverters inventoryFields)

-- Now the InventoryItem data type is defined and available for use
-- in this module and any other module that imports Inventory.
-- item :: InventoryItem
-- item = InventoryItem { itemId = 1, name = "Widget", quantity = 10, price = 5.99 }

```

Listing 3: Splicing Template Haskell in Inventory Module (app/Inventory.hs)

When the compiler processes ‘app/Inventory.hs’ and encounters the splice ‘\$(mkInventoryItemType inventoryFields)’, it executes the ‘mkInventoryItemType’ function defined in ‘THGen.hs’. This function reads the ‘inventoryFields’ list and programmatically constructs the AST for the ‘data InventoryItem = ... deriving (...)’ declaration. This AST is then inserted into the source code of ‘Inventory.hs’ at the location of the splice, and the compiler continues compiling the now-modified source code.

The generated code, which is compiled as if it were manually written, would conceptually look like this:

```
data InventoryItem = InventoryItem
  { itemId :: Int
  , name :: String
  , quantity :: Int
  , price :: Double
  } deriving (Show, Eq, Generic, FromJSON, ToJSON)
```

Listing 4: Generated Haskell Data Type (Conceptual Result of TH)

This process demonstrates compile-time metaprogramming because the definition of the ‘InventoryItem’ data type is not explicitly written by the developer in ‘Inventory.hs’ but is *generated* by another piece of Haskell code (‘mkInventoryItemType’) that runs during the compilation phase. This significantly reduces boilerplate, especially if you have many data types with similar structures or if you need to derive many type class instances. It also ensures consistency: if you need to add a field, you modify the ‘inventoryFields’ list, and the TH code automatically updates the data type and derived instances upon recompilation.

In a more advanced scenario, Template Haskell could also be used to generate code for:

- Functions to convert between the ‘InventoryItem’ type and a database row format, mapping Haskell types to database column types.
- SQL queries as strings or more structured query representations, potentially embedding field names from the ‘inventoryFields’ definition directly into queries.

```
=====
WELCOME TO ROOT CLI INVENTORY SYSTEM
[ ADMIN SIDE ]
=====

Press Enter to continue...

Select a table:
1. Address
2. Category
3. Coupon
4. Customer
5. Inventory
6. Payment
7. Product
8. ProductCategory
9. Review
10. Exit
1
Select operation:
1. Insert
2. Update
3. Delete
4. GetByID
5. GetAll
6. Go Back
1
addressId (INTEGER): 11
city (TEXT): bangalore
```

CLI Init

```
1
Select operation:
1. Insert
2. Update
3. Delete
4. GetByID
5. GetAll
6. Go Back
1
addressId (INTEGER): 11
city (TEXT): bangalore
customerId (INTEGER): 12
line1 (TEXT): hosur road
line2 (TEXT): electronic city
state (TEXT): karnataka
zipCode (TEXT): 560100

Running SQL: INSERT INTO Address (addressId
Inserted successfully.
```

Insert Operation

```
Select operation:
1. Insert
2. Update
3. Delete
4. GetByID
5. GetAll
6. Go Back
1
Running SQL: SELECT * FROM Address;
addressId | city | customerId | line1 | line2 | state | zipCode
-----
addressId 1 | city 1 | customerId 12 | MC Road | Near Mall | Mumbai | state MH | zipCode 400001
addressId 2 | city 2 | customerId 20 | Ring Road | | | Delhi | state DL | zipCode 110001
addressId 3 | city 3 | customerId 30 | Residency Rd | Apt 10 | Bangalore | state KA | zipCode 560001
addressId 4 | city 4 | customerId 70 | Canal Street | 1st Floor | Kolkata | state WB | zipCode 700001
addressId 5 | city 5 | customerId 20 | Nageswara | | | | state TN | zipCode 600004
addressId 11 | city 11 | customerId | Hosur Road | | | Bangalore | state Karnataka | zipCode 560100

Do you want to quit?
```

View All Operation

```
Select operation:
1. Insert
2. Update
3. Delete
4. GetByID
5. GetAll
6. Go Back
4
addressId (INTEGER): 11

Running SQL: SELECT * FROM Address WHERE addressId='11';
addressId | city | customerId | line1 | line2 | state | zipCode
-----
addressId 11 | city 11 | customerId 12 | Hosur Road | | | Bangalore | state Karnataka | zipCode 560100
```

View By Id Operation

```
Select operation:
1. Insert
2. Update
3. Delete
4. GetByID
5. GetAll
6. Go Back
2
addressId (INTEGER): 11
city (TEXT): bangalore
customerId (INTEGER): 12
line1 (TEXT): hosur road
line2 (TEXT): electronic city
state (TEXT): karnataka
zipCode (TEXT): 560100

Running SQL: UPDATE Address SET WHERE addressId='11' AND city='bangalore' AND customerId='12' AND line1='HOSUR ROAD' AND 11;
Error: SQLited returned I/O error while attempting to perform prepare "UPDATE Address SET WHERE addressId='11' AND city='11' AND customerId='HOSUR ROAD' AND zipCode='HOSUR ROAD' (2) code 'SQLITE_ERROR', syntax error.
```

Failed Update Operation

```
Select operation:
1. Insert
2. Update
3. Delete
4. GetByID
5. GetAll
6. Go Back
3
addressId (INTEGER): 11

Running SQL: DELETE FROM Address WHERE addressId='11';
Deleted successfully.
```

Delete Operation

- Boilerplate code for interacting with specific database libraries, generating functions for inserting, selecting, updating, and deleting ‘InventoryItem’ records.
- API endpoint definitions or client code based on the data structure, ensuring that the frontend and backend agree on the data format.

By using Template Haskell for database structure generation and related code, we effectively move computations and code definition from manual coding or runtime configuration to compile time, leveraging the compiler’s power to automate repetitive tasks and ensure static correctness based on the provided structure definition.

4.2 Runtime Metaprogramming

Runtime metaprogramming involves the ability of a program to manipulate or generate code and modify its behavior while it is executing. This offers significant flexibility and adaptability, allowing the program to respond dynamically to external input, changing conditions, or user commands. In our Inventory Management System, we demonstrate runtime metaprogramming through two key features: the dynamic handling of CRUD operations and the implementation of a user-definable function pipeline mechanism.

4.2.1 Dynamic CRUD Operations

In many applications, the logic for performing CRUD (Create, Read, Update, Delete) operations is hardcoded with specific function calls for each type of operation. For example, a web server might have distinct handler functions for ‘/create-item’, ‘/read-item’, ‘/update-item’, and ‘/delete-item’. In our system, we introduce a layer of runtime metaprogramming by designing a mechanism that can interpret generic requests and dynamically dispatch to the correct CRUD handling logic based on information contained within the request data itself.

This dynamic behavior is achieved by defining a single data type that encapsulates all possible inventory operations and then using a central processing function that analyzes the structure of this runtime data to determine which specific action to perform.

We define a data type, ‘InventoryRequest’, which represents the different kinds of requests the server can receive related to inventory management. This data type uses sum types (different constructors) to distinguish between the operations.

```
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE DeriveGeneric #-}

module Database (
    -- ... other exports
    InventoryRequest(..)
    -- ...
) where

import Data.Aeson
import Data.Text (Text, pack)
import qualified Data.Map as M
import Inventory -- Import the InventoryItem type (generated by TH)
import GHC.Generics

-- Data type to represent incoming requests, determined at runtime
data InventoryRequest =
    CreateItemReq InventoryItem
```

```

| ReadItemReq Int -- Item ID
| UpdateItemReq Int InventoryItem -- Item ID and new data
| DeleteItemReq Int -- Item ID
| ListItemsReq
deriving (Show, Generic) -- Derive Generic for Aeson instances

-- Automatically derive FromJSON and ToJSON instances for InventoryRequest
-- This allows parsing JSON requests into this Haskell data type at runtime.
instance FromJSON InventoryRequest
instance ToJSON InventoryRequest

-- ... rest of the Database module

```

Listing 5: Conceptual Runtime Request Data Type (app/Database.hs)

The server receives requests, likely in a data format like JSON, from the frontend. This JSON is then parsed into a value of the ‘InventoryRequest’ type. The specific constructor of the ‘InventoryRequest’ value (‘CreateItemReq’, ‘ReadItemReq’, etc.) is determined by the structure and content of the incoming JSON data, which is available only at runtime.

We then implement a central function, ‘processRequest’, which takes an ‘InventoryRequest’ value and the current state of the inventory database (‘InventoryDB’) and performs the appropriate action.

```

{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE DeriveGeneric #-}

module Database (
    InventoryDB,
    emptyDB,
    InventoryRequest(..),
    processRequest,
    InventoryResponse(..)
) where
import Data.Aeson hiding (json)
import Data.Text (Text, pack)
import qualified Data.Map as M
import Inventory -- Import InventoryItem
import Data.Maybe (fromMaybe)
import GHC.Generics

-- Conceptual In-Memory Database
type InventoryDB = M.Map Int InventoryItem

emptyDB :: InventoryDB
emptyDB = M.empty

-- Data type to represent incoming requests, determined at runtime
data InventoryRequest =
    CreateItemReq InventoryItem
  | ReadItemReq Int -- Item ID
  | UpdateItemReq Int InventoryItem -- Item ID and new data
  | DeleteItemReq Int -- Item ID
  | ListItemsReq
deriving (Show, Generic)

```

```

instance FromJSON InventoryRequest
instance ToJSON InventoryRequest

-- Data type to represent responses
data InventoryResponse =
    SuccessResponse Value
  | ErrorResponse Text
  deriving (Show, Generic)

instance ToJSON InventoryResponse

-- Process requests dynamically based on the request type received at runtime
processRequest :: InventoryRequest → InventoryDB → (InventoryDB, InventoryResponse)
processRequest request db = case request of
    CreateItemReq item →
        let newId = if M.null db then 1 else fst (M.findMax db) + 1
            -- In a real scenario, you'd handle ID assignment carefully,
            itemWithId = item { itemId = newId } -- Conceptual update, might need Lens
            newDB = M.insert newId itemWithId db
        in (newDB, SuccessResponse (toJSON itemWithId)) -- Return the created item
    ReadItemReq itemId →
        case M.lookup itemId db of
            Just item → (db, SuccessResponse (toJSON item))
            Nothing → (db, ErrorResponse "Item not found")
    UpdateItemReq itemId updatedItem →
        if M.member itemId db
        then let newDB = M.insert itemId updatedItem db
            in (newDB, SuccessResponse (toJSON updatedItem)) -- Return the updated
                item
        else (db, ErrorResponse "Item not found for update")
    DeleteItemReq itemId →
        let (maybeItem, newDB) = M.updateLookupWithKey (\_ _ → Nothing) itemId db
        in case maybeItem of
            Just _ → (newDB, SuccessResponse (toJSON $ object ["deleted" .= True, "
                itemId" .= itemId]))
            Nothing → (db, ErrorResponse "Item not found for deletion")
    ListItemsReq →
        (db, SuccessResponse (toJSON $ M.elems db))

```

Listing 6: Conceptual Runtime Request Processing (app/Database.hs)

In this implementation, the ‘processRequest’ function uses pattern matching on the ‘InventoryRequest’ value. The specific branch of the ‘case’ statement that is executed is determined by the constructor of the ‘request’ value, which is derived from the incoming runtime data. This is a form of runtime metaprogramming because the program’s execution path and the specific database operation performed are not hardcoded to specific API endpoints or function calls but are dynamically decided based on the interpretation of the data received during runtime. The structure of the incoming data effectively dictates which part of the program’s logic is activated and executed.

This approach provides a flexible way to handle inventory operations. If a new type of operation were needed, we would extend the ‘InventoryRequest’ data type with a new constructor and add a corresponding case to the ‘processRequest’ function. The server’s request handling layer would remain relatively stable, focusing on parsing the generic ‘InventoryRequest’ type.

4.2.2 User-Definable Function Pipelines

A more explicit and powerful demonstration of runtime metaprogramming in our project is the implementation of a user-definable and executable function pipeline mechanism. This feature allows users to compose sequences of available functions dynamically at runtime, name these compositions, and then execute them on the current inventory data. This is akin to allowing users to create and run small, custom data processing scripts or workflows on the fly, without requiring changes to the compiled code.

The core idea is to treat functions as data that can be selected, arranged in a sequence, stored, and executed during the program's runtime. This involves several steps:

- **Representing Available Functions:** We maintain a registry of functions that are available for use in pipelines. These functions operate on the inventory data (typically a list of 'InventoryItem's) and perform specific transformations or final operations. We store these functions in a data structure (like a 'Map') where they can be looked up by a runtime identifier, such as a string name.
- **Parsing User Input:** The system receives user input, likely as a string, that describes the desired sequence of functions to be applied in the pipeline. A parser interprets this string to identify the names of the functions and their intended order.
- **Constructing the Pipeline:** Based on the parsed input, the system constructs an internal representation of the pipeline. This representation is essentially a data structure (like a list) that holds the actual function values corresponding to the names provided by the user. This step involves looking up the function names in the registry of available functions.
- **Executing the Pipeline:** The system provides a mechanism to execute a constructed pipeline. This involves iterating through the list of functions in the pipeline and applying each function sequentially to the output of the previous function, starting with the initial inventory data.
- **Naming and Storing Pipelines:** Users can name their created pipeline sequences and store them in the server's runtime state. This allows them to reuse previously defined pipelines without having to redefine them each time.

Let's consider the implementation details. We define the types of functions that can be part of a pipeline. Intermediate functions will transform a list of 'InventoryItem's into another list of 'InventoryItem's ('InventoryTransform'). Final operations will take a list of 'InventoryItem's and produce a result that can be sent back to the user (e.g., a summary or formatted output), which we'll represent as 'Value' for JSON serialization.

To store functions with potentially different return types ('[InventoryItem]' for transforms, 'Value' for finals) in the same list representing the pipeline, we use existential quantification in the 'PipelineOperation' data type.

This conceptual pipeline logic demonstrates runtime metaprogramming in several ways:

- **Dynamic Code Construction:** The 'buildPipeline' function takes a list of 'Text' values (operation names) provided at runtime and constructs a list of actual function values ('InventoryPipeline'). This process of assembling a sequence of executable functions based on runtime input is a form of dynamic code construction.
- **Code as Data:** The functions themselves ('InventoryTransform', 'FinalOperation') are treated as data that can be stored in maps ('availableTransforms', 'availableFinalOperations') and lists ('InventoryPipeline'). This ability to manipulate functions as data is fundamental to enabling dynamic code manipulation.

- **Dynamic Execution:** The ‘executePipeline’ function iterates through the runtime-constructed ‘InventoryPipeline’ (the list of functions) and applies each function sequentially to the data. The specific sequence of operations performed is determined by the pipeline structure, which was defined by the user at runtime. This is a clear example of dynamic code execution.

The use of higher-order functions is absolutely crucial for this pipeline mechanism. Functions like ‘sortBy’ (which takes a comparison function), ‘foldl’ (which takes a combining function), and the ability to store and apply functions from a list are all enabled by treating functions as first-class citizens. Without higher-order functions, implementing such a flexible and dynamic pipeline system would be significantly more complex, likely requiring explicit pattern matching on operation names and manual application logic for each function type.

This dynamic pipeline creation and execution feature provides a powerful way for users to customize data processing workflows at runtime, showcasing the flexibility that runtime metaprogramming, combined with functional programming principles, can offer.

4.2.3 Server Module (‘app/Server.hs’ - Conceptual)

The server module is the central component that orchestrates the interaction between the frontend, the inventory data, and the metaprogramming logic. It is responsible for receiving requests, processing them using the dynamic CRUD and pipeline mechanisms, and sending back responses. We use a lightweight Haskell web framework (conceptually, like **Scotty**) to handle HTTP requests and routing.

The server maintains the application’s state, which includes the current inventory database and the collection of user-defined, named pipelines. Since the server might handle concurrent requests, we need a mechanism for safe access to this shared, mutable state, such as ‘MVar’.

```
{-# LANGUAGE OverloadedStrings #-}

module Main where

import Web.Scotty
import Network.Wai.Middleware.RequestLogger (logStdoutDev)
import Network.Wai.Middleware.Cors (cors, simpleCorsResourcePolicy,
    CorsResourcePolicy(..))
import Database.SQLite.Simple (open, Connection)

import qualified Links.CustomerLinks as Customer
import qualified Links.ProductLinks as Product
import qualified Links.ReviewLinks as Review
import qualified Links.AddressLinks as Address
import qualified Links.InventoryLinks as Inventory
import qualified Links.PaymentLinks as Payment
import qualified Links.CouponLinks as Coupon
import qualified Links.CategoryLinks as Category
import qualified Links.ProductCategoryLinks as ProductCategory

dbPath :: String
dbPath = "database/INVENTORY.db"

main :: IO ()
main = do
    conn <- open dbPath
```



```

putStrLn " Starting server on http://localhost:3000 ..."
scotty 3000 $ do
  middleware logStdoutDev
  middleware $ cors (const $ Just corsPolicy)
  get "/" $ text "WELCOME TO THE INVENTORY MANAGEMENT SYSTEM"
  Customer.registerRoutes conn
  Product.registerRoutes conn
  Review.registerRoutes conn
  Address.registerRoutes conn
  Inventory.registerRoutes conn
  Payment.registerRoutes conn
  Coupon.registerRoutes conn
  Category.registerRoutes conn
  ProductCategory.registerRoutes conn

-- CORS Policy to allow frontend (localhost:3001) to communicate
corsPolicy :: CorsResourcePolicy
corsPolicy = simpleCorsResourcePolicy
  { corsOrigins = Just (["http://localhost:3001"], True)
  , corsRequestHeaders = ["Content-Type", "Authorization"]
  , corsMethods = ["GET", "POST", "PUT", "DELETE", "OPTIONS"]
  }

```

This conceptual server code demonstrates how the runtime CRUD handling (‘processRequest’) and the dynamic pipeline management (‘buildPipeline’, ‘executePipeline’, ‘namedPipelines’) are integrated into a web application using the Scotty framework. The server receives runtime requests (HTTP requests with JSON payloads), parses them, and dynamically calls the appropriate functions based on the request data (for CRUD) or the stored pipeline definitions (for pipeline execution). The use of ‘MVar’ is crucial for safely managing the shared, mutable server state (the inventory database and the defined pipelines) in a concurrent environment, ensuring that multiple requests can be handled without data corruption. The server acts as the central hub, interpreting runtime input and orchestrating the execution of the appropriate metaprogramming logic.

5 How to Build and Run

This section provides instructions on how to build and run the Inventory Management System project. These steps assume you have the necessary Haskell development environment set up, including GHC and Stack.

5.1 Prerequisites

Before you can build and run the project, ensure you have the following installed on your system:

- **GHC (Glasgow Haskell Compiler):** The project is built using GHC. The specific version is typically specified in the ‘stack.yaml’ file. Stack will automatically download and manage the correct GHC version for you if it’s not already installed.
- **Stack or Cabal:** You need a Haskell build tool to manage dependencies and build the project. These instructions use Stack, which is generally recommended for its ease of use and reliable dependency management through build plans.
- **Git:** To clone the project repository from GitHub.

If you don't have Stack installed, you can follow the instructions on the official Stack website: https://docs.haskellstack.org/en/stable/install_and_upgrade/

5.2 Building the Project

Follow these steps to clone the repository and build the Haskell backend:

- **Clone the repository:** Open your terminal or command prompt and clone the project repository from GitHub.

```
git clone https://github.com/Tanish-pat/Programming_Languages_Project.git
```

- **Navigate to the project directory:** Change your current directory to the cloned repository's root folder.

```
cd Programming_Languages_Project
```

- **Build the project:** Use Cabal to build the project and its dependencies. This step is where the compile-time metaprogramming using Template Haskell will be executed.

```
cabal build all
```

Or you can use the direct executable file also which is the the deploy file

```
./deploy.sh
```

- Read the ‘.cabal’ file and ‘cabal.project’ (if present) to determine the build configuration, dependencies, and source files.
- Use the installed GHC version (configured globally or via a ‘cabal.project’) to compile the project.
- If any dependencies are missing or outdated, Cabal will download and install them in a local sandbox under ‘dist-newstyle/’.
- Cabal will then compile your project's source files (‘.hs’ files). If any modules contain Template Haskell splices, the embedded Template Haskell code will be executed during compilation to generate code on the fly.
- Finally, the resulting executables or libraries will be placed in the appropriate ‘dist-newstyle’ build directory.

5.3 Running the Server

Once the project is successfully built, you can run the Haskell backend server:

- **Execute the server:** Use Stack to execute the compiled server program.

```
cabal run server
```

The server should start and print messages to the console indicating that it is running and on which port it is listening (e.g., "Listening on port 3000").

- **Access the system:** With the server running, you can now interact with the Inventory Management System.
 - If you have a separate frontend application, ensure it is running and configured to send API requests to the address and port where your Haskell server is listening (, 'http://localhost:3000').
 - Alternatively, you can use command-line tools like 'curl' or API development tools like Postman or Insomnia to send HTTP requests directly to the server's defined API endpoints ('POST http://localhost:3000/inventory').

5.4 Running the Frontend

If your project includes a separate frontend application (e.g., built with JavaScript/React/Vue), follow the specific instructions provided for that frontend project to build and run it. Ensure that the frontend is configured to communicate with the Haskell backend server running on the correct address and port. The frontend acts as the client, sending requests to the backend API to perform inventory operations and manage pipelines.

6 Demonstrations

This section delineates three targeted demonstrations that validate the system's support for metaprogramming (both compile-time and runtime) and higher-order functions in a real-world application context.

6.1 Compile-Time Metaprogramming: Template Haskell Code Generation

This demonstration validates the generation of typed inventory structures via Template Haskell (TH):

- **Input Specification:** Present the field specification list (e.g., `inventoryFields i`) as the single source of truth for inventory structure. Highlight the abstraction from boilerplate code.
- **Code Generator:** Show the TH generator (e.g., `mkInventoryItemType`) that transforms the field list into a `data` declaration via TH AST constructors (`mkName`, `RecC`, `DataD`, etc.).
- **Splice Invocation:** Show where the splice `$(mkInventoryItemType inventoryFields)` is invoked (e.g., `app/Inventory.hs`), triggering TH execution during compilation.
- **Value Proposition:** Emphasize:
 - Elimination of boilerplate.
 - Strong consistency across modules.
 - Centralized schema definition.
 - Compile-time safety for type structure.

6.2 Runtime Metaprogramming: Dynamic CRUD Dispatch

This demonstration validates runtime logic dispatch based on deserialized input types:

- **Runtime Request Type:** Present the `InventoryRequest` ADT (in `app/Database.hs`)—each constructor maps to a specific CRUD operation.
- **Dispatch Logic:** Show `processRequest`, which pattern matches on `InventoryRequest` to invoke corresponding operations dynamically.
- **Request Execution:** Use `curl` or Postman to invoke each CRUD endpoint:
 - Create: JSON maps to `CreateItemReq`.
 - Read/Update/Delete: JSON includes item ID and maps to corresponding constructors.
 - List: maps to `ListItemsReq`.
- **Dynamic Branching:** Reinforce that logic is selected at runtime based on input structure—matching data to behavior dynamically.

6.3 Runtime Function Composition: Pipeline Construction and Execution

This demonstration validates dynamic function composition and higher-order function usage:

- **Function Registry:** Show `availableTransforms` and `availableFinalOperations` (in `app/Pipeline.hs`)—string-keyed maps holding Haskell function values.
- **User Input:** Present a runtime-defined list of transformation names from user input (e.g., via frontend or API).
- **Pipeline Assembly:** Show `buildPipeline`, which constructs a list of function references from input strings by lookup and stores them in an

```

tensingTensingDELL:/mnt/c/Users/office/OneDrive/Desktop/cork Env/Semester-6/PL/Programming_Language_Project/Pipeline/Run_Time_Pipeline$ cabal run
MetaPipeline: Runtime + Compile-Time Power
-----
[1] Create New Pipeline
[2] Load Existing Pipeline
[3] View Built-in Templates
[4] Test Compile-Time Functions
[5] Exit
-----
> 2
Enter pipeline name to load: test
-----
>> Pipeline: test
  1. \x -> reverse x
  2. \x -> map toLower x
-----
[1] Apply Pipeline
[2] Return to Main Menu
> 1
Enter input string: Hello world!!
Output: !!dlrow olleh
-----

```

(a) Loading an Existing Pipeline

```

MetaPipeline: Runtime + Compile-Time Power
-----
[1] Create New Pipeline
[2] Load Existing Pipeline
[3] View Built-in Templates
[4] Test Compile-Time Functions
[5] Exit
-----
> 1
Enter pipeline name: new
Enter snippet #1 (or type DONE): \x -> reverse x
Snippet compiled!
Enter snippet #2 (or type DONE): \x -> map toLower x
Snippet compiled!
Enter snippet #3 (or type DONE): DONE
Pipeline saved successfully.
-----

```

(b) Defining a New Pipeline

```

MetaPipeline: Runtime + Compile-Time Power
-----
[1] Create New Pipeline
[2] Load Existing Pipeline
[3] View Built-in Templates
[4] Test Compile-Time Functions
[5] Exit
-----
> 2
Enter pipeline name to load: new
-----
>> Pipeline: new
  1. \x -> reverse x
  2. \x -> map toLower x
-----
[1] Apply Pipeline
[2] Return to Main Menu
> 1
Enter input string: Hello WORLD!!!
Output: !!dlrow olleh
-----

```

(c) Running the "new" Pipeline

```

MetaPipeline: Runtime + Compile-Time Power
-----
[1] Create New Pipeline
[2] Load Existing Pipeline
[3] View Built-in Templates
[4] Test Compile-Time Functions
[5] Exit
-----
> 3
-----
[1] Reverse string: \x -> reverse x
[2] Uppercase: \x -> map toUpper x
[3] Add prefix 'Hi: ': \x -> "Hi: " ++ x
[4] Add suffix '!!!': \x -> x ++ "!!!"
[5] First 5 characters: \x -> take 5 x
-----
MetaPipeline: Runtime + Compile-Time Power
-----
[1] Create New Pipeline
[2] Load Existing Pipeline
[3] View Built-in Templates
[4] Test Compile-Time Functions
[5] Exit
-----
> 4
-----
Compile-Time Functions Demo:
reverseString "MetaProgramming" = gnirmangorPateM
toUpperCase "haskell" = HASKELL
addExclaim "Wow" = Wow!!!
-----

```

(d) Viewing and Running "test" Pipeline

Figure 5: Runtime Pipeline Demonstration: Load, Define, Execute, and View Pipelines

InventoryPipeline.

- **Definition Demonstration:** Define a pipeline using `POST /pipeline/define/:name`; verify with `GET /pipeline/list`.
- **Execution Logic:** Show `executePipeline`, which applies stored functions to inventory data using `foldl`.
- **HOF Utilization:** Emphasize that `foldl` is a higher-order function that applies a lambda over a list of wrapped transformations, enabling compositional execution: `foldl (_acc (Transform f) -> f acc) items transforms`.
- **Execution Demonstration:** Use `GET /pipeline/execute/:name` to run the pipeline and show result. Optionally define and execute a new pipeline in the same session to demonstrate runtime mutability of system behavior.

```
al run Generator
Enter Module Name:
Hello
How many functions to create?
2
Function 1
Enter function name:
compOne
Enter argument names (space-separated):
x y z
Enter function body (in terms of arguments):
x * y / z
Function 2
Enter function name:
compTwo
Enter argument names (space-separated):
x y z
Enter function body (in terms of arguments):
x + y - z
```

```
compTwo
Enter argument names (space-separated):
x y z
Enter function body (in terms of arguments):
x + y - z
Module app/Generator/Hello.hs generated successfully.
Testing the generated module...
Testing the generated module interactively...
Function: compOne
Function: compOne
Expected 3 arguments.
Enter arguments (space-separated):
10 5 3
compOne 10 5 3 = 16.666666666666668
Function: compTwo
Expected 3 arguments.
Enter arguments (space-separated):
11 5 12
compTwo 11 5 12 = 4.0
```

(a) Compile-Time Metaprogramming Example 1

(b) Compile-Time Metaprogramming Example 2

Figure 6: Code Generator via Template Haskell

7 Team and Contributions

This project was a collaborative effort by the team members. The successful implementation of the Inventory Management System, showcasing both compile-time and runtime metaprogramming techniques in Haskell, was the result of shared effort and expertise. While specific tasks were divided during the development process to leverage individual strengths and manage workload, the interconnected nature of the project's components (Template Haskell influencing data structures used by runtime logic, runtime logic relying on higher-order functions and the generated data types) fostered a high degree of collaboration, communication, and shared understanding of the entire system's design and implementation. The breakdown of contributions is as follows:

Tanish Pathania (IMT2022049)

- Participated in **Project Planning and Design**, especially in outlining core features and determining integration points between compile-time and runtime metaprogramming.

- Took lead on the **Compile-Time Metaprogramming Implementation** using Template Haskell, including defining data specifications and writing TH code for type generation.
- Contributed to the **Testing and Debugging** efforts focused on validating Template Haskell code and integration with runtime components.
- Took ownership of drafting key portions of the **Documentation and Presentation**, detailing compile-time logic and technical insights.

Hemang Seth (IMT2022098)

- Contributed actively to **Project Planning and Design**, particularly in designing the runtime architecture and request pipeline.
- Led the implementation of **Runtime Metaprogramming**, focusing on ‘InventoryRequest’, ‘processRequest’, and the function pipeline execution framework.
- Engaged in **Testing and Debugging**, especially runtime dispatch logic and concurrency-related issues.
- Authored the runtime architecture sections in the **Documentation and Presentation**, and assisted in preparing presentation visuals.

Rutul Patel (IMT2022021)

- Contributed to **Project Planning and Design**, with a focus on interfacing between the backend and frontend.
- Developed the **Haskell Server Backend**, including API endpoints, HTTP request handling, and concurrency-safe state management.
- Took charge of the **Frontend Development (Side Chunk)**, creating a basic UI for testing inventory operations and pipeline executions.
- Participated in **Testing and Debugging** with focus on end-to-end system tests.
- Documented the server design and frontend interaction logic in the **Documentation and Presentation**.

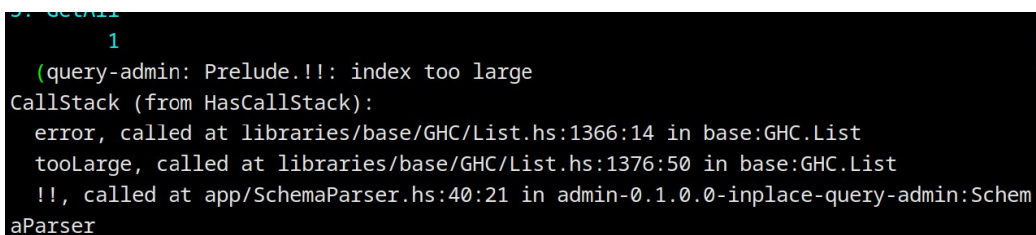
All members engaged in mutual reviews of each other’s components to ensure full-system understanding and maintain design coherence. The equal distribution of effort across planning, implementation, testing, and documentation reflects a balanced and cooperative team dynamic that led to the successful delivery of the project.

8 Challenges and Learnings

Developing a project that effectively integrates advanced metaprogramming techniques with functional programming principles in a language like Haskell presented a unique set of challenges. Navigating these challenges was an integral part of the learning process and provided valuable insights into the practical application of these powerful concepts.

8.1 Challenges Encountered

- **Mastering Template Haskell:** The initial learning curve for Template Haskell was significant. Understanding the monadic nature of ‘Q’ computations, the structure of Abstract Syntax Trees (ASTs) representing Haskell code, and the functions available for manipulating these ASTs required dedicated effort and experimentation. Debugging errors in Template Haskell code can be particularly challenging, as they occur during the compilation phase and error messages can sometimes be cryptic, referring to internal compiler representations. Generating correct and well-typed Haskell code programmatically demanded a deep understanding of both the target syntax and the TH API.
- **Designing the Interface Between Compile-Time and Runtime Code:** A key challenge was ensuring a seamless and type-safe interaction between the code generated at compile time by Template Haskell and the runtime logic of the application. For example, the runtime CRUD operations and pipeline execution needed to operate on the ‘InventoryItem’ data type, which was generated by TH. Any inconsistencies or mismatches between the assumptions made by the runtime code and the actual structure generated by TH could lead to difficult-to-diagnose errors. Careful design of interfaces and relying on Haskell’s type system to catch potential issues early was crucial.
- **Implementing Robust Runtime Metaprogramming:** Building dynamic systems that interpret runtime input to determine execution flow or construct code sequences, such as the function pipeline, introduced complexity. Challenges included:
 - **Parsing User Input:** Designing a robust and user-friendly way for users to define pipelines (e.g., parsing a string representation) and handling potential errors in the input (unknown function names, incorrect syntax).
 - **Managing Available Functions:** Creating and managing a registry of available functions for the pipeline, ensuring they have compatible types or providing mechanisms to handle type transformations between pipeline stages.
 - **Type Safety with Dynamic Behavior:** Reconciling Haskell’s strong static typing with the dynamic nature of runtime code execution was a significant challenge. Ensuring that a dynamically constructed pipeline of functions would be type-safe when executed required careful design of the ‘PipelineOperation’ type (e.g., using existential quantification) and the ‘executePipeline’ logic to handle potential type mismatches gracefully or report meaningful errors.
 - **Error Handling during Runtime Execution:** Implementing robust error handling for issues that might occur during pipeline execution (e.g., a function in the pipeline failing) and reporting these errors back to the user in a clear way.



```
1
(query-admin: Prelude.!!: index too large
CallStack (from HasCallStack):
  error, called at libraries/base/GHC/List.hs:1366:14 in base:GHC.List
  tooLarge, called at libraries/base/GHC/List.hs:1376:50 in base:GHC.List
  !!, called at app/SchemaParser.hs:40:21 in admin-0.1.0.0-inplace-query-admin:SchemaParser
```

Figure 7: Error Call Stack

- **State Management in a Concurrent Server:** While not directly a metaprogramming challenge, managing the shared, mutable state of the server (the ‘InventoryDB’ and the

‘namedPipelines’) in a concurrent web server environment required careful use of Haskell’s concurrency primitives like ‘MVar’. Ensuring thread-safe access to this state to prevent race conditions and data corruption added a layer of complexity to the backend implementation.

- **Balancing Power and Complexity:** Metaprogramming is a powerful tool, but it can also lead to code that is more abstract and potentially harder for developers unfamiliar with the techniques to understand and maintain. A challenge was to use metaprogramming judiciously where it provided clear benefits (like reducing boilerplate or enabling dynamic features) without making the codebase overly complex or obscure.

8.2 Learnings Gained

Despite the challenges, the project provided invaluable learning experiences and significantly deepened our understanding of functional programming and metaprogramming:

- **Practical Template Haskell Proficiency:** Gained hands-on experience writing and debugging Template Haskell code, understanding how to use quotes, splices, and the ‘Q’ monad to generate and manipulate Haskell code at compile time. This provided a concrete understanding of compile-time metaprogramming’s capabilities and limitations.
- **Designing for Runtime Flexibility:** Learned how to design systems in a statically-typed language like Haskell that exhibit dynamic behavior at runtime. This involved using data types to represent runtime commands (like ‘InventoryRequest’) and leveraging pattern matching for dynamic dispatch.
- **Deepened Understanding of Higher-Order Functions:** Solidified the understanding of higher-order functions not just as a stylistic feature but as fundamental tools for code abstraction, composition, and manipulation. The pipeline implementation clearly demonstrated how HOFs are essential for treating functions as data and enabling dynamic code execution.
- **Integrating Compile-Time and Runtime Concerns:** Gained experience in designing and implementing systems where code generated at compile time is consumed and utilized by runtime logic, highlighting the interplay between these two metaprogramming stages.
- **Haskell Concurrency and State Management:** Improved skills in designing and building concurrent applications in Haskell and safely managing shared mutable state using ‘MVar’.
- **Appreciating the Power of Functional Abstractions:** The project reinforced how functional programming principles and abstractions (purity, immutability, HOFs) provide a strong foundation for building complex and dynamic systems, including those that employ metaprogramming.
- **Strategic Use of Metaprogramming:** Learned to evaluate where metaprogramming techniques can provide the most value, considering the trade-offs between increased power/flexibility and potential complexity.

These challenges and the process of addressing them were fundamental to achieving the project goals and gaining a practical understanding of leveraging metaprogramming in functional programming.

9 Future Work

The current Inventory Management System successfully demonstrates the core concepts of compile-time and runtime metaprogramming in Haskell, along with the crucial role of higher-order functions. However, there are several avenues for future work that could extend the project’s functionality, improve its robustness, and explore more advanced metaprogramming techniques:

- **More Sophisticated Template Haskell Usage:** The current TH usage focuses on basic data type generation. Future work could explore more advanced applications:
 - **Database Integration Code Generation:** Generate comprehensive database interaction code (e.g., SQL queries, functions for converting between ‘InventoryItem’ and database rows) directly from the ‘inventoryFields’ definition using TH. This could involve integrating with a database library like Persistent or Opaleye and leveraging their TH capabilities.
 - **API Endpoint Generation:** Use Template Haskell to generate web server routes and request handling boilerplate based on the ‘InventoryRequest’ and ‘InventoryResponse’ data types, ensuring consistency between the API definition and the backend logic.
 - **Client Code Generation:** Explore generating client-side code (e.g., Haskell, or even code in other languages like TypeScript) for interacting with the backend API based on the backend data type definitions, further reducing boilerplate and ensuring type safety across the client-server boundary.
- **Enhanced Runtime Pipeline System:** The current pipeline system is a conceptual demonstration. Future work could significantly enhance its capabilities:
 - **Robust Parser with Arguments:** Implement a more sophisticated parser for pipeline definition strings that can handle function arguments (e.g., ‘filterQuantity(10)’), different data types flowing through the pipeline, and potentially more complex control flow constructs (e.g., conditional execution, looping).
 - **Runtime Type Checking:** Introduce a mechanism to perform type checking of the pipeline at definition time or before execution. This would provide better error messages to the user if a pipeline is ill-typed (e.g., the output type of one function doesn’t match the input type of the next). This could involve representing types at runtime and performing checks based on function signatures.
 - **Extensible Function Registry:** Allow users to define and register their own custom functions with the pipeline system at runtime. This would involve more advanced techniques like dynamic code loading or interpretation within a safe sandbox environment.
 - **Visual Pipeline Builder:** Develop a graphical user interface in the frontend that allows users to build pipelines visually by dragging and connecting function blocks, providing a more intuitive user experience.
- **Persistent Data Storage:** Replace the current in-memory ‘InventoryDB’ with a connection to a real database system (e.g., PostgreSQL, SQLite). This would require integrating a database library and implementing the necessary data access logic, potentially leveraging the TH-generated code for database interaction.

- **Improved Frontend:** Develop a more feature-rich and user-friendly frontend interface for the Inventory Management System, providing a complete user experience for managing inventory and interacting with the dynamic pipeline system.
- **Comprehensive Error Handling and Validation:** Implement more granular and user-friendly error handling and input validation on both the frontend and backend to make the system more robust and provide clearer feedback to the user.
- **Concurrency and Performance Optimizations:** Explore optimizing the server’s performance for handling a larger number of concurrent requests and a larger inventory dataset. This could involve profiling the application and identifying bottlenecks.
- **Exploring Other Metaprogramming Techniques:** Investigate other metaprogramming techniques relevant to functional programming that were not covered in this project, such as more advanced uses of generic programming (‘GHC.Generics’), or exploring DSLs for other aspects of the system (e.g., a DSL for defining business rules).

These potential future enhancements represent significant undertakings that would build upon the foundation established in this project, further exploring the capabilities and applications of metaprogramming in functional programming.

10 Conclusion

This project validated the synergy between **metaprogramming** and **functional programming** by implementing an **Inventory Management System** in **Haskell**. We illustrated how **code-as-data** techniques apply at both **compile time** and **runtime**, exploiting Haskell’s strengths.

Compile-Time Metaprogramming leveraged **Template Haskell** to generate typed data definitions and derive instances from a concise schema, drastically reducing **boilerplate**, ensuring **consistency**, and enhancing **maintainability**.

Runtime Metaprogramming appeared in two dimensions: dynamic **CRUD dispatch**, where `processRequest` routes based on incoming `InventoryRequest`, and a **user-definable function pipeline**, which composes, names, stores, and executes sequences of functions via `foldl`, demonstrating the treatment of **functions as data**.

The essential role of **higher-order functions** was underscored: first-class functions enabled dynamic pipeline assembly and execution. Despite challenges with Template Haskell and designing **robust runtime logic** in a statically typed context, the solutions devised enriched our practical expertise.

In summary, this case study proves that metaprogramming—often seen in dynamic or imperative languages—thrives in a **pure functional** setting. Combining compile-time generation and runtime adaptability yields systems that are **expressive**, **composable**, and **maintainable**, showcasing the power and versatility of modern functional programming.