tinyGHC - Report

1. Introduction

To be done later.

2. Lexer

tinyGHC's lexer is fully implemented using what is known as *"Lexer Combinators"*. Lexer combinators are a collection of functions that help break down code into smaller, more manageable pieces called tokens. These tokens are easier for a parser to understand and process. They are particularly useful for languages like Haskell, which has a complex syntax with whitespace playing a significant role in defining code structure.

Here's a breakdown of how lexer combinators work:

1. **Matching:** The most basic function is a matcher that recognizes a specific pattern in the code. For example, a matcher could identify keywords like "let" or "in".
2. **Sequencing:** Once you have matchers for individual tokens, you can combine them to create more complex patterns. Sequencers allow you to match a series of tokens in a particular order.
3. **Alternatives:** Sometimes, a piece of code can be represented by multiple token sequences. Alternators let you define multiple options for the lexer to consider during matching.
4. **Repetitions:** Certain patterns may involve repetitions of a token. Repeaters allow you to specify how many times a particular token should be matched.
5. **Transformations:** In some cases, you might want to modify the matched token before passing it to the parser. Transformers provide a way to apply transformations to the tokens during the lexing process.

By combining these fundamental building blocks, lexer combinators offer a powerful and flexible way to construct lexers for various programming languages. The lexers built with lexer combinators are often more concise, readable, and maintainable compared to lexers written from scratch.

The following are two examples on how matching an `Integer Literal` and an `Idenitifer` is done using lexer combinators. *For the rest of the tokens, more examples, and more technical details on the implementation, refer to the source code for the lexer and its comprehensive documentation.*

Example 1: Integer Literals

Given the above, here is the code for matching integer literals:

```
data LexerError = UnimplementedError deriving (Eq, Show)
data Token = IntLit Integer | ... -- other tokens

-- | `lexer` is a function that takes a stream of characters as an input and return eithe
lexer :: String -> Either LexerError [Token]
...


-- | Lexical matcher that succeeds if the character satisfies a predicate.
-- The `satisfies` function takes a predicate function `(Char -> Bool)` as input. This
```

```haskell
-- predicate determines whether a character matches the criteria. The function returns a
-- `Lexer Char` monadic computation.
satisfies :: (Char -> Bool) -> Lexer Char
satisfies p =
    Lexer <| \case
        c : cs | p c -> Right (c, cs)
        rest -> Left (unexpected rest)

-- This function matches one or more digits
digit :: Lexer Char
digit = satisfies isDigit

-- This function matches an optional minus sign
optionalMinus :: Lexer Char
optionalMinus = char '-' <|> pure ' '

-- This function combines the optional minus sign and the digit lexer
intLit :: Lexer Token
intLit = optionalMinus >> many digit >> fmap (IntLit . readMaybe)
```

Let's break down the code step by step:

1. **Data Types:**
   - `LexerError`: This type is used to represent errors that may occur during lexing. In this example, the only error defined is `UnimplementedError`.
   - `Token`: This is a variant type that can hold different kinds of tokens. Here, we only define `IntLit` for integer literals, but there could be other constructors for different types of tokens like strings or keywords.
2. `lexer` function: This function takes a string of source code as input and tries to return a list of tokens that represent the code. If an error occurs during lexing, it returns an `Either LexerError [Token]`, indicating an error.
3. **Matching Characters:**
   - `digit`: This function is a lexer that matches one or more digits. The `satisfies` function is a helper function that takes a predicate (a function that returns True or False) and returns a lexer that matches characters for which the predicate is True. In this case, the predicate `isDigit` checks if the character is a digit.
   - `optionalMinus`: This function is a lexer that matches an optional minus sign. The `<|>` operator is the disjunction operator in the `Alternative` type class. It allows us to combine two lexers such that the resulting lexer will succeed if either of the original lexers succeeds. So, here the lexer will match either a '-' or a space.
4. **Combining Lexers:**
   - `intLit`: This function combines the `optionalMinus` and `digit` lexers using the `>>` operator. The `>>` operator sequences two lexers. The resulting lexer will try to match a string that starts with an optional minus sign, followed by one or more digits.
5. **Transforming Results:**
   - `fmap`: The `fmap` function applies a function to the output of a lexer. In this case, the `fmap` function applies the `readMaybe` function to the string of digits that were matched by the lexer. The `readMaybe` function attempts to read a string as an integer and returns `Nothing` if the string cannot be parsed as an integer.

So, if the `intLit` lexer successfully matches a sequence of digits optionally preceded by a minus sign, the `fmap` function tries to convert the matched string of digits into an integer. *If the conversion is successful, an `IntLit` token is created with the integer value.* Otherwise, an error occurs during lexing.

Example 2: Identifiers

```haskell
-- Assuming data types from previous example

-- This function matches a single letter character
letter :: Lexer Char
letter = satisfies isAlpha

-- This function matches an underscore character
underscore :: Lexer Char
underscore = char '_'

-- This function matches the first character of an identifier (letter or underscore)
identifierStart :: Lexer Char
identifierStart = letter <|> underscore

-- This function matches subsequent characters in an identifier (letter, underscore, or d
identifierRest :: Lexer Char
identifierRest = letter <|> underscore <|> digit

-- This function combines the start and subsequent characters of an identifier
identifier :: Lexer String
identifier = identifierStart >> many identifierRest
```

Explanation:

1. **Matching Characters:**
   - `letter`: This lexer matches a single letter character using the `satisfies` function similar to the integer example.
   - `underscore`: This lexer matches an underscore character using the `char` function that takes the literal character to match.
2. **Identifier Components:**
   - `identifierStart`: This defines the first character of an identifier which can be either a letter or an underscore.
   - `identifierRest`: This defines subsequent characters in an identifier which can be letters, underscores, or digits. Identifiers in Haskell can include digits after the initial letter or underscore.
3. **Combining Lexers:**
   - `identifier`: This combines the `identifierStart` and `many identifierRest` using the `>>` operator. The `many` function takes a lexer and creates a new lexer that matches zero or more repetitions of the original lexer. So, `many identifierRest` matches any number of characters allowed in the rest of the identifier.

3. Parser

tinyGHC's parser is also fully implemented in what is known as "*Parser Combinators*—"similar to Lexer Combinators. Parser combinators are a powerful technique in Haskell for building parsers for various data formats. They provide a modular approach where you combine smaller parsing functions to create more complex ones. This section will go through the concepts and functionalities of parser combinators in Haskell.

How they work

Here's a step-by-step breakdown of how parser combinators work:

1. **Parser Representation:** The first step is defining the type of a parser. *It's typically a function that takes the input string and returns a pair. The first element of the pair is either a list of errors (representing parsing failures) or the successfully parsed value. The second element is the remaining unparsed string.* This allows for backtracking and handling of ambiguities.
2. **Basic Parsers:** We start with basic parsers that handle simple elements like characters (e.g., `satisfy (\c -> c == 'a')` matches the character 'a') or strings (e.g., `string "hello"` matches the literal string "hello").
3. **Combining Parsers:** The power comes from combining these basic parsers. Common combinators include:
   - **Sequencing:** `p >> q` parses `p` followed by `q`, returning the result of `q`.
   - **Choice:** `p <|> q` tries `p` first, if it fails, tries `q`. Returns the successful parse result.
   - **Repetition:** `many p` parses zero or more repetitions of `p`, returning a list of parsed values.
   - **Others**:
     - `many`: Parses zero or more occurrences of a parser.
     - `oneOrMore`: Parses one or more occurrences of a parser.
     - `sat`: Parses any character that satisfies a predicate.
     - `char`: Parses a specific character.
     - `string`: Parses a specific string literal.
     - `*>`: Sequence two parsers, applying the first and then the second.

Because of the complexity of the tinyGHC parser, the following section will give a simple example for parsing mathematical expressions using parser combinators—for demonstration purposes. *For full implementation details on how the tinyGHC's parses tinyHaskell and outputs the **Abstract Syntax Tree**, refer to the source code for the parser (`parser.hs`) and its full, comprehensive documentation.*

Example: Mathematical Expressions

```haskell
data Expr = Val Int | Plus Expr Expr | Minus Expr Expr deriving Show

data Parser a = Fail [String] | Success a String deriving Show

parse :: Parser a -> String -> Maybe (a, String)
parse (Success a rest) _ = Just (a, rest)
parse (Fail err) _ = Nothing

sat :: (Char -> Bool) -> Parser Char
sat pred str = case head str of
  c | pred c -> Success c (tail str)
  _ -> Fail ["Unexpected character"]
```

```haskell
char :: Char -> Parser Char
char expected = sat (== expected)

string :: String -> Parser String
string expected str = case take (length expected) str of
  subStr | subStr == expected -> Success expected (drop (length expected) str)
  _ -> Fail ["Expected string literal"]

digit :: Parser Char
digit = sat (\c -> elem c ['0'..'9'])

parseInt :: Parser Int
parseInt = many1 digit >>= (\digits -> readMaybe (concat digits) :: Maybe Int)
  >>= Maybe.fromJust . \case
      Nothing -> Fail ["Invalid integer format"]
      Just val -> return val

whitespace :: Parser ()
whitespace = many (sat (\c -> elem c [' ', '\t', '\n']))

-- Operator parsers
op :: Char -> Parser Char
op c = char c >> whitespace

addop :: Parser (Expr -> Expr -> Expr)
addop = op '+' >> return Plus

subop :: Parser (Expr -> Expr -> Expr)
subop = op '-' >> return Minus

-- Term parser (handles numbers and parentheses)
term :: Parser Expr
term = parens <|> parseInt

parens :: Parser Expr
parens = char '(' *> expr <* char ')'

-- Expression parser (handles terms and operators)
expr :: Parser Expr
expr = chainl1 term (addop <|> subop)

chainl1 :: Parser a -> Parser (b -> a -> a) -> Parser a
chainl1 p f = do
  x <- p
  rest <- many (f >>= (\g -> p >>= (\y -> return (\z -> g z y))))
  return (foldl (\acc y -> acc y) x rest)
```

Explanation:

1. **Data Type for Expressions:** We define a `Expr` data type to represent mathematical expressions like values, addition, and subtraction.
2. **Whitespace Handling:** We add a `whitespace` parser to skip whitespace characters between elements.

3. **Operator Parsers:** We define parsers for addition (`addop`) and subtraction (`subop`) operators using `op` and `whitespace`.
4. **Term Parser:** The `term` parser handles both numbers using `parseInt` and parenthesized expressions using `parens`.
5. **Expression Parser:** The `expr` parser uses `chainl1` to handle left-associative binary operations. It starts with a `term`, applies optional operators (`addop` or `subop`) followed by another `term`, and recursively combines them using a fold operation.

4. Abstract Syntax Tree

1. AST Data Types as Building Blocks:

- The AST is constructed using the data types defined earlier. These types act as nodes in the tree, encapsulating the syntactic elements of the parsed code.

2. Nested Structure:

- The AST mirrors the hierarchical structure of the Haskell code. It captures relationships between expressions, data types, and functions using nested data structures.

3. Example AST for a Simple Expression:

Haskell

```
parse expression "x + 5"  -- Assuming `x` is a defined variable
```

- AST Representation: Haskell

  ```
  Add (Var "x") (Lit 5)
  ```

- Graphical Visualization:

  ```
      Add
      /    \
    Var    Lit
    "x"      5
  ```

4. AST for a Conditional Expression:

Haskell

```
parse expression "if x > 0 then y else z"
```

- AST Representation: Haskell

  ```
  If (Greater (Var "x") (Lit 0)) (Var "y") (Var "z")
  ```

- Graphical Visualization:

```
        If
       / \
   Greater   Var
          / \ "z" Var Lit "x" 0
   Var "y"
```

5. AST for a Data Type Declaration:

```
parse statement "data List a = Nil | Cons a (List a)"
```

- AST Representation: Haskell

  ```
  DataDecl "List" ["a"] [Constructor "Nil" []
                         Constructor "Cons" ["a", Constructor "List" ["a"]]]
  ```

6. AST for a Function Definition:

Haskell

```
parse statement "f x y = x + y"
```

- AST Representation: Haskell

  ```
  FuncDecl "f" [Pattern "x", Pattern "y"] (Add (Var "x") (Var "y"))
  ```

Key Points:

- Each node in the AST holds specific information about the parsed code element.
- The structure accurately reflects the relationships and hierarchy of the original code.
- ASTs are instrumental for further analysis, interpretation, or transformation of the code

More Examples:

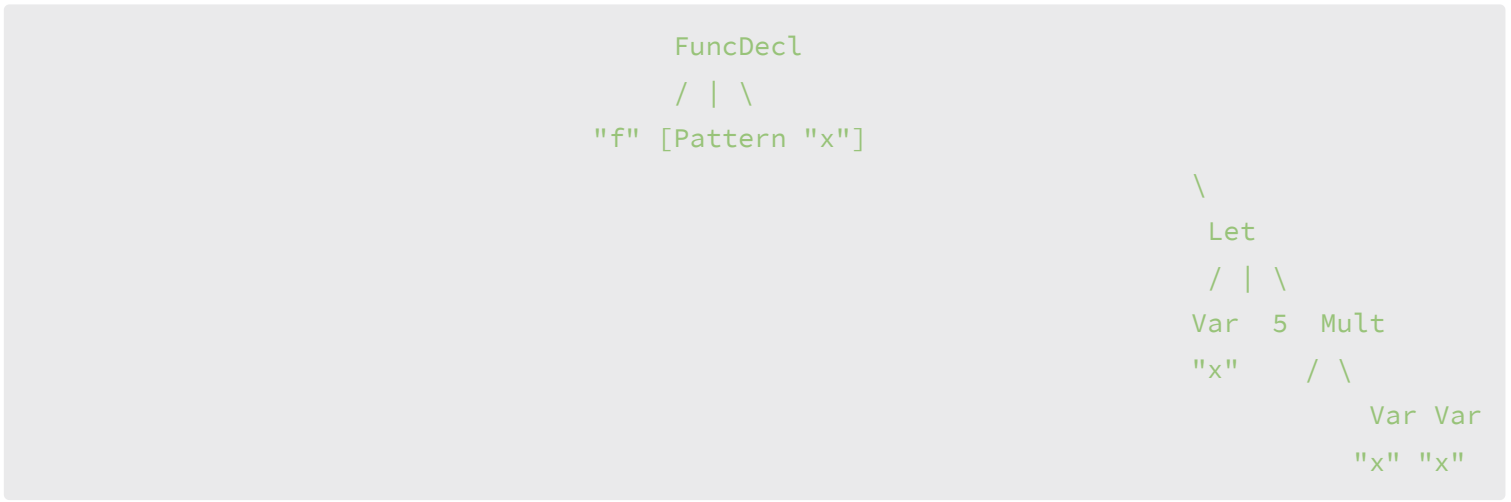
1. Let Expression:

- **Code:** `let x = 10 in x + 20`
- **AST:** `Let "x" (Lit 10) (Add (Var "x") (Lit 20))`
- **Visualization:**

```
                                    Let
                                   / | \
                                 Var  10  Add
                                 "x"     / \
                                            Var Lit
                                          "x" 20
```

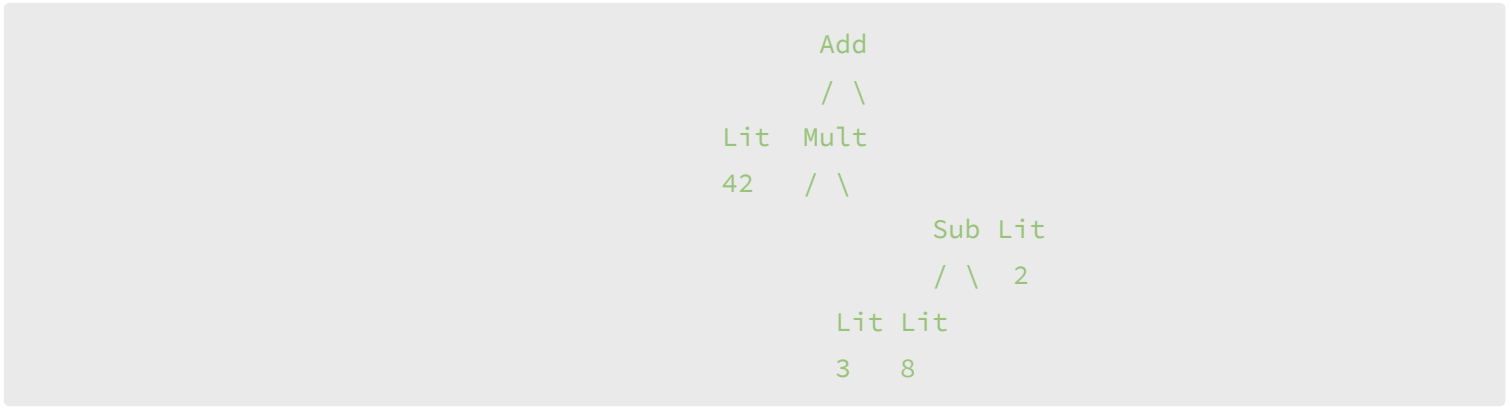
2. Where Expression:

- **Code:** `f x = x * x where x = 5`
- **AST:** `FuncDecl "f" [Pattern "x"] (Let "x" (Lit 5) (Mult (Var "x") (Var "x")))`
- **Visualization:**



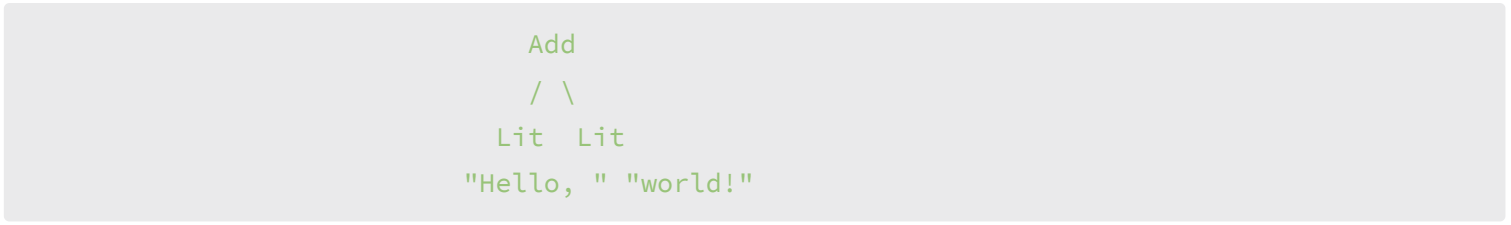
3. Arithmetic Expression:

- **Code:** `42 + (3 - 8) * 2`
- **AST:** `Add (Lit 42) (Mult (Sub (Lit 3) (Lit 8)) (Lit 2))`
- **Visualization:**



4. String Concatenation:

- **Code:** `"Hello, " ++ "world!"`
- **AST:** `Add (Lit "Hello, ") (Lit "world!")`
- **Visualization:**



5. Conditional Expression:

- **Code:** `if even x then "Even" else "Odd"`
- **AST:** `If (Call "even" [Var "x"]) (Lit "Even") (Lit "Odd")`
- **Visualization:**

```
        If
        / \
     Call  Lit
     |   "Odd"
   "even"
      / \
    Var Lit
    "x" "Even"
```

6. Case Expression:

- **Code:** `case x of Just y -> y; Nothing -> 0`
- **AST:** `Case (Var "x") [Match "Just" [Pattern "y"] (Var "y"), Match "Nothing" [] (Lit 0)]`
- **Visualization:**

```
          Case
          / | \
       Var   ... Match
       "x"   ... / \
                 "Nothing" Lit
                           0
                         /
                    Match
                    / \
                  "Just" ...
```

7. Multiple Function Heads:

- **Code:** `f (x, y) = x + y; f [x] = x * x`
- **AST:** `FuncDecl "f" [Pattern (Tuple [Pattern "x", Pattern "y"])] (Add (Var "x") (Var "y")), FuncDecl "f" [Pattern (List [Pattern "x"])] (Mult (Var "x") (Var "x"))`
- **Visualization:**

```
          FuncDecl
          / | \
       "f" Pattern
                    \
                     FuncDecl
                           / | \
                      "f" Pattern

\

...
```

8. Data Type Declaration:

- **Code:** `data Shape = Circle Float | Rectangle Float Float`

- **AST:** `DataDecl "Shape" [] [Constructor "Circle" [Constructor "Float" []], Constructor "Rectangle" [Constructor "Float" [], Constructor "Float" []]]`
- **Visualization:**

```
                                                DataDecl
                                                / | \
                                              "Shape"  ... Constructor
                                                                / | \
                                                              "Circle" ...
                                                                       \


Constructor
                                                                          /
| \

"Rectangle" ...
```

9. Type Synonym:

- **Code:** `type Name = String`
- **AST:** `TypeDecl "Name" (Constructor "String" [])`
- **Visualization:**

```
                      TypeDecl
                     / | \
                "Name" Constructor
```

5. Type Checker

5.1 Introduction

Haskell, a purely functional programming language, has established itself as a leader in the field of safe and reliable program development. At the heart of this reliability lies its robust type system, enforced by the Glasgow Haskell Compiler's (GHC) type checker. This introductory section delves into the core features of Haskell that necessitate a type checker, explores the intricate workings of GHC's type checking mechanism, and ultimately sheds light on its paramount role in ensuring program correctness.

Haskell is distinguished by its emphasis on strong typing, a paradigm where every expression and variable possesses a well-defined type. This type specifies the kind of data the value can hold (e.g., integers, strings, lists) and the operations that are valid on that data. Strong typing enforces a strict discipline upon the programmer, preventing nonsensical operations like adding a string to a boolean value. Imagine the chaos that would ensue if such an operation were allowed to pass unnoticed! By catching these type mismatches early in the development cycle, the type checker acts as a vigilant guardian, eliminating potential runtime errors and fostering more robust programs.

Furthermore, Haskell boasts a powerful feature called type inference. Unlike many other languages where the programmer must explicitly declare the type of every variable, Haskell allows the compiler

to infer the types based on the context in which a variable is used. This not only reduces boilerplate code but also empowers the programmer to focus on the logic of the program rather than low-level type annotations. However, this freedom comes with a caveat. While the compiler is adept at inferring types in most cases, there can be situations where the information gleaned from the code is insufficient to pinpoint a unique type. This is where the type checker steps in, playing a crucial role in disambiguating these situations and ensuring type consistency throughout the program.

The Glasgow Haskell Compiler (GHC) sits at the heart of the Haskell development ecosystem, responsible for translating human-written Haskell code into efficient machine code. A critical stage in this translation process is type checking, where GHC's meticulously crafted type checker scrutinizes the program for any potential type inconsistencies. Through a series of sophisticated algorithms and data structures, the type checker meticulously analyzes the program's structure, verifies type annotations (if provided), and infers types where necessary. This rigorous examination ensures that all operations are performed on compatible types, preventing a multitude of errors that could otherwise manifest at runtime and lead to program crashes or unexpected behavior.

In essence, GHC's type checker serves as the cornerstone of Haskell's robust type system. By enforcing strong typing, facilitating type inference, and meticulously checking for type mismatches, the type checker safeguards program correctness, fostering confidence in the reliability and predictability of Haskell code. The subsequent sections of this report will delve deeper into the intricate workings of GHC's type checker, demystifying its techniques and highlighting its significance in creating secure and dependable Haskell programs.

5.2 Background

The success of any programming language hinges on its ability to manage the complexities of data and the operations performed upon it. Type systems, a cornerstone of modern programming languages, play a pivotal role in achieving this very objective. In essence, a type system acts as a formal set of rules that assigns a type to every element within a program. These types serve a dual purpose:

2. **Data Classification:** Types act as labels that categorize data, specifying the kind of information a value can hold. Common types include integers (`Int`), floating-point numbers (`Float`), strings (`String`), and user-defined types like lists or custom data structures. By associating a type with each variable and expression, the type system ensures clarity and consistency in how data is manipulated within the program. Imagine a program treating a string as an integer; the results would be nonsensical! Types prevent such mishaps by explicitly stating what kind of data each variable represents.
3. **Operation Validation:** Types go beyond mere classification; they also dictate the permissible operations on that data. For instance, adding two integers makes perfect sense, but adding an integer to a string is meaningless. The type system acts as an enforcer, ensuring that only valid operations are applied to specific types. This not only enhances program safety by preventing nonsensical calculations but also improves code readability by making the intended use of data abundantly clear.

The benefits of a well-designed type system are manifold. By catching type mismatches early in the development cycle, during the compilation phase, the type checker significantly reduces the likelihood of runtime errors. These errors, if left undetected, can manifest as program crashes, unexpected behavior, or security vulnerabilities. Furthermore, type systems promote code maintainability and reusability. By explicitly stating the types of variables and functions, the code becomes self-

documenting, allowing other developers to readily understand the expected inputs and outputs of different program components.

One of the most influential type systems in the realm of functional programming is the Hindley-Milner (HM) type system. Developed by Roger Hindley and Robin Milner in the 1970s, HM laid the foundation for type inference, a cornerstone of Haskell's type system. HM operates on a core set of principles:

- **Simple Types:** HM primarily deals with basic types like integers, booleans, and function types. Function types specify the expected input and output types of a function, promoting clarity and allowing the type checker to verify that functions are called with compatible arguments.
- **Type Variables:** HM employs type variables, denoted by uppercase letters like `T`, to represent unknown types. These variables act as placeholders, allowing the type checker to infer the most general type that fits a given expression. Type inference empowers programmers to write more concise code without explicitly declaring types in every instance.
- **Unification:** A core concept in HM type checking is unification, the process of merging two types to obtain a more specific type. For instance, unifying the type variables `T` and `Int` would result in the more specific type `Int`. Unification plays a crucial role in type inference, enabling the type checker to gradually refine the types of expressions as it traverses the program.

It's important to distinguish HM from another closely related type system, the Damas-Hindley (DH) type system. DH, a subset of HM, focuses solely on monomorphic functions, meaning functions that can only operate on a single, fixed type. While less flexible than HM, DH offers advantages in terms of simplicity and efficiency. Haskell, however, leverages the full power of HM, allowing for polymorphic functions that can operate on a wider range of types. This flexibility comes at the cost of increased complexity in the type checking process, but it aligns perfectly with Haskell's core philosophy of functional programming and code reusability.

The introduction of type systems and, more specifically, the HM type system, marked a significant leap forward in programming language design. By providing a mechanism for static type checking and type inference, HM paved the way for more robust, reliable, and maintainable programs. The following sections will delve deeper into how tinuGHC builds upon the foundation of HM to implement a sophisticated type checking mechanism for Haskell, ensuring the correctness and safety of Haskell programs.

5.3 Type Checker Architecture

5.3.1 The Type Checker Monad: Context and State Management

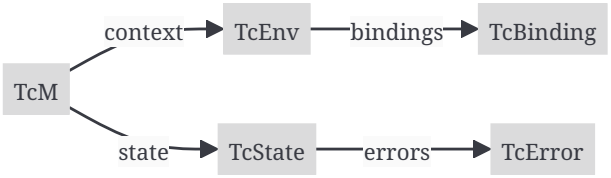
The type checker is implemented as a monad (from Category Theory), which provides a way to manage context and state during type checking. tinyGHC's type checker fully leverages the powerful abstraction level of monads. A monad, in essence, is a design pattern that encapsulates computations along with their context and state. This proves particularly beneficial for type checking as it allows the type checker to manage the following aspects:

- **Context:** During type checking, the type checker needs to maintain information about the surrounding code, such as the types of variables in scope and the types of function arguments. The type checker monad provides a structured way to store and access this contextual information, ensuring consistency throughout the type checking process.

- **State:** Type checking can involve accumulating information as the type checker traverses the AST. For instance, constraints generated for different expressions need to be collected and unified later on. The type checker monad acts as a container, allowing the type checker to store and manipulate this state information effectively.
- **Error Handling:** Type checking can encounter errors like type mismatches or missing type annotations. The type checker monad facilitates a structured approach to error handling. Errors can be wrapped within the monad and propagated back through the compilation process, allowing for informative error messages and graceful termination.

In essence, the type checker monad empowers tinyGHC's type checker to operate efficiently and reliably. By providing a framework for managing context, state, and errors, the monad ensures that type checking is robust, informative, and seamlessly integrated within the overall compilation pipeline.

The type checker monad, `TcM`, is defined as follows:



The `TcM` monad has three components:

- **TcEnv**: The type checker environment, which stores the current set of bindings and type constraints.
- **TcState**: The type checker state, which stores the current set of errors and warnings.
- **TcBinding**: A type checker binding, which represents a single binding of a variable to a type.

The type checker monad provides a way to thread the type checker environment and state through the type checking process, allowing the type checker to maintain a consistent view of the program's type structure.

The type checking process can be broken down into several stages:

1. **Type Inference**: The type checker infers the types of expressions and patterns, using a combination of type inference algorithms and type annotations.
2. **Type Checking**: The type checker checks that the inferred types are consistent with the program's type structure, using a combination of type checking rules and type constraints.
3. **Constraint Solving**: The type checker solves type constraints, using a combination of constraint solving algorithms and type inference.

5.3.2 Role of the Type Checker Monad

The type checker monad plays a crucial role in managing context and state during type checking. The monad provides a way to:

- **Thread the type checker environment**: The monad allows the type checker to maintain a consistent view of the program's type structure, by threading the type checker environment through the type checking process.
- **Manage type checker state**: The monad allows the type checker to manage its internal state, including errors and warnings, during the type checking process.

- **Provide a way to abstract over type checking**: The monad provides a way to abstract over the type checking process, allowing the type checker to be composed with other components of the tinyGHC compiler.

5.4 Type Checking and Inference Process

tinyGHC's type checker examines the program to ensure type consistency throughout the code. This section delves into the intricate steps involved in type checking a Haskell expression or statement, exploring how the type checker traverses the program's Abstract Syntax Tree (AST) and enforces specific type rules for various Haskell constructs.

5.4.1 Traversing the AST

The type checking process hinges on the concept of the Abstract Syntax Tree (AST). The AST serves as a structured representation of the program's syntax, capturing the relationships between expressions, variables, and functions. tinyGHC's type checker acts as a tireless traveler, meticulously traversing this tree-like structure, analyzing each node and applying the appropriate type checking rules.

Here's a breakdown of the general flow:

1. **Root Node:** The journey begins at the root node of the AST, which typically represents the entire program or a top-level function definition.
2. **Recursive Descent:** The type checker employs a recursive descent strategy. For each node encountered, it performs the following actions:
    - **Identify Node Type:** It determines the specific type of the node (e.g., function application, variable declaration, pattern matching).
    - **Apply Type Checking Rules:** Based on the node type, the type checker applies a predefined set of rules to verify type consistency. These rules might involve inferring types, performing unification, or checking for adherence to specific type constraints.
    - **Recurse on Children:** For nodes with child nodes (e.g., function applications with arguments), the type checker recursively descends down the tree, applying the same process to each child node.
3. **Bottom Up Type Inference:** It's important to note that type inference in tinyGHC often follows a bottom-up approach. As the type checker traverses the AST from leaves towards the root, it gathers information about the types of subexpressions. This information is then used to infer the types of more complex expressions higher up in the tree.

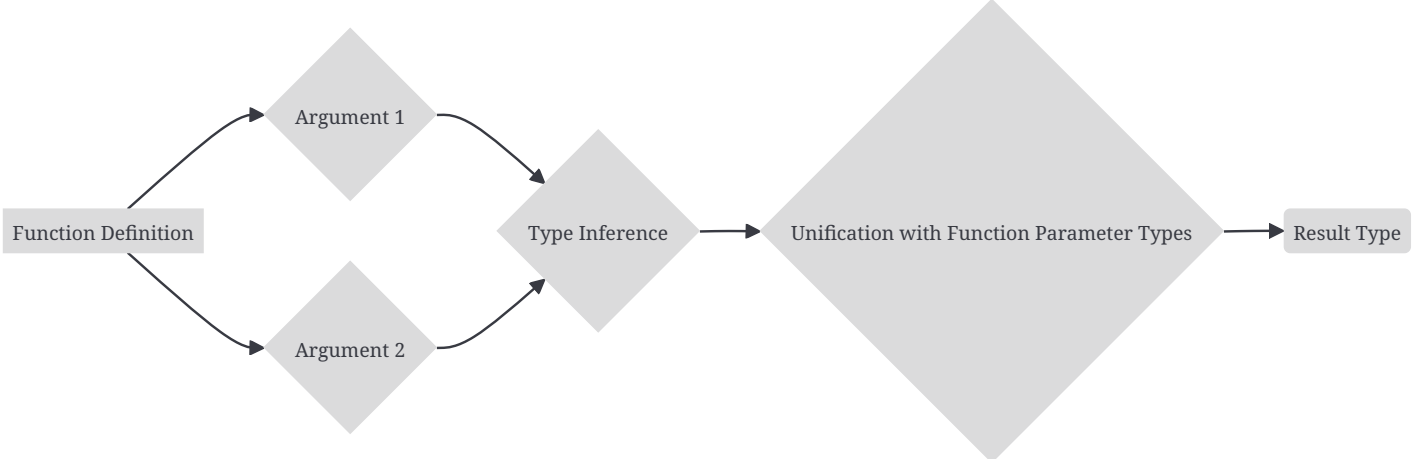
5.4.2 Type Checking Rules for Common Haskell Constructs

To illustrate the type checking process in more concrete terms, let's delve into specific type checking rules for some fundamental Haskell constructs:

1. Function Applications:

When encountering a function application, the type checker performs the following actions:

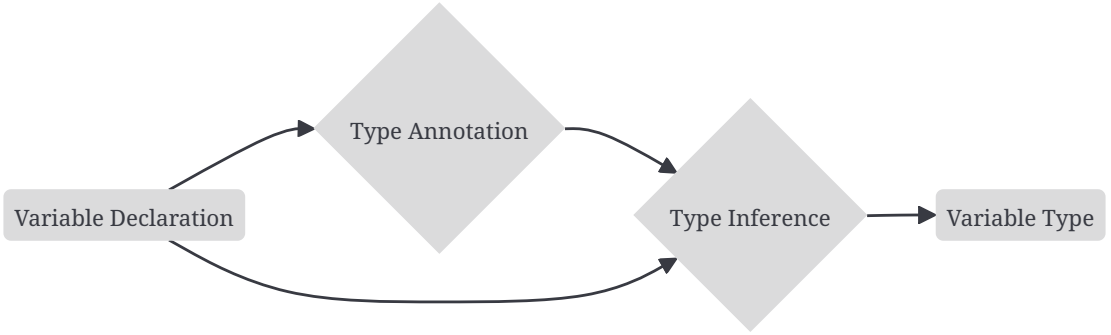
- It retrieves the type information for the function being applied (obtained from the function definition during a previous pass). This type typically specifies the expected types of the function's arguments and its return value.
- For each argument of the function application, the type checker infers a type based on the context in which the argument is used.
- The inferred argument types are then unified with the expected types specified in the function definition. If unification fails, a type error is reported.
- Finally, based on the successful unification, the type checker determines the return type of the function application.



2. Variable Declarations:

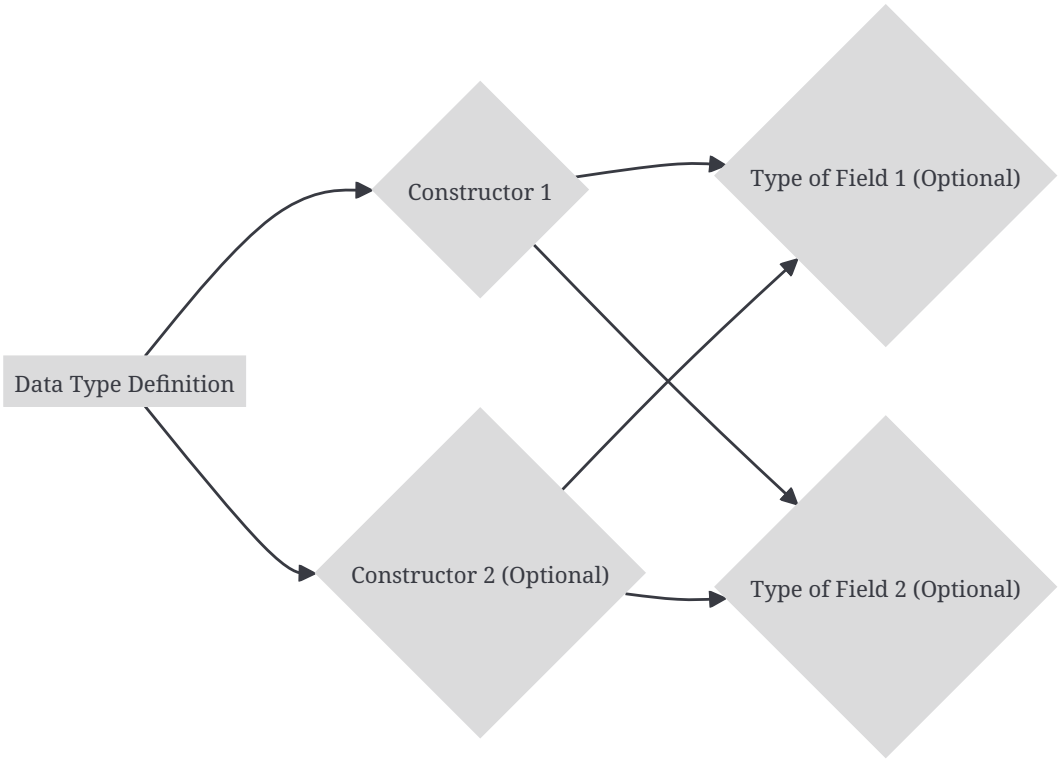
For variable declarations, the type checking process involves:

- Checking if the programmer has provided an explicit type annotation for the variable.
- If a type annotation is present, the type checker verifies its validity.
- If no type annotation is provided, the type checker infers a type based on the expression on the right-hand side of the declaration (if applicable) and the context in which the variable is used.
- The inferred or annotated type becomes the type associated with the variable throughout the program.



3. Data Types:
When encountering a data type definition, the type checker:

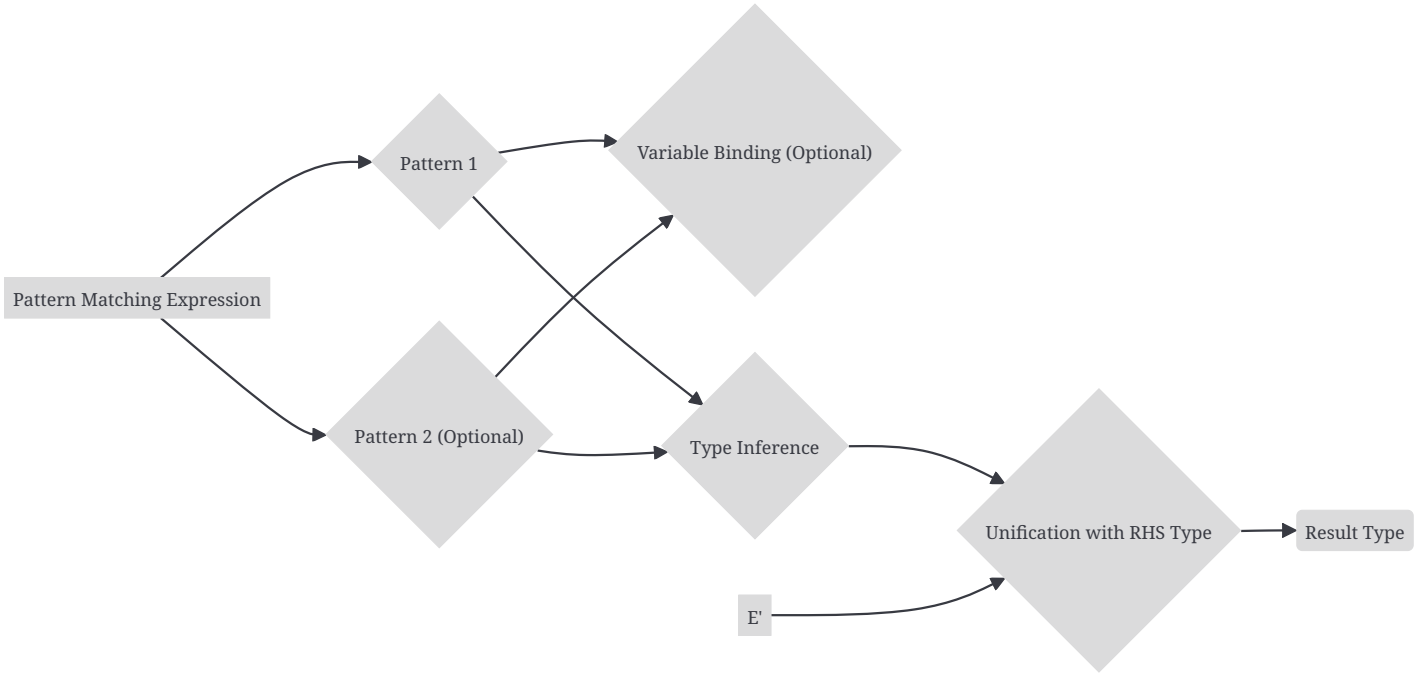
- Analyzes the structure of the data type, including its constructors and their associated field types (if any).
- Verifies that the types of fields within a constructor are valid and consistent.



4. Pattern Matching:
When encountering a pattern matching expression, the type checker performs a more intricate process:

- It analyzes each pattern within the match expression. Patterns can involve variables, constructors, or more complex combinations.

- For patterns that introduce new variables through bindings (e.g., `x <- expr`), the type checker assigns a type variable to the bound variable.
- The type checker infers a type for the entire pattern matching expression based on the types of the subexpressions used within the patterns.
- This inferred type is then unified with the type of the right-hand side (RHS) of the pattern matching expression. If unification fails, a type error is reported.
- **Crucially, the type checker ensures that all scrutinees (expressions being matched against) within the patterns have a compatible type.** This prevents nonsensical matches where data of different types is compared.
- Finally, based on the successful unification, the type checker determines the result type of the entire pattern matching expression.



5.5 Type Error Handling and Reporting

The robustness of a type system hinges not only on its ability to prevent errors but also on its capacity to effectively identify and report them. This section delves into tinyGHC's error handling mechanisms, exploring how the compiler detects type mismatches and generates informative error messages to guide the programmer towards a resolution.

5.5.1 Type Errors

During the type checking process, tinyGHC's type checker meticulously scrutinizes the program for any potential violations of type rules. Here's how type errors are identified:

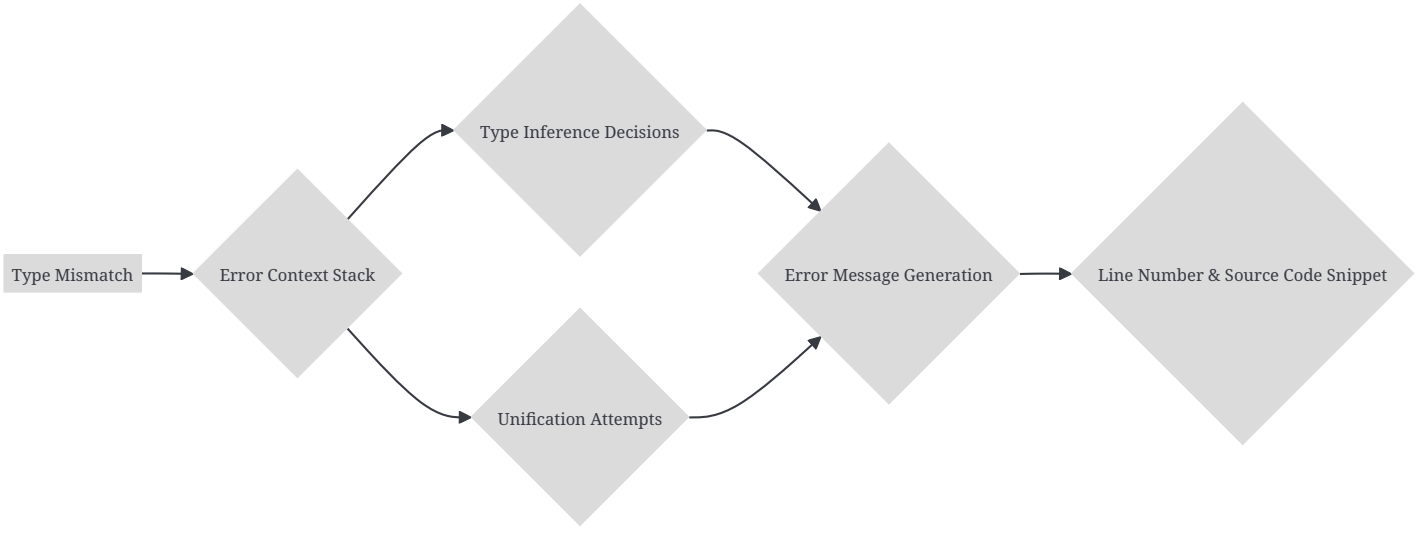
1. **Unification Failure:** A core mechanism for identifying type errors lies in unification. As the type checker attempts to unify types during various stages (e.g., function applications, pattern matching), unification might fail if the types involved are inherently incompatible (e.g., unifying an integer and a string). Such failures signify a type mismatch, triggering the error handling process.

2. **Constraint Violation:** Constraints, generated during type inference, encode requirements for expressions to be type-compatible. If the type checker encounters a constraint that cannot be satisfied, it indicates a type error. For instance, a constraint requiring an expression to be both an integer and a boolean simultaneously would be unsatisfiable, leading to an error.
3. **Rule Violations:** Specific type checking rules for different Haskell constructs (e.g., data type constructors, function definitions) might be violated during analysis. For instance, attempting to use a data constructor with the wrong number of arguments would constitute a rule violation and trigger an error.

5.5.2 Error Reporting

Once a type error is identified, tinyGHC's error reporting system swings into action. The goal is to provide the programmer with clear and concise information that pinpoints the location of the error and explains the nature of the type mismatch. Here's how tinyGHC achieves this:

1. **Error Location:** The error message explicitly identifies the line number and source code snippet where the error originated. This allows the programmer to readily locate the problematic section of code.
2. **Error Message:** The core of the error message delves into the nature of the type mismatch. It might mention the specific types involved and explain why they are incompatible. For instance, the message might state that a function expects an integer argument but was provided with a string.
3. **Error Context Stack:** A crucial aspect of tinyGHC's error reporting is the error context stack. This stack maintains a history of type-related decisions made during type checking, leading up to the point of the error. By utilizing the context stack, tinyGHC can provide more informative error messages that reflect the context in which the error arose.



Consider the following example:

```
head (map (+1) "hello")
```

Here, the type checker would encounter a type error when attempting to apply the `head` function to the result of `map (+1) "hello"`. The error message might look like this:

```
Error: Could not apply 'head' to an argument of type 'String'
In the expression: head (map (+1) "hello")
In the function application: map (+1) "hello"
The argument type 'String' does not match the expected type 'Int'
```

This message effectively leverages the error context stack. It not only pinpoints the location of the error (the `head` application) but also explains the type mismatch at each step:

- `map (+1) "hello"` tries to apply `map` (expects a list of `Int`s) to a string, resulting in a type mismatch.
- Consequently, the result of `map` has the type `String`, which is incompatible with the expected argument type `Int` of the `head` function.

By providing this level of detail within the error message, tinyGHC empowers the programmer to pinpoint the root cause of the error and make necessary corrections to the code.

Through meticulous error identification and informative reporting, tinyGHC's error handling system acts as a valuable tool for developers, fostering a more efficient and productive development experience in Haskell. The following section will delve into advanced topics like type classes and polymorphism, further highlighting the capabilities of tinyGHC's type system.

6. Spinless Tagless G-machine Intermediate Represntation

6.1. Introduction

One of the key challenges in compiling functional languages lies in their use of lazy evaluation, a paradigm where expressions are evaluated only when their results are strictly required. This approach contrasts with eager evaluation, where expressions are evaluated immediately upon encountering them in the code. While lazy evaluation offers advantages like improved memory management and reduced redundant computations, it necessitates specialized techniques for code generation.

In this context, compilers often employ intermediate representations (IRs) to bridge the gap between the high-level syntax of the source language and the low-level instructions of the target machine. An IR serves as an abstraction layer, allowing for optimizations and transformations independent of the specific details of the source and target languages. For functional languages with lazy evaluation, the chosen IR must effectively capture the semantics of the language while enabling efficient code generation for delayed computations.

This report delves into the Spineless Tagless G-machine (STG) IR, a crucial component of the tinyGHC compiler. TinyGHC is a compiler for a subset of the Haskell programming language known as tinyHaskell. TinyHaskell offers a rich set of features that showcase the power of functional programming, including:

- **Lazy evaluation:** As mentioned earlier, expressions in tinyHaskell are evaluated only when their results are needed, leading to potentially more efficient program execution.
- **Limited data types:** tinyHaskell supports three built-in data types - integers (Int), strings (String), and Booleans (Bool) - allowing for the study of core functional language concepts without the

complexity of user-defined data structures.

- **Conditional expressions (if):** TinyHaskell allows for conditional branching based on boolean expressions, enabling control flow within programs.
- **Where expressions:** This construct facilitates local variable bindings within expressions, promoting code readability and modularity.
- **Let expressions:** Similar to where expressions, let allows for variable binding, but additionally allows for function definitions within the scope of the expression, enabling powerful abstractions.
- **Function currying:** This technique allows functions to take their arguments one at a time, leading to more versatile function definitions.
- **Pattern matching:** A cornerstone of functional programming, pattern matching allows for elegant decomposition of data structures, facilitating concise and expressive code.
- **Lambda expressions:** These anonymous functions provide a powerful tool for creating concise and flexible code blocks within expressions.
- **Basic mathematical expressions:** Arithmetic and logical operations are essential for manipulating numerical and boolean data.

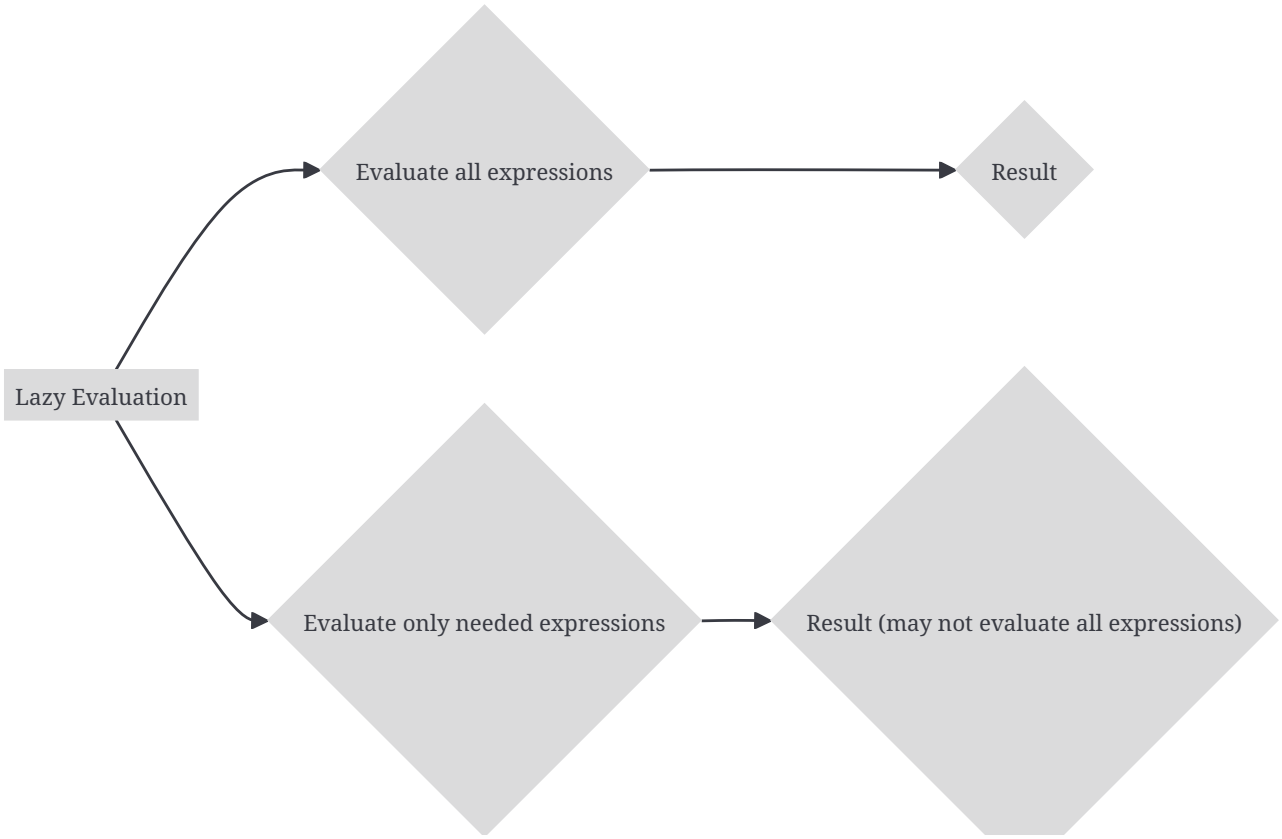
The STG IR in tinyGHC plays a pivotal role in representing these key features of tinyHaskell. The design of the STG IR specifically addresses the challenges of lazy evaluation by employing closures and function pointers to manage unevaluated expressions (thunks). This approach enables efficient code generation that can delay computations until their results are truly required. Additionally, the STG IR provides a clean and expressive representation for the various control flow constructs and data structures found in tinyHaskell.

By studying the STG IR, we gain a deeper understanding of how tinyGHC translates the high-level abstractions of tinyHaskell into a form suitable for efficient code generation. This report will delve into the details of the STG IR, its instructions, data structures, and the translation process from tinyHaskell expressions to STG code. Through this exploration, we will gain valuable insights into the challenges and techniques involved in compiling lazy functional languages.

6.2. Background: Lazy Evaluation and G-Machines

The realm of functional programming is characterized by a paradigm known as lazy evaluation. In contrast to "***eager evaluation***," where expressions are evaluated immediately upon encountering them in the code, lazy evaluation delays the evaluation of an expression until its result is strictly required. This approach offers several advantages for functional languages:

- **Improved memory management:** By deferring evaluation, lazy languages avoid creating data structures that might not be used in the final computation. This can lead to a more efficient use of memory, particularly when dealing with large or potentially unused data.
- **Reduced redundant computations:** Lazy evaluation ensures that expressions are evaluated only once, even if their results are used multiple times within a program. This can significantly improve performance by avoiding unnecessary work.



6.2.1 Understanding Lazy Evaluation

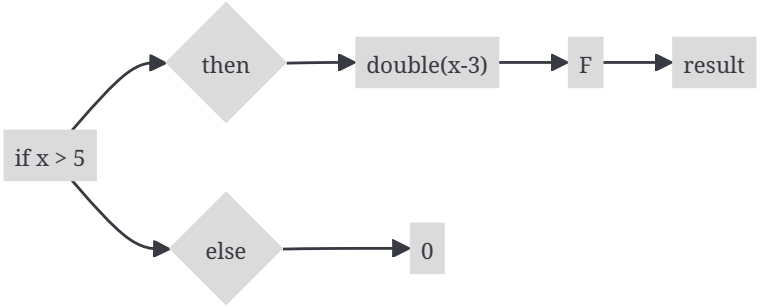
Consider the following example in a lazy functional language:

```
double x = x + x

result = if (x > 5) then double(x - 3) else 0
```

In an eager evaluation scheme, the expression `double(x - 3)` would be evaluated regardless of the value of `x` in the if statement. This could be wasteful if `x` is less than or equal to 5, as the result of `double(x - 3)` would never be used.

However, in a lazy evaluation setting, the expression `double(x - 3)` is not evaluated until the outcome of the `if` statement is determined. If `x` is indeed less than or equal to 5, the expression remains unevaluated, eliminating unnecessary computations.



This simplified graph illustrates a potential reduction sequence in a G-machine. The `if` expression (node A) acts as the initial redex. Depending on the value of `x`, the machine may follow the `then` or `else` branch. If the `then` branch is chosen (`x > 5`), the `double(x-3)` expression (node C) becomes the next redex. However, the machine may not immediately evaluate this expression (create a thunk) if the final result is not required (e.g., if `result` is never used).

6.2.2 G-Machines: A Framework for Lazy Evaluation

G-machines are a class of abstract machines specifically designed to facilitate the efficient execution of programs written in lazy functional languages. These machines operate on a graph-based representation of the program, where nodes represent expressions and edges denote dependencies between them.

A key concept in G-machines is the notion of a ***"thunk"***. A thunk is a data structure that encapsulates an unevaluated expression. When a thunk is encountered during program execution, the G-machine does not immediately evaluate the expression within the thunk. Instead, it performs a substitution operation. This substitution involves replacing the thunk with the result of evaluating the expression it holds, only if that result is actually needed for the final computation.

The specific design and implementation details of G-machines can vary. However, they typically share some common features:

- **Graph Reduction:** The G-machine operates by iteratively reducing the expression graph. During each reduction step, the machine identifies a *redex* (reducible expression), which is typically an application of a function to its arguments. The redex is then replaced with its corresponding value (normal form) if available, or with a thunk representing the unevaluated expression.
- **Sharing:** G-machines employ a sharing mechanism to avoid redundant computations. When multiple references to the same expression exist within the graph, the machine ensures that only one copy of the expression is evaluated. This optimization prevents unnecessary duplication of effort.

6.2.3 Limitations of Traditional G-Machines

While G-machines provide a powerful framework for lazy evaluation, they can suffer from certain limitations:

- **Complexity:** Traditional G-machine designs can be intricate, requiring careful management of the expression graph and handling of various edge cases. This complexity can make them challenging to implement and optimize effectively.
- **Overhead:** The overhead associated with maintaining the expression graph and managing thunks can potentially impact performance. Striking a balance between efficient evaluation and minimizing overhead is crucial for practical G-machine implementations.

The next section will delve into the Spineless Tagless G-machine (STG) IR, a specific design that addresses some of these limitations and offers a streamlined approach for representing lazy evaluation in tinyGHC.

6.3. Main Characteristics

The Spineless Tagless G-machine (STG) IR serves as a cornerstone of the tinyGHC compiler, providing an efficient and expressive representation for lazy evaluation in the context of tinyHaskell. This section delves into the core characteristics of the STG IR, highlighting its key features and how they address the challenges of lazy evaluation.

6.3.1 Core Characteristics

The STG IR is distinguished by two key design choices:

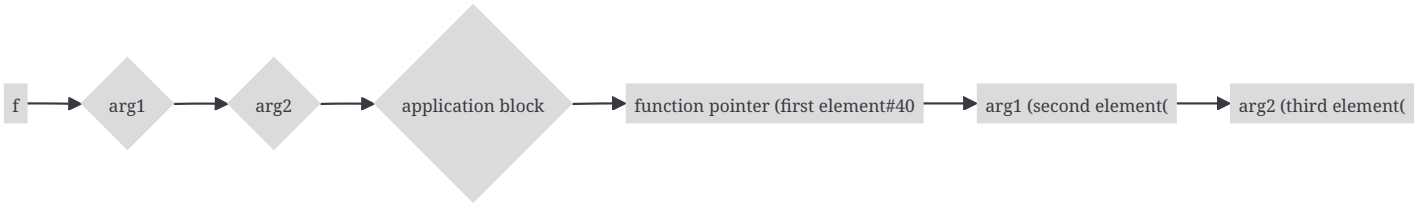
- **Spineless:** Traditional G-machine designs often utilize a "spine" data structure to represent function applications. The spine explicitly stores the function and its arguments in a linear

sequence. However, the STG IR departs from this approach. Instead, it employs a contiguous block of memory to represent a function application. The first element in this block points to the function itself, while subsequent elements hold the function's arguments. This eliminates the need for a separate spine data structure, simplifying the overall design.
Example (Function Application without Spine):

```
+----------+----------+----------+
| Function | Argument 1 | Argument 2 |
+----------+----------+----------+
```

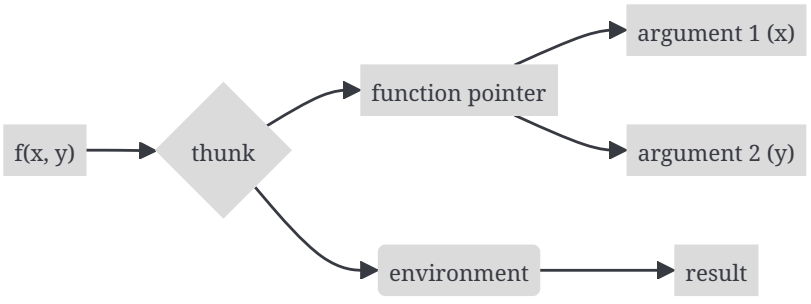
- **Tagless:** In some G-machine implementations, different types of expressions might be distinguished using tags. These tags allow the machine to identify the specific operation or data type associated with an expression. However, the STG IR adopts a tagless approach. Instead of relying on explicit tags, it leverages the function pointer stored in the first element of the application block to determine the appropriate operation to be performed. This simplifies the instruction set and reduces the overhead associated with tag decoding.

STG Example #1 for Function Application:



This graph illustrates the STG representation of a function application. Node A represents the function `f`, while nodes B and C denote its arguments `arg1` and `arg2`. Node D represents the application block, which stores the function pointer (`E`) in the first element, followed by the arguments (`F` and `G`) in subsequent elements.

STG Example #2 for Function Application:



This simplified graph illustrates how STG represents a function application. The initial expression `f(x, y)` is converted into a thunk (node B) containing the function pointer (node C), arguments (nodes D and E), and potentially an environment pointer (node F). When the thunk is evaluated, the function pointer is used to directly locate and execute the appropriate function code.

By combining closures and function pointers, the STG IR achieves a compact and efficient representation for lazy evaluation. Closures capture the necessary context for unevaluated expressions, while function pointers enable direct invocation of the corresponding code when

evaluation becomes necessary. This approach facilitates optimized code generation within the tinyGHC compiler, allowing it to effectively handle the lazy evaluation paradigm of tinyHaskell.

6.3.2 Achieving Efficient Evaluation with Closures and Function Pointers

The STG IR leverages two key mechanisms to achieve efficient lazy evaluation:

- **Closures:** Closures capture the environment (variable bindings) in which a function is defined. When a function application is encountered in the STG code, the function pointer in the application block references the appropriate closure. This closure contains the function's code and any necessary variable bindings from its defining environment. This approach enables access to the required variables during function execution, even if they are not explicitly included as arguments in the application itself.

Example (Closure for Function Application):

```
+----------+----------+----------+----------+
| Function | Argument 1 | Argument 2 | Environment |
+----------+----------+----------+----------+
```

- **Function Pointers:** As mentioned earlier, the STG IR utilizes function pointers within application blocks. These pointers directly reference the code for the function to be executed. When the G-machine encounters an application block, it retrieves the function pointer and uses it to jump to the corresponding function's code. This eliminates the need for complex interpretation or decoding of instructions, leading to efficient execution.

The combination of closures and function pointers in the STG IR offers several advantages:

- **Reduced Overhead:** By eliminating the need for a spine or tags, the STG IR minimizes memory overhead and simplifies instruction processing.
- **Efficient Caching:** Closures can be effectively cached, as they encapsulate frequently used function definitions and their associated environments. This caching mechanism can improve performance by avoiding redundant code generation.
- **Flexible Function Definitions:** Function pointers enable the creation of closures for functions defined within let expressions or as lambda expressions. This flexibility caters to the diverse function definitions supported in tinyHaskell.

By adopting these core characteristics, the STG IR provides a streamlined and efficient representation for lazy evaluation in tinyGHC. The next section will explore the specific instructions and data structures employed within the STG IR to represent the various features of tinyHaskell.

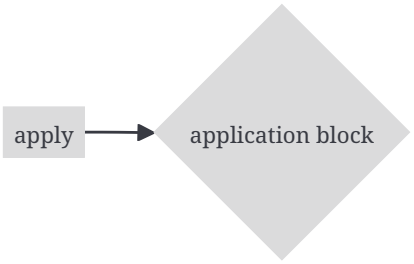
6.4 STG Instructions and Data

The STG IR utilizes a specific set of instructions and data structures to represent the various features and expressions found in tinyHaskell. This section delves into these core components, explaining their roles in facilitating efficient lazy evaluation.

6.4.1 STG Instructions

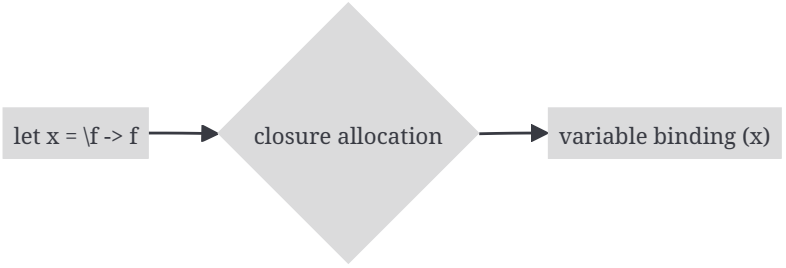
The STG instruction set provides a concise and expressive way to represent the computations within tinyHaskell programs. Here's a breakdown of the key instructions:

- **Function Application:** This instruction plays a pivotal role in STG. It operates on an application block, as described in Section 3.1. The instruction retrieves the function pointer from the block and uses it to jump to the corresponding function's code. Additionally, the instruction might handle argument passing based on the specific function definition.
  Example (Function Application):

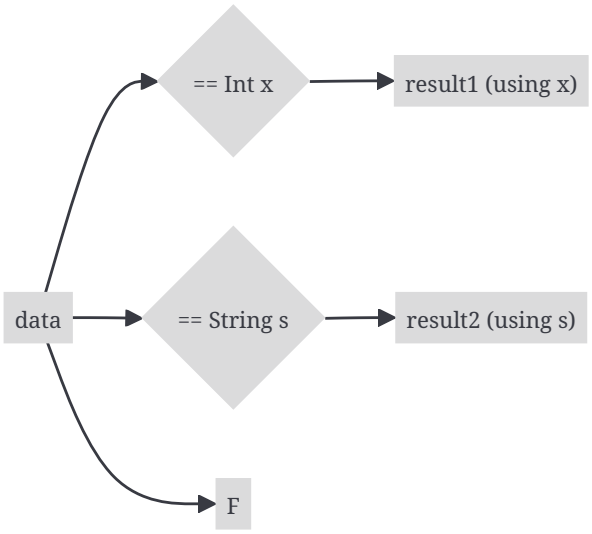


- **Tail Calls:** A special form of function application optimized for efficiency. In tail calls, the caller's frame can be deallocated immediately, as the callee does not require a return address. The STG IR employs a dedicated instruction to identify and handle tail calls for optimized code generation.

- **Let Expressions:** This instruction facilitates variable binding and closure creation. It allocates memory for a closure that encapsulates the function code and any referenced variables from the let expression's scope. Additionally, the instruction binds the variable name within the current environment to point to the newly created closure.
  Example (Let Expression):



- **Case Expressions (Pattern Matching):** This instruction enables the decomposition of data structures based on patterns. It takes a value and a list of clauses as arguments. Each clause specifies a pattern and the corresponding code to execute if the pattern matches the value. The STG IR employs a dedicated instruction to iterate through the clauses and perform the matching process.
  Example (Case Expression):



- **Primitive Operations:** The STG IR includes instructions for performing basic arithmetic (addition, subtraction, multiplication, etc.), logical (and, or, not), and string manipulation operations. These instructions directly map to the corresponding machine code operations of the target architecture.

- **Heap Allocation:** For data types beyond the built-in types (Int, String, Bool), the STG IR utilizes heap allocation instructions. These instructions allocate memory on the heap to store the data structure and its associated fields.

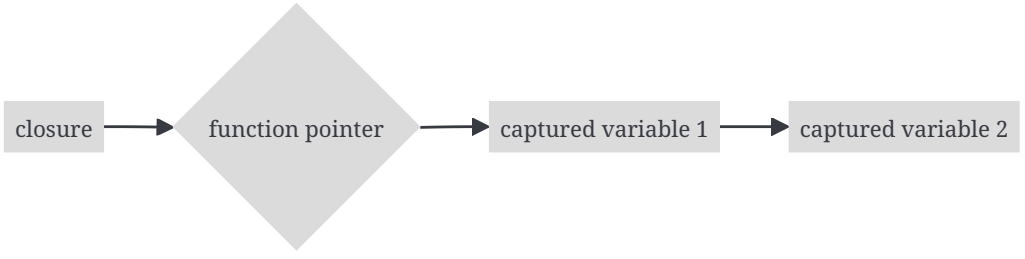
- **Control Flow Instructions:** The STG IR provides instructions for managing control flow within programs. These include jump instructions for non-local jumps, conditional jumps (if statements), and return instructions for function calls.

6.4.2 STG Data Structures

The STG IR relies on several core data structures to represent values and program state:

- **Constants:** These represent primitive data values like integers, strings, and booleans. They are stored directly in the STG code and do not require any additional allocation.
- **Variables:** Variables within the STG IR reference closures or constant values. A variable binding associates a variable name with a pointer to the corresponding data structure.
- **Closures:** As mentioned earlier, closures encapsulate a function's code and the environment (variable bindings) in which it was defined. A closure typically consists of a pointer to the function's code and a list of pointers to the captured variables.

Mermaid Graph for Closure:



By effectively combining these instructions and data structures, the STG IR provides a versatile and efficient representation for tinyHaskell programs. The next section will explore the translation process from tinyHaskell expressions to STG code, demonstrating how these components work together to achieve lazy evaluation.

6.5 Translation from tinyHaskell to STG

Bridging the gap between the high-level syntax of tinyHaskell and the efficient execution facilitated by the STG IR necessitates a robust translation process. This section delves into the key principles and translation rules employed to convert tinyHaskell expressions into their corresponding STG code.

6.5.1 General Translation Approach

The translation process from tinyHaskell to STG follows a recursive descent approach. The translator traverses the abstract syntax tree (AST) representing the tinyHaskell program. At each node in the AST, the translator applies specific rules based on the node's type (e.g., if expression, let expression, function call, etc.). These rules generate the corresponding STG instructions and data structures to accurately capture the semantics of the tinyHaskell expression.

6.5.2 Translation Rules for tinyHaskell Features to STG:

Here's a breakdown of the translation rules for various features supported by tinyHaskell:

- **If Expressions:**
  - The condition expression in the if statement is translated into STG code that evaluates the expression and stores the result (true or false) in a temporary variable.
  - Jump instructions are used to conditionally branch to the code blocks for the then and else clauses based on the value stored in the temporary variable.
  - The then and else clauses are translated recursively into their corresponding STG code.

    Haskell Example:

    ```
    if x > 5 then double x else 0
    ```

    Corresponding STG Representation:

    ```
    ; Pseudo-STG Assembly

    ; Assume gt5, double, and 0 are defined elsewhere

    ; Check if x > 5
    push x ; Push x onto the stack
    call gt5 ; Call the gt5 function

    ; Conditional branch
    jz FalseBranch ; Jump to FalseBranch if the result is False

    ; Double x
    push x ; Push x onto the stack again
    call double ; Call the double function
    jmp Return ; Jump to Return after execution

    FalseBranch:
    push 0 ; Push 0 onto the stack
    jmp Return ; Jump to Return after execution
    ```

```
Return:
; Pop the result off the stack and continue execution
```

- **Where Expressions:**
  - Variable bindings within the where expression are translated into memory allocation for the variables and assignment of the corresponding STG code for the initializing expressions.
  - The main expression of the where expression is then translated recursively, using the previously allocated memory locations for the bound variables.
    Haskell Example:

```
let y = 3
where x = y + 2
in x * 5
```

Corresponding STG Representation:

```
; Pseudo-STG Assembly

; Define y
alloc y ; Allocate memory for y
store y 3 ; Store the value 3 in y

; Calculate x
load y ; Load the value of y
add 2 ; Add 2 to y
store x ; Store the result in x

; Multiply x by 5
load x ; Load the value of x
mul 5 ; Multiply x by 5
store result ; Store the result in a variable named 'result'

; Continue execution with 'result'
```

Explanation:

1. Create a new environment record to store the binding for `y`.
2. Assign the value `3` to `y` within the new environment.
3. Evaluate `x = y + 2` using the new environment.
4. Evaluate `x * 5` using the same environment.

- **Let Expressions (including function definitions and currying):**
  - Let expressions involving variable bindings follow a similar approach as where expressions, allocating memory and assigning the results of the binding expressions.
  - When a function is defined within a let expression, the translator creates a closure. This closure encapsulates the function's code and references the environment record containing the bound variables from the let expression's scope.
  - Function currying in tinyHaskell translates to creating a closure that captures the arguments received so far and the remaining function body. Subsequent argument applications involve creating new closures based on the captured arguments and the remaining function.

- Similar to function definitions within let expressions, lambda expressions are translated into closures. The closure captures the lambda expression's code and references the environment record containing the variables in scope at the point of lambda definition.
  Haskell Code Example:

```
let double x = x * 2
in double 3
```

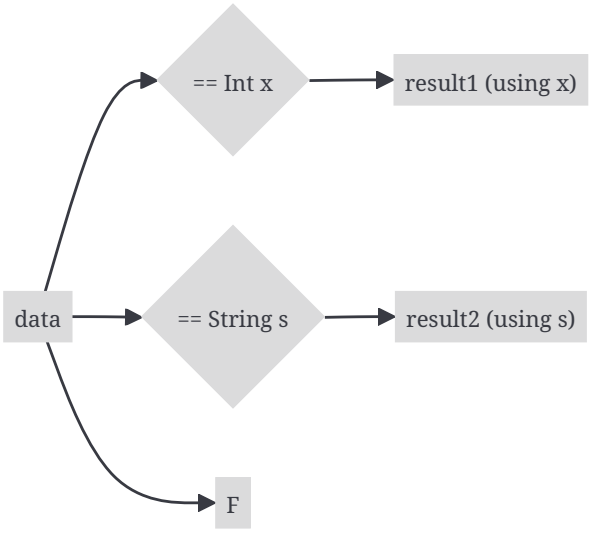
  Corresponding STG Representation:

```
; Pseudo-STG Assembly

; Define double
define double ; Define the double function
push 2 ; Push 2 onto the stack (for x * 2)
pop ; Pop the top of the stack (the result of x * 2)
ret ; Return the result

      ; Apply double to 3
    push 3 ; Push 3 onto the stack (the argument for double)
    call double ; Call the double function
    ```
```

- **Lambda Expressions:**
- **Algebraic Data Types and Pattern Matching:**
  - Data constructors for algebraic data types are translated into instructions for allocating memory on the heap and storing the appropriate tag value to identify the specific data variant.
  - Pattern matching expressions are translated into a series of jump instructions. Each pattern is converted into STG code that checks if the data structure matches the pattern. If a match is found, a jump directs the machine to the corresponding code block for handling that specific pattern variant.
    **Example (1) for Pattern Matching:**




    This simplified graph illustrates pattern matching on algebraic data. Node A represents the data value. Depending on the data type (Int or String), the machine jumps to the
```

corresponding pattern check (nodes B or D). If a match is found, a jump leads to the appropriate result code block (C or E).

Example (2) for ADTs:

- **Haskell Code:**

```haskell
data Shape = Circle Int | Square Int

area shape = case shape of
  Circle r -> pi * r * r
  Square s -> s * s
```

```
- **Corresponding STG Representation:**
  ```armasm
  ; Pseudo-STG Assembly

  ; Assume pi is defined elsewhere
  ; Calculate area of a shape
  push shape ; Push the shape onto the stack

  ; Test for Circle
  test Circle ; Test if the shape is a Circle
  jnz NotCircle ; Jump if not a Circle
  ; Calculate area of Circle
  push r ; Push radius r onto the stack
  mul r ; Multiply r by itself
  push pi ; Push pi onto the stack
  mul pi ; Multiply pi by the result of r * r
  ret ; Return the result
  NotCircle:
  ; Test for Square
  test Square ; Test if the shape is a Square
  jnz EndAreaCalculation ; Jump if not a Square
  ; Calculate area of Square
  push s ; Push side length s onto the stack
  mul s ; Multiply s by itself
  ret ; Return the result
  EndAreaCalculation:
  ; Handle other cases or error
```

- **Explanation**:
  1. Define STG instructions for constructors `Circle` and `Square` (not shown here).
  2. Translate the `case` expression:
     - Evaluate the expression `shape`.
     - Jump to the appropriate code block based on the constructor of `shape`:
       - `Circle r`: Evaluate `pi * r * r`.
       - `Square s`: Evaluate `s * s`.
```

- **Mathematical Expressions:**
  - Basic arithmetic, logical, and string manipulation expressions in tinyHaskell are translated into their equivalent STG instructions that directly map to the corresponding operations provided by the target machine (typically C).
    Example:
    - **Haskell**:

      ```
      3 + 4 * 5
      ```

  - **Corresponding STG Representation**:

    ```
    ; Pseudo-STG Assembly

    ; Perform multiplication
    push 4 ; Push 4 onto the stack
    push 5 ; Push 5 onto the stack
    call mul ; Call the multiplication operation

    ; Store the result of the multiplication temporarily
    push result ; Push the result of 4 * 5 onto the stack

    ; Perform addition
    push 3 ; Push 3 onto the stack
    call add ; Call the addition operation

    ; The final result is now on the top of the stack
    ```

By applying these translation rules recursively throughout the AST, the compiler generates the corresponding STG code that effectively captures the semantics of the tinyHaskell program.

7. C minus minus Intermediate Representation

7.1 Overview

C Minus Minus (Cmm) is an intermediate representation (IR) used in the tinyGHC compiler to translate Haskell code into C code. Cmm is a crucial component of the compiler pipeline, serving as a platform-independent, human-readable representation of the program that can be easily translated into machine-specific code.

- **Definition and Purpose**
  Cmm is a portable assembly language designed to facilitate the implementation of compilers that produce high-quality machine code. Its primary purpose is to provide a common intermediate representation that can be used as a target for various frontends and as a source for various backends. In the context of tinyGHC, Cmm is used as an intermediate representation between the Spinless Tagless G-machine (STG) IR and the final C code generation.

- **Design Philosophy**
  The design philosophy of Cmm is centered around the idea of providing a simple, platform-independent, and human-readable representation of the program. Cmm's syntax is inspired by C,

but it omits or changes certain features to make it more suitable for code generation and optimization. For example, Cmm does not support variadic functions, pointer syntax, or advanced type system features that can hinder essential features of Cmm, such as terminal recursion and ease of code generation.

- **Comparison to Other Intermediate Representations**
  Cmm is often compared to other intermediate representations, such as LLVM IR. While both Cmm and LLVM IR are designed to be platform-independent and human-readable, they differ in their expressiveness and design goals. LLVM IR is a more expressive IR, with a richer type system and more advanced features for optimization and code generation. Cmm, on the other hand, is designed to be a simpler, more lightweight IR that can be easily translated into machine-specific code. In the context of tinyGHC, Cmm is a more suitable choice than LLVM IR due to its simplicity and ease of use. Additionally, Cmm's design philosophy aligns well with the goals of tinyGHC, which aims to provide a lightweight and efficient compiler for a subset of Haskell.

- **Relationship to C--**
  Cmm is often referred to as the GHC implementation of the C-- language. C-- is a portable assembly language designed to ease the implementation of compilers that produce high-quality machine code. While C-- and Cmm share many similarities, they are not identical. Cmm is a specific dialect of C-- that is used in the GHC compiler, and it has undergone extensions and modifications to suit the needs of GHC.

In conclusion, Cmm is a crucial component of the tinyGHC compiler pipeline, providing a platform-independent, human-readable representation of the program that can be easily translated into machine-specific code. Its design philosophy and simplicity make it an attractive choice for tinyGHC, and its relationship to C-- and LLVM IR highlights its position as a unique and important intermediate representation in the compiler landscape.

7.2 Cmm Language Features—with examples

Cmm, the intermediate representation within the tinyGHC compiler pipeline, adopts a subset of C features to represent the core functionalities of tinyHaskell. This section delves into the fundamental building blocks of Cmm, encompassing data types, expressions, statements, and control flow constructs, providing illustrative C-- code examples alongside.

7.2.1 Data Types

Cmm inherits the simplicity of tinyHaskell by supporting a limited set of built-in data types:

- **Basic Types:**
  - **Integer (int):** Represents whole numbers for performing arithmetic operations.
  - **Boolean (bool):** Represents logical truth values (true or false) used in conditional statements.
  - **String (char):** Represents sequences of characters for storing textual data.

These basic types provide a foundation for constructing more complex data structures in tinyHaskell through features like algebraic data types (discussed later).

C-- Examples:

- **Integer:** `int x = 10;`

- **Boolean:** `bool isTrue = true;`
- **String:** `char* name = "Alice";`

7.2.2 Expressions

Expressions in Cmm form the core building blocks for computations and assignments. Cmm supports a variety of expressions inspired by C, including:

- **Arithmetic Expressions:** Utilize basic arithmetic operators (+, -, *, /) on integer operands.
- **Comparison Expressions:** Employ relational operators (==, !=, <, >, <=, >=) to compare integers or booleans.
- **Logical Expressions:** Combine boolean values using logical operators (&&, ||, !) for conditional logic.
- **Variable References:** Access the values stored in variables declared within the current scope.
- **Function Calls:** Invoke functions defined within the program, potentially passing arguments for evaluation.
- **Literal Expressions:** Represent constant values directly, including integers, booleans, and string literals enclosed in quotes.

Cmm adheres to operator precedence rules similar to C, ensuring proper evaluation order within complex expressions.

C-- Examples:

- **Arithmetic Expression:** `int result = x + 5;`
- **Comparison Expression:** `bool isEven = (x % 2) == 0;`
- **Logical Expression:** `if (isTrue && isEven) { ... }`
- **Variable Reference:** `int y = x;`
- **Function Call:** `int area = calculateArea(length, width);`
- **Literal Expression:** `int age = 25;`

7.2.3 Statements

Statements in Cmm dictate the program's flow of control and execution. Cmm incorporates essential control flow constructs and basic operations:

- **Variable Declarations:** Introduce new variables with a specific data type, allocating memory for them.
- **Assignment Statements:** Assign the result of an expression to a previously declared variable.
- **Compound Statements:** Group multiple statements within curly braces ({}), allowing for the creation of code blocks.
- **Conditional Statements:**
  - **if-else:** Executes a block of code based on a condition being true, with an optional else block for the alternative case.
- **Function Definitions:** Define functions with a return type, name, parameter list, and a code block representing the function body.

These statements enable the translation of fundamental control flow constructs and imperative programming features from tinyHaskell.

C-- Examples:

- **Variable Declaration:** `int counter;`
- **Assignment Statement:** `counter = counter + 1;`
- **Compound Statement:** `{ int a = 3; int b = 5; printf("%d\n", a + b); }`
- **if-else Statement:** `if (x > 0) { ... } else { ... }`
- **Function Definition:**

```
int add(int a, int b) {
  int sum = a + b;
  return sum;
}
```

7.2.4 Control Flow Constructs

Cmm provides essential control flow constructs to manage the execution sequence of the program:

- **if-else Statements:** Allow for branching based on a conditional expression. The if statement executes its associated code block if the condition is true, while the else block (if present) executes when the condition is false.
- **Function Calls:** Function calls transfer control to the invoked function's code block. Arguments can be passed to the function, and the return value can be used within the calling code.

While features like loops are not directly supported in tinyHaskell due to its focus on lazy evaluation, they can be implemented by translating recursive functions in tinyHaskell to iterative code within Cmm.

C-- Example (Recursive Function to Factorial):

```
int factorial(int n) {
  if (n == 0) {
    return 1;
  } else {
    return n * factorial(n - 1);
  }
}
```

Equivalent Iterative Code in Cmm:

```
int factorial(int n) {
  int result = 1;
  for (int i = 2; i <= n; ++i) {
    result *= i;
  }
  return result;
}
```

In conclusion, Cmm's language features offer a balance between simplicity and expressiveness. By supporting basic data types, expressions, statements, and control flow constructs, Cmm effectively represents the core functionalities of tinyHaskell, paving the way for a smooth translation process to C code in the final stage of the compilation pipeline.

7.3 Translation Process

Cmm, the intermediate representation (IR) within the tinyGHC compiler pipeline, plays a crucial role in bridging the gap between the high-level abstractions of tinyHaskell and the platform-specific details of C code. This section delves into the translation process from various tinyHaskell constructs to their corresponding Cmm representations. We will also explore the challenges associated with translating certain features, particularly those related to lazy evaluation.

7.3.1 Challenges of Translation

While tinyHaskell and C share some similarities, translating certain features presents unique challenges:

- **Lazy Evaluation:** Unlike C's eager evaluation, tinyHaskell employs lazy evaluation, where expressions are evaluated only when their results are needed. Cmm must represent this behavior without directly implementing lazy evaluation in C, which can be inefficient.

7.3.2 Translation Breakdown: tinyHaskell to Cmm

Here's a detailed breakdown of how various tinyHaskell constructs are translated into Cmm:

1. If Expressions:

- Conditional statements in tinyHaskell are translated to `if-else` statements in Cmm.
- The tinyHaskell condition is directly mapped to the Cmm condition expression.
- The `then` and `else` expressions in tinyHaskell correspond to the code blocks within the `if` and `else` branches of the Cmm statement.

Example:

```
if x > 0 then x * 2 else 0
```

Equivalent Cmm:

```
if (x > 0) {
  int result = x * 2;
} else {
  int result = 0;
}
```

2. Where Expressions:

- Pattern matching in tinyHaskell is translated into a series of `if-else` statements or switch statements (if applicable) in Cmm.
- Each pattern arm is evaluated within a separate `if` condition, checking if the value matches the pattern.
- The corresponding expression associated with the matching arm is then executed within the `if` block.

Example:

```
case color of
  Red -> "Stop"
  Green -> "Go"
  Blue -> "Wait"
```

Equivalent Cmm (using if-else):

```
if (color == Red) {
  char* message = "Stop";
} else if (color == Green) {
  char* message = "Go";
} else {
  char* message = "Wait";
}
```

3. Let Expressions:

- Variable bindings and scoping in tinyHaskell are directly translated into variable declarations within the corresponding Cmm code block.
- The scope of a variable in Cmm aligns with the block structure within which it is declared.

Example:

```
let x = 5
    y = x + 3
in x * y
```

Equivalent Cmm:

```
int x = 5;
int y = x + 3;
int result = x * y;
```

7.3. Lambda Expressions:

- Anonymous functions (lambda expressions) in tinyHaskell are translated into regular function definitions in Cmm.
- The lambda expression's arguments become the function's parameters in Cmm.
- The body of the lambda expression becomes the function's code block in Cmm.

Example:

```
(\x -> x * 2) 5
```

Equivalent Cmm:

```
int double(int x) {
  return x * 2;
}
```

```
int result = double(5);
```

5. Algebraic Data Types:

- User-defined data types in tinyHaskell are translated into struct definitions in Cmm.
- Each constructor variant of the data type becomes a separate field within the Cmm struct.
- Pattern matching on these data types can then leverage the struct fields for comparison.

Example:

```
data Shape = Circle Float
           | Rectangle Float Float
```

Equivalent Cmm (simplified):

```
struct Shape {
  float radius;  // For Circle constructor
  float width;
  float height;   // For Rectangle constructor
};
```

6. Mathematical Expressions:

- Arithmetic and logical operations in tinyHaskell are directly translated into their corresponding Cmm operators.
- tinyHaskell expressions involving built-in types (Int, Bool) are mapped to equivalent Cmm expressions with the same operators (+, -, *, /, ==, !=, &&, ||, etc.).

Example:

```
a + (b * c) && not d
```

Equivalent Cmm:

```
int result = a + (b * c)
```

7.3.3 Translation Breakdown: STG to Cmm Correspondence

While the previous section focused on tinyHaskell to Cmm translation, this section delves deeper, showcasing how the translation process might be reflected in the intermediate stages. Here, we provide examples of how tinyHaskell expressions might be represented in the Spinless Tagless G-machine (STG) IR and their corresponding Cmm code.

Note: The actual STG instructions may vary depending on the specific implementation of tinyGHC. This section provides a general illustration of the correspondence.

1. If Expressions:

STG (Example):

```
if_then_else (compare x 0 GT) (push 2) (push 0)
```

Explanation:

- This STG instruction performs a conditional branching based on the comparison of `x` with 0 using the `GT` (greater than) operator.
- If the condition is true (`GT`), the value `2` is pushed onto the STG stack.
- If the condition is false, the value `0` is pushed onto the stack.

Equivalent Cmm:

```
if (x > 0) {
  int result = 2;
} else {
  int result = 0;
}
```

2. Where Expressions:

STG (Example):

```
match color
  | Red -> push "Stop"
  | Green -> push "Go"
  | _ -> push "Wait"
```

Explanation:

- This STG instruction performs a pattern match on the value of `color`.
- It checks for three patterns: `Red`, `Green`, and a wildcard (`_`) for any other value.
- Based on the matching pattern, the corresponding string ("Stop", "Go", or "Wait") is pushed onto the STG stack.

Equivalent Cmm (using if-else):

```
if (color == Red) {
  char* message = "Stop";
} else if (color == Green) {
  char* message = "Go";
} else {
  char* message = "Wait";
}
```

3. Let Expressions:

STG (Conceptual):

- STG might introduce temporary stack frames or symbol tables to manage variable bindings within a `let` expression.
- The variable assignments (`x = 5`, `y = x + 3`) would be translated into STG instructions for value evaluation and storage.

Equivalent Cmm:

```
int x = 5;
int y = x + 3;
```

7.3. Lambda Expressions:

STG (Conceptual):

- STG might convert anonymous functions (lambda expressions) into closures, which encapsulate the function body and any captured variables.
- The STG representation would likely involve creating a closure object and pushing it onto the stack.

Equivalent Cmm:

```
int double(int x) {
  return x * 2;
}

int result = double(5);
```

5. Algebraic Data Types:

STG (Conceptual):

- STG might represent constructors of user-defined data types as specific tags or instructions.
- Pattern matching on these data types would involve checking the tags and accessing corresponding fields.

Equivalent Cmm (simplified):

```
struct Shape {
  // ... (as before)
};
```

6. Mathematical Expressions:

STG (Example):

```
push x
push 0
mult
push b
push c
mult
add
not d
and
```

Explanation:

- This STG sequence evaluates the expression `a + (b * c) && not d`.

- Individual operands (`x`, `0`, `b`, `c`, `d`) are pushed onto the stack.
- Arithmetic and logical operations (`mult`, `add`, `not`, `and`) are applied to the stack elements in the correct order.

Equivalent Cmm:

```
int result = a + (b * c) && not d;
```

8. Runtime Enviornment and Garbage Collection

Memory Management

The Garbage Collector (GC) uses a generational copying collection algorithm. The heap is divided into two generations: the young generation and the old generation. Newly allocated objects are placed in the young generation. When the young generation becomes full, a collection cycle is triggered. During a collection cycle, the GC scans the roots of the object graph (objects that are reachable from the registers and the stack) and copies all live objects from the young generation to the old generation. Any objects that are not copied are considered garbage and are reclaimed.

Data Structures

The GC uses a few data structures to manage the heap and track the liveness of objects.

- **Heap (`Heap`):** This structure represents the heap memory. It contains a pointer to the allocated memory (`data`), a pointer to the current write position (`cursor`), and the total capacity of the heap (`capacity`).
- **InfoTable:** An InfoTable is a data structure that is associated with each object in the heap. It contains information about the object, such as its type and its evacuation function.
- **StackA (`StackA`):** This structure represents the argument stack. It holds pointers to closures (`data`) for function arguments. It also maintains the base (`base`) and top (`top`) pointers to track stack usage.
- **StackB (`StackB`):** This structure represents the secondary stack. It stores various data types like integers (`int64_t`), continuation pointers (`CodeLabel`), and closure pointers (`uint8_t *`). Similar to `StackA`, it has `base`, `top`, and `data` pointers.
- **CAFCell (`CAFCell`):** This structure represents a Continuable Abstract Function (CAF) cell. It holds an info table pointer (`table`) for the closure information, a pointer to the closure itself (`closure`), and a pointer to the next CAF cell in the list (`next`).

Code

The GC code is implemented in several functions.

- `panic`: This function is used to print an error message and exit the program.
- `setup`: This function is called at the beginning of the program to initialize the heap and the stacks.
- `cleanup`: This function is called at the end of the program to free the memory that was allocated by the GC.
- `collect_garbage`: This function is called to trigger a garbage collection cycle.
- `heap_reserve`: This function is called to reserve a certain amount of bytes in the heap.

- `<closure>_evac`: These functions are the evacuation functions for different types of closures. They are responsible for moving a closure from the young generation to the old generation during a collection cycle.
- `<closure>_entry`: These functions are the entry points for different types of closures. They are responsible for calling the appropriate continuation or performing some other operation.

Explanation of the Code

- `panic` **function:** This function is a simple helper function that prints an error message to the standard error stream and then exits the program.
- `setup` **function:** This function is called at the beginning of the program to initialize the heap and the stacks. It allocates memory for the heap, the argument stack, and the secondary stack.
- `cleanup` **function:** This function is called at the end of the program to free the memory that was allocated by the GC. It frees the memory that was allocated for the heap, the argument stack, and the secondary stack.
- `collect_garbage` **function:** This function is called to trigger a garbage collection cycle. It first saves the state of the registers and the stacks. Then, it scans the roots of the object graph and copies all live objects from the young generation to the old generation. Finally, it reclaims any objects that are not copied.
- `heap_reserve` **function:** This function is called to reserve a certain amount of bytes in the heap. It checks if there is enough space in the heap to allocate the requested amount of memory. If not, it triggers a garbage collection cycle to collect any garbage and free up space in the heap.

Evacuation Functions

Each type of closure has a corresponding evacuation function. The evacuation function is responsible for copying a closure to a new location in the heap if necessary. It also needs to update any pointers within the closure that point to other objects.

- `static_evac`: This function does nothing, as static objects don't need to be moved during GC.
- `already_evac`: This function is used for closures that have already been evacuated. It simply returns the new location of the closure.
- `string_evac`: This function evacuates string closures. It copies the string data to a new location in the heap and updates the closure pointer.
- `partial_application_evac`: This function evacuates partial application closures. It copies the closure data and updates any pointers to arguments or the saved stack.
- `indirection_evac`: This function removes an indirection layer by updating the closure to point directly to its target.
- **Entry Functions:** Each closure type also has its own entry function. The entry function is called when the closure is invoked. The entry function typically performs the following steps:

    2. It retrieves the continuation that is stored in the closure.
    3. It calls the continuation.

Global Variables

The runtime system uses several global variables to store the GC state:

- `g_Heap`: This global variable holds the `Heap` structure.
- `g_SA`: This global variable holds the `StackA` structure for the argument stack.
- `g_SB`: This global variable holds the `StackB` structure for the secondary stack.

- `g_CAFListHead`: This is a pointer to the head of the CAF cell list.
- `g_CAFListLast`: This is a pointer to the last element in the CAF cell list.
- **Registers:** Several global variables are used as registers to hold values during function execution. These include:
  - `g_IntRegister` (int64_t): Holds integer return values.
  - `g_StringRegister` (uint8_t *): Holds a pointer to the string closure.
  - `g_TagRegister` (uint16_t): Holds the constructor tag for constructor closures.
  - `g_ConstructorArgCountRegister` (int64_t): Holds the number of arguments for a constructor closure.
  - `g_NodeRegister` (uint8_t *): Holds a pointer to the current closure.