Lecture Summary: Expression Trees as Algebraic Data Types

Generated by LectureMate

February 22, 2025

1 Introduction

This lecture focuses on expression trees as algebraic data types in Haskell. The discussion begins with an overview of syntax and semantics in programming languages, particularly in the context of Haskell.

2 Event Announcement

Before diving into the main content, the lecturer announces an upcoming event regarding exchange opportunities for students. This event will provide information on studying abroad during the third year of their program.

3 Algebraic Data Types

3.1 Definition

Algebraic data types (ADTs) allow the definition of types that can take on different forms. The lecturer previously introduced basic types such as booleans and lists, and now extends this to define the syntax of languages.

3.2 Syntax vs. Semantics

The distinction between syntax (the structure of expressions) and semantics (the meaning of expressions) is emphasized. Syntax refers to the rules governing the formation of valid expressions, while semantics pertains to the interpretation of these expressions.

4 Expression Trees

4.1 Simple Arithmetic Expressions

The lecturer introduces a simple algebraic data type called `Exp` for representing arithmetic expressions involving literals, addition, and multiplication. The constructors for this type are:

- `Lit` for literals (numbers)

- `Add` for addition
- `Mul` for multiplication

4.2 Example Expressions

Two example expressions are defined:

- $E_0 = \texttt{Add}(\texttt{Lit}(2), \texttt{Mul}(\texttt{Lit}(3), \texttt{Lit}(3)))$
- $E_1 = \texttt{Mul}(\texttt{Add}(\texttt{Lit}(2), \texttt{Lit}(3)), \texttt{Lit}(3))$

4.3 Expression Trees Visualization

The structure of these expressions can be visualized as trees, where:

- The root represents the main operation (either addition or multiplication).
- The leaves represent the literals.

5 Evaluation of Expressions

5.1 Evaluation Function

The lecturer presents a function to evaluate expressions defined by the `Exp` type. The evaluation function uses pattern matching to handle each constructor:

- For `Lit`, return the integer value.
- For `Add`, recursively evaluate both sub-expressions and return their sum.
- For `Mul`, recursively evaluate both sub-expressions and return their product.

5.2 Example Evaluations

The evaluation of E_0 results in 11 and E_1 results in 15.

6 String Representation of Expressions

6.1 Custom Show Function

A custom function is defined to convert expressions into a string representation. This function also uses pattern matching to handle each constructor and format the output appropriately.

6.2 Alternative Representations

The lecturer discusses alternative ways to represent expressions using infix notation and symbolic constructors, which can make the expressions look more like standard arithmetic.

7 Propositional Logic

7.1 Propositions as Algebraic Data Types

The lecture transitions to propositional logic, defining an algebraic data type for propositions that includes:

- `Var` for variables
- `False` and `True` for boolean constants
- `Not`, `Or`, and `And` for logical operations

7.2 Evaluation and String Representation

Similar to arithmetic expressions, functions are defined to evaluate propositions and convert them to string representations, following the structure of the data type definitions.

8 Conclusion

The lecture concludes with an invitation for questions and a reminder that the topic will continue in the next session. The importance of understanding algebraic data types in Haskell is reiterated, as they are foundational for functional programming.