Slide 1: Title Slide – The Hindley-Milner Type System

- **Context**: Introductory slide introducing Hindley-Milner (HM) and the key figures.
- **Class Explanation**:

  "Today, we're covering one of the most elegant and influential topics in modern programming language theory: the **Hindley-Milner Type System**. Named after J. Roger Hindley and Robin Milner, HM provides the formal mathematical foundation for type inference in functional languages like Haskell, OCaml, and ML. It lets us enjoy full, strong static typing without forcing developers to write explicit type annotations everywhere."

Slide 2: Introduction to HM & Lambda Calculus Basics

- **Context**: Connects HM to System F / Lambda Calculus and shows untyped syntax.
- **Class Explanation**:

  "Hindley-Milner is designed to infer types for the **polymorphic lambda calculus**. Before diving into types, let's refresh the basic syntax of untyped Lambda Calculus. Every expression e is built from just three constructs: **Variables** (x), **Abstractions** ($\lambda x.\, e$, which are anonymous functions), and **Applications** ($e_1\ e_2$, which represent calling a function with an argument)."

Slide 3: Simply-Typed Lambda Calculus ($\lambda^{\rightarrow}$)

- **Context**: Introduces STLC, parameter types, base types (B), and constants.
- **Class Explanation**:

  "If we want to add safety, we move to the **Simply-Typed Lambda Calculus** ($\lambda^{\rightarrow}$). Here, parameters require explicit type annotations, like $\lambda x{:}\,\tau.\, e$. We define a set of primitive base types B (such as integers or booleans) and build function types like $a \rightarrow (a \rightarrow b)$. We also need primitive constants c to perform concrete operations—since we aren't using pure Church encodings in practical systems."

Slide 4: Polymorphic Lambda Calculus Syntax

- **Context**: Removes explicit type annotations and introduces let bindings.
- **Class Explanation**:

  "Now, our goal in Hindley-Milner is to type expressions *without* forcing the programmer to write explicit annotations everywhere. Notice we've added a new syntax construct here: **let $x = e_1$ in e_2**. As we'll see shortly, let expressions are not just syntactic sugar in HM—they are fundamental to enabling polymorphism without making type inference undecidable."

Slide 5: What is a Type System?

- **Context**: Defines formal type systems and ill-typed terms.
- **Class Explanation**:

  "What actually *is* a type system? Formally, it's a deductive framework that decides whether a given expression is syntactically valid (well-typed) and determines its resulting type. For instance, if you take a function $(\lambda x{:}\,\alpha.\,x)$ expecting type α and try to apply it to a constant c_1 of incompatible type b, the type system rejects it as an **ill-typed** term."

Slide 6: Properties of Type Systems & Limits

- **Context**: Discusses Decidability, Soundness, Completeness, and Gödel's Incompleteness.
- **Class Explanation**:

  "When designing a type system, we evaluate three key mathematical properties:

  1. **Decidability**: Can an algorithm terminate in finite time with a yes/no answer on type correctness?
  2. **Soundness**: If the system proves an expression has type τ, it never produces a run-time type error ('Well-typed programs cannot go wrong').
  3. **Completeness**: Can the system prove types for *every* semantically safe program?

  Due to fundamental limits like Gödel's Incompleteness Theorem, no non-trivial type system can be completely sound and complete for all valid runtime behavior, so practical languages prioritize soundness and decidability."

Slide 7: Deductive Type Rules for $\lambda^{\rightarrow}$

- **Context**: Explains inference rules, contexts (Γ), turnstile ($\vdash$), and standard rules.
- **Class Explanation**:

  "We write type systems using formal inference rules structured as $\frac{\text{premises}}{\text{conclusion}}$.

  - Γ (Gamma) is our typing context—a dictionary of variable-to-type assumptions.
  - $\vdash$ (turnstile) means 'proves'.

  Rules (1)–(4) show how we look up variables in Γ, type constants, type function abstractions by assuming parameter types, and type function applications by matching the argument type to the domain."

Slide 8: Towards Hindley-Milner – The Need for Generic Types

- **Context**: Shows that standard $\lambda^{\rightarrow}$ forces functions like Identity to a single specific type.
- **Class Explanation**:

  "In Simply-Typed Lambda Calculus, the identity function $(\lambda x.\, x)$ must be assigned a fixed monomorphic type, like Int → Int or Bool → Bool. If you want an identity function for strings, you have to write a second identical function! To make functions truly reusable, we need a mechanism where $(\lambda x.\, x)$ can take on *many* types depending on context."

Slide 9: Monotypes vs. Polytypes (∀)

- **Context**: Introduces monotypes τ, universal quantifiers ∀, and polytypes σ.
- **Class Explanation**:

  "HM solves this by separating types into two levels:

  - **Monotypes (τ)**: Concrete types like integers, simple type variables α, or function types $\tau_1 \rightarrow \tau_2$.
  - **Polytypes / Type Schemes (σ)**: Types that contain universal quantifiers, written as $\forall \alpha.\, \sigma$.

  This allows the identity function to have the polymorphic scheme $\forall \alpha.\, \alpha \rightarrow \alpha$, meaning 'for *any* type α, this function takes an α and returns an α'."

Slide 10: The Problem with Reuse in Untyped Lambda Calculus

- **Context**: Demonstrates using identity with both integers and characters in a single term.
- **Class Explanation**:

  "Consider this term: applying `id` to `1` and also applying `id` to `'s'` in the same body. Under basic STLC, this fails because `id` cannot be both Int → Intand Char → Charat the same time. We need inference rules that allow a quantified type scheme to be generalized and instantiated on demand."

Slide 11: Generalization Rule [Gen]

- **Context**: Shows how to generalize a monotype variable to a universally quantified polytype.
- **Class Explanation**:

"To create polytypes, we introduce the **Generalization [Gen]** rule. If we derive that expression $e$has type σcontaining variable α, and $\alpha$is not free in our environment Γ, we can safely generalize it to $\forall\alpha.\sigma$. This turns a generic monotype into a fully polymorphic scheme."

Slide 12: Instantiation Rule [Inst]

- **Context**: Shows how to instantiate a polytype back into a specific monotype.
- **Class Explanation**:

  "Complementing generalization is the **Instantiation [Inst]** rule. When a polymorphic scheme like $\forall\alpha.\alpha \rightarrow \alpha$is referenced, we can replace the quantified variable $\alpha$with any concrete target monotype, such as Int → Int. Generalization packages types up; Instantiation unpacks them for specific calls."

Slide 13: Let-Polymorphism

- **Context**: Explains why `let` is required rather than standard lambda application $(\lambda x.e_2)\,e_1$.
- **Class Explanation**:

  "In standard lambda calculus, `let x = e1 in e2` is syntactically equivalent to $(\lambda x.e_2)\,e_1$. But in HM, they behave differently! Type inference for general System F (full polymorphic lambda calculus) is undecidable. HM solves this by restricting polymorphism specifically to `let` bindings. The `[Let]` rule types e_0, generalizes its type to a polytype scheme σ, and puts $x:\sigma$into the context while typing e_1."

Slide 14: Declarative Rules vs. Practical Algorithms

- **Context**: Differentiates semantic type rules from algorithmic type inference (Algorithm W/J).
- **Class Explanation**:

  "The 6 inference rules we've looked at define the **declarative semantics** of Hindley-Milner—they tell us *what* a valid typing proof looks like, but not *how* to find it programmatically. To implement this in a compiler, we use deterministic **typing algorithms** like **Algorithm W** or **Algorithm J**. These algorithms use unification (Robinson's Algorithm) to discover the principal (most general) type automatically."

Slide 15: Summary

- **Context**: Recaps formal rules, HM features, and real-world relevance.
- **Class Explanation**:

  "To wrap up:

  - Formal type systems use inference rules to build dependency proofs.
  - Hindley-Milner uses just 6 key rules to enable complete, automatic type inference with parametric polymorphism.
  - It balances strong theoretical safety with practical convenience, forming the foundation of modern functional compiler pipelines."

Slide 16: Practical Applications & Next Topics

- **Context**: Concludes the lecture and transitions to semantic analysis (Uranium framework).
- **Class Explanation**:

  "In practice, typing can be viewed as an attribute grammar or dependency graph across abstract syntax trees. Next time, we'll move from formal proof systems to practical compiler construction by exploring semantic analysis and attribute evaluation using the Uranium framework."