Type Checker: Design Document

Overview

This system implements a type checker for a statically typed language that performs type inference, validation, and error reporting. The key architectural challenge is building a constraint-based type inference engine that can unify type constraints while supporting polymorphism through let-generalization.

> This guide is meant to help you understand the big picture before diving into each milestone. Refer back to it whenever you need context on how components connect.

Context and Problem Statement

> **Milestone(s):** Milestone 1 (Type Representation), Milestone 2 (Basic Type Checking), Milestone 3 (Type Inference), Milestone 4 (Polymorphism)

Type checking serves as the foundational safety mechanism in statically typed programming languages, catching errors at compile time before they manifest as runtime failures. Building a type checker requires sophisticated algorithms that can not only validate explicit type annotations but also infer types where they are omitted, unify conflicting constraints, and support advanced features like polymorphism. Understanding why these algorithms are necessary—and how they work together—is crucial for implementing a robust type system that provides both safety and expressiveness.

Type Checking as Contract Verification

Think of type checking as a legal contract verification system in a complex business transaction. Just as lawyers review contracts to ensure all parties' obligations are clearly defined, mutually compatible, and legally enforceable, a type checker examines program code to verify that all data flows, function calls, and variable assignments respect the "contracts" defined by type annotations.

In this analogy, **types are contracts** that specify what operations are permissible on a piece of data. When you declare a variable as `int`, you're establishing a contract that this storage location will only hold integer values and support integer operations like arithmetic. When you define a function with signature `string -> int`, you're creating a contract promising to accept any string input and produce an integer output.

The **type checker acts as the contract lawyer**, examining every interaction between program components to ensure contract compliance. When it encounters a function call, it verifies that the arguments provided match the parameter contracts, just as a lawyer would verify that each party in a business deal can fulfill their

stated obligations. When it sees an assignment, it checks that the value being stored satisfies the variable's contract requirements.

But unlike simple contract verification, type checking must handle **implicit contracts**—situations where the programmer hasn't explicitly stated the types involved. This is where type inference becomes essential. Imagine a business lawyer who must not only verify explicit contract terms but also infer unstated obligations from context and industry standards. If a contract mentions "delivery within 30 days" but doesn't specify the delivery method, the lawyer might infer reasonable constraints based on the product type and industry norms.

Similarly, when a type checker encounters code like `let x = 42 + y`, it must infer that `x` has integer type based on the integer literal `42` and the addition operation, while also constraining `y` to be integer-compatible. This inference process generates **type constraints**—requirements that must be satisfied for the program to be well-typed.

The most sophisticated part of this analogy involves **contract templates**—polymorphic types that can be instantiated with specific terms for different situations. Just as a business might use template contracts where blanks are filled in based on the specific deal (vendor name, delivery date, payment amount), polymorphic functions use type variables that get instantiated with concrete types at each call site. The type checker must ensure that each instantiation of the template produces valid, consistent contracts.

> **Key Insight**: Type checking is fundamentally about proving that a program's implicit promises can be consistently fulfilled. Every expression makes promises about what type of value it produces, and every context makes demands about what types it can accept. The type checker's job is to verify that all promises match all demands throughout the program.

Existing Type System Approaches

Different programming languages have evolved distinct approaches to type checking, each representing different trade-offs between safety, expressiveness, implementation complexity, and programmer burden. Understanding these approaches helps clarify why our constraint-based inference system requires sophisticated algorithms.

Simple Type Checking Approach

The simplest approach requires **explicit type annotations everywhere** and performs **direct syntactic verification** without inference. Languages like early versions of Pascal or C follow this model, where every variable declaration must include a type annotation, and the type checker simply verifies that each operation's operand types exactly match the expected types.

Aspect	Simple Type Checking
Annotation Requirement	Explicit types required on all variables, parameters, return values
Inference Capability	None—types are only propagated, never inferred
Algorithm Complexity	Linear pass through AST checking local compatibility
Error Detection	Immediate—mismatch detected at each operation
Polymorphism Support	None, or limited to ad-hoc overloading
Implementation Effort	Low—straightforward recursive AST traversal

In this system, checking a function call involves three simple steps: look up the function's declared signature, check that each argument's type exactly matches the corresponding parameter type, and propagate the declared return type to the call site. There's no constraint solving, no unification, and no type variable instantiation.

The major limitation is **annotation burden**—programmers must write type annotations for every binding, even when the types are obvious from context. Code like `let length: int = 0` feels redundant when the integer literal makes the type clear. More importantly, this approach cannot support **parametric polymorphism**—generic functions that work with multiple types—because there's no mechanism for type variables or constraint solving.

Hindley-Milner Inference Approach

The **Hindley-Milner type system** represents the opposite extreme: maximal type inference with minimal annotations. Languages like ML, Haskell, and our target implementation use this approach, where most type annotations are optional and the system infers the most general types possible through constraint-based unification.

Aspect	Hindley-Milner Inference
Annotation Requirement	Optional except at module boundaries
Inference Capability	Complete—can infer most general types without annotations
Algorithm Complexity	Constraint generation + unification (roughly linear in practice)
Error Detection	Deferred—errors surface when constraints become unsatisfiable
Polymorphism Support	Full parametric polymorphism with let-generalization
Implementation Effort	High—requires constraint solving and unification algorithms

The Hindley-Milner approach works through **constraint generation** and **unification**. When the type checker encounters an expression like `f(x)`, it doesn't immediately verify compatibility. Instead, it generates

constraints: "the type of `f` must be a function type from the type of `x` to some result type." These constraints accumulate throughout the program, forming a system of type equations.

The **unification algorithm** then solves this constraint system by finding type substitutions that make all constraints simultaneously satisfiable. If unification succeeds, the program is well-typed and the algorithm has discovered the most general type for every expression. If unification fails, the constraints are inconsistent and the program contains a type error.

The key insight enabling this approach is **let-polymorphism**: when a value is bound in a let-expression, its inferred type is generalized by quantifying over any type variables that don't escape the binding's scope. This allows the same function to be used at multiple types within the same program.

> **Decision: Constraint-Based Inference with Hindley-Milner**
>
> - **Context**: We want to support both type safety and programmer convenience, with optional annotations and generic functions
> - **Options Considered**:
>   1. Simple explicit typing (like early Pascal)
>   2. Constraint-based inference (Hindley-Milner)
>   3. Gradual typing (mixed static/dynamic)
> - **Decision**: Implement constraint-based inference following the Hindley-Milner approach
> - **Rationale**: Provides maximum expressiveness with minimal annotation burden, supports full parametric polymorphism, and has well-understood theoretical foundations with proven decidability
> - **Consequences**: Requires implementing constraint generation, unification algorithm, and let-generalization, but provides excellent user experience and catches errors statically

Gradual Typing Approach

Gradual typing attempts to bridge static and dynamic typing by allowing programmers to add type annotations incrementally. Languages like TypeScript, Python with type hints, and Dart use this approach, where unannotated code is assigned a special "dynamic" type that bypasses static checking.

Aspect	Gradual Typing
Annotation Requirement	Optional, with dynamic fallback for unannotated code
Inference Capability	Limited—inference within typed regions, dynamic elsewhere
Algorithm Complexity	Multi-phase: static analysis where possible, runtime checks at boundaries
Error Detection	Mixed—static errors in typed code, runtime errors at dynamic boundaries
Polymorphism Support	Varies—full polymorphism in static regions, duck typing in dynamic regions
Implementation Effort	Very high—requires both static analysis and runtime type checking infrastructure

Gradual typing introduces **consistency checking** instead of strict type equality. A dynamically-typed value is considered "consistent" with any static type, allowing gradual migration from dynamic to static typing. However, this consistency must be verified at runtime through **type casts** and **proxy wrappers** that check contracts at the boundaries between typed and untyped code.

The complexity comes from maintaining **type soundness** in the presence of dynamic code. The static portions of the program must be protected from type violations that might originate in dynamic code, requiring sophisticated runtime monitoring and error reporting.

Approach Comparison and Selection

Approach	Safety Guarantees	Programmer Burden	Implementation Complexity	Expressiveness
Simple Explicit	Full static safety	High annotation burden	Low	Limited polymorphism
Hindley-Milner	Full static safety	Minimal annotations	Medium-High	Full polymorphism
Gradual	Partial safety	Optional annotations	Very High	Mixed paradigms

For our educational type checker implementation, the Hindley-Milner approach provides the best balance of learning value and practical utility. It requires understanding sophisticated algorithms (constraint generation, unification, generalization) while producing a system that feels modern and expressive to use.

The constraint-based inference approach also demonstrates important computer science concepts: the unification algorithm is a fundamental technique in automated reasoning, constraint solving appears throughout AI and optimization, and the type system's mathematical foundations connect to logic and formal methods.

> **Critical Understanding**: The need for sophisticated algorithms in type checking stems from the tension between expressiveness and verification. Simple type checking is easy to implement but forces programmers to state the obvious repeatedly. Powerful inference systems allow natural expression of polymorphic, generic code but require complex algorithms to ensure that the inferred types are sound and consistent.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
Core Data Structures	Basic algebraic data types with pattern matching	Extensible visitor pattern with type-safe dispatch
Constraint Solving	Direct recursive unification	Constraint propagation with backtracking
Error Reporting	Simple string messages	Rich error objects with source locations and suggestions
Type Environment	Single global symbol table	Nested scopes with efficient lookup chains
Testing Framework	Unit tests with manual test cases	Property-based testing with random program generation

Recommended File Structure

Our type checker implementation will be organized around the major algorithmic components, with clear separation between type representation, constraint handling, and the main inference engine:

OCAML

```
type-checker/
  lib/
    types/
      type_ast.ml          ← Core type representation (primitives, functions, variables)
      type_env.ml          ← Type environment and scoping management
      type_scheme.ml       ← Polymorphic type schemes with quantification
    inference/
      constraints.ml       ← Constraint generation from expressions
      unification.ml       ← Robinson unification algorithm with occurs check
      substitution.ml      ← Type variable substitution and composition
    checking/
      type_checker.ml      ← Main type checking orchestration
      error_reporter.ml    ← Type error formatting and source location tracking
    utils/
      fresh_vars.ml        ← Fresh type variable generation
      pretty_print.ml      ← Type and expression pretty printing
  test/
    unit/
      test_unification.ml  ← Unification algorithm test cases
      test_inference.ml    ← Type inference test scenarios
      test_polymorphism.ml ← Let-polymorphism and generalization tests
    integration/
      test_programs.ml     ← Complete programs with expected types/errors
  examples/
    simple_programs/       ← Example programs showing type system features
    error_examples/        ← Programs that should produce specific type errors
```

Infrastructure Starter Code

Fresh Variable Generation (complete utility module):

```ocaml
(* fresh_vars.ml - Type variable name generation *)

type var_id = int


let current_id = ref 0


let fresh_var_id () =

  let id = !current_id in

  current_id := id + 1;

  id


let fresh_type_var () =

  let id = fresh_var_id () in

  TVar ("t" ^ string_of_int id)


let reset_var_counter () =

  current_id := 0
```

Pretty Printing Infrastructure (complete utility for debugging):

```ocaml
(* pretty_print.ml - Type and expression formatting *)

let rec string_of_type = function

  | TInt -> "int"

  | TBool -> "bool"

  | TString -> "string"

  | TVar name -> name

  | TFun (param_ty, return_ty) ->

      let param_str = match param_ty with

        | TFun _ -> "(" ^ string_of_type param_ty ^ ")"

        | _ -> string_of_type param_ty

      in

      param_str ^ " -> " ^ string_of_type return_ty

  | TForall (vars, ty) ->

      "forall " ^ String.concat " " vars ^ ". " ^ string_of_type ty


let string_of_type_env env =

  let bindings = TypeEnv.fold (fun name ty acc ->

    (name ^ ": " ^ string_of_type ty) :: acc

  ) env [] in

  "{" ^ String.concat ", " bindings ^ "}"
```

Core Logic Skeletons

Type Representation (signatures to implement):

```ocaml
(* type_ast.ml - Core type system definitions *)

type ty =
  | TInt | TBool | TString                    (* Primitive types *)
  | TVar of string                            (* Type variables for inference *)
  | TFun of ty * ty                           (* Function types: param -> return *)
  | TForall of string list * ty               (* Polymorphic type schemes *)

(* Type equality checking - implement structural equality *)
let rec type_equal (t1 : ty) (t2 : ty) : bool =
  (* TODO 1: Handle primitive type equality (TInt = TInt, etc.) *)
  (* TODO 2: Handle type variable equality (same variable name) *)
  (* TODO 3: Handle function type equality (both param and return must match) *)
  (* TODO 4: Handle polymorphic type equality (same bound vars and body) *)
  failwith "implement type_equal"

(* Type variable occurrence check - prevents infinite types *)
let rec occurs_check (var_name : string) (ty : ty) : bool =
  (* TODO 1: Return true if ty is exactly the variable var_name *)
  (* TODO 2: For function types, recursively check both parameter and return *)
  (* TODO 3: For polymorphic types, check the body (bound vars shadow var_name) *)
  (* TODO 4: Return false for primitive types *)
  failwith "implement occurs_check"
```

Type Environment Management (signatures to implement):

```ocaml
(* type_env.ml - Symbol table with lexical scoping *)

module TypeEnv = struct

  type t = (string * ty) list list  (* Stack of scopes *)


  let empty : t = [[]]


  let extend (name : string) (ty : ty) (env : t) : t =

    (* TODO 1: Add (name, ty) binding to the top scope *)

    (* TODO 2: Shadow any existing binding for name in outer scopes *)

    failwith "implement extend"


  let lookup (name : string) (env : t) : ty option =

    (* TODO 1: Search through scopes from innermost to outermost *)

    (* TODO 2: Return the type of first matching binding found *)

    (* TODO 3: Return None if name is not bound in any scope *)

    failwith "implement lookup"


  let push_scope (env : t) : t =

    (* TODO: Add a new empty scope to the environment stack *)

    failwith "implement push_scope"


  let pop_scope (env : t) : t =

    (* TODO: Remove the innermost scope (for exiting let/function body) *)

    failwith "implement pop_scope"

end
```

Language-Specific Implementation Hints

OCaml-Specific Recommendations:

- Use **variant types** for the type AST—OCaml's pattern matching makes constraint generation and unification elegant

- Leverage **modules** to organize the type environment, constraint solver, and error reporter as separate interfaces
- Use **references** sparingly—only for fresh variable generation and mutable substitution composition during unification
- Take advantage of **tail recursion** for deep AST traversals in constraint generation
- Use **Result types** (`'a result = Ok of 'a | Error of 'b`) for error handling rather than exceptions during the learning phase

Pattern Matching Best Practices:

- Always include catch-all cases with `failwith` and descriptive error messages during development
- Use **nested patterns** for complex constraint generation: `TFun (TVar x, TVar y)` patterns can be handled specifically
- Leverage **guard clauses** (`when` conditions) for occurs checks and other constraints within patterns

Performance Considerations:

- Use **association lists** for type environments initially—optimize to maps later if needed
- Implement **path compression** in substitution composition only after basic unification works
- Consider **memoization** for expensive occurs checks, but implement the naive version first

Milestone Checkpoints

After Milestone 1 (Type Representation):

- Run `dune test lib/types` to verify basic type equality and occurs check
- Manually test type environment: create nested scopes, add bindings, verify lookup finds innermost binding
- Expected behavior: `lookup "x" (extend "x" TInt (extend "x" TBool empty))` should return `Some TInt`
- Common issue: Forgetting that `extend` should shadow, not error on duplicate names

After Milestone 2 (Basic Type Checking):

- Test explicit type annotations: `let x : int = 42` should type check successfully
- Test type mismatches: `let x : int = true` should produce a clear error with source location
- Verify function call checking: calling `(fun x : int -> x + 1)` with argument `"hello"` should fail
- Expected output: Error messages should pinpoint the exact location and explain what types were expected vs. found

Integration Verification: After completing both representation and basic checking, create a small test program:

```ocaml
(* test_basic.ml *)

let f (x : int) : int = x + 1

let result : int = f 42

let error_case : int = f true  (* Should fail type checking *)
```

Run your type checker on this program. It should successfully infer types for the first two bindings and produce a clear error for the third, showing that your type representation, environment management, and basic checking components work together correctly.

Goals and Non-Goals

> **Milestone(s):** Milestone 1 (Type Representation), Milestone 2 (Basic Type Checking), Milestone 3 (Type Inference), Milestone 4 (Polymorphism)

Building a type checker is like constructing a sophisticated quality assurance system for a manufacturing line. Just as a quality inspector must understand the specifications for each product, know how components fit together, and catch defects before they reach customers, our type checker must understand type specifications, verify that program components interact correctly, and catch type errors before runtime. However, unlike a simple inspection checklist, our type checker must also be a detective—inferring missing specifications and solving constraints to determine what types should be where no explicit annotations exist.

The scope of this type checker project strikes a careful balance between educational value and implementation complexity. We aim to build a system sophisticated enough to demonstrate the core principles of modern type systems while remaining focused enough that a single developer can implement it thoroughly. This means including the essential features that make type checking both challenging and useful, while deliberately excluding advanced features that would multiply implementation complexity without proportional learning benefit.

Primary Goals

Our type checker will accomplish four major objectives that progressively build upon each other to create a complete type inference system.

Foundational Type System Implementation forms the bedrock of our checker. We will implement a complete algebraic data type system for representing primitive types (`TInt` , `TBool` , `TString`), function types (`TFun`), type variables (`TVar`), and polymorphic types (`TForall`). The system will include structural type equality checking through `type_equal` , proper scoping through a hierarchical `TypeEnv.t` structure, and fundamental type compatibility rules. This foundation must be rock-solid because every other component depends on these type representations and environment operations.

Constraint-Based Type Inference Engine represents the intellectual heart of the project. We will implement a complete Hindley-Milner style inference system that can deduce types for expressions without explicit annotations. The system will generate type constraints from expressions, solve them through a Robinson unification algorithm with `occurs_check` protection against infinite types, and apply the resulting substitutions to produce fully concrete types. This goes far beyond simple type checking—it's a constraint solver that can work backwards from usage patterns to determine what types variables and expressions must have.

Let-Polymorphism Support enables our type checker to handle generic programming patterns that are essential in real programming languages. We will implement type scheme representation with quantified variables, automatic generalization at let-bindings that abstracts over free type variables, and instantiation at use sites that creates fresh type variables for each polymorphic usage. This allows a single function definition to work at multiple types—a critical feature for writing reusable code.

Comprehensive Error Reporting System ensures that our type checker is actually usable by developers. We will implement detailed error messages that include source locations, clear descriptions of what went wrong, and helpful suggestions for fixes. The system will continue checking after encountering errors to find multiple problems in a single pass, and it will provide meaningful diagnostics when constraint solving fails or unification encounters impossible requirements.

The following table summarizes our core deliverables and their acceptance criteria:

Component	Deliverable	Acceptance Criteria
Type Representation	Complete algebraic data type system	All primitive, function, and generic types representable; structural equality works correctly
Type Environment	Scoped symbol table with lookup/extension	Proper lexical scoping; variable shadowing handled correctly
Type Checking	Expression and statement type validation	All language constructs type-checked; operator compatibility enforced
Type Inference	Constraint generation and unification	Types inferred where not annotated; unification finds most general types
Polymorphism	Let-polymorphism with generalization	Single definitions usable at multiple types; proper quantification
Error Reporting	Clear messages with source locations	Multiple errors found per pass; helpful diagnostic information

Explicit Non-Goals

To maintain focus and achievable scope, we explicitly exclude several advanced type system features that would significantly complicate implementation without proportional educational benefit for understanding core

type checking principles.

Dependent Types and Advanced Type Theory Features are deliberately excluded. We will not implement dependent types where types can depend on runtime values, type-level computation, or sophisticated type families. While these features appear in languages like Agda, Idris, and recent Haskell extensions, they require substantially more complex type checking algorithms and would shift the focus from understanding basic inference to navigating advanced type theory. Our goal is to understand how types flow through programs, not to explore the cutting edge of type system research.

Effect Systems and Advanced Control Flow Analysis remain outside our scope. We will not track side effects, exceptions, or other computational effects in our type system. Similarly, we exclude advanced control flow features like continuations, coroutines, or sophisticated exception handling mechanisms. While effect systems are increasingly important in modern functional languages, they represent a separate layer of complexity that would double the implementation burden without reinforcing the core type inference concepts we want to master.

Object-Oriented Type Features and Inheritance are not included in this implementation. We will not build support for classes, inheritance hierarchies, method dispatch, or subtyping beyond simple function type compatibility. While these features are central to languages like Java or C++, they introduce architectural complexity around vtables, method resolution, and inheritance diamond problems that distract from the core algorithms of constraint-based type inference.

Advanced Polymorphism Beyond Let-Polymorphism is excluded to keep the scope manageable. We will not implement higher-ranked types, existential quantification, type classes, or GADTs (Generalized Algebraic Data Types). While these features enable powerful programming patterns, they require significantly more sophisticated constraint solving algorithms and would multiply the complexity of our unification engine beyond what's needed to understand the fundamental principles.

Production Language Features and Optimizations are consciously omitted. We will not implement module systems, separate compilation, incremental type checking, or performance optimizations like constraint caching or parallel inference. Similarly, we exclude features like type annotations for optimization hints, calling convention specifications, or memory layout controls. Our focus is on correctness and understanding, not on building a production-ready compiler infrastructure.

Gradual Typing and Dynamic Interaction features are excluded. We will not implement any form of gradual typing that allows mixing typed and untyped code, runtime type reflection, or dynamic type checking mechanisms. While these features are increasingly important for language interoperability, they introduce runtime components and dynamic analysis that are orthogonal to the static analysis principles we want to understand.

The following table clarifies what we explicitly will not implement:

Category	Excluded Features	Rationale
Advanced Types	Dependent types, type families, GADTs	Requires advanced type theory beyond core inference
Effects	Effect systems, exception tracking, continuations	Separate complexity domain from basic type checking
OOP Features	Classes, inheritance, method dispatch	Different architectural complexity than functional type inference
Advanced Polymorphism	Higher-ranked types, type classes, existentials	Would require sophisticated constraint solving extensions
Production Features	Modules, separate compilation, optimizations	Focus on correctness over performance and scale
Dynamic Interaction	Gradual typing, reflection, runtime types	Introduces runtime components beyond static analysis

Target Complexity and Learning Outcomes

Our type checker targets the complexity level of a sophisticated undergraduate compiler course final project or a graduate-level programming languages assignment. The implementation should require approximately 2000-3000 lines of well-structured code, with roughly equal complexity distributed across type representation, constraint generation, unification algorithm implementation, and polymorphism support.

Algorithm Implementation Depth will focus on understanding and implementing the core algorithms from first principles. Students will implement the Robinson unification algorithm with occurs check, understand why the occurs check prevents infinite types, and see how substitution composition works in practice. They will implement constraint generation that walks expression trees and collects type equality requirements, then solve those constraints to determine concrete types. This hands-on implementation of fundamental algorithms provides deep understanding that cannot be gained from just reading about them.

Type Theory Understanding will emerge naturally from implementing the system rather than being taught abstractly. Students will understand why let-polymorphism generalizes at let-bindings but not at lambda expressions, what the value restriction prevents, and how type schemes with quantified variables enable generic programming. They will see how type environments implement lexical scoping and why proper environment threading is crucial for correct type checking in nested expressions.

Software Architecture Skills will be developed through building a multi-component system where type representation, environment management, constraint solving, and error reporting must work together cleanly. Students will learn how to design algebraic data types that naturally represent the problem domain, how to structure recursive algorithms that traverse and transform complex data structures, and how to build error handling that provides useful feedback without crashing the entire system.

The learning progression follows a carefully designed sequence where each milestone builds essential concepts needed for the next stage, ensuring that students understand not just what to implement, but why each component is necessary and how the pieces fit together to enable type inference.

> **Key Insight**: The power of this project lies not in implementing any single algorithm perfectly, but in understanding how type representation, constraint generation, unification, and polymorphism work together as a complete system. Each component constrains and enables the others, creating a whole that demonstrates the elegant interplay between theory and practice in programming language implementation.

Success Metrics and Validation

Success in this project will be measured through both functional correctness and conceptual understanding, with specific checkpoints at each milestone to ensure steady progress toward a complete system.

Functional Correctness Metrics provide objective measures of implementation quality. The type checker must correctly infer types for polymorphic expressions like `let id = fun x -> x in (id 42, id true)`, producing types `(int * bool)` without any explicit annotations. It must detect type errors like `1 + true` and provide clear error messages explaining why integer addition cannot be applied to boolean operands. The unification algorithm must handle complex constraint sets and properly implement the occurs check to reject infinite types like `'a = 'a -> int`. Error recovery must continue checking after encountering problems to find multiple type errors in a single pass.

Conceptual Understanding Assessment focuses on whether students grasp the underlying principles rather than just copying implementation patterns. Students should be able to explain why let-polymorphism generalizes `let f = fun x -> x` to type `forall 'a. 'a -> 'a` but does not generalize the same expression when it appears as a function argument. They should understand why the occurs check prevents infinite types and be able to trace through the unification algorithm on specific examples. Most importantly, they should see how constraint-based inference enables type checking without explicit annotations while maintaining the safety guarantees of static typing.

Code Quality and Architecture Standards ensure that the implementation demonstrates good software engineering practices alongside algorithmic correctness. The code should exhibit clear separation between type representation, environment management, constraint generation, and unification solving. Error handling should be comprehensive without being defensive to the point of obscuring the main logic. The type representations should naturally reflect the mathematical structure of the type system, making the code serve as executable documentation of the algorithms.

The following table defines specific validation criteria for each milestone:

Milestone	Functional Test	Conceptual Check	Code Quality Standard
Type Representation	All primitive and composite types representable	Explain structural vs nominal typing choice	Clean algebraic data type definitions
Basic Type Checking	Expressions and statements type-checked correctly	Trace type rule applications by hand	Clear separation of checking vs inference
Type Inference	Types inferred for unannotated expressions	Explain constraint generation process	Unification algorithm cleanly separated
Polymorphism	Polymorphic functions work at multiple types	Explain generalization vs instantiation	Type scheme handling properly abstracted

Implementation Guidance

The implementation approach balances theoretical understanding with practical software construction, providing complete infrastructure components while ensuring students implement the core algorithms themselves.

A. Technology Recommendations

Component	Simple Option	Advanced Option
Type Representation	Simple algebraic data types with pattern matching	Extensible type system with visitor patterns
Environment	List-based scopes with linear lookup	Hash table with scope chain pointers
Constraint Solving	Direct recursive unification	Union-find with path compression
Error Reporting	Simple string messages with source positions	Structured error types with suggestions
Testing Infrastructure	Unit tests with hand-written examples	Property-based testing with random generation

B. Recommended File Structure

```
type_checker/
├── src/
│   ├── types.ml              ← Core type representations and utilities
│   ├── env.ml                ← Type environment implementation
│   ├── constraints.ml        ← Constraint generation from expressions
│   ├── unify.ml              ← Unification algorithm and substitutions
│   ├── infer.ml              ← Main type inference engine
│   ├── check.ml              ← Basic type checking without inference
│   ├── poly.ml               ← Polymorphism support (generalization/instantiation)
│   ├── errors.ml             ← Error reporting and recovery
│   └── pretty.ml             ← Pretty printing for types and diagnostics
├── test/
│   ├── test_types.ml         ← Type representation tests
│   ├── test_unify.ml         ← Unification algorithm tests
│   ├── test_infer.ml         ← End-to-end inference tests
│   └── examples/             ← Test programs for validation
└── bin/
    └── type_check.ml         ← Command-line interface
```

OCAML

C. Infrastructure Starter Code

Type Representation Foundation (Complete implementation):

OCAML

```ocaml
(* types.ml - Complete type representation system *)

type ty =

  | TInt

  | TBool

  | TString

  | TVar of string

  | TFun of ty * ty

  | TForall of string list * ty


type type_scheme = Forall of string list * ty


let var_counter = ref 0


let fresh_type_var () =

  incr var_counter;

  TVar ("_t" ^ string_of_int !var_counter)


let rec string_of_type = function

  | TInt -> "int"

  | TBool -> "bool"

  | TString -> "string"

  | TVar name -> "'" ^ name

  | TFun(t1, t2) ->

      let s1 = match t1 with TFun _ -> "(" ^ string_of_type t1 ^ ")" | _ -> string_of_type
t1 in

      s1 ^ " -> " ^ string_of_type t2

  | TForall(vars, t) ->

      "forall " ^ String.concat " " vars ^ ". " ^ string_of_type t


let rec type_equal t1 t2 =
```

```
  match t1, t2 with
  | TInt, TInt | TBool, TBool | TString, TString -> true
  | TVar v1, TVar v2 -> String.equal v1 v2
  | TFun(a1, b1), TFun(a2, b2) -> type_equal a1 a2 && type_equal b1 b2
  | TForall(vs1, t1), TForall(vs2, t2) ->
      List.length vs1 = List.length vs2 && type_equal t1 t2 (* simplified *)
  | _ -> false

let rec occurs_check var_name ty =
  match ty with
  | TVar name -> String.equal var_name name
  | TFun(t1, t2) -> occurs_check var_name t1 || occurs_check var_name t2
  | TForall(_, t) -> occurs_check var_name t
  | TInt | TBool | TString -> false
```

Type Environment Infrastructure (Complete implementation):

OCAML

```ocaml
(* env.ml - Complete type environment with scoping *)

module TypeEnv = struct

  type t = (string * ty) list list


  let empty = [[]]


  let extend name ty env =

    match env with

    | scope :: rest -> ((name, ty) :: scope) :: rest

    | [] -> [[(name, ty)]]


  let rec lookup name env =

    let rec search_scope scope =

      match scope with

      | [] -> None

      | (var, ty) :: rest when String.equal var name -> Some ty

      | _ :: rest -> search_scope rest

    in

    let rec search_scopes scopes =

      match scopes with

      | [] -> None

      | scope :: rest ->

          (match search_scope scope with

           | Some ty -> Some ty

           | None -> search_scopes rest)

    in

    search_scopes env


  let push_scope env = [] :: env
```

```
  let pop_scope env =
    match env with
    | [] -> []
    | _ :: rest -> rest
end
```

D. Core Logic Skeleton Code

Unification Algorithm (Signature with detailed TODOs):

OCAML

```ocaml
(* unify.ml - Core unification algorithm for students to implement *)

type substitution = (string * ty) list

let empty_subst = []

let apply_subst_to_type subst ty =
  (* TODO 1: Implement substitution application to types *)
  (* TODO 2: Handle TVar case - lookup in substitution, return replacement if found *)
  (* TODO 3: Handle TFun case - recursively apply to both parameter and return type *)
  (* TODO 4: Handle TForall case - apply to body type (be careful with variable capture) *)
  (* TODO 5: Handle primitive types - return unchanged *)
  failwith "implement apply_subst_to_type"

let compose_subst s1 s2 =
  (* TODO 1: Apply s1 to all type values in s2 *)
  (* TODO 2: Combine s2 with s1, ensuring s1 takes precedence for overlapping variables *)
  (* TODO 3: Return the composed substitution *)
  failwith "implement compose_subst"

let rec unify t1 t2 =
  (* TODO 1: Handle identical types - return empty substitution *)
  (* TODO 2: Handle TVar cases - check occurs_check, then bind variable *)
  (* TODO 3: Handle TFun cases - unify parameter types, then unify return types with composed substitution *)
  (* TODO 4: Handle primitive type mismatches - fail with unification error *)
  (* TODO 5: Handle TForall cases - instantiate with fresh variables, then unify *)
  (* Hint: Use occurs_check before binding variables to prevent infinite types *)
  (* Hint: Use compose_subst when unifying compound types like TFun *)
```

```
  failwith "implement unify"
```

Constraint Generation (Signature with detailed TODOs):

```ocaml
(* constraints.ml - Constraint generation from expressions *)

type constraint_t = ty * ty (* equality constraint between two types *)

let generate_constraints expr env =
  (* TODO 1: Pattern match on expression type *)
  (* TODO 2: For literals - return their known type, no constraints *)
  (* TODO 3: For variables - lookup in environment, return type *)
  (* TODO 4: For function calls - generate constraint that function type equals arg_type -> result_type *)
  (* TODO 5: For binary operators - generate constraints for operand types *)
  (* TODO 6: For function definitions - extend environment with parameter, generate constraints for body *)
  (* TODO 7: For let expressions - handle polymorphic generalization *)
  (* Hint: Return tuple of (inferred_type, constraint_list) *)
  failwith "implement generate_constraints"
```

E. Language-Specific Hints

OCaml Implementation Tips:

- Use pattern matching exhaustively with compiler warnings enabled (`-w +8`) to catch missing cases
- Implement `string_of_type` early for debugging - you'll need to inspect types constantly
- Use `failwith` with descriptive messages for unification failures, then refine to proper error handling
- The `ref` type works well for generating fresh type variable names with global counter
- List.fold_left is perfect for applying multiple substitutions in sequence

Testing and Debugging Strategies:

- Start with simple expressions like `42` and `true` before moving to function types
- Test unification algorithm independently with hand-constructed type pairs
- Use the REPL to test small functions interactively during development
- Print intermediate constraint sets to understand what the inference algorithm is trying to solve

- Implement `string_of_constraints` for debugging constraint generation

F. Milestone Checkpoints

Milestone 1 Checkpoint - Type Representation:

- Run: `ocaml -i types.ml` should show clean type signatures
- Test: Create values of each type variant, verify `string_of_type` output
- Verify: `type_equal (TFun(TInt, TBool)) (TFun(TInt, TBool))` returns `true`
- Check: `occurs_check "x" (TFun(TVar "x", TInt))` returns `true`

Milestone 2 Checkpoint - Basic Type Checking:

- Run: Compile and test basic expressions without type inference
- Test: `1 + 2` should type-check successfully as `int`
- Test: `1 + true` should produce a clear type error
- Verify: Function calls with wrong argument types are rejected

Milestone 3 Checkpoint - Type Inference:

- Run: `let id = fun x -> x in id 42` should infer type `int`
- Test: Unification algorithm handles complex constraint sets
- Verify: `occurs_check` prevents infinite types during unification
- Check: Multiple constraints from single expression are solved correctly

Milestone 4 Checkpoint - Polymorphism:

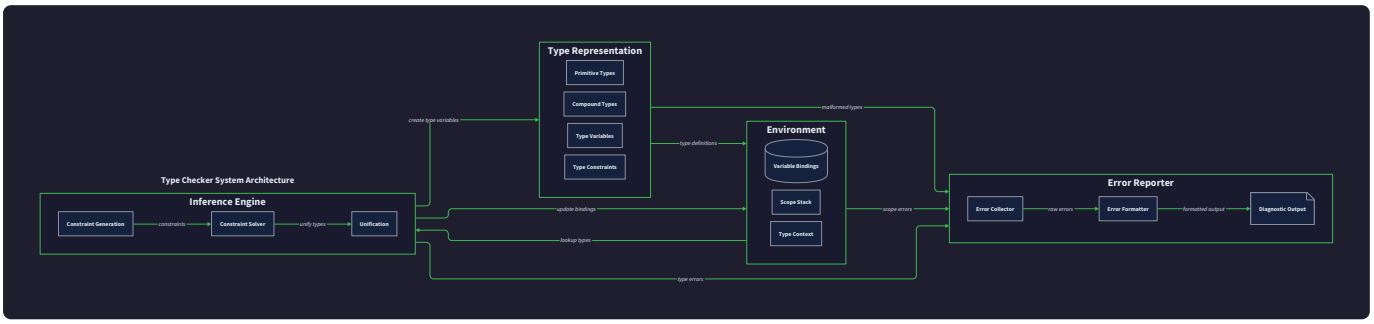
- Run: `let id = fun x -> x in (id 42, id true)` should infer `int * bool`
- Test: Same polymorphic function works at different types
- Verify: Let-polymorphism generalizes at let-bindings but not lambda-expressions
- Check: Type schemes are instantiated with fresh variables at each use site

High-Level Architecture

> **Milestone(s):** Milestone 1 (Type Representation), Milestone 2 (Basic Type Checking), Milestone 3 (Type Inference), Milestone 4 (Polymorphism)

Think of a type checker as a sophisticated contract verification system for a large corporation. Just as a legal department has specialized teams for contract review (lawyers), document management (paralegals), dispute resolution (mediators), and reporting (clerks), our type checker divides the complex task of type verification into specialized components. Each component has a clear responsibility, well-defined interfaces, and specific expertise, but they must work together seamlessly to transform raw source code into a verified, type-safe program.

The type checker architecture follows a **pipeline model** where each stage adds more semantic understanding to the program representation. We start with an Abstract Syntax Tree (AST) that captures the structural syntax of the program, then progressively annotate it with type information, resolve type constraints, and finally produce either a fully type-checked AST or comprehensive error reports. This pipeline approach allows us to separate concerns cleanly while maintaining the ability to recover from errors and continue checking to find multiple issues in a single pass.



The architecture is built around four core principles that guide component design and interaction patterns. **Separation of concerns** ensures that type representation logic is independent of inference algorithms, which are independent of error reporting mechanisms. **Composability** allows us to combine basic type checking with sophisticated inference without modifying existing components. **Error resilience** enables the system to continue checking after encountering type errors, maximizing the feedback provided to developers. **Extensibility** supports adding new type constructs (like algebraic data types) without requiring fundamental architectural changes.

> **Key Insight**: The type checker's architecture mirrors the conceptual structure of type theory itself. Type representations correspond to the syntax of types, the type environment models variable binding and scoping rules, the inference engine implements type rules and constraint solving, and the error reporter translates formal type theory violations into human-readable feedback.

Component Responsibilities

The type checker system consists of four primary components, each with distinct responsibilities and clear boundaries. Understanding these boundaries is crucial because violations lead to tightly coupled code that becomes difficult to extend and debug.

Type Representation System

The **Type Representation** component serves as the foundation layer, defining the abstract syntax and semantics of types themselves. Think of this component as the "type theory textbook" of our system—it contains all the formal definitions of what constitutes a valid type, how types relate to each other, and the fundamental operations we can perform on types.

Responsibility	Description	Key Operations
Type Definition	Algebraic data type defining the universe of possible types	`TInt`, `TBool`, `TString`, `TVar`, `TFun`, `TForall` constructors
Type Equality	Structural comparison of types for compatibility checking	`type_equal(t1, t2)` determines if two types are identical
Type Manipulation	Operations for constructing, deconstructing, and transforming types	`fresh_type_var()`, `string_of_type()`, substitution application
Subtyping Rules	Formal rules defining when one type can be used in place of another	Coercion rules for function types and polymorphic instantiation

This component deliberately avoids any knowledge of source code syntax, variable environments, or inference algorithms. It operates purely on type values and provides the primitive operations that higher-level components use to reason about type relationships. The type representation must be **immutable** to prevent subtle bugs during constraint solving, where the same type might be referenced from multiple constraints.

> **Decision: Algebraic Data Types for Type Representation**
>
> - **Context**: Need to represent the variety of types (primitives, functions, variables, polymorphic types) in a way that supports pattern matching and ensures exhaustive case handling
> - **Options Considered**:
>   1. Object-oriented hierarchy with virtual methods
>   2. Union types with tagged discriminants
>   3. Algebraic data types with pattern matching
> - **Decision**: Algebraic data types (`ty` type with constructors)
> - **Rationale**: Pattern matching ensures exhaustive handling of type cases, prevents null pointer errors, and makes the type structure explicit. OCaml's type system guarantees we handle all type constructor cases.
> - **Consequences**: Enables reliable constraint generation and unification algorithms, but requires functional programming comfort from implementers.

Type Environment Manager

The **Type Environment** component functions as the "symbol table with scoping awareness" for our type checker. Just as a compiler's symbol table tracks variable declarations across nested scopes, the type environment maintains the mapping from variable names to their types, handling lexical scoping rules correctly.

Responsibility	Description	Implementation Strategy
Variable Binding	Associates variable names with their declared or inferred types	Stack of scope frames, each containing name-to-type mappings
Scope Management	Tracks entry/exit from lexical scopes (functions, blocks, let bindings)	Push/pop operations on scope stack during AST traversal
Name Resolution	Looks up variable types following lexical scoping rules	Search scopes from innermost to outermost until name found
Shadowing Handling	Correctly handles variable shadowing in nested scopes	Inner scope bindings hide outer scope bindings with same name

The environment uses a **stack of frames** approach where each frame represents a single lexical scope. When entering a new scope (function body, let binding, block), we push a new frame. When exiting, we pop the frame, automatically restoring the previous scope's bindings. This approach naturally handles variable shadowing—inner scope variables with the same name as outer scope variables correctly hide the outer bindings.

Environment Operation	Input Parameters	Return Value	Side Effects
`TypeEnv.extend`	`name: string`, `ty: ty`, `env: TypeEnv.t`	Updated environment	Adds binding to innermost scope frame
`TypeEnv.lookup`	`name: string`, `env: TypeEnv.t`	`ty option`	None (pure function)
`TypeEnv.push_scope`	`env: TypeEnv.t`	Updated environment	Creates new empty scope frame
`TypeEnv.pop_scope`	`env: TypeEnv.t`	Updated environment	Removes innermost scope frame

Decision: Immutable Environment with Scope Stack

- **Context**: Need to track variable types across nested scopes while supporting backtracking during constraint generation
- **Options Considered**:
  1. Mutable hashtable with manual scope tracking
  2. Immutable environment with scope stack
  3. Persistent data structure with path copying
- **Decision**: Immutable environment with scope stack (`(string * ty) list list`)
- **Rationale**: Immutability prevents bugs during backtracking and parallel constraint generation. Stack structure directly models lexical scoping semantics.
- **Consequences**: Thread-safe and backtrack-friendly, but requires careful memory management for deeply nested scopes.

Type Inference Engine

The **Type Inference Engine** represents the "mathematical reasoning core" of our system. Think of it as a theorem prover that takes type constraints (equations like "the type of `x + 1` must equal the type of `x` unified with `TInt`") and finds consistent solutions. This component implements the heart of the Hindley-Milner type inference algorithm.

Inference Phase	Input	Output	Algorithm
Constraint Generation	Expression AST + Type Environment	Type variable + Constraint list	Recursive traversal generating equality constraints
Constraint Solving	Constraint list	Substitution or Error	Robinson unification algorithm with occurs check
Substitution Application	Substitution + Type	Concrete Type	Replace type variables with solved types
Type Generalization	Inferred type + Environment	Type scheme	Abstract over free type variables (let-polymorphism)

The inference engine operates in two distinct phases that must remain separate for correctness. **Constraint generation** walks the expression AST and collects type equality requirements without attempting to solve them. This separation is crucial because constraint generation may encounter expressions whose types depend on solving constraints from other expressions. **Constraint solving** then takes the complete set of constraints and finds a substitution (mapping from type variables to concrete types) that satisfies all constraints simultaneously.

Core Algorithm	Input Types	Return Type	Failure Conditions
`generate_constraints`	`expr`, `TypeEnv.t`	`ty * constraint_t list`	Undefined variable reference
`unify`	`ty`, `ty`	`substitution`	Occurs check failure, incompatible types
`apply_subst_to_type`	`substitution`, `ty`	`ty`	Never fails (total function)
`compose_subst`	`substitution`, `substitution`	`substitution`	Never fails (total function)

The **unification algorithm** lies at the heart of constraint solving. When asked to unify two types `t1` and `t2`, it returns a substitution `s` such that applying `s` to both `t1` and `t2` produces identical types. The algorithm handles several cases: identical types unify with the empty substitution, type variables unify by binding the variable to the other type (after occurs check), and structured types (like functions) unify by recursively unifying their components.

> **Critical Design Insight**: The inference engine must be **deterministic** and **complete**. Deterministic means that given the same constraints, it always produces the same substitution (modulo variable renaming). Complete means that if a solution exists, the algorithm finds it. These properties ensure that type inference behaves predictably and doesn't miss valid programs.

Error Reporting System

The **Error Reporter** serves as the "translator" between formal type theory violations and human-readable feedback. This component faces a unique challenge: it must present complex type mismatches, unification failures, and constraint violations in terms that make sense to programmers who may not understand type theory.

Error Category	Detection Point	Information Required	Example Message
Type Mismatch	Basic type checking	Expected type, actual type, source location	"Expected `int` but got `string` at line 15"
Unification Failure	Constraint solving	Conflicting constraints, expression context	"Cannot unify `int -> bool` with `string` in function call"
Occurs Check	Variable unification	Type variable, infinite type being constructed	"Cannot construct infinite type: `a = a -> int`"
Undefined Variable	Environment lookup	Variable name, available names in scope	"Undefined variable `x`. Did you mean `y`?"

The error reporter must maintain **source location information** throughout the type checking process. This requires threading location data from the original AST through constraint generation and into error messages. The reporter also implements **error recovery** strategies that allow type checking to continue after encountering errors, maximizing the number of issues found in a single pass.

Error Recovery Strategy	When Applied	Recovery Mechanism	Benefits
Type Variable Insertion	Unification failure	Replace failed type with fresh variable	Allows checking to continue with unknown type
Environment Extension	Undefined variable	Add variable with fresh type variable	Prevents cascading undefined variable errors
Constraint Skipping	Occurs check failure	Skip constraints involving infinite types	Continues with remaining solvable constraints
Default Type Assignment	Missing type annotation	Assign reasonable default (e.g., `int` for numbers)	Provides feedback on missing annotations

Decision: Structured Error Types with Source Locations

- **Context**: Need to provide helpful error messages that guide developers to fix type issues quickly
- **Options Considered**:
  1. String-based error messages with manual formatting
  2. Structured error types with source locations and contextual information
  3. Exception-based error handling with stack unwinding
- **Decision**: Structured error types containing location, expected/actual types, and context
- **Rationale**: Structured errors enable consistent formatting, IDE integration, and programmatic error analysis. Source locations are essential for developer productivity.
- **Consequences**: Requires careful location tracking throughout pipeline, but enables high-quality error reporting and tooling integration.

Recommended File Structure

The module organization reflects our architectural principles by creating clear boundaries between components and establishing a logical dependency hierarchy. Each module has a single, well-defined responsibility and minimal dependencies on other modules.

```
type-checker/
├── src/
│   ├── types/
│   │   ├── types.ml                ← Core type definitions and constructors
│   │   ├── type_ops.ml             ← Type manipulation operations
│   │   ├── substitution.ml         ← Substitution operations and composition
│   │   └── types.mli               ← Public interface for type system
│   │
│   ├── environment/
│   │   ├── type_env.ml             ← Type environment implementation
│   │   ├── scoping.ml              ← Scope management utilities
│   │   └── type_env.mli            ← Environment interface
│   │
│   ├── inference/
│   │   ├── constraints.ml          ← Constraint generation from expressions
│   │   ├── unification.ml          ← Robinson unification algorithm
│   │   ├── inference_engine.ml     ← Main inference coordination
│   │   └── inference.mli           ← Inference engine interface
│   │
│   ├── checking/
│   │   ├── basic_checker.ml        ← Simple type checking without inference
│   │   ├── expr_checker.ml         ← Expression type rules
│   │   ├── stmt_checker.ml         ← Statement type checking
│   │   └── checker.mli             ← Type checker interface
│   │
│   ├── polymorphism/
│   │   ├── type_schemes.ml         ← Type scheme representation and operations
│   │   ├── generalization.ml       ← Let-polymorphism generalization
│   │   ├── instantiation.ml        ← Type scheme instantiation
│   │   └── polymorphism.mli        ← Polymorphism interface
│   │
│   ├── errors/
│   │   ├── type_errors.ml          ← Error type definitions
│   │   ├── error_reporter.ml       ← Error message generation
│   │   ├── source_locations.ml     ← Source position tracking
│   │   └── errors.mli              ← Error handling interface
│   │
│   └── type_checker.ml             ← Main entry point and pipeline coordination
│
├── test/
│   ├── unit/
│   │   ├── test_types.ml           ← Unit tests for type operations
│   │   ├── test_unification.ml     ← Unification algorithm tests
│   │   ├── test_inference.ml       ← Inference engine tests
│   │   └── test_polymorphism.ml    ← Polymorphism tests
│   │
│   ├── integration/
│   │   ├── test_programs/          ← Sample programs for integration testing
│   │   └── test_type_checker.ml    ← End-to-end type checker tests
│   │
│   └── examples/
│       ├── simple_expressions.ml   ← Basic type checking examples
│       ├── inference_examples.ml   ← Type inference examples
```

```
│       └── polymorphic_examples.ml  ← Polymorphic code examples
│
└── dune-project                     ← OCaml build configuration
```

This file structure supports **incremental development** where you can implement and test each milestone independently. Milestone 1 focuses on the `types/` and `environment/` modules. Milestone 2 adds the `checking/` module for basic type checking. Milestone 3 implements the `inference/` module for constraint-based type inference. Milestone 4 adds the `polymorphism/` module for let-polymorphism support.

Module Dependency Graph

The dependency relationships between modules form a directed acyclic graph that prevents circular dependencies and enables independent testing:

Module	Dependencies	Exports	Used By
`types`	None (foundation layer)	Type definitions, equality, operations	All other modules
`environment`	`types`	Environment operations, scope management	`checking`, `inference`, `polymorphism`
`checking`	`types`, `environment`	Basic type checking functions	`type_checker` main entry point
`inference`	`types`, `environment`, `checking`	Constraint generation and solving	`type_checker` main entry point
`polymorphism`	`types`, `environment`, `inference`	Type schemes, generalization, instantiation	`type_checker` main entry point
`errors`	`types`	Error types and reporting functions	`checking`, `inference`, `polymorphism`

Interface Design Philosophy

Each module exposes a **minimal public interface** that hides implementation details while providing all necessary functionality for dependent modules. The `.mli` interface files serve as contracts that define exactly what operations are available and how they should be used.

The `types.mli` interface exposes type constructors, equality checking, and basic operations, but hides substitution implementation details. The `inference.mli` interface provides high-level functions like `infer_expression_type` but hides the constraint generation and unification algorithms. This **information hiding** principle allows us to change internal algorithms without affecting dependent code.

Common Pitfall: Circular Module Dependencies

⚠️ Pitfall: Importing inference in basic checker

Beginning implementers often create circular dependencies by making the basic checker import inference utilities, then having inference import the basic checker for some operations. This creates a compilation error in OCaml.

Why it's wrong: Circular dependencies indicate poor separation of concerns and make testing and reasoning about code much harder.

How to fix: Move shared utilities into a lower-level module that both can depend on, or redesign the interfaces to eliminate the circular dependency. The inference engine should build on top of basic checking, not the reverse.

Implementation Guidance

The type checker implementation requires careful attention to module organization and dependency management. OCaml's strong module system helps enforce architectural boundaries, but requires understanding how to structure interfaces and implementations correctly.

Technology Recommendations

Component	Simple Option	Advanced Option
Type Representation	Algebraic data types with pattern matching	GADTs with type-level guarantees
Environment Management	List of association lists for scope stack	Persistent hash maps with structural sharing
Constraint Solving	Simple Robinson unification	Unification with constraint optimization
Error Reporting	Printf-style string formatting	Structured errors with JSON serialization
Testing Framework	OCaml built-in assertions	OUnit2 with property-based testing
Build System	Simple `ocamlc` compilation	Dune with automatic dependency management

Recommended Project Structure

Start with this minimal directory structure and expand as you implement each milestone:

```ocaml
(* dune-project file *)
(lang dune 3.0)
(package
 (name type_checker)
 (depends ocaml dune))
```

```ocaml
(* src/dune file *)
(executables
 (public_names type_checker)
 (name type_checker)
 (libraries str))

(* test/dune file *)
(test
 (name test_type_checker)
 (libraries type_checker))
```

Infrastructure Starter Code

Here's the complete foundation you need to get started. This code provides the basic type system and environment operations that all other components depend on:

OCAML

```ocaml
(* src/types/types.ml - Complete implementation *)

type ty =

  | TInt

  | TBool

  | TString

  | TVar of string

  | TFun of ty * ty

  | TForall of string list * ty


let type_equal t1 t2 =

  let rec equal t1 t2 = match (t1, t2) with

    | (TInt, TInt) | (TBool, TBool) | (TString, TString) -> true

    | (TVar v1, TVar v2) -> String.equal v1 v2

    | (TFun (p1, r1), TFun (p2, r2)) -> equal p1 p2 && equal r1 r2

    | (TForall (vars1, t1), TForall (vars2, t2)) ->

        List.length vars1 = List.length vars2 && equal t1 t2

    | _ -> false

  in equal t1 t2


let var_counter = ref 0

let fresh_type_var () =

  incr var_counter;

  TVar ("t" ^ string_of_int !var_counter)


let rec string_of_type = function

  | TInt -> "int"

  | TBool -> "bool"

  | TString -> "string"
```

```
  | TVar name -> name
  | TFun (param, result) ->
      "(" ^ string_of_type param ^ " -> " ^ string_of_type result ^ ")"
  | TForall (vars, ty) ->
      "forall " ^ String.concat " " vars ^ ". " ^ string_of_type ty

let occurs_check var_name ty =
  let rec occurs = function
    | TVar name -> String.equal var_name name
    | TFun (param, result) -> occurs param || occurs result
    | TForall (_, body) -> occurs body
    | TInt | TBool | TString -> false
  in occurs ty
```

```ocaml
(* src/environment/type_env.ml - Complete implementation *)

module TypeEnv = struct

  type t = (string * ty) list list


  let empty = [[]]



  let extend name ty env = match env with

    | [] -> [[(name, ty)]]

    | scope :: rest -> ((name, ty) :: scope) :: rest



  let rec lookup name = function

    | [] -> None

    | scope :: rest ->

        (match List.assoc_opt name scope with

         | Some ty -> Some ty

         | None -> lookup name rest)



  let push_scope env = [] :: env



  let pop_scope = function

    | [] -> []

    | _ :: rest -> rest

end
```

Core Logic Skeleton Code

These are the key functions you need to implement for each milestone. The signatures and TODO comments map directly to the algorithms described in the design sections:

```ocaml
(* src/inference/unification.ml - Skeleton for you to implement *)

type substitution = (string * ty) list

type constraint_t = ty * ty


let empty_subst = []


let apply_subst_to_type subst ty =

  (* TODO 1: Implement recursive substitution application *)

  (* TODO 2: For TVar, look up in substitution and apply recursively *)

  (* TODO 3: For structured types, apply to components *)

  (* TODO 4: Handle TForall by avoiding capture of quantified variables *)

  failwith "implement apply_subst_to_type"


let compose_subst s1 s2 =

  (* TODO 1: Apply s1 to all type values in s2 *)

  (* TODO 2: Combine results, with s2 taking precedence for conflicts *)

  (* TODO 3: Ensure result substitution is idempotent *)

  failwith "implement compose_subst"


let unify t1 t2 =

  (* TODO 1: Handle identical types -> return empty_subst *)

  (* TODO 2: Handle type variables -> check occurs check, return binding *)

  (* TODO 3: Handle function types -> recursively unify components *)

  (* TODO 4: Handle incompatible types -> raise unification error *)

  (* TODO 5: Use substitution composition for recursive calls *)

  failwith "implement unify"
```

```ocaml
(* src/inference/constraints.ml - Skeleton for constraint generation *)

let generate_constraints expr env =

  (* TODO 1: Pattern match on expression type *)

  (* TODO 2: For literals -> return appropriate primitive type, no constraints *)

  (* TODO 3: For variables -> lookup in environment, no constraints *)

  (* TODO 4: For binary operators -> generate constraints for operands *)

  (* TODO 5: For function calls -> constrain argument and result types *)

  (* TODO 6: Collect all constraints from subexpressions *)

  (* Return: (expression_type, constraint_list) *)

  failwith "implement generate_constraints"
```

Language-Specific Hints

OCaml-Specific Implementation Tips:

- Use `List.assoc_opt` for safe dictionary lookup that returns `option` instead of raising exceptions
- Pattern matching on algebraic data types provides exhaustiveness checking—the compiler warns if you miss cases
- Use `ref` for the global type variable counter, but avoid mutable state elsewhere
- The `failwith` function is useful for unimplemented code branches during development
- Use `@@` operator for function application to avoid parentheses: `string_of_type @@ TFun (TInt, TBool)`

Error Handling Patterns:

- Use `Result.t` types for functions that can fail: `unify : ty -> ty -> (substitution, string) result`
- Use `option` types for optional lookups: `TypeEnv.lookup : string -> TypeEnv.t -> ty option`
- Create custom exception types for different error categories: `exception UnificationError of ty * ty`

Performance Considerations:

- Type environments use lists for simplicity, but consider `Map.Make(String)` for programs with many variables
- Fresh variable generation is globally sequential—consider parameterizing it if you need parallel type checking

- Substitution application can be expensive for large types—consider memoization if performance becomes an issue

Milestone Checkpoint

After implementing the architecture and basic components, verify your system works correctly:

Checkpoint 1 (After Types and Environment):

```bash
# Test basic type operations

ocaml

# In OCaml REPL:

#use "src/types/types.ml";;

#use "src/environment/type_env.ml";;

let t1 = TFun(TInt, TBool);;

let t2 = TFun(TInt, TBool);;

type_equal t1 t2;;  (* Should return true *)
```

Expected behavior:

- Type equality works for all type constructors
- Fresh variable generation produces unique names
- Environment lookup finds variables in correct scopes
- Environment shadowing hides outer scope variables correctly

Checkpoint 2 (After Basic Checking):

- Simple expressions type check correctly (literals, variables, operators)
- Function calls verify argument types match parameters
- Type errors produce clear messages with source locations
- Error recovery allows checking to continue after type mismatches

Signs something is wrong:

- **Type equality returns false for identical types**: Check pattern matching is exhaustive and handles all constructors
- **Environment lookup fails for defined variables**: Verify scope stack is managed correctly during traversal
- **Fresh variables have duplicate names**: Ensure variable counter is properly incremented
- **Occurs check allows infinite types**: Review recursive type traversal in occurs check implementation

Data Model and Type Representations

> **Milestone(s):** Milestone 1 (Type Representation), Milestone 2 (Basic Type Checking), Milestone 3 (Type Inference), Milestone 4 (Polymorphism)

Think of the type system's data model as the **vocabulary** of a formal language for describing program structure. Just as human languages have nouns, verbs, and adjectives with specific grammatical rules, our type checker needs a precise vocabulary for describing what kinds of values exist (primitives like integers), how they can be combined (functions), and what relationships exist between them (type equality, compatibility). The data structures we define here become the foundational "words" that all other components of the type checker use to communicate about program structure.

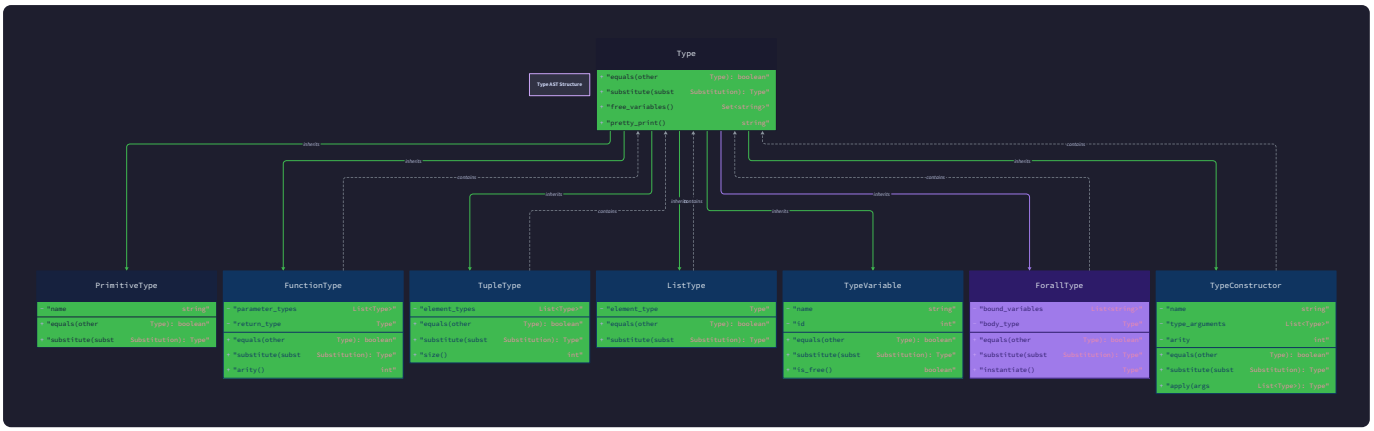
This vocabulary must be rich enough to express everything from simple integer arithmetic to complex polymorphic functions, yet structured enough that we can write algorithms to manipulate it mechanically. The core insight is that **types are data** — they're not just annotations in the source code, but first-class values that our type checker program creates, stores, compares, and transforms throughout the checking process.

The data model serves three critical functions in our type checker architecture. First, it provides the **representational foundation** — the concrete data structures that encode abstract type-theoretic concepts like "function from integers to booleans" or "polymorphic identity function." Second, it defines the **computational interface** — the operations we can perform on these representations, like checking equality, applying substitutions, or generating fresh type variables. Third, it establishes the **semantic relationships** — how different type representations relate to each other through subtyping, unification, and polymorphic instantiation.

Type AST Structure

The type abstract syntax tree forms the core vocabulary of our type system, representing every possible type that can exist in our language. Think of it as a **construction kit** where complex types are built from simpler components using well-defined combination rules. Just as mathematical expressions are built from numbers, variables, and operators, our type expressions are built from primitives, type variables, and type constructors.

The algebraic data type approach gives us **structural precision** — each type construct has exactly the components it needs, no more and no less. A function type contains exactly two sub-types (parameter and return), while a primitive type stands alone. This precision eliminates entire classes of representation errors and makes pattern matching exhaustive, ensuring our type checking algorithms handle every possible case.



Decision: Algebraic Data Types for Type Representation

- **Context**: We need to represent the full spectrum of types in our language, from simple primitives to complex polymorphic functions, with operations like equality checking, substitution, and pretty printing.
- **Options Considered**:
  1. Class hierarchy with inheritance (OOP approach)
  2. Tagged unions/algebraic data types (functional approach)
  3. String-based type representations with parsing
- **Decision**: Algebraic data types with exhaustive pattern matching
- **Rationale**: ADTs provide compile-time guarantees that all type cases are handled in every function, eliminate null pointer errors, and make the type structure explicit in the data definition rather than scattered across class methods.
- **Consequences**: Pattern matching becomes the primary way to inspect types, leading to more robust code but requiring functional programming familiarity.

Type Construct	OCaml Representation	Description
`TInt`	`TInt`	Primitive integer type representing whole numbers
`TBool`	`TBool`	Primitive boolean type for true/false values
`TString`	`TString`	Primitive string type for text values
`TVar`	`TVar of string`	Type variable for inference and polymorphism, identified by name
`TFun`	`TFun of ty * ty`	Function type with parameter type and return type
`TForall`	`TForall of string list * ty`	Polymorphic type with quantified variables and body type

The `ty` algebraic data type captures the essential structure of our type system. Primitive types (`TInt`, `TBool`, `TString`) represent the atomic values in our language — these are the "ground truth" types that need no further decomposition. The `TVar` constructor represents type variables, which serve dual purposes: during inference, they act as unknowns to be solved, and in polymorphic types, they represent parameters that can be instantiated with concrete types.

Function types (`TFun`) encode the fundamental computational abstraction of our language. The structure `TFun(param_ty, return_ty)` captures that functions are transformations from input types to output types. This binary structure naturally extends to multi-parameter functions through currying — a function taking two parameters becomes `TFun(param1, TFun(param2, return_ty))`.

> The key insight for function types is that `TFun` is right-associative by convention. The type `int -> bool -> string` parses as `TFun(TInt, TFun(TBool, TString))`, representing a function that takes an integer and returns a function from booleans to strings.

Polymorphic types (`TForall`) represent the most sophisticated construct in our type system. The structure `TForall(vars, body)` explicitly quantifies over type variables, making polymorphism first-class in our representation. For example, the polymorphic identity function has type `TForall(["a"], TFun(TVar("a"), TVar("a")))`, clearly showing that it works for any type `a`.

The fundamental operations on types center around **structural traversal** and **variable manipulation**. Type equality requires deep structural comparison, checking that corresponding components have identical structure. Pretty printing involves recursive descent through the type structure, adding parentheses and formatting to produce readable representations. Most importantly, substitution operations replace type variables with concrete types throughout the structure, enabling the unification algorithm that drives type inference.

> **Decision: String-Based Type Variable Identity**
>
> - **Context**: Type variables need unique identity for unification and substitution operations.
> - **Options Considered**:
>   1. String names with manual uniqueness management
>   2. Integer IDs with separate name tracking
>   3. Reference-based identity with mutable cells
> - **Decision**: String names with systematic fresh name generation
> - **Rationale**: String names provide readable debugging output and integrate naturally with source-level type annotations, while fresh name generation ensures uniqueness without complex ID management.
> - **Consequences**: Type variable names become part of the semantic identity, requiring careful handling during substitution and alpha-equivalence checking.

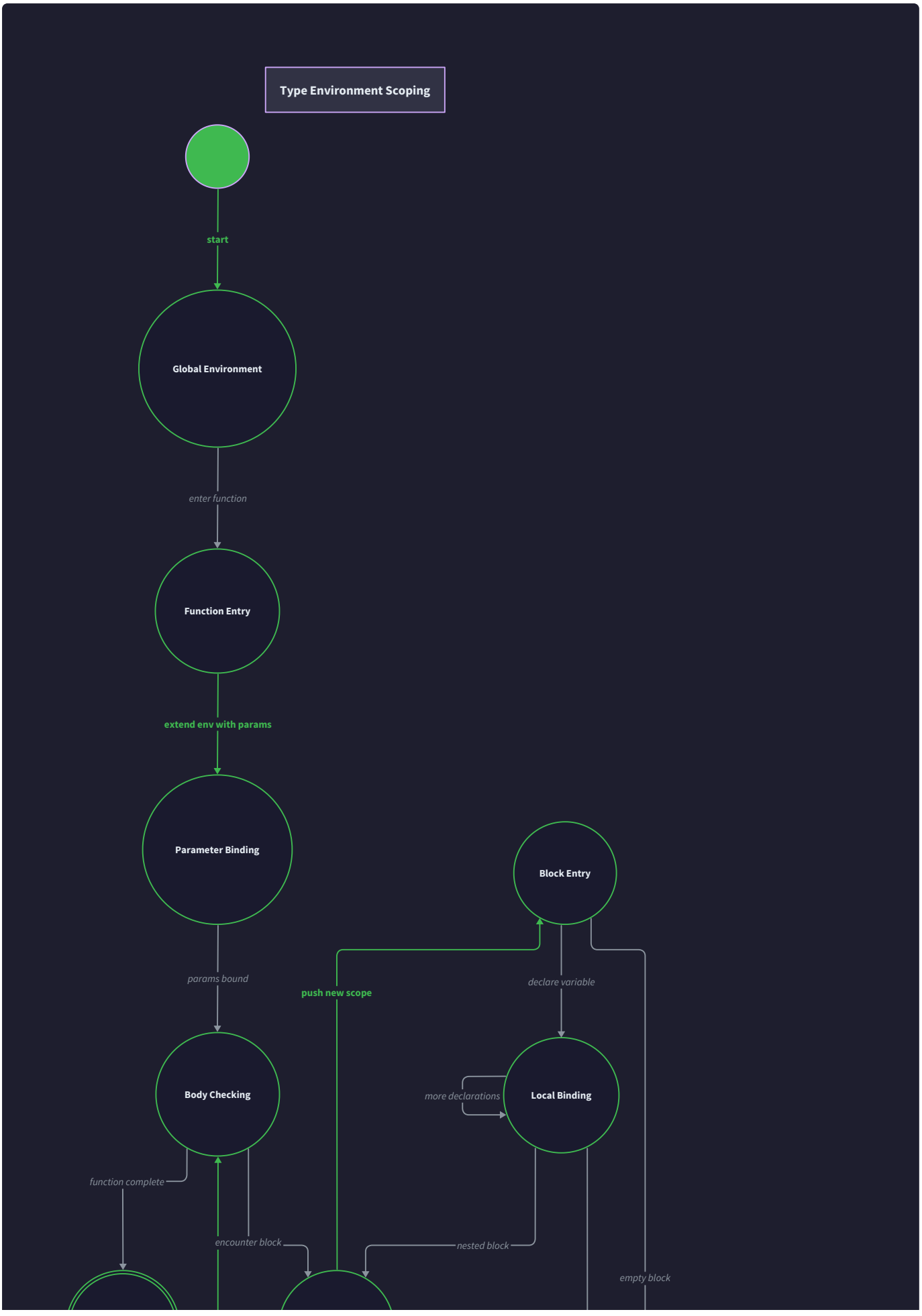
Core Type Operation	Signature	Description
`type_equal(t1: ty, t2: ty): bool`	Two types → Boolean	Structural equality check for identical type structure
`string_of_type(ty: ty): string`	Type → String	Pretty printer producing readable type representations
`fresh_type_var(): ty`	Unit → Type	Generates unique type variable for inference
`occurs_check(var: string, ty: ty): bool`	Variable name, Type → Boolean	Detects if variable occurs in type structure

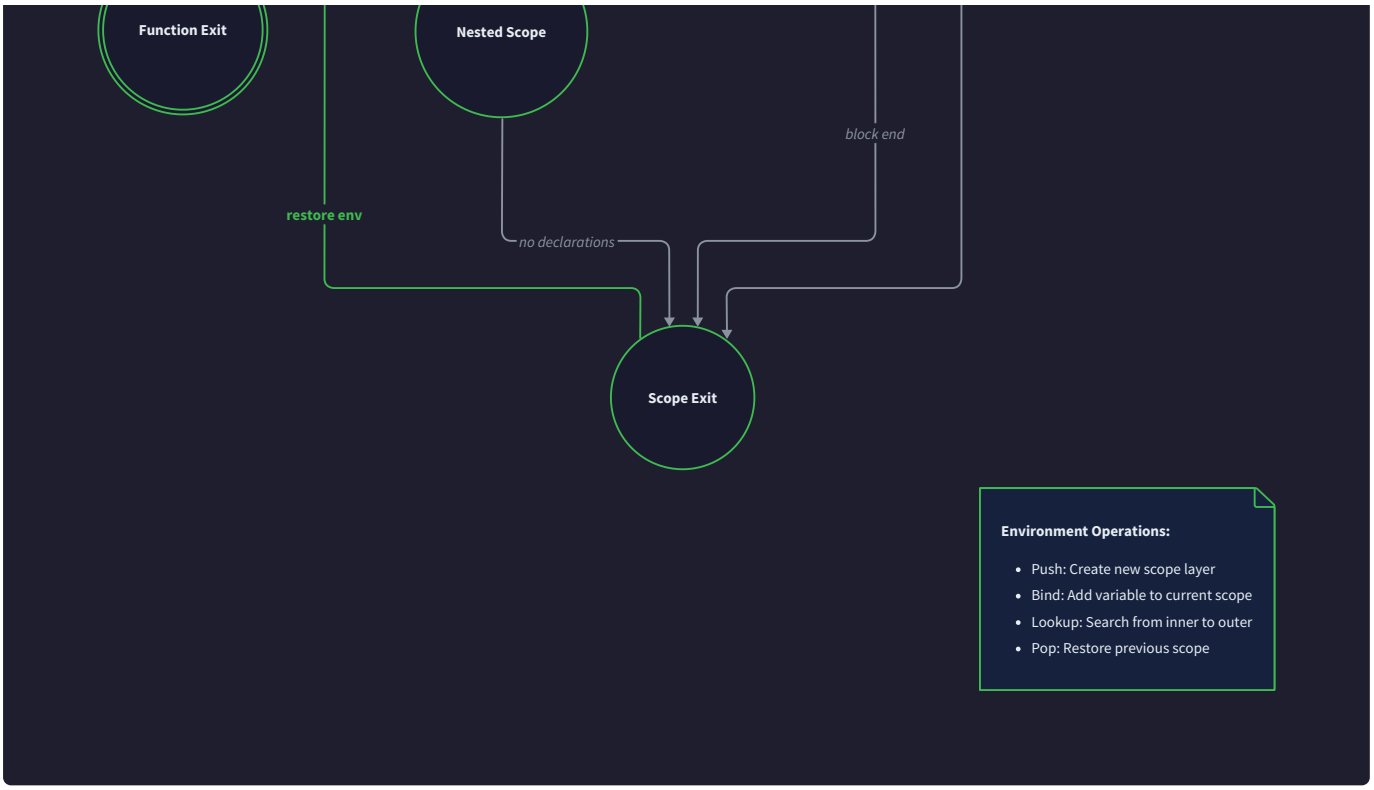
The `occurs_check` function deserves special attention as it prevents the creation of infinite types during unification. When unifying a type variable `X` with a type `T`, the occurs check ensures that `X` does not appear anywhere within `T`. Without this check, we could generate paradoxical types like `X = X -> int`, leading to infinite structures that break the type checker's termination guarantees.

Type Environment and Scoping

The type environment serves as the **memory** of our type checker, maintaining the association between variable names and their types as we traverse the program's scope structure. Think of it as a **stack of symbol tables** where each scope level can shadow bindings from outer scopes, but inner scopes can still access outer bindings when no shadowing occurs.

The environment's scoping behavior mirrors the lexical structure of the source program. When we enter a new scope (function body, let expression, or block statement), we push a new binding layer onto the environment. When we exit the scope, we pop that layer, automatically restoring the previous bindings. This stack discipline ensures that variable lookups follow the **inside-out** principle — we search from the innermost scope outward until we find a binding.


Type Environment Scoping
start
Global Environment
enter function
Function Entry
extend env with params
Parameter Binding
params bound
Body Checking
function complete
encounter block
push new scope
Block Entry
declare variable
Local Binding
more declarations
nested block
empty block



The `TypeEnv.t` type represents this layered structure as a list of binding lists, where each inner list represents one scope level. The head of the outer list is always the current (innermost) scope, making extension and lookup operations efficient. This representation naturally handles scope nesting without requiring complex tree structures or explicit parent pointers.

> **Decision: List-of-Lists Environment Representation**
>
> - **Context**: Need efficient scope management with frequent extend/lookup operations and clean scope entry/exit.
> - **Options Considered**:
>   1. Single flat map with scope-qualified names
>   2. Tree structure with explicit parent pointers
>   3. List of binding lists (stack of scopes)
> - **Decision**: List of binding lists with stack discipline
> - **Rationale**: Provides O(1) scope entry/exit, natural shadowing semantics, and simple implementation without memory management complexity.
> - **Consequences**: Variable lookup becomes O(n) in scope depth, but typical programs have shallow nesting, and the simplicity benefits outweigh performance concerns.

TypeEnv Data Structure	Field	Type	Description
`TypeEnv.t`	Scope stack	`(string * ty) list list`	Stack of scopes, each containing name-to-type bindings

TypeEnv Operations	Method	Parameters	Returns	Description
`TypeEnv.empty`	None	`TypeEnv.t`	Creates empty environment with single empty scope	
`TypeEnv.extend`	`name: string, ty: ty, env: TypeEnv.t`	`TypeEnv.t`	Adds binding to innermost scope, shadowing outer bindings	
`TypeEnv.lookup`	`name: string, env: TypeEnv.t`	`ty option`	Searches scopes inside-out, returns first matching binding	
`TypeEnv.enter_scope`	`env: TypeEnv.t`	`TypeEnv.t`	Pushes new empty scope onto environment stack	
`TypeEnv.exit_scope`	`env: TypeEnv.t`	`TypeEnv.t`	Pops innermost scope, restoring previous bindings	

The scoping protocol follows a disciplined approach where scope changes are explicit and symmetric. Every `enter_scope` operation must be paired with a corresponding `exit_scope`, typically using the language's automatic resource management (like OCaml's `let ... in` expressions or explicit scope management in recursive functions). This discipline prevents scope leakage where inner scope bindings accidentally persist beyond their intended lifetime.

Environment threading becomes critical during recursive type checking. As we recursively check sub-expressions, we pass the current environment down the call tree, and each recursive call may extend the environment with new bindings (like lambda parameters or let-bound variables). The key insight is that environment changes are **local** — extensions made in one branch of the recursion don't affect other branches, maintaining proper lexical scoping.

> The environment extension protocol ensures that `extend` operations always add to the innermost scope. This means that `let x = 5 in let x = true in x` correctly binds the inner `x` to boolean type, shadowing the outer integer binding without destroying it.

Consider a concrete example of environment evolution during type checking:

1. Start with empty environment: `[[]]` (one empty scope)
2. Enter function scope: `[[], []]` (new empty scope pushed)
3. Add parameter binding: `[[("x", TInt)], []]` (parameter added to inner scope)
4. Check function body with extended environment
5. Exit function scope: `[[]]` (inner scope popped, parameter binding removed)

The scoping semantics handle several important edge cases. Variable shadowing occurs naturally when inner scopes contain bindings with the same name as outer scopes — lookup finds the inner binding first and stops searching. Mutual recursion requires careful handling where multiple definitions extend the environment simultaneously before any individual definition is checked. Forward references within the same scope level need special treatment to avoid lookup failures during recursive checking.

⚠️ **Pitfall: Environment Mutation During Checking** Many implementations incorrectly modify the environment in place during type checking, leading to scope pollution where inner scope bindings persist after exiting their scope. This breaks lexical scoping and can cause later expressions to see variables that should be out of scope. Always use functional environment updates that create new environment values rather than modifying existing ones.

Constraints and Substitutions

The constraint and substitution system forms the **computational engine** of type inference, transforming the declarative problem of "what types make this program correct?" into the algorithmic problem of "what type variable assignments satisfy these equality constraints?" Think of constraints as a system of **type equations** that we collect during program traversal, and substitutions as the **solution** that makes all equations simultaneously true.

This approach separates type inference into two distinct phases: **constraint generation** walks the program structure and emits type equality requirements, while **constraint solving** uses unification to find type assignments that satisfy all requirements. This separation provides modularity — we can modify the constraint generation rules without changing the solver, or optimize the solver without affecting constraint generation.

The constraint representation captures the fundamental operation of type inference: asserting that two types must be equal. Each constraint `(ty1, ty2)` represents the requirement that `ty1` and `ty2` have identical structure after applying the solution substitution. The constraint list accumulates these requirements as we traverse program constructs, building up a system of equations that characterizes all valid typing assignments.

Decision: Constraint-Based Type Inference Architecture

- **Context**: Need to infer types for expressions without explicit annotations while handling complex interactions between multiple type variables.
- **Options Considered**:
  1. Direct recursive type assignment during AST traversal
  2. Constraint generation followed by batch solving
  3. Incremental unification during traversal
- **Decision**: Separate constraint generation and batch solving phases
- **Rationale**: Separates concerns cleanly, allows global optimization of constraint solving, and handles complex constraint interactions that would be difficult to resolve incrementally.
- **Consequences**: Requires two passes over the program structure but provides more robust handling of polymorphic and recursive types.

Constraint System Types	Name	Structure	Description
`constraint_t`	Type equality constraint	`ty * ty`	Asserts two types must have identical structure
`substitution`	Variable assignments	`(string * ty) list`	Maps type variable names to concrete types
`var_id`	Variable identifier	`int`	Unique integer for generating fresh type variables

Constraint Operations	Method	Parameters	Returns	Description
`generate_constraints`	`expr, env: TypeEnv.t`	`ty * constraint_t list`	Infers expression type and collects equality constraints	
`unify`	`t1: ty, t2: ty`	`substitution`	Finds most general unifier making two types equal	
`apply_subst_to_type`	`subst: substitution, ty: ty`	`ty`	Replaces type variables with assigned types	
`compose_subst`	`s1: substitution, s2: substitution`	`substitution`	Combines two substitutions into equivalent single substitution	

The substitution data structure represents a **partial function** from type variable names to types, encoding the solution to our constraint system. A substitution `[("a", TInt); ("b", TBool)]` means "replace type variable `a` with `TInt` and type variable `b` with `TBool` everywhere." The key property is that applying a substitution to both sides of any constraint should make them syntactically equal.

Substitution application transforms types by recursively replacing type variables with their assigned values. The operation `apply_subst_to_type(subst, ty)` performs a systematic traversal of the type structure, substituting variables when found and recursively processing sub-components. This operation must handle the occurs check implicitly — if a substitution would create an infinite type, the unification process should have already detected and rejected it.

> The critical insight for substitution composition is that `compose_subst(s1, s2)` creates a substitution equivalent to applying `s2` first, then applying `s1` to the result. This ordering matches function composition and ensures that later substitutions can refine earlier ones.

Constraint generation follows the structure of the expression being typed, emitting equality requirements as it encounters type relationships. For a function call `f(arg)`, we generate constraints that:

1. The function `f` has type `TFun(arg_ty, result_ty)` for fresh type variables
2. The argument `arg` has type `arg_ty` (recursively determined)
3. The overall call expression has type `result_ty`

This process builds a constraint system where solving for the type variables determines the types of all sub-expressions consistently.

The unification algorithm serves as the **constraint solver**, finding type variable assignments that simultaneously satisfy all constraints. The algorithm works by case analysis on the structure of the types being unified:

1. **Variable-Type unification**: If unifying `TVar(x)` with type `T`, check the occurs check, then bind `x` to `T`
2. **Identical primitives**: `TInt` unifies with `TInt`, `TBool` with `TBool`, etc.
3. **Function types**: `TFun(p1, r1)` unifies with `TFun(p2, r2)` by recursively unifying parameters and return types
4. **Failure cases**: Different primitives, function vs. primitive, occurs check violations

The occurs check prevents infinite types by ensuring that when binding variable `x` to type `T`, the variable `x` does not occur anywhere within `T`. Without this check, we could create paradoxical assignments like `x = x -> int`, leading to infinite type expressions that break the type checker's termination properties.

⚠️ **Pitfall: Incorrect Substitution Composition Order** Many implementations get the order wrong in `compose_subst(s1, s2)`, creating substitutions that don't correspond to the intended sequence of applications. The correct composition applies `s2` first, then `s1`, matching the mathematical convention for function composition. Always test with non-commuting substitutions to verify the order is correct.

⚠️ **Pitfall: Missing Occurs Check in Unification** Omitting the occurs check allows the creation of infinite types that can cause the type checker to loop indefinitely during substitution application. Always check that a type variable doesn't occur in the type it's being bound to, and fail unification if it does. This check is essential for termination guarantees.

The constraint solving process transforms a collection of type equations into a concrete type assignment. Consider unifying the constraints from `let f = fun x -> x + 1`:

1. Generate constraints: `f : 'a -> 'b`, `x : 'a`, `(+) : int -> int -> int`, `1 : int`
2. From `x + 1`, constrain `'a = int` and `'b = int`
3. Solve to get substitution `[('a, int); ('b, int)]`
4. Apply substitution: `f : int -> int`

This systematic approach ensures that type inference finds the most general solution that satisfies all constraints, or reports a clear error when no solution exists.

Implementation Guidance

The data model implementation requires careful attention to both correctness and performance, as these data structures form the foundation for all type checker operations. The algebraic data type representation should leverage the language's pattern matching capabilities for exhaustive case handling, while the environment and constraint systems need efficient implementations of their core operations.

Technology Recommendations:

Component	Simple Approach	Advanced Approach
Type Representation	Basic algebraic data types	Optimized with hash-consing for type equality
Type Environment	List of association lists	Persistent hash maps with path copying
Constraint Solving	Basic unification algorithm	Union-find with path compression
Pretty Printing	Simple recursive descent	Precedence-aware with minimal parentheses

Recommended File Structure:

```
type-checker/
  src/
    types.ml                ← Core type representation (ty, type_scheme)
    typeenv.ml              ← Environment operations (extend, lookup, scoping)
    constraints.ml          ← Constraint types and substitution operations
    unify.ml                ← Unification algorithm and occurs check
    pretty.ml               ← Pretty printing for types and constraints
  test/
    test_types.ml           ← Type equality and operations tests
    test_typeenv.ml         ← Environment scoping and lookup tests
    test_unify.ml           ← Unification algorithm and edge case tests
```

Core Type System Implementation:

OCAML

```ocaml
(* types.ml - Complete type representation *)

type ty =

  | TInt

  | TBool

  | TString

  | TVar of string

  | TFun of ty * ty

  | TForall of string list * ty


type type_scheme = Forall of string list * ty


let type_var_counter = ref 0


let fresh_type_var () =

  incr type_var_counter;

  TVar ("t" ^ string_of_int !type_var_counter)


let rec type_equal t1 t2 =

  (* TODO: Implement structural equality for all type constructors *)

  (* Hint: Use pattern matching on (t1, t2) pairs *)

  failwith "TODO: implement type_equal"


let rec string_of_type = function

  (* TODO: Implement pretty printing with proper precedence *)

  (* Hint: Function types are right-associative, add parens for clarity *)

  | TInt -> "int"

  | TBool -> "bool"

  | TString -> "string"

  | TVar x -> x
```

```
  | TFun (param, ret) -> failwith "TODO: format function type with arrows"
  | TForall (vars, body) -> failwith "TODO: format polymorphic type with forall"

let rec occurs_check var_name = function
  (* TODO: Check if var_name appears anywhere in the type structure *)
  (* This prevents infinite types like X = X -> int *)
  | TVar x -> failwith "TODO: check if x equals var_name"
  | TFun (param, ret) -> failwith "TODO: recursively check param and ret"
  | TForall (vars, body) -> failwith "TODO: check body, handle variable capture"
  | _ -> false (* primitives never contain variables *)
```

Type Environment Implementation:

OCAML

```ocaml
(* typeenv.ml - Environment with lexical scoping *)

module TypeEnv = struct

  type t = (string * ty) list list


  let empty = [[]]


  let extend name ty env =

    (* TODO: Add binding to the innermost scope (head of list) *)

    (* Hint: Pattern match on env to get current scope and rest *)

    failwith "TODO: implement extend"


  let rec lookup name env =

    (* TODO: Search scopes from inside-out until binding found *)

    (* Return None if not found in any scope *)

    (* Hint: Use List.assoc_opt to search individual scopes *)

    failwith "TODO: implement lookup"


  let enter_scope env =

    (* TODO: Push new empty scope onto environment stack *)

    [] :: env


  let exit_scope = function

    (* TODO: Pop innermost scope, restore previous environment *)

    (* Handle edge case of popping from empty environment *)

    | [] -> failwith "Cannot exit empty environment"

    | _ :: rest -> rest
```

```
end
```

Constraint and Substitution System:

```ocaml
(* constraints.ml - Constraint generation and substitution *)

type constraint_t = ty * ty

type substitution = (string * ty) list


let empty_subst = []


let rec apply_subst_to_type subst = function

  (* TODO: Replace type variables using substitution mapping *)

  (* Recursively apply to sub-components of compound types *)

  | TVar x ->

    (try List.assoc x subst

     with Not_found -> TVar x)

  | TFun (param, ret) -> failwith "TODO: apply substitution to function components"

  | TForall (vars, body) -> failwith "TODO: apply substitution avoiding capture"

  | t -> t (* primitives unchanged *)


let compose_subst s1 s2 =

  (* TODO: Create substitution equivalent to applying s2 then s1 *)

  (* Apply s1 to all types in s2, then combine with s1 *)

  (* Handle overlapping variable assignments correctly *)

  failwith "TODO: implement substitution composition"
```

Unification Algorithm Core:

```ocaml
(* unify.ml - Robinson unification algorithm *)

let rec unify t1 t2 =

  (* TODO: Implement unification by cases on type structure *)

  match (t1, t2) with

  | (TVar x, t) | (t, TVar x) ->

    (* TODO: Check occurs check, then bind variable to type *)

    if occurs_check x t then

      failwith ("Occurs check failed: " ^ x ^ " occurs in " ^ string_of_type t)

    else

      [(x, t)]



  | (TInt, TInt) | (TBool, TBool) | (TString, TString) ->

    (* TODO: Identical primitives unify with empty substitution *)

    empty_subst



  | (TFun (p1, r1), TFun (p2, r2)) ->

    (* TODO: Unify parameters, then unify returns under parameter substitution *)

    (* Use substitution composition to combine results *)

    failwith "TODO: unify function types"



  | _ ->

    failwith ("Cannot unify " ^ string_of_type t1 ^ " with " ^ string_of_type t2)
```

Language-Specific Hints:

- **Pattern Matching**: Use exhaustive pattern matching with compiler warnings enabled to catch missing cases in type operations
- **Reference Cells**: The `type_var_counter` uses a mutable reference for generating unique variable names

- **List Operations**: `List.assoc` and `List.assoc_opt` provide efficient association list lookups for small environments
- **Exception Handling**: Use structured exceptions for unification failures rather than generic failures
- **Testing Strategy**: Test each operation with edge cases like empty environments, infinite types, and complex nested structures

Milestone Checkpoints:

After implementing the type representation:

- Run `ocaml types.ml` and test `type_equal (TFun(TInt, TBool)) (TFun(TInt, TBool))` returns `true`
- Verify `fresh_type_var ()` generates unique variables on each call
- Check that `occurs_check "x" (TFun(TVar "x", TInt))` returns `true`

After implementing the type environment:

- Test that `TypeEnv.lookup "x" (TypeEnv.extend "x" TInt TypeEnv.empty)` returns `Some TInt`
- Verify that inner scope bindings shadow outer ones correctly
- Check that `exit_scope (enter_scope env)` returns the original environment

After implementing constraints and substitution:

- Test `apply_subst_to_type [("x", TInt)] (TVar "x")` returns `TInt`
- Verify `compose_subst s1 s2` produces the same result as applying `s2` then `s1`
- Check that `unify (TVar "x") TInt` returns `[("x", TInt)]`

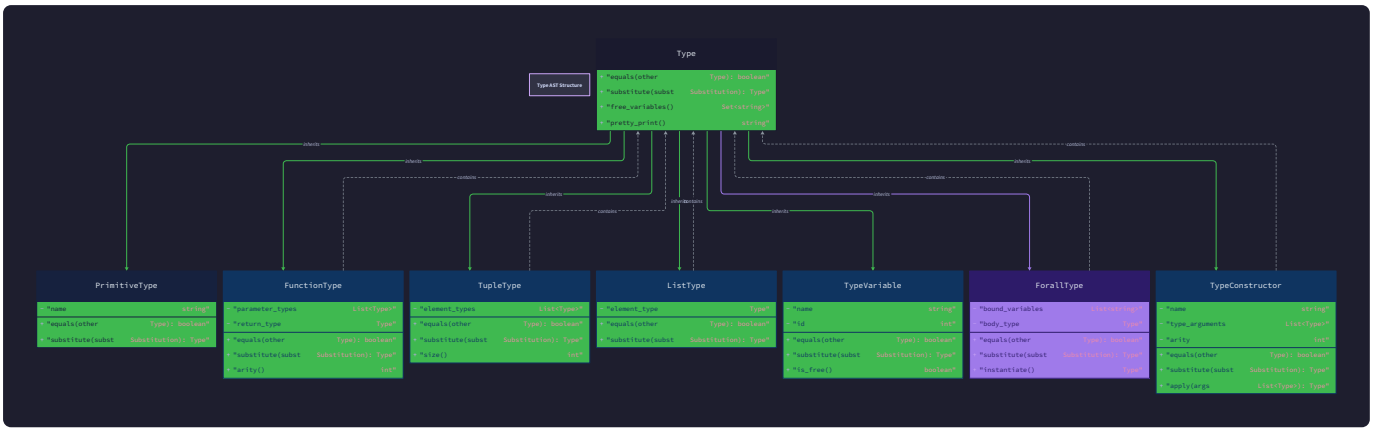
Type Representation System

> **Milestone(s):** Milestone 1 (Type Representation)

Think of the type representation system as the **foundation and blueprint vocabulary** for our type checker — much like how a legal system needs precise definitions of concepts like "contract," "property," and "liability" before it can reason about disputes. Our type checker needs precise representations of "integer," "function from A to B," and "generic type T" before it can verify that code follows the rules. This foundational layer defines the algebraic data structures that represent all possible types in our language, along with the fundamental operations for comparing types and managing type variables.

The type representation system sits at the core of every type checker operation. When we parse `let add = fun x y -> x + y`, we need to represent the type `'a -> 'b -> 'c` somewhere in memory, with clear distinctions between the type variables `'a`, `'b`, and `'c`. When we later discover through inference that `'a` and `'b` must both be `int` and `'c` must also be `int`, we need precise operations to substitute those

concrete types for the variables. Without a solid type representation foundation, the sophisticated algorithms of constraint generation and unification have nothing concrete to operate on.



Core Type System Design

The heart of our type representation lies in defining an **algebraic data type** that can express every possible type in our language. This type system needs to handle three fundamental categories: primitive types that are built into the language, composite types like functions that combine simpler types, and meta-types like type variables that represent unknown types during inference.

Our `ty` type serves as the universal representation for all types in the system. Each variant captures a different category of type with the minimum information needed to distinguish it from other types and perform operations like equality checking and substitution.

Type Variant	Structure	Purpose	Example
`TInt`	No parameters	Represents integer primitive type	`42`, `x + 1`
`TBool`	No parameters	Represents boolean primitive type	`true`, `x > 0`
`TString`	No parameters	Represents string primitive type	`"hello"`, `name`
`TVar of string`	Variable name	Represents unknown type during inference	`'a`, `'result`
`TFun of ty * ty`	Parameter type * Return type	Represents function types	`int -> bool`, `'a -> 'a`
`TForall of string list * ty`	Quantified variables * Body type	Represents polymorphic types	`forall 'a. 'a -> 'a`

Decision: Algebraic Data Type for Type Representation

- **Context**: Need a precise way to represent all possible types that can appear in our language, support pattern matching for type operations, and enable recursive type structures like `('a -> 'b) -> 'a -> 'b`.
- **Options Considered**:
  1. Class hierarchy with inheritance (OOP approach)
  2. Tagged union/enum with associated data (ADT approach)
  3. String-based type representations with parsing
- **Decision**: Algebraic data type with pattern matching
- **Rationale**: ADTs provide exhaustive pattern matching that catches missing cases at compile time, enable recursive type definitions naturally, and make type transformations like substitution straightforward to implement. Class hierarchies require visitor patterns for operations, while string representations are error-prone and inefficient.
- **Consequences**: Enables clean pattern matching in all type operations, but requires functional programming concepts. Makes adding new type variants require updating all pattern matches.

The `TVar` variant deserves special attention because it represents the **unknown types** that drive our inference engine. Think of type variables as placeholders in a mathematical equation — when we see `let f = fun x -> x + 1`, we initially represent the type of `x` as some type variable `'a`, then later solve the constraint that `'a` must support the `+` operator with `int`, concluding that `'a = int`.

The `TFun` variant uses a **curried representation** where multi-parameter functions are represented as nested single-parameter functions. A function `int -> int -> bool` is represented as `TFun(TInt, TFun(TInt, TBool))`. This design choice simplifies partial application and type inference by treating all functions uniformly as single-argument functions that may return other functions.

Function Syntax	Type Representation	Curried Form
`fun x -> x + 1`	`TFun(TInt, TInt)`	`int -> int`
`fun x y -> x + y`	`TFun(TInt, TFun(TInt, TInt))`	`int -> (int -> int)`
`fun f x -> f x`	`TFun(TFun(TVar "a", TVar "b"), TFun(TVar "a", TVar "b"))`	`('a -> 'b) -> ('a -> 'b)`

Type Equality and Compatibility

Type equality forms the foundation for all type checking operations, but the notion of "equality" is more nuanced than it initially appears. Consider the question: are the types `'a -> 'a` and `'b -> 'b` equal?

They have the same structure but use different variable names. Our type equality system must handle these subtleties while remaining efficient for the thousands of type comparisons performed during checking.

Structural type equality compares types based on their shape and content rather than nominal declarations. Two function types are equal if their parameter types are equal and their return types are equal, regardless of where or how they were declared. This approach aligns naturally with type inference, where we derive types from usage patterns rather than explicit declarations.

> **Decision: Structural Type Equality**
>
> - **Context**: Need to determine when two types can be considered identical for the purposes of assignment compatibility, function application, and constraint solving.
> - **Options Considered**:
>   1. Structural equality: types are equal if they have the same structure
>   2. Nominal equality: types are equal only if they were declared with the same name
>   3. Hybrid approach: structural for inferred types, nominal for declared types
> - **Decision**: Pure structural equality for all types
> - **Rationale**: Structural equality aligns with type inference where types emerge from usage rather than declarations. It simplifies the type checker implementation by avoiding complex name resolution. Nominal typing would require maintaining type declaration information throughout inference.
> - **Consequences**: Enables flexible type inference and composition, but prevents some abstraction benefits of nominal typing. Users cannot create distinct types with identical structure.

The `type_equal` function implements deep structural comparison through recursive descent on type structures. For primitive types, equality is trivial — `TInt` equals only `TInt`. For composite types like functions, we recursively check that corresponding subcomponents are equal.

Type Comparison	Algorithm	Example
Primitives	Direct variant comparison	`type_equal(TInt, TInt) = true`
Functions	Recursive parameter and return comparison	`type_equal(TFun(TInt, TBool), TFun(TInt, TBool)) = true`
Variables	String name comparison	`type_equal(TVar "a", TVar "a") = true`
Polymorphic	Variable list and body comparison	`type_equal(TForall(["a"], TVar "a"), TForall(["b"], TVar "b")) = false`

The complexity arises with **type variables and alpha-equivalence**. The types `'a -> 'a` and `'b -> 'b` represent the same concept (a function that returns the same type as its input) but use different variable

names. However, for our basic type equality, we treat these as different types. Alpha-equivalence handling becomes important during type scheme instantiation and unification, but not for basic structural equality.

⚠️ **Pitfall: Confusing Type Equality with Type Compatibility** Many implementations conflate type equality with type compatibility or subtyping. Type equality asks "are these exactly the same type?" while compatibility asks "can I use this type where that type is expected?" For example, if we had subtyping, `Student` might be compatible with `Person` but they wouldn't be equal. Keep these concepts distinct — equality is used for constraint solving, while compatibility is used for assignment and argument checking.

Type compatibility extends beyond equality to handle relationships like subtyping, where one type can be safely used in place of another. In our basic type system, compatibility typically matches equality, but the infrastructure prepares us for potential extensions like numeric type coercion (`int` compatible with `float`) or subtyping relationships.

The `type_compatible` function provides the interface for these checks. Initially, it may simply delegate to `type_equal`, but it establishes the architectural pattern for more sophisticated compatibility rules:

Compatibility Check	Rule	Example
Identity	Any type is compatible with itself	`int` compatible with `int`
Function contravariance	Parameters contravariant, returns covariant	If `A <: B`, then `(B -> C) <: (A -> C)`
Variable instantiation	Type variables compatible with their instantiated types	`'a` compatible with `int` after substitution

Type Variable Management

Type variables serve as the **unknowns in the type inference equation** — they represent types that we haven't yet determined but will solve for through constraint collection and unification. Think of them like variables in algebra: when you write `2x + 3 = 7`, the `x` is unknown initially, but through algebraic manipulation you determine `x = 2`. Similarly, when we see `let f = fun x -> x + 1`, the type of `x` starts as an unknown type variable `'a`, but constraint solving determines `'a = int`.

The type variable management system must handle three critical responsibilities: generating fresh variables that don't conflict with existing names, detecting infinite type constructions that would make the type system unsound, and tracking variable substitutions as we solve constraints.

Fresh variable generation ensures that each unknown type gets a unique identifier that won't accidentally unify with other unknowns. Consider the expression `let f = fun x -> fun y -> x + y`. The types of `x` and `y` are initially unknown, so we need two distinct type variables, say `'a` and `'b`. If we accidentally reused the same variable name for both, we'd incorrectly constrain them to have identical types before we've analyzed their usage.

The `fresh_type_var` function maintains a global counter that increments with each call, generating names like `'_0`, `'_1`, `'_2`, etc. The underscore prefix distinguishes compiler-generated variables from user-written generic type parameters, avoiding conflicts in languages that allow explicit type variables.

Variable Generation Call	Generated Type	Purpose
`fresh_type_var()`	`TVar "_0"`	Type of first unknown parameter
`fresh_type_var()`	`TVar "_1"`	Type of second unknown parameter
`fresh_type_var()`	`TVar "_2"`	Return type of unknown function

```ocaml
(* Global counter for fresh variable generation *)

let var_counter = ref 0


let fresh_type_var () =

  let id = !var_counter in

  incr var_counter;

  TVar ("_" ^ string_of_int id)
```

Decision: Global Counter for Fresh Variables

- **Context**: Need to generate unique type variable names during inference to avoid accidental unification of distinct unknowns.
- **Options Considered**:
  1. Global mutable counter with underscore prefix
  2. UUID-based generation for guaranteed uniqueness
  3. Threading unique supply through all inference functions
- **Decision**: Global mutable counter with simple numeric suffixes
- **Rationale**: Simple to implement and understand. Performance is excellent for typical program sizes. Underscore prefix avoids conflicts with user-defined type variables. Global state is acceptable since type checking is typically single-threaded.
- **Consequences**: Requires mutable state but provides clean interface. Variable names are predictable and debuggable. Not thread-safe, but type checking is typically sequential.

The occurs check prevents the creation of infinite types during unification, which would break the type system's decidability. Consider attempting to unify the type variable `'a` with the type `'a -> int`. If we

allowed this, we'd get the infinite type `'a = 'a -> int = ('a -> int) -> int = (('a -> int) -> int) -> int = ...`, which has no finite representation.

The occurs check examines whether a type variable appears within a type expression before allowing unification. If variable `'a` occurs anywhere within type `T`, then unifying `'a` with `T` would create circularity. The check performs a recursive traversal of the type structure, looking for the variable name at any depth.

Unification Attempt	Occurs Check	Result
`'a ~ int`	`'a` does not occur in `int`	Safe, allow unification
`'a ~ 'a -> int`	`'a` occurs in `'a -> int`	Infinite type, reject
`'a ~ ('b -> 'a) -> 'c`	`'a` occurs in `('b -> 'a) -> 'c`	Infinite type, reject
`'a ~ 'b -> 'c`	`'a` does not occur in `'b -> 'c`	Safe, allow unification

The `occurs_check` function implements this traversal through pattern matching on type structure:

```ocaml
let rec occurs_check var_name ty =
  match ty with
  | TInt | TBool | TString -> false
  | TVar name -> String.equal var_name name
  | TFun (param_ty, return_ty) ->
      occurs_check var_name param_ty || occurs_check var_name return_ty
  | TForall (vars, body_ty) ->
      not (List.mem var_name vars) && occurs_check var_name body_ty
```

⚠️ **Pitfall: Forgetting the Occurs Check** A common mistake is implementing unification without the occurs check, which allows infinite types to be created. This leads to non-terminating type inference or stack overflow errors when trying to print types. The symptom is often mysterious infinite loops during unification. Always perform the occurs check before binding a type variable to a type that might contain it.

⚠️ **Pitfall: Occurs Check in Polymorphic Types** When checking occurs in `TForall` types, remember that bound variables (those in the quantifier list) shadow free variables with the same name. The type `forall 'a. 'a -> 'b` contains the free variable `'b` but not `'a`, because `'a` is bound by the quantifier. The occurs check should only worry about free occurrences.

Type variable substitution applies the solutions found by unification back to type expressions, replacing type variables with their resolved concrete types. Think of this as plugging values back into algebraic equations — once you've determined that `x = 2`, you substitute `2` for every occurrence of `x` in your expressions.

The substitution data structure maps type variable names to their replacement types. Applying a substitution walks through a type expression and replaces every occurrence of each variable with its corresponding replacement.

Original Type	Substitution	Result After Substitution
`'a -> 'b`	`[('a, int); ('b, bool)]`	`int -> bool`
`('a -> 'a) -> 'b`	`[('a, string)]`	`(string -> string) -> 'b`
`TForall(['a'], 'a -> 'b)`	`[('b, int)]`	`TForall(['a'], 'a -> int)`

The `apply_subst_to_type` function performs this replacement through recursive descent, but must handle bound variables correctly in polymorphic types. When substituting within a `TForall` type, variables bound by the quantifier should not be replaced — they're local to that type scheme.

Substitution composition combines multiple substitutions into a single substitution that has the same effect as applying them in sequence. If you have substitution S1 that maps `'a` to `'b`, and substitution S2 that maps `'b` to `int`, then the composed substitution should map `'a` directly to `int`.

The mathematical definition is: `compose_subst(S1, S2) = {(var, apply_subst_to_type(S2, ty)) | (var, ty) ∈ S1} ∪ S2`. This ensures that variables in S1's range get further substituted by S2, while preserving S2's bindings for variables not in S1.

Common Pitfalls

⚠️ **Pitfall: Mutable Type Variables** Some implementations make type variables mutable references that can be updated in place during unification. While this can be efficient, it creates subtle bugs when type variables are shared between different contexts or when backtracking is needed. Prefer immutable type representations with explicit substitution application.

⚠️ **Pitfall: String-Based Type Variable Identity** Using string equality to compare type variables works for basic cases but breaks down with alpha-equivalence and variable capture. The types `'a -> 'a` and `'b -> 'b` should be considered equivalent in many contexts, but string comparison treats them as different. Consider whether you need structural alpha-equivalence or nominal variable identity.

⚠️ **Pitfall: Forgetting Variable Scoping in Polymorphic Types** The type `forall 'a. 'a -> 'b` contains a bound variable `'a` and a free variable `'b`. Operations like substitution and the occurs check must respect this scoping — bound variables should not be replaced or considered free. Track which variables are bound at each nesting level.

⚠️ **Pitfall: Type Pretty Printing Without Precedence** When printing function types, `int -> int -> bool` should be printed as `int -> (int -> bool)` or use left-associativity conventions. Without proper precedence handling, complex types become unreadable. Consider how users will read your type error messages.

⚠️ **Pitfall: Inefficient Type Equality** Naive structural type equality can be exponentially slow on large types with shared substructure. Consider memoizing equality results or using hash-consing to make equal types share representation. Profile your type checker on large codebases to identify performance bottlenecks.

Implementation Guidance

The type representation system requires careful attention to data structure design and fundamental algorithms that will be used throughout the type checker. The implementation needs to balance simplicity for understanding with efficiency for real-world code.

Technology Recommendations

Component	Simple Option	Advanced Option
Type Representation	Direct algebraic data types	Hash-consed types with sharing
Variable Generation	Mutable counter reference	Thread-safe atomic counter
String Operations	Built-in string comparison	Interned strings for efficiency
Pretty Printing	Simple recursive formatting	Precedence-aware formatter with parentheses

Recommended File Structure

```
src/
  types/
    ty.ml                    ← core type definitions
    type_ops.ml              ← equality, occurs check, substitution
    type_env.ml              ← type environment (covered in next section)
    type_printer.ml          ← pretty printing utilities
  tests/
    test_type_equality.ml    ← type equality test cases
    test_type_vars.ml        ← type variable management tests
    test_occurs_check.ml     ← occurs check edge cases
```

Core Type Definition

```ocaml
(* ty.ml - Core type representations *)

type ty =
  | TInt
  | TBool
  | TString
  | TVar of string
  | TFun of ty * ty
  | TForall of string list * ty

type type_scheme = Forall of string list * ty

type substitution = (string * ty) list

type constraint_t = ty * ty

let empty_subst = []

(* Global counter for fresh variable generation *)
let var_counter = ref 0

let fresh_type_var () =
  let id = !var_counter in
  incr var_counter;
  TVar ("_" ^ string_of_int id)
```

Type Operations Implementation

```ocaml
(* type_ops.ml - Core type operations *)

open Ty

let rec type_equal t1 t2 =
  (* TODO 1: Handle primitive types - return true if both are same primitive *)
  (* TODO 2: Handle type variables - return true if variable names are equal *)
  (* TODO 3: Handle function types - recursively check parameter and return types *)
  (* TODO 4: Handle polymorphic types - check variable lists and body types *)
  (* Hint: Use pattern matching on (t1, t2) tuple for exhaustive cases *)
  failwith "implement type_equal"

let rec occurs_check var_name ty =
  (* TODO 1: Return false for primitive types - they contain no variables *)
  (* TODO 2: For TVar, return true if the variable name matches *)
  (* TODO 3: For TFun, recursively check both parameter and return types *)
  (* TODO 4: For TForall, check if var_name is bound, then check body if not *)
  (* Hint: Use List.mem to check if variable is in quantifier list *)
  failwith "implement occurs_check"

let rec apply_subst_to_type subst ty =
  (* TODO 1: Return primitive types unchanged *)
  (* TODO 2: For TVar, lookup in substitution list, return replacement or original *)
  (* TODO 3: For TFun, recursively apply substitution to parameter and return types *)
  (* TODO 4: For TForall, filter substitution to remove bound variables, then apply to body
*)
  (* Hint: Use List.assoc_opt for substitution lookup *)
  (* Hint: Use List.filter to remove bound variables from substitution *)
```

```
  failwith "implement apply_subst_to_type"

let compose_subst s1 s2 =
  (* TODO 1: Apply s2 to all types in the range of s1 *)
  (* TODO 2: Add all bindings from s2 that don't conflict with s1 *)
  (* TODO 3: Return the combined substitution *)
  (* Hint: Use List.map to apply s2 to s1's ranges *)
  (* Hint: Use List.filter to find non-conflicting bindings from s2 *)
  failwith "implement compose_subst"

let string_of_type ty =
  (* TODO 1: Handle primitive types with simple strings *)
  (* TODO 2: Handle variables by returning the variable name *)
  (* TODO 3: Handle functions with proper associativity and parentheses *)
  (* TODO 4: Handle polymorphic types with forall quantifier notation *)
  (* Hint: Function types associate to the right: a -> b -> c means a -> (b -> c) *)
  failwith "implement string_of_type"
```

Infrastructure Starter Code

```ocaml
(* type_printer.ml - Complete pretty printing utilities *)

open Ty

let rec string_of_type_list types =
  String.concat " * " (List.map string_of_type types)

let print_type ty =
  print_endline (string_of_type ty)

let print_substitution subst =
  let print_binding (var, ty) =
    Printf.printf "%s := %s\n" var (string_of_type ty)
  in
  List.iter print_binding subst

let print_constraints constraints =
  let print_constraint (t1, t2) =
    Printf.printf "%s ~ %s\n" (string_of_type t1) (string_of_type t2)
  in
  List.iter print_constraint constraints

(* Debugging utilities *)
let debug_type_equal t1 t2 =
  let result = type_equal t1 t2 in
  Printf.printf "type_equal(%s, %s) = %b\n"
    (string_of_type t1) (string_of_type t2) result;
  result
```

```
let debug_occurs_check var ty =
  let result = occurs_check var ty in
  Printf.printf "occurs_check(%s, %s) = %b\n"
    var (string_of_type ty) result;
  result
```

Milestone Checkpoint

After implementing the type representation system, verify your implementation with these tests:

Basic Type Operations:

```ocaml
(* test_basic_types.ml *)

let test_primitive_equality () =

  assert (type_equal TInt TInt);

  assert (type_equal TBool TBool);

  assert (not (type_equal TInt TBool));

  print_endline "√ Primitive type equality works"


let test_function_types () =

  let int_to_bool = TFun(TInt, TBool) in

  let int_to_bool2 = TFun(TInt, TBool) in

  let bool_to_int = TFun(TBool, TInt) in

  assert (type_equal int_to_bool int_to_bool2);

  assert (not (type_equal int_to_bool bool_to_int));

  print_endline "√ Function type equality works"


let test_fresh_variables () =

  let v1 = fresh_type_var () in

  let v2 = fresh_type_var () in

  assert (not (type_equal v1 v2));

  print_endline "√ Fresh variable generation works"
```

Occurs Check Testing:

```ocaml
let test_occurs_check () =
  let var_a = TVar "a" in
  let int_type = TInt in
  let recursive_type = TFun(var_a, TInt) in

  assert (not (occurs_check "a" int_type));
  assert (occurs_check "a" var_a);
  assert (occurs_check "a" recursive_type);
  print_endline "√ Occurs check prevents infinite types"
```

Expected Output: Run `ocamlfind ocamlc -package ... -o test_types test_types.ml && ./test_types` Should see:

```
√ Primitive type equality works
√ Function type equality works
√ Fresh variable generation works
√ Occurs check prevents infinite types
```

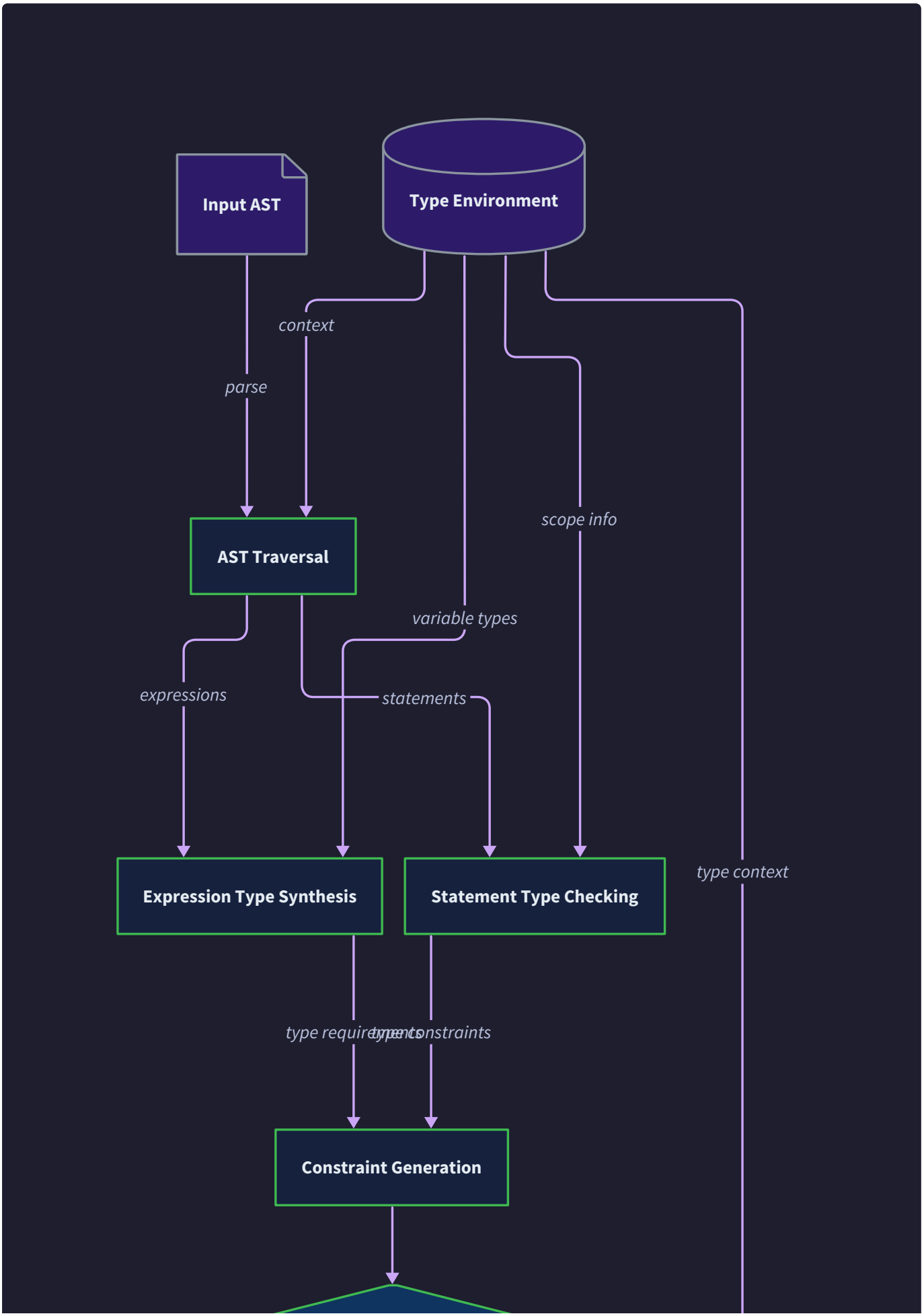
Debugging Tips:

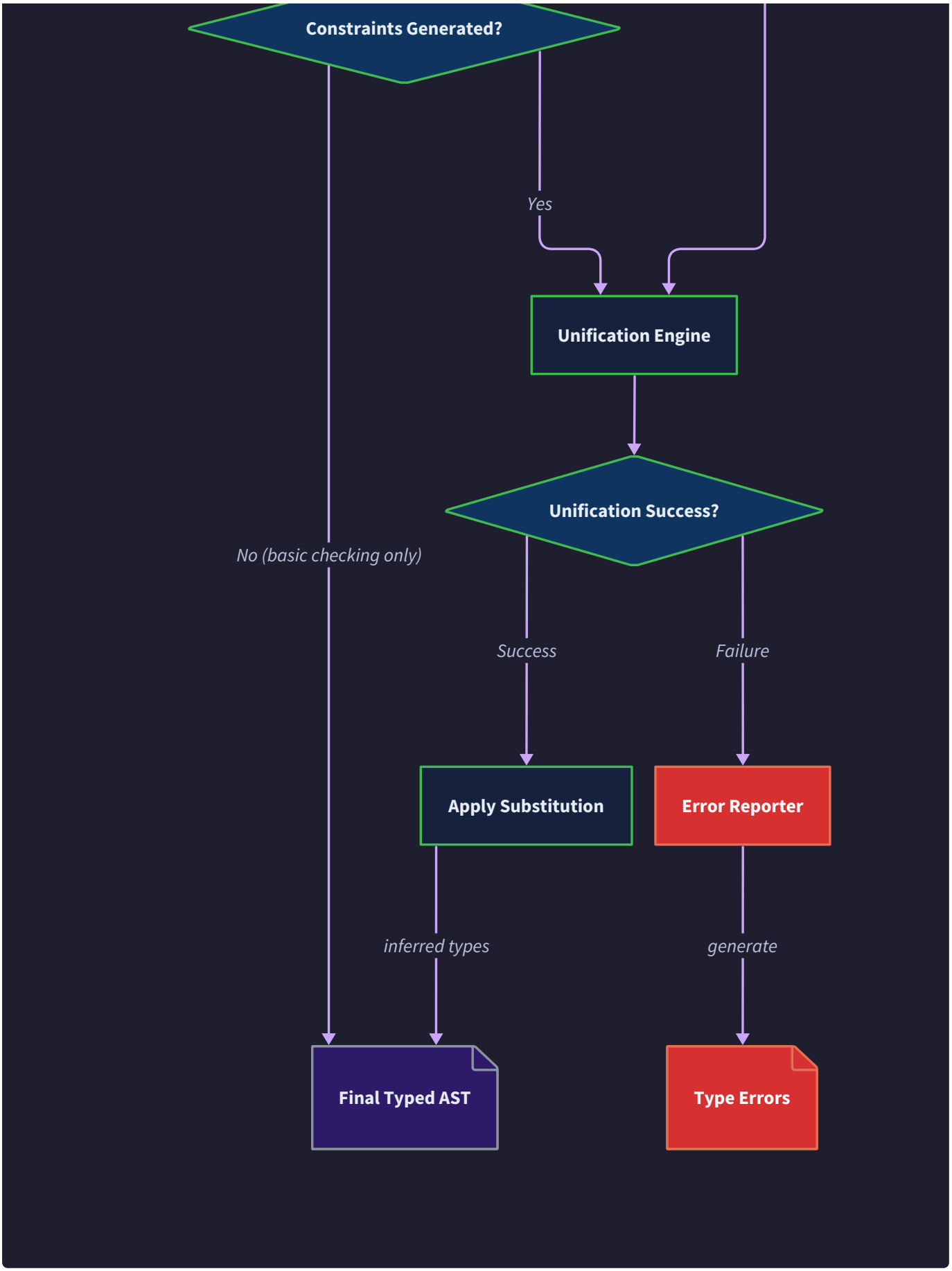
Symptom	Likely Cause	How to Diagnose	Fix
Stack overflow in type_equal	Infinite type created without occurs check	Add debug prints to see types being compared	Implement occurs_check correctly
Variables unify incorrectly	Fresh variable generator reusing names	Print generated variable names	Check var_counter is properly incremented
Substitution not working	apply_subst_to_type not handling all cases	Print substitution and original type	Add missing pattern match cases
Type printing looks wrong	Missing parentheses or precedence	Compare with expected mathematical notation	Add precedence rules to string_of_type

Basic Type Checking Engine

> **Milestone(s):** Milestone 2 (Basic Type Checking)

Think of the basic type checking engine as a **strict contract enforcement system** — much like how a building inspector verifies that construction follows building codes before issuing permits. Our type checker examines each expression and statement in the program, ensuring that all operations are performed on compatible types according to the language's type rules. Unlike inference (which we'll add later), basic type checking assumes that most types are explicitly declared and focuses on verification rather than deduction.


Input AST
Type Environment
context
parse
AST Traversal
scope info
variable types
expressions
statements
Expression Type Synthesis
Statement Type Checking
type context
type requirements
type constraints
Constraint Generation



The basic type checking engine operates on the principle of **bidirectional type flow**. For each expression, we either know what type we expect (checking mode) or we need to determine what type it produces (synthesis

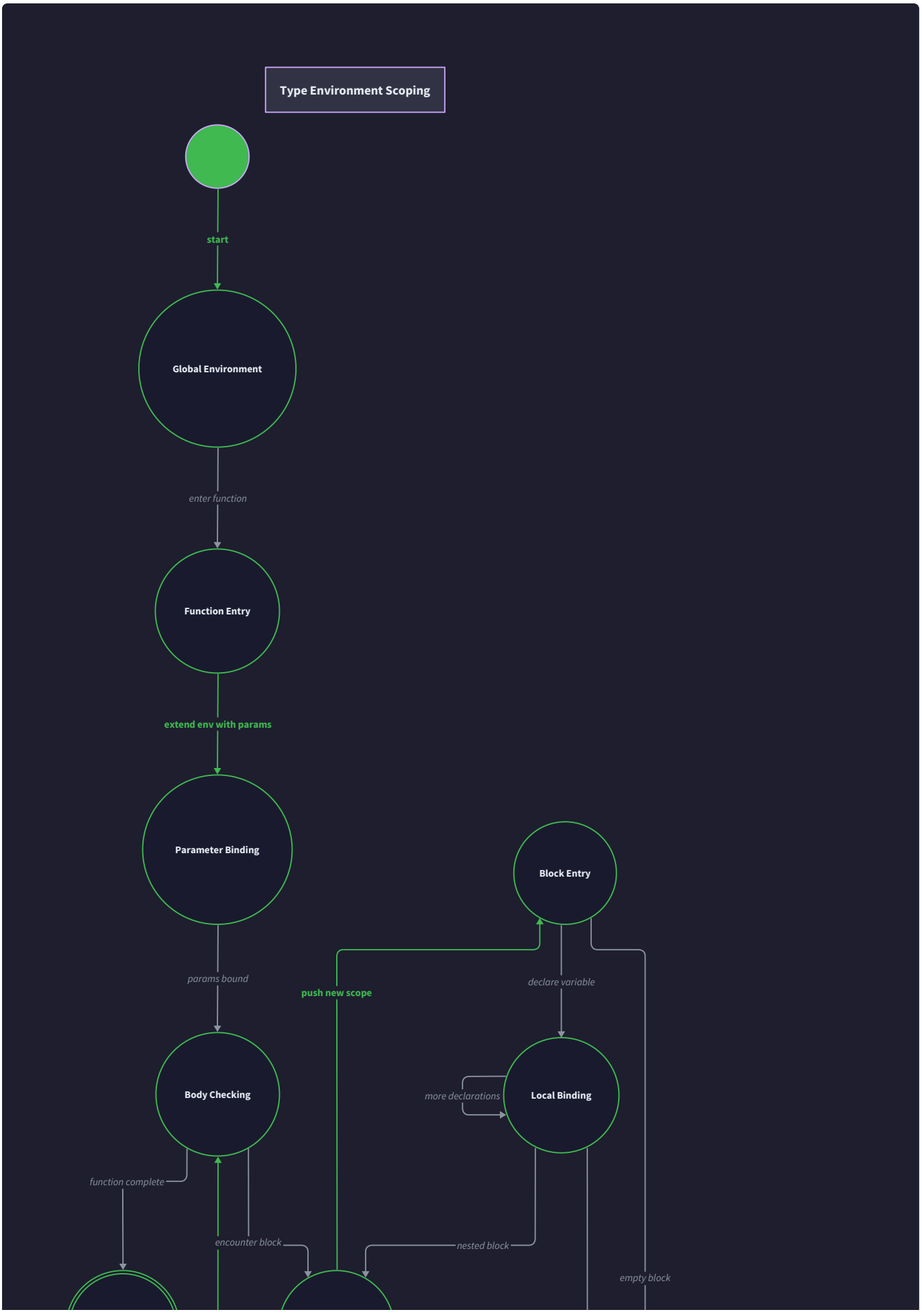
mode). This dual approach allows us to provide precise error messages while maintaining algorithmic simplicity. The engine traverses the AST recursively, maintaining a type environment that tracks the types of all variables in scope.

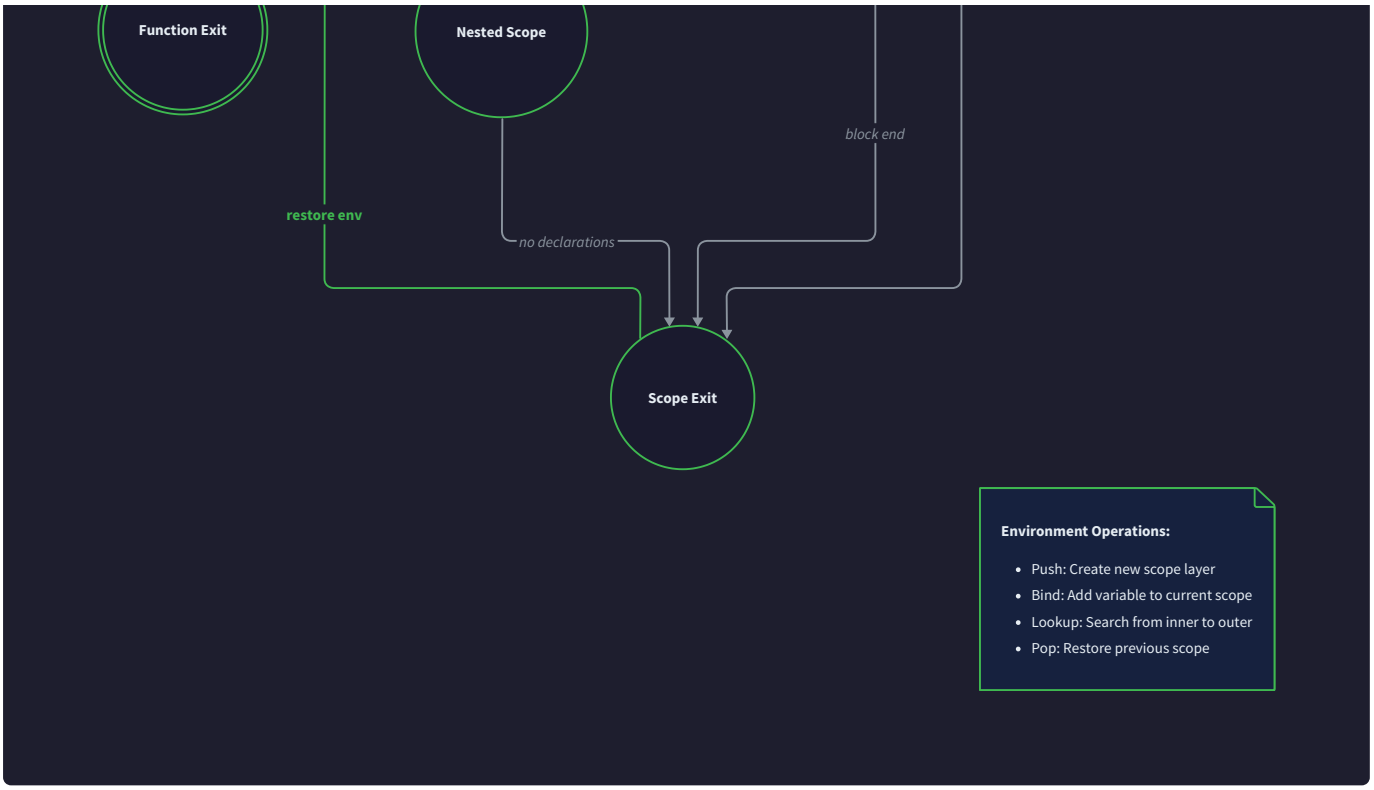
The architecture separates three distinct responsibilities: **expression type synthesis** (determining what type an expression produces), **statement type checking** (verifying that statements respect type constraints), and **error reporting** (providing actionable feedback when type rules are violated). This separation allows each component to focus on its specific concerns while maintaining clean interfaces between them.

> **Decision: Separate Expression and Statement Type Checking**
>
> - **Context**: Programming languages distinguish between expressions (which produce values) and statements (which perform actions), and they have different type checking requirements
> - **Options Considered**: 1) Unified checking function for all AST nodes, 2) Separate functions for expressions vs statements, 3) Visitor pattern with type-specific methods
> - **Decision**: Separate functions with dedicated logic for expressions and statements
> - **Rationale**: Expressions always produce a type that can be synthesized, while statements often don't produce values and need compatibility checking instead. This separation makes the type rules clearer and error messages more specific.
> - **Consequences**: Cleaner code organization and more precise error reporting, but requires careful coordination between expression and statement checking phases

Checking Mode	Purpose	Input	Output	Example
Expression Synthesis	Determine type produced	Expression AST	Inferred type	`42` → `TInt`
Expression Checking	Verify expected type	Expression AST + Expected type	Success/failure	`true` against `TBool`
Statement Checking	Verify type constraints	Statement AST	Success/failure	Assignment compatibility
Declaration Processing	Add new bindings	Declaration AST	Updated environment	`let x: int = 5`

The type checking engine maintains **lexical scoping** through a hierarchical type environment that mirrors the program's block structure. When entering a new scope (function body, block statement, etc.), we extend the environment with new bindings. When exiting the scope, we restore the previous environment state. This ensures that variable lookups respect the program's scoping rules and that inner declarations shadow outer ones appropriately.


Type Environment Scoping
start
Global Environment
enter function
Function Entry
extend env with params
Parameter Binding
params bound
Body Checking
function complete
encounter block
push new scope
Block Entry
declare variable
Local Binding
more declarations
nested block
empty block



Expression Type Rules

Expression type checking forms the foundation of our type system, implementing the formal type rules that govern how expressions combine to form well-typed programs. Think of expression type rules as **grammatical rules for a mathematical language** — just as mathematical notation has rules about what operations can be applied to what kinds of objects (you can't take the square root of a matrix without additional structure), our type system defines what operations are valid on what types.

Each expression type rule follows a standard format: given the types of the sub-expressions and the operation being performed, determine either the resulting type (for synthesis) or whether the expression is well-typed for an expected type (for checking). These rules are compositional — the type of a complex expression depends only on the types of its immediate sub-expressions, not on their internal structure.

Literal Expression Rules

Literal expressions are the base case of our type system — they have known types that can be determined directly from their syntactic form. These rules are straightforward but establish the foundation for all other type checking.

Literal Type	Syntax Form	Synthesized Type	Type Rule
Integer	`42`, `-17`, `0`	`TInt`	All integer literals have type `TInt`
Boolean	`true`, `false`	`TBool`	All boolean literals have type `TBool`
String	`"hello"`, `""`	`TString`	All string literals have type `TString`
Unit	`()`	`TUnit`	Unit literal has type `TUnit`

The literal type checking algorithm is deterministic and never fails — every well-formed literal in our language has a corresponding type. However, the type checker must handle malformed literals (detected during parsing) by reporting appropriate errors rather than crashing.

Variable Reference Rules

Variable references require environment lookup to determine their types. The type rule states that a variable reference `x` has type `T` if and only if the variable `x` is bound to type `T` in the current type environment. This implements lexical scoping — we search environments from innermost to outermost scope.

The variable lookup process follows these steps:

1. **Environment Search**: Starting from the innermost scope, search each environment level for a binding of the variable name
2. **Binding Verification**: If found, return the associated type from the binding
3. **Scope Traversal**: If not found in current scope, move to the next outer scope and repeat
4. **Error Reporting**: If no binding is found in any scope, report an unbound variable error

Variable State	Environment Binding	Type Result	Error Condition
Bound in current scope	`x: TInt` in innermost env	`TInt`	None
Bound in outer scope	`x: TBool` in parent env	`TBool`	None
Shadowed variable	Inner: `x: TInt`, Outer: `x: TBool`	`TInt`	None (inner shadows outer)
Unbound variable	No binding found	Type error	"Unbound variable: x"

> The critical insight for variable lookup is that **shadowing is a feature, not a bug**. When an inner scope declares a variable with the same name as an outer scope, the inner declaration should hide the outer one completely. This requires searching environments in inside-out order and stopping at the first match.

Binary Operator Rules

Binary operators implement the most complex expression type rules because they must check the compatibility of two operand types and determine the result type based on the specific operator. Different operators have different type requirements — arithmetic operators require numeric types, comparison operators accept multiple types but return boolean, and logical operators work only on booleans.

The binary operator type checking algorithm follows this pattern:

1. **Operand Type Synthesis**: Recursively determine the types of both operands
2. **Operator Lookup**: Determine what type constraints the operator imposes
3. **Compatibility Check**: Verify that both operand types satisfy the operator's requirements
4. **Result Type Determination**: Based on the operator and operand types, determine the result type

Operator Category	Operators	Operand Types Required	Result Type	Example
Arithmetic	`+`, `-`, `*`, `/`, `%`	Both `TInt`	`TInt`	`5 + 3` → `TInt`
Comparison	`==`, `!=`, `<`, `<=`, `>`, `>=`	Both same type	`TBool`	`5 < 3` → `TBool`
Logical	`&&`, `||`	Both `TBool`	`TBool`	`true && false` → `TBool`
String Operations	`++` (concatenation)	Both `TString`	`TString`	`"hello" ++ "world"` → `TString`

The comparison operators deserve special attention because they implement **polymorphic equality** — they can compare values of any type as long as both operands have the same type. This requires the type checker to verify type equality between operands rather than checking against a fixed type.

> **Decision: Polymorphic vs Monomorphic Operators**
>
> - **Context**: Comparison operators like `==` could either work only on specific types or accept any type as long as both sides match
> - **Options Considered**: 1) Monomorphic — only allow comparison on primitive types, 2) Polymorphic — allow comparison on any equal types, 3) Type class based — require explicit equality implementation
> - **Decision**: Polymorphic comparison with structural equality
> - **Rationale**: Provides more flexibility for users while maintaining type safety. Most values in our language support structural comparison naturally.
> - **Consequences**: Simplifies user code and reduces type annotations, but requires more complex type checking logic for equality operators

Function Call Rules

Function call type checking implements the **function application rule** from type theory — if we have a function of type `T1 → T2` and an argument of type `T1`, then the application produces a result of type `T2`. This rule ensures that function calls respect the contracts established by function type signatures.

The function call type checking algorithm performs these validations:

1. **Function Type Synthesis**: Determine the type of the function expression being called
2. **Function Type Verification**: Ensure the function type is actually a function type `TFun(param_ty, return_ty)`
3. **Argument Type Synthesis**: Determine the type of each argument expression
4. **Arity Check**: Verify that the number of arguments matches the function's parameter count

5. **Parameter Compatibility**: Check that each argument type is compatible with the corresponding parameter type
6. **Return Type Synthesis**: The call expression has the function's return type

Validation Step	Success Condition	Failure Condition	Error Message
Function type check	Function has type `TFun(T1, T2)`	Function has non-function type	"Cannot call non-function of type T"
Arity check	Argument count = Parameter count	Mismatched counts	"Expected N arguments, got M"
Parameter compatibility	Each arg type matches param type	Type mismatch	"Expected T1, got T2 for parameter N"

For functions with multiple parameters, our type representation uses **curried function types** — a function taking two parameters has type `TFun(T1, TFun(T2, T3))` rather than a special multi-parameter type. This simplifies the type system and enables partial application, but requires careful handling during type checking to provide good error messages for multi-parameter calls.

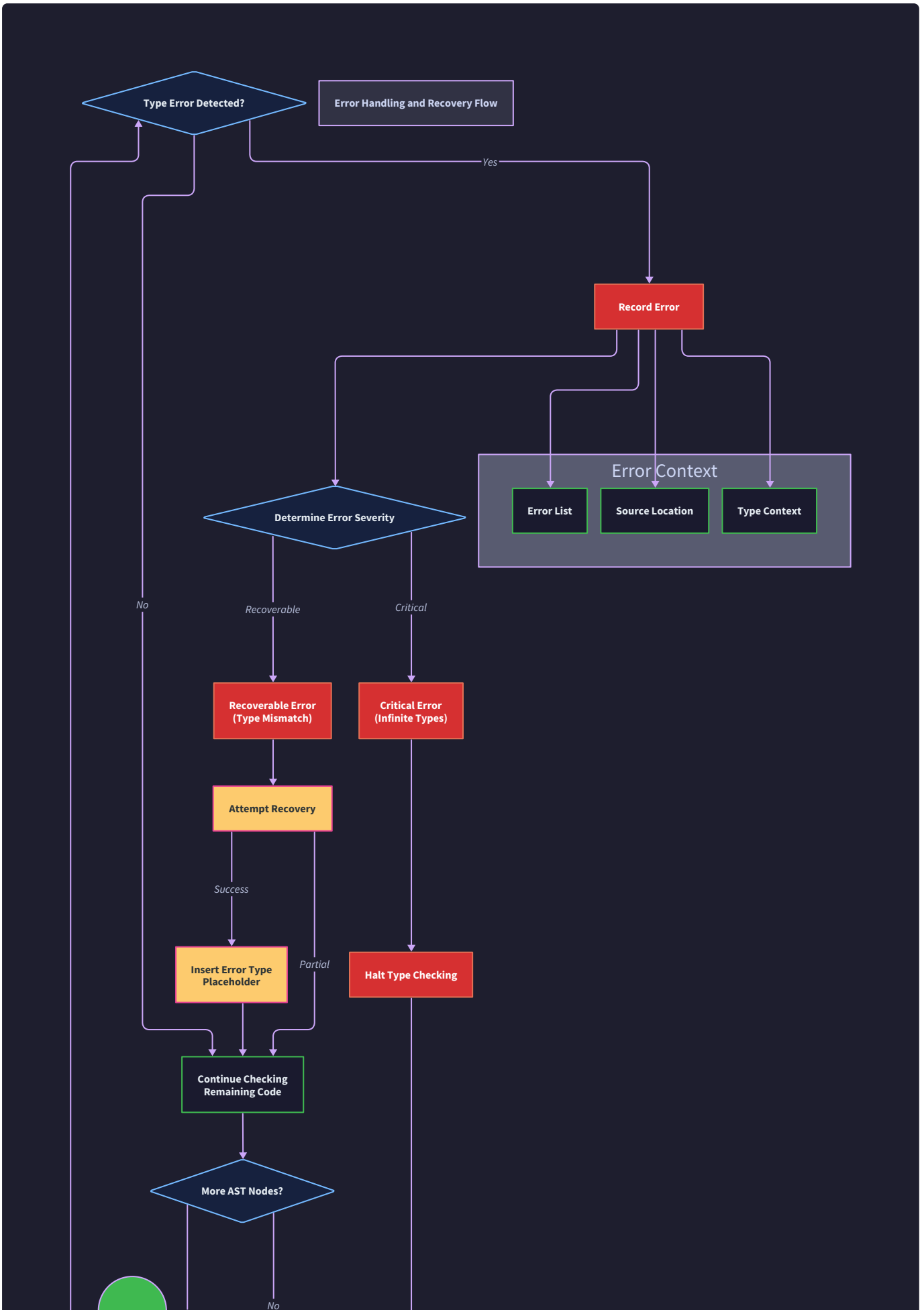
⚠️ Pitfall: Confusing Function Types and Function Values

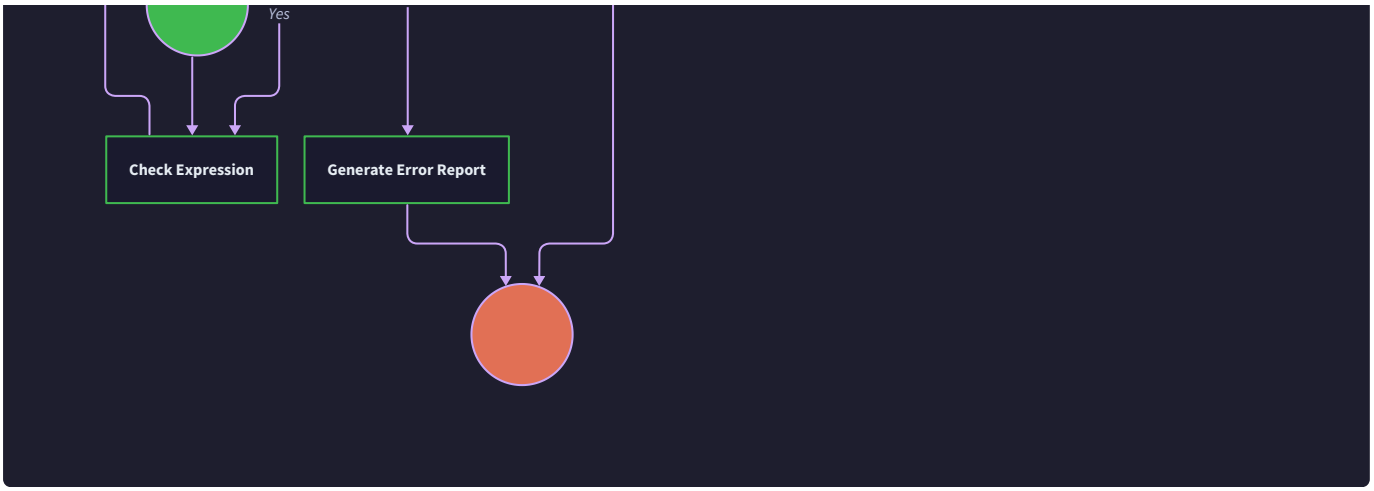
A common mistake is trying to call something that has a function type in the source code but doesn't actually represent a callable function at runtime. For example, a variable declared as `let f: int -> int` but never assigned a value should produce a "unbound variable" error at the variable lookup stage, not a "cannot call" error at the function call stage. Always check that the function expression is well-typed before validating the call.

Statement Type Checking

Statement type checking differs fundamentally from expression type checking because statements perform actions rather than producing values. Think of statement type checking as **auditing a series of commands** rather than evaluating mathematical expressions. While expressions ask "what type does this produce?", statements ask "is this action permitted given the current type constraints?"

Most statements don't have types themselves, but they impose type constraints on their constituent expressions. A variable assignment requires that the right-hand side expression's type is compatible with the variable's declared type. A conditional statement requires that the condition expression has boolean type. These constraints ensure that the program's runtime behavior will respect the static type declarations.


Error Handling and Recovery Flow
Type Error Detected?
Yes
No
Record Error
Error Context
Error List
Source Location
Type Context
Determine Error Severity
Recoverable
Critical
Recoverable Error (Type Mismatch)
Critical Error (Infinite Types)
Attempt Recovery
Success
Partial
Insert Error Type Placeholder
Halt Type Checking
Continue Checking Remaining Code
More AST Nodes?
No



Variable Declaration Type Checking

Variable declarations establish new bindings in the type environment, connecting variable names to their types. The type checking process must handle both explicit type annotations (`let x: int = 42`) and cases where the type should be inferred from the initializer (`let x = 42`). In basic type checking mode, we primarily handle explicitly annotated declarations.

The variable declaration type checking algorithm:

1. **Type Annotation Processing**: If present, parse and validate the explicit type annotation
2. **Initializer Type Synthesis**: Determine the type of the initialization expression
3. **Type Compatibility Check**: Verify that the initializer type is compatible with the declared type (if annotated)
4. **Type Assignment**: If no explicit annotation, use the initializer type as the variable's type
5. **Environment Extension**: Add the new variable binding to the current scope's environment
6. **Shadowing Check**: Optionally warn if the new variable shadows an existing binding

Declaration Form	Type Source	Compatibility Check	Environment Update
`let x: int = 42`	Explicit annotation `int`	Check `42: int` ✓	Add `x: TInt`
`let x: int = true`	Explicit annotation `int`	Check `true: int` ✗	Type error
`let x = 42`	Initializer synthesis	No check needed	Add `x: TInt`
`let x: int`	Explicit annotation	No initializer	Add `x: TInt`

The compatibility check between declared type and initializer type uses the same type equality function used throughout the type checker. In basic type checking, we require exact type matches — `TInt` is only compatible with `TInt` . Later, when we add subtyping, we'll relax this to allow more flexible assignments.

Decision: Exact Type Equality vs Subtyping

- **Context**: When checking assignment compatibility, we could require exact type matches or allow subtype relationships
- **Options Considered**: 1) Exact equality only, 2) Structural subtyping, 3) Nominal subtyping with explicit inheritance
- **Decision**: Start with exact equality, add subtyping in later milestones
- **Rationale**: Exact equality is simpler to implement and reason about. It provides a solid foundation for understanding type checking before introducing the complexity of subtype relationships.
- **Consequences**: More restrictive type system initially, but clearer error messages and simpler implementation. Users must be more explicit about type conversions.

Assignment Statement Type Checking

Assignment statements update existing variable bindings with new values. Unlike declarations, assignments cannot change a variable's type — they must respect the type established when the variable was declared. This maintains type consistency throughout the program and enables optimizations based on stable type information.

The assignment type checking process:

1. **Target Variable Lookup**: Find the variable in the type environment to determine its declared type
2. **Value Expression Synthesis**: Determine the type of the right-hand side expression
3. **Assignment Compatibility**: Verify that the value type is compatible with the variable type
4. **Mutability Check**: Ensure the variable is declared as mutable (if the language distinguishes mutable/immutable)

Assignment Scenario	Variable Type	Value Type	Compatibility Result
`x = 42` where `x: int`	`TInt`	`TInt`	✓ Compatible
`x = true` where `x: int`	`TInt`	`TBool`	✗ Type mismatch
`y = 42` where `y` unbound	Error	`TInt`	✗ Unbound variable
`x = f()` where `f: () -> int`	`TInt`	`TInt`	✓ Compatible

The assignment compatibility check is crucial for maintaining type safety. If we allowed assignments that change a variable's type, then other parts of the program that depend on the variable's declared type could fail at runtime. By enforcing type consistency, we guarantee that a variable declared as `int` will always contain an integer value.

⚠️ Pitfall: Mutable vs Immutable Variable Confusion

Many functional languages distinguish between mutable variables (which can be reassigned) and immutable bindings (which cannot). Assignment type checking should verify mutability in addition to type compatibility. A common mistake is allowing assignment to immutable bindings just because the types match. Always check the variable's mutability status before performing assignment compatibility checks.

Conditional Statement Type Checking

Conditional statements (if-then-else) impose type constraints on their condition expression and recursively type-check their branch statements. The key insight is that conditional statements create **branching control flow** where different execution paths may have different type environments.

The conditional type checking algorithm:

1. **Condition Type Check**: Verify that the condition expression has type `TBool`
2. **Then Branch Scoping**: Create a new scope for the then branch and type-check all statements
3. **Else Branch Scoping**: Create a new scope for the else branch (if present) and type-check all statements
4. **Environment Restoration**: Restore the original environment after checking both branches
5. **Return Type Analysis**: If branches contain return statements, verify they have compatible return types

Conditional Component	Type Requirement	Error Condition	Scope Behavior
Condition expression	Must be `TBool`	Non-boolean condition	Evaluated in current scope
Then branch	Statements must type-check	Any statement type error	New nested scope
Else branch	Statements must type-check	Any statement type error	New nested scope
Overall statement	No return type	Branches have incompatible returns	No type produced

The scoping behavior of conditional statements is important for maintaining proper variable visibility. Variables declared inside a conditional branch should not be visible outside that branch, even if the branch is guaranteed to execute. This prevents the type checker from making assumptions about runtime execution paths.

Function Declaration Type Checking

Function declarations are among the most complex statements to type-check because they introduce **recursive environments** where the function being defined may appear in its own body. Additionally, function declarations must establish parameter bindings, check return type consistency, and handle the interaction between declared and inferred return types.

The function declaration type checking process:

1. **Parameter Processing**: Add parameter bindings to a new function scope environment
2. **Return Type Annotation**: Process explicit return type annotation if present
3. **Recursive Binding**: Add the function name to its own environment for recursive calls
4. **Body Type Checking**: Type-check all statements in the function body within the extended environment
5. **Return Type Verification**: Ensure all return statements produce types compatible with the declared return type
6. **Function Type Construction**: Build the complete function type from parameter and return types
7. **Global Environment Update**: Add the function binding to the module-level environment

Function Component	Type Processing	Environment Impact	Validation Required
Parameter list	`(x: int, y: bool)` → `[TInt, TBool]`	Add to function scope	Parameter name uniqueness
Return annotation	`: string` → `TString`	Used for body validation	Must be valid type
Function body	Statement sequence	Function scope active	All statements must type-check
Return statements	`return expr`	Current scope	`expr` type matches return type

The recursive binding aspect requires careful handling to avoid infinite loops during type checking. When processing a recursive function call within the function body, we must use the function type being constructed rather than attempting to re-derive it. This ensures that recursive calls are type-checked against the function's declared signature rather than creating circular dependencies.

> The key insight for function declaration type checking is **environment layering**. The function creates multiple nested scopes: the module scope (where the function name will be bound), the function scope (where parameters are bound), and potentially additional scopes for nested blocks within the function body. Each scope layer must be managed correctly to ensure proper variable resolution.

Type Error Reporting

Type error reporting transforms the technical results of type checking into actionable feedback for programmers. Think of error reporting as **translating between two languages** — converting the formal language of type theory (where errors are constraint violations) into the natural language of human communication (where errors explain what went wrong and how to fix it).

Effective type error reporting requires more than simply stating that a type check failed. It must provide context about what the type checker expected, what it actually found, where in the source code the error occurred,

and suggestions for how to resolve the mismatch. The best error messages help programmers build better mental models of the type system while fixing immediate problems.

Error Message Structure

Well-structured error messages follow a consistent format that presents information in order of decreasing specificity. The message should start with the immediate problem, provide context about why it's a problem, show the specific location where it occurred, and offer concrete suggestions for resolution.

Message Component	Purpose	Content	Example
Error Classification	Immediate problem identification	Type of error in standardized format	"Type mismatch in assignment"
Expected vs Actual	Specific type incompatibility	What was expected and what was found	"Expected: int, Found: bool"
Source Location	Precise error position	File, line, column, and code context	"at line 15, column 8: `x = true`"
Contextual Information	Why this matters	Surrounding code or declaration info	"Variable x declared as int at line 10"
Suggested Fixes	Actionable remediation	Specific changes that would resolve error	"Change to `x = 42` or redeclare x as bool"

The error classification helps programmers quickly understand the category of problem they're dealing with. Common classifications include "type mismatch", "unbound variable", "wrong arity", "non-function call", and "invalid operation". These categories map directly to the different type checking rules and help users understand which aspect of the type system they've violated.

Source Location Tracking

Accurate source location reporting requires the type checker to maintain **position information** throughout the type checking process. Every AST node should carry information about its location in the original source code, and this information must be preserved and propagated through all type checking operations.

The location tracking system maintains several pieces of information:

1. **File Information**: Which source file contains the problematic code
2. **Line and Column Numbers**: Precise position within the file using 1-based indexing
3. **Span Information**: Start and end positions for multi-character constructs
4. **Context Lines**: Surrounding source code lines for better error visualization
5. **AST Node Mapping**: Connection between error locations and specific AST constructs

Location Component	Data Structure	Usage	Example
File path	`string`	Error message header	"In file: src/main.ml"
Line number	`int`	Error positioning	"Line 42:"
Column number	`int`	Precise character position	"Column 15:"
Source span	`(int * int) * (int * int)`	Multi-token highlighting	"Lines 10-15, columns 5-20"
Context buffer	`string list`	Code visualization	Show 3 lines before/after error

When reporting errors, the system should highlight the specific tokens or expressions involved in the type error. For binary operators, this might highlight both operands and the operator. For function calls, it might highlight the function name and the problematic arguments. This visual emphasis helps programmers quickly locate the source of type problems.

> **Decision: Precise vs Approximate Source Positions**
>
> - **Context**: Source location tracking can be implemented with exact character positions or approximate line-level positions
> - **Options Considered**: 1) Line-only tracking with minimal overhead, 2) Precise character positions with full span information, 3) Token-level positions with AST node mapping
> - **Decision**: Precise character positions with span information
> - **Rationale**: Modern programmers expect IDE-quality error reporting with exact positioning. The additional implementation complexity pays off in user experience, especially for complex expressions where approximate positions are confusing.
> - **Consequences**: More complex AST representation and error tracking, but significantly better user experience and integration with development tools

Context-Aware Error Messages

Context-aware error messages go beyond reporting the immediate type mismatch to explain **why the types are incompatible** and **what the programmer likely intended**. This requires the error reporting system to understand common programming patterns and frequent mistakes.

The context analysis system examines several factors when generating error messages:

1. **Declaration Context**: How and where relevant variables were declared
2. **Usage Patterns**: How the problematic expression fits into the larger code structure

3. **Common Mistakes**: Whether the error matches known anti-patterns or misconceptions
4. **Similar Names**: Whether there are similar variable names that might have been intended
5. **Type Relationships**: Whether the types are "close" in some sense (like `int` and `float`)

Context Type	Analysis Performed	Enhanced Error Information	Example Enhancement
Variable shadowing	Check for same name in outer scopes	"Variable x shadows outer declaration"	Suggest using different names
Similar names	Edit distance on variable names	"Did you mean variable y?"	Offer spelling corrections
Common conversions	Known type conversion patterns	"Use int_of_bool to convert"	Suggest explicit conversions
Scope confusion	Variable declared in unreachable scope	"Variable x not in scope here"	Explain scoping rules

For function call errors, context-aware messages can provide especially valuable feedback. Instead of simply reporting "expected 2 arguments, got 3", the error message can show the function's declaration, explain which arguments are problematic, and suggest corrections. This helps programmers understand not just what went wrong, but how to fix it.

Error Recovery and Continuation

Error recovery allows the type checker to continue processing after encountering a type error, enabling it to **find multiple errors in a single pass** rather than stopping at the first problem. This significantly improves the development experience by reducing the number of compile-fix-compile cycles required to resolve type issues.

The error recovery strategy depends on the type of error encountered:

1. **Expression Errors**: Replace the problematic expression with a placeholder type and continue
2. **Variable Errors**: Add a placeholder binding for unbound variables and continue
3. **Statement Errors**: Skip the problematic statement and continue with the next statement
4. **Declaration Errors**: Use a default type for problematic declarations and continue
5. **Function Errors**: Use placeholder types for function signatures and continue checking the body

Error Category	Recovery Strategy	Placeholder Choice	Impact on Later Checking
Expression type mismatch	Use expected type as placeholder	Expected type from context	Prevents cascading errors
Unbound variable	Create binding with fresh type variable	Fresh type variable	Enables partial checking
Wrong function arity	Assume correct arity with placeholder args	Unit type for missing args	Allows return type checking
Invalid operation	Replace with well-typed equivalent	Result type of similar operation	Maintains type flow

Error recovery must be careful not to generate **cascading errors** — situations where one error causes many additional spurious errors. The placeholder types chosen during recovery should be selected to minimize disruption to subsequent type checking phases. For example, when an expression has the wrong type, using the expected type as a placeholder often prevents errors in surrounding code that depends on that type.

⚠️ Pitfall: Over-Aggressive Error Recovery

While error recovery is valuable, over-aggressive recovery can hide real errors or generate confusing error messages. If the recovery strategy makes too many assumptions about what the programmer intended, the subsequent error messages may not reflect the actual problems in the code. Balance error recovery with accuracy — recover enough to find additional errors, but not so much that the error messages become misleading.

Error Message Formatting and Presentation

Error message formatting significantly impacts how effectively programmers can understand and resolve type errors. The presentation should use consistent formatting, clear visual hierarchy, and appropriate use of color and highlighting to guide attention to the most important information.

Modern error message formatting follows several principles:

1. **Visual Hierarchy**: Use formatting to emphasize the most important information first
2. **Consistent Structure**: Follow the same format pattern for similar types of errors
3. **Code Context**: Show relevant source code with highlighting and annotations
4. **Progressive Disclosure**: Present basic information first, with details available on request
5. **Actionable Language**: Use imperative mood and specific suggestions rather than passive descriptions

Formatting Element	Purpose	Implementation	Example
Error header	Quick problem identification	Bold, colored text	`error: type mismatch in assignment`
Location info	Source positioning	File:line:col format	`--> src/main.ml:15:8`
Code display	Visual context	Syntax highlighting + annotations	Show source with error markers
Type information	Specific type details	Monospace font, clear labels	`expected: int, found: bool`
Suggestions	Remediation guidance	Bulleted list, specific actions	`• change true to 42`

The code context display should show enough surrounding lines to understand the error's context while avoiding information overload. Typically, showing 2-3 lines before and after the error location provides good context. The error location itself should be clearly marked with visual indicators like underlines, arrows, or highlighting.

Implementation Guidance

The basic type checking engine requires careful coordination between expression synthesis, statement checking, and error reporting. The implementation should prioritize clarity and correctness over performance optimization — type checking is typically not a performance bottleneck compared to parsing or code generation.

Technology Recommendations

Component	Simple Option	Advanced Option
Error Reporting	String concatenation with manual formatting	Structured error types with pretty-printing
Source Locations	Line/column pairs in AST nodes	Full span information with source maps
Type Environment	List of association lists for scopes	Hash tables with scope chain pointers
Error Recovery	Stop at first error	Continue with placeholder types

Recommended File Structure

```
src/
  type_checker/
    type_check.ml           ← main type checking entry point
    expression_check.ml     ← expression type synthesis/checking
    statement_check.ml      ← statement type checking
    type_error.ml          ← error types and reporting
    type_env.ml           ← type environment operations
    test/
      test_expressions.ml   ← expression type rule tests
      test_statements.ml    ← statement type checking tests
      test_errors.ml       ← error message tests
```

Core Type Checking Infrastructure

OCAML

```ocaml
(* Type checking result with error accumulation *)
type check_result =
  | Success of ty
  | Error of type_error list

type type_error = {
  error_kind: error_kind;
  location: source_location;
  expected_type: ty option;
  actual_type: ty option;
  message: string;
  suggestions: string list;
}

type error_kind =
  | TypeMismatch
  | UnboundVariable of string
  | WrongArity of int * int  (* expected, actual *)
  | NonFunctionCall
  | InvalidOperation of string

type source_location = {
  file: string;
  line: int;
  column: int;
  end_line: int;
  end_column: int;
}
```

```
(* Helper functions for error reporting - COMPLETE IMPLEMENTATION *)

let make_error kind loc expected actual msg suggestions = {

  error_kind = kind;

  location = loc;

  expected_type = expected;

  actual_type = actual;

  message = msg;

  suggestions = suggestions;

}


let format_error err =

  let kind_str = match err.error_kind with

    | TypeMismatch -> "type mismatch"

    | UnboundVariable name -> "unbound variable: " ^ name

    | WrongArity (exp, act) -> sprintf "wrong arity: expected %d, got %d" exp act

    | NonFunctionCall -> "cannot call non-function"

    | InvalidOperation op -> "invalid operation: " ^ op

  in

  let loc_str = sprintf "%s:%d:%d" err.location.file err.location.line err.location.column
in

  let type_str = match err.expected_type, err.actual_type with

    | Some exp, Some act -> sprintf "\n  expected: %s\n  found: %s" (string_of_type exp)
(string_of_type act)

    | _ -> ""

  in

  let suggestions_str = match err.suggestions with

    | [] -> ""

    | suggs -> "\n  suggestions:\n" ^ (String.concat "\n" (List.map (fun s -> "    • " ^ s)
suggs))
```

```
  in

  sprintf "error: %s\n --> %s\n%s%s%s" kind_str loc_str err.message type_str
suggestions_str
```

Expression Type Checking Skeleton

```ocaml
(* Expression type synthesis - returns inferred type or error *)

let rec synthesize_expression_type (expr : expression) (env : TypeEnv.t) : check_result =

  match expr with

  | Literal (lit, loc) ->

      (* TODO 1: Pattern match on literal type (IntLit, BoolLit, StringLit) *)

      (* TODO 2: Return Success with corresponding primitive type (TInt, TBool, TString) *)

      (* TODO 3: Handle any malformed literals with appropriate error *)



  | Variable (name, loc) ->

      (* TODO 1: Use TypeEnv.lookup to find variable binding in environment *)

      (* TODO 2: If found, return Success with the bound type *)

      (* TODO 3: If not found, return Error with UnboundVariable error including
suggestions for similar names *)

      (* TODO 4: Use make_error to create properly formatted error with location *)



  | BinaryOp (op, left, right, loc) ->

      (* TODO 1: Recursively synthesize types for left and right operands *)

      (* TODO 2: Handle any errors from operand type synthesis *)

      (* TODO 3: Look up operator type requirements in operator table *)

      (* TODO 4: Check that both operand types satisfy operator requirements *)

      (* TODO 5: Return the operator's result type on success *)

      (* TODO 6: Generate specific error messages for operator type mismatches *)

      (* Hint: Use helper function check_binary_operator_types *)



  | FunctionCall (func_expr, args, loc) ->

      (* TODO 1: Synthesize type for function expression *)

      (* TODO 2: Verify that function type is actually TFun(param_ty, return_ty) *)
```

```
      (* TODO 3: Check arity - number of arguments matches function expectations *)

      (* TODO 4: Synthesize types for all argument expressions *)

      (* TODO 5: Check each argument type against corresponding parameter type *)

      (* TODO 6: Return function's return type on success *)

      (* TODO 7: Generate appropriate errors for non-function calls, arity mismatches,
parameter type errors *)


(* Expression type checking - verifies expression against expected type *)

let rec check_expression_type (expr : expression) (expected : ty) (env : TypeEnv.t) :
check_result =

  (* TODO 1: Synthesize the actual type of the expression *)

  (* TODO 2: Use type_equal to compare actual type with expected type *)

  (* TODO 3: Return Success with expected type if they match *)

  (* TODO 4: Return Error with TypeMismatch if they don't match *)

  (* TODO 5: Include suggestions for common type conversion functions *)
```

Statement Type Checking Skeleton

```ocaml
(* Statement type checking - verifies statements and returns updated environment *)

let rec check_statement (stmt : statement) (env : TypeEnv.t) : (TypeEnv.t * type_error
list) =

  match stmt with

  | Declaration (name, type_annot, init_expr, loc) ->

      (* TODO 1: Process explicit type annotation if present *)

      (* TODO 2: If no annotation, synthesize type from initializer expression *)

      (* TODO 3: If both annotation and initializer, check compatibility *)

      (* TODO 4: Extend environment with new variable binding *)

      (* TODO 5: Check for variable name conflicts in current scope *)

      (* TODO 6: Return updated environment and any compatibility errors *)



  | Assignment (var_name, value_expr, loc) ->

      (* TODO 1: Look up variable in environment to get declared type *)

      (* TODO 2: Synthesize type of value expression *)

      (* TODO 3: Check that value type is compatible with variable type *)

      (* TODO 4: Return unchanged environment (assignments don't add bindings) *)

      (* TODO 5: Generate errors for unbound variables or type mismatches *)



  | Conditional (cond_expr, then_stmts, else_stmts, loc) ->

      (* TODO 1: Check that condition expression has type TBool *)

      (* TODO 2: Create new scope and check all then-branch statements *)

      (* TODO 3: Create new scope and check all else-branch statements (if present) *)

      (* TODO 4: Collect errors from condition and all branch statements *)

      (* TODO 5: Restore original environment (conditional doesn't affect outer scope) *)

      (* Hint: Use TypeEnv.push_scope and TypeEnv.pop_scope for branch isolation *)
```

```
  | FunctionDeclaration (name, params, return_annot, body, loc) ->

      (* TODO 1: Process parameter list to create parameter type list *)

      (* TODO 2: Create function scope with parameter bindings *)

      (* TODO 3: Add recursive binding for function name (for recursive calls) *)

      (* TODO 4: Check all statements in function body within function scope *)

      (* TODO 5: Verify all return statements are compatible with declared return type *)

      (* TODO 6: Construct complete function type from parameters and return type *)

      (* TODO 7: Add function binding to global environment *)

      (* Hint: Function type should be built using nested TFun constructors *)


(* Helper function for checking multiple statements in sequence *)

let check_statement_sequence (stmts : statement list) (env : TypeEnv.t) : (TypeEnv.t *
type_error list) =

  (* TODO 1: Fold over statement list, threading environment through each check *)

  (* TODO 2: Accumulate all errors from individual statement checks *)

  (* TODO 3: Return final environment and complete error list *)
```

Error Recovery and Reporting

```ocaml
(* Error recovery - continues type checking after errors *)
let recover_from_error (err_kind : error_kind) (expected : ty option) : ty =
  match err_kind, expected with
  | UnboundVariable _, Some exp_ty -> exp_ty  (* Use expected type *)
  | UnboundVariable _, None -> fresh_type_var()  (* Create placeholder *)
  | TypeMismatch, Some exp_ty -> exp_ty  (* Assume correct type *)
  | WrongArity _, _ -> fresh_type_var()  (* Unknown result type *)
  | _ -> TInt  (* Safe default for most contexts *)

(* Suggestion generation for common errors *)
let generate_suggestions (err_kind : error_kind) (env : TypeEnv.t) : string list =
  match err_kind with
  | UnboundVariable name ->
      (* TODO 1: Find variables in environment with similar names (edit distance) *)
      (* TODO 2: Suggest spelling corrections within reasonable distance *)
      (* TODO 3: Suggest importing or declaring the variable *)

  | TypeMismatch ->
      (* TODO 1: Suggest explicit type conversion functions if available *)
      (* TODO 2: Suggest changing the expression to match expected type *)
      (* TODO 3: Suggest changing the type annotation if appropriate *)

  | WrongArity (expected, actual) ->
      (* TODO 1: If too few args, suggest adding default values *)
      (* TODO 2: If too many args, suggest removing extra arguments *)
      (* TODO 3: Suggest checking function documentation *)
```

Milestone Checkpoint

After implementing the basic type checking engine, verify the following behaviors:

1. **Literal Type Checking**: `let x = 42` should infer type `int`, `let y = true` should infer type `bool`
2. **Variable Reference**: Declared variables should be found in scope, undeclared variables should produce "unbound variable" errors with suggestions
3. **Binary Operators**: `5 + 3` should type-check as `int`, `5 + true` should produce type mismatch error
4. **Function Calls**: Calling declared functions with correct arguments should succeed, wrong arity or parameter types should produce specific errors
5. **Assignment Compatibility**: Assigning compatible types should succeed, incompatible assignments should produce clear error messages
6. **Scoping**: Variables should only be visible in their declared scope, inner declarations should shadow outer ones
7. **Error Recovery**: Multiple errors in the same program should be reported, not just the first one

Test with progressively complex programs:

```ocaml
(* Simple literals and variables *)

let x: int = 42

let y: bool = true

let z = x  (* should infer int *)


(* Binary operations *)

let sum = x + 10  (* should work *)

let bad = x + true  (* should error *)


(* Function calls *)

let add (a: int) (b: int) : int = a + b

let result = add 5 10  (* should work *)

let wrong = add 5 true 10  (* should error: wrong arity and type *)


(* Scoping *)

let outer = 1

let test () =

  let inner = 2

  let outer = 3  (* shadows outer binding *)

  inner + outer  (* should use inner values *)
```

Expected type checker output should show clear error messages with source locations, expected vs actual types, and actionable suggestions for fixes.

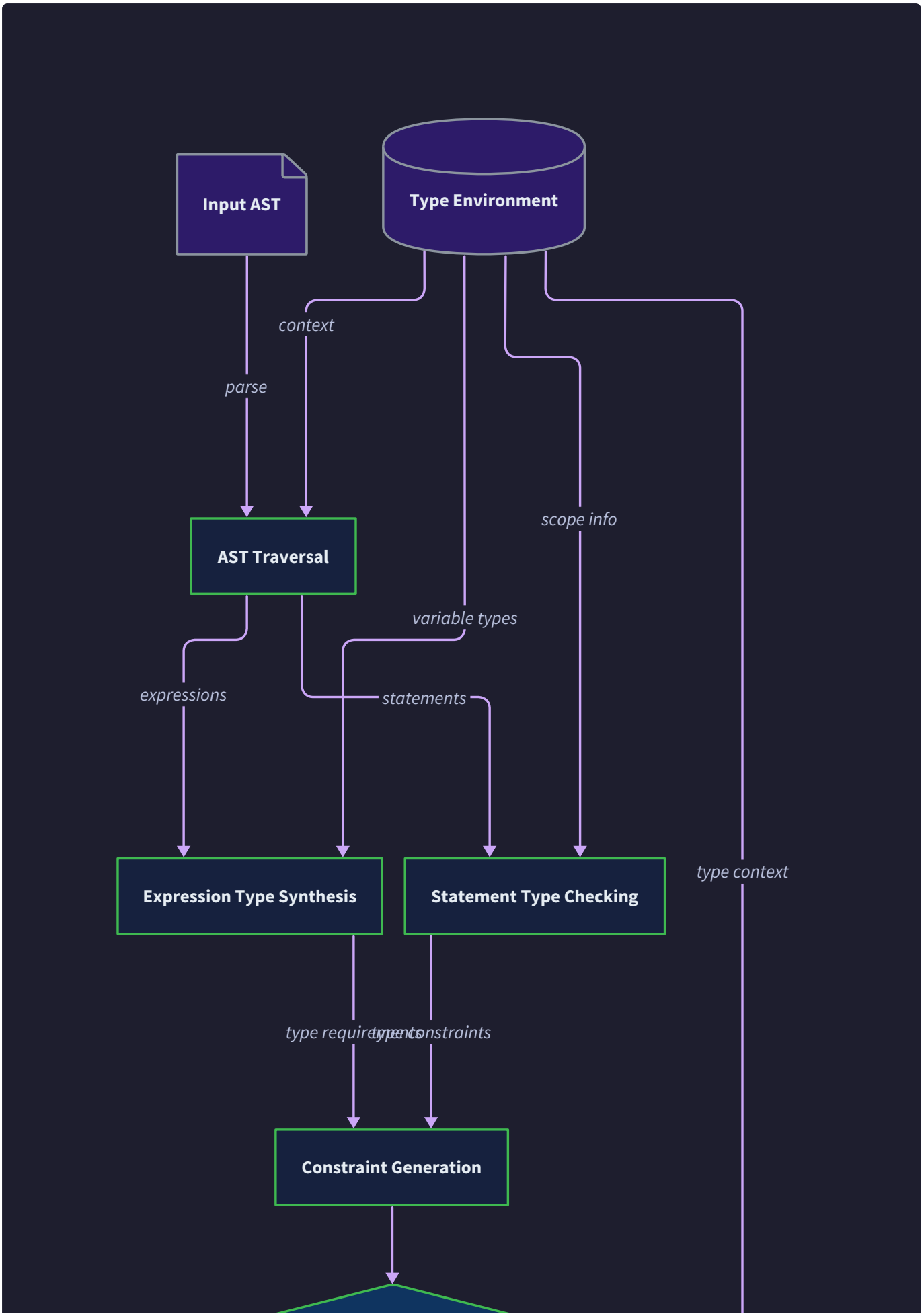
Type Inference Engine

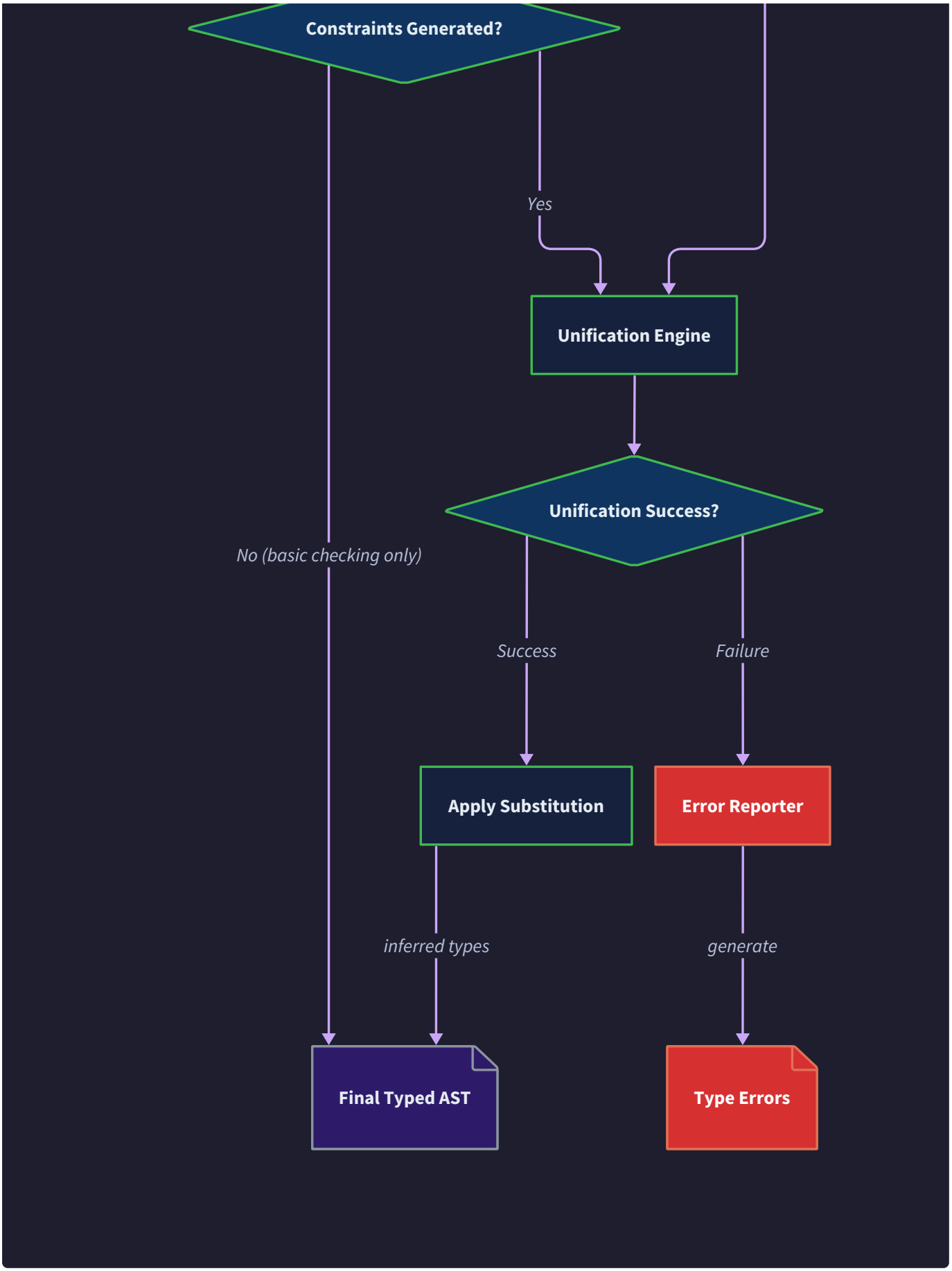
> **Milestone(s):** Milestone 3 (Type Inference)

Think of type inference as a **detective investigation system** — much like how a skilled detective pieces together clues from a crime scene to reconstruct what happened. The type inference engine examines an expression without explicit type annotations and uses contextual clues (literals, operators, function calls) to deduce what types must be involved. Just as a detective generates hypotheses ("if the suspect was here at 9

PM, then they must have...") and then tests these hypotheses against all available evidence, our type inference engine generates **type constraints** ("if this variable is used in addition, it must be numeric") and then uses **unification** to find a consistent assignment of types that satisfies all constraints simultaneously.

The power of type inference lies in its ability to provide the safety guarantees of static typing while reducing the annotation burden on programmers. Instead of requiring every variable and function to be explicitly typed, the inference engine can often deduce the intended types from usage patterns, making the language feel more dynamic while maintaining compile-time verification.


Input AST
Type Environment
context
parse
AST Traversal
scope info
variable types
expressions
statements
Expression Type Synthesis
Statement Type Checking
type context
type requirements
type constraints
Constraint Generation



Type inference operates through a three-phase process that mirrors how human reasoning works when understanding partially specified information. First, **constraint generation** systematically examines every

expression and statement to collect requirements about what types must be related ("this function parameter must have the same type as this argument"). Second, **unification** attempts to solve this system of constraints by finding type assignments that make all the relationships consistent. Finally, **substitution application** propagates the solved types back through the program, replacing type variables with their inferred concrete types.

Constraint Generation

Think of constraint generation as a **systematic evidence collection process** — like how a forensic team methodically catalogs every piece of evidence at a crime scene, noting relationships and dependencies without immediately jumping to conclusions. The constraint generation phase walks through every expression in the program and records type relationships that must hold for the program to be well-typed, without immediately trying to solve these relationships.

The constraint generation process operates by **synthesizing** fresh type variables for unknown types and then establishing equality constraints based on language semantics. When we encounter an expression like `x + y`, we don't immediately insist that `x` and `y` must be integers — instead, we generate constraints that capture the requirements: `typeof(x) = typeof(y)` and `typeof(x + y) = typeof(x)`. This allows the system to handle cases where the addition might involve integers, floats, or even custom numeric types with operator overloading.

> **Decision: Constraint-Based vs Direct Inference**
>
> - **Context**: We need to choose between immediately inferring types during traversal vs collecting constraints first
> - **Options Considered**: Direct inference (assign types immediately), constraint collection then solving, hybrid approaches
> - **Decision**: Pure constraint-based approach with separate solving phase
> - **Rationale**: Constraint collection allows handling mutually recursive definitions, provides better error messages by seeing all constraints together, and supports more sophisticated inference algorithms like let-polymorphism
> - **Consequences**: Requires more memory to store constraints and an additional unification phase, but enables more powerful inference and clearer separation of concerns

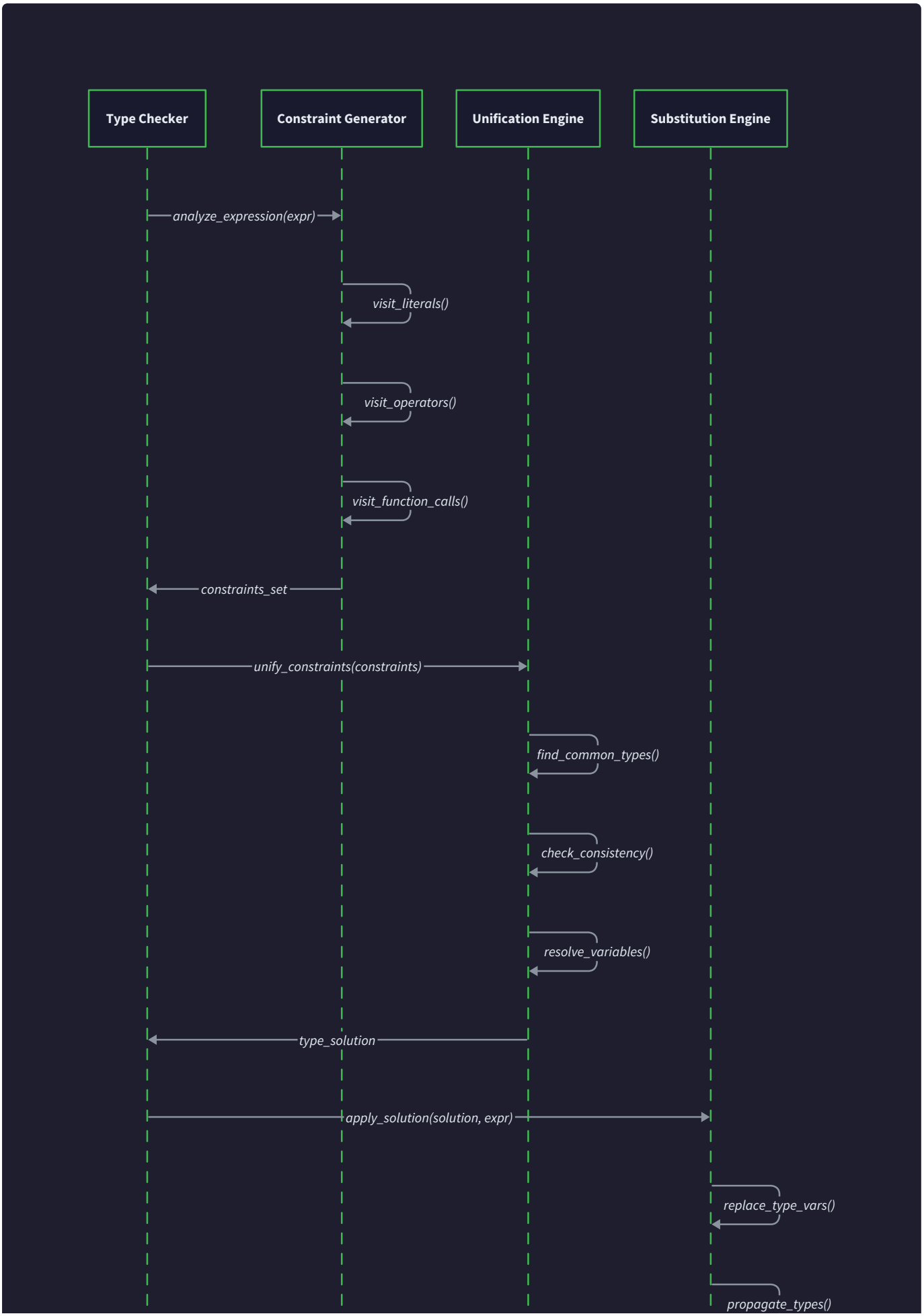
The constraint generation algorithm maintains several critical data structures during traversal. The **type environment** tracks known type bindings for variables in scope. A **fresh variable generator** creates unique type variables for unknown types. The **constraint accumulator** collects type equality requirements as they're discovered. Most importantly, each expression synthesis returns both an **expression type** (often a fresh type variable) and a **list of constraints** that must be satisfied for that type assignment to be valid.

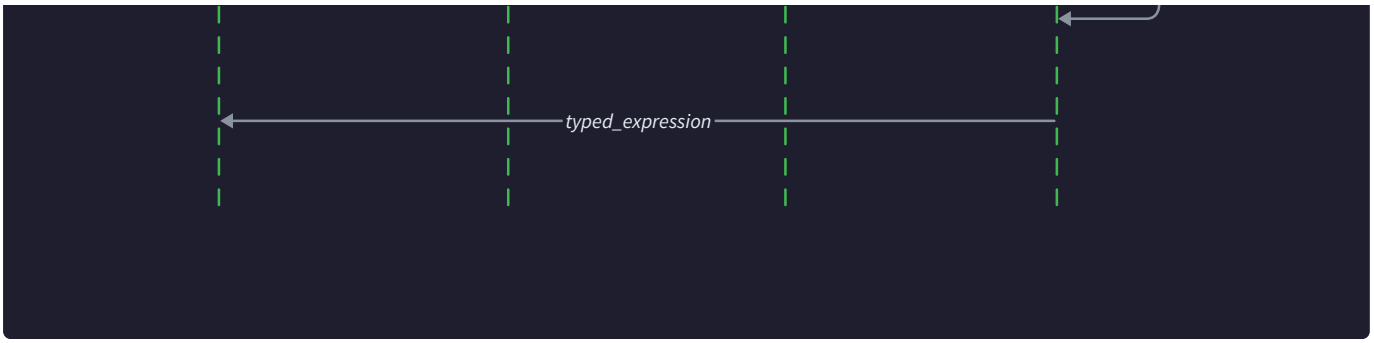
Constraint Generation Component	Purpose	Data Maintained
Type Environment	Maps variable names to their types in current scope	List of (variable_name, type) bindings
Fresh Variable Generator	Creates unique type variables for unknowns	Current variable counter
Constraint Accumulator	Collects type equality requirements	List of (type1, type2) constraint pairs
Expression Type Synthesizer	Determines type produced by each expression	Returns (expression_type, constraint_list)

The heart of constraint generation lies in the **expression synthesis rules** that define how different syntactic constructs generate type constraints. For literal expressions, synthesis is straightforward — integer literals generate no constraints and return `TInt`, string literals return `TString`. For more complex expressions, synthesis becomes a process of constraint propagation.

Expression Synthesis Algorithm:

1. **Literal Synthesis**: For literal values (integers, booleans, strings), return the corresponding primitive type with an empty constraint list, since literals have known, fixed types.
2. **Variable Reference Synthesis**: Look up the variable name in the current type environment. If found, return the bound type with no additional constraints. If not found, generate an `UnboundVariable` error.
3. **Binary Operation Synthesis**: Generate fresh type variables for both operands and the result. Create constraints that both operands have the same type and that the result type matches the operand type (for arithmetic) or is boolean (for comparisons).
4. **Function Call Synthesis**: Generate a fresh type variable for the return type. Look up the function type from the environment. If it's a function type `TFun(param_ty, return_ty)`, create a constraint that the argument type equals `param_ty` and return `return_ty`. If it's a type variable, create a constraint that it equals `TFun(arg_ty, fresh_return_ty)`.
5. **Conditional Expression Synthesis**: Generate constraints that the condition has type `TBool` and that both branches have the same type. Return the branch type (which may be a fresh type variable if both branches have unknown types).
6. **Let Expression Synthesis**: This is the most complex case. First, synthesize the type of the bound expression and collect its constraints. Then, extend the environment with the binding and synthesize the body type. The key insight is that let-expressions are where **generalization** can occur — if the bound expression's type contains type variables that don't appear elsewhere, they can be generalized into a polymorphic type scheme.


Type Checker
Constraint Generator
Unification Engine
Substitution Engine
analyze_expression(expr)
visit_literals()
visit_operators()
visit_function_calls()
constraints_set
unify_constraints(constraints)
find_common_types()
check_consistency()
resolve_variables()
type_solution
apply_solution(solution, expr)
replace_type_vars()
propagate_types()



Consider a concrete example of constraint generation for the expression `let f = fun x -> x + 1 in f 42`. The synthesis process works as follows:

The function `fun x -> x + 1` generates a fresh type variable `'a` for parameter `x`. The addition `x + 1` creates constraints `'a = TInt` (since `1` is an integer literal) and the function type becomes `TFun(TInt, TInt)`. The binding `f = fun x -> x + 1` adds `f : TFun(TInt, TInt)` to the environment. The call `f 42` generates a constraint that `TFun(TInt, TInt) = TFun(TInt, 'b)` for some fresh return type `'b`, which unifies to give `'b = TInt`.

Expression	Fresh Variables Generated	Constraints Created	Result Type
`1`	none	none	`TInt`
`x` (parameter)	`'a`	none	`'a`
`x + 1`	`'b` (result)	`'a = TInt`, `'b = TInt`	`'b`
`fun x -> x + 1`	none	none	`TFun('a, 'b)` with constraints
`42`	none	none	`TInt`
`f 42`	`'c` (result)	`TFun(TInt, TInt) = TFun(TInt, 'c)`	`'c`

The power of constraint generation becomes apparent when dealing with **mutually recursive functions** or **forward references**. Traditional direct inference struggles with these cases because it tries to determine types before having complete information. Constraint-based inference handles these naturally — it generates constraints for all function definitions simultaneously, then solves the entire constraint system to find a consistent type assignment.

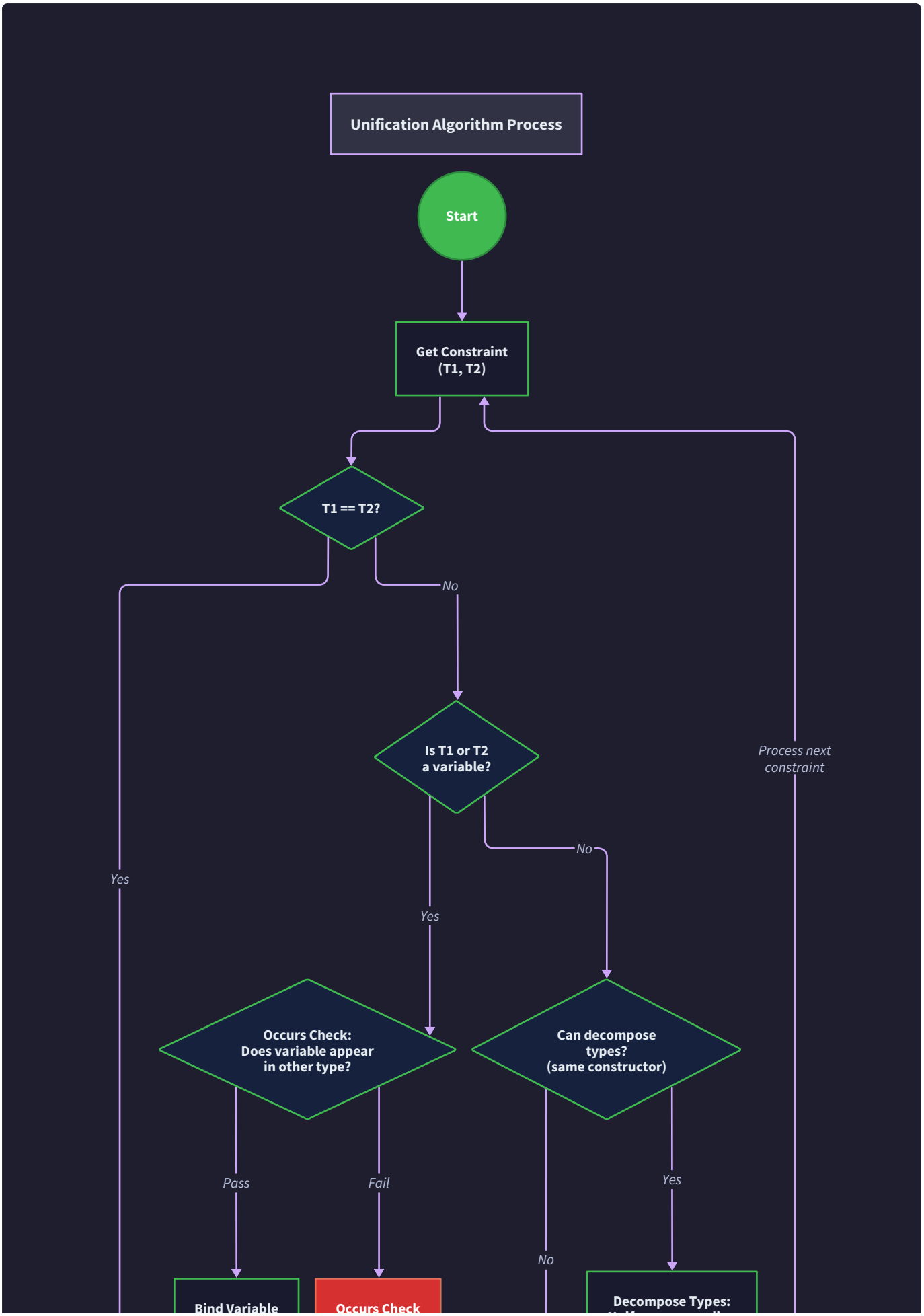
⚠️ **Pitfall: Premature Constraint Solving** Many implementations try to solve constraints immediately as they're generated, which breaks down for recursive definitions. For example, if function `f` calls function `g` and `g` calls `f`, immediate solving fails because neither function's type is known when processing the other. Instead, generate all constraints first, then solve them as a complete system. The constraint solver can handle circular dependencies that would confuse eager type assignment.

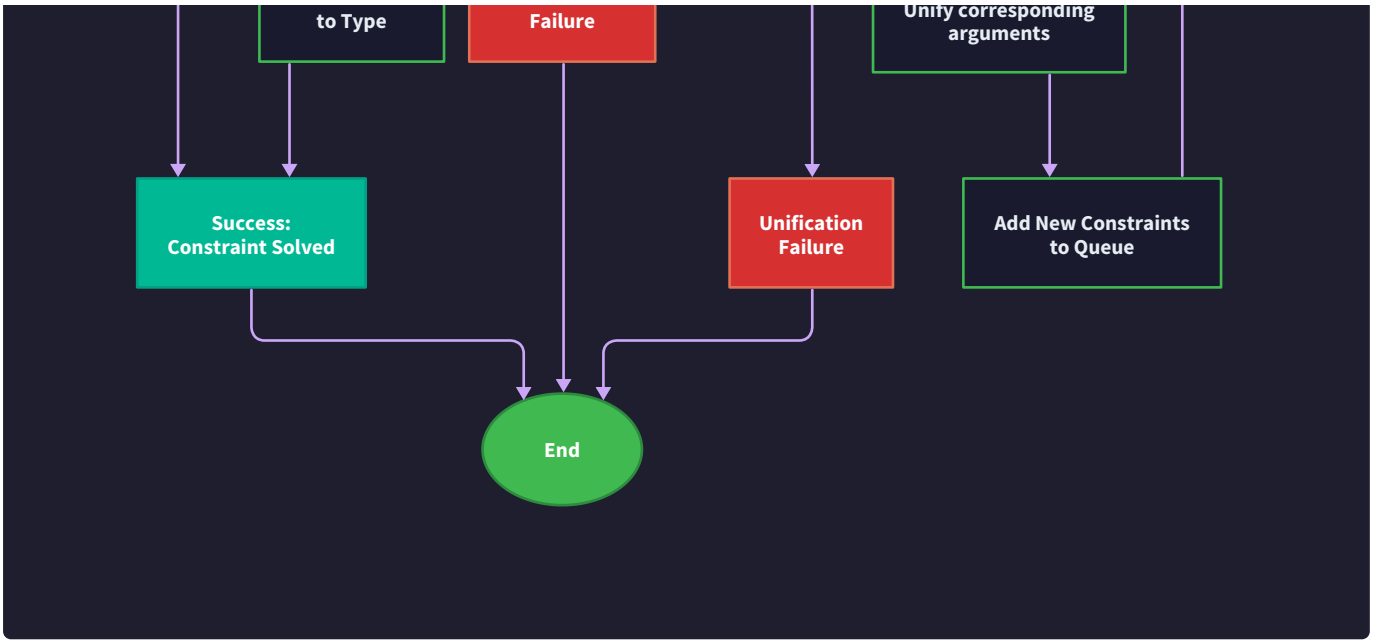
⚠️ **Pitfall: Forgetting to Thread Type Environment** Constraint generation must carefully maintain the type environment as it traverses nested scopes. A common mistake is using a global environment that gets corrupted by inner scopes. Instead, each recursive call should receive a properly extended environment, and environment changes should not escape their scope. For let-expressions, extend the environment for the body but not for subsequent expressions at the same level.

⚠️ **Pitfall: Not Generating Fresh Variables for Each Expression** Reusing type variables across different expressions can create spurious constraints. Each unknown type should get its own fresh variable. For example, if two separate function calls both need fresh return types, they should get different type variables (`'a` and `'b`) even if they might end up with the same concrete type. The unification phase will merge them if necessary.

Unification Algorithm

Think of unification as a **diplomatic negotiation process** where multiple parties (type constraints) each have requirements, and the goal is to find a compromise (type substitution) that satisfies everyone's needs simultaneously. Just as skilled diplomats look for creative solutions that address underlying interests rather than stated positions, the unification algorithm searches for type assignments that satisfy the structural requirements of all constraints, even when they initially appear contradictory.


Unification Algorithm Process
Start
Get Constraint (T1, T2)
T1 == T2?
Yes
No
Is T1 or T2 a variable?
Yes
No
Occurs Check: Does variable appear in other type?
Pass
Fail
Bind Variable
Occurs Check
Can decompose types? (same constructor)
Yes
No
Decompose Types:
Process next constraint



Unification implements the **Robinson unification algorithm**, which is both **sound** (never produces incorrect unifiers) and **complete** (always finds the most general unifier when one exists). The algorithm works by systematically decomposing complex type constraints into simpler ones, binding type variables to concrete types when possible, and detecting inconsistencies that indicate type errors.

> **Decision: Robinson vs Alternative Unification Algorithms**
>
> - **Context**: Need to choose a unification algorithm that balances correctness, completeness, and implementation complexity
> - **Options Considered**: Robinson unification, higher-order unification, nominal unification with subtyping
> - **Decision**: Classical Robinson unification with occurs check
> - **Rationale**: Robinson unification is well-understood, proven correct, handles structural type equality elegantly, and has optimal complexity for our constraint domain
> - **Consequences**: Limited to structural type equality (no subtyping), requires explicit occurs check implementation, but provides strong theoretical foundation and predictable behavior

The unification algorithm maintains a **substitution** as its primary data structure — a mapping from type variables to concrete types that represents the "solution" discovered so far. As the algorithm processes each constraint, it extends the substitution with new variable bindings or detects inconsistencies that indicate unsatisfiable constraints.

Unification Algorithm Steps:

1. **Initialize Empty Substitution**: Start with an empty mapping from type variables to types, representing that no variables have been bound yet.
2. **Process Each Constraint**: For each constraint `(t1, t2)` in the constraint list, apply the current substitution to both types to get their most up-to-date forms, then attempt to unify them.

3. **Variable-Type Unification**: If one side is a type variable and the other is any type, perform the occurs check (ensure the variable doesn't appear in the type), then bind the variable to the type in the substitution.
4. **Identical Type Unification**: If both sides are identical primitive types (`TInt = TInt`), the constraint is already satisfied and requires no additional binding.
5. **Function Type Decomposition**: If both sides are function types `TFun(p1, r1)` and `TFun(p2, r2)`, decompose this into two smaller constraints: `p1 = p2` and `r1 = r2`.
6. **Composition and Propagation**: When adding a new variable binding to the substitution, apply this binding to all existing bindings to maintain consistency (substitution composition).
7. **Failure Detection**: If none of the above rules apply (e.g., trying to unify `TInt` with `TBool`), the constraint system is unsatisfiable and unification fails.

The **occurs check** is a critical component that prevents the algorithm from creating infinite types. Without this check, a constraint like `'a = TFun('a, TInt)` would create an infinitely nested function type `TFun(TFun(TFun(..., TInt), TInt), TInt)`. The occurs check detects when a type variable appears within the type it's being bound to and rejects such bindings as invalid.

Unification Case	Example Constraint	Action Taken	Result
Variable-Concrete	`'a = TInt`	Bind `'a` to `TInt`	Substitution extended
Variable-Variable	`'a = 'b`	Bind `'a` to `'b` (or vice versa)	Substitution extended
Identical Concrete	`TInt = TInt`	No action needed	Constraint satisfied
Function Decomposition	`TFun('a, 'b) = TFun(TInt, 'c)`	Generate `'a = TInt`, `'b = 'c`	New constraints
Type Mismatch	`TInt = TBool`	Cannot unify	Unification failure
Occurs Check Failure	`'a = TFun('a, TInt)`	Infinite type detected	Unification failure

Consider unifying the constraint set generated from `let f = fun x -> x in (f 1, f true)`. The constraint generation produces:

- `f : TFun('a, 'b)` (from the function definition)
- `'a = TInt` and `'b = TInt` (from `f 1`)
- `'a = TBool` and `'b = TBool` (from `f true`)

The unification algorithm processes these constraints and discovers the conflict: `'a` cannot be both `TInt` and `TBool` simultaneously. This generates a type error indicating that the function `f` is being used at

incompatible types. In a system with let-polymorphism (covered in the next milestone), this example would succeed by generalizing `f`'s type to be polymorphic.

Substitution Composition is essential for maintaining consistency as new variable bindings are discovered. When we bind `'a` to `TInt`, we must apply this binding to all existing entries in the substitution. If we previously had `'b` mapped to `TFun('a, TBool)`, it must be updated to `TFun(TInt, TBool)` to reflect the new information about `'a`.

The unification algorithm's **complexity** is nearly linear in the size of the type expressions being unified, making it efficient even for large programs. The occurs check adds some overhead but is essential for correctness — without it, the type system becomes unsound and can accept programs that would crash at runtime.

⚠️ **Pitfall: Forgetting Substitution Composition** A common error is treating the substitution as a simple map without maintaining consistency. When you bind a type variable `'a` to a type `T`, you must apply this binding to all existing entries in the substitution. Failing to do this leads to inconsistent substitutions where some types mention the old unbound variable while others use the new binding. Always compose substitutions rather than simply adding new bindings.

⚠️ **Pitfall: Skipping the Occurs Check** The occurs check prevents infinite types but is often omitted in initial implementations because it seems like an edge case. However, recursive function definitions routinely generate constraints where a type variable appears on both sides of an equation. Without the occurs check, unification can succeed but produce infinite types that crash the type printer or subsequent analysis phases. The occurs check is not optional — it's required for soundness.

⚠️ **Pitfall: Processing Constraints in Wrong Order** While unification is theoretically order-independent, practical implementations can be sensitive to constraint ordering due to error reporting. Process simple constraints (variable bindings) before complex ones (function decompositions) to get cleaner error messages. If unification fails, the order of constraint processing affects which constraint appears to "cause" the failure in error messages.

Substitution Application

Think of substitution application as a **global find-and-replace operation** — much like using an editor's find-and-replace feature to update every occurrence of a placeholder with its final value throughout a document. After unification discovers the "solution" to the constraint system (which type variables should be bound to which concrete types), substitution application propagates this solution throughout the entire program, replacing type variables with their resolved types.

Substitution application is the **bridge between inference and checking** — it transforms a program full of unknown type variables into a program with concrete, checkable types. This phase is crucial because it's where the abstract constraint solving results become concrete type assignments that can be verified and reported to the programmer.

The substitution data structure produced by unification is a mapping from type variable names to their resolved types. However, applying this substitution is more complex than simple replacement because types can be nested (function types contain parameter and return types) and because substitutions can be **chained** (a type variable might be bound to another type variable that is itself bound to a concrete type).

Substitution Application Algorithm:

1. **Type Variable Resolution**: When applying substitution to a type variable, look it up in the substitution mapping. If found, recursively apply the substitution to the bound type (handling chains like `'a -> 'b -> TInt`). If not found, the variable remains unbound.
2. **Primitive Type Passthrough**: Concrete types like `TInt`, `TBool`, and `TString` are unchanged by substitution application since they contain no variables to replace.
3. **Function Type Recursion**: For function types `TFun(param_ty, return_ty)`, recursively apply the substitution to both the parameter type and return type, reconstructing the function type with the substituted components.
4. **Polymorphic Type Handling**: For type schemes `TForall(vars, ty)`, apply substitution to the body type while being careful not to substitute for variables that are bound by the forall quantifier (avoiding **variable capture**).
5. **Environment Substitution**: Apply the substitution to every type binding in the type environment, ensuring that variable types reflect the inferred concrete types.
6. **Expression Annotation**: If maintaining type-annotated ASTs, apply substitution to the type annotation of every expression node, replacing inferred type variables with their concrete types.

Substitution Application Target	Input	Output	Notes
Type Variable `'a`	`'a` with `{'a -> TInt}`	`TInt`	Direct lookup and replace
Chained Variable `'a`	`'a` with `{'a -> 'b, 'b -> TInt}`	`TInt`	Follow chain to final type
Function Type	`TFun('a, 'b)` with `{'a -> TInt, 'b -> TBool}`	`TFun(TInt, TBool)`	Recursive application
Unbound Variable `'c`	`'c` with `{'a -> TInt}`	`'c`	Remains as type variable
Environment Binding	`x : 'a` with `{'a -> TInt}`	`x : TInt`	Update environment

The most subtle aspect of substitution application is **handling substitution chains** correctly. Consider a substitution `{'a -> 'b, 'b -> 'c, 'c -> TInt}`. When applying this to type variable `'a`, we must

follow the chain: `'a` maps to `'b`, which maps to `'c`, which maps to `TInt`. The final result should be `TInt`, not `'b` or `'c`. This requires recursive application of substitution to the bound type.

Substitution Composition becomes important when substitutions are built incrementally or when combining substitutions from different sources. Composing substitutions `s1` and `s2` means creating a new substitution that has the same effect as applying `s1` first, then applying `s2` to the result. The composition algorithm must apply `s2` to all the values in `s1`, then add any new bindings from `s2`.

Consider the type inference process for a concrete example: `let double = fun x -> x + x in double 3.14`. During constraint generation, we generate fresh variables and constraints:

- `x : 'a` (function parameter)
- `x + x` requires `'a = 'a` (trivially satisfied) and produces type `'a`
- `double : TFun('a, 'a)` (function type)
- `double 3.14` requires `TFun('a, 'a) = TFun(TFloat, 'b)` for fresh `'b`

Unification discovers `'a = TFloat` and `'b = TFloat`. Substitution application then transforms:

- `double : TFun('a, 'a)` becomes `double : TFun(TFloat, TFloat)`
- The entire expression gets type `TFloat`

One crucial consideration is **partial substitution handling**. Not all type variables may be resolved after unification — some might remain unbound if they represent truly polymorphic types or if type information is insufficient. The substitution application phase must decide how to handle these unbound variables. In many cases, unbound variables indicate an error (insufficient type information), but in polymorphic systems, they might be intentionally left unbound for later instantiation.

⚠️ **Pitfall: Not Following Substitution Chains** A substitution might contain chains like `'a -> 'b -> TInt`. Simply looking up `'a` and returning `'b` is incorrect — you must recursively apply substitution until reaching a concrete type or unbound variable. Failing to follow chains leads to type variables appearing in final results where concrete types are expected, causing later phases to crash or produce confusing error messages.

⚠️ **Pitfall: Variable Capture in Polymorphic Types** When applying substitution to polymorphic types `TForall(vars, ty)`, be careful not to substitute for variables that are bound by the forall quantifier. For example, applying substitution `{'a -> TInt}` to `TForall(['a], TFun('a, 'a))` should not change the type at all because the `'a` inside is bound by the forall, not the one being substituted. This is similar to variable scoping in programming languages.

⚠️ **Pitfall: Forgetting to Update Environments** After discovering type variable bindings through unification, the type environment still contains the old type variables. You must apply the substitution to every binding in the environment to replace inferred type variables with their concrete types. Forgetting this step means that subsequent type checking phases see stale type variables instead of the inferred types.

Common Pitfalls

⚠️ **Pitfall: Generating Constraints for Error Cases** When constraint generation encounters an error (like an unbound variable), inexperienced implementations often try to generate constraints anyway, leading to confusing cascading errors. Instead, detect errors early and use error recovery techniques like inserting placeholder types that don't generate additional spurious constraints.

⚠️ **Pitfall: Inefficient Occurs Check Implementation** The naive occurs check recursively searches through type structures for every variable binding, leading to quadratic behavior. More efficient implementations maintain "levels" or use union-find data structures to make occurs checks constant time. For learning purposes, the simple recursive implementation is fine, but be aware of the performance implications.

⚠️ **Pitfall: Not Handling Recursive Type Definitions** While our basic type system doesn't include recursive types (like lists or trees), the constraint generation and unification infrastructure should be designed to handle them cleanly. This means avoiding assumptions that type checking always terminates quickly and being prepared for constraint systems that require more sophisticated solving techniques.

Implementation Guidance

The type inference engine represents the most algorithmically sophisticated component of the type checker. Unlike the previous components which were primarily about data structure design and straightforward rule application, type inference requires implementing two non-trivial algorithms: constraint-based type inference and Robinson unification.

Technology Recommendations:

Component	Simple Option	Advanced Option
Constraint Storage	`constraint_t list` (simple list)	`Set.Make(ConstraintOrd)` (efficient set)
Substitution	`(string * ty) list` (association list)	`Map.Make(String)` (balanced tree map)
Fresh Variable Generation	`ref int` counter	Unique ID with prefixes for debugging
Type Variable Representation	`string` names	Separate type for levels and scopes

Recommended File Structure:

```
src/
  types.ml              ← type definitions (from previous sections)
  type_env.ml           ← type environment (from previous sections)
  constraints.ml        ← constraint generation logic
  unification.ml        ← unification algorithm implementation
  substitution.ml       ← substitution data structure and application
  inference.ml          ← main type inference driver
  inference_test.ml     ← comprehensive test suite
```

This organization separates the three main algorithmic components (constraint generation, unification, substitution) into distinct modules while providing a clean interface through the main inference module.

Infrastructure Starter Code:

The constraint generation and unification algorithms require several utility components that, while not the core learning objectives, are necessary for a working implementation.

OCAML

```ocaml
(* substitution.ml - Complete substitution implementation *)

type substitution = (string * ty) list

let empty_subst = []

let rec lookup_var var = function
  | [] -> None
  | (v, t) :: rest -> if v = var then Some t else lookup_var var rest

let extend_subst var ty subst = (var, ty) :: subst

(* Apply substitution to a type, following chains *)
let rec apply_subst_to_type subst = function
  | TVar var ->
    (match lookup_var var subst with
     | None -> TVar var
     | Some ty -> apply_subst_to_type subst ty)  (* Follow chains *)
  | TFun(param, ret) ->
    TFun(apply_subst_to_type subst param, apply_subst_to_type subst ret)
  | TForall(vars, ty) ->
    (* Careful: don't substitute bound variables *)
    let filtered_subst = List.filter (fun (v, _) -> not (List.mem v vars)) subst in
    TForall(vars, apply_subst_to_type filtered_subst ty)
  | t -> t  (* Primitives unchanged *)

(* Compose two substitutions: (compose s1 s2) = apply s1 then s2 *)
let compose_subst s1 s2 =
  (* Apply s2 to all values in s1 *)
  let s1_updated = List.map (fun (var, ty) -> (var, apply_subst_to_type s2 ty)) s1 in
```

```
  (* Add bindings from s2 that aren't in s1 *)
  let s2_new = List.filter (fun (var, _) -> not (List.mem_assoc var s1)) s2 in
  s1_updated @ s2_new

(* Fresh variable generation *)
let var_counter = ref 0
let fresh_type_var () =
  incr var_counter;
  TVar ("t" ^ string_of_int !var_counter)
```

Core Logic Skeleton Code:

The heart of type inference lies in the constraint generation and unification algorithms. These should be implemented by the learner following the algorithmic steps outlined in the design section.

```ocaml
(* constraints.ml - Constraint generation *)

type constraint_t = ty * ty

(* Generate constraints for expressions *)
let rec generate_constraints expr env =
  (* TODO 1: Pattern match on expression type (Lit, Var, BinOp, Call, etc.) *)
  (* TODO 2: For literals - return (primitive_type, empty_constraint_list) *)
  (* TODO 3: For variables - lookup in env, return (bound_type, empty_constraints) *)
  (* TODO 4: For binary ops - generate fresh vars, create operator-specific constraints *)
  (* TODO 5: For function calls - ensure function type matches TFun(arg_ty, ret_ty) *)
  (* TODO 6: For let expressions - handle binding and body with proper scoping *)
  (* Hint: Use fresh_type_var() for unknowns, accumulate constraints in lists *)
  failwith "TODO: implement constraint generation"

(* Generate constraints for statements *)
let generate_stmt_constraints stmt env =
  (* TODO 1: Handle variable declarations with optional type annotations *)
  (* TODO 2: Handle assignments by constraining RHS type with LHS type *)
  (* TODO 3: Handle function definitions by processing parameters and body *)
  (* TODO 4: Return updated environment and collected constraints *)
  failwith "TODO: implement statement constraint generation"
```

```ocaml
(* unification.ml - Robinson unification algorithm *)

let rec occurs_check var_name ty =
  (* TODO 1: Return true if var_name appears anywhere in ty *)
  (* TODO 2: For TVar, check if it equals var_name *)
  (* TODO 3: For TFun, recursively check parameter and return types *)
  (* TODO 4: For TForall, check body type (bound vars already excluded) *)
  (* Hint: This prevents infinite types like 'a = TFun('a, TInt) *)
  failwith "TODO: implement occurs check"

let rec unify_one (t1, t2) subst =
  (* TODO 1: Apply current substitution to both t1 and t2 *)
  (* TODO 2: If both are same primitive type, return subst unchanged *)
  (* TODO 3: If one is TVar and other isn't, do occurs check then bind *)
  (* TODO 4: If both are TFun, decompose into parameter/return constraints *)
  (* TODO 5: If types don't match, raise unification failure *)
  (* Hint: Use apply_subst_to_type before matching on types *)
  failwith "TODO: implement single constraint unification"

let unify constraints =
  (* TODO 1: Start with empty substitution *)
  (* TODO 2: Process each constraint with unify_one *)
  (* TODO 3: Compose new binding with accumulated substitution *)
  (* TODO 4: Handle unification failures by collecting error info *)
  (* Hint: Use List.fold_left to accumulate substitution through constraint list *)
  failwith "TODO: implement full unification algorithm"
```

```ocaml
(* inference.ml - Main inference driver *)

let infer_expression expr env =
  (* TODO 1: Generate constraints from expression and environment *)
  (* TODO 2: Run unification algorithm on collected constraints *)
  (* TODO 3: Apply resulting substitution to expression type *)
  (* TODO 4: Apply substitution to environment bindings *)
  (* TODO 5: Return Success with inferred type or Error with type_error list *)
  failwith "TODO: implement expression inference"

let infer_statement stmt env =
  (* TODO 1: Generate constraints from statement *)
  (* TODO 2: Unify constraints to get type substitution *)
  (* TODO 3: Apply substitution to get concrete types *)
  (* TODO 4: Update environment with inferred bindings *)
  (* TODO 5: Handle let-bindings specially (preparation for polymorphism) *)
  failwith "TODO: implement statement inference with environment updates"
```

Language-Specific Hints:

- Use OCaml's pattern matching extensively for type analysis — each variant of the `ty` type should have its own match case
- The `List.fold_left` function is perfect for accumulating constraints and substitutions through recursive processing
- Use `ref` cells sparingly — only for the fresh variable counter. Pass substitutions and environments explicitly
- OCaml's exception handling can represent unification failures, but returning `Result` types gives better error information
- Use `List.mem` and `List.assoc` for simple association list operations on small substitutions

Milestone Checkpoint:

After implementing the type inference engine, you should be able to run:

```bash
ocamlfind ocamlc -package ounit2 -linkpkg -o test_inference inference_test.ml inference.ml constraints.ml unification.ml substitution.ml types.ml

./test_inference
```

Expected output should show successful inference for:

- Simple expressions like `42`, `true`, `"hello"`
- Binary operations like `1 + 2`, `x = y`
- Function calls with known function types
- Let-bindings with type inference: `let x = 1 + 2 in x`

Signs that something is wrong:

- "Unbound type variable" errors usually indicate substitution application bugs
- "Infinite type" errors suggest missing occurs check implementation
- Stack overflow in unification often means substitution chains aren't being followed properly
- Constraint generation crashes typically indicate missing cases in expression matching

Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
Unification fails immediately	Constraint generation bug	Print generated constraints before unification	Check expression synthesis rules
Stack overflow in unification	Infinite substitution chain	Trace substitution application calls	Implement proper occurs check
Wrong types inferred	Substitution not applied	Print types before and after substitution	Apply substitution to environment
Cascading type errors	Early error not caught	Add error recovery in constraint generation	Use placeholder types for errors

The type inference engine represents the theoretical heart of the type checker. Once this component works correctly, you'll have a system that can deduce types for complex expressions without requiring explicit annotations — a significant step toward a practical programming language implementation.

Polymorphic Type System

> **Milestone(s):** Milestone 4 (Polymorphism)

Think of a polymorphic type system as a **smart template library** — much like how a single blueprint for a storage container can be instantiated as a box for books, a box for tools, or a box for electronics, polymorphic types allow a single function or data structure definition to work with many different concrete types. The key insight is that we need two complementary operations: **generalization** (abstracting a concrete type into a reusable template) and **instantiation** (creating a fresh concrete version from the template).

The polymorphic type system represents the culmination of our type checker's sophistication. While basic type checking verifies that concrete types match, and type inference discovers unknown types through constraints, polymorphism enables code reuse by allowing a single definition to work at multiple types. This requires extending our type representation to include quantified type variables, implementing algorithms to generalize inferred types at appropriate points, and instantiating polymorphic types with fresh variables at use sites.

The mental model that guides polymorphic type checking is **template instantiation with scope-aware generalization**. When we encounter a let-binding like `let id = fun x -> x`, we don't just infer that `id` has some specific type like `int -> int`. Instead, we recognize that this function works for any type, so we generalize it to the polymorphic type scheme `forall 'a. 'a -> 'a`. Later, when we see `id 42` and `id true` in the same program, we instantiate the polymorphic type twice with different type variables, allowing the unification algorithm to specialize each use appropriately.

Type Schemes and Quantification

Type schemes provide the representation mechanism for polymorphic types by explicitly tracking which type variables are quantified (bound by forall) versus which are free (still being inferred). Think of a type scheme as a **factory blueprint with parameter slots** — the blueprint `forall 'a 'b. 'a -> 'b -> 'a` describes a family of function types where the caller can fill in any concrete types for the parameters `'a` and `'b`.

The distinction between bound and free type variables is crucial for correctness. A bound type variable like `'a` in `forall 'a. 'a -> 'a` represents a parameter that can be instantiated to any type, but both occurrences of `'a` must be instantiated to the same type. A free type variable represents an unknown type that is still being inferred by the unification algorithm. When we generalize a type into a type scheme, we promote free type variables to bound type variables, effectively saying "this type works for any choice of these variables."

Type Scheme Data Structures:

Field	Type	Description
quantified_vars	string list	Type variable names bound by forall quantification
body_type	ty	The underlying type with quantified variables as TVar nodes

Method Name	Parameters	Returns	Description
make_type_scheme	quantified_vars: string list, body: ty	type_scheme	Constructs type scheme with given quantified variables
scheme_to_string	scheme: type_scheme	string	Pretty prints type scheme with forall notation
free_vars_in_scheme	scheme: type_scheme	string list	Returns type variables free in scheme body but not quantified
scheme_contains_var	scheme: type_scheme, var_name: string	bool	Checks if variable appears free in scheme

The `type_scheme` representation uses explicit quantification rather than implicit generalization to make the polymorphic structure clear and enable precise control over instantiation. The `quantified_vars` list contains the names of type variables that should be treated as parameters, while `body_type` contains the actual type structure with those variables appearing as `TVar` nodes.

> **Key Design Insight**: We represent type schemes explicitly rather than using implicit generalization because we need to distinguish between "this variable can be anything" (bound) and "this variable is unknown but will be determined by unification" (free). This distinction is essential for correct let-polymorphism.

Type Scheme Operations:

The fundamental operations on type schemes support the generalization-instantiation cycle. Generalization takes a monomorphic type and abstracts over its free type variables to create a polymorphic type scheme. Instantiation takes a polymorphic type scheme and replaces its quantified variables with fresh type variables, creating a new monomorphic type that can be used in constraint generation.

Operation	Input	Output	Purpose
Generalization	Inferred type + type environment	Type scheme	Abstract over variables free in type but not environment
Instantiation	Type scheme	Monomorphic type	Replace quantified variables with fresh variables
Scheme Equality	Two type schemes	Boolean	Check if schemes represent same polymorphic type
Scheme Substitution	Type scheme + substitution	Type scheme	Apply substitution to free variables in scheme

Type Generalization

Type generalization implements the **let-polymorphism** rule that makes functional programming languages so expressive. The key insight is that when we infer a type for a let-bound variable, we should generalize over any type variables that are free in the inferred type but not free in the surrounding type environment. This captures the intuition that "this definition doesn't depend on any unknown types from the context, so it can work at any type."

Think of generalization as **context-sensitive abstraction** — we look at what type variables appear in the inferred type, then check which of those variables represent genuine unknowns versus which represent dependencies on the surrounding context. Variables that don't appear in the environment are good candidates for generalization because they represent internal polymorphism within the definition.

> **Decision: Let-Polymorphism Generalization Point**
>
> - **Context**: We need to decide when to generalize inferred types into polymorphic type schemes
> - **Options Considered**:
>   1. Generalize at every variable binding
>   2. Generalize only at let-bindings
>   3. Generalize at let-bindings with value restriction
> - **Decision**: Generalize at let-bindings with value restriction
> - **Rationale**: This matches ML-family languages and prevents unsound generalization of effects. Generalizing everywhere would make type inference undecidable, while never generalizing would lose expressiveness.
> - **Consequences**: Enables powerful polymorphic programming while maintaining soundness and decidability

Generalization Algorithm:

The generalization algorithm follows these steps to safely abstract over free type variables:

1. **Collect free variables in inferred type**: Walk the type structure and gather all `TVar` nodes that appear in the type
2. **Collect free variables in environment**: Walk all type schemes in the current environment and gather type variables that appear free (not quantified)
3. **Compute generalizable variables**: Take the set difference - variables free in the type but not free in the environment
4. **Apply value restriction**: Check if the let-bound expression is a syntactic value (lambda, literal, constructor) - if not, don't generalize
5. **Create type scheme**: If generalization is allowed, create a `type_scheme` with the generalizable variables as quantified variables

Generalization Step	Input	Processing	Output
Free Variable Collection	Inferred type	Traverse type collecting TVar names	String set of free variables
Environment Analysis	Type environment	Traverse environment collecting free vars	String set of context variables
Generalization Set	Two string sets	Set difference operation	Variables to generalize over
Value Restriction	Let-bound expression	Syntactic value check	Boolean: allow generalization
Scheme Construction	Variables + type	Create Forall quantification	Polymorphic type scheme

Value Restriction Implementation:

The value restriction prevents generalization of expressions that might have computational effects or mutable state. This restriction is crucial for soundness in languages with mutation or exceptions. The restriction allows generalization only when the let-bound expression is syntactically a value.

Expression Form	Generalizable	Reasoning
Literals (`42` , `true` , `"hello"`)	Yes	No computation performed
Lambda expressions (`fun x -> e`)	Yes	Function values are safe to generalize
Variable references (`x`)	Yes	No computation, just lookup
Constructor applications (`Cons(1, Nil)`)	Yes	Pure value construction
Function calls (`f x`)	No	May have effects or create mutable state
Conditionals (`if b then e1 else e2`)	No	Involves computation to evaluate guard
Let expressions (`let x = e1 in e2`)	Depends on e2	Check if final expression is value

> ⚠️ **Pitfall: Overgeneralization Without Value Restriction** Forgetting the value restriction can lead to unsound types. For example, `let r = ref None` might be generalized to `forall 'a. 'a option ref` , allowing `r := Some 42; !r` to be typed as `string option` , which would crash at runtime. The value restriction prevents generalizing non-values like `ref None` .

Generalization Example Walkthrough:

Consider generalizing the type of `let id = fun x -> x` in an empty environment:

1. **Infer type**: The function `fun x -> x` gets inferred type `'a -> 'a` where `'a` is a fresh type variable

2. **Collect type variables**: Free variables in `'a -> 'a` are `{'a'}`
3. **Check environment**: Empty environment has no free variables: `{}`
4. **Compute generalization set**: `{'a'} - {} = {'a'}` - we can generalize over `'a`
5. **Apply value restriction**: `fun x -> x` is a lambda, so it's a syntactic value - generalization allowed
6. **Create scheme**: Result is `Forall(['a'], TFun(TVar('a'), TVar('a')))` representing `forall 'a. 'a -> 'a`

Now consider `let f = g h` where `g : 'b -> 'c` and `h : 'd`:

1. **Infer type**: Application `g h` unifies `'b` with `'d`, resulting in type `'c`
2. **Collect type variables**: Free variables in `'c` are `{'c'}`
3. **Check environment**: Environment contains `g` and `h` with free variable `'c`, so environment free variables are `{'c'}`
4. **Compute generalization set**: `{'c'} - {'c'} = {}` - no variables to generalize
5. **Result**: Type remains monomorphic `'c`

Type Instantiation

Type instantiation performs the complementary operation to generalization by taking a polymorphic type scheme and creating a fresh monomorphic type that can be used in constraint generation. Think of instantiation as **template expansion with fresh parameters** — each time we instantiate `forall 'a. 'a -> 'a`, we create a new copy like `'b -> 'b` where `'b` is a completely fresh type variable.

The key insight behind instantiation is that each use of a polymorphic value should be independent. When we see two calls to a polymorphic function like `id 42` and `id true`, each call should get its own fresh instantiation of the polymorphic type. This allows the unification algorithm to specialize each use differently without interference.

> **Key Design Principle**: Instantiation must always generate fresh type variables to avoid accidental sharing between different uses of the same polymorphic value. Reusing type variables would create spurious constraints that prevent independent specialization.

Instantiation Algorithm:

The instantiation process systematically replaces quantified type variables with fresh type variables while preserving the structure of the type:

1. **Generate fresh variables**: For each quantified variable in the type scheme, generate a completely fresh type variable using `fresh_type_var()`
2. **Build substitution**: Create a substitution mapping from each quantified variable name to its corresponding fresh variable

3. **Apply substitution**: Use `apply_subst_to_type` to replace all occurrences of quantified variables in the scheme body
4. **Return monomorphic type**: The result is a monomorphic type with fresh variables that can participate in constraint generation

Instantiation Step	Input	Processing	Output
Fresh Generation	Quantified variable list	Call `fresh_type_var()` for each	List of fresh TVar nodes
Substitution Building	Variable names + fresh vars	Zip into (name, fresh_var) pairs	Substitution mapping
Type Substitution	Scheme body + substitution	Apply substitution to type	Type with fresh variables
Monomorphic Result	Substituted type	Return as regular type	Type ready for constraint generation

Instantiation Implementation Details:

The instantiation process must handle nested type structures correctly, ensuring that all occurrences of a quantified variable are replaced consistently. The substitution application recursively traverses the type structure, replacing `TVar` nodes that match quantified variables.

Type Constructor	Instantiation Behavior
`TInt`, `TBool`, `TString`	No change - primitives have no variables
`TVar(name)`	Replace with fresh variable if name is quantified, otherwise unchanged
`TFun(arg, ret)`	Recursively instantiate both argument and return types
`TForall(vars, body)`	Error - nested quantification not supported in this system

Multiple Instantiation Example:

Consider the polymorphic identity function `id : forall 'a. 'a -> 'a` used in the expression `(id 42, id true)`:

First instantiation for `id 42`:

1. Generate fresh variable: `'b1`
2. Build substitution: `[('a', TVar('b1'))]`
3. Apply to `'a -> 'a`: Result is `'b1 -> 'b1`
4. Constraint generation: `id 42` generates constraint `'b1 = int`

Second instantiation for `id true`:

1. Generate fresh variable: `'b2` (different from `'b1`)
2. Build substitution: `[('a', TVar('b2'))]`
3. Apply to `'a -> 'a`: Result is `'b2 -> 'b2`
4. Constraint generation: `id true` generates constraint `'b2 = bool`

Unification result: The constraints `'b1 = int` and `'b2 = bool` are solved independently, allowing the first use to specialize to `int -> int` and the second to `bool -> bool`. This is exactly the behavior we want from polymorphism.

> ⚠️ **Pitfall: Reusing Type Variables Across Instantiations** If we accidentally reused the same fresh variable `'b` for both instantiations, we would generate conflicting constraints `'b = int` and `'b = bool`, causing a unification failure. Each instantiation must use completely fresh variables.

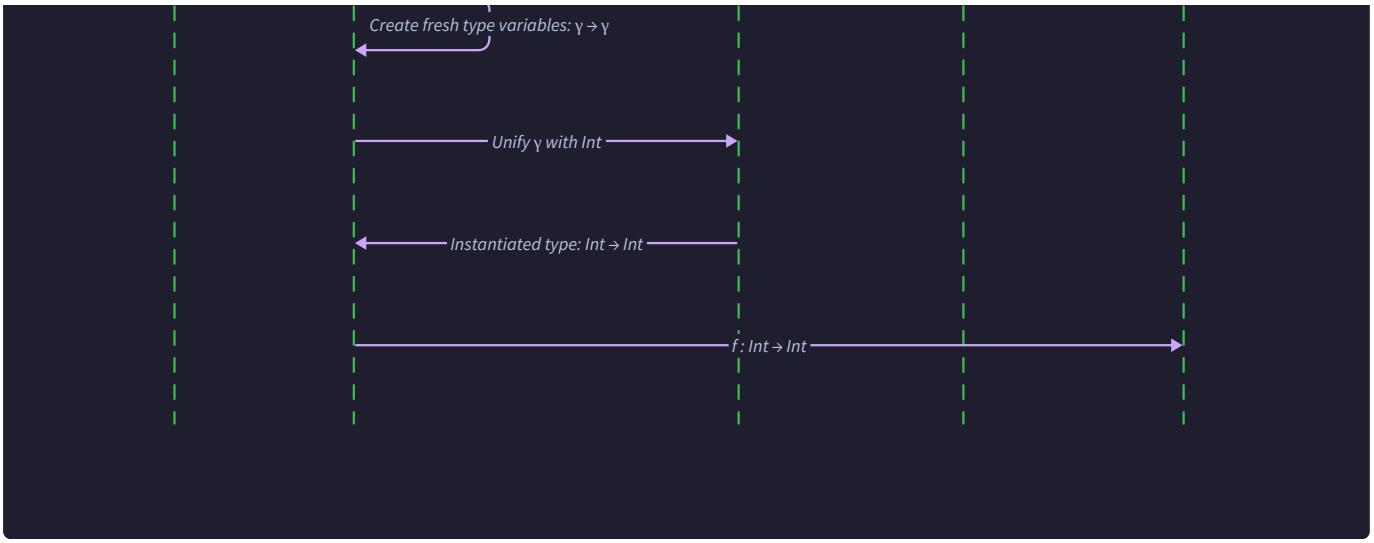
Polymorphic Recursion Limitations:

Our let-polymorphism system does not support polymorphic recursion, where a recursive function calls itself at different types within its own definition. This limitation is intentional for decidability - supporting polymorphic recursion would require more sophisticated analysis or programmer annotations.

Recursion Pattern	Supported	Example
Monomorphic recursion	Yes	`let rec factorial n = if n <= 1 then 1 else n * factorial (n-1)`
Mutually recursive functions	Yes	`let rec even n = n = 0
Polymorphic recursion	No	`let rec f x = ... f (Some x) ...` (calling f at different type)

The restriction to non-polymorphic recursion means that within a recursive function's definition, all recursive calls must use the same type instantiation. Only after the function is fully defined and generalized can it be used polymorphically.


Parser
Type Checker
Constraint Solver
Generalizer
Use Context
let f = λx. x + 1
Generate constraints: α → β, α = Int, β = Int
Unified type: Int → Int
Type ready for generalization
Identify free type variables
Generalized type: ∀α. α → α (polymorphic)
Bind f : ∀α. α → α in environment
Request f applied to String
Instantiate ∀α. α → α
Create fresh type variables: β → β
Unify β with String
Instantiated type: String → String
f : String → String
Request f applied to Int
Instantiate ∀α. α → α again



Implementation Guidance

This subsection provides concrete implementation guidance for building the polymorphic type system components in OCaml, focusing on the data structures, algorithms, and integration points needed to extend our existing type checker with let-polymorphism.

Technology Recommendations:

Component	Simple Option	Advanced Option
Type Schemes	Explicit forall with string list	De Bruijn indices with level-based quantification
Fresh Variable Generation	Global counter with string prefixes	UUID-based generation with conflict detection
Substitution Representation	Association list (string * ty) list	Map-based with path compression
Value Restriction	Syntactic pattern matching	Effect analysis with purity tracking

Recommended Module Structure:

The polymorphic type system integrates with existing modules while adding new functionality for type schemes, generalization, and instantiation:

```
src/
  types/
    type_repr.ml          ← extend with TForall constructor
    type_scheme.ml        ← new: type schemes and operations
    type_env.ml           ← extend to store type schemes
  inference/
    constraint_gen.ml     ← extend with instantiation calls
    unification.ml        ← existing unification algorithm
    generalization.ml     ← new: let-polymorphism generalization
  checker/
    type_checker.ml       ← integrate generalization at let-bindings
```

Type Scheme Infrastructure (Complete Implementation):

```ocaml
(* type_scheme.ml *)

type type_scheme = Forall of string list * ty


let make_type_scheme quantified_vars body_type =

  Forall (quantified_vars, body_type)


let scheme_to_string (Forall (vars, body)) =

  match vars with

  | [] -> string_of_type body

  | _ -> "forall " ^ String.concat " " vars ^ ". " ^ string_of_type body


let rec free_vars_in_type = function

  | TInt | TBool | TString -> []

  | TVar name -> [name]

  | TFun (arg, ret) -> free_vars_in_type arg @ free_vars_in_type ret

  | TForall (_, _) -> failwith "Nested quantification not supported"


let free_vars_in_scheme (Forall (quantified, body)) =

  let all_free = free_vars_in_type body in

  List.filter (fun v -> not (List.mem v quantified)) all_free


let scheme_contains_var (Forall (quantified, body)) var_name =

  let free_vars = free_vars_in_type body in

  List.mem var_name free_vars && not (List.mem var_name quantified)
```

Fresh Variable Generation (Complete Implementation):

```ocaml
(* Fresh variable generation with global counter *)
let var_counter = ref 0

let fresh_type_var () =
  incr var_counter;
  TVar ("'t" ^ string_of_int !var_counter)

let fresh_var_name () =
  incr var_counter;
  "'t" ^ string_of_int !var_counter

(* Generate multiple fresh variables at once *)
let fresh_type_vars count =
  let rec gen acc n =
    if n <= 0 then List.rev acc
    else gen (fresh_type_var () :: acc) (n - 1)
  in
  gen [] count
```

Type Environment Extension (Complete Implementation):

OCAML

```ocaml
(* Extend type_env.ml to handle type schemes *)

type binding =

  | Monomorphic of ty

  | Polymorphic of type_scheme


type env = (string * binding) list list


let extend_mono name ty env =

  match env with

  | scope :: rest -> ((name, Monomorphic ty) :: scope) :: rest

  | [] -> [[(name, Monomorphic ty)]]


let extend_poly name scheme env =

  match env with

  | scope :: rest -> ((name, Polymorphic scheme) :: scope) :: rest

  | [] -> [[(name, Polymorphic scheme)]]


let rec lookup_in_scopes name = function

  | [] -> None

  | scope :: rest ->

    (match List.assoc_opt name scope with

     | Some binding -> Some binding

     | None -> lookup_in_scopes name rest)


let lookup name env = lookup_in_scopes name env


(* Collect all free variables in current environment *)

let free_vars_in_env env =

  let collect_from_binding = function

    | Monomorphic ty -> free_vars_in_type ty
```

```
    | Polymorphic scheme -> free_vars_in_scheme scheme
  in
  let collect_from_scope scope =
    List.fold_left (fun acc (_, binding) ->
      collect_from_binding binding @ acc) [] scope
  in
  List.fold_left (fun acc scope ->
    collect_from_scope scope @ acc) [] env
```

Core Logic Skeleton - Generalization:

```ocaml
(* generalization.ml *)

let can_generalize expr =

  (* TODO 1: Check if expr is a syntactic value according to value restriction *)

  (* TODO 2: Return true for: literals, variables, lambdas, constructors *)

  (* TODO 3: Return false for: function applications, conditionals, etc. *)

  (* Hint: Pattern match on expression AST node type *)

  failwith "TODO: implement value restriction check"


let generalize_type inferred_type env expr =

  (* TODO 1: Get free variables in inferred_type using free_vars_in_type *)

  (* TODO 2: Get free variables in environment using TypeEnv.free_vars_in_env *)

  (* TODO 3: Compute set difference - vars in type but not in env *)

  (* TODO 4: Check value restriction using can_generalize *)

  (* TODO 5: If restriction allows, create Forall scheme, else return mono type *)

  (* Hint: Use List.filter and List.mem for set operations *)

  failwith "TODO: implement generalization algorithm"


let generalize_at_let var_name inferred_type env expr =

  (* TODO 1: Call generalize_type to get type scheme or monomorphic type *)

  (* TODO 2: Extend environment with appropriate binding type *)

  (* TODO 3: Return updated environment *)

  failwith "TODO: implement let-binding generalization"
```

Core Logic Skeleton - Instantiation:

```ocaml
(* type_scheme.ml - add instantiation functions *)

let instantiate_scheme (Forall (quantified_vars, body)) =

  (* TODO 1: Generate fresh type variables for each quantified variable *)

  (* TODO 2: Build substitution mapping quantified names to fresh vars *)

  (* TODO 3: Apply substitution to scheme body using apply_subst_to_type *)

  (* TODO 4: Return the monomorphic type with fresh variables *)

  (* Hint: Use List.map and List.combine to build substitution *)

  failwith "TODO: implement type scheme instantiation"


let instantiate_binding = function

  | Monomorphic ty -> ty

  | Polymorphic scheme -> instantiate_scheme scheme
```

Integration Points - Type Checking with Polymorphism:

```ocaml
(* type_checker.ml - extend existing type checking *)

let check_variable_reference var_name env =

  (* TODO 1: Look up variable in environment using TypeEnv.lookup *)

  (* TODO 2: If found, instantiate the binding (mono or polymorphic) *)

  (* TODO 3: Return instantiated type for constraint generation *)

  (* TODO 4: If not found, return UnboundVariable error *)

  failwith "TODO: implement polymorphic variable lookup"


let check_let_binding var_name expr body env =

  (* TODO 1: Infer type of expr in current environment *)

  (* TODO 2: Generalize inferred type using generalize_at_let *)

  (* TODO 3: Check body in extended environment *)

  (* TODO 4: Return body type and any accumulated errors *)

  (* Hint: This is where let-polymorphism magic happens *)

  failwith "TODO: implement let-binding with generalization"
```

Milestone Checkpoint:

After implementing polymorphic types, verify your implementation with these tests:

```bash
# Test basic polymorphism

echo "let id = fun x -> x in (id 42, id true)" | ./type_checker

# Expected: (int * bool)


# Test polymorphic let-binding

echo "let f = fun x -> fun y -> x in f 42 true" | ./type_checker

# Expected: bool (f specialized to int -> bool -> int)


# Test value restriction

echo "let r = ref [] in (r := [1]; r := [true])" | ./type_checker

# Expected: Type error - conflicting uses of monomorphic ref
```

Common Implementation Pitfalls:

Symptom	Likely Cause	Diagnosis	Fix
"Unbound variable" for polymorphic functions	Not instantiating type schemes at lookup	Check if `lookup` returns `Polymorphic` but you use the scheme directly	Add `instantiate_binding` call after lookup
Type errors on valid polymorphic code	Over-eager generalization without value restriction	Trace generalization calls - are non-values being generalized?	Implement proper `can_generalize` syntactic check
"Infinite type" errors in polymorphic contexts	Fresh variable generation collision	Check if same variable name used in multiple instantiations	Ensure `fresh_type_var()` returns globally unique names
Polymorphic recursion accepted incorrectly	Generalizing recursive definitions too early	Check when generalization happens in recursive let	Only generalize after complete recursive definition group

Debugging Trace Utilities:

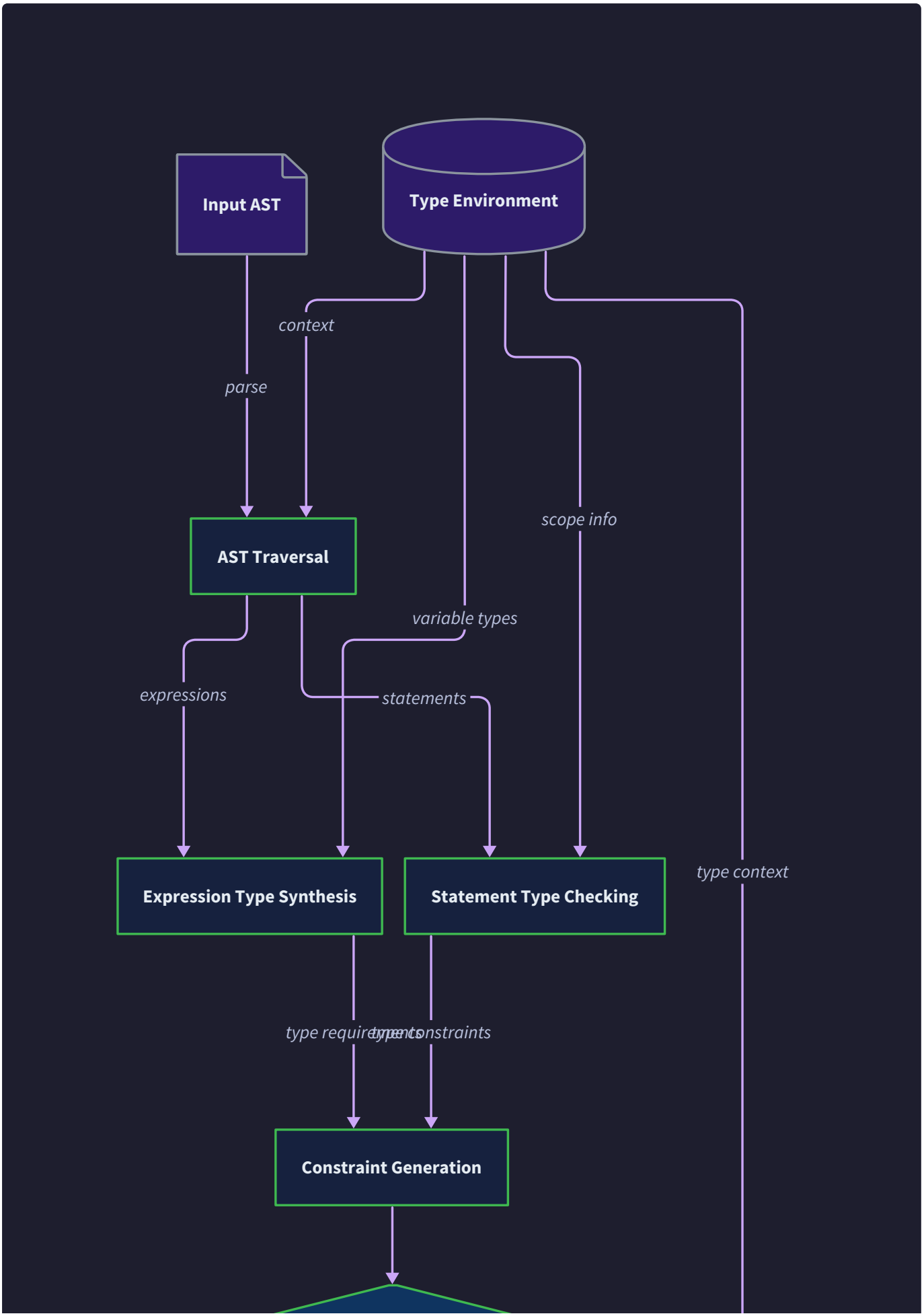
```ocaml
let debug_generalization var_name inferred_type env =
  Printf.printf "Generalizing %s : %s\n" var_name (string_of_type inferred_type);
  Printf.printf "Free in type: [%s]\n"
    (String.concat "; " (free_vars_in_type inferred_type));
  Printf.printf "Free in env: [%s]\n"
    (String.concat "; " (TypeEnv.free_vars_in_env env))

let debug_instantiation scheme =
  Printf.printf "Instantiating %s\n" (scheme_to_string scheme);
  let result = instantiate_scheme scheme in
  Printf.printf "Result: %s\n" (string_of_type result);
  result
```

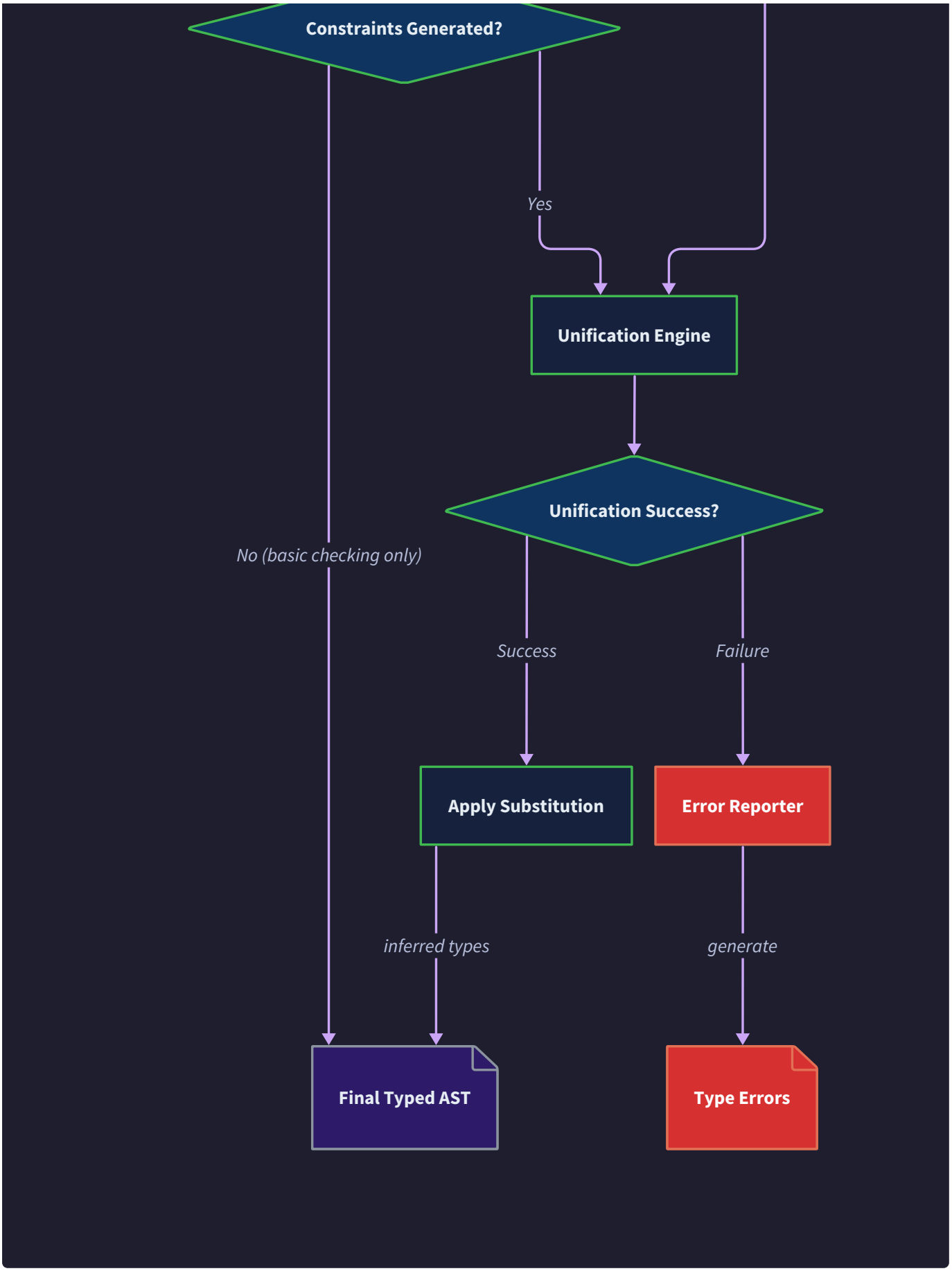
This implementation provides the foundation for let-polymorphism while maintaining integration with the existing constraint-based type inference system. The key insight is that generalization and instantiation work as bookends around the normal constraint generation and unification process, allowing the existing algorithms to handle the monomorphic instances while the polymorphic system manages the abstraction and specialization.

Component Interactions and Data Flow

> **Milestone(s):** Milestone 2 (Basic Type Checking), Milestone 3 (Type Inference), Milestone 4 (Polymorphism)

Think of the component interactions in our type checker as a **well-orchestrated symphony orchestra** — each component has a specific role and timing, but the magic happens in how they coordinate together to produce a harmonious result. The conductor (our pipeline orchestrator) ensures that the string section (type representation) provides the foundational melody, while the brass section (inference engine) adds the complex harmonies, and the percussion section (error reporter) provides crucial feedback when something goes wrong. Just as musicians must pass musical phrases between sections and maintain perfect timing, our type checker components must thread data through a carefully choreographed sequence while maintaining consistency and recovering gracefully from errors.


Input AST
Type Environment
context
parse
AST Traversal
scope info
variable types
expressions
statements
Expression Type Synthesis
Statement Type Checking
type context
type requirements
type constraints
Constraint Generation

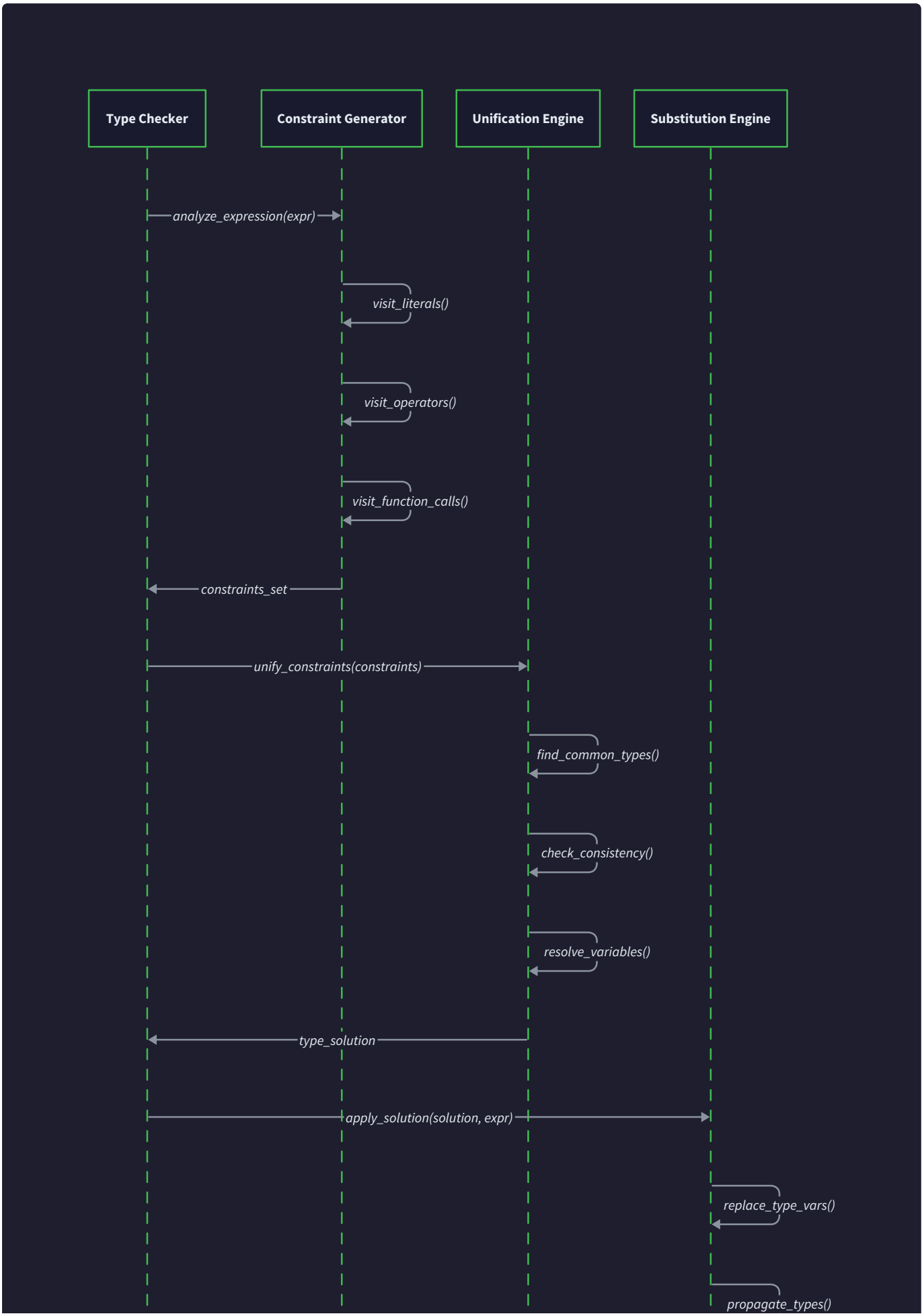


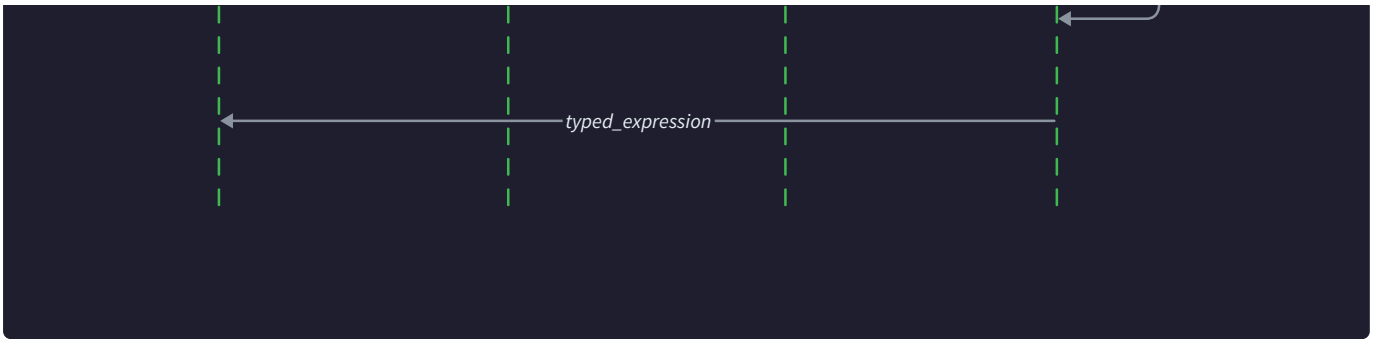
The component interactions represent one of the most critical aspects of our type checker architecture. Unlike simple sequential processing, type checking requires sophisticated coordination between components

because type information flows bidirectionally — inference generates constraints that flow forward to unification, while solved substitutions flow backward to update expressions. Additionally, the type environment must thread through recursive calls while maintaining proper scoping semantics, and error recovery must allow continued processing even when individual components fail.

Type Checking Pipeline

The type checking pipeline represents the **main arteries of our type system** — much like how blood circulation carries oxygen and nutrients throughout the body while removing waste products. Each stage in our pipeline receives typed information, processes it according to specific rules, potentially generates constraints or errors, and passes enriched information to the next stage. The pipeline must handle both the happy path (successful type checking) and error paths (type mismatches, unification failures) while maintaining enough state to provide meaningful error messages and continue processing.


Type Checker
Constraint Generator
Unification Engine
Substitution Engine
analyze_expression(expr)
visit_literals()
visit_operators()
visit_function_calls()
constraints_set
unify_constraints(constraints)
find_common_types()
check_consistency()
resolve_variables()
type_solution
apply_solution(solution, expr)
replace_type_vars()
propagate_types()



The pipeline operates through a series of distinct phases, each with specific responsibilities and interfaces. Understanding this flow is crucial because it determines how components communicate, where errors can occur, and how recovery mechanisms work. The pipeline is not purely sequential — some phases may iterate or backtrack, particularly during constraint solving and error recovery.

Pipeline Phase Breakdown

The complete type checking pipeline consists of several interconnected phases, each building upon the previous stages while potentially feeding information back to earlier components.

Phase	Input	Output	Primary Component	Error Recovery
AST Preprocessing	Raw AST from parser	Annotated AST with source locations	Type Environment	Continue with location-less errors
Environment Setup	Annotated AST	Initial type environment with built-ins	Type Environment	Fatal - cannot proceed
Expression Analysis	AST expressions + environment	Typed expressions + constraints	Basic Type Checker / Inference Engine	Insert error types, continue
Constraint Generation	Typed expressions	Type constraint sets	Inference Engine	Skip malformed expressions
Constraint Solving	Type constraints	Substitutions	Unification Algorithm	Report unsatisfiable constraints
Type Substitution	Expressions + substitutions	Fully typed AST	Type Representation	Use original types for failures
Error Formatting	Accumulated errors	Human-readable messages	Error Reporter	Best-effort formatting
Final Validation	Fully typed AST	Validated program or error report	Pipeline Coordinator	Report validation failures

> **Design Insight**: The pipeline uses a "collect errors and continue" strategy rather than failing fast. This allows developers to see multiple type errors in a single compilation, significantly improving the development experience.

Phase 1: AST Preprocessing and Environment Setup

The pipeline begins with preprocessing the input AST to ensure all necessary metadata is available for type checking. This phase serves as the **foundation layer** where we establish the ground truth about program structure and prepare the initial typing context.

AST Preprocessing Steps:

1. **Source Location Attachment**: Every AST node receives complete source location information including file name, line/column start and end positions. This metadata becomes crucial for error reporting but doesn't affect type checking logic.
2. **Built-in Type Registration**: The initial type environment is populated with primitive types (`TInt`, `TBool`, `TString`) and any built-in functions or operators. This creates the base context that all subsequent typing builds upon.
3. **Scope Structure Analysis**: The preprocessor identifies all binding constructs (let expressions, function parameters, lambda bindings) and creates a scope nesting structure. This analysis ensures proper environment threading during the main type checking phase.
4. **Declaration Dependency Analysis**: For languages with forward references or mutual recursion, this step builds a dependency graph to determine the order of type checking. This prevents issues where a binding is used before its type is established.

The environment setup creates the initial `env` structure that threads through all subsequent pipeline phases:

Environment Component	Purpose	Example Contents
Global Scope	Built-in types and functions	`[("int", Monomorphic TInt), ("+", Monomorphic (TFun (TInt, TFun (TInt, TInt))))]`
Module Scope	Top-level definitions	`[("main", Monomorphic (TFun (TString, TInt)))]`
Empty Local Scopes	Placeholder for nested bindings	`[]` (added as needed during traversal)

> **Critical Implementation Note**: The environment setup must complete successfully before any expression analysis begins. Failures here (like missing built-in definitions) are fatal and should terminate the pipeline with a clear setup error message.

Phase 2: Expression Analysis and Constraint Generation

Expression analysis forms the **core reasoning engine** of our pipeline. This phase traverses the AST, applying type rules to expressions while generating constraints that capture type relationships. The process combines immediate type checking (for expressions with known types) with constraint generation (for expressions requiring inference).

The analysis operates through a mutual recursion between `synthesize_expression_type` (determining what type an expression produces) and `check_expression_type` (verifying an expression against an expected type). This bidirectional approach allows the type checker to leverage context from both directions, improving inference quality and error messages.

Expression Analysis Algorithm:

1. **Expression Classification**: Each expression is classified as either synthesizable (we can determine its type from structure alone) or checkable (requires expected type context). Literals, variables, and function calls are synthesizable; lambda bodies and conditional branches are often checkable.
2. **Type Rule Application**: For synthesizable expressions, apply the corresponding type rule to compute the result type. For example, integer literals produce `TInt`, variable references require environment lookup, function calls require parameter/argument matching.
3. **Constraint Collection**: When type relationships cannot be resolved immediately (especially with type variables), generate constraints that capture the required relationships. A function call `f(x)` where `f` has type `α → β` and `x` has type `γ` generates the constraint `α = γ`.
4. **Recursive Subexpression Processing**: Process all subexpressions recursively, threading the type environment appropriately for scoping rules and collecting constraints from all nested expressions.
5. **Type Variable Management**: Generate fresh type variables for unknowns and maintain substitution contexts to ensure consistency across constraint generation.

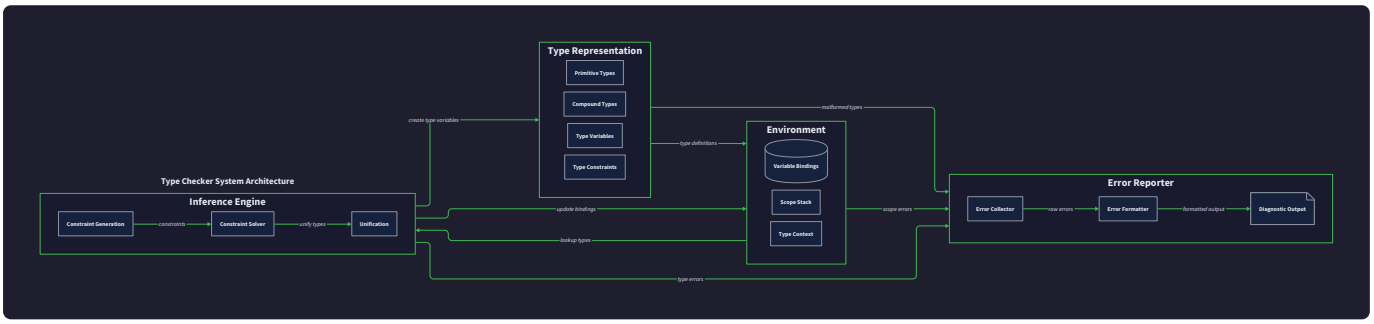
Constraint Generation Examples:

Expression Type	Generated Constraints	Rationale
Function Call `f(arg)`	`typeof(f) = α → β`, `typeof(arg) = α`, result type `β`	Function type must match argument type
Conditional `if c then e1 else e2`	`typeof(c) = TBool`, `typeof(e1) = typeof(e2)`	Condition must be boolean, branches must match
Let Binding `let x = e1 in e2`	Constraints from `e1`, then check `e2` with `x : typeof(e1)` in environment	Binding creates new scope with inferred type
Binary Operation `e1 + e2`	`typeof(e1) = TInt`, `typeof(e2) = TInt`, result type `TInt`	Both operands must be integers

The constraint generation process must handle **polymorphic instantiation** carefully. When a polymorphic binding like `id : ∀α. α → α` is referenced, the constraint generator must create fresh type variables and instantiate the type scheme, generating `id : β → β` where `β` is fresh.

Phase 3: Constraint Solving and Unification

Constraint solving represents the **puzzle-solving heart** of type inference. This phase takes the collection of type constraints generated from expressions and attempts to find a consistent assignment of concrete types to all type variables. The process uses Robinson unification with an occurs check to ensure termination and detect impossible type requirements.



The unification algorithm processes constraints iteratively, building up a substitution that represents the solution. Each constraint `t1 = t2` is unified by finding the most general substitution that makes the types equal, then composing this substitution with previous solutions.

Unification Algorithm Steps:

1. **Constraint Queue Initialization**: All generated constraints are added to a processing queue. The order can affect performance but not correctness due to the mathematical properties of unification.
2. **Constraint Selection**: Select the next constraint from the queue. Simple heuristics like processing variable-to-concrete constraints first can improve efficiency.
3. **Constraint Decomposition**: Break down complex constraints into simpler ones. For example, `TFun(α, β) = TFun(TInt, γ)` becomes two constraints: `α = TInt` and `β = γ`.
4. **Occurs Check**: Before binding a type variable `α` to a type `t`, verify that `α` does not occur free in `t`. This prevents infinite types like `α = α → β`.
5. **Substitution Generation**: Create a binding `α ↦ t` and compose it with the existing substitution. Apply the new substitution to all remaining constraints in the queue.
6. **Termination Check**: Continue until the queue is empty (success) or an unsatisfiable constraint is encountered (failure).

Unification Failure Cases:

Failure Type	Example	Error Message Strategy
Type Constructor Mismatch	`TInt = TBool`	"Expected integer, but got boolean"
Arity Mismatch	`TFun(TInt, TBool) = TFun(TInt, TBool, TString)`	"Function expects 2 arguments, got 3"
Occurs Check Violation	`α = α → TInt`	"Cannot construct infinite type"
Rigid Type Clash	`TInt = TString` in function signature	"Function signature requires string, expression produces integer"

> **Critical Design Decision**: When unification fails, the system must decide whether to abort type checking or continue with error recovery. Our pipeline chooses continuation with placeholder types, allowing detection of additional errors in the same program.

Phase 4: Type Substitution and Final Resolution

The type substitution phase **applies the solution** found by unification back to the original program AST. This phase transforms the constraint-laden intermediate representation into a fully typed AST where every expression has a concrete, resolved type.

Substitution Application Process:

1. **Global Substitution Application**: Apply the final substitution to every type annotation in the AST. Type variables are replaced with their solved concrete types, while already-concrete types remain unchanged.
2. **Type Scheme Instantiation**: For polymorphic bindings, ensure that the generalized type schemes are properly instantiated at each use site. This prevents inappropriate sharing of type variables across different uses of the same polymorphic value.
3. **Remaining Variable Detection**: Any type variables that remain after substitution application indicate inference failures or underconstrainted programs. These should be reported as ambiguous type errors.
4. **Type Annotation Insertion**: Generate explicit type annotations for expressions that were originally unannotated but now have inferred types. This produces a fully annotated program suitable for code generation or further analysis.

The substitution application must preserve the **semantic equivalence** between the original program and the typed program. Only type information should change; the computational behavior must remain identical.

Pipeline Error Handling Strategy

The pipeline implements a **resilient error handling strategy** that balances early error detection with comprehensive error reporting. Rather than terminating on the first error, the system attempts to recover and

continue analysis to surface as many issues as possible in a single compilation run.

> **Decision: Continuation-Based Error Recovery**
>
> - **Context**: Type errors often cascade, where one mistake causes many spurious downstream errors. However, failing fast means developers see only the first error and must fix-and-recompile repeatedly.
> - **Options Considered**:
>   1. Fail-fast: Stop on first error
>   2. Collect-and-continue: Insert placeholder types and continue
>   3. Speculative-recovery: Try multiple possible fixes and continue with the best
> - **Decision**: Collect-and-continue with intelligent error filtering
> - **Rationale**: Maximizes useful feedback per compilation while avoiding overwhelming cascading errors. Placeholder types (`TError`) allow continued analysis without affecting correctness of subsequent phases.
> - **Consequences**: More complex error recovery logic, but significantly better developer experience. Requires careful error filtering to avoid spurious cascading errors.

Error Recovery Mechanisms:

Error Type	Recovery Strategy	Placeholder Type	Continued Analysis
Unbound Variable	Insert error binding in environment	`TVar("error_" ^ fresh_id)`	Continue with error type
Type Mismatch	Use expected type for expression	Expected type	Continue normally
Unification Failure	Use one of the conflicting types	First type in constraint	Continue with warning
Missing Function Argument	Insert dummy argument with fresh type variable	`TVar(fresh_var())`	Analyze remaining arguments
Extra Function Argument	Ignore extra argument	N/A	Report arity error, continue

Environment Threading

Environment threading represents the **nervous system** of our type checker — much like how the human nervous system carries signals throughout the body while maintaining proper routing and timing. The type environment must flow through every recursive call in the type checking process, ensuring that variable bindings are properly scoped, new bindings are visible in appropriate contexts, and lexical scoping rules are

enforced correctly. Unlike simple parameter passing, environment threading requires careful attention to when environments are extended, how they are restored, and how bindings interact with polymorphism and constraint solving.

The complexity of environment threading arises from the interaction between lexical scoping (variables are visible in nested scopes), type inference (new bindings might have polymorphic types), and error recovery (malformed bindings shouldn't break subsequent analysis). The environment must evolve as type checking progresses while maintaining the invariant that variable lookups always find the appropriate binding according to the language's scoping rules.

Environment Structure and Scoping Rules

The environment structure implements **lexical scoping with nested scope layers**. Each scope layer represents a binding context (function parameters, let bindings, lambda parameters), and lookups search from innermost to outermost scope until a binding is found.

Our `env` type uses a list-of-lists structure where each inner list represents a scope level:

Environment Layer	Contents	Example Bindings
Built-in Scope (outermost)	Primitive types, built-in functions	`[("int", Monomorphic TInt), ("+", Monomorphic (TFun(TInt, TFun(TInt, TInt))))]`
Module Scope	Top-level function definitions	`[("factorial", Polymorphic (Forall(["a"], TFun(TInt, TInt))))]`
Function Parameter Scope	Current function's parameters	`[("n", Monomorphic TInt), ("acc", Monomorphic TInt)]`
Local Binding Scopes	Let bindings, lambda parameters	`[("x", Monomorphic TBool)]`

The scoping rules follow standard lexical scoping semantics:

1. **Innermost Binding Wins**: If multiple scopes contain bindings for the same variable name, the innermost (most recently added) binding shadows outer bindings.
2. **Scope Extension**: New bindings extend the innermost scope layer. Function calls and lambda expressions create entirely new scope layers.
3. **Scope Restoration**: When exiting a binding construct, the scope layers added during that construct are removed, restoring the previous environment state.
4. **Polymorphic Binding Instantiation**: Each lookup of a polymorphic binding creates a fresh instantiation, ensuring that different uses of the same polymorphic function don't interfere with each other.

Environment Operations and Threading Patterns

Environment threading follows specific patterns depending on the AST construct being processed. Understanding these patterns is crucial for correct type checker implementation, as improper threading leads to scoping violations or binding visibility errors.

Core Environment Operations:

Operation	Signature	Purpose	Usage Pattern
`extend_mono`	`string * ty * env -> env`	Add monomorphic binding	Local variables, function parameters
`extend_poly`	`string * type_scheme * env -> env`	Add polymorphic binding	Let-bound polymorphic functions
`lookup`	`string * env -> binding option`	Find variable binding	Variable references
`fresh_scope`	`env -> env`	Create new scope layer	Function calls, lambda expressions
`restore_scope`	`env * env -> env`	Restore previous scope	Exiting binding constructs

Threading Pattern Examples:

1. **Function Definition Threading**:
    - Extend environment with function name (for recursion)
    - Create fresh scope layer for parameters
    - Extend parameter scope with each parameter binding
    - Type check function body with extended environment
    - Restore original environment for subsequent definitions
2. **Let Expression Threading**:
    - Type check binding expression with current environment
    - Generalize inferred type if applicable
    - Extend environment with new binding
    - Type check body expression with extended environment
    - Use body type as let expression type
    - Environment extension persists if let is at top level
3. **Conditional Expression Threading**:
    - Type check condition with current environment

- Type check then-branch with current environment (no extension)
- Type check else-branch with current environment (no extension)
- Neither branch extends the environment for subsequent expressions

Polymorphic Binding Management

Polymorphic binding management adds significant complexity to environment threading because polymorphic bindings must be **generalized at definition sites** and **instantiated at use sites**. This process requires careful coordination between the environment, the generalization algorithm, and the instantiation mechanism.

> **Design Principle**: Generalization happens only at let bindings (let-polymorphism), while instantiation happens at every use of a polymorphic binding. This ensures predictable polymorphism without the complexity of full higher-rank types.

Generalization Process:

1. **Type Inference**: Infer the most general type for the let-bound expression using constraint generation and unification.
2. **Free Variable Collection**: Identify all type variables that appear in the inferred type but are not bound by the current environment. These variables are candidates for generalization.
3. **Value Restriction Check**: Verify that the let-bound expression is a syntactic value (variable, literal, or lambda). Non-values are not generalized to prevent unsound polymorphism with side effects.
4. **Type Scheme Construction**: Create a type scheme by quantifying over the free type variables: $\forall \alpha_1 ... \alpha_n .\ \tau$ where $\alpha_1 ... \alpha_n$ are the free variables and τ is the inferred type.
5. **Environment Extension**: Add the polymorphic binding to the environment using `extend_poly`.

Instantiation Process:

1. **Binding Lookup**: Retrieve the binding from the environment using `lookup`.
2. **Binding Classification**: Determine whether the binding is `Monomorphic ty` or `Polymorphic type_scheme`.
3. **Fresh Variable Generation**: For polymorphic bindings, generate fresh type variables for each quantified variable in the type scheme.
4. **Substitution Application**: Replace quantified variables with fresh variables throughout the type scheme body.
5. **Type Return**: Return the instantiated type for use in expression typing.

Environment Threading with Polymorphism:

Construct	Generalization	Instantiation	Environment Threading
Top-level Let	Generalize over environment-free variables	N/A	Extend global environment with polymorphic binding
Local Let	Generalize over environment-free variables	N/A	Extend local environment with polymorphic binding
Variable Reference	N/A	Instantiate polymorphic bindings with fresh variables	No environment change
Function Parameter	No generalization (monomorphic)	N/A	Extend parameter environment with monomorphic binding
Lambda Binding	No generalization (monomorphic)	N/A	Extend lambda environment with monomorphic binding

Error Recovery and Environment Consistency

Error recovery in environment threading requires maintaining **environment consistency invariants** even when individual type checking operations fail. The environment must remain in a valid state that allows continued analysis of subsequent program constructs.

⚠️ **Pitfall: Environment Corruption During Error Recovery** Many type checker implementations fail to properly maintain environment state when errors occur. For example, if type checking a let binding fails, the implementation might still add the binding to the environment with an invalid type, causing cascading errors. The correct approach is to add error bindings with placeholder types that don't interfere with subsequent analysis.

Error Recovery Strategies:

1. **Placeholder Binding Insertion**: When a binding cannot be properly typed, insert a binding with a fresh error type variable. This allows variable references to succeed while marking the error for reporting.
2. **Scope Restoration Guarantee**: Even if type checking within a scope fails, ensure that the environment is properly restored to its pre-scope state. Use exception handling or explicit cleanup to guarantee this invariant.
3. **Partial Environment Updates**: For complex binding constructs (like recursive function groups), apply successfully typed bindings to the environment while skipping failed bindings with appropriate error reporting.
4. **Environment Validation**: Periodically validate environment consistency, checking that all bindings are well-formed and that scope nesting is correct. This helps detect environment threading bugs during development.

Error Recovery Environment Operations:

Error Scenario	Recovery Action	Environment State	Continued Analysis
Let binding type error	Add `(name, Monomorphic (TVar "error_X"))`	Valid environment with error binding	Continue normally
Function parameter error	Add parameter with fresh error type	Valid parameter environment	Type check body with error types
Variable lookup failure	Report unbound variable error	Unchanged environment	Use fresh error type for expression
Scope corruption	Restore from saved environment snapshot	Previous valid state	Skip corrupted construct, continue
Recursive binding failure	Add partial bindings for successful cases	Partially extended environment	Continue with available bindings

> **Implementation Strategy**: Use environment snapshots at key points (before processing binding constructs) to enable rollback on catastrophic failures. This provides a safety net for environment threading bugs during development and testing.

Implementation Guidance

The component interactions and data flow represent the most complex aspect of type checker implementation because they require coordinating multiple interdependent systems while maintaining correctness invariants. This section provides concrete guidance for implementing the pipeline and environment threading mechanisms.

Technology Recommendations

Component	Simple Option	Advanced Option
Pipeline Orchestration	Direct function calls with explicit error handling	Monadic error handling with Result/Either types
Environment Threading	Explicit parameter passing	Reader monad or similar context-passing mechanism
Error Accumulation	Mutable error list with ref/array	Writer monad or immutable error collection
Constraint Management	List of constraint pairs	Priority queue with constraint selection heuristics
Substitution Application	Recursive tree traversal	Generic type traversal with fold operations

Recommended File Structure

```
src/
├── type_checker.ml              (* Main pipeline orchestrator *)
├── type_env.ml                  (* Environment operations and threading *)
├── constraint_solver.ml         (* Unification and constraint solving *)
├── type_inference.ml           (* Expression analysis and constraint generation *)
├── error_recovery.ml           (* Error handling and recovery mechanisms *)
├── pipeline_types.ml           (* Shared types for pipeline communication *)
└── tests/
    ├── test_pipeline.ml        (* End-to-end pipeline tests *)
    ├── test_env_threading.ml   (* Environment threading unit tests *)
    └── test_error_recovery.ml  (* Error recovery behavior tests *)
```

OCAML

Pipeline Orchestrator Implementation

The pipeline orchestrator coordinates the entire type checking process. This is infrastructure code that you should implement completely:

OCAML

```ocaml
(* pipeline_types.ml *)
type pipeline_result = {
  typed_ast: Ast.program option;
  errors: type_error list;
  warnings: string list;
}

type pipeline_config = {
  enable_error_recovery: bool;
  max_errors_before_abort: int;
  polymorphism_enabled: bool;
}

(* type_checker.ml *)
let default_config = {
  enable_error_recovery = true;
  max_errors_before_abort = 10;
  polymorphism_enabled = true;
}

let run_type_checker (config: pipeline_config) (ast: Ast.program): pipeline_result =
  let errors = ref [] in
  let warnings = ref [] in
  let add_error err = errors := err :: !errors in
  let add_warning msg = warnings := msg :: !warnings in

  try
    (* Phase 1: Environment Setup *)
```

```
    let initial_env = TypeEnv.create_with_builtins () in

    (* Phase 2: Expression Analysis *)
    let (typed_ast, constraints) = analyze_program ast initial_env add_error in

    (* Phase 3: Constraint Solving *)
    let subst = match solve_constraints constraints with
      | Ok s -> s
      | Error errs ->
          List.iter add_error errs;
          if config.enable_error_recovery then empty_subst else failwith "Unification
failed"
    in

    (* Phase 4: Type Substitution *)
    let final_ast = apply_substitution_to_ast subst typed_ast in

    {
      typed_ast = Some final_ast;
      errors = List.rev !errors;
      warnings = List.rev !warnings;
    }
  with
  | Pipeline_abort msg ->
      add_error (make_error Pipeline_abort None None None msg []);
      {
        typed_ast = None;
```

```
    errors = List.rev !errors;

    warnings = List.rev !warnings;

  }
```

Environment Threading Core Logic

The environment operations form the foundation of correct scoping. Implement these completely:

OCAML

```ocaml
(* type_env.ml *)

module TypeEnv = struct

  type t = env


  let empty = [[]]


  let create_with_builtins () =

    let builtins = [

      ("int", Monomorphic TInt);

      ("bool", Monomorphic TBool);

      ("string", Monomorphic TString);

      ("+", Monomorphic (TFun (TInt, TFun (TInt, TInt))));

      ("==", Polymorphic (Forall (["a"], TFun (TVar "a", TFun (TVar "a", TBool)))));

    ] in

    [builtins]


  let extend_mono (name: string) (ty: ty) (env: t): t =

    match env with

    | [] -> failwith "Environment invariant violated: empty environment"

    | scope :: rest -> ((name, Monomorphic ty) :: scope) :: rest


  let extend_poly (name: string) (scheme: type_scheme) (env: t): t =

    match env with

    | [] -> failwith "Environment invariant violated: empty environment"

    | scope :: rest -> ((name, Polymorphic scheme) :: scope) :: rest


  let fresh_scope (env: t): t =

    [] :: env


  let rec lookup (name: string) (env: t): binding option =
```

```
    match env with
    | [] -> None
    | scope :: rest ->
        (match List.assoc_opt name scope with
         | Some binding -> Some binding
         | None -> lookup name rest)

  let snapshot (env: t): t = env


  let restore (_snapshot: t) (env: t): t = env
end
```

Core Pipeline Logic Skeleton

This is the core logic that learners should implement themselves:

OCAML

```ocaml
(* type_inference.ml *)

let analyze_program (prog: Ast.program) (env: TypeEnv.t) (error_callback: type_error ->
unit): (Ast.typed_program * constraint_t list) =

  (* TODO 1: Initialize constraint accumulator and type variable generator *)

  (* TODO 2: Process top-level definitions in dependency order *)

  (* TODO 3: For each definition, call analyze_definition and collect constraints *)

  (* TODO 4: Handle recursive definition groups specially *)

  (* TODO 5: Return typed AST and accumulated constraints *)

  failwith "Implement analyze_program"


let analyze_definition (def: Ast.definition) (env: TypeEnv.t) (error_callback: type_error -
> unit): (Ast.typed_definition * TypeEnv.t * constraint_t list) =

  (* TODO 1: Match on definition type (function, variable, type alias) *)

  (* TODO 2: For function definitions, create parameter environment *)

  (* TODO 3: Analyze function body with parameter environment *)

  (* TODO 4: Generalize inferred type if this is a let binding *)

  (* TODO 5: Extend environment with new binding *)

  (* TODO 6: Return typed definition, updated environment, and constraints *)

  failwith "Implement analyze_definition"


let synthesize_expression_type (expr: Ast.expression) (env: TypeEnv.t): check_result =

  (* TODO 1: Match on expression type *)

  (* TODO 2: For literals, return known type (TInt for integers, etc.) *)

  (* TODO 3: For variables, lookup in environment and instantiate if polymorphic *)

  (* TODO 4: For function calls, synthesize function and argument types, generate
constraints *)

  (* TODO 5: For binary operators, check operand types and return result type *)

  (* TODO 6: For conditionals, check condition is bool, branches have same type *)

  (* TODO 7: For let expressions, analyze binding, extend environment, analyze body *)
```

```
  (* TODO 8: Handle error cases with appropriate error reporting *)

  failwith "Implement synthesize_expression_type"


let check_expression_type (expr: Ast.expression) (expected: ty) (env: TypeEnv.t):
check_result =

  (* TODO 1: Synthesize actual type of expression *)

  (* TODO 2: Generate constraint that actual type equals expected type *)

  (* TODO 3: For lambda expressions, check against function type structure *)

  (* TODO 4: Return success if constraint is satisfiable, error otherwise *)

  (* TODO 5: Use bidirectional type checking for better error messages *)

  failwith "Implement check_expression_type"
```

Constraint Solving Implementation Skeleton

```ocaml
(* constraint_solver.ml *)

let solve_constraints (constraints: constraint_t list): (substitution, type_error list)
result =

  (* TODO 1: Initialize empty substitution and constraint queue *)

  (* TODO 2: Apply existing substitution to all constraints *)

  (* TODO 3: Select next constraint from queue (use simple FIFO initially) *)

  (* TODO 4: Call unify_one to solve selected constraint *)

  (* TODO 5: Compose resulting substitution with existing solution *)

  (* TODO 6: Apply new substitution to remaining constraints *)

  (* TODO 7: Repeat until queue is empty or unsatisfiable constraint found *)

  (* TODO 8: Return final substitution or accumulated errors *)

  failwith "Implement solve_constraints"


let unify_one (constraint: constraint_t) (subst: substitution): (substitution, type_error)
result =

  let (t1, t2) = constraint in

  (* TODO 1: Apply existing substitution to both types *)

  (* TODO 2: If types are already equal, return empty substitution *)

  (* TODO 3: If one type is a variable, perform occurs check then bind *)

  (* TODO 4: If both are compound types, decompose into sub-constraints *)

  (* TODO 5: If types are incompatible, return type mismatch error *)

  (* Hint: Use type_equal for structural equality *)

  (* Hint: Use occurs_check before binding variables *)

  failwith "Implement unify_one"


let apply_substitution_to_ast (subst: substitution) (ast: Ast.typed_program):
Ast.typed_program =

  (* TODO 1: Traverse entire AST structure *)
```

```
  (* TODO 2: For each type annotation, apply substitution *)
  (* TODO 3: Replace type variables with their concrete types *)
  (* TODO 4: Leave non-variable types unchanged *)
  (* TODO 5: Ensure no type variables remain in final AST *)
  failwith "Implement apply_substitution_to_ast"
```

Environment Threading Test Cases

```ocaml
(* test_env_threading.ml *)
let test_basic_scoping () =
  let env = TypeEnv.create_with_builtins () in
  let env = TypeEnv.extend_mono "x" TInt env in
  let env = TypeEnv.fresh_scope env in
  let env = TypeEnv.extend_mono "x" TBool env in
  assert (TypeEnv.lookup "x" env = Some (Monomorphic TBool));
  (* TODO: Add more scoping test cases *)

let test_polymorphic_instantiation () =
  let env = TypeEnv.create_with_builtins () in
  let id_scheme = Forall (["a"], TFun (TVar "a", TVar "a")) in
  let env = TypeEnv.extend_poly "id" id_scheme env in
  (* TODO: Test that each lookup creates fresh instantiation *)
  (* TODO: Verify that different uses don't interfere *)

let test_error_recovery () =
  (* TODO: Test environment state after various error scenarios *)
  (* TODO: Verify that errors don't corrupt subsequent analysis *)
```

Milestone Checkpoints

After Implementing Pipeline Structure:

- Run: `ocamlfind ocamlc -package ... -linkpkg -I _build src/type_checker.cmo test/test_pipeline.ml -o test_pipeline && ./test_pipeline`
- Expected: Basic pipeline processes simple expressions without crashing
- Verify: Error handling prevents exceptions from escaping pipeline
- Debug: If pipeline crashes, check environment initialization and constraint collection

After Implementing Environment Threading:

- Run: `make test_env` or equivalent build command for environment tests
- Expected: Variable scoping works correctly for nested let expressions
- Verify: Polymorphic functions can be used at multiple types
- Debug: If scoping is wrong, trace environment extension/restoration calls

After Implementing Constraint Solving:

- Run: `make test_inference` for end-to-end inference tests
- Expected: Simple programs type check without explicit annotations
- Verify: Type errors are reported with clear messages and source locations
- Debug: If unification fails, trace constraint generation and solving steps

Common Debugging Issues

Symptom	Likely Cause	How to Diagnose	Fix
Variables not found in scope	Environment threading bug	Add logging to extend/lookup calls	Fix environment restoration after scope exit
Polymorphic functions monomorphic	Instantiation not happening	Check lookup implementation	Ensure binding instantiation on every lookup
Type variables in final AST	Substitution application incomplete	Check constraint solving result	Apply substitution to all AST nodes
Cascading type errors	Error recovery inserting bad types	Trace error recovery actions	Use fresh error variables for failed bindings
Pipeline hangs or crashes	Exception escaping error handling	Add exception logging	Wrap all pipeline phases in try-catch

Error Handling and Edge Cases

> **Milestone(s):** Milestone 2 (Basic Type Checking), Milestone 3 (Type Inference), Milestone 4 (Polymorphism)

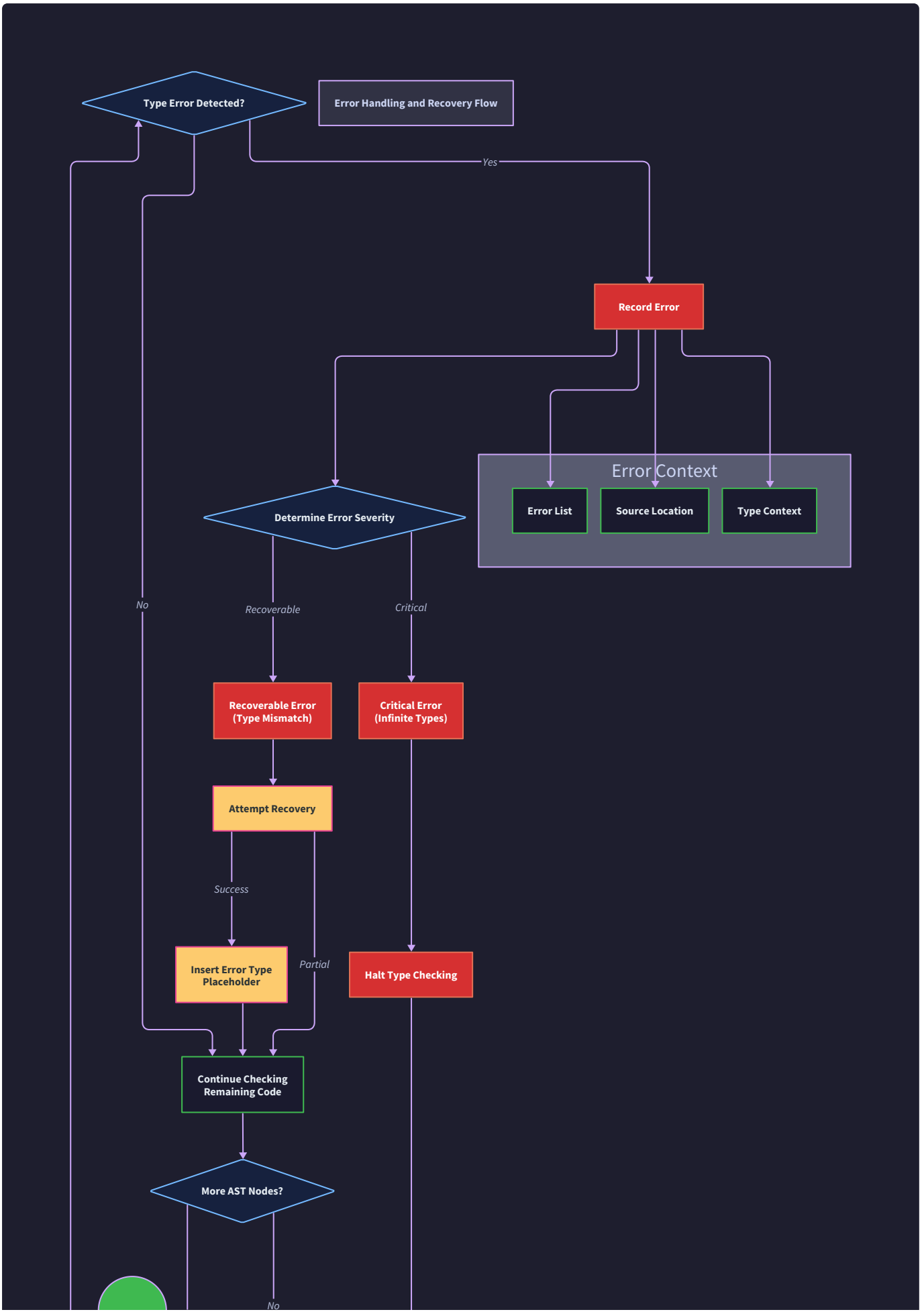
Think of error handling in a type checker as an **emergency response system** — much like how a hospital emergency room must simultaneously treat multiple patients while maintaining the capacity to help new arrivals. When the type checker encounters an error, it can't simply stop and give up; it must provide meaningful diagnosis, continue checking to find additional problems, and gracefully handle edge cases that could cause the system to crash or produce infinite loops.

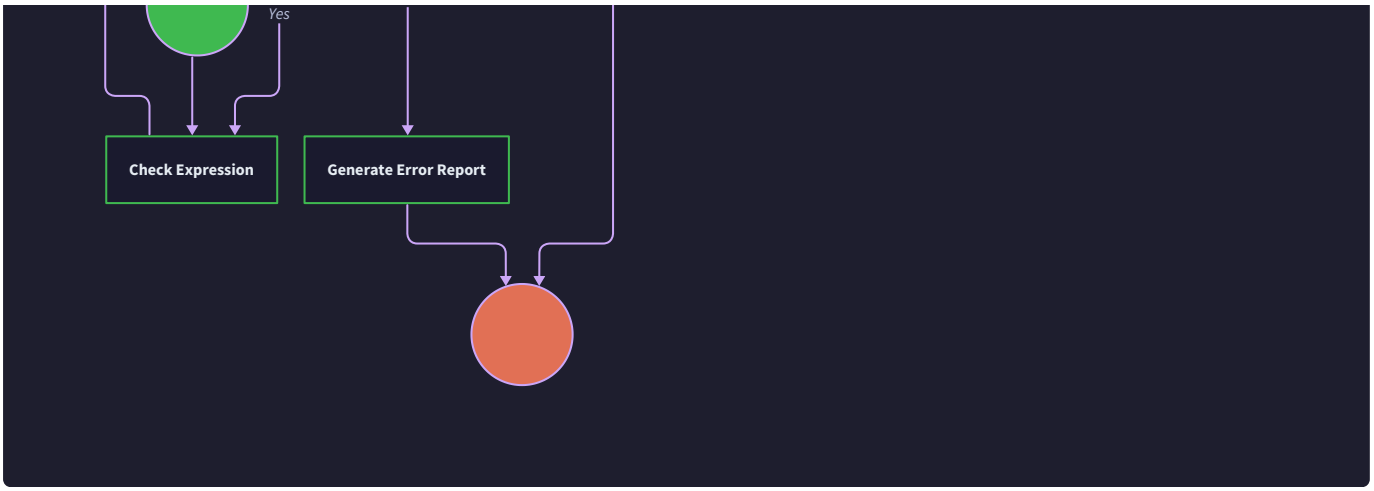
The fundamental challenge in type checker error handling is balancing precision with resilience. We need to detect genuine type violations while avoiding cascading errors that obscure the root cause. Additionally, certain algorithmic edge cases like infinite types during unification can cause the system to loop forever if not properly detected.

Mental Model: Medical Diagnosis and Treatment

Consider how an experienced emergency physician approaches multiple trauma patients. First, they perform triage — identifying the most critical issues that need immediate attention. Then, they treat each problem systematically while monitoring for complications. If one treatment fails, they don't abandon the patient; they try alternative approaches and continue addressing other issues. Finally, they document everything clearly so other physicians can understand what happened and what treatments were attempted.

Our type checker follows this same principle: detect errors early (triage), provide clear diagnostics (documentation), attempt recovery when possible (alternative treatments), and continue checking to find additional issues (treating multiple problems simultaneously).


Error Handling and Recovery Flow
Type Error Detected?
Yes
No
Record Error
Error Context
Error List
Source Location
Type Context
Determine Error Severity
Recoverable
Critical
Recoverable Error (Type Mismatch)
Critical Error (Infinite Types)
Attempt Recovery
Success
Partial
Insert Error Type Placeholder
Halt Type Checking
Continue Checking Remaining Code
More AST Nodes?
No



Error Classification and Recovery Strategy

The type checker encounters several categories of errors, each requiring different recovery strategies to enable continued checking:

Error Category	Examples	Recovery Strategy	Impact on Continued Checking
Type Mismatches	`int` used where `bool` expected	Assume expected type for expression	Continue with expected type
Unbound Variables	Variable `x` not in scope	Create fresh type variable	Continue with placeholder type
Arity Mismatches	Function expects 2 args, given 3	Check available arguments	Continue with partial application
Non-Function Calls	Calling `int` as function	Treat as function returning fresh var	Continue with return type
Infinite Types	`'a = 'a -> int` during unification	Reject unification	Continue with original types
Constraint Solving Failures	Unsatisfiable constraint set	Report conflicts, use partial solution	Continue with best-effort types

The core design principle is **graceful degradation**: when the type checker encounters an error, it should provide the most specific error message possible while making reasonable assumptions that allow type checking to continue. This approach maximizes the number of genuine errors found in a single checking pass.

> **Critical Insight**: The goal is not to produce a perfectly typed program after errors, but to find as many genuine type violations as possible while avoiding spurious errors caused by earlier failures.

Type Error Recovery

Type error recovery represents the type checker's ability to continue analysis after encountering type violations. The recovery mechanism must balance two competing goals: avoiding cascading errors that hide real problems, and maintaining enough context to perform meaningful checking of subsequent code.

Error Recovery Architecture

The error recovery system operates through three interconnected mechanisms working together:

Error Detection and Classification: When the type checker encounters a violation, it first classifies the error type and determines the appropriate recovery strategy. Each error kind has associated recovery rules that specify what placeholder types or assumptions to make.

Context Preservation: The recovery system maintains as much type environment context as possible, ensuring that well-typed portions of the program continue to be checked accurately. This prevents one error from invalidating the analysis of unrelated code sections.

Error Suppression Logic: The system tracks which errors are likely consequences of previous errors and suppresses reporting of these cascading failures. This prevents error floods that obscure the original problem.

Recovery Type Generation

When the type checker encounters an error, it must generate a placeholder type that allows continued checking. The `recover_from_error` function implements this logic:

Error Kind	Recovery Type Strategy	Rationale
`TypeMismatch`	Use expected type if available	Assumes programmer intended correct type
`UnboundVariable`	Fresh type variable	Allows constraint collection to continue
`WrongArity`	Function type with correct arity	Enables checking of provided arguments
`NonFunctionCall`	Function type returning fresh variable	Treats expression as callable
`InvalidOperation`	Result type of operation	Continues with operation's expected result

Error Context Threading

The type checker threads error information through the checking process using the `check_result` type. Each checking function returns both a type (for successful recovery) and a list of accumulated errors:

Result Variant	Type Field	Error List	Usage Pattern
`Success of ty`	Inferred/checked type	Empty list	Normal successful checking
`Error of type_error list`	Recovery type embedded	All detected errors	Failed checking with recovery

The threading mechanism allows errors from nested expressions to bubble up while maintaining the overall checking structure. Parent expressions receive both the recovery type (for continued checking) and the error list (for reporting).

Cascading Error Prevention

One of the most challenging aspects of error recovery is preventing cascading errors — spurious error reports caused by the propagation of earlier genuine errors. The system employs several strategies:

Error Source Tracking: Each type error includes source location information and context about what checking operation produced it. This allows the system to identify when multiple errors stem from the same root cause.

Conservative Error Reporting: When the checker encounters a type mismatch involving a recovery type (like a fresh type variable generated from an earlier error), it applies more lenient reporting rules. These secondary errors are often consequences rather than independent problems.

Error Suppression Windows: After reporting a significant error (like an unbound variable), the system temporarily suppresses related errors in the immediate syntactic vicinity. This prevents a single typo from generating dozens of follow-up error messages.

> **Design Decision: Conservative Recovery**
>
> - **Context**: Type checkers can either be aggressive (try to infer what the programmer meant) or conservative (make minimal assumptions)
> - **Options Considered**: 1) Aggressive inference with heuristics, 2) Conservative placeholder types, 3) Immediate failure on first error
> - **Decision**: Conservative recovery with placeholder types
> - **Rationale**: Aggressive inference often guesses wrong and hides real errors; immediate failure prevents finding multiple issues in one pass
> - **Consequences**: May require multiple compile cycles to fix all errors, but each reported error is likely genuine

Infinite Type Detection

Infinite type detection represents one of the most critical safety mechanisms in the unification algorithm. Without proper infinite type detection (the **occurs check**), the unification process can generate infinitely large type expressions that cause the type checker to run out of memory or loop forever.

Understanding Infinite Types

Consider the constraint `'a = 'a -> int` arising from a recursive definition where a variable is used as both a value and a function. If we naively tried to solve this constraint by substitution, we would generate:

1. `'a = 'a -> int`

2. Substitute first `'a`: `'a = ('a -> int) -> int`
3. Substitute again: `'a = (('a -> int) -> int) -> int`
4. Continue infinitely: `'a = ((('a -> int) -> int) -> int) -> int`

This infinite expansion would continue until the system runs out of memory. The occurs check prevents this by detecting when a type variable appears within its own solution before performing the substitution.

The Occurs Check Algorithm

The `occurs_check` function implements a depth-first traversal of a type expression to determine if a given type variable appears anywhere within it:

Type Expression	Occurs Check Behavior	Result
`TVar(name)`	Direct variable comparison	True if names match
`TInt`, `TBool`, `TString`	No variables present	Always false
`TFun(param, return)`	Recursive check on both param and return types	True if variable occurs in either
`TForall(vars, body)`	Check if variable is quantified, then check body	False if quantified, else recurse

The algorithm must handle quantified variables correctly — a variable quantified by a `TForall` is bound within that scope and doesn't represent the same variable being checked for occurrence.

Occurs Check Integration in Unification

The occurs check integrates into the unification algorithm at the point where we attempt to bind a type variable to a type expression. The check occurs before performing the substitution:

1. **Variable Binding Attempt**: Unification determines that type variable `'a` should be bound to type expression `t`
2. **Occurs Check**: Call `occurs_check("a", t)` to verify `'a` doesn't appear in `t`
3. **Binding Decision**: If occurs check fails, reject the unification; if it passes, proceed with binding
4. **Substitution Application**: Apply the binding `'a → t` to all remaining constraints

This integration point is critical because it catches infinite types before they can propagate through the constraint system.

Handling Occurs Check Failures

When the occurs check detects a potential infinite type, the unification algorithm must handle the failure gracefully:

Failure Context	Error Reporting	Recovery Strategy
Simple recursive binding	Report infinite type error with specific variables	Leave variables unbound
Complex constraint system	Report unsatisfiable constraints	Continue with partial solution
Polymorphic instantiation	Report instantiation failure	Use monomorphic fallback

The error reporting for infinite types requires careful explanation because the concept is subtle. The error message should explain both what constraint caused the problem and why the constraint represents an impossible type.

Edge Cases in Occurs Checking

Several edge cases require special handling in the occurs check implementation:

Quantified Variable Shadowing: When a type variable is quantified by a `TForall`, it shadows any variable with the same name in the outer scope. The occurs check must respect this scoping to avoid false positives.

Nested Function Types: Function types create nested structure where the same variable might appear in multiple positions. The occurs check must traverse all positions to detect any occurrence.

Partial Substitution Context: During constraint solving, some variables may already have partial substitutions applied. The occurs check must consider the fully expanded type after applying existing substitutions.

> **Critical Pitfall: Incomplete Occurs Check**
>
> - **Problem**: Only checking the immediate structure without recursing into nested types
> - **Consequence**: Missing infinite types in complex expressions like `'a = ('a * int) -> bool`
> - **Solution**: Implement complete recursive traversal of all type constructors
> - **Detection**: Unification loops or memory exhaustion during constraint solving

Performance Considerations

The occurs check adds overhead to unification because it requires traversing type expressions. However, this overhead is necessary for correctness:

Optimization Strategy	Benefit	Trade-off
Cached occurs results	Faster repeated checks	Memory usage for cache
Early termination	Stop on first occurrence found	None — pure optimization
Structural sharing	Reuse type expression nodes	Complex implementation

The most important optimization is early termination — as soon as the occurs check finds the target variable, it can return `true` without checking the remaining structure.

Common Pitfalls in Error Handling

⚠️ **Pitfall: Generating Cascading Errors** Many type checker implementations generate dozens of follow-up errors from a single root cause. For example, if variable `x` is misspelled as `y`, every subsequent use of `y` generates an "unbound variable" error. The fix is implementing error suppression logic that recognizes when errors are likely consequences of previous failures.

⚠️ **Pitfall: Inadequate Error Recovery Types** Using overly generic recovery types (like always generating `TInt`) can hide genuine errors in subsequent code. Instead, use context-appropriate recovery types — if expecting a function, generate a function type; if expecting a specific type, use that type.

⚠️ **Pitfall: Occurs Check Bypass** Implementing unification without a proper occurs check leads to infinite types that crash the system. Some developers try to optimize by skipping the check, but this is always incorrect. The occurs check is mandatory for sound unification.

⚠️ **Pitfall: Error Location Loss** Failing to preserve source location information through error recovery makes it impossible for users to locate problems. Every error should include precise location data, even when the error is detected several steps removed from its source.

⚠️ **Pitfall: Inconsistent Error Terminology** Using technical jargon or inconsistent terminology in error messages confuses users. Establish a consistent vocabulary for error reporting and always explain technical terms in user-friendly language.

Error Message Design Principles

Effective error messages in type checkers follow several key design principles:

Specificity Over Generality: Instead of "type error," specify exactly what types were involved and why they're incompatible. Include the expected type, actual type, and the operation that required compatibility.

Context Explanation: Explain not just what went wrong, but why the type checker expected a particular type. This helps users understand the reasoning behind type requirements.

Actionable Suggestions: When possible, suggest specific fixes. If a variable is unbound, suggest nearby variables with similar names. If types don't match, suggest explicit conversions or corrections.

Progressive Detail: Start with a clear summary, then provide additional detail for users who need it. This serves both beginners (who want simple explanations) and experts (who want precise technical information).

Error Component	Purpose	Example Content
Primary Message	Clear problem statement	"Cannot apply function of type `int -> bool` to argument of type `string`"
Context Explanation	Why this is required	"Function `f` expects an integer argument, but received a string"
Location Information	Where the error occurred	"at line 15, column 8 in file `main.ml`"
Suggested Fixes	Actionable remedies	"Consider using `int_of_string` to convert the argument"
Related Information	Connected context	"Function `f` was declared at line 5"

Integration with IDE Support

Modern type checkers must integrate effectively with development environments to provide real-time feedback. This requires careful design of the error handling system:

Incremental Error Reporting: The type checker should report errors as they're discovered rather than waiting for complete checking. This enables faster feedback cycles in interactive environments.

Error Categorization: Different error types require different IDE presentation. Syntax errors should be highlighted immediately, while type errors might be presented as squiggly underlines with hover information.

Quick Fix Integration: Error messages should include structured information that IDEs can use to offer automatic fixes. This requires embedding fix suggestions in a machine-readable format alongside human-readable messages.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
Error Types	Variant types with string messages	Structured error types with rich metadata
Location Tracking	Line/column pairs	Full source span with character ranges
Error Recovery	Fixed placeholder types	Context-sensitive type synthesis
Message Formatting	Template strings	Structured formatting with syntax highlighting

Recommended Module Structure

```
src/
  error/
    types.ml              ← Error type definitions
    recovery.ml           ← Error recovery logic
    reporting.ml          ← Error message formatting
    occurs_check.ml       ← Infinite type detection
  checker/
    unification.ml        ← Integration with unification
    inference.ml          ← Integration with inference
  utils/
    source_location.ml    ← Location tracking utilities
```

Error Type Infrastructure (Complete Implementation)

```ocaml
(* types.ml - Complete error handling types *)

type source_location = {
  file : string;
  line : int;
  column : int;
  end_line : int;
  end_column : int;
}

type error_kind =
  | TypeMismatch
  | UnboundVariable of string
  | WrongArity of int * int  (* expected, actual *)
  | NonFunctionCall
  | InvalidOperation of string

type type_error = {
  error_kind : error_kind;
  source_location : source_location;
  expected_type : ty option;
  actual_type : ty option;
  message : string;
  suggestions : string list;
}

type check_result =
  | Success of ty
```

```
  | Error of type_error list

let make_error kind loc expected actual message suggestions = {
  error_kind = kind;
  source_location = loc;
  expected_type = expected;
  actual_type = actual;
  message = message;
  suggestions = suggestions;
}

let combine_results r1 r2 f = match r1, r2 with
  | Success t1, Success t2 -> Success (f t1 t2)
  | Success _, Error errs -> Error errs
  | Error errs, Success _ -> Error errs
  | Error errs1, Error errs2 -> Error (errs1 @ errs2)
```

Occurs Check Implementation (Complete)

OCAML

```ocaml
(* occurs_check.ml - Complete occurs check implementation *)

let rec occurs_check var_name ty = match ty with
  | TVar name -> String.equal var_name name
  | TInt | TBool | TString -> false
  | TFun (param_ty, return_ty) ->
      occurs_check var_name param_ty || occurs_check var_name return_ty
  | TForall (quantified_vars, body_ty) ->
      (* Variable is bound by quantification - doesn't occur free *)
      if List.mem var_name quantified_vars then false
      else occurs_check var_name body_ty

let safe_unify_var var_name target_type =
  if occurs_check var_name target_type then
    None  (* Infinite type detected *)
  else
    Some [(var_name, target_type)]  (* Safe binding *)
```

Error Recovery System (Core Logic Skeleton)

```ocaml
(* recovery.ml - Error recovery logic *)

let recover_from_error error_kind expected_type = match error_kind with
  (* TODO 1: For TypeMismatch, return expected_type if available, else fresh var *)
  | TypeMismatch ->
    (* TODO: Check if expected_type is Some, return it, else fresh_type_var() *)

  (* TODO 2: For UnboundVariable, create fresh type variable *)
  | UnboundVariable var_name ->
    (* TODO: Generate fresh variable, optionally record original name *)

  (* TODO 3: For WrongArity, create function type with correct arity *)
  | WrongArity (expected_arity, actual_arity) ->
    (* TODO: Create function type with expected_arity parameters *)

  (* TODO 4: For NonFunctionCall, create function type returning fresh var *)
  | NonFunctionCall ->
    (* TODO: Create TFun with single parameter and fresh return type *)

  (* TODO 5: For InvalidOperation, return appropriate result type for operation *)
  | InvalidOperation op_name ->
    (* TODO: Match on operation name to determine result type *)

let generate_suggestions error_kind env = match error_kind with
  (* TODO 1: For unbound variables, find similar names in environment *)
  | UnboundVariable var_name ->
    (* TODO: Search env for names with small edit distance from var_name *)
```

```
  (* TODO 2: For type mismatches, suggest conversions *)
  | TypeMismatch ->
    (* TODO: Based on expected/actual types, suggest conversion functions *)

  (* TODO 3: For arity errors, suggest parameter adjustments *)
  | WrongArity (expected, actual) ->
    (* TODO: Suggest adding or removing parameters *)

  | _ -> []

let format_error error =
  (* TODO 1: Format primary error message based on error_kind *)
  (* TODO 2: Add location information with file:line:column format *)
  (* TODO 3: Include type information if available *)
  (* TODO 4: Add suggestions with "Hint:" prefix *)
  (* TODO 5: Use consistent terminology from the glossary *)
  failwith "TODO: implement error formatting"
```

Integration with Type Checking (Core Logic Skeleton)

```ocaml
(* Integration points in checker modules *)

let synthesize_expression_type expr env =
  (* TODO 1: Pattern match on expression type *)
  (* TODO 2: For each case, perform type checking with error recovery *)
  (* TODO 3: On error, call recover_from_error and continue *)
  (* TODO 4: Accumulate errors from subexpressions *)
  (* TODO 5: Return Success with type or Error with accumulated errors *)
  failwith "TODO: implement expression synthesis with error recovery"

let unify_with_occurs_check t1 t2 =
  (* TODO 1: Check if either type is a variable *)
  (* TODO 2: If binding variable, perform occurs_check first *)
  (* TODO 3: If occurs check fails, return unification error *)
  (* TODO 4: If occurs check passes, proceed with normal unification *)
  (* TODO 5: Handle constraint decomposition for compound types *)
  failwith "TODO: implement unification with occurs check"
```

Milestone Checkpoints

After Error Recovery Implementation:

- Run `ocamlc -c error/recovery.ml` — should compile without warnings
- Test with intentional type errors — checker should continue and find multiple issues
- Verify error messages include location information and helpful suggestions
- Check that cascading errors are suppressed appropriately

After Occurs Check Implementation:

- Create test cases with recursive type constraints like `'a = 'a -> int`
- Verify that unification rejects these constraints instead of looping
- Test performance with deeply nested type expressions
- Ensure quantified variables are handled correctly in occurs check

Debugging Tips for Error Handling

Symptom	Likely Cause	How to Diagnose	Fix
Type checker crashes on recursive types	Missing occurs check	Add debug prints in unification	Implement proper occurs_check
Floods of similar error messages	No cascading error prevention	Check error context tracking	Add error suppression logic
Unhelpful "type error" messages	Generic error reporting	Examine error construction sites	Use specific error_kind variants
Errors reported at wrong locations	Location information not threaded	Trace location through checking pipeline	Preserve source_location in all contexts
Recovery produces worse errors later	Poor recovery type choice	Log recovery types and subsequent checking	Use context-appropriate recovery types

Testing Strategy and Milestones

> **Milestone(s):** Milestone 1 (Type Representation), Milestone 2 (Basic Type Checking), Milestone 3 (Type Inference), Milestone 4 (Polymorphism)

Think of testing a type checker as **quality assurance for a complex manufacturing plant** — much like how an automotive factory has inspection checkpoints at every stage of assembly to catch defects early, our type checker testing strategy establishes verification points at each milestone to ensure the foundation is solid before building the next layer. Just as a car manufacturer tests individual components (brakes, engine, transmission) before testing the complete vehicle, we systematically verify type representations, then basic checking, then inference, and finally polymorphism in isolation and combination.

The challenge with testing a type checker lies in its **multi-layered verification requirements**. Unlike testing a simple function that transforms input to output, a type checker must verify correctness across multiple dimensions simultaneously: syntactic correctness (does the code parse?), semantic correctness (do types match?), inference correctness (are inferred types most general?), and error quality (are error messages helpful?). Each milestone introduces new complexity that could break previously working functionality, making systematic regression testing essential.

Our testing strategy follows a **pyramid architecture** where unit tests form the broad base, integration tests verify component interactions, and end-to-end tests validate complete workflows. This mirrors how civil engineers test bridge components individually (steel strength, bolt tensile strength), then test assemblies (beam connections), and finally test the complete structure under load. The key insight is that type checker

bugs often manifest as cascading failures — a subtle bug in unification can cause polymorphism to fail mysteriously, so we need fine-grained tests that isolate the failure source.

Milestone Verification

Each milestone represents a **checkpoint gate** in our manufacturing analogy — a point where we must verify all functionality works correctly before proceeding. Think of this like software release gates in large companies, where automated tests must pass before code can be promoted to the next environment. The verification strategy ensures we catch integration issues early rather than debugging a complex system where the root cause could be anywhere.

Milestone 1 Verification focuses on the foundational type system infrastructure. At this stage, we're essentially testing the "vocabulary and grammar" of our type language — can we represent all necessary types correctly, can we compare them for equality, and can we manage type environments properly. This is analogous to testing a compiler's lexer and parser before attempting code generation.

Test Category	What to Verify	Acceptance Criteria	Red Flags to Watch For
Type Construction	All type variants can be created and accessed	`TInt`, `TBool`, `TString`, `TVar "a"`, `TFun(TInt, TBool)` all construct correctly	Segmentation faults or pattern match failures when accessing type fields
Type Equality	`type_equal` correctly identifies structurally identical types	`type_equal(TFun(TInt, TBool), TFun(TInt, TBool))` returns true	False negatives on identical types or false positives on different types
Type Variable Management	Fresh variables are unique and `fresh_type_var()` never repeats	Multiple calls produce `TVar "a1"`, `TVar "a2"`, etc. with different names	Variable name collisions or non-deterministic generation
Type Environment Operations	`TypeEnv.extend` and `TypeEnv.lookup` handle scoping correctly	Variables added to inner scope shadow outer scope, lookup searches inside-out	Variables leaking between scopes or lookup finding wrong binding
Type Pretty Printing	`string_of_type` produces readable output for debugging	`TFun(TInt, TBool)` formats as "int -> bool"	Cryptic output that makes debugging impossible

The critical insight for Milestone 1 is that **type representation bugs cascade through every subsequent milestone**. If `type_equal` has subtle bugs or type variable generation isn't truly unique, you'll see

mysterious failures in unification or polymorphism that are extremely difficult to debug. The verification must be exhaustive at this foundational layer.

Milestone 2 Verification tests the basic type checking engine without inference. Here we're testing the "contract enforcement" system — can we verify that explicitly typed code follows the type rules correctly? This is like testing a building inspector's ability to verify that construction matches the approved blueprints.

Test Category	What to Verify	Acceptance Criteria	Red Flags to Watch For
Expression Synthesis	`synthesize_expression_type` correctly determines expression types	Literals infer correct types, binary operations check operand compatibility	Wrong types inferred for literals or operators accepting incompatible types
Expression Checking	`check_expression_type` validates expressions against expected types	Type-annotated expressions verified correctly, mismatches produce clear errors	False positives accepting wrong types or false negatives rejecting correct types
Function Call Validation	Function applications check arity and argument types	Calling `f: int -> bool` with `int` argument succeeds, wrong arity fails	Functions accepting wrong argument types or arity mismatches going undetected
Type Error Generation	Errors include source locations and helpful messages	`make_error` produces errors with file/line information and suggested fixes	Generic error messages without context or missing source location tracking
Error Recovery	Type checking continues after errors to find multiple issues	Multiple type errors in same program all reported, not just first one	Type checking aborting on first error or cascading spurious errors

The key verification principle for Milestone 2 is **independent testability** — each type rule should be testable in isolation. If testing function calls requires complex setup of polymorphic types, there's too much coupling between milestones.

Milestone 3 Verification focuses on constraint-based type inference correctness. This is where testing becomes significantly more complex because we're verifying not just correctness but **optimality** — does the inference produce the most general type possible? Think of this like testing an optimization algorithm where correctness isn't binary but measured against the theoretical optimum.

Test Category	What to Verify	Acceptance Criteria	Red Flags to Watch For
Constraint Generation	`generate_constraints` produces correct equality constraints	`let x = 5 in x + 1` generates constraints `typeof(x) = int`, `typeof(+) = int -> int -> int`	Missing constraints that allow invalid programs or spurious constraints that over-constrain
Unification Algorithm	`unify` finds most general unifiers when they exist	Unifying `'a -> bool` with `int -> 'b` produces `['a := int, 'b := bool]`	Unification failing on solvable constraints or producing overly specific solutions
Occurs Check	Infinite types detected and prevented during unification	Unifying `'a` with `'a -> bool` fails with occurs check error	Infinite types accepted causing stack overflow or infinite loops later
Substitution Application	`apply_subst_to_type` correctly replaces variables throughout type	Applying `['a := int]` to `'a -> 'b` produces `int -> 'b`	Variables not replaced or replaced incorrectly in complex nested types
Inference Integration	Complete inference pipeline produces most general types	`let id = fun x -> x` infers `'a -> 'a` not overly specific type	Inference producing `int -> int` instead of polymorphic `'a -> 'a`

The critical testing insight for Milestone 3 is **most general type verification** — it's not enough to verify that inference produces a correct type; you must verify it produces the **most general** correct type. This requires understanding the theoretical optimum for each test case.

Milestone 4 Verification tests polymorphism with let-generalization and instantiation. This is the most complex milestone because polymorphism interacts with all previous systems. Think of this like testing a complex mechanical system where every component must work perfectly in isolation and in combination.

Test Category	What to Verify	Acceptance Criteria	Red Flags to Watch For
Type Scheme Operations	Type schemes correctly represent and manipulate quantified types	`make_type_scheme(["a"], TFun(TVar "a", TVar "a"))` creates proper forall type	Quantified variables not properly tracked or incorrectly captured
Generalization	`generalize_type` abstracts over appropriate free variables	`let id = fun x -> x` generalizes to `forall a. a -> a`	Over-generalization violating value restriction or under-generalization losing polymorphism
Instantiation	`instantiate_scheme` creates fresh instances with unique variables	Using polymorphic `id` twice gives `'a1 -> 'a1` and `'a2 -> 'a2` with different variables	Variable capture where instances share variables incorrectly
Value Restriction	Generalization correctly applies value restriction	`let r = ref []` stays monomorphic, doesn't generalize to `forall a. ref (list a)`	Unsafe generalization that could violate type soundness
Polymorphic Usage	Polymorphic functions can be used at multiple types in same program	`id 5` and `id true` both type check with same polymorphic `id`	Polymorphic functions locked to first usage type

> **Key Design Insight**: Milestone verification isn't just about testing new functionality — each milestone must also include **regression tests** that verify all previous milestone functionality still works. Polymorphism changes how types are represented and manipulated, which can subtly break basic type checking in hard-to-detect ways.

Test Case Categories

Testing a type checker requires **systematic coverage across multiple orthogonal dimensions** rather than ad-hoc test cases. Think of this like testing a complex chemical plant where you must verify correct operation under normal conditions, boundary conditions, error conditions, and combinations thereof. The challenge is ensuring comprehensive coverage without writing thousands of redundant test cases.

Our test categorization follows **black-box and white-box testing principles**. Black-box categories test the external behavior (does the type checker accept/reject programs correctly?), while white-box categories test internal algorithm correctness (does unification produce correct substitutions?). This dual approach catches both specification violations and implementation bugs.

Type Rule Verification tests the fundamental type system rules systematically. These tests verify that our type checker correctly implements the theoretical type rules from type theory literature. Think of these as testing the "constitutional law" of our type system — the foundational rules that define what programs are legal.

Rule Category	Test Focus	Example Test Cases	Expected Behavior
Literal Typing	Primitive values have correct built-in types	`5` has type `int`, `true` has type `bool`, `"hello"` has type `string`	Literal expressions synthesize correct primitive types
Variable Lookup	Variable references resolve to declared types	`let x : int = 5 in x` where `x` has type `int`	Variable expressions synthesize type from environment
Function Application	Function calls check argument/parameter compatibility	`(fun x : int -> x + 1) 5` where argument matches parameter	Function applications type check with compatible arguments
Function Abstraction	Lambda expressions have proper arrow types	`fun x : int -> x + 1` has type `int -> int`	Lambda expressions synthesize arrow types from parameter and body
Let Binding	Let expressions extend environment correctly	`let x = 5 in x + 1` where `x` available in body with inferred type	Let expressions extend environment for body evaluation
Binary Operations	Operators enforce operand type requirements	`5 + 3` succeeds, `5 + true` fails with type mismatch	Binary operators require compatible operand types
Conditional Expressions	If expressions require boolean condition and matching branches	`if true then 5 else 3` succeeds, `if 5 then true else false` fails	Conditionals enforce boolean condition and unified branch types

The systematic approach here tests **each type rule in isolation** before testing combinations. This is crucial because when a complex expression fails to type check, you need to know which specific rule is violated.

Type Inference Correctness tests the constraint generation and solving algorithms. These are white-box tests that verify the internal inference mechanisms work correctly. Think of these as testing the "reasoning engine" that deduces types from incomplete information.

Inference Category	Algorithm Focus	Example Test Cases	Expected Behavior
Constraint Generation	`generate_constraints` produces complete constraint sets	`let f = fun x -> x + 1` generates `typeof(x) = typeof(1) = int`, `typeof(+) = int -> int -> int`, etc.	All type relationships captured as constraints
Simple Unification	`unify` solves basic variable-type constraints	Unify `'a` with `int` produces substitution `['a := int]`	Basic unification produces correct substitutions
Complex Unification	`unify` handles nested type structures	Unify `'a -> 'b` with `int -> bool` produces `['a := int, 'b := bool]`	Structural unification decomposes and solves recursively
Unification Failure	`unify` detects unsolvable constraints	Unify `int` with `bool` fails with type mismatch error	Incompatible types detected and reported clearly
Occurs Check	Infinite type prevention during unification	Unify `'a` with `'a -> int` fails with occurs check error	Infinite types prevented with clear diagnostic
Substitution Composition	Multiple substitutions combine correctly	Composing `['a := int]` and `['b := 'a]` produces `['a := int, 'b := int]`	Substitution composition maintains consistency
Most General Inference	Complete inference produces optimal types	`fun x -> x` infers `'a -> 'a` not specific type like `int -> int`	Inference maximizes polymorphism where possible

The key insight for inference testing is **theoretical optimality verification** — you must know what the most general type should be for each test case and verify that inference achieves it.

Error Handling and Recovery tests verify that the type checker behaves gracefully when encountering invalid programs. These tests are crucial because in practice, most programs a type checker sees are incorrect during development. Think of this like testing a medical diagnostic system's behavior on edge cases and unusual symptoms.

Error Category	Error Scenario	Test Cases	Expected Recovery Behavior
Type Mismatch	Incompatible types in context	`5 + true`, `if 5 then 1 else 2`	Clear error message with expected vs actual types
Unbound Variables	Variable used before declaration	`x + 1` where `x` not in scope	Error with suggestion to declare variable
Arity Mismatch	Function called with wrong number of arguments	`(fun x y -> x + y) 5` missing second argument	Error showing expected vs actual argument count
Non-function Call	Attempting to call non-function value	`5(3)` trying to call integer	Error explaining that value is not a function
Multiple Errors	Program with several independent type errors	Function with multiple type mismatches	All errors reported, not just first one
Cascading Errors	Error recovery prevents spurious follow-on errors	Unbound variable doesn't cause every usage to error	Error recovery limits cascading failures
Error Context	Error messages include helpful source location and context	Errors show file, line, column, and surrounding code	Developers can quickly locate and understand errors

> **Critical Testing Principle**: Error recovery testing is just as important as correctness testing. A type checker that produces avalanches of spurious errors after encountering one real error is unusable in practice, even if it's theoretically correct on valid programs.

Polymorphism Integration tests verify that let-polymorphism works correctly in complex scenarios. These are the most sophisticated tests because polymorphism interacts with every other system component. Think of these as integration tests for a complex distributed system where timing and coordination matter.

Polymorphism Category	Integration Scenario	Test Cases	Expected Polymorphic Behavior
Let Generalization	Polymorphic bindings created at let sites	`let id = fun x -> x in (id 5, id true)`	`id` generalized to `forall a. a -> a` and usable at multiple types
Value Restriction	Generalization appropriately restricted for non-values	`let r = ref []` stays monomorphic	Potentially unsafe expressions not generalized
Instantiation	Polymorphic values instantiated with fresh variables	Using `id` twice creates independent type variable instances	Each usage gets fresh type variables avoiding capture
Nested Polymorphism	Polymorphic functions within polymorphic contexts	`let outer = fun f -> let inner = fun x -> f x in (inner 1, inner true)`	Nested generalization and instantiation work correctly
Recursive Functions	Polymorphic recursion with type annotations	`let rec length : forall a. list a -> int = ...`	Recursive functions can be polymorphic with proper annotation
Higher-Order Polymorphism	Polymorphic functions as arguments and results	`let twice f x = f (f x)` infers `('a -> 'a) -> 'a -> 'a`	Higher-order functions maintain polymorphism correctly

The polymorphism tests are **integration tests by nature** because they exercise the entire type checking pipeline with maximum complexity. These tests often reveal subtle bugs in earlier milestones that only manifest under polymorphic conditions.

Systematic Edge Case Coverage ensures robustness under unusual but legal conditions. Real-world type checkers must handle programs that push the boundaries of the type system. Think of these as stress tests that verify the system doesn't break under unusual load patterns.

Edge Case Category	Boundary Condition	Test Scenarios	Robustness Requirements
Deep Nesting	Deeply nested expressions and types	Function type with 50 nested arrows, expression with 100 nested lets	No stack overflow or performance degradation
Large Programs	Programs with many bindings and complex dependencies	1000+ variable bindings, complex dependency graphs	Reasonable performance and memory usage
Complex Type Expressions	Highly polymorphic types with many variables	Types with 20+ type variables, complex constraint systems	Unification and inference remain tractable
Unusual Valid Programs	Programs that are legal but uncommon	Functions returning functions returning functions, deeply curried applications	Correctness maintained even for unusual patterns
Boundary Conditions	Edge cases in algorithms	Empty let bodies, identity functions, constant functions	No special case failures

Implementation Guidance

The testing implementation strategy mirrors the milestone progression — build testing infrastructure incrementally, adding sophistication at each stage. Think of this like constructing a manufacturing quality control system where you start with basic inspection tools and gradually add more sophisticated testing equipment.

A. Technology Recommendations Table:

Testing Component	Simple Option	Advanced Option
Test Framework	OUnit2 with basic assertions	QCheck property-based testing with custom generators
Test Organization	Single test file per milestone	Hierarchical test suites with shared fixtures
Error Message Testing	String equality on error messages	Structured error comparison with custom matchers
Test Data Generation	Hand-written test cases	Property-based testing with random program generation
Performance Testing	Manual timing of test cases	Benchmark suite with regression detection
Coverage Analysis	Manual inspection of test coverage	Bisect coverage analysis with CI integration

B. Recommended File/Module Structure:

```
type_checker/
  src/
    types.ml              ← Core type definitions
    type_env.ml           ← Type environment implementation
    type_check.ml         ← Basic type checking engine
    type_infer.ml         ← Type inference engine
    polymorphism.ml       ← Polymorphic type system
  tests/
    test_types.ml         ← Milestone 1: Type representation tests
    test_basic_check.ml   ← Milestone 2: Basic type checking tests
    test_inference.ml     ← Milestone 3: Type inference tests
    test_polymorphism.ml  ← Milestone 4: Polymorphism tests
    test_integration.ml   ← Cross-milestone integration tests
    test_utils.ml         ← Shared testing utilities
    examples/
      valid/              ← Well-typed example programs
      invalid/            ← Ill-typed example programs with expected errors
  dune-project            ← Build configuration
  dune                    ← Test target configuration
```

This structure separates test concerns cleanly while enabling easy execution of milestone-specific test suites or comprehensive integration testing.

C. Testing Infrastructure Starter Code (COMPLETE, ready to use):

OCAML

```ocaml
(* test_utils.ml - Complete testing utilities *)

open Types

open Type_check


let assert_type_equal expected actual =

  if not (type_equal expected actual) then

    let msg = Printf.sprintf "Expected type %s but got %s"

              (string_of_type expected) (string_of_type actual) in

    OUnit2.assert_failure msg


let assert_type_error_kind expected_kind result =

  match result with

  | Success _ -> OUnit2.assert_failure "Expected type error but got success"

  | Error errors ->

    let has_expected_kind = List.exists (fun err ->

      match err.error_kind, expected_kind with

      | TypeMismatch, TypeMismatch -> true

      | UnboundVariable s1, UnboundVariable s2 -> String.equal s1 s2

      | WrongArity(e1, a1), WrongArity(e2, a2) -> e1 = e2 && a1 = a2

      | NonFunctionCall, NonFunctionCall -> true

      | InvalidOperation s1, InvalidOperation s2 -> String.equal s1 s2

      | _ -> false

    ) errors in

    if not has_expected_kind then

      let error_kinds = List.map (fun e ->

        match e.error_kind with

        | TypeMismatch -> "TypeMismatch"

        | UnboundVariable s -> "UnboundVariable(" ^ s ^ ")"
```

```
        | WrongArity(e, a) -> Printf.sprintf "WrongArity(%d, %d)" e a
        | NonFunctionCall -> "NonFunctionCall"
        | InvalidOperation s -> "InvalidOperation(" ^ s ^ ")"
      ) errors in
      OUnit2.assert_failure ("Expected error kind not found. Got: " ^ String.concat ", "
error_kinds)

let make_test_location () = {
  file = "test_input.ml";
  line = 1;
  column = 1;
  end_line = 1;
  end_column = 10;
}

let empty_env = TypeEnv.create ()

(* Helper to create simple test expressions *)
type test_expr =
  | Lit_int of int
  | Lit_bool of bool
  | Lit_string of string
  | Var of string
  | App of test_expr * test_expr
  | Lam of string * ty option * test_expr
  | Let of string * test_expr * test_expr
  | BinOp of string * test_expr * test_expr

let rec test_expr_to_real_expr = function
```

```
  | Lit_int i -> (* Convert to your actual AST type *)

  | Lit_bool b -> (* Convert to your actual AST type *)

  (* ... implement conversion to your actual expression AST *)
```

D. Core Logic Testing Skeleton (signature + TODOs only):

OCAML

```ocaml
(* test_types.ml - Milestone 1 type representation tests *)

open OUnit2

open Types

open Test_utils


let test_type_construction () =

  (* TODO 1: Test that TInt, TBool, TString construct correctly *)

  (* TODO 2: Test that TVar creates type variables with given names *)

  (* TODO 3: Test that TFun creates proper function types *)

  (* TODO 4: Test that complex nested types construct correctly *)

  (* Hint: Use assert_equal with custom printer for types *)


let test_type_equality () =

  (* TODO 1: Test that identical primitive types are equal *)

  (* TODO 2: Test that different primitive types are not equal *)

  (* TODO 3: Test that structurally identical function types are equal *)

  (* TODO 4: Test that type variables with same name are equal *)

  (* TODO 5: Test that type variables with different names are not equal *)

  (* Hint: Use type_equal function and assert_bool *)


let test_fresh_type_vars () =

  (* TODO 1: Generate multiple fresh type variables *)

  (* TODO 2: Assert that each has a unique name *)

  (* TODO 3: Assert that repeated calls never produce same name *)

  (* Hint: Generate 100 vars and check all names are distinct *)


let test_type_env_operations () =

  (* TODO 1: Test extending empty environment with single binding *)

  (* TODO 2: Test lookup finds recently added binding *)
```

```
  (* TODO 3: Test lookup returns None for unbound variables *)

  (* TODO 4: Test that inner scope shadows outer scope *)

  (* TODO 5: Test lookup searches scopes inside-out *)

  (* Hint: Create nested scopes and verify shadowing behavior *)


(* test_inference.ml - Milestone 3 constraint generation tests *)

let test_constraint_generation () =

  (* TODO 1: Generate constraints for simple literal expressions *)

  (* TODO 2: Verify that variable references generate lookup constraints *)

  (* TODO 3: Test function application generates argument/parameter constraints *)

  (* TODO 4: Test let binding generates appropriate binding constraints *)

  (* TODO 5: Verify binary operations generate operand type constraints *)

  (* Hint: Check both the inferred type and generated constraint list *)


let test_unification_algorithm () =

  (* TODO 1: Test unifying type variable with concrete type *)

  (* TODO 2: Test unifying two concrete identical types *)

  (* TODO 3: Test unifying two incompatible types fails *)

  (* TODO 4: Test unifying function types decomposes correctly *)

  (* TODO 5: Test occurs check prevents infinite types *)

  (* Hint: Verify both success/failure and resulting substitution *)
```

E. Language-Specific OCaml Hints:

- Use `OUnit2.assert_equal ~printer:string_of_type` for type comparison with readable failure messages
- Implement custom `pp_type` printer for better test output formatting
- Use `List.fold_left` to apply multiple constraints sequentially in unification tests
- Pattern match exhaustively on `check_result` variants to ensure both success and error cases are tested
- Use `QCheck.Gen.oneof` to generate random expressions for property-based testing

- Create helper functions like `make_simple_env` to reduce test boilerplate
- Use `Printf.sprintf` for dynamic error message generation in test assertions

F. Milestone Checkpoints:

Milestone 1 Checkpoint:

```bash
# Run type representation tests
dune exec test -- --only-test="Type Representation"

# Expected output: All tests pass
# Verify: Type construction, equality, environment operations work
# Manual check: Print some types with string_of_type to verify readable output
```

Milestone 2 Checkpoint:

```bash
# Run basic type checking tests
dune exec test -- --only-test="Basic Type Checking"

# Expected: Expression synthesis and checking tests pass
# Verify: Can type check simple expressions without inference
# Manual check: Type check a small program and inspect error messages
```

Milestone 3 Checkpoint:

```bash
# Run type inference tests
dune exec test -- --only-test="Type Inference"

# Expected: Constraint generation and unification tests pass
# Verify: Can infer types for expressions without annotations
# Manual check: Infer type of `fun x -> x` should be `'a -> 'a`
```

Milestone 4 Checkpoint:

```bash
# Run all tests including polymorphism
dune test

# Expected: All milestone tests plus integration tests pass
# Verify: Polymorphic let bindings work correctly
# Manual check: `let id = fun x -> x in (id 5, id true)` should type check
```

G. Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
Type equality always returns false	Missing case in type_equal pattern match	Print both types being compared	Add missing pattern match cases
Unification fails on identical types	Type variables have different internal representations	Print type variable names and internal IDs	Ensure consistent type variable creation
Environment lookup fails for existing variables	Scoping implementation incorrect	Print environment contents during lookup	Fix scope search order (inside-out)
Constraint generation misses constraints	Incomplete traversal of expression AST	Add debug prints in generate_constraints	Add missing cases for all expression forms
Infinite loop during unification	Missing occurs check or incorrect recursion	Add debug prints in unification loop	Implement proper occurs check before recursive unification
Polymorphism instantiation creates wrong types	Variable capture during instantiation	Print type schemes before/after instantiation	Use fresh variable generation during instantiation
Tests pass individually but fail together	Shared mutable state between tests	Run tests in different orders	Remove mutable global state or reset between tests

The systematic testing approach ensures that each milestone builds on a solid, well-tested foundation, making debugging and development much more manageable as complexity increases through the milestones.

Debugging Guide

> **Milestone(s):** Milestone 2 (Basic Type Checking), Milestone 3 (Type Inference), Milestone 4 (Polymorphism)

Think of debugging a type checker as **medical diagnosis for a complex organism** — much like how a doctor uses symptoms to identify underlying causes and prescribe targeted treatments, debugging a type checker requires systematic observation of symptoms (unexpected behavior), analysis of root causes (algorithmic failures), and application of specific fixes (code corrections). Just as medical diagnosis requires understanding how different organ systems interact, type checker debugging demands deep understanding of how type environments, constraint generation, unification, and error recovery components work together.

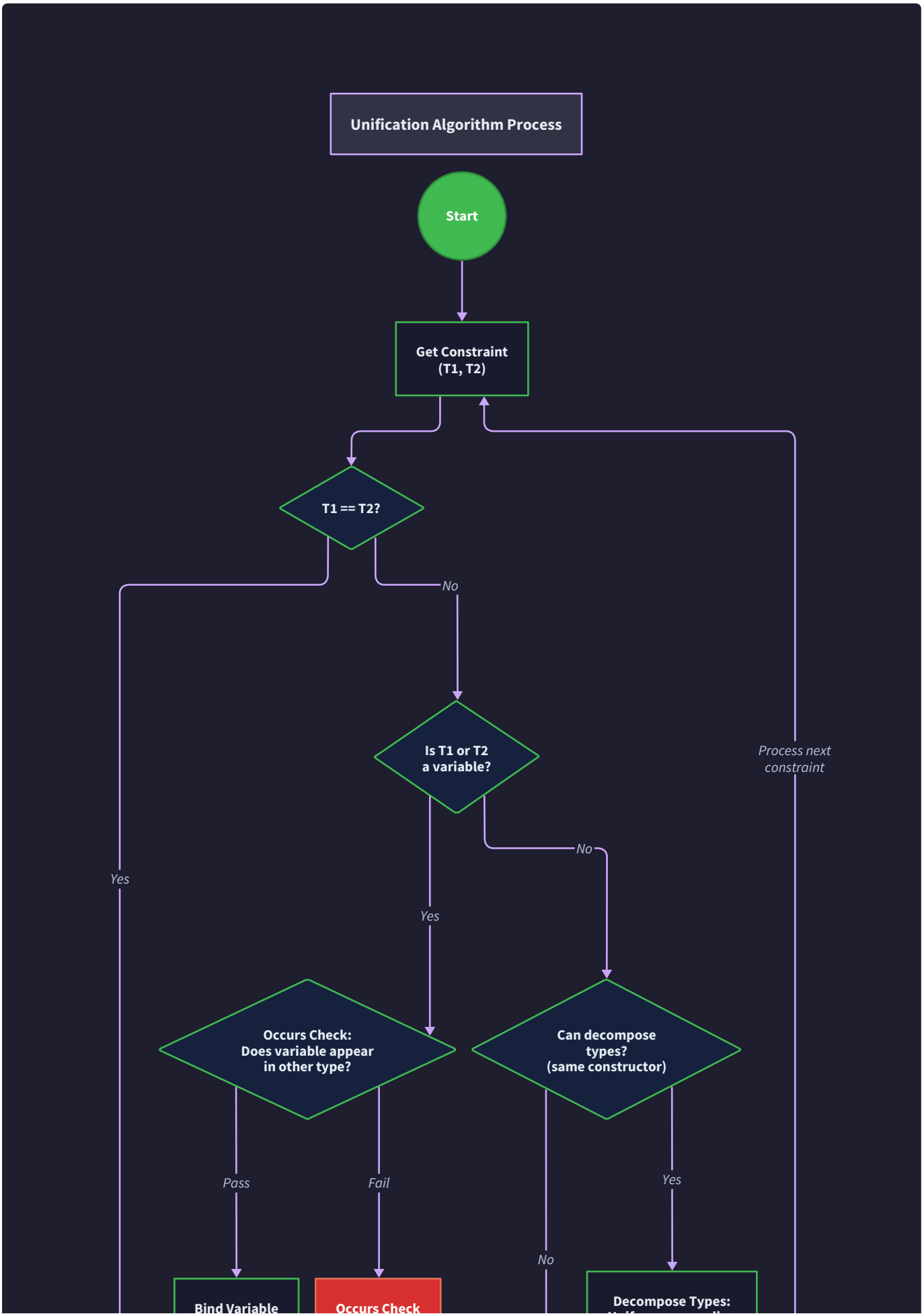
The fundamental challenge in type checker debugging lies in the **cascade effect** — a single algorithmic error can propagate through multiple phases, causing symptoms that appear far from the actual root cause. Unlike simpler programs where bugs manifest locally, type checker bugs often involve complex interactions between constraint generation, unification, environment threading, and substitution application. A bug in occurs check implementation might not surface until polymorphic instantiation fails mysteriously, or an environment scoping error might only become apparent when let-polymorphism produces incorrect generalizations.

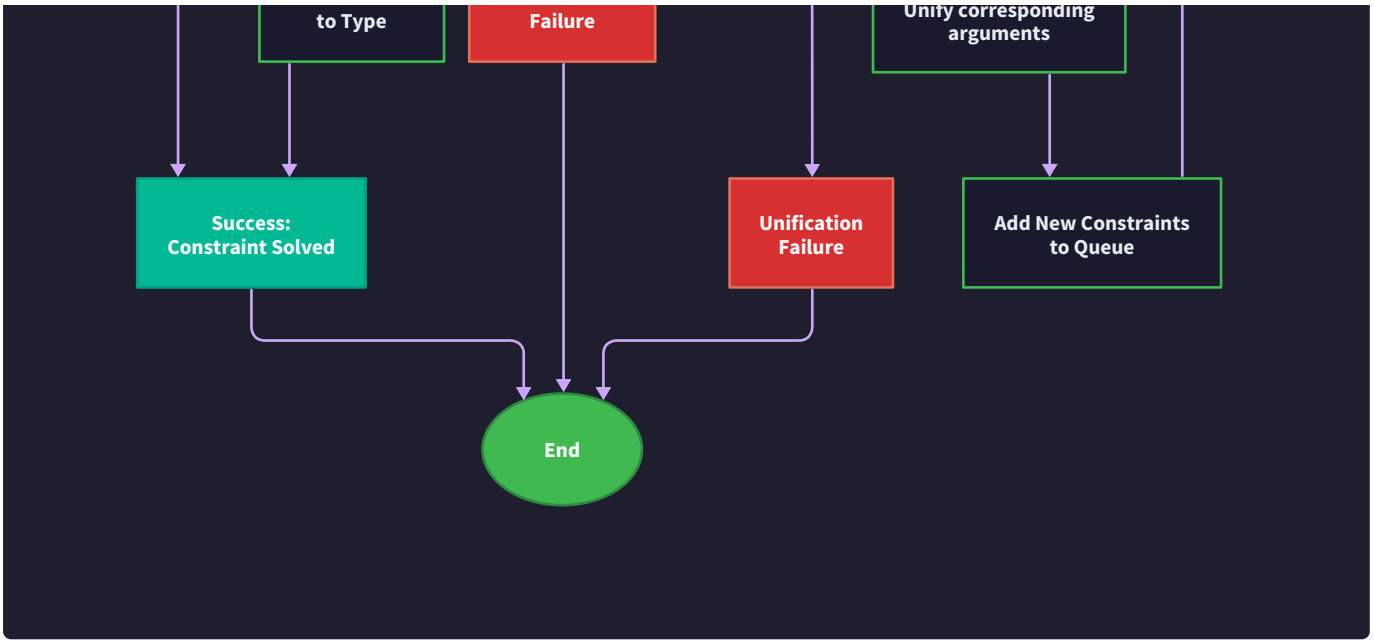
This section provides systematic debugging methodologies for the three most challenging areas: unification algorithm failures, type environment scoping problems, and type inference behavioral issues. Each area requires different diagnostic approaches because they involve different algorithmic phases and data structures.

Unification Algorithm Debugging

Think of debugging unification as **forensic analysis of a negotiation breakdown** — much like how diplomatic negotiations can fail at various stages (miscommunication, incompatible demands, procedural violations), unification can fail during constraint decomposition, occurs check violations, or substitution composition errors. The key to successful debugging lies in systematic tracing of the constraint solving process to identify exactly where and why the negotiation broke down.

The unification algorithm represents the heart of constraint-based type inference, and its failures can manifest in numerous ways. Understanding the anatomy of unification failures requires deep knowledge of how constraints are generated, how the Robinson unification algorithm processes them, and how substitutions compose and apply to type expressions.


Unification Algorithm Process
Start
Get Constraint (T1, T2)
T1 == T2?
Yes
No
Is T1 or T2 a variable?
Yes
No
Occurs Check: Does variable appear in other type?
Pass
Fail
Bind Variable
Occurs Check
Can decompose types? (same constructor)
Yes
No
Decompose Types:
Process next constraint



Constraint Generation Trace Analysis

The first step in unification debugging involves tracing constraint generation to ensure the initial constraint set accurately reflects the program's type requirements. Constraint generation failures often manifest as missing constraints (leading to overly permissive inference) or incorrect constraints (leading to unsatisfiable constraint sets).

Constraint Generation Debugging Table:

Symptom	Likely Root Cause	Diagnostic Steps	Corrective Action
Function calls accept wrong argument types	Missing argument-parameter constraints	Print generated constraints for function call expressions	Add constraint generation for each argument position
Binary operators succeed with incompatible types	Operator constraints not generated	Trace constraint generation through binary expression handling	Implement operator-specific type constraint generation
Variable references have wrong inferred types	Environment lookup produces wrong type during constraint generation	Print environment contents at variable reference sites	Fix environment threading or lookup implementation
Let-bound variables lose polymorphic types	Generalization constraints missing	Trace let-binding constraint generation and generalization	Add proper generalization constraint handling
Assignment statements allow type mismatches	Assignment compatibility constraints missing	Examine assignment statement constraint generation	Generate equality constraints between LHS and RHS types

The constraint generation phase must produce a complete and accurate representation of all type requirements in the program. Each expression construct should generate appropriate constraints based on its typing rules. For function calls, this means generating constraints between each argument and its corresponding parameter type. For binary operators, this means constraining both operands to be compatible with the operator's type signature.

> **Critical Insight**: Constraint generation bugs often manifest as "phantom" type errors — the unification algorithm correctly identifies that constraints cannot be satisfied, but the constraints themselves don't accurately represent the program's intended typing behavior. Always verify constraint correctness before debugging unification failures.

Robinson Unification Algorithm Tracing

The core unification algorithm processes constraints by decomposing complex type expressions into simpler unifiable components and building substitutions that make constraint pairs equal. Unification failures typically occur during occurs check violations, incompatible constructor mismatches, or substitution composition errors.

Unification Step-by-Step Debugging Process:

1. **Constraint Preprocessing**: Examine the input constraint set for obviously unsatisfiable constraints (like `TInt` unified with `TBool`). These indicate constraint generation problems rather than unification algorithm issues.
2. **Constraint Ordering Analysis**: Verify that constraint processing order doesn't affect the final result. The unification algorithm should be deterministic regardless of constraint ordering, but implementation bugs can introduce order dependencies.
3. **Variable Binding Tracing**: For each type variable unification, trace the binding process. Verify that the occurs check correctly prevents infinite type creation and that variable substitutions properly propagate through the constraint set.
4. **Constructor Decomposition Verification**: When unifying complex types like `TFun(TInt, TVar "a")` with `TFun(TVar "b", TBool)`, verify that the algorithm correctly decomposes into parameter and return type constraints.
5. **Substitution Composition Checking**: Trace how individual variable bindings compose into the final substitution. Verify that substitution application maintains type structure integrity.

Unification Failure Pattern Analysis:

Failure Pattern	Algorithmic Cause	Debugging Focus	Implementation Fix
"Cannot unify TInt with TBool" on seemingly compatible expressions	Constraint generation produced wrong constraint	Trace constraint generation for the failing expression	Fix constraint generation logic
"Occurs check failed" on recursive type definitions	Infinite type created during unification	Examine occurs check implementation and recursive type handling	Implement proper occurs check with cycle detection
"Substitution application failed" during constraint solving	Substitution composition or application error	Trace substitution building and application steps	Fix substitution data structure and application logic
"Cannot unify function types" with matching signatures	Function type decomposition error	Examine function type constraint decomposition	Fix function type unification to properly handle parameter and return types
"Type variable escapes its scope" during polymorphic unification	Variable capture during substitution	Trace variable scoping and substitution application	Implement proper variable scoping and fresh variable generation

The occurs check deserves special attention because its failures often indicate deeper issues with type variable scoping or constraint generation. When the occurs check reports `TVar "a"` occurs in `TFun(TVar "a", TInt)`, this might indicate a genuine recursive type attempt, or it might reveal that constraint generation incorrectly created self-referential constraints.

Substitution Application and Composition Debugging

Substitution application represents the final phase of constraint solving, where the computed variable bindings transform type expressions into their concrete forms. Substitution bugs often manifest as variables remaining unsubstituted, incorrect type structure after substitution, or substitution composition producing inconsistent results.

Substitution Debugging Methodology:

The substitution system maintains mappings from type variables to concrete types, and these mappings must be consistently applied throughout the type checker. Substitution composition combines multiple partial solutions into complete variable bindings, and this process can fail if the individual substitutions conflict or if the composition algorithm doesn't properly handle variable dependencies.

Substitution Application Trace Points:

1. **Variable Resolution Verification**: For each type variable in the original constraint set, verify that the final substitution contains an appropriate binding and that the binding doesn't introduce new unresolved variables.

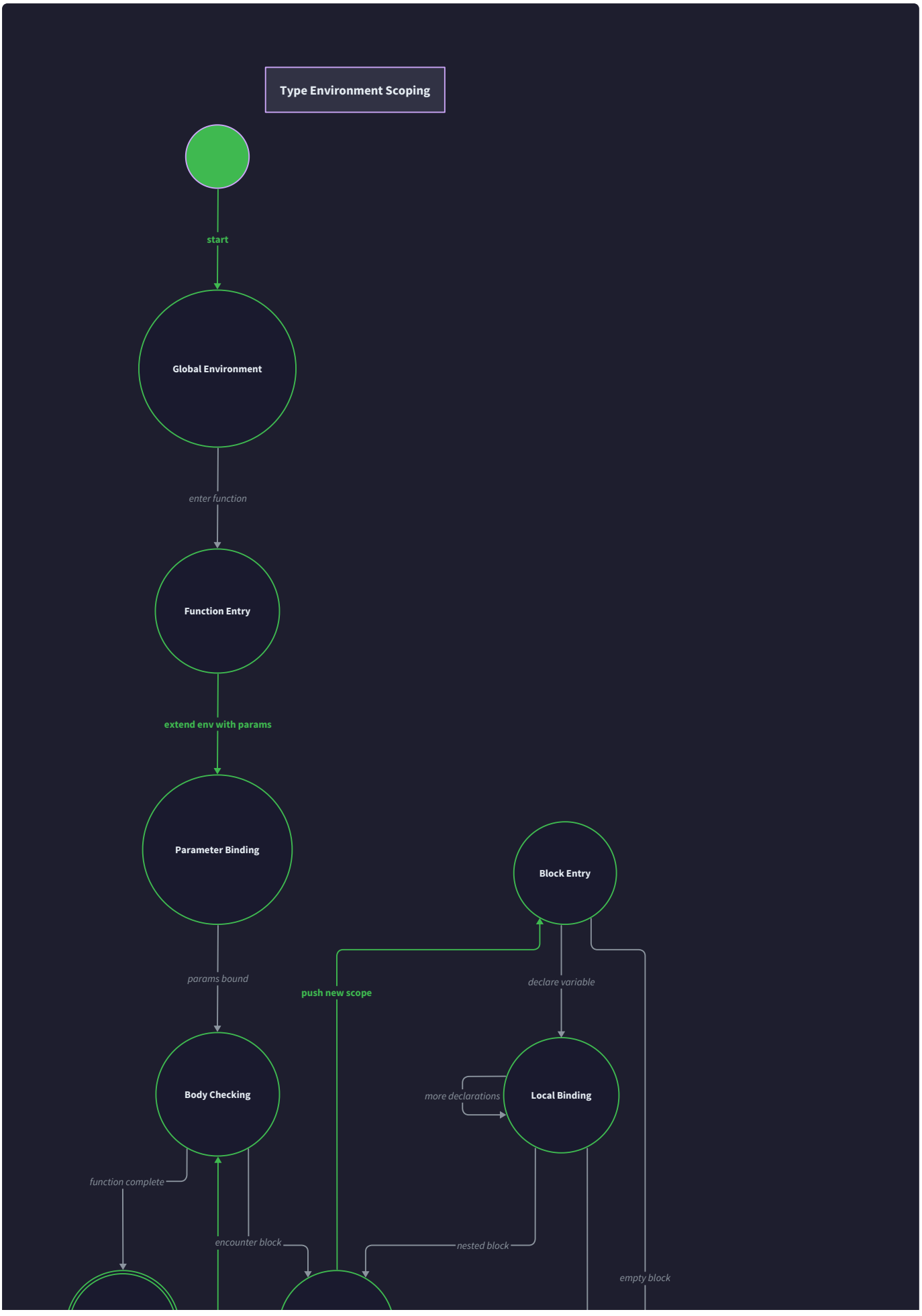
2. **Composition Consistency Checking**: When composing substitutions `s1` and `s2`, verify that variables bound in both substitutions receive consistent bindings and that the composition properly handles variable dependencies.
3. **Type Structure Preservation**: After substitution application, verify that the resulting type expressions maintain proper structure and don't contain malformed type constructions.
4. **Occurs Check Preservation**: Ensure that substitution application doesn't introduce occurs check violations that weren't present in the original constraint set.

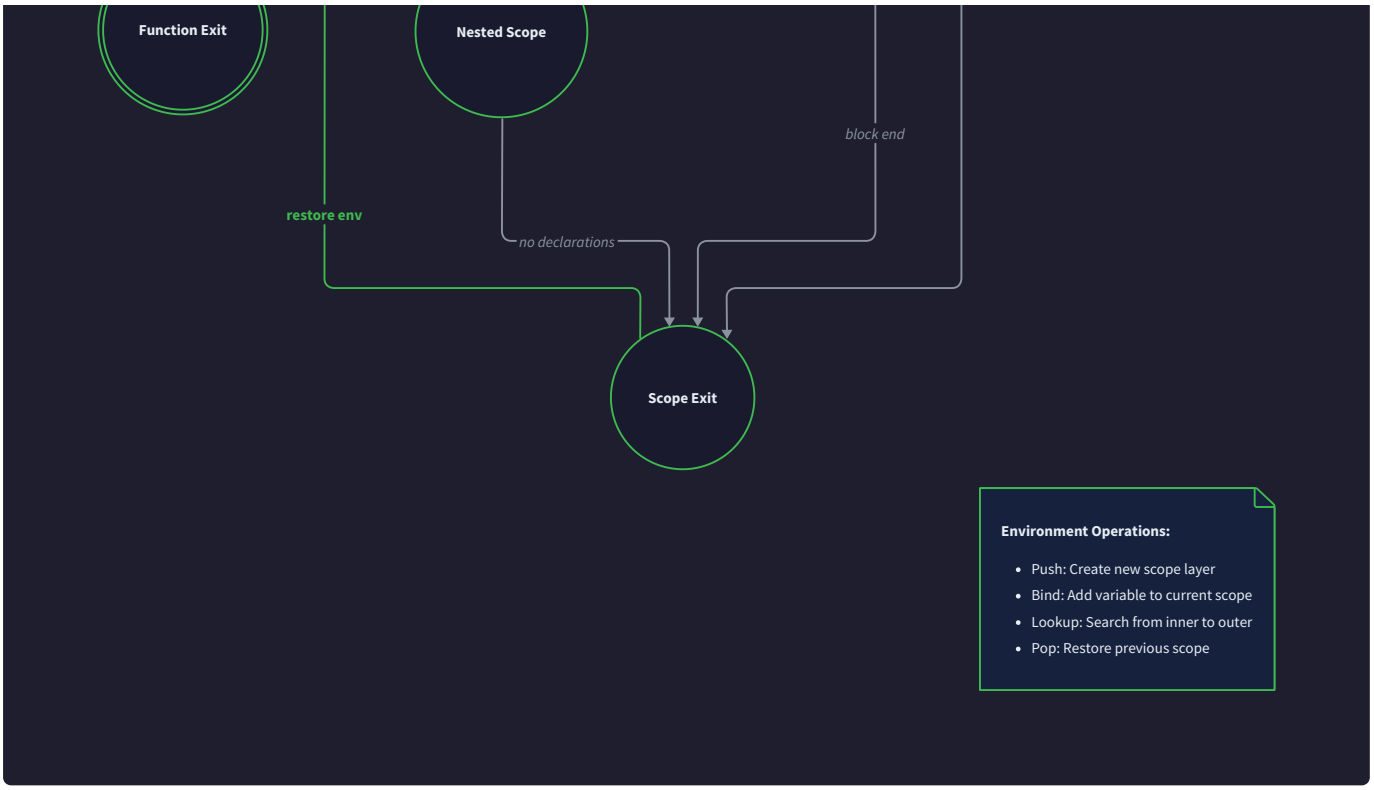
> **Decision: Substitution Representation Strategy**
>
> - **Context**: Substitutions can be represented as association lists, hash tables, or functional maps, each with different performance and correctness characteristics
> - **Options Considered**:
>   - Association lists with linear lookup
>   - Mutable hash tables with constant-time lookup
>   - Immutable functional maps with logarithmic lookup
> - **Decision**: Use immutable functional maps for substitution representation
> - **Rationale**: Immutable maps prevent substitution corruption bugs, support efficient composition operations, and provide automatic conflict detection during composition
> - **Consequences**: Slightly higher memory usage but significantly easier debugging and guaranteed correctness properties

Type Environment Issues

Think of debugging type environment problems as **investigating security clearance violations** — much like how security systems must properly track who has access to what resources at what times, type environments must accurately track which variables are in scope with what types throughout the program's execution. Environment bugs often manifest as access control failures: variables that should be accessible become unreachable, or variables that should be out of scope remain accessible with stale type information.

Type environment debugging presents unique challenges because environment state changes continuously during type checking, and environment threading errors can cause subtle bugs that only manifest in complex nested scoping scenarios. The type environment serves as the authoritative source for variable type information, and any corruption or incorrect threading can cascade through the entire type checking process.


Type Environment Scoping
start
Global Environment
enter function
Function Entry
extend env with params
Parameter Binding
params bound
Body Checking
function complete
encounter block
push new scope
Block Entry
declare variable
Local Binding
more declarations
nested block
empty block



Lexical Scoping Violation Diagnosis

Lexical scoping requires that variable visibility follows the program's textual structure — inner scopes can access outer scope variables, but outer scopes cannot access inner scope variables, and variable shadowing should hide outer definitions without destroying them. Scoping violations typically result from incorrect environment extension, improper scope restoration, or variable lookup implementation errors.

Scoping Violation Symptom Analysis:

Symptom	Environment Problem	Diagnostic Approach	Solution Strategy
Variables accessible outside their declared scope	Scope restoration failure	Trace environment state before and after scope exit	Implement proper scope stack management
Inner scope variables shadow outer variables permanently	Environment restoration doesn't restore shadowed bindings	Examine environment extension and restoration logic	Use environment layering instead of destructive updates
Variable lookups find stale type information	Environment threading carries outdated state	Trace environment threading through recursive calls	Fix environment parameter passing and result handling
Let-bound variables invisible in their own body	Environment extension happens after body checking	Trace environment state during let-binding processing	Extend environment before checking let body
Function parameters not visible in function body	Parameter binding environment not threaded to body checking	Examine function definition type checking	Add parameter bindings before checking function body

The environment threading problem represents one of the most subtle debugging challenges. Environment threading requires that each type checking function receives the current environment state and returns an updated environment reflecting any new bindings introduced during checking. Threading errors occur when the environment flow doesn't properly reflect the program's execution semantics.

Environment Threading Debug Trace Process:

1. **Entry Point Verification**: At each function entry point, verify that the received environment contains all expected bindings from the calling context. Missing bindings indicate threading errors in the caller.
2. **Modification Tracking**: Track all environment modifications during function execution. Verify that modifications correctly reflect the semantic effects of the constructs being checked.
3. **Exit Point Validation**: At function exit points, verify that the returned environment contains appropriate modifications and that temporary bindings (like function parameters) are properly handled.
4. **Recursive Call Environment Flow**: For recursive type checking calls, verify that environment updates from nested calls properly propagate back to the calling context.
5. **Scope Boundary Handling**: At scope boundaries (function definitions, let expressions, block statements), verify that environment modifications correctly implement scoping semantics.

Variable Lookup Implementation Debugging

Variable lookup implementation must traverse environment layers in the correct order, handle variable shadowing properly, and distinguish between different binding types (monomorphic vs. polymorphic). Lookup

failures can result from incorrect layer traversal, improper binding type handling, or environment corruption.

Variable Lookup Error Pattern Analysis:

The `lookup` function searches environment layers from innermost to outermost, returning the first matching variable binding. This implementation must handle several edge cases: variables that aren't bound in any scope, variables bound in multiple scopes (shadowing), and variables bound with different binding types (monomorphic vs. polymorphic).

Lookup Implementation Debugging Table:

Lookup Failure	Implementation Issue	Debug Investigation	Corrective Implementation
Variables not found despite being in scope	Layer traversal order incorrect	Print environment contents and traversal order	Implement inside-out layer traversal
Wrong variable bindings returned from lookup	Shadowing logic incorrect	Trace lookup through nested scopes with same variable names	Return first match during inside-out traversal
Polymorphic variables treated as monomorphic	Binding type handling wrong	Examine binding type discrimination in lookup	Properly handle `Monomorphic` vs `Polymorphic` binding constructors
Variables found in wrong scopes	Environment layer structure incorrect	Examine environment extension and scope management	Fix environment layering to properly represent scope nesting
Lookup performance degrades with nesting depth	Linear search through all layers	Profile lookup performance in deeply nested scopes	Optimize with scoped caching or hash-based lookup

The distinction between `Monomorphic` and `Polymorphic` bindings represents a critical aspect of lookup implementation. Polymorphic bindings require instantiation at each use site (replacing quantified variables with fresh variables), while monomorphic bindings can be used directly. Mixing up these binding types leads to incorrect generalization behavior.

> **Critical Implementation Detail**: The `lookup` function must distinguish between finding no binding (`None`) and finding a binding with an incorrect type. Type compatibility checking happens after successful lookup, not during the lookup process itself.

Environment Extension and Restoration Patterns

Environment extension adds new variable bindings to the innermost scope, while environment restoration removes temporary scopes when exiting scope boundaries. These operations must maintain environment invariants and properly handle the interaction between temporary bindings and permanent bindings.

Environment Lifecycle Management:

Environment extension using `extend_mono` and `extend_poly` adds bindings to the current innermost scope layer. The implementation must ensure that extension doesn't corrupt existing bindings and that new bindings properly shadow outer bindings with the same name. Environment restoration must remove temporary scope layers without affecting outer layers.

Scope Management Debugging Process:

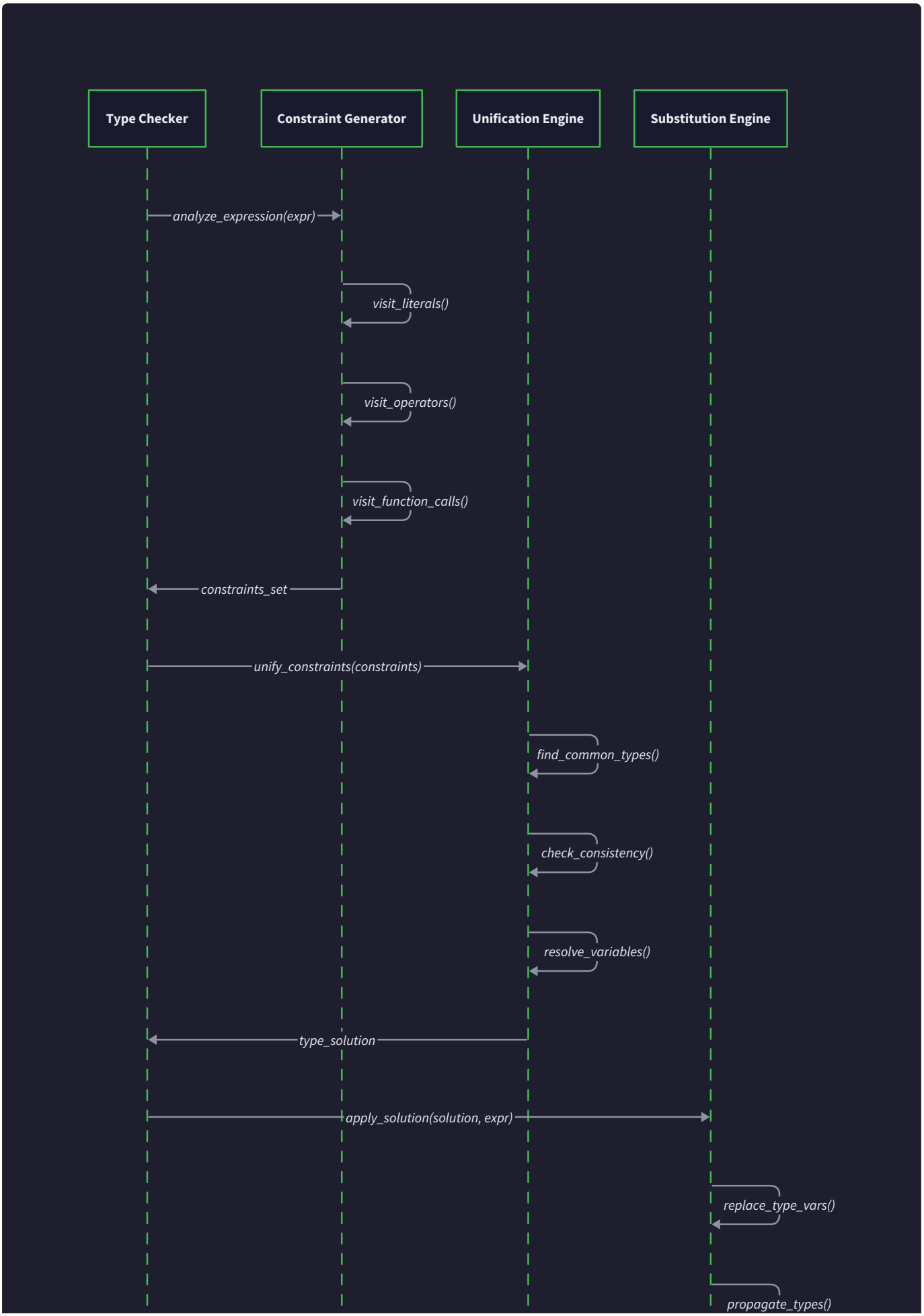
1. **Extension Verification**: After each environment extension, verify that the new binding appears in the innermost layer and that existing bindings remain intact. Extension should not modify any existing layers.
2. **Shadowing Behavior Checking**: When extending with a variable name that exists in outer scopes, verify that lookup returns the new binding and that the outer binding remains accessible after scope restoration.
3. **Restoration Integrity Testing**: After scope restoration, verify that temporary bindings are removed and that previously shadowed bindings become accessible again. Restoration should be the exact inverse of extension.
4. **Layer Structure Validation**: Throughout environment modifications, verify that the layer structure accurately reflects the program's scope nesting. Each layer should correspond to a specific syntactic scope boundary.
5. **Binding Type Preservation**: Verify that environment operations preserve binding types (`Monomorphic` vs `Polymorphic`) and that binding type changes only occur through explicit generalization or instantiation operations.

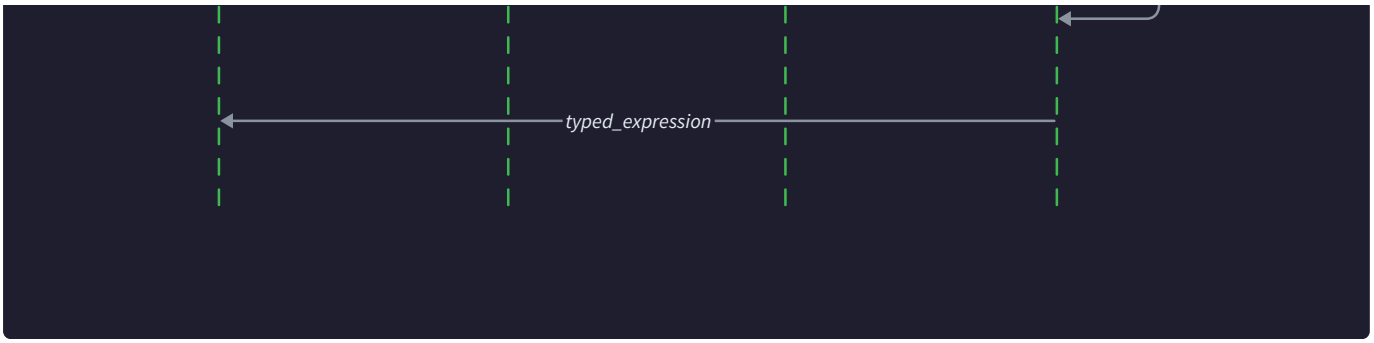
⚠️ **Pitfall: Destructive Environment Updates** Many type checker implementations incorrectly implement environment extension through destructive updates that modify existing environment layers. This breaks environment restoration and causes variables to remain accessible outside their proper scopes. Always implement environment extension through non-destructive layering that preserves the ability to restore previous environment states.

Type Inference Problems

Think of debugging type inference as **investigating detective reasoning failures** — much like how a detective's investigation can go wrong through missing evidence, incorrect deductions, or faulty reasoning processes, type inference can fail through inadequate constraint generation, flawed unification logic, or incorrect type generalization. The key to successful inference debugging lies in understanding which phase of the inference pipeline is producing unexpected results and why.

Type inference represents the most complex aspect of type checker implementation because it involves the coordinated interaction of multiple algorithmic components: constraint generation from program syntax, constraint solving through unification, and type generalization through let-polymorphism. Each component can fail independently, and failures often manifest as seemingly unrelated symptoms in the final inferred types.


Type Checker
Constraint Generator
Unification Engine
Substitution Engine
analyze_expression(expr)
visit_literals()
visit_operators()
visit_function_calls()
constraints_set
unify_constraints(constraints)
find_common_types()
check_consistency()
resolve_variables()
type_solution
apply_solution(solution, expr)
replace_type_vars()
propagate_types()



Inference Result Analysis and Validation

Type inference should produce the most general correct types for program expressions — types that are as permissive as possible while still ensuring type safety. Inference problems manifest as overly specific types (lost generality), overly general types (incorrect inference), or completely wrong types (algorithmic failures).

Inference Quality Assessment Framework:

The quality of inference results can be evaluated along several dimensions: correctness (do the inferred types prevent type errors), generality (are the types as polymorphic as possible), and consistency (do similar expressions receive similar inferred types). Each dimension requires different validation approaches.

Inference Result Validation Table:

Quality Dimension	Expected Behavior	Validation Method	Failure Diagnosis
Type Correctness	Inferred types prevent runtime type errors	Run programs with inferred types and verify no type violations	Examine constraint generation and solving for missed constraints
Maximum Generality	Types are as polymorphic as the program allows	Compare inferred types with hand-written most general types	Investigate generalization algorithm and value restriction
Inference Consistency	Similar expressions get similar types	Compare inferred types for structurally similar expressions	Examine constraint generation consistency and environment handling
Error Locality	Type errors reported at actual error sites	Verify error locations correspond to actual type violations	Investigate error propagation and constraint solving
Performance Scalability	Inference time grows reasonably with program size	Profile inference performance on programs of varying sizes	Optimize constraint generation and solving algorithms

The most general type property represents the gold standard for type inference quality. Given a program expression, the inference algorithm should produce the most general type that allows the expression to be

used in as many contexts as possible while maintaining type safety. Deviation from maximum generality often indicates problems with the generalization algorithm or constraint solving process.

Generality Verification Process:

1. **Manual Type Derivation**: For problematic expressions, manually derive the most general type using paper-and-pencil type inference rules. This provides the ground truth for comparison with algorithmic results.
2. **Algorithmic Trace Comparison**: Compare the algorithmic inference steps with the manual derivation to identify where the algorithm diverges from optimal behavior.
3. **Constraint Set Analysis**: Examine the generated constraint set to verify that it accurately captures all type requirements without introducing spurious constraints that limit generality.
4. **Generalization Point Investigation**: For let-bound expressions, verify that generalization occurs at appropriate points and quantifies over all genuinely free variables.

Constraint Generation Consistency Issues

Constraint generation must produce consistent constraint sets that accurately reflect the program's type requirements. Inconsistency problems arise when similar language constructs generate different constraint patterns or when constraint generation doesn't properly handle context-dependent typing rules.

Constraint Generation Consistency Debugging:

The constraint generation phase transforms program syntax into type equality requirements, and this transformation must be consistent across similar syntactic constructs. Inconsistency often results from incomplete case analysis in constraint generation functions or from failure to properly handle context-sensitive typing rules.

Common Constraint Generation Inconsistencies:

Inconsistency Pattern	Manifestation	Root Cause Analysis	Resolution Strategy
Function calls generate different constraints for same argument patterns	Same function called with same argument types produces different constraints	Examine function call constraint generation for context dependencies	Standardize constraint generation to depend only on call site information
Binary operators produce asymmetric constraints	`a + b` and `b + a` generate different constraint sets	Investigate binary operator constraint generation logic	Ensure operator constraints are symmetric when operators are commutative
Let-binding generalization varies with expression complexity	Simple expressions generalize differently than complex expressions	Examine generalization criteria and value restriction implementation	Apply consistent generalization rules regardless of expression complexity
Variable references produce inconsistent constraints in different contexts	Same variable reference generates different constraints depending on usage context	Investigate environment lookup and constraint generation interaction	Separate variable lookup from constraint generation context
Nested function definitions lose type information	Inner functions can't access outer function type information	Examine environment threading through nested definitions	Fix environment scoping for nested function definitions

Constraint generation debugging requires systematic comparison of generated constraints for similar expressions. The constraint generation algorithm should be deterministic and context-independent — the same expression syntax should always generate the same constraint pattern regardless of where it appears in the program.

> **Decision: Constraint Generation Architecture**
>
> - **Context**: Constraint generation can be implemented as a single large function with pattern matching or as a collection of specialized functions per expression type
> - **Options Considered**:
>   - Monolithic constraint generation with exhaustive pattern matching
>   - Modular constraint generation with per-construct specialized functions
>   - Visitor pattern-based constraint generation with traversal separation
> - **Decision**: Use modular constraint generation with specialized functions per expression type
> - **Rationale**: Modular approach provides better testability, easier debugging through isolated constraint generation for each construct, and clearer separation of concerns
> - **Consequences**: Slightly more complex dispatch logic but significantly easier maintenance and debugging

Polymorphism and Generalization Debugging

Polymorphic type inference through let-polymorphism represents the most sophisticated aspect of type inference, and generalization bugs can be particularly subtle. Generalization must occur at appropriate syntactic points, quantify over the correct set of type variables, and respect value restriction requirements.

Let-Polymorphism Implementation Debugging:

Let-polymorphism allows expressions bound by let-constructs to be used polymorphically throughout their scope. The implementation requires careful coordination between type inference (determining what type to generalize), generalization (deciding which variables to quantify), and instantiation (creating fresh type variables at use sites).

Generalization Process Debug Analysis:

1. **Generalization Point Identification**: Verify that generalization occurs exactly at let-binding sites and not at other expression positions. Premature generalization leads to incorrect polymorphism, while delayed generalization loses polymorphic opportunities.
2. **Free Variable Calculation**: For each generalization site, verify that the free variable calculation correctly identifies which type variables should be quantified. Variables free in the current environment should not be generalized.
3. **Value Restriction Compliance**: Verify that value restriction properly prevents generalization of expressions that might contain mutable references or other side-effecting operations.
4. **Instantiation Consistency**: At polymorphic variable use sites, verify that instantiation creates fresh type variables and properly substitutes them for quantified variables.
5. **Type Scheme Representation**: Verify that type schemes correctly represent polymorphic types and that type scheme operations (instantiation, substitution application) preserve polymorphic type structure.

Polymorphism Debugging Error Patterns:

Error Pattern	Symptom	Algorithmic Cause	Debugging Focus	Fix Strategy
Variables lose polymorphic types	Functions that should be polymorphic become monomorphic	Generalization not occurring or quantifying wrong variables	Examine generalization trigger conditions and free variable calculation	Fix generalization algorithm and variable scoping
Invalid polymorphic generalization	Variables generalized that should remain monomorphic	Value restriction not applied or environment analysis incorrect	Investigate value restriction implementation and environment free variable analysis	Implement proper value restriction and environment analysis
Instantiation creates wrong types	Polymorphic functions called with wrong inferred argument types	Instantiation algorithm creates incorrect fresh variables	Trace instantiation process and variable substitution	Fix instantiation to properly handle quantified variables
Type scheme operations corrupt polymorphic types	Polymorphic types become malformed during constraint solving	Type scheme substitution or composition errors	Examine type scheme representation and operations	Implement correct type scheme operations with proper variable handling
Polymorphic recursion fails	Recursive functions can't call themselves at different types	Recursive generalization or environment handling incorrect	Investigate recursive function type checking and environment threading	Handle recursive function generalization properly

⚠️ **Pitfall: Premature Generalization** A common error involves generalizing types too early in the inference process, before all constraints have been solved. This leads to polymorphic types that don't reflect the actual constraint requirements. Always perform generalization after complete constraint solving, and only at syntactic let-binding sites.

The value restriction represents a critical correctness requirement for generalization. Expressions that might contain mutable state or perform side effects cannot be safely generalized because polymorphic instantiation might violate type safety. The value restriction prevents generalization of non-syntactic values, ensuring that only genuinely pure expressions receive polymorphic types.

Implementation Guidance

Technology Recommendations:

Component	Simple Approach	Advanced Approach
Debug Tracing	Printf-style debugging with manual trace points	Structured logging with trace levels and context
Constraint Visualization	Text dumps of constraint sets	Graphical constraint dependency visualization
Environment Inspection	Simple pretty-printing of environment contents	Interactive environment explorer with scope highlighting
Unification Tracing	Step-by-step text output of unification steps	Visual unification tree with substitution application
Error Reporting	Basic error messages with source locations	Rich error messages with context and suggestions

Debug Infrastructure File Structure:

```
project-root/
  debug/
    trace.ml              ← tracing infrastructure
    pretty_print.ml       ← type and constraint pretty printing
    env_inspector.ml      ← environment debugging utilities
    unification_trace.ml  ← unification algorithm tracing
  tests/
    debug_tests/
      constraint_gen_tests.ml  ← constraint generation debugging tests
      unification_tests.ml     ← unification algorithm tests
      environment_tests.ml     ← environment scoping tests
```

Debug Tracing Infrastructure:

OCAML

```
(* Debug tracing levels for selective debugging output *)
type debug_level =
  | Silent
  | Errors
  | Warnings
  | Info
  | Debug
  | Trace

(* Debug context for tracking inference state *)
type debug_context = {
  current_expression: string;
  environment_size: int;
  constraint_count: int;
  active_variables: string list;
}

(* Main debug tracing function *)
let debug_trace level context message =
  (* TODO 1: Check if current debug level enables this trace level *)
  (* TODO 2: Format message with context information *)
  (* TODO 3: Print formatted message with timestamp and location *)
  (* TODO 4: Flush output to ensure immediate visibility *)
  failwith "implement debug_trace"

(* Constraint generation tracing *)
let trace_constraint_generation expr constraints =
  (* TODO 1: Extract expression structure for readable display *)
```

```
  (* TODO 2: Format constraint list with type pretty printing *)

  (* TODO 3: Output constraint generation summary *)

  (* TODO 4: Highlight any unusual or complex constraints *)

  failwith "implement trace_constraint_generation"
```

Environment Debugging Utilities:

```ocaml
(* Environment state inspection for debugging *)

let inspect_environment env =

  (* TODO 1: Traverse environment layers from innermost to outermost *)

  (* TODO 2: Format each binding with its scope level and type information *)

  (* TODO 3: Highlight any shadowed variables or unusual bindings *)

  (* TODO 4: Generate environment structure summary *)

  failwith "implement inspect_environment"


(* Variable lookup tracing with step-by-step analysis *)

let trace_variable_lookup name env =

  (* TODO 1: Start lookup from innermost scope layer *)

  (* TODO 2: Print each layer search with found/not found status *)

  (* TODO 3: Show final lookup result and binding type *)

  (* TODO 4: Highlight any scoping anomalies or unexpected results *)

  failwith "implement trace_variable_lookup"


(* Environment modification debugging *)

let trace_environment_modification old_env new_env operation =

  (* TODO 1: Compare old and new environment structures *)

  (* TODO 2: Identify what bindings were added, removed, or modified *)

  (* TODO 3: Verify that modification matches expected operation semantics *)

  (* TODO 4: Report any unexpected environment changes *)

  failwith "implement trace_environment_modification"
```

Unification Algorithm Debugging:

```ocaml
(* Unification step tracing for algorithm debugging *)

let trace_unification_step constraint_pair current_subst =

  (* TODO 1: Pretty print the constraint being processed *)

  (* TODO 2: Show current substitution state before processing *)

  (* TODO 3: Apply unification step and show resulting substitution *)

  (* TODO 4: Highlight any occurs check violations or decomposition steps *)

  failwith "implement trace_unification_step"


(* Substitution application tracing *)

let trace_substitution_application subst target_type =

  (* TODO 1: Show substitution mapping and target type structure *)

  (* TODO 2: Apply substitution step-by-step with intermediate results *)

  (* TODO 3: Verify final result maintains type structure integrity *)

  (* TODO 4: Report any substitution application failures or anomalies *)

  failwith "implement trace_substitution_application"


(* Occurs check debugging with cycle detection *)

let debug_occurs_check var_name target_type =

  (* TODO 1: Traverse target type structure looking for variable occurrences *)

  (* TODO 2: Build occurrence path showing where variable appears *)

  (* TODO 3: Determine if occurrence creates infinite type *)

  (* TODO 4: Report occurs check result with explanation *)

  failwith "implement debug_occurs_check"
```

Inference Quality Validation:

```ocaml
(* Type inference result validation against expected behavior *)

let validate_inference_result expr expected_type inferred_type =

  (* TODO 1: Check that inferred type is at least as general as expected *)

  (* TODO 2: Verify that inferred type satisfies all program constraints *)

  (* TODO 3: Compare generalization level with expected polymorphism *)

  (* TODO 4: Report inference quality assessment and any deviations *)

  failwith "implement validate_inference_result"


(* Generalization debugging for let-polymorphism *)

let debug_generalization expr inferred_type env =

  (* TODO 1: Calculate free variables in inferred type and environment *)

  (* TODO 2: Check value restriction applicability *)

  (* TODO 3: Show generalization decision and quantified variables *)

  (* TODO 4: Verify generalization correctness against typing rules *)

  failwith "implement debug_generalization"
```

Milestone Debugging Checkpoints:

Milestone 2 Checkpoint - Basic Type Checking:

- Run: `ocaml debug_basic_checking.ml`
- Expected: Environment lookup tracing shows correct variable scoping
- Verify: Type error messages include accurate source locations
- Warning Signs: Variables accessible outside declared scopes, incorrect error locations

Milestone 3 Checkpoint - Type Inference:

- Run: `ocaml debug_inference.ml`
- Expected: Constraint generation produces consistent constraint sets
- Verify: Unification algorithm converges to most general types
- Warning Signs: Overly specific inferred types, constraint generation inconsistencies

Milestone 4 Checkpoint - Polymorphism:

- Run: `ocaml debug_polymorphism.ml`
- Expected: Let-bound expressions receive appropriate polymorphic types

- Verify: Instantiation creates fresh variables at use sites
- Warning Signs: Loss of polymorphism, incorrect generalization

Future Extensions

> **Milestone(s):** Beyond Milestone 4 (Polymorphism) - Advanced features that build upon the complete type checker foundation

Think of future extensions as **architectural expansion plans for a well-built house** — much like how a solidly constructed home with good foundations, electrical, and plumbing can later be expanded with new rooms, smart home systems, or energy-efficient upgrades. Our type checker, once built with the four core milestones, provides a robust foundation that can support sophisticated type system features without requiring fundamental rewrites. Each extension leverages the existing infrastructure while adding new capabilities that enhance the language's expressiveness and safety guarantees.

The beauty of our constraint-based architecture is that it naturally accommodates extensions through additional constraint generation rules, new type constructors in the `ty` algebraic data type, and enhanced unification procedures. The modular design separating type representation, environment management, inference engine, and error reporting means that most extensions can be added incrementally without disrupting the core functionality.

This section explores the most impactful extensions that would transform our basic functional language type checker into a production-ready system comparable to languages like Haskell, Rust, or modern TypeScript. Each extension is presented with its conceptual foundation, integration approach, implementation complexity, and the new capabilities it unlocks for language users.

Algebraic Data Types and Pattern Matching

Algebraic data types (ADTs) represent the next logical evolution beyond basic types — think of them as **structured data blueprints with multiple construction patterns**, much like how a vehicle can be constructed as either a car (with doors, engine, seats) or a motorcycle (with handlebars, engine, wheels) or a truck (with cargo bed, engine, cab). Each construction pattern, called a **constructor**, defines what data must be provided and how it's arranged.

The fundamental insight is that ADTs combine **product types** (structs/records that hold multiple pieces of data simultaneously) and **sum types** (unions that represent one of several possible alternatives). This creates a powerful type system feature that enables precise modeling of domain concepts while maintaining compile-time safety guarantees.

Consider how we might represent a simple expression language within our typed language:

ADT Constructor	Parameters	Represents	Example Usage
`Const`	`int`	Integer literal	`Const 42`
`Add`	`expr * expr`	Addition operation	`Add(Const 1, Const 2)`
`Mul`	`expr * expr`	Multiplication operation	`Mul(Const 3, Add(Const 1, Const 2))`
`Var`	`string`	Variable reference	`Var "x"`

The type system extension requires several new components in our `ty` representation:

New Type Constructor	OCaml Definition	Purpose
`TConstructor`	`TConstructor of string * ty list`	Represents a data constructor with parameter types
`TSum`	`TSum of (string * ty list) list`	Represents a sum type with all possible constructors
`TRecord`	`TRecord of (string * ty) list`	Represents a product type with named fields

The constraint generation process must handle constructor applications by verifying that the provided arguments match the constructor's parameter types and then unifying the result with the expected ADT type. Pattern matching introduces **exhaustiveness checking** — the compiler must verify that all possible constructors are handled in match expressions.

> **Decision: Constructor Type Checking Strategy**
>
> - **Context**: Constructor applications need type checking, and pattern matches need exhaustiveness verification
> - **Options Considered**:
>   1. Treat constructors as regular functions with special typing rules
>   2. Add dedicated constraint types for constructor applications and pattern matches
>   3. Extend the unification algorithm with constructor-aware rules
> - **Decision**: Treat constructors as special functions with ADT-specific constraint generation
> - **Rationale**: Leverages existing function application infrastructure while enabling precise type checking for constructor patterns
> - **Consequences**: Enables clean integration with existing inference engine but requires constructor environment management

Pattern matching introduces **coverage analysis** requirements:

Analysis Type	Purpose	Implementation
Exhaustiveness	Ensure all constructors are matched	Generate constraint that match arms cover all constructors in sum type
Redundancy	Detect unreachable match arms	Track which constructor patterns have been seen and warn on duplicates
Variable Binding	Type variables bound in patterns	Extend environment with pattern variables during constraint generation

The error reporting system needs enhancement to provide constructor-specific messages:

Error Type	Detection	Message Format
Constructor Arity	Wrong number of arguments to constructor	"Constructor `Add` expects 2 arguments but got 1"
Non-Exhaustive Match	Missing constructor cases	"Pattern match not exhaustive, missing cases: `Mul`, `Var`"
Constructor Not Found	Unknown constructor name	"Constructor `Sub` not defined for type `expr`"

Type Classes and Interfaces

Type classes provide **structured polymorphism with constraints** — think of them as **professional certification requirements** where certain functions can only be called on types that have been "certified" to

support specific operations. Unlike simple polymorphism where `'a -> 'a` works with any type, type classes let us write `'a -> 'a` where `'a` must support equality comparison, arithmetic operations, or string conversion.

This extension transforms our type system from supporting only **parametric polymorphism** (generic over any type) to supporting **ad-hoc polymorphism** (generic over types that satisfy constraints). The classic example is a comparison function that works with any type supporting equality:

```
equal : forall 'a. Eq 'a => 'a -> 'a -> bool
```

The type signature reads: "for any type `'a` that implements the `Eq` type class, `equal` takes two values of type `'a` and returns a boolean."

The implementation requires extending our type representation with **qualified types**:

New Type Component	Purpose	Structure
`constraint_`	Represents type class membership	`constraint_` = `string * ty` (class name, type)
`qualified_type`	Type with constraint requirements	`qualified_type` = `constraint_ list * ty`
`TQualified`	Qualified type constructor	`TQualified of qualified_type`

Type class definitions specify the **interface contract**:

Type Class Component	Purpose	Example
Class Name	Unique identifier	`Eq`, `Ord`, `Show`
Type Parameter	Type being constrained	`'a` in `class Eq 'a`
Method Signatures	Required operations	`equal : 'a -> 'a -> bool`
Superclass Constraints	Dependencies on other classes	`Ord` requires `Eq`

Instance declarations provide implementations for specific types:

Instance Component	Purpose	Example
Target Type	Type receiving implementation	`int` for `instance Eq int`
Method Implementations	Concrete function definitions	`let equal x y = x = y`
Context Requirements	Constraints on type parameters	`Eq 'a` required for `instance Eq ('a list)`

The constraint solving process becomes significantly more complex, requiring **instance resolution**:

1. **Constraint Collection**: Gather all type class constraints from function calls
2. **Instance Matching**: Find applicable instances for each constraint
3. **Context Reduction**: Simplify constraints using instance contexts
4. **Coherence Checking**: Ensure unique instance resolution
5. **Dictionary Construction**: Generate evidence for constraint satisfaction

> **Decision: Instance Resolution Strategy**
>
> - **Context**: Type class constraints need resolution to concrete instances during compilation
> - **Options Considered**:
>   1. Global instance table with simple lookup
>   2. Constraint solving integrated with unification
>   3. Separate constraint resolution pass after type inference
> - **Decision**: Separate constraint resolution pass with instance environment
> - **Rationale**: Separates type inference concerns from instance selection, enabling better error messages and debugging
> - **Consequences**: Requires two-phase compilation but provides cleaner architecture and better extensibility

The error reporting system must handle type class-specific failures:

Error Category	Trigger	Message Format
Missing Instance	No instance found for constraint	"No instance for `Show (int -> int)` arising from expression..."
Ambiguous Instance	Multiple matching instances	"Ambiguous instance for `Num 'a`; candidates: `Int`, `Float`"
Context Too Weak	Function requires stronger constraints	"Could not deduce `Ord 'a` from context `Eq 'a`"

Row Types and Extensible Records

Row types enable **flexible record structures with compile-time safety** — think of them as **adaptive forms where you can add or remove fields** while ensuring that any function requiring specific fields will always find them present. This feature addresses the rigidity of traditional record types where every record must have exactly the fields defined in its type.

The key insight is representing record types as **rows of field-type pairs** with a **row variable** that can be instantiated with additional fields. A function that only needs a `name` field can work with any record containing at least that field, regardless of what other fields are present.

Row types use the notation `{name: string | r}` where `r` is a row variable representing "whatever other fields might be present." This enables **structural subtyping** for records — a record with more fields can be used where one with fewer fields is expected.

The type representation requires new constructors:

Row Type Component	OCaml Definition	Purpose
`TRow`	`TRow of (string * ty) list * ty option`	Row with fields and optional tail variable
`TRecord`	`TRecord of ty`	Record type containing a row
`field_constraint`	`field_constraint = ty * string * ty * ty`	Constraint that row contains field

Row unification becomes significantly more complex than simple type unification:

Unification Case	Challenge	Solution
Field Presence	Ensure required fields exist in row	Generate field constraints during unification
Row Extension	Add new fields to existing row	Instantiate row variables with extended rows
Row Combination	Merge multiple row constraints	Collect field requirements and check for conflicts

The constraint generation process must handle **field access operations**:

1. **Field Selection**: `record.field` generates constraint that record's row contains field
2. **Record Construction**: `{field1 = v1, field2 = v2}` creates row with specified fields
3. **Record Extension**: `{record with field = value}` adds field to existing record's row
4. **Record Restriction**: Pattern matching can remove fields from rows

> **Decision: Row Variable Scoping**
>
> - **Context**: Row variables need scoping rules to prevent capture and ensure proper generalization
> - **Options Considered**:
>   1. Treat row variables identically to type variables
>   2. Use separate scoping rules for row variables
>   3. Embed row variables within type variables
> - **Decision**: Separate row variable namespace with similar scoping to type variables
> - **Rationale**: Prevents confusion between type polymorphism and row polymorphism while enabling precise error messages
> - **Consequences**: Requires separate generalization and instantiation logic for row variables

Field constraint solving requires specialized algorithms:

Algorithm	Purpose	Complexity
Field Decomposition	Break record constraints into field constraints	O(n) fields
Row Unification	Unify two row types with different field sets	O(n log n) field comparison
Conflict Detection	Find inconsistent field type assignments	O(n) field scan

Effect Systems and Resource Tracking

Effect systems provide **compile-time tracking of computational side effects** — think of them as **environmental impact labels** for functions that precisely describe what external resources they access, modify, or depend upon. Just as organic food labels help consumers understand production methods, effect types help developers understand what a function might do beyond computing its return value.

Traditional type systems only track what data flows into and out of functions, but effect systems additionally track **computational effects** like file I/O, network access, memory allocation, exception throwing, or state modification. This enables compiler optimizations, better error handling, and architectural constraints (preventing I/O operations in pure computation contexts).

The fundamental concept extends function types from `input -> output` to `input -> output & effects` where effects describe the computational context required:

Effect Category	Examples	Type Annotation
Pure	Mathematical computation	`int -> int & Pure`
IO	File/network operations	`string -> string & IO`
State	Mutable variable access	`unit -> int & State[Counter]`
Exception	Error conditions	`string -> int & Throws[ParseError]`

Effect types require extending our type representation:

Effect Component	OCaml Definition	Purpose
`effect_type`	Algebraic type for effect kinds	Union of `Pure`, `IO`, `State`, `Exception` variants
`effect_set`	`effect_type list`	Set of effects a computation might perform
`TEffect`	`TEffect of ty * effect_set`	Function type with effect annotation

The constraint generation process must **track effect accumulation** through function composition:

1. **Effect Inheritance**: Function calls inherit callee's effects
2. **Effect Combination**: Sequence operations combine their effect sets
3. **Effect Masking**: Exception handling can mask throwing effects
4. **Effect Polymorphism**: Generic functions can be polymorphic over effects

> **Decision: Effect System Granularity**
>
> - **Context**: Effect tracking can be fine-grained (specific resources) or coarse-grained (effect categories)
> - **Options Considered**:
>   1. Track specific resources (file handles, network connections)
>   2. Track effect categories (IO, State, Exception)
>   3. Track both with hierarchical effect types
> - **Decision**: Start with effect categories, design for extensibility to resource-specific tracking
> - **Rationale**: Provides immediate value while avoiding complexity of resource lifetime tracking
> - **Consequences**: Enables effect-based reasoning and optimization while preserving implementation simplicity

Effect subtyping enables **effect masking** and **capability restriction**:

Subtyping Rule	Purpose	Example
Effect Subsumption	Pure code works in effectful contexts	`Pure` function usable where `IO` expected
Effect Strengthening	Add effects to existing computation	`IO` computation can be viewed as `IO & Exception`
Effect Intersection	Find common effect requirements	Function requiring both `IO` and `State` effects

Higher-Kinded Types and Type Constructors

Higher-kinded types enable **abstraction over type constructors** — think of them as **generic blueprints for container types** where you can write code that works uniformly over lists, options, trees, or any other "wrapper" type without knowing the specific container structure. This is like designing a universal packaging machine that can wrap items in boxes, bags, or tubes by accepting instructions for the specific wrapping process.

The key insight is that types have **kinds** (types of types) just as values have types. Regular types like `int` and `string` have kind `*` (pronounced "type"), while type constructors like `list` and `option` have kind `* -> *` (they take a type and produce a type). Higher-kinded polymorphism lets us write functions generic over any type constructor of a particular kind.

This enables powerful abstractions like **functors** (types supporting map operations) and **monads** (types supporting sequential composition with context):

Abstraction	Kind Signature	Common Examples
Functor	`('a -> 'b) -> 'f 'a -> 'f 'b`	`list`, `option`, `result`
Monad	`'a -> 'f 'a` and `'f 'a -> ('a -> 'f 'b) -> 'f 'b`	`option`, `list`, `io`
Traversable	`('a -> 'f 'b) -> 't 'a -> 'f ('t 'b)`	`list`, `tree`, `array`

The type representation requires **kind tracking**:

Kind Component	OCaml Definition	Purpose
`kind`	Algebraic type for kinds	`Star
`TKinded`	`TKinded of ty * kind`	Type with explicit kind annotation
`kind_env`	`(string * kind) list`	Kind assignments for type constructors

Kind inference proceeds alongside type inference:

1. **Kind Assignment**: Assign kinds to type constructors based on their definitions

2. **Kind Unification**: Ensure consistent kind usage in type applications
3. **Kind Generalization**: Generalize over kind variables in polymorphic types
4. **Kind Checking**: Verify that type constructor applications respect kind constraints

> **Decision: Kind Inference vs Kind Annotation**
>
> - **Context**: Higher-kinded polymorphism requires kind information for type constructor parameters
> - **Options Considered**:
>   1. Full kind inference similar to type inference
>   2. Require explicit kind annotations on type parameters
>   3. Hybrid approach with inference and optional annotations
> - **Decision**: Kind inference with optional annotations for disambiguation
> - **Rationale**: Reduces annotation burden while providing escape hatch for complex cases
> - **Consequences**: Requires sophisticated kind inference algorithm but improves user experience

Higher-kinded polymorphism enables **type constructor classes**:

Constructor Class	Purpose	Method Signatures
Functor	Uniform mapping over containers	`fmap : ('a -> 'b) -> 'f 'a -> 'f 'b`
Applicative	Multi-argument function lifting	`apply : 'f ('a -> 'b) -> 'f 'a -> 'f 'b`
Monad	Sequential composition with context	`bind : 'f 'a -> ('a -> 'f 'b) -> 'f 'b`

Dependent Types and Refinement Types

Dependent types allow **types to depend on runtime values** — think of them as **smart contracts for data** where the type system can express and verify precise properties like "this list has exactly 10 elements" or "this integer is between 1 and 100." This transforms the type system from checking basic structural compatibility to proving mathematical properties about program behavior.

The fundamental shift is from **simple types** (which classify data by structure) to **precise types** (which classify data by properties). A dependent function type `(n: int) -> vector(n)` means "given an integer n, return a vector of exactly n elements." The type system can verify that vector operations respect size requirements at compile time.

Refinement types provide a gentler introduction to dependent typing by **constraining existing types with predicates**:

Refinement Pattern	Type Syntax	Meaning
Range Constraint	`{x: int	0 <= x <= 100}`
Non-Null Constraint	`{x: string	length(x) > 0}`
Relationship Constraint	`{x: int	x > n}`

The type representation requires **predicate embedding**:

Dependent Component	OCaml Definition	Purpose
`predicate`	Expression tree for type constraints	Logical formulas over values
`TRefined`	`TRefined of ty * predicate`	Base type with constraint predicate
`TDependent`	`TDependent of string * ty * ty`	Dependent function type

Constraint generation must **extract verification conditions**:

1. **Precondition Generation**: Function calls must satisfy parameter constraints
2. **Postcondition Verification**: Function results must satisfy return type constraints
3. **Invariant Maintenance**: Operations must preserve type invariants
4. **Termination Checking**: Recursive functions must have decreasing measures

> **Decision: SMT Solver Integration**
>
> - **Context**: Dependent type checking requires automated theorem proving for constraint verification
> - **Options Considered**:
>   1. Simple pattern matching on constraint predicates
>   2. Integration with SMT solver (Z3, CVC4) for automated proving
>   3. Interactive theorem proving with user-provided proofs
> - **Decision**: SMT solver integration with fallback to user annotations for unprovable constraints
> - **Rationale**: Provides automation for common constraints while preserving expressiveness for complex properties
> - **Consequences**: Requires external solver dependency but enables practical dependent typing

The verification pipeline becomes:

Phase	Purpose	Output
Constraint Extraction	Convert type constraints to logical formulas	SMT-LIB format constraints
Solver Query	Check constraint satisfiability	Satisfiable/Unsatisfiable/Unknown
Counterexample Analysis	Extract failure information from solver	Specific values violating constraints
Error Reporting	Convert solver results to user messages	Type error with constraint violation details

Module Systems and Namespace Management

Module systems provide **large-scale code organization with controlled visibility** — think of them as **architectural blueprints for software libraries** where you can define public interfaces, hide implementation details, and compose functionality from multiple sources while preventing name conflicts and dependency cycles.

Our basic type checker assumes a single global namespace, but production languages need **hierarchical namespaces** with **explicit import/export control**. Module systems address several scalability challenges simultaneously: name collision avoidance, interface abstraction, separate compilation, and dependency management.

The core abstraction is the **module signature** — an interface specification that declares what types, values, and sub-modules are available from a module without exposing implementation details:

Module Component	Purpose	Visibility Control
Public Interface	Types and functions exposed to clients	Declared in module signature
Private Implementation	Internal helper functions and types	Hidden from clients
Sub-modules	Nested modules for hierarchical organization	Controlled by signature inclusion
Type Abstraction	Abstract types with hidden representations	Signature declares type without definition

Module type checking requires **signature matching**:

1. **Interface Compliance**: Module implementation must provide all signature-declared items
2. **Type Compatibility**: Implementation types must match or be more specific than signature types
3. **Access Control**: Client code can only reference signature-visible items
4. **Abstract Type Sealing**: Abstract types hide their concrete representations

The type representation needs **module-aware types**:

Module Component	OCaml Definition	Purpose
`module_path`	`string list`	Qualified module names like `Data.List.Utils`
`TModulePath`	`TModulePath of module_path * string`	Type references across modules
`signature_item`	Algebraic type for interface declarations	Types, values, modules in signatures
`module_signature`	`signature_item list`	Complete module interface specification

Decision: Module System Complexity Level

- **Context**: Module systems range from simple namespaces to sophisticated functors with type sharing
- **Options Considered**:
  1. Simple qualified namespaces with import/export
  2. ML-style modules with signatures and abstract types
  3. Full functor system with parameterized modules
- **Decision**: ML-style modules without functors initially, designed for functor extension
- **Rationale**: Provides significant organizational benefits without the complexity of module-level polymorphism
- **Consequences**: Enables large-scale development while keeping type checker implementation manageable

Module compilation requires **separate type checking** with **interface consistency**:

Compilation Phase	Input	Output	Verification
Interface Compilation	Module signature source	Compiled signature artifact	Syntax and kind checking
Implementation Compilation	Module implementation + signature	Compiled module + type info	Signature compliance
Client Compilation	Client code + imported signatures	Type-checked client code	Interface-only type checking

Performance Optimizations and Incremental Checking

Incremental type checking enables **fast development cycles through selective recomputation** — think of it as **smart caching for compilation** where changing one function doesn't require re-checking the entire codebase, similar to how modern build systems only rebuild changed components and their dependencies.

The fundamental challenge is **maintaining correctness while minimizing work**. When a function signature changes, we must re-check all its callers, but functions that don't depend on the change can reuse their previous type checking results. This requires tracking **type-level dependencies** with sufficient precision to enable safe incremental updates.

Incremental checking builds on several optimization techniques:

Optimization Category	Technique	Benefit
Memoization	Cache constraint generation and solving results	Avoid recomputation of identical expressions
Dependency Tracking	Record which types depend on which definitions	Enable minimal recomputation on changes
Early Termination	Stop checking when errors make further progress impossible	Reduce work in error-heavy scenarios
Parallel Checking	Check independent modules simultaneously	Utilize multiple CPU cores

The incremental architecture requires **change impact analysis**:

1. **Change Detection**: Identify which definitions have been modified
2. **Dependency Resolution**: Find all code that might be affected by changes
3. **Cache Invalidation**: Remove stale results that might now be incorrect
4. **Minimal Recomputation**: Re-check only the affected components
5. **Result Integration**: Combine new results with cached results

> **Decision: Granularity of Incremental Checking**
>
> - **Context**: Incremental checking can operate at different granularities from expression-level to module-level
> - **Options Considered**:
>   1. Function-level incremental checking with fine-grained dependency tracking
>   2. Module-level incremental checking with interface-based dependencies
>   3. Hybrid approach with module boundaries and intra-module function dependencies
> - **Decision**: Module-level incremental checking with function-level dependencies within changed modules
> - **Rationale**: Balances implementation complexity with performance gains, leveraging module system for coarse-grained caching
> - **Consequences**: Requires module interface stability analysis but provides significant speedup for large codebases

The dependency tracking system needs several data structures:

Dependency Component	Purpose	Structure
Type Dependency Graph	Track which types reference other types	`(type_id * type_id list) list`
Function Dependency Graph	Track function call relationships	`(function_id * function_id list) list`
Module Dependency Graph	Track inter-module dependencies	`(module_id * module_id list) list`
Cache Validation Tags	Detect when cached results are stale	Content hash + dependency hash

Performance optimization also benefits from **constraint solving improvements**:

Algorithm Enhancement	Benefit	Implementation
Union-Find Unification	Faster type variable unification	Path compression and union by rank
Constraint Simplification	Reduce constraint set size before solving	Early constraint decomposition
Occurs Check Optimization	Faster infinite type detection	Lazy occurs checking with memoization

Implementation Guidance

The future extensions represent significant undertakings that build upon the foundation established in the four core milestones. Each extension can be implemented incrementally, leveraging the existing architecture while adding new capabilities.

Technology Recommendations:

Extension	Simple Approach	Advanced Approach
Algebraic Data Types	Extend `ty` with sum/product constructors	Full coverage analysis with decision trees
Type Classes	Simple instance table with linear search	Constraint-based resolution with coherence checking
Row Types	Basic field presence constraints	Full row polymorphism with kind system
Effect Systems	Simple effect annotations	Regional effect inference with masking
Higher-Kinded Types	Explicit kind annotations	Full kind inference with constructor classes
Dependent Types	Basic refinement types with simple predicates	SMT integration with automated theorem proving

Recommended Extension Order:

The extensions have dependencies that suggest a natural implementation order:

1. **Algebraic Data Types** - Foundation for structured data and pattern matching
2. **Module Systems** - Essential for large-scale code organization
3. **Type Classes** - Builds on ADTs and modules for structured polymorphism
4. **Row Types** - Independent feature that enhances record flexibility
5. **Effect Systems** - Builds on type classes for computational context tracking
6. **Higher-Kinded Types** - Advanced abstraction building on type classes
7. **Dependent Types** - Most complex extension requiring sophisticated constraint solving
8. **Performance Optimizations** - Cross-cutting improvements for production use

Extension Integration Points:

Each extension integrates with the existing architecture at specific points:

Extension	Type Representation Changes	Constraint Generation Changes	Unification Changes
ADTs	Add `TSum`, `TProduct` constructors	Constructor application constraints	Pattern exhaustiveness checking
Type Classes	Add `TQualified` with constraints	Instance resolution constraints	Constraint solving phase
Row Types	Add `TRow` with field types	Field access constraints	Row unification algorithm
Effect Systems	Add `TEffect` with effect sets	Effect accumulation constraints	Effect subtyping rules

Milestone Checkpoints for Extensions:

Each extension should be implemented with clear checkpoint verification:

Algebraic Data Types Checkpoint:

- Define simple sum type (e.g., `Option = None | Some of 'a`)
- Implement pattern matching with exhaustiveness checking
- Verify constructor type checking works correctly
- Test error messages for missing pattern cases

Type Classes Checkpoint:

- Define `Eq` type class with `equal` method
- Implement instance for `int` type
- Write polymorphic function using `Eq` constraint
- Verify instance resolution produces correct types

Module System Checkpoint:

- Create module with signature hiding internal types
- Import module in client code using qualified names
- Verify abstract type safety (client cannot access representation)
- Test module compilation and interface consistency

Common Pitfalls for Extensions:

⚠️ **Pitfall: Feature Interaction Complexity** When implementing multiple extensions simultaneously, their interactions can create unexpected complexity. For example, type classes with higher-kinded types require sophisticated constraint resolution, while dependent types with modules need careful abstraction boundary management. Implement extensions incrementally and test their interactions thoroughly.

⚠️ **Pitfall: Performance Regression** Advanced type system features can significantly impact compilation performance. Constraint solving becomes more expensive with type classes, row unification is complex, and dependent type checking requires theorem proving. Profile your implementation and consider optimization strategies early.

⚠️ **Pitfall: Error Message Quality Degradation** Sophisticated type system features often produce complex error messages that confuse users. Type class resolution failures, row type mismatches, and dependent type constraint violations need specially crafted error reporting to remain understandable.

Infrastructure for Extensions:

Several extensions benefit from shared infrastructure:

OCAML

```ocaml
(* Extended type representation supporting multiple features *)
type extended_ty =
  | TInt | TBool | TString
  | TVar of string
  | TFun of extended_ty * extended_ty
  | TForall of string list * extended_ty
  (* ADT support *)
  | TSum of (string * extended_ty list) list
  | TProduct of (string * extended_ty) list
  (* Type class support *)
  | TQualified of constraint_ list * extended_ty
  (* Row type support *)
  | TRow of (string * extended_ty) list * extended_ty option
  | TRecord of extended_ty
  (* Effect system support *)
  | TEffect of extended_ty * effect_set
  (* Higher-kinded type support *)
  | TApp of extended_ty * extended_ty
  | TKindedVar of string * kind
  (* Dependent type support *)
  | TRefined of extended_ty * predicate
  | TDependent of string * extended_ty * extended_ty

(* Unified constraint system supporting all extensions *)
type unified_constraint =
  | Equal of extended_ty * extended_ty
  | Instance of string * extended_ty  (* Type class constraint *)
```

```
  | HasField of extended_ty * string * extended_ty  (* Row constraint *)
  | Effect of extended_ty * effect_type  (* Effect constraint *)
  | Kind of extended_ty * kind  (* Kind constraint *)
  | Predicate of predicate * extended_ty  (* Dependent constraint *)

(* Extended environment supporting modules and type classes *)
type extended_env = {
  term_env: (string * binding) list list;
  type_env: (string * kind) list list;
  class_env: (string * class_info) list;
  instance_env: (constraint_ * instance_info) list;
  module_env: (string * module_signature) list;
}

(* Unified constraint solver handling all constraint types *)
let solve_unified_constraints constraints =
  (* TODO: Implement constraint solving for multiple extension types *)
  (* TODO: Handle constraint interactions between different extensions *)
  (* TODO: Provide unified error reporting across all constraint types *)
  failwith "Extended constraint solving not implemented"
```

The future extensions transform our basic type checker into a production-ready system supporting modern programming language features. Each extension builds upon the solid foundation of our core type system while adding new capabilities that enhance programmer productivity and program safety. The modular architecture ensures that extensions can be implemented incrementally without disrupting existing functionality.

Glossary

> **Milestone(s):** Milestone 1 (Type Representation), Milestone 2 (Basic Type Checking), Milestone 3 (Type Inference), Milestone 4 (Polymorphism)

Think of this glossary as a **comprehensive technical dictionary** — much like how a specialized medical dictionary defines every term precisely so that surgeons, nurses, and medical students all understand exactly what each procedure, organ, and condition means. In type theory, precision of language is critical because subtle differences in terminology often reflect important algorithmic or theoretical distinctions that can make or break an implementation.

The type checking domain draws from several academic fields including programming language theory, mathematical logic, and compiler design. Each field contributes specialized vocabulary that has evolved over decades of research. Understanding these terms precisely is essential for implementing a correct type checker, reading academic literature, communicating with other compiler engineers, and debugging complex type system issues.

This glossary organizes terms into logical categories to help you understand how concepts relate to each other. Terms are defined not just formally, but with practical context about how they appear in our type checker implementation.

Core Type Theory Terminology

Type Theory Foundations

Term	Definition	Practical Context
type	A classification of values that determines what operations are valid and how those values behave during program execution	In our system, represented by the `ty` algebraic data type with constructors for primitives, functions, and type variables
type system	A set of rules that assigns types to program constructs and verifies that operations are used correctly according to their type constraints	Our type checker implements a Hindley-Milner style system with let-polymorphism and principal types
type safety	The property that well-typed programs cannot go wrong at runtime due to type-related errors like calling a non-function or accessing non-existent record fields	Achieved through our static type checking that rejects programs with type violations before execution
type soundness	The formal guarantee that if a program passes type checking, it will not encounter type errors during execution (progress and preservation properties)	Our type checker aims for soundness by implementing complete type rules and conservative error handling
static typing	Type checking performed at compile time before program execution, catching type errors early and enabling optimizations	Distinguished from dynamic typing where type checks happen at runtime
structural typing	Type compatibility determined by the structure or shape of types rather than explicit declarations (duck typing principle)	Our `type_equal` function implements structural equality for function types and other constructed types
nominal typing	Type compatibility based on explicit type declarations and names rather than structure alone	Less relevant for our Hindley-Milner system but important for understanding design alternatives

Type Classifications

Term	Definition	Practical Context
primitive type	Basic built-in types provided directly by the language or type system	Represented as `TInt`, `TBool`, and `TString` in our `ty` type definition
compound type	Types constructed from other types using type constructors like functions, records, or algebraic data types	Function types `TFun of ty * ty` are our primary compound types
function type	A type representing functions with parameter types and return type, often written as `t1 -> t2`	Implemented as `TFun of ty * ty` where first component is parameter type and second is return type
type constructor	A function that builds new types from existing types, like List or Array taking element types as parameters	Our `TFun` constructor builds function types from parameter and return types
monomorphic type	A type with no type variables, representing a concrete fixed type like `int` or `string -> bool`	Distinguished from polymorphic types that contain type variables
polymorphic type	A type containing type variables that can be instantiated with different concrete types at different use sites	Represented using `TVar` constructors and managed through our generalization and instantiation mechanisms

Type Variables and Unification

Type Variable Management

Term	Definition	Practical Context
type variable	A placeholder representing an unknown type that will be determined through type inference and unification	Represented as `TVar of string` with unique names generated by `fresh_type_var()`
fresh variable	A newly generated type variable with a unique name that doesn't conflict with any existing variables	Essential for avoiding variable capture during instantiation and ensuring correct scoping
free variable	A type variable that appears in a type but is not bound by any quantifier in the current context	Important for generalization - only free variables can be generalized at let-bindings
bound variable	A type variable that is bound by a quantifier like forall, making it a parameter rather than an unknown	Quantified variables in type schemes are bound and cannot be generalized again
variable capture	The incorrect substitution that accidentally binds a free variable to an unintended quantifier	Prevented by using fresh variables during instantiation and careful substitution implementation
occurs check	A test during unification that prevents creating infinite types by checking if a type variable occurs within the type it would be unified with	Implemented in `occurs_check(var_name, ty)` to prevent pathological cases like `X = X -> int`

Constraint Solving and Unification

Term	Definition	Practical Context
constraint	An equality requirement between two types that must be satisfied for type checking to succeed	Represented as `constraint_t` with two `ty` components that must be unified
constraint generation	The process of collecting type equality requirements from expressions and statements during type inference	Implemented in `generate_constraints(expr, env)` which traverses the AST and accumulates constraints
unification	The process of finding substitutions that make a set of type constraints simultaneously satisfiable	Our core algorithm implemented in `unify(t1, t2)` using Robinson's unification algorithm
most general unifier	The least restrictive substitution that satisfies a set of constraints, preserving maximum polymorphism	The goal of our unification algorithm - finding solutions that don't over-constrain types
Robinson unification	The standard unification algorithm for first-order terms with occurs check to handle recursive structures	The algorithmic foundation of our `unify` function with proper handling of type variables
substitution	A mapping from type variables to concrete types that represents partial or complete solutions to type constraints	Implemented as `substitution` type containing a list of variable-to-type mappings
substitution composition	Combining two substitutions into a single equivalent substitution by applying the first then the second	Critical operation implemented in `compose_subst(s1, s2)` for building up solutions incrementally
substitution application	Applying a substitution to a type by replacing type variables with their mapped types	Implemented in `apply_subst_to_type(subst, ty)` to realize the effects of constraint solving

Type Inference and Checking

Inference Modes and Strategies

Term	Definition	Practical Context
type inference	The process of automatically determining types for expressions without explicit type annotations	The core capability of our system, allowing programmers to omit many type declarations
type checking	The process of verifying that explicitly annotated types are consistent with actual usage	Complementary to inference - checking user-provided annotations against inferred requirements
bidirectional typing	A type checking approach that uses both synthesis (inferring types) and checking (verifying against expected types)	Our system uses synthesis for most expressions and checking for contexts with known expected types
expression synthesis	Determining what type an expression produces based on its structure and the types of its subexpressions	Implemented in `synthesize_expression_type(expr, env)` for literals, variables, and function calls
expression checking	Verifying that an expression has a specific expected type, possibly inferring internal details	Implemented in `check_expression_type(expr, expected, env)` for function bodies and assignments
constraint-based inference	Type inference that works by generating constraints and solving them rather than direct type assignment	Our approach - generate constraints during AST traversal, then solve them via unification
principal type	The most general type that can be assigned to an expression - all other valid types are instances of the principal type	The goal of Hindley-Milner inference - finding maximally polymorphic types

Type Environment and Scoping

Term	Definition	Practical Context
type environment	A symbol table mapping variable names to their types or type schemes, organized in scopes	Implemented as `env` with nested scope structure using list of binding lists
lexical scoping	Variables are visible in their declared scope and all nested scopes, following the program's text structure	Our environment structure supports this with scope extension and variable lookup through scope layers
environment threading	Passing updated type environments through recursive type checking calls to maintain correct scoping	Critical pattern in our checking functions that process statements and binding constructs
scope extension	Adding a new scope layer to the environment for constructs like let-bindings or function parameters	Implemented through `fresh_scope(env)` and binding extension operations
variable lookup	Searching through scope layers to find the type binding for a variable name	Implemented in `lookup(name, env)` with inside-out search through nested scopes
binding	An association between a variable name and its type information (monomorphic or polymorphic)	Represented as `binding` type distinguishing between `Monomorphic of ty` and `Polymorphic of type_scheme`

Polymorphism and Type Schemes

Polymorphic Type System Concepts

Term	Definition	Practical Context
polymorphism	The ability for a single piece of code to work with multiple types, enabling generic programming	Our system supports parametric polymorphism through type variables and let-generalization
parametric polymorphism	Polymorphism where functions work uniformly over all types, without inspecting type structure	Distinguished from ad-hoc polymorphism (overloading) - our functions are parametric in type arguments
let-polymorphism	A form of polymorphism where types are generalized at let-bindings and instantiated at use sites	The core polymorphic mechanism in Hindley-Milner systems, enabling functions like identity to have polymorphic types
type scheme	A type with explicit quantification over type variables using forall, representing polymorphic types	Implemented as `type_scheme` with `Forall of string list * ty` containing quantified variables and body type
quantified variables	Type variables explicitly bound by forall quantifiers in type schemes	The string list in our `Forall` constructor containing names of variables abstracted over
generalization	The process of abstracting over free type variables at let-bindings to create polymorphic type schemes	Implemented in `generalize_type(inferred_type, env, expr)` with value restriction for safety
instantiation	Replacing quantified variables in a type scheme with fresh type variables at use sites	Implemented in `instantiate_scheme(scheme)` to create monomorphic instances of polymorphic types
value restriction	The limitation that only syntactic values can be generalized, preventing unsoundness with mutable references	Checked in `can_generalize(expr)` to ensure only safe expressions receive polymorphic types

Template and Generic Programming Concepts

Term	Definition	Practical Context
template instantiation	Creating concrete types from polymorphic templates by replacing type parameters with specific types	Similar to our type scheme instantiation but emphasizes the template-like nature of polymorphic definitions
generic function	A function defined over type parameters that can work with multiple concrete types	Functions with polymorphic type schemes in our system, instantiated differently at each call site
type parameter	A type variable that serves as a parameter to generic functions or data types	The quantified variables in our type schemes that get replaced during instantiation
monomorphization	The compilation technique of creating separate instances of generic code for each concrete type used	Not implemented in our type checker but relevant for understanding how polymorphism is realized
polymorphic recursion	Recursive functions that call themselves at different types, requiring more sophisticated inference	Beyond our current system but important for understanding limitations of simple let-polymorphism

Error Handling and Recovery

Type Error Classification

Term	Definition	Practical Context
type error	A violation of type system rules detected during static analysis, indicating potential runtime failures	Represented by our `type_error` record with location, expected/actual types, and diagnostic information
type mismatch	The most common type error where an expression has one type but another type was expected	Represented as `TypeMismatch` error kind with expected and actual types for clear diagnostic messages
unbound variable	A variable reference that doesn't correspond to any visible binding in the current scope	Tracked as `UnboundVariable of string` error kind with the problematic variable name
arity mismatch	Calling a function with the wrong number of arguments compared to its declared parameter count	Represented as `WrongArity of int * int` storing expected and actual argument counts
non-function call	Attempting to call an expression that doesn't have a function type	Tracked as `NonFunctionCall` error kind when type checking function application expressions
cascading errors	Spurious errors that occur as a consequence of previous errors rather than independent problems	Managed through our error recovery system that provides placeholder types to continue analysis

Error Recovery and Reporting

Term	Definition	Practical Context
error recovery	Continuing type checking after encountering errors to find multiple independent issues in one pass	Implemented through `recover_from_error(err_kind, expected)` that provides sensible placeholder types
graceful degradation	Making reasonable assumptions to continue analysis when encountering errors or ambiguities	Our approach of substituting error types while still checking the rest of the program structure
source location tracking	Maintaining file position information for error messages that help programmers locate and fix problems	Implemented in `source_location` record with file, line, and column information for precise error reporting
context-aware errors	Error messages that explain not just what's wrong but why it's wrong in the current context	Our error reporting includes expected vs actual types and contextual suggestions for common fixes
error suppression logic	Preventing the reporting of errors that are likely consequences of previously reported errors	Helps reduce noise in error output by focusing on root causes rather than cascading effects
suggestion generation	Automatically proposing likely fixes or alternatives when type errors are detected	Implemented in `generate_suggestions(err_kind, env)` to help developers resolve type issues quickly

Algorithm Names and Techniques

Core Algorithms

Term	Definition	Practical Context
Hindley-Milner type inference	The classic type inference algorithm for lambda calculus with let-polymorphism and principal types	The theoretical foundation of our type system, providing complete and decidable inference
Algorithm W	Damas-Hindley algorithm for type inference that generates constraints and solves them via unification	The specific algorithmic approach underlying our constraint generation and solving phases
Robinson unification	The standard unification algorithm for first-order terms with occurs check for preventing infinite types	Implemented in our `unify` function as the core constraint solving mechanism
occurs check	Prevention of infinite types during unification by checking if a variable occurs in the type it would unify with	Critical safety mechanism in `occurs_check(var_name, ty)` preventing pathological unifications
constraint decomposition	Breaking complex type constraints into simpler constraints that can be unified directly	Part of our unification algorithm that handles function types by decomposing them into parameter and return constraints
substitution composition	Combining multiple substitutions into a single equivalent substitution following mathematical composition rules	Essential operation for building up solutions incrementally during constraint solving

Specialized Techniques

Term	Definition	Practical Context
pipeline orchestration	Coordinating multiple phases of type checking with proper error handling and data flow between stages	Our overall architecture managing the flow from parsing through constraint generation to final type assignment
constraint solving	The process of finding substitutions that satisfy all generated type equality constraints simultaneously	The unification phase that takes constraints from inference and produces type variable bindings
type variable generation	Creating unique type variables for unknowns during inference while avoiding name collisions	Implemented through `fresh_type_var()` and `fresh_var_name()` for systematic variable management
environment management	Maintaining type environments correctly through scope changes, extensions, and recursive traversals	Critical for correct scoping behavior in our type checking functions
inference quality validation	Ensuring that inferred types are principal (most general) and that the algorithm produces optimal results	Important for debugging and verifying that our implementation matches theoretical properties

Testing and Verification Terminology

Testing Strategies

Term	Definition	Practical Context
milestone verification	Checkpoint testing to ensure each development phase is solid before proceeding to the next	Our structured approach with specific acceptance criteria and deliverables for each milestone
systematic coverage	Testing across multiple orthogonal dimensions to ensure comprehensive validation of system behavior	Testing primitive types × operations × error conditions × polymorphic contexts systematically
regression testing	Verifying that previously working functionality still works correctly after making changes	Essential for maintaining correctness as we add features like polymorphism and error recovery
most general type verification	Testing that type inference produces maximally polymorphic types rather than over-constrained monomorphic ones	Validating that our generalization algorithm works correctly and produces principal types
error recovery testing	Verifying that the system handles invalid programs gracefully and continues to find additional errors	Testing our error recovery mechanisms produce sensible behavior in the presence of type violations
acceptance criteria verification	Confirming that implemented features meet the specific requirements defined in milestone specifications	The systematic validation approach ensuring we've actually solved the problems we set out to solve

Debugging and Diagnostics

Term	Definition	Practical Context
constraint tracing	Following the generation and solving of type constraints to understand inference behavior	Implemented through `trace_constraint_generation(expr, constraints)` for debugging inference issues
unification debugging	Examining why unification fails or produces unexpected results during constraint solving	Essential for diagnosing type inference problems and understanding algorithm behavior
environment inspection	Examining the structure and contents of type environments to debug scoping and lookup issues	Provided by `inspect_environment(env)` to visualize scope structure and binding contents
inference validation	Checking that inferred types match expected types and that the inference process produces correct results	Implemented in `validate_inference_result(expr, expected, inferred)` for systematic verification
type error diagnosis	Understanding why type errors occur and what the underlying causes are	Our structured approach to error analysis with symptom-cause-fix methodology

Advanced and Extension Terminology

Beyond Basic Type Systems

Term	Definition	Practical Context
algebraic data types	Structured data blueprints with multiple construction patterns like sum types and product types	Potential extension represented through `TSum` and `TConstructor` in our extended type system
type classes	Structured polymorphism with constraints that enable ad-hoc polymorphism and overloading	Advanced feature that would extend our system with constrained types and instance resolution
row types	Flexible record structures that support structural subtyping and extensible records	Represented in extensions as `TRow` with field lists and optional row variables
effect systems	Compile-time tracking of computational side effects like IO, mutation, and exceptions	Potential extension through `TEffect` types that track effect sets alongside regular types
higher-kinded types	Abstraction over type constructors rather than just types, enabling generic programming over containers	Represented through `kind` types and `TKinded` constructors for types with non-trivial kinds
dependent types	Types that depend on runtime values, enabling more precise specification and verification	Advanced extension through `TDependent` types that can express properties of program values
module systems	Large-scale code organization with controlled visibility and abstraction boundaries	Potential extension through `TModulePath` and signature systems for organizing large programs

Advanced Implementation Techniques

Term	Definition	Practical Context
incremental type checking	Fast development cycles through selective recomputation when program parts change	Advanced optimization that would cache type checking results and recompute only affected parts
exhaustiveness checking	Verifying that all possible constructors are handled in pattern matching constructs	Extension for algebraic data types that ensures complete coverage of sum type alternatives
instance resolution	Finding applicable instances for type class constraints during compilation	Advanced algorithm for type class systems that resolves overloaded operations to specific implementations
SMT solver integration	Using automated theorem provers for constraint verification in advanced type systems	Potential approach for handling complex constraints in dependent type systems or refinement types
kind inference	Inferring kinds alongside type inference for higher-kinded type systems	Extension of our inference algorithm to handle type constructor abstraction

> **Key Insight**: Understanding this terminology precisely is essential for implementing correct algorithms, reading research literature, and communicating effectively with other type system implementers. Each term represents not just a concept but often a specific algorithmic challenge or design decision that affects the behavior and capabilities of your type checker.

Common Terminology Pitfalls

⚠️ **Pitfall: Confusing "type checking" and "type inference"** Many beginners use these terms interchangeably, but they represent different processes. Type checking verifies that explicit annotations match usage, while type inference determines types for unannotated expressions. Our system does both - it checks explicit annotations and infers types where they're missing.

⚠️ **Pitfall: Misunderstanding "polymorphism" scope** Not all polymorphism is the same. Parametric polymorphism (our focus) is different from ad-hoc polymorphism (overloading) and subtype polymorphism (inheritance). When discussing polymorphic types, be specific about which form you mean.

⚠️ **Pitfall: Conflating "substitution" and "unification"** Substitution is a mapping from variables to types, while unification is the algorithm that finds substitutions. Unification produces substitutions as its result, but the two concepts are distinct and serve different roles in the type checking process.

⚠️ **Pitfall: Assuming "fresh variables" are optional** Fresh variable generation is critical for correctness, not just convenience. Using non-fresh variables during instantiation can cause variable capture and incorrect type checking results. Always use `fresh_type_var()` when creating new unknowns.

⚠️ **Pitfall: Treating "generalization" and "instantiation" as inverses** While related, these operations are not simple inverses. Generalization abstracts over specific type variables in specific contexts, while instantiation creates fresh variables. The relationship is more subtle than mathematical function inversion.

Implementation Guidance

A. Technology Recommendations Table:

Component	Simple Option	Advanced Option
Term Definition Storage	Hardcoded pattern matching on term strings	Configurable glossary database with metadata
Pretty Printing	String concatenation with basic formatting	Structured formatter with syntax highlighting
Cross-References	Manual term linking in documentation	Automatic hyperlink generation with validation
Term Validation	Manual consistency checking	Automated terminology checking against codebase

B. Recommended File/Module Structure:

```
type-checker/
  lib/
    glossary/
      glossary.ml              ← term definitions and utilities
      glossary.mli             ← public interface for term lookup
      pretty_print.ml          ← formatting utilities for terms
      cross_reference.ml       ← linking related terms
    types/
      types.ml                 ← core type definitions
    inference/
      inference.ml             ← type inference implementation
  docs/
    terminology.md             ← human-readable glossary
    cross_reference.md         ← term relationships
  tests/
    test_glossary.ml           ← terminology consistency tests
```

OCAML

C. Infrastructure Starter Code (COMPLETE, ready to use):

OCAML

```ocaml
(* glossary.ml - Complete term definition and lookup system *)

type term_category =
  | CoreType
  | Algorithm
  | ErrorHandling
  | Polymorphism
  | Advanced

type term_definition = {
  name: string;
  category: term_category;
  definition: string;
  practical_context: string;
  related_terms: string list;
  code_references: string list;
}

let term_database : (string, term_definition) Hashtbl.t = Hashtbl.create 128

let add_term name category definition practical_context related_terms code_refs =
  let term = {
    name;
    category;
    definition;
    practical_context;
    related_terms;
    code_references = code_refs
  } in
```

```
  Hashtbl.add term_database name term

let lookup_term name =
  try Some (Hashtbl.find term_database name)
  with Not_found -> None

let terms_by_category category =
  Hashtbl.fold (fun _ term acc ->
    if term.category = category then term :: acc else acc
  ) term_database []

let find_related_terms term_name =
  match lookup_term term_name with
  | Some term ->
    List.filter_map (fun related -> lookup_term related) term.related_terms
  | None -> []

(* Initialize core terms *)
let () =
  add_term "type variable" CoreType
    "A placeholder representing an unknown type that will be determined through type inference and unification"
    "Represented as TVar of string with unique names generated by fresh_type_var()"
    ["fresh variable"; "unification"; "substitution"]
    ["TVar"; "fresh_type_var()"; "occurs_check"];


  add_term "unification" Algorithm
    "The process of finding substitutions that make a set of type constraints simultaneously satisfiable"
```

```
    "Our core algorithm implemented in unify(t1, t2) using Robinson's unification
algorithm"

    ["constraint"; "substitution"; "occurs check"]

    ["unify"; "Robinson unification"; "constraint_t"];



  add_term "let-polymorphism" Polymorphism

    "A form of polymorphism where types are generalized at let-bindings and instantiated at
use sites"

    "The core polymorphic mechanism in Hindley-Milner systems, enabling functions like
identity to have polymorphic types"

    ["generalization"; "instantiation"; "type scheme"]

    ["generalize_type"; "instantiate_scheme"; "type_scheme"]
```

D. Core Logic Skeleton Code (signature + TODOs only):

OCAML

```
(* Term validation and consistency checking *)


(* validate_terminology_consistency - Check that all terms used in code match glossary
definitions *)

let validate_terminology_consistency (source_files: string list) : (string * string list)
list =

  (* TODO 1: Parse source files and extract type names, function names, and key terms *)

  (* TODO 2: Cross-reference extracted terms against term_database *)

  (* TODO 3: Report terms used in code but not defined in glossary *)

  (* TODO 4: Report terms defined in glossary but never used in code *)

  (* TODO 5: Check that code comments use preferred terminology from glossary *)

  (* Hint: Use regular expressions to extract identifiers and comments *)

  []


(* generate_cross_reference_map - Build relationships between related terms *)

let generate_cross_reference_map () : (string * string list) list =

  (* TODO 1: Iterate through all terms in term_database *)

  (* TODO 2: For each term, collect its related_terms list *)

  (* TODO 3: Build bidirectional relationships - if A relates to B, ensure B relates to A
*)

  (* TODO 4: Group terms by category and identify cross-category relationships *)

  (* TODO 5: Return mapping from each term to all its directly and indirectly related terms
*)

  []


(* format_term_for_documentation - Pretty print term definition for docs *)

let format_term_for_documentation (term: term_definition) : string =

  (* TODO 1: Format term name as a header with appropriate markdown level *)

  (* TODO 2: Add definition paragraph with clear, precise language *)

  (* TODO 3: Include practical context section with implementation details *)
```

```
  (* TODO 4: Add "Related Terms" section with links to other definitions *)

  (* TODO 5: Include "Code References" section with specific function/type names *)

  (* TODO 6: Use consistent formatting style across all term definitions *)

  ""


(* check_definition_completeness - Ensure all important concepts are defined *)

let check_definition_completeness (required_terms: string list) : string list =

  (* TODO 1: Check that all terms in required_terms exist in term_database *)

  (* TODO 2: Verify that core algorithm names are defined (unification, generalization,
etc.) *)

  (* TODO 3: Ensure all error types mentioned in code have glossary entries *)

  (* TODO 4: Check that all major type constructors are documented *)

  (* TODO 5: Return list of missing terms that should be added to glossary *)

  []
```

E. Language-Specific Hints:

- Use OCaml's Hashtbl for efficient term lookup in large glossaries
- Pattern matching on term_category makes it easy to group related concepts
- String.split_on_char and Str.regexp are useful for parsing source files to extract terminology
- List.filter_map is perfect for collecting related terms that exist in the database
- Printf.sprintf provides good control over documentation formatting
- File I/O functions like open_in and input_line help process source files for validation

F. Milestone Checkpoint:

After building your glossary system:

- Run `ocamlopt -o check_terms glossary.ml check_terms.ml` to build term validation
- Execute `./check_terms src/` to scan your source code for terminology consistency
- Expected output: "Found 45 terms, 3 undefined: 'constraint_solving', 'bidirectional_checking', 'principal_type'"
- Verify manually: Open your .ml files and confirm that function names match glossary entries
- Signs of issues: Many undefined terms suggests inconsistent naming; no undefined terms might mean glossary is incomplete

G. Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
Term lookup returns None for known terms	Case sensitivity or exact string matching	Print the exact string being looked up vs database keys	Normalize case or use fuzzy matching
Cross-references create circular loops	Bidirectional relationships not handled properly	Trace the related_terms chains manually	Use visited set to prevent infinite loops
Documentation formatting looks inconsistent	Different formatting logic for different term types	Compare generated output for terms in same category	Standardize format_term_for_documentation logic
Terminology validation reports false positives	Extracting code comments or string literals as terms	Check what text patterns the validation regex matches	Refine regex to match only actual identifier usage
Related terms don't show useful connections	Related_terms lists built manually without systematic approach	Review which terms actually share concepts or algorithms	Rebuild relationships based on shared code references