AFP Project: Dependent Type Checker

Radu Nicolae 5527104

June 17, 2025

1 Design Decisions

Rather than building a language that mimics Agda's behavior through special-purpose primitives or syntactic sugar, the central design decision in this project was to implement each construct listed in the grading rubric (e.g., `Bool`, `Nat`, `Vec`) as a user-definable inductive type. These types are not built into the core language; instead, they are implemented as standard inductive declarations and bundled as part of a prelude in the `libraries/` folder. This design choice places the implementation closer to a pure formulation of the **dependent typed lambda calculus**, treating constructs like `True`, `zero`, and `equal` not as primitives but as library-defined entities.

This decision was inspired by Andrej Bauer's minimalist approach to dependent type theory [1]. It emphasizes a small core calculus with user-extensible semantics rather than a large, hard-coded language with dozens of typing rules. A major benefit of this approach is conceptual minimalism: only a few constructs (lambdas, Pis, matches, etc.) need to be type-checked, and all higher-level abstractions are compiled into these.

However, this purity comes with significant drawbacks:

- **Tight Coupling of Phases:** Type checking, compilation, and execution are inherently interleaved. For example, evaluating a match expression may require normalization that includes applying type-level functions.
- **Difficult Debugging and Reasoning:** Unlike more segmented pipelines, it's harder to pinpoint errors as they may surface in intertwined phases.
- **Steep Learning Curve:** For newcomers, the absence of named primitives makes it harder to grasp what is "built in" versus user-defined.
- **Performance Costs:** Pattern matching and normalization are handled in the type checker itself, leading to deep recursion and higher complexity in examples that would otherwise be trivial.

Despite these limitations, this design enforces a deep understanding of dependent type theory and treats type-level computation as a first-class citizen. It also facilitates advanced patterns like type-indexed constructors and equality proofs without extending the core language or adding ad hoc mechanisms.

1.1 Language Goals

Our language is designed to support:

- First-class dependent types, including types indexed by values (e.g., `Vec A n`)
- Pattern matching over inductive data types
- Type-level reasoning and normalization
- A minimal, lambda-calculus-style core representation

These goals shaped every layer of the system, especially how we desugar user syntax into a uniform intermediate representation (IR) and check it against a dependent type system.

1.2 Overall Architecture

The implementation follows a staged transformation pipeline:

1. **Parsing:** Surface syntax is parsed into an abstract syntax tree (`Lang.Abs`) using BNFC.
2. **Desugaring:** Multiple passes eliminate syntactic sugar (e.g., natural number literals, multi-argument lambdas, simple Pi types).
3. **IR Conversion:** All named binders are replaced with de Bruijn indices for robust variable tracking.
4. **Type Checking:** The type checker infers or validates types for all expressions using a context and normalization strategy.
5. **Evaluation:** Final results are normalized and presented in a human-readable form.

Each stage is implemented in a dedicated module, allowing for clarity, testing, and debugging at each abstraction level.

> For additional documentation on the implementation — including monads, effects, function descriptions, and data types — see the `README.md` in the project repository.

1.3 Desugaring Pipeline

We chose to perform desugaring in distinct passes (e.g., `NatDesugar`, `FunctionDesugar`, `LambdaDesugar`, `InductiveDesugar`) rather than in a single monolithic pass. This modular approach offered several benefits:

- Improved debugging and logging at each stage
- Simpler, composable transformations
- Clear dependency management (e.g., lambdas must be nested before applying de Bruijn conversion)

Each transformation outputs a valid `Program`, allowing validation after each stage if needed.

1.4 Use of De Bruijn Indices

We convert named binders to de Bruijn indices using the `BruijnConvertor` module. This was a key decision to:

- Avoid name capture and scoping errors
- Enable efficient substitution and normalization
- Simplify α-equivalence during type checking

Although it complicates debugging (indices are less human-readable), we mitigate this by restoring names during pretty-printing using the context.

1.5 Context Representation

Our context is a list of 4-tuples: `(type, value, name, domain)`. This allows:

- Access to declared and inferred types
- Value substitutions during normalization
- Name resolution for user-friendly output
- Tracking of domain restrictions in dependent pattern matches

This design trades performance for simplicity and completeness — which is acceptable given our focus on correctness over speed.

1.6 Pattern Matching and Normalization

One of the most complex aspects was implementing recursive normalization for match expressions. To support dependent pattern matching, we:

- Built substitution contexts dynamically based on matched patterns
- Applied shifting and substitution in branches
- Ensured branch types unify properly, supporting recursive inference

Pattern checking is implemented through first-order unification between the scrutinee and the patterns. During type checking, the system constructs substitution contexts and applies normalization recursively to determine whether a pattern matches. This mechanism is generic and supports all user-defined inductive types with no need for primitive hardcoded cases.

This required careful bookkeeping and recurrence control to avoid nontermination and false negatives.

1.7 Trade-offs and Alternatives

- We opted for explicit desugaring rather than parser-level rewrites for maintainability.
- De Bruijn indices added complexity in debugging but avoided errors from shadowing or renaming.
- We used list-based contexts for simplicity, though a more optimized structure (e.g., zipper trees or indexed maps) could improve performance.

1.8 Example Programs

This section presents illustrative programs written in our language to demonstrate its dependent type system, inductive data modeling, and elimination mechanisms.

The first listing below shows the definition of the `Bool` type from the standard library. It includes:

- An inductive declaration of `Bool` with two constructors: `true` and `false`.
- A dependent eliminator `elimBool`, which takes a motive (a type family over `Bool`), a case for `true`, a case for `false`, and a value of type `Bool`.

This structure enables case analysis over `Bool` in a dependently typed way, similar to eliminators in proof assistants like Agda or Coq.

```
data Bool : U 0 where
    true : Bool ,
    false : Bool,
end ;

bind elimBool : (P : Bool -> U 0) -> P true
            -> P false -> (b : Bool) -> P b ;
let elimBool P pt pf b = case b of
  true => pt,
  false => pf
end ;

Bool
```

Listing 1: Library definition of the `Bool` type and its dependent eliminator `elimBool`.

The second listing contains a test program that uses `elimBool` to compute a result based on a value of type `Bool`. It declares:

- A motive `P : Bool → U 0` that assigns a different result type depending on the boolean value.
- A branch for `true` returning a natural number, and a branch for `false` returning a boolean.
- A final call to `elimBool` that dispatches on the boolean input and returns the corresponding value.

This demonstrates how elimination over inductive types can guide result types based on input — a core feature of dependent pattern matching.

These programs highlight the system's ability to express proofs and computations simultaneously within the type system.

1.9 Reflection and Future Work

While the core goals were met, we encountered several limitations and areas for potential improvement:

```
bind natOrBoolHelper : Bool -> U 0 ;
let natOrBoolHelper b = case b of
    true => Nat ,
    false => Bool
end ;

bind natOrBool : (b : Bool) -> natOrBoolHelper b ;
let natOrBool b = elimBool natOrBoolHelper  zero  true  b;

natOrBool false
```

Listing 2: A test program demonstrating elimination over `Bool` with type-dependent branches.

- **No Implicit Argument Inference:** All parameters must be explicitly passed. Implementing unification and argument inference would make the language more ergonomic.
- **Error Reporting:** Error messages could be improved by showing both the expected and inferred types in a more readable format, especially in deeply nested expressions.
- **Performance:** Pattern matching and normalization are implemented in a straightforward but unoptimized way. Memoization or evaluation strategy tuning (e.g., call-by-need) could significantly speed up complex programs.

2 Typing Rules

This section defines typing rules for the surface language, based on the grammar in `Lang.cf`. We cover expressions, statements, and full programs.

3 Program Desugar Pipeline

This section outlines the transformation stages applied to user programs before type checking and evaluation. The goal is to simplify syntax into a canonical form for semantic analysis. This pipeline is defined in `IR.IrGen`.

3.1 Desugaring Natural Numbers

The `IR.NatDesugar` module converts natural number literals into Peano-style constructors. For instance:

```
3  => suc (suc (suc zero))
```

This enables dependent pattern matching over numeric values.

3.2 Desugaring Inductive Types

The `IR.InductiveDesugar` module lowers inductive type declarations into explicit bindings. It converts:

$$\frac{k \in \mathbb{N}}{\Gamma \vdash \texttt{Type}_k : \texttt{Type}_{k+1}}\ [\text{T-Type}] \qquad \frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ [\text{T-Var}]$$

$$\frac{\Gamma \vdash f : (x : A) \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B[x \mapsto a]}\ [\text{T-App}]$$

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x.e : (x : A) \to B}\ [\text{T-Lam}]$$

$$\frac{\Gamma \vdash A : \texttt{Type}_i \qquad \Gamma, x : A \vdash B : \texttt{Type}_j}{\Gamma \vdash (x : A) \to B : \texttt{Type}_{\max(i,j)}}\ [\text{T-Pi}]$$

$$\frac{\Gamma \vdash A : \texttt{Type}_i \qquad \Gamma \vdash B : \texttt{Type}_j}{\Gamma \vdash A \to B : \texttt{Type}_{\max(i,j)}}\ [\text{T-PiSimple}]$$

$$\frac{\Gamma \vdash e : T \qquad \forall\ (p \to b) \in \texttt{branches},\ \Gamma, p : T \vdash b : R}{\Gamma \vdash \texttt{match}\ e\ \texttt{with}\ \ldots : R}\ [\text{T-Match}]$$

Figure 1: Typing rules for surface expressions

$$\frac{\Gamma \vdash \lambda x_1 \ldots x_n.e : T}{\Gamma \vdash \texttt{let}\ f\ x_1 \ldots x_n = e \Rightarrow \Gamma[f \mapsto T]}\ [\text{T-Let}]$$

$$\frac{\Gamma \vdash e : \texttt{Type}_k}{\Gamma \vdash \texttt{bind}\ x : e \Rightarrow \Gamma[x \mapsto e]}\ [\text{T-Bind}]$$

$$\frac{\Gamma \vdash \texttt{inductive}\ T\ (x_1 : A_1) \ldots (x_n : A_n) : R\ \texttt{with}\ c_1, \ldots, c_m}{\Gamma \vdash T,\ c_1, \ldots, c_m\ \texttt{well-formed}}\ [\text{T-Inductive}]$$

Figure 2: Typing rules for statements

$$\frac{\Gamma_0 \vdash s_1 \Rightarrow \Gamma_1 \qquad \Gamma_1 \vdash s_2 \Rightarrow \Gamma_2 \qquad \cdots \qquad \Gamma_n \vdash e : T}{\Gamma_0 \vdash s_1; s_2; \ldots;\ e : T}\ [\text{T-Program}]$$

Figure 3: Typing rule for full programs

```
inductive Vec (A : Type) : Nat -> Type
| nil : Vec A 0
| cons : A -> Vec A n -> Vec A (suc n)
```

into a `SBind` and multiple `SBindDom` declarations, thus integrating constructors into the typing environment.

3.3 Simple Pi Conversion

The `IR.SimplePiConvertor` transforms shorthand function types into full dependent Pi types:

```
A -> B      =>     (_ : A) -> B
```

This standardization aids later type inference and unification.

3.4 Function Desugaring

In `IR.FunctionDesugar`, function definitions with multiple arguments are converted into nested lambdas:

```
let f x y = e      =>     let f = \x -> \y -> e
```

This aligns with the lambda calculus core language model.

3.5 Lambda Desugaring

`IR.LambdaDesugar` enforces a unary lambda structure, further simplifying function representation.

3.6 De Bruijn Index Conversion

`IR.BruijnConvertor` replaces named variables with de Bruijn indices. This eliminates variable shadowing and simplifies substitution and alpha-equivalence.

- The typing context in `TypeCheck.Context` is implemented as an immutable list, with functions like `extendCtx` and `updateValue` building new versions of the context without side effects.
- Substitution environments use `Data.Map.Strict` to maintain efficient, immutable key-value mappings of expressions.
- The `IR.BruijnConvertor` module employs a `Map` from identifiers to integer indices to convert named variables into de Bruijn indices. This map is updated functionally during traversal and scoping, demonstrating the use of purely functional data structures for environment tracking.

3.7 Summary

These transformations and techniques bring the input program into a normalized, type-safe, and pattern-matchable intermediate form, ready for interpretation and type inference.

4 Functional Programming Techniques

4.1 Purely Functional Data Structures

The project extensively uses immutable and persistent data structures, in line with functional programming principles:

4.2 Lenses and Traversals

Modules like `NatDesugar`, `LambdaDesugar`, and `SimplePiConvertor` use the `Plated` typeclass and `Control.Lens.Plated.transformM` to recursively rewrite expressions. See `IR.PlatedExp`, where lenses are used to generically traverse and modify expression trees.

4.3 Monad Transformers / Free Monads

The main evaluation pipeline in `Main.hs` and `IR.IrGen` uses the custom `AppM` monad from `App.hs`, which wraps monad transformer stacks (e.g., `ReaderT`, `IO`) for structured effectful computation. This allows flexible dependency injection (e.g., debug logger).

4.4 Concurrency

Concurrency is leveraged in two parts of the system:

- **Parallel File Loading:** The function `loadStaticFiles` in `Main.hs` uses `mapConcurrently` from `Control.Concurrent.Async` to load standard library files in parallel, improving startup performance.
- **Concurrent Logging:** The logging infrastructure (in `Logger.hs`, used throughout the pipeline) is designed to be concurrency-safe. Logs are emitted from different parts of the program (e.g., during type checking and parsing) and handled asynchronously, ensuring thread-safe and consistent debug output.

4.5 Type-level Programming

The language supports dependent types (e.g., types indexed by values such as `Vec A n`), as shown in inductive types and Pi expressions. Type-level reasoning is implemented through normalization and explicit universe checking in `TypeCheck.Expr` via `EType level` constructors.

References

[1] A. Bauer, *How to implement dependent type theory (i)*, Accessed: 2025-06-17, 2012. [Online]. Available: `https://math.andrej.com/2012/11/08/how-to-implement-dependent-type-theory-i/`.