

Logical frameworks and language workbenches for dependent type systems

Presentation at AIM XLIII

Jesper Cockx

19-24 October 2026

Technical University Delft

Making software verification accessible

To build better software, we need **verification-aware** languages:

1. Declarative *specifications* of intended behavior
2. Efficient *implementations* of our algorithms
3. Trustworthy *verification* that these two match

To be used widely, verification should be as **accessible** and **intuitive** as possible.

Examples. Dafny, Liquid Haskell, F*

Proving software correctness is just programming

State-of-the-art languages for verification use **dependent types** to write *algorithms* and *proofs* in the same language:

```
data _∈_ {A : Set} : A → List A → Set where
```

```
  here :          x ∈ (x :: xs)
```

```
  there : x ∈ xs → x ∈ (y :: xs)
```

```
split : x ∈ (ys ++ zs) → (x ∈ ys) ⊔ (x ∈ zs)
```

```
split {ys = []}      p      = right p
```

```
split {ys = y :: ys} here    = left here
```

```
split {ys = y :: ys} (there p) = [ there , id ] (split p)
```

The value of language experiments

How to make a verification-aware language **expressive** yet **accessible**?

We don't know (yet).

The proper language design depends on the *application domain* and the *programming paradigm*, so we need to **experiment** with new verification-aware languages.

Language experiments in Agda

Since its conception, Agda has been a **playground** for new type systems and language features:

- Definitional irrelevance
- Copattern matching
- Sized types
- Cubical type theory
- Rewrite rules
- Guarded type theory
- Controlled unfolding
- ...

The limits to extending Agda

This is great for prototyping, but does not scale:

- Each feature needs to be maintained
- Interactions between features are often untested
- Optimizations and refactorings can easily break an assumption of a specific feature

The limits to extending Agda

This is great for prototyping, but does not scale:

- Each feature needs to be maintained
- Interactions between features are often untested
- Optimizations and refactorings can easily break an assumption of a specific feature

Agda's implementation is *crumbling under all its features...*

Can we instead make it easier to build **new** languages for prototyping a feature?

Implementing new dependently typed languages

Implementing a new dependent type checker from scratch is a challenging and highly specialized task.¹

The **kernel** is just the start: we need parsing, desugaring, optimization, code generation, IDE tooling, ...

¹The *Elaboration Zoo* and *smalltt* by András Kovács might be the best way to learn.

Implementing new dependently typed languages

Implementing a new dependent type checker from scratch is a challenging and highly specialized task.¹

The **kernel** is just the start: we need parsing, desugaring, optimization, code generation, IDE tooling, ...

Key question.

*How can we make it easier to experiment
with new dependent type systems?*

¹The *Elaboration Zoo* and *smalltt* by András Kovács might be the best way to learn.

Outline

1. Three challenges
2. Two approaches
3. One vision

Three challenges with implementing dependent types

The challenges that come up when implementing a full dependently typed language are hard to count.

For now, let us focus on three major ones:

1. Representing syntax with **names and binders**
2. Dealing with non-trivial equalities between types
(a.k.a. **type conversion**)
3. Automatic inference of **implicit arguments**
(both types and terms)

Challenge 1: Name binding

Any language needs *names* for variables and definitions, and *scopes* for determining the validity of these names.

Raw names are vulnerable to **name capture**:

$$(\lambda x y. x + y) y \rightsquigarrow \lambda y. \textcolor{red}{y} + y$$

Nameless representations (de Bruijn) are hard to read and require frequent **shifting**:

$$(\lambda. \lambda. @1 + @0) @0 \rightsquigarrow \lambda. @0 + @1$$

Name binding: state of the art

Modern systems use different approaches to name binding:

- *Locally nameless* representations use de Bruijn indices for **local** variables but names for **global** variables.
- *Higher-order abstract syntax (HOAS)* represents variables by meta-level functions: substitution is just application!
- *Scope graphs* are a data structure for representing **non-lexical binding** (e.g. module imports).

Challenge 2: Type conversion

In dependently typed languages, typechecking requires (partial) evaluation of expressions.

Example. To use $p : x \in ([\] ++ zs)$ at type $x \in zs$, we need to evaluate $[\] ++ zs \rightsquigarrow zs$.

Some systems also support more equalities:

- η law for functions: $f = \lambda x. f\ x$
- η law for tuples: $p = (\text{fst } p, \text{snd } p)$
- Definitional irrelevance: $p_1 = p_2 : \text{NonZero } 42$

Type conversion: state of the art

Normalisation-by-evaluation (NbE) computes (weak head) normal forms by interpretation terms as *semantic values* and then *quoting* them back into syntax.

To compare two terms at a given type, we:

0. Apply type-based rules if applicable (η , irrelevance)
 1. Compute their weak-head normal forms
 2. Compare the head symbols
 3. If they are equal, recursively compare subterms

Challenge 3: Implicit argument inference

For usability, the type checker should be able to *infer* arguments that are uniquely determined by typing.

Example. When checking that `here` : $x \in (x :: xs)$ has type $1 \in (1 :: 2 :: [])$, Agda can infer $x := 1$ and $xs := 2 :: []$.

Challenge 3: Implicit argument inference

For usability, the type checker should be able to *infer* arguments that are uniquely determined by typing.

Example. When checking that `here` : $x \in (x :: xs)$ has type $1 \in (1 :: 2 :: [])$, Agda can infer $x := 1$ and $xs := 2 :: []$.

Implicit arguments may also *depend* on local variables.

Example. When applying `cast` : $\{P : A \rightarrow \text{Type}\} \rightarrow P\ x \rightarrow P\ y$ to $v : \text{Vec}\ B\ n$, Agda can infer $A := \mathbb{N}$ and $P := \lambda x. \text{Vec}\ B$.

Implicit arguments: state of the art

- Represent implicit arguments as **contextual metavariables**: placeholders with a variable context
- Solve equations in the **pattern fragment** using Miller's algorithm for higher-order unification
- **Postpone** other equations until they fit the pattern fragment or no more progress can be made²

²Abel & Pientka: *Higher-Order Dynamic Pattern Unification for Dependent Types and Records* (TLCA 2011)

Outline

1. Three challenges
2. Two approaches
3. One vision

Building a language? There's a language for that!

From the early days³ people have built **tools** for building programming languages.

These tools often take the form of **meta-languages**, i.e. languages for modeling languages.

Two approaches that are still relevant today:

1. Language workbenches
2. Logical frameworks

³Johnson, *Yacc: Yet Another Compiler-Compiler* (Bell Labs, 1975)

Approach 1: Language workbenches

A **language workbench**⁴ is a collection of meta-languages and tools for specifying different aspects of a language.

- Syntax and parsing
- Typing and static analysis
- Program transformation and compilation
- Tooling and editor support

Examples. Spoofax, Rascal

⁴Martin Fowler: *Language Workbenches: The Killer-App for DSLs?* (2005)

Lambda binding in language workbenches

Statix/Spoofax. (type inference rule)

```
typeOf(s, Lam(x, A, e)) = Arr(A, B) :-  
  typeOf(s, A) == Type(),  
  s' = sPutType(s, x, A),  
  typeOf(s', e) == B.
```

Rascal. (type inference rule)

```
Ty infer(Lam(x, A, body), Env env) =  
  Arr(A, infer(body, env + (x : A)));
```

Approach 2: Logical frameworks

A **logical framework**⁵ is a minimal meta-language to specify logics and type systems.

- Higher-order abstract syntax
- Type-level computation
- Higher-order unification and/or rewriting
- Often accompanied by tools for doing metatheory

Examples. Twelf, ELPI, Dedukti

⁵Harper, Honsell, and Plotkin: *A Framework For Defining Logics* (LICS 1987)

Typing lambda binding in logical frameworks

Twelf. (extrinsic typing rule snippet)

```
of-lam : ({x:exp} of x A -> of (F x) B)  
        -> of (lam A F) (arr A B).
```

ELPI. (extrinsic typing rule snippet)

```
of (lam A F) (arr A B) :-  
  pi x\ of x A => of (F x) B.
```

Dedukti. (intrinsic typing rule)

```
symbol Lam :  $\Pi$  (A : Ty) (B : Ty),  
  (Tm A  $\rightarrow$  Tm B)  $\rightarrow$  Tm (Arr A B);
```

Standalone typechecking DSLs/libraries

Some other frameworks instead focus on providing good *tools* for implementing typecheckers:

- **Turnstile+** is a Racket library for building dependent type checkers⁶
- **TypOS** is an ‘operating system’ for implementing concurrent type checkers.⁷

⁶Chang et al: *Dependent type systems as macros* (POPL 2020)

⁷Allais et al: TypOS: An “Operating System” for Typechecking Actors (TYPES 2022)

Lambda-binding in other systems

Turnstile+. (type inference rule)

```
(define-tyrule ( $\lambda$  [ $x : A$ ]  $e$ )  $\gg$ 
  [ $\vdash A \gg A \Leftarrow \text{Type}$ ]
  [ $[x \gg x : A] \vdash e \gg e \Leftarrow B$ ]
  -----
  [ $\vdash (\lambda (x) e) \Rightarrow (A \rightarrow B)$ ])
```

TypOS. (typechecking actor)

```
check@p = p?ty. p?tm. case tm
{ ['Lam \x. body] ->
  'Type?S. 'Type?T.
  ( ty ~ ['Arr S T]
    | \x.
      ctxt { x -> S }.
      check@q. q!T. q!body. )
}
```

Outline

1. Three challenges
2. Two approaches
3. One vision

A Workbench for Fancy Type Systems?

Language workbenches provide parsing, compilation, editor services, incremental analysis, ...

Logical frameworks provide higher-order syntax, type-level evaluation/conversion rules, HO unification, ...

A Workbench for Fancy Type Systems?

Language workbenches provide parsing, compilation, editor services, incremental analysis, ...

Logical frameworks provide higher-order syntax, type-level evaluation/conversion rules, HO unification, ...

Why can't we have both?

Vision. A language workbench for both programming and verification, combining the strengths of language workbenches and logical frameworks.

Challenges in combining LF and workbenches

- How do we represent evaluation rules for typechecking and compilation uniformly?
- How do we combine non-lexical binding and higher-order unification?
- How do we ensure meta-theoretic properties (e.g. **confluence**)?
- How do we do this all efficiently, incrementally, and concurrently?
- How do we make interactive editing language-parametric?

Preliminary work 1: Extensible constraints

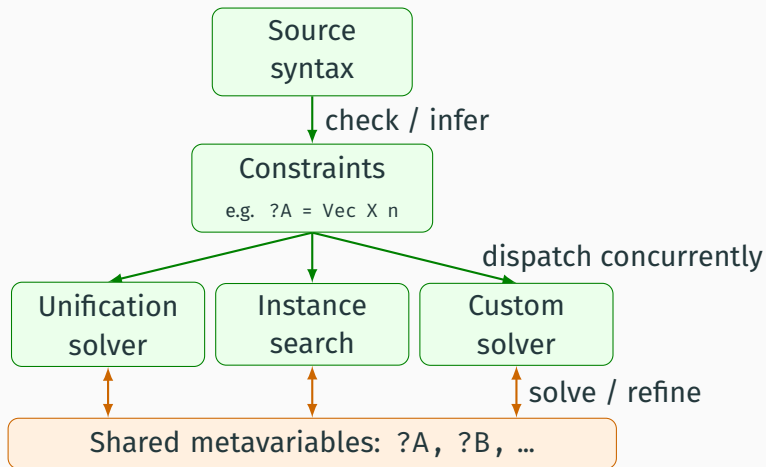
Modern dependently typed languages combine **bidirectional typechecking** and **constraint solving**.

We can make the typechecker modular:⁸

- Make the type of constraints **extensible**
- Dispatch constraints to **concurrent** solvers
- Use metavariables as **communication channels** between solvers

⁸Liesnikov and Cockx: *Building an Elaborator Using Extensible Constraints* (IFL 2023)

Extensible constraints: data flow



Preliminary work 2: Dependent types in Statix

Statix doesn't natively support type-level computation, but we can encode it with some additional predicates:⁹

- Implement type conversion `expectBetaEq(a, b)`:
 - Remember α -equivalence when going under binder:
`AlphaEqVars((x, y))`
 - Do β -reduction with a **Krivine machine**:
`betaReduceHead((s, e), ps)`
- Use conversion when checking function application:

```
typeOfExpr_(s, FnDestruct(fn, arg)) = ... :- {fn_arg_type}  
    betaReduceHead(typeOfExpr(s, fn)) == FnType(fn_arg_name, fn_arg_type, fn_body_type),  
    expectBetaEq(typeOfExpr(s, arg), fn_arg_type).
```

⁹Brouwer: *Dependently Typed Languages in Statix* (2023)

Dependent types in Statix: avoiding capture

Question. What is the type of $\lambda (T : \text{Type}). \lambda (T : T). T$?

A naive implementation infers $(T : \text{Type}) \rightarrow (T : T) \rightarrow T$, but this is wrong: the final T is not even a type!

Since Statix doesn't support generating fresh names, the implementation **tags** each variable *with its original scope*:

$$(T^{s1} : \text{Type}) \rightarrow (T^{s2} : T^{s1}) \rightarrow T^{s1}$$

Dependent types in Statix: problem solved?

This shows we can implement dependent types in Spoofax.
We can even encode higher-order unification!¹⁰

Problem solved?

¹⁰Tudor Andrei: *Higher-order unifications with scope graphs* (soon)

Dependent types in Statix: problem solved?

This shows we can implement dependent types in Spoofax.
We can even encode higher-order unification!¹⁰

Problem solved?

No: Encoding these features makes the specification
more verbose and less declarative.

They should instead be built-in like in the LF approach.

¹⁰Tudor Andrei: *Higher-order unifications with scope graphs* (soon)

Possible next steps

- Combine scope graphs, type conversion, and HOU
 - Extend Statix or another system?
 - Build a new logical framework?
- Use **multi-stage programming** to specialize the typechecking / elaboration code
- Add support for other algorithms such as SMT or congruence closure
- Build a new proof assistant with this approach (Agda 3?)

We don't yet know what new type systems we need,
but we know what **tools** we need to build them.

We don't yet know what new type systems we need,
but we know what **tools** we need to build them.

*Let's build these tools and see
1000 new type systems flourish!*