Extended Abstract: Towards a Performance Comparison of Syntax and Type-Directed NbE

Abstract

A key part of any dependent type-checker is the method for checking whether two types are equal. A common claim is that syntax-directed equality is more performant, although type-directed equality is more expressive. However, this claim is difficult to make precise, since implementations choose only one or the other approach, making a direct comparison impossible. We present some work-in-progress developing a realistic platform for direct, apples-to-apples, comparison of the two approaches, quantifying how much slower type-directed equality checking is, and analyzing why and how it can be improved.

1 Introduction

Dependently typed languages treat types as first class values, enabling computation in types. Consider the following.

```
int_or_str : (b : Bool) -> (if b then Int else String)
int_or_str = \b -> if b then 3141 else "Hello, World!"
```

When applied to `true`, it will return the `Int` 3141. When applied to `false`, it will return the `String` `"Hello, World!"`. We can even express such facts as types.

```
int_or_str_theorem : int_or_str(true) = 3141
int_or_str_theorem = reflexivity
```

Type-checking these requires deciding equality between two expressions. The typing judgement $\Gamma \vdash e : A$ (expression e has type A with free variables Γ), must reason about equality of expressions. In the true branch of `int_or_str` we make the judgement $\vdash 3141 :$ (`if true then Int else String`), and must reason that this type is equal to `Int`.

Equality is captured by judgements of the form $\Gamma \vdash e_1 = e_2 : A$ (e_1 is equal to e_2 at type A with free variables Γ). The two judgements interact through the following typing rule.

$$\textsc{Conv}\ \frac{\Gamma \vdash e : B \qquad \Gamma \vdash A = B : \texttt{Type}}{\Gamma \vdash e : A}$$

An expression e of type B can also be considered of type A so long as A and B are equal.

How to implement decision procedures for judgements of this form is an oft debated topic.

Our example relies on the following two rules.

$$\textsc{App-}\beta\ \frac{}{\Gamma \vdash (\backslash x \to b)a = b[x \mapsto a] : A}$$

$$\textsc{if-true-}\beta\ \frac{}{\Gamma \vdash \text{if true then } a \text{ else } b = a : A}$$

Both are β rules, which capture the execution of a program. App-β is the familiar λ-calculus β rule, where a function literal applied to an argument is equal to the body of the function with the argument substituted for the parameter variable. These rules are syntax-directed; we can see the type A is not relevant in either rule.

Another group of rules, called η rules, give us additional equalities that rely on the type, rather than capturing the execution of a program.

$$\textsc{Fun-}\eta\ \frac{\Gamma, x : A \vdash f\ x = g\ x : B}{\Gamma \vdash f = g : (x : A) \to B}$$

$$\textsc{Unit-}\eta\ \frac{}{\Gamma \vdash a = b : \text{Unit}}$$

The Fun-η rule looks similar to the function extensionality principle and says that two functions are definitionally equal if they are definitionally equal when applied to an abstract variable. The Unit-η rule implies the Unit type has the single element $\langle\rangle$. Since there is only one element of the Unit type, all expressions of Unit type are equal. This rule lets us type-check the following program.

```
unit_contractible : (x : Unit) -> (y : Unit) -> x = y
unit_contractible x y = reflexivity
```

As Abel [1] shows, a widely used approach for deciding judgemental equality is normalization by evaluation (NbE). An NbE algorithm maps the syntax of a language into a semantic domain and then back into syntax. Since judgementally equal pieces of syntax will be equal in the semantic domain, going there, then back into syntax makes them syntactically equal (a process referred to as "normalization"). NbE uses this process to turn the problem of deciding judgemental equality into one of giving semantics to syntax and then deciding syntactic equality.

Simple NbE algorithms implement an environment passing interpreter such as Coquand [4] or Chapman et al. [3]. These interpreters can have performance competitive with the widely used dependently typed languages, as Kovács [6] demonstrates with smalltt, a small but realistic, and very performant, type-checker for a dependent type theory.

However, another design decision remains: how to handle the type-directed η rules (if at all).

Claims abound that syntax-directed approaches are more performant than those that integrate type-directed rules. But how much more performant, in what situations, and is that performance worth the loss in expressivity? It could be the case that handling type-directed rules causes untenable performance loss, or requires unwieldy implementation, meaning theories which require them should be avoided. On the other hand, perhaps handling type-directed rules has only minimal performance cost and is quite simple to implement. The problem is that no apples-to-apples comparison between type-directed and syntax-directed approaches has been made, and so there is currently no good way to decide between them beyond consulting a developer who has tried both, in an unknown context, and applying their subjective judgement to your context. We would like to make these folklore claims more precise, and enable better informed choices.

2 Benchmarking Platform

Our approach is to compare two versions of smalltt [6]. The existing version uses a syntax-directed approach with η rules for functions.

We implement two modified versions of smalltt with type-directed rules, trying to stay true to the performance consideration of smalltt. The first merely transitions to a type-directed approach, while the second extends the type theory with η for dependent pairs (Σ) and Unit. Our two modified versions are available at https://github.com/ChesterJFGould/smalltt on the `master` and `sigma-unit` branches respectively.

To implement the type-directed algorithm, we follow the approach of Chapman et al. [3], who provide a systematic approach to derive an algorithm that supports type-directed rules by giving the type of the two normalized terms as an argument to the procedure deciding their equality. The procedure can then inspect the type when it comes across a situation in which a type-directed rule is applicable. This algorithm uses an approach similar to bidirectional typing [5] to reduce the amount of type information that is carried through the equality judgement. Chapman et al. [3] divide equality into two judgements: the check judgement $\Gamma \vdash n = n' \Leftarrow A$, which takes a type as an input and in which η rules can be implemented, and the synth judgement $\Gamma \vdash \nu = \nu' \Rightarrow A$ which checks two neutral forms for equality and additionally outputs their type. We give an excerpt of the type-directed equality rules in Figure 1. In these judgements, the metavariable n indicates terms that are normal with respect to β rules, while ν indicates a neutral term, that is, a series of destructors applied to a free variable.

In Figure 2, we present an overview of key definitions from smalltt that implement judgmental equality, and the changes we made to implement type-directed equality. We have simplified the code shown here slightly to elide meta variables and unification, which are irrelevant to our purposes, but the implementation supports these.

$$\Sigma\text{-}\eta\ \frac{\begin{array}{c}\text{fst } n \Downarrow n_f \qquad \text{fst } n' \Downarrow n'_f \\ \Gamma \vdash n_f = n'_f \Leftarrow n_1 \qquad e_2[x \mapsto n_f] \Downarrow n_2 \\ \text{snd } n \Downarrow n_s \qquad \text{snd } n' \Downarrow n'_s \qquad \Gamma \vdash n_s = n'_s \Leftarrow n_2\end{array}}{\Gamma \vdash n = n' \Leftarrow \Sigma x : n_1.\ e_2}$$

$$\Sigma\text{-T}{=}\ \frac{\begin{array}{c}\Gamma \vdash n_1 = n'_1 \Leftarrow \text{Type} \qquad \Gamma \vdash y \text{ fresh} \qquad e_2[x \mapsto y] \Downarrow n_2 \\ e'_2[x' \mapsto y] \Downarrow n'_2 \qquad \Gamma \vdash n_2 = n'_2 \Leftarrow \text{Type}\end{array}}{\Gamma \vdash \Sigma x : n_1.\ e_2 = \Sigma x' : n'_1.\ e'_2 \Leftarrow \text{Type}}$$

$$\Sigma\text{-E1}{=}\ \frac{\Gamma \vdash \nu = \nu' \Rightarrow \Sigma x : n_1.\ e_2}{\Gamma \vdash \text{fst } \nu = \text{fst } \nu' \Rightarrow n_1}$$

$$\Sigma\text{-E2}{=}\ \frac{\begin{array}{c}\Gamma \vdash \nu = \nu' \Rightarrow \Sigma x : n_1.\ e_2 \\ e_2[x \mapsto \text{fst } \nu] \Downarrow n_2\end{array}}{\Gamma \vdash \text{snd } \nu = \text{snd } \nu' \Rightarrow n_2}$$

$$\frac{}{\Gamma \vdash n = n' \Leftarrow \text{Unit}} \qquad \frac{}{\Gamma \vdash \text{Unit} = \text{Unit} \Leftarrow \text{Type}}$$

Figure 1. Unit and Dependent Pair Type Directed Equality

On the left, the `Tm` type corresponds to our expressions, `Val` to normal forms, and `Spine` to the neutral terms. The `Val` type uses de Bruijn levels to represent variables instead of names. For example, with de Bruijn levels, the term $\lambda x.\lambda y.x$ is represented as $\lambda.\lambda.0$. The `Env` type represents a group of substitutions, and so the `eval` function corresponds to a combination of the $e \Downarrow n$ relation and the $e[x \mapsto e']$ function. A `Closure` represents an `Env` applied to a term. Finally, `unify` and `unifySp` correspond to the syntax-directed equality judgment $\Gamma \vdash n = n'$, which elides types, but uses the current de Bruijn level to generate fresh variables instead of Γ. Both `unify` and `unifySp` return an **IO** () since they will throw an exception if the terms aren't equal or evaluate to **return** () if they are.

On the right, we present the modifications made to convert smalltt to implement type-directed equality. We only needed to change the type of `unifyChk` and `unifySp`. `unifyChk` now corresponds to the $\Gamma \vdash n = n' \Leftarrow A$ judgment, while `unifySp` now corresponds to the $\Gamma \vdash \nu = \nu' \Rightarrow A$ judgment. We also add the unit and dependent pair types to our type-directed version of smalltt.

3 Results and Conclusion

In Figure 3 we can see the results of the performance comparison between the original syntax-directed smalltt and our modified type-directed smalltt. Each benchmark was run ten times on an AMD Ryzen 9 3900x processor with 16G of memory. The results for each benchmark are

```
data Tm = LocalVar Ix        data Val = VLocalVar Lvl Spine
  | App Tm Tm Icit             | VLam NameIcit Closure
  | Lam NameIcit Tm            | VPi NameIcit VTy Closure
  | Pi NameIcit Ty Ty          | VU
  | U
                             type VTy = Val
type Ty = Tm
                             data Spine = SId
type Ix = Int                  | SApp Spine Val

type Lvl = Int               data Env = ENil
                               | EDef Env Val

data Closure = Closure Env Tm

eval :: Env → Tm →Val

unify :: Lvl → Val → Val → IO ()

unifySp :: Lvl → Spine → Spine → IO ()
```

```
data Tm = LocalVar Ix           data Val = VLocalVar Lvl Spine
  | App Tm Tm Icit                | VLam NameIcit Closure
  | Lam NameIcit Tm               | VPi NameIcit VTy Closure
  | Pi NameIcit Ty Ty             | VSigma NameIcit VTy Closure
  | Sigma NameIcit Ty Ty          | VSigmaI Val Val
  | SigmaI Tm Tm                  | VUnit
  | Fst Tm                        | VUnitI
  | Snd Tm                        | VU
  | Unit
  | UnitI                       data Spine = SId
  | U                             | SApp Spine Val
                                  | SFst Spine
type TypeCxt = Map Lvl VTy        | SSnd Spine

type Cxt = Cxt {lvl :: Lvl, localTypes :: TypeCxt}

unifyChk :: Cxt → Val → Val → VTy →IO ()

unifySp :: Cxt → VTy →Spine → Spine → IO VTy
```

Figure 2. syntax-directed smalltt, key definitions (left); type-directed smalltt, key definitions (right)

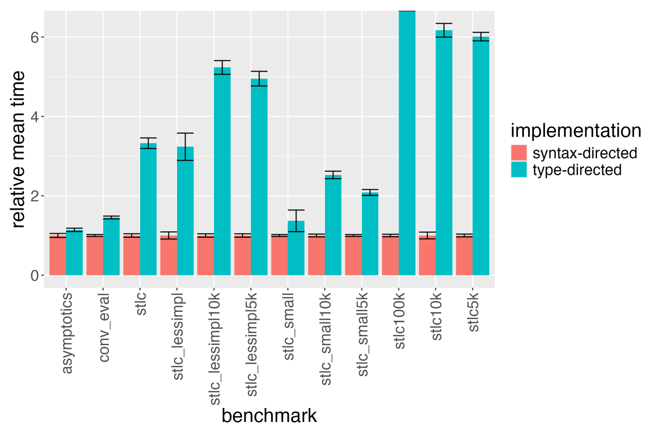

Figure 3. Benchmark Results

presented so that the times for the type-directed implementation are given as a multiple of the syntax-directed times, with the syntax-directed times normalized to have a mean of 1.

Overall, the type-directed implementation performs worse, being on average 3.4 times slower than the syntax-directed implementation (excluding the benchmarks which didn't finish). In the case of the stlc100k benchmark, the type-directed implementation failed to complete due to running out of memory. We represent this as a column which continues off the top of the graph to infinity. In comparison, benchmark numbers taken from equivalent benchmark suites show that Agda and Lean respectively perform 80 and 38 times slower on average than the syntax-directed implementation [6].

We conjecture that the main reason for the discrepancy between the syntax and type-directed implementations, is that the syntax-directed implementation is able to take greater advantange of glued evaluation. Glued evaluation exploits the fact that both substitution and evaluation respect judgmental equality to speed up the judgmental equality check. In other words, if $\Gamma, x : A \vdash e = e' : B$ then both $\Gamma \vdash e[x \mapsto d] = e'[x \mapsto d] : B$ and $\Gamma \vdash n = n' : B$ where $e \Downarrow n$ and $e' \Downarrow n'$. Using these facts, we can avoid normalizing or performing a substitution on terms we are checking for equality if they are already judgmentally equal.

Glued evaluation comes to a head when comparing terms whose types contain many redexes. Since we can only compare terms with the same type for equality, we must first compare the types of the terms, and glued evaluation lets us perform this comparison without computing many of the redexes. In the type-directed implementation, however, we still fully normalize the type of both terms, since we need to know if it is a type at which we can apply an η rule.

This conjecture leads us to the prediction that the performance difference between the syntax and type-directed implementations should correlate with the complexity of types used in the benchmark. Indeed, we find that the benchmark for which the syntax and type-directed implementations are closest is the asymptotics benchmark which contains very little type-level computation, while the difference for the stlc benchmarks, which use quite complicated types to encode the syntax of the simply typed λ-calculus, is much larger.

As future work, we would like to further investigate this conjecture, and if it proves to be true, investigate if we can improve the performance of the type-directed approach.

Finally, there are other procedures for deriving a type-directed algorithm, and we'd like to integrate these into our comparison. For example, in Kovács [7], the type of each term is calculated during normalization and then stored with the normalized term, so it can be inspected as needed during the equality check. This approach may have different performance characteristics, but only supports the type-directed rule for Unit, using a syntax-directed method for Σ. It's unclear whether either approach scales to other rules that require types, such as coproducts [2].

References

[1] Andreas Abel. 2013. *Normalization by Evaluation: Dependent Types and Impredicativity.* Habilitation thesis. Ludwig-Maximilians-Universität München. http://www.cse.chalmers.se/~abela/habil.pdf Accessed 2025-06-06.

[2] Thorsten Altenkirch, Peter Dybjer, Martin Hofmann, and Philip J. Scott. 2001. Normalization by evaluation for typed lambda calculus with coproducts. In *Symposium on Logic in Computer Science (LICS)*. 303–310. https://doi.org/10.1109/lics.2001.932506

[3] James Chapman, Thorsten Altenkirch, and Conor McBride. 2005. *Epigram Reloaded: A Standalone Typechecker for ETT.* Technical Report. https://people.cs.nott.ac.uk/psztxa/publ/checking.pdf Accessed 2025-06-06.

[4] Thierry Coquand. 1996. An algorithm for type-checking dependent types. *Science of Computer Programming* 26, 1–3 (May 1996), 167–177. https://doi.org/10.1016/0167-6423(95)00021-6

[5] Jana Dunfield and Neel Krishnaswami. 2021. Bidirectional Typing. *Comput. Surveys* 54, 5 (May 2021), 1–38. https://doi.org/10.1145/3450952

[6] Andras Kovács. 2023. https://github.com/AndrasKovacs/smalltt Accessed on 2025-06-06.

[7] Andras Kovács. 2025. https://github.com/AndrasKovacs/elaboration-zoo Accessed on 2025-06-09.