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-direct 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 function.

```
int_or_string : (b : Bool) -> (if b then Int else String)
int_or_string = \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_string_theorem : int_or_string(true) = 3141
int_or_string_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 execution (or at least equality) of programs. In the true branch of `int_or_string` we make the judgement $\vdash 3141 :$ (`if true then Int else String`), and must reason this type is equal `Int`.

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

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

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

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

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.

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

The Fun-η rule gives a function extensionality principle for terms of a function type. 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 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 algorithm for deciding judgemental equality is said to use NbE when it maps the syntax of a language into a semantic domain and then back into syntax. Since judgementally equal pieces of syntax will actually 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 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 (henceforth referred to as NbE interpreters) such as Coquand [3] or Chapman et al. [2]. These interpreters can have performance competitive with the widely used dependently typed languages, as Kovacs [4] demonstrates with smalltt.

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, and is that performance worth the loss in expressivity? Citations for such folklore claims?

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, one of the approaches to handling them described above could have minimal performance cost and be quite simple to implement. The problem is that no apples-to-apples comparison between them has been made, and so there is currently no good way to decide between them.

We would like to change that.

2 Benchmarking Platform

Our approach is to compare two versions of smalltt [4], a small but realistic, and very performant, type check for type theory. The existing version uses a syntax-directed approach without η rules.

We implement two modified version 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. [2], who provide systematic approach to deriving an algorithm which 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 a bi-directional-like approach to reduce the amount of type information that is carried through the equality judgement.

cite. Chester: I'm not sure that there is a citation here, the Chapman paper doesn't really analyze this connection Is this explanation clear? Maybe describe as an explicit

We give an excerpt of the type-directed equality rules in Figure 1. Following Chapman et al. [2], we have two equality 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 $\Gamma \vdash e = e' \Rightarrow n$ in which β rules are implemented and which additional output the normalized type.

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.

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 represented 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. Additionally, a Closure represents a Env applied to a term. Finally, unify and unifySp correspond to the syntax-directed equality judgment $\Gamma \vdash n_a = n_b$ judgement, which elides types, but uses the current de Bruijn level to generate fresh variables instead of Γ. Both unify and unifySp return an **IO** ()

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

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

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

$$\Sigma\text{-E2=}\ \frac{\Gamma \vdash v = v' \Rightarrow \Sigma x : n_1.\, e_2 \qquad e_2[x \mapsto \text{fst } v] \Downarrow n_2}{\Gamma \vdash \text{snd } v = \text{snd } v' \Rightarrow n_2}$$

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

Figure 1. Unit and Dependent Pair Type Directed Equality

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 correponds to the $\Gamma \vdash n_a = n_b \Leftarrow n_t$ judgment, while unifySp now correponds to the $\Gamma \vdash n_a = n_b \Rightarrow n_t$ judgment. We also add the unit and dependent pair types to our type-directed version of smalltt.

Anything else we want to say?

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 normalized so that the times for the typed-directed implementation are given as a multiple of the syntax-directed times.

Overall, the type-directed implementation performs worse, being on average 3.4 times slower than the syntax-directed implementation. 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.

We conjecture that the main reason for this discrepancy is that the syntax-directed implementation is able to take greater advantange of an optimization referred to as glued evaluation. Glued evaluation exploits two facts to speed up the judgmental equality check. Both that substitution and evaluation are functions with respect to judgmental equality, i.e. if $\Gamma, x : A \vdash e = e' : B$ then both $\Gamma \vdash e[x \mapsto d] = e'[x \mapsto$

```
data Tm = LocalVar Ix
  | App Tm Tm Icit
  | Lam NameIcit Tm
  | Pi  NameIcit Ty Ty
  | U

type Ty = Tm

type Ix = Int

type Lvl = Int

data Val = VLocalVar Lvl Spine
  | VLam NameIcit Closure
  | VPi NameIcit VTy Closure
  | VU

type VTy = Val

data Spine = SId
  | SApp Spine Val

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
  | App Tm Tm Icit
  | Lam NameIcit Tm
  | Pi  NameIcit Ty Ty
  | Sigma NameIcit Ty Ty
  | SigmaI Tm Tm
  | Fst Tm
  | Snd Tm
  | Unit
  | UnitI
  | U

type TypeCxt = Map Lvl VTy

data Val = VLocalVar Lvl Spine
  | VLam NameIcit Closure
  | VPi NameIcit VTy Closure
  | VSigma NameIcit VTy Closure
  | VSigmaI Val Val
  | VUnit
  | VUnitI
  | VU

data Spine = SId
  | SApp Spine Val
  | SFst Spine
  | 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-direct smalltt, key definitions (left); type-directed smalltt, key definitions (right)

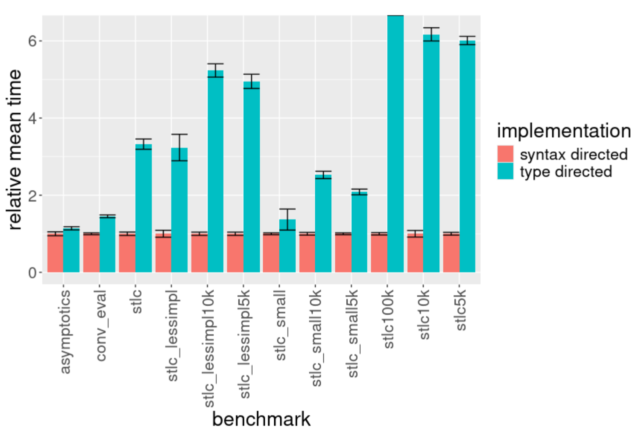

Figure 3. Benchmark Results

$d] : B$ and $\Gamma \vdash v = v' : B$ where $e \Downarrow n$ and $e' \Downarrow n'$. Using this fact, 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 lambda 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.

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] 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.

[3] 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

[4] Andras Kovacs. [n. d.]. https://github.com/AndrasKovacs/smalltt Accessed on 2025-06-06.