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

Abstract

Dependent type theories are a popular way of formulating proof assistants which allow a program to be developed alongside its proof of correctness. A key part of the type checker for these type theories is the method for checking if two types are the same. Historically, many different methods for performing this check have been proposed, but not many comparisons of them have been done. In this thesis we perform a performance evaluation of a common method against one which can handle more general equalities, and find that the common method outperforms the more general method.

ACM Reference Format:

. 2025. Extended Abstract: Towards a Performance Comparison of Syntax and Type Directed NbE. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

As Dijkstra [1] argues, formal methods form a critical part of computer programming. The digital nature of the computer means that even a slight error in a program results not in a slight error in output, but entirely unexpected output. Formal proof forces the programmer to reason through why he believes his program will do what he intends, and using a proof assistant forces this reasoning to be thorough enough that each step can be justified by some formal system.

The family of proof assistants under consideration of this thesis are those based on a dependent type theory (e.g. Agda, Coq, Lean). These systems have an advantage over others, as programs and proofs are written in a common language [2].

For any proof assistant, there is a problem akin to "Who watches the watchmen?", how does one know that the proof checker is itself correct? As a solution to this problem, Barendregt and Wiedijk [3] propose that proof assistants should admit the checking of a proof by a small, independent program (commonly referred to as the "de Bruijn criterion"). By their logic, we can trust a proof assistant whose proof checker is simple enough for us to understand, and which allows us to implement our own proof checker if we aren't convinced existing implementations are correct.

Simplicity isn't all there is to a proof checker, however. As Geuvers and Wiedijk [4] argues, if the proof checker takes longer than is reasonable to wait, not only is the proof assistant not much use, there can be situations wherein the status of the "proof" as a proof come into question. Therefore, the proof checker must be both simple and reasonably efficient (we take "reasonably efficient" here to mean "at least as fast as the others").

Accepting this, one might then ask if a dependently typed proof assistant can satisfy this criterion. To assess this, we must first know what it takes to check a proof in a dependent type theory, and even more basically, what a dependent type theory is.

1.1 Dependent Types

Just as functional languages treat functions as first class values, dependently typed languages treat types as first class values. An example is the following program,

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

Here the function `int_or_string` has domain `Bool`, but its codomain depends on which boolean it is passed. If it is passed the boolean `true`, then it will return an `Int`, specifically `3141`. Otherwise if it is passed `false`, then it will return the `String "Hello, World!"`.

As an example of the theorem proving capabilities of this language we could prove the theorem `int_or_string(true) = 3141`.

```
int_or_string_theorem : int_or_string(true) = 3141
int_or_string_theorem = reflexivity
```

Here we define a new variable `int_or_string_theorem` whose *type* is `int_or_string(true) = 3141`. Just like how function types are inhabited by functions, or the integer type is inhabited by integers, the equality types are inhabited by proof of the equality. Here the proof of equality we have defined `int_or_string_theorem` to be is simply reflexivity, meaning that both sides of the equality are already the same. This exhibits one of the more confusing aspects of dependent type theory, equality is not syntactic but up to symbolic execution. Two expressions or types are equal if they can be "run" to the same value.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

In fact, we had already seen this in the definition of `int_or_string`. One of the judgments of the formal system underlying dependent type theory is the typing judgment taking the form $\Gamma \vdash e : A$, meaning expression e has type A with free variables contained in Γ. So in the true branch of `int_or_string` we make the judgment $b : \texttt{Bool} \vdash 3141 : (\texttt{if true then Int else String})$, and since we are reasoning up to symbolic execution this is equal to to $b : \texttt{Bool} \vdash 3141 : \texttt{Int}$.

This "reasoning up to symbolic execution" is formalized using the second judgment of dependent type theory, the equality judgment. This takes the form $\Gamma \vdash a = b : A$, meaning a is judgmentally equal to b at type A with free variables Γ. The two judgments then interact with each other with the following deduction rule

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

Which states that an expression e of type B can also be considered of type A so long as A and B are judgmentally equal types.

We've already seen two instances of this equality judgment at play in the `int_or_string` and `int_or_string_theorem` example.

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

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

App-β is the familiar lambda 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. This rule was used when type checking `int_or_string_theorem` to judge that `int_or_string(true)` is equal to `if true then 3141 else "Hello, World!"`. The other rule, if-true-β, was then used to judge that `if true then 3141 else "Hello, World!"` is equal to `3141`. Since both these rules capture the execution of a program, we group them together and call them both β rules. We then call an expression which looks like the left-hand side of a β rule a "redex", for reducible expression.

There is, however, another group of rules called η rules. Instead of capturing the execution of a program, they give us additional equalities that we can glean from the type of a term.

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

The most familiar of these is the Fun-η rule, which gives us the function extensionality principle for terms of a function type. Another is the Unit-η rule for the Unit type which has the single element $\langle\rangle$. Since there is only one element of the Unit type, we can judge 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
```

The main part of this thesis will be comparing two systems, one of which supports this Unit-η rule and one which doesn't.

1.2 Implementation

So, the two main components of a proof checker for a dependent type theory are the algorithms for deciding the typing and equality judgments. Therefore if our proof assistant based on this theory is to satisfy the de Bruijn criterion, there must be simple, efficient algorithms for deciding these judgments.

When writing code which decides if a judgment is provable for the subjects of the judgment, there is a decision to be made about which subjects will be treated as input and which as output. For example, in a language with complete type inference (e.g. Haskell, OCaml), the type in the typing judgment is always treated as output while the context and expression are input.

1.2.1 Typing Judgement. The Pfenning bidirectional type checking recipe [5] provides a simple algorithm for deciding the typing judgment by carefully choosing when to switch between the input and output modes. This recipe splits the operators in the language into two camps, constructors which build an element of a specific type and whose typing rule takes the type as input, and destructors which perform some operation on elements of a specific type and whose typing rule produce their type as output. In the example code given above, `\b -> ...`, `3141`, and `"Hello, World!"` are constructors while the `if` operator is a destructor. Following this recipe makes the code deciding the typing judgment very regular, making it both simple to read and simple to write. Additionally, the algorithm performs a single traversal of the syntax tree, making it reasonably efficient.

1.2.2 Equality Judgement. The story for the equality judgment, however, is not so straightforward. As Abel [6] shows, the most widely used approach for deciding judgmental equality is normalization by evaluation (NbE). An algorithm for deciding judgmental equality is said to use NbE when it maps the syntax of a language into a semantic domain and then back into syntax. Since judgmentally 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 judgmental equality into one of giving semantics to syntax and then deciding syntactical equality.

NbE can be much simpler to implement when compared with other algorithms, which may require the notoriously tricky capture avoiding substitution, and it can also be faster

than other techniques. However, the "can"s here are important. As described above, NbE isn't specifically a single algorithm, but more of an approach an algorithm can take to decide an equivalence relation.

The simplest NbE algorithms are those based on an environment passing interpreter (henceforth referred to as NbE interpreters) such as Coquand [7] or Chapman et al. [8]. Comparatively, the implementations of other algorithms are either much larger in scale, such as the compiler into bytecode described in Grégoire and Leroy [9], or require the knowledge of very dense mathematics to understand such as Ahman and Staton [10]. In addition, these interpreters can have performance competitive with the most widely used dependently typed languages, as Andras Kovacs' *smalltt*[1] demonstrates. Therefore, NbE interpreters are the best candidate algorithms for deciding judgmental equality.

However, previous work on NbE interpreters follow an ad-hoc recipe for designing their algorithms. When adding new language constructs, it is unclear how to add accompanying equality rules. In contrast, bidirectional type checking provides a systematic recipe for adding typing rules for new language constructs by splitting them up into constructors and destructors. An algorithm for deciding judgmental equality should also provide a systematic recipe for extending the algorithm to language constructs beyond those considered in its original description.

1.2.3 Type-Directed Rules.

Another problem with NbE interpreters, is if and how they can handle "type-directed" rules.

When comparing two terms for judgmental equality, after the terms are normalized, most judgmental equality rules can be decided by simply comparing the syntax of the two terms. There are some rules, however, where it is necessary to inspect the type of the two terms to decide if they are equal. One example is the Unit-η rule shown at the end of subsection 1.1.

The algorithm described by Coquand [7] takes the easy way out and excludes these rules from the theory, resulting in a weaker, syntax-directed, judgmental equality. However, doing so puts the burden of proof on the user of the theory when a type-directed rule is required for a program to type check.

Chapman et al. [8] describe 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. Another approach is documented by Andras Kovacs in his *elaboration-zoo* [2]. There, 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.

Here we have three different approaches where the tradeoffs they make are unclear. 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 comparison between them has been made, and so there is currently no good way to decide between them.

1.3 Thesis Statement

We propose that the type-directed algorithm described by Chapman et al. [8] will perform as well as a purely syntax-directed approach while allowing for theories with a stronger judgmental equality. The rest of this thesis is split into two parts. First we define exactly what the syntax and type-directed approaches to judgmental equality are, and how an algorithm for deciding the type-directed equality can be derived from the bidirectional recipe. Then we present the results of a performance comparison between Kovac's smalltt, which uses a syntax-directed equality, and version of smalltt we modified to use a type-directed equality.

2 Main Ideas

Here we define the syntax and type-directed judgmental equalities, and also demonstrate the close connection between the type-directed equality rules and the bidirectional typing rules. We first present both the syntax and type-directed equalities for a small dependent type theory in which the two equalities differ only in presentation. Then, we present an extension to the theory with two new types. We add the unit type to show that the type-directed equality supports more rules than the syntax-directed equality. And we add dependent pairs to further demonstrate the connection between the equality and typing rules.

Our purpose here is to give the reader a rigorous understanding of what these two equalities are and how they are implemented in smalltt. Therefore, we present the theory in a form very close to that of both smalltt implementations, but we don't prove any theorems about this theory. That being said, the theory we present is very similar to that proposed in Altenkirch et al. [11].

2.1 Syntax and Semantics

Before defining the two kinds of judgmental equality, we must first define the theory whose terms we want to judge equal. The syntax is presented in Figure 1. It consists of expressions e which are either variables x, function introduction or elimination forms, dependent function types, the type universe, or a type annotation. We also define a subset

[1] https://github.com/AndrasKovacs/smalltt
[2] https://github.com/AndrasKovacs/elaboration-zoo

$$
\begin{array}{llll}
\text{Expressions} & e & ::= & x \\
& & | & \lambda x.\ e \\
& & | & e_f\ e_a \\
& & | & \Pi x : e_d.\ e_c \\
& & | & \text{Type} \\
& & | & e : e_t \\
\text{Normal Forms} & n & ::= & \lambda x.\ e \\
& & | & \Pi x : n_d.\ e_c \\
& & | & \text{Type} \\
& & | & v \\
\text{Neutral Terms} & v & ::= & x \\
& & | & v_f\ n_a \\
\text{Type Contexts} & \Gamma & ::= & \cdot \\
& & | & \Gamma, x : n
\end{array}
$$

Figure 1. Syntax

of expressions called normal forms. These are expressions which only contain redexes that are inside a binding form. We will use these to represent the types during type checking since we don't need to reduce them to check if they are type constructors. For example, if we used expressions to represent types, we could have $(\lambda x.\ \Pi y : \text{Unit}.\ \text{Unit})\ \langle\rangle$ which we would need to reduce to $\Pi y : \text{Unit}.\ \text{Unit}$ to know that it is a type constructor.

$$\boxed{e \Downarrow n}$$

$$\textsc{Var}\Downarrow\ \frac{}{x \Downarrow x} \qquad \Pi\text{-I}\Downarrow\ \frac{}{\lambda x.\ e \Downarrow \lambda x.\ e}$$

$$\beta\Downarrow\ \frac{e_f \Downarrow \lambda x.\ e_b \qquad e_b[x \mapsto e_a] \Downarrow n_b}{e_f\ e_a \Downarrow n_b}$$

$$\Pi\text{-E}\Downarrow\ \frac{e_f \Downarrow v_f \qquad e_a \Downarrow n_a}{e_f\ e_a \Downarrow v_f\ n_a}$$

$$\Pi\text{-T}\Downarrow\ \frac{e_d \Downarrow n_d}{\Pi x : n_d.\ e_c \Downarrow \Pi x : n_d.\ e_c} \qquad \textsc{Type-T}\Downarrow\ \frac{}{\text{Type} \Downarrow \text{Type}}$$

$$\textsc{Ann}\Downarrow\ \frac{e \Downarrow n}{e : e_t \Downarrow n}$$

Figure 2. Big Step Semantics

In Figure 2 we define the operational semantics for the theory. In addition to being a semantics for the theory, we can view these rules as a method for turning an expression into its corresponding normal form.

$$\boxed{\Gamma \vdash e \Leftarrow n_t} \qquad \boxed{\Gamma \vdash e \Rightarrow n_t}$$

$$\textsc{Var}\ \frac{(x : n) \in \Gamma}{\Gamma \vdash x \Rightarrow n}$$

$$\textsc{Conv}\ \frac{\Gamma \vdash e \Rightarrow n_t \qquad \Gamma \vdash n_t = n_s : \text{Type}}{\Gamma \vdash e \Leftarrow n_s}$$

$$\textsc{Ann}\ \frac{\Gamma \vdash e_t \Leftarrow \text{Type} \qquad e_t \Downarrow n_t \qquad \Gamma \vdash e \Leftarrow n_t}{\Gamma \vdash e : e_t \Rightarrow e_t}$$

$$\Pi\text{-I}\ \frac{\Gamma \vdash z \text{ fresh} \qquad e_c[y \mapsto z] \Downarrow n_c \qquad \Gamma, z : n_d \vdash e[x \mapsto z] \Leftarrow n_c}{\Gamma \vdash \lambda x.\ e \Leftarrow \Pi y : n_d.\ e_c}$$

$$\Pi\text{-E}\ \frac{\Gamma \vdash e_f \Rightarrow \Pi x : n_d.\ e_c \qquad \Gamma \vdash e_a \Leftarrow n_d \qquad e_c[x \mapsto e_a] \Downarrow n_c}{\Gamma \vdash e_f\ e_a \Rightarrow n_c}$$

$$\Pi\text{-T}\ \frac{\Gamma \vdash e_d \Leftarrow \text{Type} \qquad \Gamma, x : e_d \vdash e_c \Leftarrow \text{Type}}{\Gamma \vdash \Pi x : e_d.\ e_c \Leftarrow \text{Type}}$$

$$\textsc{Type-in-Type}\ \frac{}{\Gamma \vdash \text{Type} \Leftarrow \text{Type}}$$

Figure 3. Typing Rules

Finally, we present the typing rules for the theory in a bidirectional style [12]. The check judgment $\Gamma \vdash e \Leftarrow e_t$ means that the term e checks against the type n_t under the context Γ. It is formulated such that it can be decided when given concrete terms for Γ, e, and n_t. The synth judgment $\Gamma \vdash e \Rightarrow n_t$ means that the type n_t can be inferred for the term e under the context Γ. It is formulated such that given the concrete terms for Γ and e, it can be decided if there exists an n_t such that the judgment holds.

Because of the Type-in-Type rule our theory is inconsistent, but we still add it for two reasons. First of all, the problem of adding a stratified hierarchy of universes to make the theory consistent has been studied extensively elsewhere and is orthogonal to our concerns. Second of all, smalltt uses this rule and so adding it ensures our presentation of the theory remains as close to it as possible.

The Π-I rule uses the $\Gamma \vdash z$ fresh judgment to specify that the variable z must not appear in the context Γ. It's conventional to assert that variables are fresh in a context, and so we do not define this judgment here.

The Conv rule uses the typed equality judgment $\Gamma \vdash n_a = n_b : n_t$ which we deliberately haven't defined yet. In the next two subsections we will present two different versions of this

judgment: the syntax-directed version which can be decided by only considering Γ, n_a, and n_b, and the type-directed version which also needs to consider n_t.

2.2 Syntax Directed Equality

$$\text{Scope} \quad \Theta \quad ::= \quad \cdot \quad | \quad \Theta, x$$

$$\boxed{\Gamma \vdash n_a = n_b : n_t} \quad \boxed{\Theta \vdash n_a = n_b} \quad \text{strip-types} : \Gamma \to \Theta$$

$$\text{strip-types}(\cdot) = \cdot$$
$$\text{strip-types}(\Gamma, x : n_t) = \text{strip-types}(\Gamma), x$$

$$\textsc{Syn} \ \frac{\text{strip-types}(\Gamma) \vdash n_a = n_b}{\Gamma \vdash n_a = n_b : n_t}$$

$$\eta\text{-L} \ \frac{\Theta \vdash y \text{ fresh} \quad (\lambda x.\ e_b)\ y \Downarrow n_b \quad n\ y \Downarrow n_a \quad \Theta, y \vdash n_b = n_a}{\Theta \vdash \lambda x.\ e_b = n}$$

$$\eta\text{-R} \ \frac{\Theta \vdash y \text{ fresh} \quad (\lambda x.\ e_b)\ y \Downarrow n_b \quad n\ y \Downarrow n_a \quad \Theta, y \vdash n_b = n_a}{\Theta \vdash n = \lambda x.\ e_b}$$

$$\Pi= \ \frac{\Theta \vdash n_d = n'_d \quad \Theta \vdash y \text{ fresh} \quad e_c[x \mapsto y] \Downarrow n_c \quad e'_c[x' \mapsto y] \Downarrow n'_c \quad \Theta, y \vdash n_c = n'_c}{\Theta \vdash \Pi x : n_d.\ e_c = \Pi x' : n'_d.\ e'_c}$$

$$\textsc{Type}= \ \frac{}{\Theta \vdash \text{Type} = \text{Type}} \qquad \textsc{Var}= \ \frac{}{\Theta \vdash x = x}$$

$$\textsc{App}= \ \frac{\Theta \vdash v_f = v'_f \quad \Theta \vdash n_a = n'_a}{\Theta \vdash v_f\ n_a = v'_f\ n'_a}$$

Figure 4. Syntax Directed Equality

In Figure 4 we define the typed equality judgment by appealing to the syntax-directed equality judgment $\Theta \vdash n_a = n_b$. The π=, η-l, and η-r rules require adding fresh variables to the context. However, we don't know what type these new variables should be given in the η cases. Therefore we use Θ in the syntax-directed equality judgment instead of Γ, and use strip-types to turn a type context into a scope.

Since we are already given terms in normal form, we don't need to consider β equalities until we get to a binding form. In these cases, specifically the π=, η-l, and η-r rules, we use the big step semantics from Figure 2 to reduce all expressions to their normal form before continuing.

In the η-l and η-r rules we use a trick to decide the η rule for dependent function from only the syntax. In a normal form, there are only two cases which could be given a function type. Either the normal form is a function literal, or it is a neutral term. If both sides of the equality are neutral terms, then applying the η rule won't change anything. Therefore, we only have to apply the function η rule in cases where at least one side of the equality is a function literal, leading us to defining our two η rules.

2.3 Type Directed Equality

$$\boxed{\Gamma \vdash n_a = n_b : n_t} \quad \boxed{\Gamma \vdash n_a = n_b \Leftarrow n_t} \quad \boxed{\Gamma \vdash v_a = v_b \Rightarrow n_t}$$

$$\frac{\Gamma \vdash n_a = n_b \Leftarrow n_t}{\Gamma \vdash n_a = n_b : n_t}$$

$$\textsc{Fun}\text{-}\eta \ \frac{\Gamma \vdash y \text{ fresh} \quad n_a\ y \Downarrow n'_a \quad n_b\ y \Downarrow n'_b \quad e_c[x \mapsto y] \Downarrow n_c \quad \Gamma, y : n_d \vdash n'_a = n'_b \Leftarrow n_c}{\Gamma \vdash n_a = n_b \Leftarrow \Pi x : n_d.\ e_c}$$

$$\Pi= \ \frac{\Gamma \vdash n_d = n'_d \Leftarrow \text{Type} \quad \Gamma \vdash y \text{ fresh} \quad e_c[x \mapsto y] \Downarrow n_c \quad e'_c[x' \mapsto y] \Downarrow n'_c \quad \Gamma \vdash n_c = n'_c \Leftarrow \text{Type}}{\Gamma \vdash \Pi x : n_d.\ e_c = \Pi x' : n'_d.\ e'_c \Leftarrow \text{Type}}$$

$$\textsc{Type}= \ \frac{}{\Gamma \vdash \text{Type} = \text{Type} \Leftarrow \text{Type}}$$

$$\textsc{Conv}= \ \frac{\Gamma \vdash v_a = v_b \Rightarrow n'_t}{\Gamma \vdash v_a = v_b \Leftarrow n_t} \qquad \textsc{Var}= \ \frac{(x : n_t) \in \Gamma}{\Gamma \vdash x = x \Rightarrow n_t}$$

$$\textsc{App}= \ \frac{\Gamma \vdash v_f = v'_f \Rightarrow \Pi x : n_d.\ e_c \quad \Gamma \vdash n_a = n'_a \Leftarrow n_d \quad e_c[x \mapsto n_a] \Downarrow n_c}{\Gamma \vdash v_f\ n_a = v'_f\ n'_a \Rightarrow n_c}$$

Figure 5. Type Directed Equality

In Figure 5 we define the typed equality judgment in terms of the equality check judgment $\Gamma \vdash n_a = n_b \Leftarrow n_t$ and the equality synth judgment $\Gamma \vdash n_a = n_b \Rightarrow n_t$. The structure of these two judgments closely follows the structure of the synth and check typing judgments from Figure 3. Like the type check judgment, the equality check judgment can be decided when all of Γ, n_a, n_b, and n_t are provided. And similarily, given Γ, n_a, and n_b, it can be decided if there exists an n_t such that the equality synth judgment holds.

We use two tricks to decide these rules. Firsly, we always start comparing normal forms using the equality check judgment. This way we always have the type we are comparing at available, so we can apply η rules based on it. Secondly, the places in which normal forms appear in neutral terms (e.g. the argument of a function application) correspond to places which use the check typing judgment during type checking. Therefore, we can infer the type of these neutral

terms and then apply the equality check judgment to them, as required by our first trick.

In fact, most of the type-directed equality rules follow the same structure as their corresponding bidirectional typing rules. For example, the App= rule corresponds to the Π-E typing rule from Figure 3 and follows the same structure where synth judgment is applied to the function, and then the check judgment is applied to the argument. However, the Fun-η rule differs from this pattern. Instead of following the structure of the Π-I typing rule, it uses the function extensionality principle to convert the problem of equality at the dependent function type into that of equality at the codomain type.

Therefore, we can reuse the Pfenning bidirectional typing recipe [12] to derive the type-directed equality rules. The only change we make is to replace the introduction rules for types with an η rule we want to decide with the η rule.

2.4 Unit and Dependent Pair Types

Up until this point, the syntax and type-directed equalities have really been the same equality but presented in two different ways. This is because the η rule for dependent functions can be decided both by the syntax and type-directed equalities. However, once we add the Unit type this changes, since the η rule for the unit type is not decidable by the syntax-directed equality. Additionally, we add the dependent pair type to further illustrate how the type-directed equality rules closely follow from to the bidirectional typing rules.

$$
\begin{array}{llll}
\text{Expressions} & e & ::= & \ldots \\
 & & \mid & \langle e_f, e_s \rangle \\
 & & \mid & \text{fst } e \\
 & & \mid & \text{snd } e \\
 & & \mid & \Sigma x : e_f.\ e_s \\
 & & \mid & \langle\rangle \\
 & & \mid & \text{Unit} \\
\\
\text{Normal Forms} & n & ::= & \ldots \\
 & & \mid & \langle n_f, n_s \rangle \\
 & & \mid & \Sigma x : n_f.\ e_s \\
 & & \mid & \langle\rangle \\
 & & \mid & \text{Unit} \\
\\
\text{Neutral Terms} & v & ::= & \ldots \\
 & & \mid & \text{fst } v \\
 & & \mid & \text{snd } v
\end{array}
$$

Figure 6. Unit and Dependent Pair Syntax

In Figure 6 we present the extension of the syntax with the unit and dependent pair types. The Unit type has type constructor Unit, and introduction form $\langle\rangle$, while the dependent pair type has type former $\Sigma x : e_f.\ e_s$, introduction form $\langle e_f, e_s \rangle$, and elimination forms fst e and snd e.

$$
\Sigma\text{-I}\Downarrow \frac{e_f \Downarrow n_f \qquad e_s \Downarrow n_s}{\langle e_f, e_s \rangle \Downarrow \langle n_f, n_s \rangle} \qquad \sigma_1 \frac{e \Downarrow \langle n_f, n_s \rangle}{\text{fst } e \Downarrow n_f}
$$

$$
\sigma_2 \frac{e \Downarrow \langle n_f, n_s \rangle}{\text{snd } e \Downarrow n_s} \qquad \sigma\text{-I1}\Downarrow \frac{e \Downarrow v}{\text{fst } e \Downarrow \text{fst } v}
$$

$$
\sigma\text{-I2}\Downarrow \frac{e \Downarrow v}{\text{snd } e \Downarrow \text{snd } v} \qquad \sigma\text{-T}\Downarrow \frac{e_1 \Downarrow n_1}{\Sigma x : e_1.\ e_2 \Downarrow \Sigma x : n_1.\ e_2}
$$

$$
\textsc{Unit-I}\Downarrow \frac{}{\langle\rangle \Downarrow \langle\rangle} \qquad \textsc{Unit-T}\Downarrow \frac{}{\text{Unit} \Downarrow \text{Unit}}
$$

Figure 7. Unit and Dependent Pair Semantics

The semantics presented for the unit and dependent pairs in Figure 7 is completely standard, as are the typing rules presented in Figure 8.

$$
\Sigma\text{-I}\ \frac{\Gamma \vdash e_f \Leftarrow n_1 \qquad e_2[x \mapsto e_s] \Downarrow n_2 \qquad \Gamma \vdash e_s \Leftarrow n_2}{\Gamma \vdash \langle e_f, e_s \rangle \Leftarrow \Sigma x : n_1.\ e_2}
$$

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

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

$$
\Sigma\text{-T}\ \frac{\Gamma \vdash e_1 \Leftarrow \text{Type} \qquad \begin{array}{c} e_1 \Downarrow n_1 \qquad \Gamma, x : n_1 \vdash e_2 \Leftarrow \text{Type} \end{array}}{\Gamma \vdash \Sigma x : e_1.\ e_2 \Leftarrow \text{Type}}
$$

$$
\textsc{Unit-I}\ \frac{}{\Gamma \vdash \langle\rangle \Leftarrow \text{Unit}} \qquad \textsc{Unit-T}\ \frac{}{\Gamma \vdash \text{Unit} \Leftarrow \text{Type}}
$$

Figure 8. Unit and Dependent Pair Typing

In Figure 9 we present the syntax-directed equality for the unit and dependent pairs. We use a trick to decide the η rule for dependent pairs similar to that used to syntactially decide the η rule for dependent functions. However, the same trick doesn't work for the unit η rule. In the case of dependent functions and pairs, if the terms being compare were both neutral, then applying the corresponding η rule doesn't make a difference. But, for the unit type it does make a difference, since we can judge any two neutrals equal using the unit η rule. Therefore, in general, we have to have the type of the

$$\Sigma\text{-}\eta\text{-L}\ \frac{\text{fst}\ n \Downarrow n'_f \qquad \text{snd}\ n \Downarrow n'_s \qquad \Theta \vdash n_f = n'_f \qquad \Theta \vdash n_s = n'_s}{\Theta \vdash \langle n_f, n_s \rangle = n}$$

$$\Sigma\text{-}\eta\text{-R}\ \frac{\text{fst}\ n \Downarrow n'_f \qquad \text{snd}\ n \Downarrow n'_s \qquad \Theta \vdash n_f = n'_f \qquad \Theta \vdash n_s = n'_s}{\Theta \vdash n = \langle n_f, n_s \rangle}$$

$$\Sigma\text{-T=}\ \frac{\Theta \vdash n_1 = n'_1 \qquad \Theta \vdash y\ \text{fresh} \qquad e_2[x \mapsto y] \Downarrow n_2 \qquad e'_2[x' \mapsto y] \Downarrow n'_2 \qquad \Theta \vdash n_2 = n'_2}{\Theta \vdash \Sigma x : n_1.\ e_2 = \Sigma x' : n'_1.\ e'_2}$$

$$\Sigma\text{-E1=}\ \frac{\Theta \vdash v = v'}{\Theta \vdash \text{fst}\ v = \text{fst}\ v'} \qquad \Sigma\text{-E2=}\ \frac{\Theta \vdash v = v'}{\Theta \vdash \text{snd}\ v = \text{snd}\ v'}$$

$$\textsc{Unit-I=}\ \frac{}{\Theta \vdash \langle\rangle = \langle\rangle} \qquad \textsc{Unit-T=}\ \frac{}{\Theta \vdash \text{Unit} = \text{Unit}}$$

Figure 9. Unit and Dependent Pair Syntax Directed Equality

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

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

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

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

$$\textsc{Unit-}\eta\ \frac{}{\Gamma \vdash n = n' \Leftarrow \text{Unit}}$$

$$\textsc{Unit-T=}\ \frac{}{\Gamma \vdash \text{Unit} = \text{Unit} \Leftarrow \text{Type}}$$

Figure 10. Unit and Dependent Pair Type Directed Equality

two terms available to know when we should apply the unit η rule.

Finally, in Figure 10 we present the type-directed equality rules for the unit and dependent pair types. Here we decide the unit and dependent pair η rules in much the same way as we do for the dependent function type. If the type we are comparing at is a unit or dependent pair, simply apply the corresponding η and continue.

3 Methods

3.1 *smalltt* Implementation

To evaluate the performance and utility of the recipe for type-directed judgmental equality, we have produced two modified versions of Andras Kovac's smalltt. The first version has the same syntax and semantics as the original smalltt, but uses a type-directed judgmental equality. The goal of this version is to evaluate the performance of the recipe as compared to the original smalltt type checker. The second version extends the first modification with the unit and dependent pair types. The primary goal of this version is to demonstrate that the type-directed equality can decide the unit η rule. Secondarily, it demonstrates the ease of adding new constructs to the language by following the bidirectional recipe.

The language in the original version of smalltt is almost identical to that presented in Figure 1. It only differs from the theory considered here in two ways. Firstly, it includes metavariables, essentially holes in a term which are inferred via unification during type checking. Secondly, it uses an optimization while deciding judgmental equality to try and prematurely decide that the two terms are equal. This optimization exploits the fact that judgmental equality is a congruence relation with respect to substitution. That is to say that if $\Gamma \vdash n_a = n_b : n_t$, then we can infer that $\Gamma \vdash n'_a = n'_b : n_t$ where $n_a[x \mapsto e] \Downarrow n'_a$ and $n_b[x \mapsto e] \Downarrow n'_b$. Therefore, if we can detect scenarios where the same substitution is being applied to both side of an equality judgment, we can first check if the terms are equal pre-substitution. This optimization is of course predicated on the assumption that n'_a and n'_b will usually be substantially larger than n_a and n_b. Neither of these should affect our performance comparison since they are present in both the original and our modified versions of smalltt.

The original version of smalltt is available at https://github.com/AndrasKovacs/smalltt/, while our two modified versions are available at https://github.com/ChesterJFGould/smalltt on the master and sigma-unit branches respectively.

3.1.1 Modifying *smalltt*.

Here we will present an overview of the part of the original smalltt which decides judgmental equality, and then present the overall changes we made to use a type-directed equality. Since inferring values for meta variables is irrelevant to the core of this thesis, we have simplified the code shown here slightly to remove most of those parts. The core data types and functions from the original smalltt are as follows.

```
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 Closure = Closure Env Tm

data Env =
    ENil
  | EDef Env Val

eval :: Env → Tm →Val

unify :: Lvl → Val → Val → IO ()

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

Here the Tm type corresponds to our expressions, Val to normal forms, and Spine to the neutral terms. Note that the Val type uses deBruijn levels to represent variables instead of names. For example, with deBruijn levels the term $\lambda x.\lambda y.x$ would be 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 representents a Env Finally, unify and unifySp correspond to the $\Theta \vdash n_a = n_b$ judgment, but they use the current deBruijn level to generate fresh variables instead of Θ. 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.

Next we present the modifications made to convert smalltt to using a type-directed equality.

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

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

type TypeCxt = Map Lvl VTy

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

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.

Finally we present the modifications made to add the unit and dependent pair types to our type-directed version of smalltt.

```
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

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
```

Here we didn't need to change the type of any functions, just add the new cases to the existing datatypes and functions.

3.2 Data Analysis

3.2.1 The Problem. Programs don't have deterministic runtimes, but their runtimes aren't completely random either. We can model each run of a program as being drawn

from some probability distribution, and so benchmarking a program can be viewed as sampling that program's runtime distribution.

In the case considered here, where we want to know if the runtime of the modified type checker performs as well as the original, the question becomes "is the mean of the runtime distribution of the modified type checker greater than that of the original program?". To answer this question, we can only use the runtime sample for each implementation collected during benchmarking. But, this begs the question, how can we use these samples to answer the question? And how large should the samples be?

3.2.2 Welch's T-Test. There are a few different methods we could use to compare the runtime distribution means of each implementation, the most popular probably being Student's t-test. This test would let us estimate the probability that we get the observed benchmarking results under the assumption that the runtime distribution mean of the modified type checker *isn't* larger that the original. If the probability is low, then we can say that it is likely that the modified type checker is slower than the original.

However, Student's t-test also assumes both that the variance in distribution of sample means for both type checkers are the same, and that they are normally distributed. The later is not an unreasonable assumption. The central limit theorem tells us that the distribution of sample means approaches a normal distribution as sample size increases. However, we have no reason to assume that the variances are equal, and so the first assumption is unreasonable. Therefore we use a generalization of Student's t-test called Welch's t-test which does not make this assumption [13].

3.2.3 Sample Size, Power, and Effect Size. To determine what sample size to use, we need to decide on how powerful to make our statistical test. In this case, the power is the probability of deciding that the modified type checker is slower under the assumption that it actually is. This also determines the probability of deciding that the modified type checker isn't slower even when it is, a situation we would like to avoid. Therefore we set the power far higher than the standard 0.8, setting it at 0.99, giving ourselves a 1% chance of a false positive. We also set the significance level, the probability of deciding that the modified type checker is slower when it isn't, to the similar value of 0.01. Additionally we set the minimum detectable effect size to 0.5, meaning that the runtime distributions must differ by at least 0.5 standard deviations for us to detect a difference between them.

To achieve these values, the G*Power tool [14] calculates that we need to run each benchmark 175 times per implementation.

3.2.4 Method. To collect the samples, we ran each smalltt benchmark 175 times using the hyperfine [3] tool. These benchmarks were run on a Intel Skylake Xeon running at 2.5GHz with 120G of ram. We then analyzed the collected data using the statistics Haskell library [4]. Our raw benchmark results and the code we used to analyze them are available at https://github.com/ChesterJFGould/HonoursThesis.

4 Results and Conclusion

Benchmark	P-Value
asymptotics	1.5×10^{-161}
conv_eval	9.9×10^{-261}
stlc	8.2×10^{-309}
stlc5k	3.6×10^{-305}
stlc10k	DNF
stlc100k	DNF
stlc_lessimpl	4.1×10^{-311}
stlc_lessimpl5k	0
stlc_lessimpl10k	DNF
stlc_small	0
stlc_small10k	2.6×10^{-284}
stlc_small5k	1.8×10^{-301}

Figure 11. Benchmark Results

In Figure 11 we can see the results of the performance comparison between the original syntax-directed smalltt and our modified type-directed smalltt. The p-value is the probability that we got the observed benchmark results under the assumption that our modified version smalltt isn't slower than the original. Across the board this probability is very small, and so it seems likely that our modified version of smalltt is slower than the original. That being said, a few cases jump out in this table.

In the stlc_lessimpl5k and stlc_small benchmarks, the p-value is 0. We interpret this as a p-value so small that it can't be represented as a double-precision floating point number, since these are what we used to do the statistical test.

Our modified version of smalltt did not finish the stlc10k, stlc100k, and stlc_lessimpl10k benchmarks due to running out of memory. That our implementation failed to complete some of the benchmarks isn't entirely surprising. smalltt includes equivalent benchmarks for some other common theorem provers such as Agda, Coq, and Idris2. In the performance comparison given in the README for smalltt, both Agda and Idris2 also failed to finish on the stlc100k.

This, however, doesn't detract from the fact that the original smalltt is clearly faster than our implementation. Therefore, we can reject our thesis statement and say that while the

[3] https://github.com/sharkdp/hyperfine
[4] https://hackage.haskell.org/package/statistics

type-directed equality supports more rules than the syntax-directed equality, the syntax-directed equality performs better.

5 Future Work

After conducting this research some questions remain.

5.1 Is the unit η rule the only rule which requires a type-directed equality?

In our presented theory, the only judgmental equality rule which required the type-directed approach was the unit η rule. Altenkirch et al. [15] present an algorithm for deciding the η rule for the coproduct type. However, it's unclear if their algorithm would fit with the type-directed approach we present here. Gilbert et al. [16] present a type theory with a universe of definitionally irrelevant propositions. While similar to the unit η rule, their η rule for propositions requires not only the type of the two terms be known, but also the type of the type.

5.2 Could our implementation be optimized?

While our smalltt implementation as is failed to perform as well as the syntax-directed smalltt, it's not out of the question that it could be optimized further. We leave this optimization as future work.

5.3 Was our method of correcting for variation in performance due to memory layout effective?

Previous work has shown that a program's memory layout can drastically affect performance, with even small things like the values of environment variables affecting this layout. We attempted to correct for this in our statistical test, but it's unclear if we did this properly. Properly developing statistical methods to solve this problem would be very useful when running any sort of performance comparison between programs.

References

[1] Dijkstra, E. W. On the Cruelty of Really Teaching Computer Science. **1988**,

[2] Nordstrom, B.; Petersson, K.; Smith, J. M. *Programming in Martin-Löf's type theory : an introduction*; International Series of Monographs on Computer Science; Clarendon Press: Oxford, England, 1990.

[3] Barendregt, H.; Wiedijk, F. The challenge of computer mathematics. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* **2005**, *363*, 2351–2375.

[4] Geuvers, H.; Wiedijk, F. A Logical Framework with Explicit Conversions. LFM@IJCAR. 2008.

[5] Dunfield, J.; Krishnaswami, N. Bidirectional Typing. *ACM Computing Surveys* **2021**, *54*, 1–38.

[6] Abel, A. Normalization by Evaluation: Dependent Types and Impredicativity. Habilitation thesis, Ludwig-Maximilians-Universität München, 2013.

[7] Coquand, T. An algorithm for type-checking dependent types. *Science of Computer Programming* **1996**, *26*, 167–177.

[8] Chapman, J.; Altenkirch, T.; McBride, C. Epigram reloaded: a standalone typechecker for ETT. Symposium on Trends in Functional Programming. 2005.

[9] Grégoire, B.; Leroy, X. A compiled implementation of strong reduction. ACM SIGPLAN International Conference on Functional Programming. 2002.

[10] Ahman, D.; Staton, S. Normalization by Evaluation and Algebraic Effects. Mathematical Foundations of Programming Semantics. 2013.

[11] Altenkirch, T.; Danielsson, N. A.; Löh, A.; Oury, N. PiSigma: Dependent Types without the Sugar. Fuji International Symposium on Functional and Logic Programming. 2010.

[12] Dunfield, J.; Krishnaswami, N. R. Bidirectional Typing. *ACM Computing Surveys (CSUR)* **2019**, *54*, 1 – 38.

[13] Welch, B. L. The generalisation of student's problems when several different population variances are involved. *Biometrika* **1947**, *34 1-2*, 28–35.

[14] Faul, F.; Erdfelder, E.; Buchner, A.; Lang, A.-G. Statistical power analyses using G*Power 3.1: Tests for correlation and regression analyses. *Behavior Research Methods* **2009**, *41*, 1149–1160.

[15] Altenkirch, T.; Dybjer, P.; Hofmann, M.; Scott, P. J. Normalization by evaluation for typed lambda calculus with coproducts. *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science* **2001**, 303–310.

[16] Gilbert, G.; Cockx, J.; Sozeau, M.; Tabareau, N. Definitional proof-irrelevance without K. *Proceedings of the ACM on Programming Languages* **2019**, *3*, 1 – 28.