Imperial College London

MEng Individual Project (Interim Report)

Department of Computing

Imperial College of Science, Technology and Medicine

Local Rewriting in Dependent Type Theory

Author:
Nathaniel Burke

Supervisor:
Dr. Steffen van Bakel

January 23, 2025

Submitted in partial fulfillment of the requirements for the MEng Computing of Imperial College London

Contents

1 Introduction and Motivation

Dependent pattern matching is the common extension of pattern matching to dependently-typed programming languages [1, 2], where, on top of acting as syntax sugar for eliminating inductively-defined types, in the bodies of matches, each matched-on variable ("scrutinee") is substituted for the corresponding pattern everywhere in the typing context.

It is this substituting that enables taking advantage of information learned over the course of a match. This allows, for example, defining equality testing functions that return actual evidence for the result of the test.

Example 1.0.1 (Boolean Equality Testing)

```
test : Π (b : 𝔹) → (b = true) + (b = false)
test true  ≡ inl refl
test false ≡ inr refl
```

Where $\_=\_$ *is the identity type introduced with reflexivity* refl $: x = x$, $A + B$ *is a sum type, introduced with injections* inl $: A \to A + B$ *and* inr $: B \to A + B$ *and* $\mathbb{B}$ *is the type of booleans introduced with literals* true $: \mathbb{B}$ *and* false $: \mathbb{B}$.

Note that in the test true branch, for example, the substitutions true / b and true / b are applied to the context, refining the goal type to (true = true) + (true = false), at which inl refl typechecks successfully.

In mainstream functional programming languages, it is common to allow pattern matching not just on the direct arguments of functions, but also on intermediary expressions (e.g. via a case_of_ construct). Extending dependent pattern-matching accordingly would have direct utility: consider this concise proof that for any boolean function f : 𝔹 → 𝔹, f b = f (f (f b)): .

Example 1.0.2 (f b = f (f (f b)), Concisely)

```
bool-lemma : Π (f : 𝔹 → 𝔹) (b : 𝔹) → f b = f (f (f b))
bool-lemma f true  ≡ case f true of
  true  → refl
  false → case f false of
    true  → refl
    false → refl
bool-lemma f false ≡ case f false of
  true  → case f true
    true  → refl
    false → refl
  false → refl
```

Unfortunately, mainstream proof assistants (such as Agda) generally do not support such a construct[1] , and we are instead forced to do the equational reasoning manually:

[1]: Coquand (1992), *Pattern matching with dependent types*
[2]: Cockx (2017), *Dependent pattern matching and proof-relevant unification*

In this report, our code snippets will generally follow Agda [3] syntax (in fact, this entire report is written in literate Agda, available at `https://github.com/NathanielB123/fyp`), but we take some liberties when typesetting. For example, writing ∀s as Πs and swapping the equality symbols $\_=\_$ and $\_\equiv\_$ to align with their conventional meanings in on-paper type theory.

[3]: Norell (2007), *Towards a practical programming language based on dependent type theory*

This example is originally from the Agda mailing list [4].

[4]: Altenkirch (2009), *Smart Case [Re: [Agda] A puzzle with "with"]*

1: Some proof assistants do allow matching on expressions to some extent via "**with**-abstractions" [5, 6]. We will cover why this feature is not quite satisfactory (including for this example) in Section 2.1.1.

[5]: McBride et al. (2004), *The view from the left*
[6]: Agda Team (2024), *With-Abstraction*

Example 1.0.3 (f b = f (f (f b)), Manually)

```
bool-lemma' : Π (f : 𝔹 → 𝔹) b
              → f true  = true + f true  = false
              → f false = true + f false = false
              → f b = f (f (f b))
bool-lemma' f true (inl p) q      ≔
  f true
   = by ⟨ sym (cong f p) ⟩
  f (f true)
   = by ⟨ sym (cong (f ∘ f) p) ⟩
  f (f (f true)) ■
bool-lemma' f true (inr p) (inl q) ≔
  f true
   = by ⟨ sym (cong f q) ⟩
  f (f false)
   = by ⟨ sym (cong (f ∘ f) p) ⟩
  f (f (f true)) ■
bool-lemma' f true (inr p) (inr q) ≔
  f true
   = by ⟨ p ⟩
  false
   = by ⟨ sym q ⟩
  f false
   = by ⟨ sym (cong f q) ⟩
  f (f false)
   = by ⟨ sym (cong (f ∘ f) p) ⟩
  f (f (f true)) ■
-- etc... (the 'false' cases are very similar)


bool-lemma : Π (f : 𝔹 → 𝔹) b → f b = f (f (f b))
bool-lemma f b ≔ bool-lemma' f b (test (f true)) (test (f false))
```

The trickyness with supporting matching on intermediary expressions is that there may not exist a unique unification solution between the scrutinee and pattern. Dependent pattern matching must modify the typing context in the branch of a match to make the scrutinee and pattern equal, but we cannot make f b equal true on-the-nose by substituting individual variables (without making any assumptions about the behaviour of f on closed booleans).

This project explores type theories with local, ground[2] , equational assumptions: a setting designed to enable this extended version of pattern matching. The idea is not novel, Altenkirch et al. first investigated such a theory during the development of ΠΣ [8, 9], naming this extended pattern-matching construct "Smart Case" [10]. However, this work was never published, ΠΣ eventually moved away from Smart Case, and both completeness and decidability (among other metatheoretical properties) remain open.

The full benefits of Smart Case are perhaps non-obvious. To motivate this work further, we will next elaborate on the small jump from extending pattern-matching in this way to a manual version of the equality reflection rule from Extensional Type Theory (ETT). Such a feature has potential to simplify a huge number of equational proofs written in modern Intensional Type Theory (ITT) based proof assistants.

2: A "ground" equation is one which is not quantified over any variables.

[8]: Altenkirch et al. (2008), *nΣ: A Core Language for Dependently Typed Programming*

[9]: Altenkirch et al. (2010), *ΠΣ: Dependent types without the sugar*

[10]: Altenkirch (2011), *The case of the smart case - How to implement conditional convertibility?*

1.1 Equality in Type Theory and Indexed Pattern Matching

Before proceeding, it is important to clarify our understanding of equality in type theory.

Remark 1.1.1 (Definitional vs Propositional Equality)
In ITT, the foundation of modern proof assistants/dependently typed programming languages Agda [3], Rocq [15], Lean [16], Idris2 [17], equality judgements are split into definitional (denoted with $\_ \equiv \_$) and propositional (denoted with $\_ = \_$).
Definitional equality (also called "conversion") judgements are made the meta-level, and typing relations in ITT are given with types always equated up to convertibility. Conversion is usually comprised of syntactic equality plus computation rules (β and sometimes η) but there are other options, which we shall examine in Chapter 2.
Propositional equality judgements, on the other hand, are made at the level of the (object) type theory itself. i.e. $\_ = \_ : A \to A \to \mathbf{Type}$ is an object-theory type constructor (forming the "identity type") and terms of type $t = u$ can be introduced with $\mathsf{refl} : t = t$ and eliminated with the J rule $(J : (P : A \to \mathbf{Type}) \to x = y \to P\ x \to P\ y$, representing the principle of "indiscernibility of identicals").
The motivation for this division is that in dependently-typed systems, types can contain terms that perform real computation, but typechecking requires comparing types for equality (e.g. when checking function application is valid). To retain decidability of typechecking, while enabling programmers to write non-trivial equational proofs, restricting the typechecker to a decidable approximation of equality is required.
The equality reflection rule that defines ETT is simply an equating of propositional and definitional equality. Specifically, adding the typing rule $\mathsf{Tm}\ \Gamma\ (t = u) \to t \equiv u$ (read as: the existence of a proof of propositional equality between t and u implies t and u are convertible) is sufficient to turn an intensional type theory into an extensional one.

[3]: Norell (2007), *Towards a practical programming language based on dependent type theory*
[15]: The Rocq Team (2025), *The Rocq Reference Manual – Release 9.0*
[16]: Moura et al. (2021), *The Lean 4 theorem prover and programming language*
[17]: Brady (2021), *Idris 2: Quantitative type theory in practice*

Since Martin-Löf's first characterisation of intensional type theory [11], propositional equality has been extended in numerous ways (the K rule [12], OTT [13], CTT [14]), but all major presentations retain the ability to introduce with refl and eliminate with J (even if such operations are no longer primitive).
Inspired by the homotopy interpretation of type theory, coercing with J is often called "transporting", and in Agda is written as "subst".

[11]: Martin-Löf (1975), *An intuitionistic theory of types: predicative part*
[12]: Streicher (1993), *Investigations into intensional type theory*
[13]: Altenkirch et al. (2007), *Observational equality, now!*
[14]: Cohen et al. (2016), *Cubical type theory: a constructive interpretation of the univalence axiom*

If we consider propositional equality $t = u$ to be an inductively defined type with one canonical element $\mathsf{refl} : x = x$, it seems reasonable to allow pattern-matching on it like other inductive types. Eliminating equality proofs is only really *useful*[3] though if the LHS and RHS are not already definitionally equal ($J' : \Pi\ (P : A \to \mathbf{Type}) \to x = x \to P\ x \to P\ x$ is just a fancy identity function).

3: Assuming the user is not interested in proving equality of equality proofs. In fact, to prevent deriving the K rule, one must actively prevent matching on $t = u$ when t and u are convertible [2].

This observation motivates "indexed pattern-matching": the extension to dependent pattern matching where arbitrary expressions are admitted as "forced patterns", and matching on elements of the identity type is allowed exactly when variables occuring in LHS/RHS are simultaneously matched with forced patterns such that in the substituted typing context, the propositional equation now holds definitionally [2]. With this feature, one can easily prove, for example $b = \mathsf{true} \to \mathsf{not}\ b = \mathsf{false}$:

[2]: Cockx (2017), *Dependent pattern matching and proof-relevant unification*

Example 1.1.1 (b = true → not b = false)

```
not : 𝔹 → 𝔹
not true ≔ false
not false ≔ true
not-true : Π b → b = true → not b = false
not-true .true refl ≔ refl
```

.pat is Agda's notation for forced patterns. Note that we do not need to provide any case for b ≡ false; conceptually, we are making use of _=_'s elimination rule here, not 𝔹's. Type theory implementations can actually go one step further and infer insertion of forced matches, including on implicit arguments (denoted with { } in Agda), allowing the even more concise:

```
not-true' : Π {b} → b = true → not b = false
not-true' refl ≔ refl
```

We have effectively turned a propositional equality into a definitional one, simply by pattern matching! The downside of course, is that this can only work unifying both sides of the equality provides a set of single-variable substitutions which can be turned into forced matches. If we replace b with an application f b:

```
not-true'' : Π {f : 𝔹 → 𝔹} {b} → f b = true → not (f b) = false
not-true'' refl ≔ refl
```

We get an error:

```
[SplitError.UnificationStuck]
I'm not sure if there should be a case for the constructor refl,
because I get stuck when trying to solve the following unification
problems (inferred index ≟ expected index) :
  f b ≟ true
when checking that the pattern refl has type f b = true
```

The conditions where indexed pattern-matching fails like this are often referred to by dependently-typed programmers as "green slime" [18]. While there are various work-arounds (helper functions that abstract over one side of the equation with a fresh variable, manual coercing with subst, defining functions as inductive relations and inducting on the "graph" of the function [19] etc...), they all require the programmer to write some form of boilerplate.

[18]: McBride (2012), *A polynomial testing principle*

[19]: McBride (2025), *My Favourite Double Category*

This is what makes a principled way of matching on general expressions so exciting (and also hints at its difficulty[4]): a proof assistant with this feature could start inferring forced matches on those expressions, providing a general way to turn propositional equalities (which typically have to be manipulated and applied manually) into definitional ones - i.e. manually invoked equality reflection.

4: Indeed, the fully general version of this feature is undecidable; we will aim to find a decidable fragment.

As an example of how manual equality reflection can simplify many equational proofs, we consider the simple inductive proof that 0 is a right-identity _+_ on ℕs. (i.e. n + 0 = n, where _+_ is addition of natural numbers defined by recursion on the left argument), first in an informal mathematical style:

Theorem 1.1.1 (0 is a right identity of _+_)
In the base case, it remains to prove $0 + 0 = 0$*, which is true by definition of* _+_.
In the inductive case, it remains to prove $(n + 1) + 0 = n + 1$*, which is true by definition of* _+_ *and IH.*

$$
\begin{aligned}
&(n + 1) + 0 \\
&= \text{-- by def \_+\_ } ((n + 1) + m \equiv (n + m) + 1) \\
&(n + 0) + 1 \\
&= \text{-- by inductive hypothesis } ((n + 0) = n) \\
&n + 1
\end{aligned}
$$

In Agda, the same proof is expressed as follows:

Example 1.1.2 (0 is a right identity of $\mathbb{N}$, _+_ , in Agda)

```
+0 : Π n → n + ze = n
+0 ze     ≔ refl
+0 (su n) ≔ cong su (+0 n)
```

As Agda's definitional equality automatically unfolds (β-reduces) pattern-matching definitions (justified by Agda only allowing structurally recursive definitions, so reduction must terminate), we do not need to explicitly appeal to the definition of _+_.

On the other hand, though the syntax makes it concise, we have did have to add more detail in one part of our Agda proof here than the informal one. cong su represents that in the inductive case, we cannot apply the inductive hypothesis directly: we have (n + ze) = n but need su (n + ze) = su n: we must to apply su to both sides. In a type theory supporting case_of_ expressions with inferred forced matches, this manual appeal to congruence of propositional equality would be unnecessary.

```
+0′ : Π n → n + ze = n
+0′ ze     ≔ refl
+0′ (su n) ≔ case +0 n of
  refl → refl
```

The difference might seem small: indeed case_of_ is perhaps overly heavy-weight syntax and so the original +0 definition could be argued more concise, but soon we will examine trickier examples where manual equational reasoning (in the form of cong and, later, subst) starts causing immense pain.

1.2 A Larger Example: First-order Unification

Inspired by [20], we present the use-case of a verified first-order unification algorithm for an untyped syntax containing variables, application and constants. The example is a bit involved, but we repeat it in detail now for a few reasons:

- Since the publication of this work, Agda has had a significant extension to its automation of equational reasoning: global REWRITE rules [21], so it will be interesting to examine where these can and cannot help.

[20]: Sjöberg et al. (2015), *Programming up to congruence*

[21]: Cockx (2020), *Type theory unchained: Extending agda with user-defined rewrite rules*

- It is just a nice example: relatively self-contained while presenting a strong case for improved equational automation.
- The results of this project are all given in terms of a type-theoretic (approximately MLTT) metatheory. In fact, where possible, I aim to mechanise proofs in Agda. Implementing first-order unification for untyped terms should provide a nice introduction to how correct-by-construction programs/proofs are written, as well as the conventions used in this report.

Before we can implement anything however, we must define our syntax. We shall "index" terms by the number of variables in scope (the "context"). For example `Tm 3` will represent a term with three different free variables available.

For this example, we could easily restrict ourselves to a concrete (perhaps infinite) collection of variables, but parameterising like this guides our later implementation of substitutions, and lines up better with the syntaxes (containing binders) that we will look at later.

Definition 1.2.1 (First-Order Terms And Variables)
Variables and terms are defined inductively:

$$\begin{aligned} &Var &&: \mathbb{N} \to \textbf{\textit{Type}} \\ &Tm &&: \mathbb{N} \to \textbf{\textit{Type}} \end{aligned}$$

Terms themselves are easy. We have no binding constructs, so the context stays the same in all cases:

$$\begin{aligned} &\texttt{`\_} &&: Var\ \Gamma \to Tm\ \Gamma && \text{-- Variable} \\ &\texttt{\_`\_} &&: Tm\ \Gamma \to Tm\ \Gamma \to Tm\ \Gamma && \text{-- Application} \\ &\langle\rangle &&: Tm\ \Gamma && \text{-- Constant} \end{aligned}$$

Variables are determined uniquely by an index into the context, or in other words, can be represented by natural numbers strictly smaller than the number of variables in scope.

$$\begin{aligned} &vz &&: Var\ (su\ \Gamma) \\ &vs &&: Var\ \Gamma \to Var\ (su\ \Gamma) \end{aligned}$$

Indexed typed like `Var`/`Fin` can be justified in dependent type theories with a propositional identity type by "Fording" [22]. e.g. for `vz` we could have instead written isomorphic signature $vz : \Delta = su\ \Gamma \to Var\ \Delta$ *- you can have any context, as long as it is* $su\ \Gamma$*!*

This `Var` datatype is common in dependently-typed programming, and is often named `Fin` for "finite set". One way to understand it is that the indexing of `vs` ensures the context is incremented at least as many times as the `Var` itself and `vz` asks for one extra `su` call to make this inequality strict. The flexibility enabling variables to exist in contexts larger than themselves comes from the polymorphism of `vz` (`vz : Var` Δ typechecks for any context $\Delta \geqslant 1$).

[22]: McBride (2000), *Dependently typed functional programs and their proofs*

Unification can be defined as the task of finding a substitution (the "unifier") that maps the terms being unified to equal terms[5] . It seems reasonable then, to define substitutions next.

5: A reasonable follow-up question here might be: what does equality on terms mean? For now, given these are first-order terms, we only consider on-the-nose syntactic equality.

There are various different approaches to defining substitutions in proof assistants. We shall use a first-order encoding of parallel substitutions; that is, a list of terms, equal in length to the context, where operationally, the variable `vz` will be mapped to the first term in the list the list, `vs vz` to the second and so on...

Definition 1.2.2 *Parallel Substitions*
We define substitutions in terms of lists of terms Tms*, indexed first by the context each of the terms in the list are in, and second by the length of the list.*

```
Tms : ℕ → ℕ → Type
```

```
ε   : Tms Δ ze
_,_ : Tms Δ Γ → Tm Δ → Tms Δ (su Γ)
```

Interpreted as a substitution, Tms *Δ Γ takes terms from context Γ to context Δ.*

```
lookup : Var Γ → Tms Δ Γ → Tm Δ
_[_]   : Tm Γ → Tms Δ Γ → Tm Δ
```

Operationally, substitution is defined by recursion on the target term.

```
lookup vz     (δ , u) ≔ u
lookup (vs i) (δ , u) ≔ lookup i δ
(` i)   [ δ ] ≔ lookup i δ
(t · u) [ δ ] ≔ (t [ δ ]) · (u [ δ ])
⟨⟩      [ δ ] ≔ ⟨⟩
```

This encoding of substitutions is nice for two reasons:
1. Substitutions can be composed while staying canonical. i.e. unlike sequences of single substitutions.
2. Higher-order encodings of substitutions (i.e. as functions) do not scale to dependently-typed syntax (without "very-dependent types" [23, 24])

[23]: Hickey (1996), *Formal objects in type theory using very dependent types*
[24]: Altenkirch et al. (2023), *The Münchhausen Method in Type Theory*

We can now define first-order unifiers (the goal of first-order unification).

Definition 1.2.3 (First-Order Unifiers)

```
Unifier : Tm Γ → Tm Γ → Type
```

```
success : Π (δ : Tms Γ Γ) → t [ δ ] = u [ δ ] → Unifier t u
```

Unification can now be specified as a function that takes two terms and attempts to find a unifier (using the standard Maybe type to deal with possible failure):

```
unify : (t u : Tm Γ) → Maybe (Unifier t u)
```

We now attempy to define unify by cases:

```
unify ⟨⟩ ⟨⟩ ≔ just (success ?0 refl)
-- ?0 : Tms Γ Γ
```

... and immediately hit a slight snag: ⟨⟩ is trivially equal to ⟨⟩ but we still need to provide a substitution. The identity substitution id : Tms Γ Γ is probably most reasonable here, and can be constructed by recursion on context length (id : Tms (su Γ) (su Γ) equals id : Tms Γ Γ with all variables incremented and ` vz appended to the end).

```
unify ⟨⟩ ⟨⟩ ≔ just (success id refl)
```

We also have a couple easy failure cases:

```
unify (t₁ · t₂) ⟨⟩      ≔ nothing
unify ⟨⟩        (u₁ · u₂) ≔ nothing
```

Note that this is only a partial specification of unification. In a perfect world, we would require evidence in the failure case that there really is no unifier, but requiring this would add significant clutter to the example.
In fact, one could go even further: in the successful cases, our specification allows returning to any old unifier, of which there might be many. One could instead aim for the minimal/most general unifier as in [25], but, again, the machinery necessary to prove a particular unifier is indeed the most general one is out of the scope of this example.

[25]: Martelli et al. (1982), *An efficient unification algorithm*

A more interesting case crops up when t is a variable:

```
unify (` i) u ≔ ?0
```

To implement this ?0, we need to perform an "occurs check" to find whether i occurs in u.

We define variables occuring free in first-order terms inductively as:

```
Occurs : Var Γ → Tm Γ → Type
eq : Occurs i (` i)
l· : Occurs i t → Occurs i (t · u)
·r : Occurs i u → Occurs i (t · u)
```

And define occurs checking itself occurs? : (i : Var Γ) (t : Tm Γ) → Dec (Occurs i t) by recursion on terms/variables. *Where Dec A is the type of decisions over whether the type A is inhabited, introduced with yes : A → Dec A and no : ¬ A → Dec A. Negation of a type can be encoded as a function to the empty type* 𝟘.

We also need a way to construct substitutions in which a single variable is replaced with a particular term:

```
_[_↦_] : Tms Δ Γ → Var Γ → Tm Δ → Tms Δ Γ
(δ , u) [ vz ↦ t ] ≔ δ , t
(δ , u) [ vs i ↦ t ] ≔ (δ [ i ↦ t ]) , u
```

Now, along with the i[i↦] : lookup i (δ [i ↦ t]) = t and t[i↦] : ¬ (Occurs i t) → t [id [i ↦ u]] = t (provable by induction on terms, variables and substitutions), and with the help of **with**-abstractions to match on the result of the occurs check (the limitations of this feature vs Smart Case are detailed in Section 2.1.1), we can implement this case (if i does not occur in u, the substition id [i ↦ u] is a valid unifier, if u equals ` i, id is a valid unifier, otherwise unification fails).

```
unify (` i) u with occurs? i u
... | no p      ≔ just (success (id [ i ↦ u ]) (
  lookup i (id [ i ↦ u ])
    = by ⟨ i[i↦] {i ≔ i} ⟩
  u
    = by ⟨ sym (t[i↦] p) ⟩
  u [ id [ i ↦ u ] ] ■))
... | yes eq    ≔ just (success id refl)
... | yes (l· _) ≔ nothing
... | yes (·r _) ≔ nothing
```

Note the manual equational reasoning required to prove id [i ↦ u] is valid. Agda's user-defined REWRITE rules (Section 2.1.3) enable turning i[i↦] into a definitional equation, reducing the proof to just sym (t[i↦] p), but t[i↦] has a pre-condition (¬ (Occurs i t)) and so cannot be turned into a global REWRITE. With Smart Case, we could merely match on i[i↦] {i ≔ i} {δ ≔ δ} {t ≔ u} and t[i↦] {u ≔ t} p, and write the proof as refl.

The remaining case (ignoring swapping of arguments) is where both sides are applications. Here, we attempt to unify the LHSs, and if that succeeds, unify the RHSs with the LHS-unifying substitution applied. If this unification also succeeds, we can compose the two unifiers, and again do some equational reasoning to prove this is a valid unifier.

We define composition of substitutions by recursion on the first:

```
_;_ : Tms Δ Γ → Tms Θ Δ → Tms Θ Γ
ε       ; σ ≔ ε
(δ , t) ; σ ≔ (δ ; σ) , (t [ σ ])
```

And prove the composition law `[][] : t [ δ ] [ σ ] = t [ δ ; σ ]` by induction on terms, variables and substitutions. This enables us to finish the implementation of unify.

```
unify (t₁ · t₂) (u₁ · u₂) with unify t₁ u₁
... | nothing                ≔ nothing
... | just (success δ p) with unify (t₂ [ δ ]) (u₂ [ δ ])
... | nothing                ≔ nothing
... | just (success σ q)     ≔ just (success (δ ; σ) (
  (t₁ [ δ ; σ ]) · (t₂ [ δ ; σ ])
    = by ⟨ sym (cong₂ (_·_) ([][] {t ≔ t₁}) ([][] {t ≔ t₂})) ⟩
  (t₁ [ δ ] [ σ ]) · (t₂ [ δ ] [ σ ])
    = by ⟨ cong₂ _·_ (cong _[ σ ] p) q ⟩
  (u₁ [ δ ] [ σ ]) · (u₂ [ δ ] [ σ ])
    = by ⟨ cong₂ (_·_) ([][] {t ≔ u₁}) ([][] {t ≔ u₂}) ⟩
  (u₁ [ δ ; σ ]) · (u₂ [ δ ; σ ]) ■))
```

This implementation is unfortunately not structurally recursive (`t₂ [ δ ]` is not structurally smaller than `t₁ · t₂`). For the purposes of this example, we just assert termination, though algorithms which Agda will accept more directly do exist [26].

[26]: McBride (2003), *First-order unification by structural recursion*

Manually applying congruence rules has gotten quite tedious. Smart Case would simplify this, by enabling matching on p, q and the four instantiations of the composition law (Smart Case cannot take care of inferring the necessary lemmas for an equational proof, but it can automatically chain them together).

Agda's REWRITE rules are also quite effective here: turning `[][]` into a definitional equation reduces the proof to `cong₂ _·_ (cong _[ σ ] p) q` (its support for non-ground equations making all possible instantiations hold definitionally). However, `[][]` alone is not confluent (and indeed Agda with `--local-confluence` checking enabled catches this). Non-confluent sets of REWRITE rules risk breaking subject reduction [27], so avoiding them are important.

Specifically, `t [ δ ] [ σ ] [ ξ ]` could reduce via `[][]` to `t [ (δ ; σ) ; ξ ]` or to `t [ δ ; (σ ; ξ) ]`, which are not definitionally equal .

[27]: Cockx et al. (2021), *The taming of the rew: a type theory with computational assumptions*

Turning associativity of `_;_ (;; : (δ ; σ) ; ξ = δ ; (σ ; ξ))` into a REWRITE simultaneously with `[][]` repairs confluence[6] , but of course requires proving this additional equation that isn't directly necessary for our result. The situation is complicated even further when we try to combine these rewrites with `i[i↦]` (which simplified the prior unify case): `lookup i (δ [ i ↦ t ]) [ σ ]` can reduce to either `lookup i (δ [ i ↦ t ] ; σ)` (by `[]lookup`) or `t [ σ ]` (by `i[i↦]`); we need to prove additional laws about how `_[_↦_]` and `_;_` interact to resolve this.

6: Technically, we also need `[]lookup : lookup i δ [ σ ] = lookup i (δ ; σ)` but this lemma is required to prove `[][]` anyway.

In general, a particular algebraic structure one may wish to work with in a proof assistant may not have a terminating and confluent presentation of its equations (let alone such a presentation that a conservative, automatic checker could identify as such). Global REWRITE rules force the programmer to make careful decisions as to which laws should be made definitional REWRITEs and which which should be kept propositional.

One could, of course, imagine a proof assistant with support for rewrite rules that can be locally enabled/disabled, but the state-of-the-art in this area [28] leaves confluence/termination checking for future work.

[28]: Komel (2021), *Meta-analysis of type theories with an application to the design of formal proofs*

1.3 One More Example: Mechanising Type Theory

The prior section gave an example where automating congruence simplifies equational reasoning. Cases like this admittedly still might not be fully convincing though: couldn't we just create a small proof-generating script (a "tactic") to automatically generate such congruence proofs?

Indeed, Lean and Rocq feature tactics for exactly these sorts of situations [29].

[29]: Selsam et al. (2016), *Congruence closure in intensional type theory*

The full pain of not being able to reflect propositional equality proofs into definitional assumptions rears its head when one works with heavily-indexed types. A common pattern (commonly known amonst dependently typed programmers "coherence" issues or "transport hell") is as follows:

- Some operations on an indexed type are forced to explicitly coerce (transport) along propositional equations relating different index expression.
- Equational proofs about these operations now need to deal with these transports, for example, explicitly using lemmas that push them inside/outside of function applications.

This situation is especially common when mechanising type theories with some form of type dependency - e.g. System F [30] or dependent type theory.

[30]: Saffrich et al. (2024), *Intrinsically Typed Syntax, a LOGICAL Relation, and the Scourge of the Transfer Lemma*

For example, using the technique of [31] to define dependently-typed syntax (indexed by the interpretation of types), we can attempt to define parallel substitutions analagous to Definition 1.2.2 and prove the composition law. Even in the raw recursive definition of the operation, heavy manual equational reasoning is required to prove type preservation.

[31]: McBride (2010), *Outrageous but meaningful coincidences: dependent type-safe syntax and evaluation*

The details here are mostly irrelevant. The main important thing to note is that the terms here are indexed by context and type (so-called "intrinsically-typed" syntax) and subst is Agda's syntax for transporting along equations (necessary to make the type of the substituted term match up with the goal).

```
_[_]tm : Tm Γ A → Π (δ : Objs s Δ Γ) → Tm Δ (A ∘ ⟦ δ ⟧os)
var x                          [ δ ]tm ≔ obj→tm _ (x [ δ ]v)
app {B ≔ B} M N                [ δ ]tm
  ≔ subst (λ N[] → Tm _ (B ∘ (⟦ δ ⟧os ,sub N[]))) (N [ δ ]tm=)
          (app (M [ δ ]tm) (N [ δ ]tm))
lam {A ≔ A} {B ≔ B} M [ δ ]tm
  ≔ subst (Tm _) (dcong₂⁻¹ Πsem A= (cong (B ∘_)
                   (to-coe=⁻1 _ (δ ↑os= A)
  • ↑[]-helper _ A= • cong (_ ,sub_) (sym (semvz-helper A=)))
  • []-helper B ⟦ δ ⟧os A=)) (lam (M [ δ ↑os _ ]tm))
  where A= ≔ A [ δ ]=
```

When attempting to prove the composition law, things get completely out-of-hand:

```
[]tm-comp : Π (M : Tm Γ A) (δ : Objs s Δ Γ) (σ : Objs t θ Δ)
          → M [ δ ]tm [ σ ]tm = M [ δ ∘os σ ]tm
[]tm-comp {s ≔ s} {t ≔ t} (var x) δ σ
  ≔ cong (obj→tm (max s t)) ([]v-comp x δ σ)
[]tm-comp (app M N) δ σ ≔ app= refl refl refl M= N=
  where M= ≔ []tm-comp M δ σ
        N= ≔ []tm-comp N δ σ
[]tm-comp {s ≔ s} {t ≔ t} (lam {A ≔ A} {B ≔ B} M) δ σ
  ≔ sym rm-subst • coes-cancel2 • sym coes-cancel
  • cong (subst (Tm _) prf)
         (to-coe= _ (lam= refl A[]= B[]= (_•P_ {p ≔ refl} M[]= ↑[]=)))
  where M[]= ≔ []tm-comp M (δ ↑os _) (σ ↑os _)
        A[]= ≔ []-comp A δ σ
        B[]= ≔ cong (B ∘_) (cong ((⟦ δ ⟧os ∘ ⟦ σ ⟧os ∘ semwk _) ,sub_)
```

```
                (cong₂ (λ v₁ v₂ → v₁ ∘ (([[ σ ]]os ∘ semwk _) ,sub v₂))
                       (vzo₌ {A :≡ A [ δ ]} s) (vzo₌ {A :≡ A [ δ ] [ σ ]} t)
        • sym (vzo₌ {A :≡ A [ δ ∘os σ ]} (max s t))))
↑[]₌    :≡ []tm₌ refl (refl ,₌ A[]₌) refl (erefl M) (↑∘os₌ A δ σ)
A₌      :≡ A [ δ ∘os σ ]₌
M₌      :≡ M [ (δ ∘os σ) ↑os _ ]tm₌
lamM₌   :≡ lamsem₌ refl refl refl M₌
prf     :≡ dcong₂⁻¹ Πsem A₌ (cong (B ∘_) (((δ ∘os σ) ↑os₌ A)
        • ↑[]-helper _ A₌ • cong (_ ,sub_) (sym (semvz-helper A₌)))
        • []-helper B [[ δ ∘os σ ]]os A₌)
Aδ₌     :≡ A [ δ ]₌
prfδ    :≡ dcong₂⁻¹ Πsem Aδ₌ (cong (B ∘_) ((δ ↑os₌ A)
        • ↑[]-helper _ Aδ₌
        • cong (_ ,sub_) (sym (semvz-helper Aδ₌)))
        • []-helper B [[ δ ]]os Aδ₌)
coes-cancel  :≡ coe-coe _ (cong (Tm _) prf)
                (cong (Tm _) (Πsem₌ refl ([[]]T₌ refl A[]₌) B[]₌))
coes-cancel2 :≡ coe-coe (lam (M [ wkos (A [ δ ]) δ , vzo s ]tm
                                 [ wkos (A [ δ ] [ σ ]) σ , vzo t ]tm))
                (cong (Tm _ ∘ (_∘ [[ σ ]]os)) prfδ) _
rm-subst     :≡ subst-application′ (Tm _) (λ _ → _[ σ ]tm) prfδ
```

Along with the huge amount of congruence reasoning, a few of the steps here (coes-cancel, coes-cancel2, rm-subst) do nit even correspond to "meaningful"[7] laws and only exist to move around or cancel out transports.

Experiencing this pain when mechanising type theory was a significant motivation in my selecting the topic of this project. The development this final example originates from is available at `https://github.com/NathanielB123/dep-ty-chk/tree/trunk`. Hopefully, Smart Case will assist.

7: Defining "meaningful" here as equations which do not trivially hold on untyped syntax/after erasing transports.

Background and Related Work 2

We begin this section looking at related type-system features and end with a discussion on different approaches for proving decidability of conversion.

2.1 Related Systems/Features

2.1.1 With-Abstraction

Both Agda and Idris 2 support matching on non-variable expression to a limited extent via **with**-abstractions (originally named "views") [5, 6, 32]. The key idea is to apply a one-off rewrite to the context, replacing every occurence of the scrutinee expression with the pattern. In Agda, the implementation also elaborates **with**-abstractions into separate top-level functions which abstract over the scrutinee expression (so the final program only contains definitions that match on individual variables).

[5]: McBride et al. (2004), *The view from the left*
[6]: Agda Team (2024), *With-Abstraction*
[32]: Various Contributors (2023), *Views and the "with" rule*

Unfortunately, the one-off nature of **with**-abstraction rewrites limits their applicability. If we re-attempt the f b = f (f (f b)) proof from Example 1.0.2, taking advantage of this feature, the goal only gets partially simplified:

```
bool-lemma : Π (f : 𝔹 → 𝔹) b → f b = f (f (f b))
bool-lemma f true with f true
bool-lemma f true | true ≔ ?0
```

At ?0, Agda replaces every occurence of f b in the goal with true and so expects a proof of true = f (f true), but it is not obvious how to prove this (we could match on f true again, but Agda will force us to cover both the true and false cases, with no memory of the prior pattern-match).

For scenarios like this, **with**-abstractions in Agda are extended with an additional piece of syntax: following a **with**-abstraction with **in** p binds evidence of the match (a proof of propositional equality) to the new variable p.

This feature can also be simulated without special syntax via the "inspect" idiom [33].

[33]: Various Contributors (2024), *Relation.Binary.PropositionalEquality*

Example 2.1.1 (f b = f (f (f b)), Using with_in_ Syntax)

```
bool-lemma : Π (f : 𝔹 → 𝔹) b → f b = f (f (f b))
bool-lemma f true  with f true in p
bool-lemma f true  | true  ≔ sym (cong f p • p)
bool-lemma f true  | false with f false in q
bool-lemma f true  | false | true  ≔ sym p
bool-lemma f true  | false | false ≔ sym q
bool-lemma f false with f false in p
bool-lemma f false | true  with f true in q
bool-lemma f false | true  | true  ≔ sym q
bool-lemma f false | true  | false ≔ sym p
bool-lemma f false | false ≔ sym (cong f p • p)
```

We can also avoid the manual equality reasoning by repeating earlier pattern matches, but this gets very verbose, even when using Agda's ... syntax for repeating above matches. E.g. the first case turns into:

```
bool-lemma' : Π (f : 𝔹 → 𝔹) b → f b = f (f (f b))
```

```
bool-lemma′ f true with f true in p
... | true          with f true | p
... | true | refl   with f true | p
... | true | refl    ≔ refl
```

Agda contains yet another piece of syntactic sugar to help us here: **rewrite** takes a propositional equality and applies a one-off rewrite to the context (similarly to **with**-abstractions).

```
bool-lemma′′ : Π (f : 𝔹 → 𝔹) b → f b = f (f (f b))
bool-lemma′′ f true with f true in p
... | true          rewrite p
                    rewrite p ≔ refl
```

But by now we are up to four distinct syntactic constructs, and the full proof still ends up more verbose than that with Smart Case.

with-abstractions also have a second critical issue that Smart Case solves: the one-off nature of the rewrite can produce ill-typed contexts. Specifically, it might be the case that for a context to typecheck, some neutral expression (such as n + m) must definitionally be of that neutral form, and replacing it with some pattern (like su l), without "remembering" their equality, causes a type error.

In practice, this forces implementations to re-check validity of the context after a **with**-abstraction and throw errors if anything goes wrong.

Example 2.1.2 (Ill-typed **with**-abstraction Involving Fin)
Fin is the standard name for Var from Definition 1.2.1, introduced with ze : Fin (su n) and su : Fin n → Fin (su n).

```
Pred : Π n m → Fin (n + m) → Type
foo : Π n m (i : Fin (n + m)) → Pred n m i → 𝟙
foo n m i       p with n + m
foo n m ze      p | . (su _) ≔ ⟨⟩
foo n m (su i) p | . (su _) ≔ ⟨⟩
```

Errors with:

```
[UnequalTerms]
w != n + m of type ℕ
when checking that the type
(n m w : ℕ) (i : Fin w) (p : Pred n m i) → 𝟙 of the generated with
function is well-formed
```

This type of error is especially prevalent when working with heavily indexed types like those in Section 1.3.

2.1.2 Type Theories with Local Equational Assumptions

As mentioned in the introduction, this work is largely inspired by, and is intended as a continuation of, Altenkirch et al.'s work on Smart Case [10]. This work primarily focussed on pattern matching on booleans (i.e. only introducing equations from neutral[1] boolean-typed terms to closed boolean values). Even in this limited form, the metatheory is non-trivial,

[10]: Altenkirch (2011), *The case of the smart case - How to implement conditional convertibility?*

1: A "neutral" term is one comprising of a spine of elimination forms blocked on a variable.

with subtleties arising from how extending the set of equational assumptions (called "constraint sets") with new equations requires renormalising all equations with respect to each other. For example:

Example 2.1.3 (Coverage Checking in the Presense of Smart Case)
Consider the program (in a dependent type theory with Smart Case).

```
foo : 𝔹 → ...
foo b ≔ case not b of
  true  → case b of
    true  → ?0
    false → ...
  false → ...
```

A proper implementation of Smart Case should rule that the case ?0 is impossible as the constraint set as the impossible equation false = true is derivable from the constraints not b ~ true and b ~ true plus β-conversion:

```
false
 = -- by def not ( not true = false)
 not true
 = -- by constraint b ~ true
 not b
 = -- by constraint not b ~ true
true
```

Note that the equation being added: b ~ true is, by itself, completely sound, and the term b cannot be reduced further even in the presense of the not b = false constraint. Seemingly, any algorithm capable of detecting impossibility here must iterate normalising all equations until a fixed-point (or impossible equation) is reached.

Note that ruling out these impossible cases is not just a convenience to avoid forcing the programmer to write code for situations that could never occur. Avoiding these cases is *necessary* to retain normalisation of the type theory.

Remark 2.1.1 (Equality Collapse and Consequences for Normalisation)
In ITT, definitionally identifying non-neutral terms is dangerous as it can lead to "equality collapse" [34]. For example, assuming conversion is a congruence relation (which is highly desirable for definitional equality to behave intuitively), and large elimination (being able to eliminate from terms to types - a feature which gives dependent type theory much of its expressivity and power) from booleans, one can derive definitional equality between arbitrary types A and B in the presense of true = false:

[34]: McBride (2010), *W-types: good news and bad news*

```
A
 = -- by def if_then_else_ (if true then t else u = t)
if true then A else B
 = -- by assumption true = false
if false then A else B
 = -- by def if_then_else_ (if false then t else u = u)
B
```

Once all types are definitionally equal, it is easy to type self-application (e.g. 𝔹 = (𝔹 → 𝔹)) and so looping terms like the classic (λ x → x x) (λ x → x x) are typeable, and normalisation of open terms is lost.

Despite these difficulties, some systems do implement similar features, to varying levels of success. GHC Haskell, is based on a System F_C core type theory, but layers on many surface features, including automation of its type-level equality constraints [35] (implemented in the "constraint solving" typechecking phase). Combined with type families, which enable real computation at the type level, we can actually "prove"[2] the `f b = f (f (f b))` example from the introduction (Example 1.0.2).

[35]: Sulzmann et al. (2007), *System F with type equality coercions*

2: There are a few caveats here:
1. Haskell does not allow types to directly depend on values, so we have to encode dependently-typed functions with so-called "singleton" encodings [36, 37].
2. Haskell is a partial language, so a "proof" of any type can be written as `undefined`. Manual inspection is required to check totality/ termination.
3. Haskell does not yet support unsaturated type families [38]. We simulate such a feature here using a concrete type family with no cases, but of course this cannot be instantiated with a "real" type-level function on booleans later.

[36]: Lindley et al. (2013), *Hasochism: the pleasure and pain of dependently typed haskell programming*
[37]: Eisenberg (2020), *Stitch: the sound type-indexed type checker (Functional Pearl)*

[38]: Kiss et al. (2019), *Higher-order type-level programming in Haskell*

Example 2.1.4 (`f b = f (f (f b))`, in Haskell)

```
type data 𝔹 ≔ True | False
type SBool :: 𝔹 → Type
data SBool b where
  STrue :: SBool True
  SFalse :: SBool False
type F :: 𝔹 → 𝔹
type family F b where
boolLemma :: (forall b. SBool b → SBool (F b))
              → forall b. SBool b → F b :~: F (F (F b))
boolLemma f STrue ≔ case f STrue of
  STrue  → Refl
  SFalse → case f SFalse of
    STrue  → Refl
    SFalse → Refl
boolLemma f SFalse ≔ case f SFalse of
  STrue → case f STrue of
    STrue  → Refl
    SFalse → Refl
  SFalse → Refl
```

Unfortunately, Haskell's constraint solving is undecidable and in practice many desirable properties of conversion (such as congruence) do not hold.

Example 2.1.5 (Conversion is not Congruent in GHC Haskell)
In GHC 9.8.2, we can try to derive equations between arbitrary types from the constraint `True ~ False`:

```
type IF :: 𝔹 → a → a → a
type family IF b t u where
  IF True t u ≔ t
  IF False t u ≔ u
bad :: True ~ False
    ⇒ IF True () (() → ()) :~: IF False () (() → ())
bad ≔ Refl
```

But this produces the following type error:

```
• Couldn't match type ' () ' with ' () → () '
    Expected: IF True () (() → ()) :~: IF False () (() → ())
      Actual: () :~: ()
• In the expression: Refl
    In an equation for bad: bad ≔ Refl
```

Haskell is not the only language to support a "Smart Case"-like feature. The dependently-typed language "Zombie" builds congruence closure right into the definitional equality of the surface language and implements Smart Case in full, while retaining decidable typechecking and congruent conversion [20]. The sacrifice is β-conversion: Zombie does not automatically apply computation rules, requiring manual assistance to unfold definitions during typechecking.

[20]: Sjöberg et al. (2015), *Programming up to congruence*

This is certainly an interesting point in the design-space of dependently-typed languages, coming with additional advantages such as the possibility of accepting non-total definitions without endangering decidability of typechecking. However, the focus of this project is justifying extending the definitional equality of existing mainstream proof assistants, which unanimously build in β-equality.

One could view traditional definitional equality as a hack, but it is undeniably effective.

The Lean proof assistant features a tactic for automatically discharing equality proofs following from congruence closure [29], but their algorithm is not capable of interleaving congruence and reduction (which is required in our setting to ensure transitivity of conversion).

[29]: Selsam et al. (2016), *Congruence closure in intensional type theory*

Sixty [39] is a dependent typechecker which also implements a form of Smart Case along with full β-conversion, but there is no published work justifying its implementation theoretically.

[39]: Fredriksson (2019), *Sixty*

Andromeda 2 [28] is a proof assistant that supports local equational assumptions via rewriting and indeed goes beyond ground equations, with the goal of supporting user-specified type theories. They focus primarily on provising soundness of the algorithm, and leave confluence/termination checking and completeness results for future work.

[28]: Komel (2021), *Meta-analysis of type theories with an application to the design of formal proofs*

2.1.3 Type Theories with Global Equational Assumptions

In the vein on Andromeda 2, there has been a significant body of work examining type theories extended with more general global rewrite rules, plus implementations in Dedukti [40], Agda [21] and Rocq [41] (though at the time of writing, the Rocq implementation is still a work-in-progress). Work in the area has examined automatic (albeit necessarily conservative) confluence [27] and termination [42] checking of these rewrites. Agda's implementation of REWRITE rules specifically checks confluence, but not termination.

[40]: Assaf et al. (2016), *Dedukti: a logical framework based on the $\lambda\Pi$-calculus modulo theory*

[21]: Cockx (2020), *Type theory unchained: Extending agda with user-defined rewrite rules*

[41]: Leray et al. (2024), *The Rewster: Type Preserving Rewrite Rules for the Coq Proof Assistant*

[27]: Cockx et al. (2021), *The taming of the rew: a type theory with computational assumptions*

[42]: Genestier (2019), *SizeChangeTool: A termination checker for rewriting dependent types*

Smart Case for infinitary and higher-order types must necessarily be somewhat conservative and reject dangerous cases (as previously mentioned, fully general Smart Case is equivalent to manual equality reflection, which is undecidable), but we will aim for a more tailored criteria for accepting equations than these works, taking advantage of the ground-ness of equations.

2.1.4 Elaboration

A principled and increasingly popular way to design and implement programming languages [43–45] is by "elaboration" into a minimal core syntax. A significant benefit of this approach is modularity [46]: multiple extensions to the surface language can be formalised and implemented without having to worry about their interactions. Elaboration can also increase trust in the resulting system, ensuring that all extensions are ultimately conservative over the, perhaps more-rigorously justfied, core theory.

[43]: Eisenberg (2015), *System FC, as implemented in GHC*

[44]: Brady (2024), *Yaffle: A New Core for Idris 2*

[45]: Ullrich (2023), *An Extensible Theorem Proving Frontend*

[46]: Cockx (2023), *Agda Core: The Dream and the Reality*

[47, 48] have investigated elaborating ETT and ITT with global rewrites respectively to an ITT with explicit transports. Both of these works rely on Uniqueness of Identity Proofs (UIP)/axiom K, which is incompatible with type theories that feature proof-relevant equality (e.g. Homotopy Type Theory).

Note that implicit transporting along equivalences between completely distinct types (such as $\mathbb{N}$ and $\mathbb{Z}$) could be used to implement coercions/subtyping where there is an "obvious" mapping, so restricting equations to those on datatypes with trivial equality is limiting.
Such use-cases in fact seem impossible to handle properly without an elaboration-like process inserting transports, given some sort of term-level computation is ultimately required to map between distinct types.

[47]: Winterhalter et al. (2019), *Eliminating reflection from type theory*
[48]: Blot et al. (2024), *From Rewrite Rules to Axioms in the λ Π-Calculus Modulo Theory*

2.1.5 Coproducts with Strict η

η-equations on type introduction/elimination forms express uniqueness principles and can be seen as "dual" to β-laws (the connection can be made concrete with category theory). For example, η for the unit type $\mathbb{1}$ can be written as $\Pi\ (t : \mathsf{Tm}\ \Gamma\ \mathbb{1}) \to t \sim \langle\rangle$; that is, any $\mathbb{1}$-typed term is convertible to $\langle\rangle$. η for non-dependent functions is written $\Pi\ (t : \mathsf{Tm}\ \Gamma\ (A \Rightarrow' B)) \to t \sim \lambda\ (t\ [\ \mathsf{wk}\ ])\cdot(\grave{}\ \mathsf{vz})^3$: any function-typed term can be expanded into a lambda abstraction over the old term immediately applied to the new variable.

Extending conversion with η for unit, (dependent) pairs and (dependent) functions is relatively well-understood and mainstream proof assistants (such as Agda) commonly support definitional (or "strict") η for these types (as long as conversion-checking is type-directed, these laws are easy to check by η-expanding neutral terms immediately before comparing syntactically). Such types are often collectively referred to as "negative" given they are characterised primarily by their elimination rules.

η laws can also be given for "positive" types (types where the introduction rules, or lack thereof, are primary, such as the empty type, coproducts, booleans, natural numbers etc...). It turns out that strict η for these types is strongly-related, and is actually more powerful than, Smart Case. In fact, presentations of coproduct η [50, 51] often include analagous constructions to Smart Case constraint sets.

3: Note that like Definition 1.2.1, we represent variables here as indices into the context. This convention is known as "de-Bruijn indices" after [49] and lets us avoid dealing with variable-freeness conditions.

[49]: De Bruijn (1972), *Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem*

[50]: Dougherty et al. (2000), *Equality between functionals in the presence of coproducts*
[51]: Altenkirch et al. (2001), *Normalization by evaluation for typed lambda calculus with coproducts*

Focussing on the case of booleans, with the simply-typed recursor $\mathsf{if\_then\_else\_} : \mathsf{Tm}\ \Gamma\ \mathbb{B} \to \mathsf{Tm}\ \Gamma\ A \to \mathsf{Tm}\ \Gamma\ A \to \mathsf{Tm}\ \Gamma\ A$, such an η-rule can be expressed as follows:

Definition 2.1.1 (η For Booleans)

$$\mathbb{B}\text{-}\eta\ :\ \Pi\ (t\ :\ \mathit{Tm}\ \Gamma\ \mathbb{B})\ (u\ :\ \mathit{Tm}\ (\Gamma\ ,\ \mathbb{B})\ A)$$
$$\to u\ [\ \langle\ t\ \rangle\ ] \sim \mathit{if}\ t\ \mathit{then}\ u\ [\ \langle\ \mathit{true}\ \rangle\ ]\ \mathit{else}\ v\ [\ \langle\ \mathit{false}\ \rangle\ ]$$

In words: every term containing a boolean-typed sub-expression can be expanded into an if_then_else_ expression, with the sub-expression replaced by true in the true branch and false in the false branch.
We can, of course, prove such a law internally (even if our theory, like Agda, does not implement η for such types definitionally) by induction on booleans (or pattern-matching), replacing explicit substitutions with function application:

$$\mathbb{B}\text{-}\eta\text{-}\mathit{prop}\ :\ \Pi\ t\ (u\ :\ \mathbb{B} \to A)$$
$$\to u\ t\ =\ \mathit{if}\ t\ \mathit{then}\ u\ \mathit{true}\ \mathit{else}\ u\ \mathit{false}$$
$$\mathbb{B}\text{-}\eta\text{-}\mathit{prop}\ \mathit{true}\ \ u\ :\equiv\ \mathit{refl}$$
$$\mathbb{B}\text{-}\eta\text{-}\mathit{prop}\ \mathit{false}\ u\ :\equiv\ \mathit{refl}$$

Like the local equational assumptions of Smart Case, η for booleans enables reducing repeated matches on the same neutral expression. We can show an example of this using our internal $\mathbb{B}$-η-prop law:

```
collapse-if :  Π (f : 𝔹 → 𝔹) b
            →  if f b then (if f b then true else false) else false
            =  if f b then true else false
collapse-if f b
  ≔ 𝔹-η-prop (f b)
  λ fb → if fb then (if fb then true else false) else false
```

In fact, Boolean η can reduce this even further:

```
collapse-again :  Π (f : 𝔹 → 𝔹) b
                  → if f b then true else false  =  f b
collapse-again f b  ≔  sym (𝔹-η-prop (f b) λ fb → fb)
```

Some further intuition for increase in power that η for booleans provides vs Smart Case can be found in the way the η-rule admits so-called "commuting conversions".

Example 2.1.6 (Commuting Conversions)
Commuting conversions express the principle that case-splits on positive types can be lifted upwards as long as the variables occuring in the scrutinee remain in scope. i.e.

```
comm : Π (f : Tm (Γ , A) B) (t : Tm Γ 𝔹) (u v : Tm Γ A)
     → f [ < if t then u else v > ]
     ~ if t then f [ < u > ] else f [ < v > ]
```

We can show an analagous rule follows internally from η as follows.

```
comm-internal : Π (f : A → B) (t : 𝔹) (u v : A)
              → f (if t then u else v)  =  if t then f u else f v
comm-internal f t u v  ≔  𝔹-η-prop t λ b → f (if b then u else v)
```

In a system with strict η for functions and another type A, definitional equality of functions on A is observational[4] .

Theorem 2.1.1 (Strict η for Functions and Booleans Implies Definitional Observational Equality of Boolean Functions)
Assuming f true ≡ g true, f false ≡ g false:

```
f
 ≡ -- by η-equality of functions
λ b → f b
 ≡ -- by η-equality of booleans
λ b → f (if b then True else False)
 ≡ -- by commuting conversions
λ b → if b then f True else f False
 ≡ -- by assumption
λ b → if b then g True else g False
 ≡ -- by commuting conversions
λ b → g (if b then True else False)
 ≡ -- by η-equality of booleans
λ b → g b
 ≡ -- by η-equality of functions
```

4: Observational equality in type theory refers to the idea that evidence of equality of terms at a particular type can follow the structure of that type [13]. For functions f and g, observational equality takes the form of a function from evidence of equal inputs x = y to evidence of equal outputs f x = f y - i.e. pointwise equality (functions are equal precisely when they agree on all inputs).

[13]: Altenkirch et al. (2007), *Observational equality, now!*

g

Subtly, propositional, observational equality of boolean functions (f true = g true → f false = g false → f = g) is not provable internally using the propositional 𝔹-η-prop principle given above (without function extensionality). We do not have any boolean term b to pass to it!
This makes some sense, given propositional η-laws for inductive types can be proven merely by induction, but observational equality of functions (also known as "function extensionality" in the general case) is not provable in intensional MLTT [12].

[12]: Streicher (1993), *Investigations into intensional type theory*

It is perhaps also worth noting that in a dependently-typed setting, η for general A + B binary coproducts can be obtained merely with η for booleans, Σ types and large elimination, via the encoding A + B $\coloneqq$ $\Sigma\ \mathbb{B}\ (\lambda\ \mathsf{b} \to \mathsf{if\ b\ then\ A\ else\ B})$ [52].

[52]: Maillard (2024), *Splitting Booleans with Normalization-by-Evaluation*

There are a couple downsides to implementing η-conversion for finite coproducts/booleans:

- First, the meta-theory gets quite complicated. Previous proofs of normalisation for STLC with of strict η for binary coproducts have relied on somewhat sophisticated rewriting [53] or sheaf [51] theory. Normalisation for dependent type theory with boolean η remains open (though some progress on this front has been made recently [52]).
- Second, efficient implementation seems challenging. Algorithms such as [51] aggressively introduce case-splits on all neutral subterms of coproduct-type and lift the splits as high as possible, in an effort to prevent the build-up of stuck terms. [52] proposes an similar algorithm for typechecking dependent types with strict boolean η, using a monadic interpreter with an effectful splitting operation. [54] is even more extreme: when a variable f of type $\mathbb{B} \to \mathbb{B}$ is bound, for example, case splits are generated on f true and f false (regardless of whether such neutral terms actually occur anywhere in the body), in essence enumerating over all possible implementations of f. One could imagine improving these algorithms, only case-splitting when a particular coproduct-typed sub-expression appears multiple times but I think Smart Case implementations are still likely to have overall more stable performance characteristics due to the lack of commuting conversions.

[53]: Lindley (2007), *Extensional rewriting with sums*
[51]: Altenkirch et al. (2001), *Normalization by evaluation for typed lambda calculus with coproducts*

[54]: Altenkirch et al. (2004), *Normalization by evaluation for $\lambda \to 2$*

Smart Case is further distinguished from η-equality due to its potential applications beyond finitary, first-order types. Specifically, in this project, I am aiming for a Smart Case that at least supports equations between infinitary-typed ($\mathbb{N}$, List A, Tree A, etc...) *neutrals* (there are dangers here, but they only really arise when non-neutral terms get involved). Decidable η-equality for such types is completely infeasible as it requires identifying functions on those types observationally (by analagous argument to Theorem 2.1.1): in other words, if we could decide conversion modulo η for $\mathbb{N}$s, we would have a way to compute whether arbitrary $\mathbb{N} \to \mathbb{B}$ functions are equal on all inputs, which is enough to solve the halting problem (consider the $\mathbb{N} \to \mathbb{B}$ function that runs a particular Turing machine for the input number of steps and returns whether it halts).

2.2 Decidability of Conversion

As mentioned in Remark 1.1.1, decidability of typechecking dependent types hinges on decidability of conversion, so this property is quite important for type theories intended to be used as programming languages.

The standard approach to proving decidability of conversion is to define a normalisation function (reducing terms to "normal forms"), and then prove this procedure sound and complete. There are multiple distinct approaches to specifying normalisation, and so we will go over the main ones.

Note that all techniques listed below rely to some extent on defining an intermediary term-predicate by recursion on types, showing the predicate holds for all terms by induction on syntax, and then proving the final result by simultaneous induction on types and the predicate (a technique that goes by the names "logical relations", "computability predicates" and "reducibility candidates" in the literature). There are alternative approaches to showing normalisation based purely on rewriting theory, but these have not been shown to scale to dependent types.

2.2.1 Reduction-based

Reduction-based techniques specify normalisation in terms of reduction rules, and normal forms as terms that cannot be reduced further.

When using a congruent small-step reduction relation (the "operational semantics"), one can justify termination of naively reducing with respect to it by proving the reduction relation is well-founded. This technique is called "strong normalisation".

Definition 2.2.1 (Strong Normalisation)
For a given reduction relation on terms $\_\gg\_ : Tm \to Tm \to \mathbf{Type}$*, we can define strong normalisation constructively in terms of the accessibility predicate* Acc*:*

$$SN \coloneqq \Pi\, t \to Acc\ (\lambda\, u\ v \to v \gg u)\ t$$

Intuitively, $Acc\ \_<\_\ x$ *is the type of trees of finite height, where each branch represents a step along the* $\_<\_$ *relation, with* x *at the top and the smallest elements (with respect to* $\_<\_$*) at the bottom. Induction on* Acc *allows us to step down the tree, one layer at a time. It is defined inductively:*

$$Acc : (A \to A \to \mathbf{Type}) \to A \to \mathbf{Type}$$
$$acc : (\Pi\, \{y\} \to y < x \to Acc\ \_<\_\ y) \to Acc\ \_<\_\ x$$

Classically, strong normalisation can be equivalently encoded as the non-existence of infinite chains of reductions:

$$SN\text{-}classical \coloneqq \Pi\, t \to \neg\, \infty Chain\ (\lambda\, u\ v \to v \gg u)\ t$$

Where $\infty Chain$ *is defined coinductively:*

$$\infty Chain : (A \to A \to \mathbf{Type}) \to A \to \mathbf{Type}$$
$$next : \infty Chain\ \_<\_\ x \to A$$
$$step : \Pi\, (c : \infty Chain\ \_<\_\ x) \to next\ c < x$$
$$steps : \Pi\, (c : \infty Chain\ \_<\_\ x) \to \infty Chain\ \_<\_\ next$$

We can easily prove $SN \to SN\text{-}classical$*:*

Note the reduction relation defined on untyped terms Tm here. The extension to typed terms Tm Γ A is easy as long as reduction is type-preserving (obeys "subject reduction").

```
acc-¬chain : Acc _<_ x → ¬ ∞Chain _<_ x
acc-¬chain (acc a) c ≔ acc-¬chain (a (step c)) (steps c)
sn-acc-class : SN → SN-classical
sn-acc-class p t ≔ acc-¬chain (p t)
```

A downside of working with a fully congruent small-step reduction relation is that proving confluence is non-trivial. Furthermore, some type theories lack obvious terminating operational semantics but still have decidable conversion (e.g. type theories with η-rules or explicit substitutions [55] and potentially type theories with a sort of impredicative strict propositions [56]).

[55]: Altenkirch et al. (2009), *Big-step normalisation*

[56]: Abel et al. (2020), *Failure of normalization in impredicative type theory with proof-irrelevant propositional equality*

To handle such theories, one can instead define a *deterministic* small-step reduction relation without congruence rules (except those that reduce scrutinees of elimination forms), and therefore reduces terms only up until they are neutral or introduction-rule headed. Such a relation is known as "weak-head reduction" and justifying its termination goes by the name "weak-head normalisation". The downside is that weak-head normalisation alone does not imply decidability of conversion (e.g. consider how function-typed terms t can be soundly η-expanded to λ (t [wk]) · (` vz), putting them into intro-headed form, without making any "real" progress reducing anything). Conversion checking and weak-head normalisation must be interleaved, and termination of this interleaving must itself be proved through a logical relations argument [57].

[57]: Abel et al. (2016), *On the decidability of conversion in type theory*

Finally, normalisation can also be defined with respect to a big-step reduction relation [58]. In fact, much of the original work on Smart Case attacked the problem using this approach [4]. Representing constraint sets as mappings from neutral terms to normalised expressions enables extending normalisation with a step that looks up stuck neutrals in the map.

[58]: Altenkirch et al. (2020), *Big Step Normalisation for Type Theory*

[4]: Altenkirch (2009), *Smart Case [Re: [Agda] A puzzle with "with"]*

Unfortunately, but problems start to occur when defining merging of constraint sets (i.e. to justify adding new constraints in the branches of case splits). As explained in Example 2.1.3, for looking up of neutrals to work properly, LHSs must all be kept normalised with respect to each other. However, adding a single new constraint might unblock multiple neutral LHSs of other constraints, which might unblock yet more etc... so seemingly the only feasible technique to obtain fully normalised constraint mappings is to iterate reducing all constraints with respect to others until a fixed-point is reached (i.e. analagously to ground completion). The only technique I am aware of to show a fixed-point like this exists is to demonstrate that there exists some well-founded ordering on constraint sets that continues to decrease across iterations: in other words we appear to end up needing a small-step reduction relation anyway.

Remark 2.2.1 (Termination of Ground Completion and E-Graphs) Rewriting-to-completion also relies on having some total order on terms, and ensuring rewrites consistently respect it (i.e. by reversing them when necessary). I think any ordering that places closed values like true/false at the bottom and acts like a term encompassment ordering on neutrals should be sufficient to support Smart Case on booleans and coproducts. Rewrites to non-neutral infinitary-typed (e.g. $\mathbb{N}$) terms are trickier, and I think some sort of "occurs check" will be necessary (the rewrite t → su t cannot be reversed, as we can only justify rewriting neutral terms, but it also clearly would lead to loops if left as-is).

Note that rewriting-to-completion is not the only algorithm for deciding equivalence modulo a set of ground equations: bottom-up techniques such as e-graphs [59, 60] are also applicable. These algorithms can be seen as extending the union-find algorithm to terms, and termination is justified merely by the number of e-classes decreasing during congruence-repairing iterations.
Unfortunately, while equations between neutral terms could likely be reasonably handled by e-graphs, rewrites targetting introduction-headed terms are trickier as these can unblock β-reductions, so we really do need to rewrite eagerly (instead of delaying until conversion checking).

[59]: Nelson (1980), *Techniques for program verification*
[60]: Willsey et al. (2021), *Egg: Fast and extensible equality saturation*

2.2.2 Reduction-free Normalisation

Normalisation does not need to specified with respect to a reduction relation. Reduction-free (also called "semantic" or "algebraic") arguments instead treat syntax as an algebraic structure (e.g. a "Category with Families" or "CwF"), where convertible terms are indistinguishable and theorems like normalisation are proved by showing a particular proof-relevant logical predicate holds for every family of equal terms [61–63].

Such an approach was used to prove normalisation for STLC with coproducts obeying strict η [51] (which, as mentioned in Section 2.1.5, is more powerful than Smart Case for the same type), with the main innovation being to evaluate into a sheaf model rather than the usual presheaf on the category of renamings.

[61]: Altenkirch et al. (1995), *Categorical reconstruction of a reduction free normalization proof*
[62]: Altenkirch et al. (2017), *Normalisation by evaluation for type theory, in type theory*
[63]: Sterling et al. (2021), *Normalization for cubical type theory*
[51]: Altenkirch et al. (2001), *Normalization by evaluation for typed lambda calculus with coproducts*

Current Progress, Aims, Plan 3

3.1 Current Progress

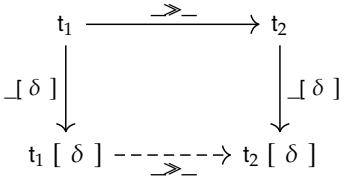
As of submitting this interim report, I have collected a few examples of situations where Smart Case is useful (most of which have been given in Chapter 1) and I have mechanised a proof of strong normalisation for STLC plus rewrites from neutrals into closed boolean values, in Agda (`https://github.com/NathanielB123/fyp/blob/main/STLC/BoolRw/StrongNorm.agda`). The idea is that this setting is the simply-typed analogue to the dependent type theory required to justify Smart Case on booleans.

The proof is based on András Kovacs' Agda translation [64] of Jean-Yves Girard's strong-normalisation proof for STLC in "Proofs and Types" [65], and features a "spontaneous" reduction relation where boolean-typed terms of non intro-form are allowed to be immediately rewritten to true or false at any time, inspired by the extended, weak reduction relation of [50] (denoted with "⇒"). Such a relation is of course not confluent, but it over-approximates the "true" set of reductions that features a convertibility (modulo the constraint set) premise on rewrites, and so strong normalisation of spontaneous reduction implies strong normalisation of the reductions we actually care about.

[64]: Kovács (2020), *StrongNorm.agda*
[65]: Girard et al. (1989), *Proofs and types*

[50]: Dougherty et al. (2000), *Equality between functionals in the presence of coproducts*

In my opinion, the most interesting part of the proof ended-up being getting around the usual requirement for reduction to respect substitution[1] (i.e. $t_1 \gg t_2 \to t_1\ [\ \delta\ ] \gg t_2\ [\ \delta\ ]$). This property usually required while proving the fundamental theorem in the case of lambda abstractions (to go from computability of $t_1 : \mathsf{Tm}\ (\Gamma\ ,\ \mathsf{A})\ \mathsf{B}$ and $u : \mathsf{Tm}\ \Gamma\ \mathsf{A}$ to computability of $t_2\ [\ <\ u\ >\ ] : \mathsf{Tm}\ \Gamma\ \mathsf{B}$ using $t_1 \gg t_2$), and can be expressed with the following diagram:



1: Spontaneous reduction fails here as it enables (` i) ≫ true, but applying the substitution true / i to both sides results in true ≫ true which cannot be allowed if we want reduction to be well-founded.

I solved this by categorising single-variable substitutions into ones that substitute for closed boolean values (Sub^-) and ones that do not (Sub^+). It then becomes possible to prove:

$$\_[\_]{\to}+ : t_1 \gg t_2 \to \mathsf{Sub}^+\ \Delta\ \Gamma\ <\ u\ > \to (t_1\ [\ <\ u\ >\ ]) \gg (t_2\ [\ <\ u\ >\ ])$$

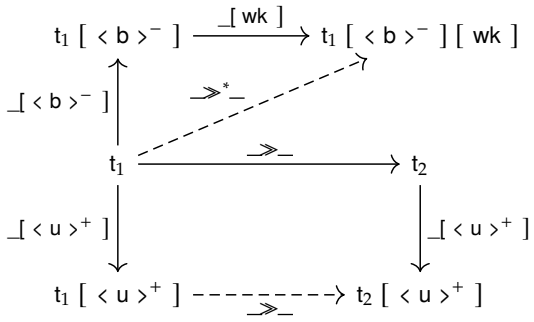
and

$$\mathsf{boolsub}{\to} : \mathsf{Sub}^-\ \Delta\ \Gamma\ <\ b\ > \to t_1 \gg^*\ t_1\ [\ <\ b\ >\ ]\ [\ \mathsf{wk}\ ]$$

Where $\_\gg^*\_$ is the reflexive, transitive closure of spontaneous reduction.

In the Agda mechanisiation, I generalise these lemmas to single-substitutions applying anywhere in the context rather than only on the first variable, but the idea is the same.

Or, as a diagram:



And it turns out this is sufficient to repair the proof.

I have spent some time trying to identify a promising path towards extending this result to dependent types, but so far I don't have anything concrete (just examples of tricky edge-cases and ideas).

Finally, I have also done a bit of hacking on implementation: I have implemented two equality saturation algorithms for first-order terms (`https://github.com/NathanielB123/fyp/blob/main/Completion.hs`): one top-down (rewriting to completion) and one bottom-up (e-graphs), and begun working on an NbE (Normalisation by Evaluation) typechecker.

3.2 The Plan

I think in the immediate future, focussing on implementation is a good idea, and I hope that a simple proof-of-concept will not actually be too difficult to get working. Dependent pattern matching is fiddly, but is also the only real complicated component I need to add (I plan on skipping features like user-defined datatypes, termination checking of recursive functions, universe hierarchies, etc... since the purpose is merely to demonstrate Smart Case rather than build an actually usable dependently-typed language). To extend NbE to deal with the equational assumptions, I plan on maintaining a map from neutrals to values and looking up neutral terms when reflecting/unquoting. The details with adding new equational assumptions might also get a bit tricky, but I think iterating normalisation of every LHS/RHS with respect to all others until a fixed point is reached (i.e. analagous to rewriting-to-completion) should be reasonable.

After I have some primitive implementation, I plan on returning to the theory-side of the project. I want to give extending my simply-typed proof to dependent types a shot, though this will be non-trivial due to how reduction and conversion in dependent type theories are so tightly linked. e.g. spontaneously replacing a neutral $\mathbb{B}$-typed term t with true risks breaking typeability (if t and true not already convertible). I am hoping a definition of (non-transitive) "spontaneous conversion" will be enough to deal with this, but I think the details will be tricky.

An alternative direction could be to focus on semantic approaches to normalisation. I currently am unsure how to justify termination when adding new equational assumptions in this setting, but I think Altenkirch et al.'s work on NbE for STLC + coproducts with strict η-laws [51] must have run into similar problems, so perhaps learning some basic sheaf theory (maybe with the help of [66]) will provide insight.

[51]: Altenkirch et al. (2001), *Normalization by evaluation for typed lambda calculus with coproducts*

[66]: Pédrot (2021), *Debunking Sheaves*

Outside of dependent types, I could also work more on the theory of the simply-typed analogue. For example, building on the strong normalisation result to mechanise a verified conversion-checker, and/or looking into confluence and completeness. I think this is likely to be easier, but perhaps less exciting.

Beyond a simple implementation and progressing the core metatheory, I still believe that most of the potential extensions I listed in the original project proposal would be exciting to look into. I think what is actually feasible will depend heavily on what progress I can make on the aforementioned main tasks though, so I don't think I can make a concrete plan with respect to these yet.

Bibliography

[1] Thierry Coquand. "Pattern matching with dependent types". In: *Informal proceedings of Logical Frameworks*. Vol. 92. Citeseer. 1992, pp. 66–79 (cited on page 1).

[2] Jesper Cockx. "Dependent pattern matching and proof-relevant unification". In: (2017) (cited on pages 1, 3).

[3] Ulf Norell. *Towards a practical programming language based on dependent type theory*. Vol. 32. Chalmers University of Technology, 2007 (cited on pages 1, 3).

[4] Thorsten Altenkirch. *Smart Case [Re: [Agda] A puzzle with "with"]*. 2009. URL: `https://lists.chalmers.se/pipermail/agda/2009/001106.html` (cited on pages 1, 21).

[5] Conor McBride and James McKinna. "The view from the left". In: *Journal of functional programming* 14.1 (2004), pp. 69–111 (cited on pages 1, 12).

[6] The Agda Team. *With-Abstraction*. 2024. URL: `https://agda.readthedocs.io/en/v2.7.0.1/language/with-abstraction.html` (visited on 01/20/2025) (cited on pages 1, 12).

[7] Various Contributors. *Relation.Binary.EqReasoning*. 2024. (Visited on 01/20/2025) (cited on page 1).

[8] Thorsten Altenkirch and Nicolas Oury. *nΣ: A Core Language for Dependently Typed Programming*. 2008 (cited on page 2).

[9] Thorsten Altenkirch et al. "ΠΣ: Dependent types without the sugar". In: *Functional and Logic Programming: 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings 10*. Springer. 2010, pp. 40–55 (cited on page 2).

[10] Thorsten Altenkirch. "The case of the smart case - How to implement conditional convertibility?" In: *NII Shonan Meeting 007*. Sept. 2011 (cited on pages 2, 13).

[11] Per Martin-Löf. "An intuitionistic theory of types: predicative part". In: *Studies in Logic and the Foundations of Mathematics*. Vol. 80. Elsevier, 1975, pp. 73–118 (cited on page 3).

[12] Thomas Streicher. "Investigations into intensional type theory". In: *Habilitiation Thesis, Ludwig Maximilian Universität* (1993), p. 57 (cited on pages 3, 19).

[13] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. "Observational equality, now!" In: *Proceedings of the 2007 workshop on Programming languages meets program verification*. 2007, pp. 57–68 (cited on pages 3, 18).

[14] Cyril Cohen et al. "Cubical type theory: a constructive interpretation of the univalence axiom". In: *arXiv preprint arXiv:1611.02108* (2016) (cited on page 3).

[15] The Rocq Team. *The Rocq Reference Manual – Release 9.0*. `https://coq.inria.fr/doc/v9.0/refman/`. 2025 (cited on page 3).

[16] Leonardo de Moura and Sebastian Ullrich. "The Lean 4 theorem prover and programming language". In: *Automated Deduction–CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings 28*. Springer. 2021, pp. 625–635 (cited on page 3).

[17] Edwin Brady. "Idris 2: Quantitative type theory in practice". In: *arXiv preprint arXiv:2104.00480* (2021) (cited on page 3).

[18] Conor McBride. "A polynomial testing principle". In: (2012), p. 37 (cited on page 4).

[19] Conor McBride. *My Favourite Double Category*. 2025. URL: `https://www.youtube.com/watch?v=aAgqu1y4Xto` (cited on page 4).

[20] Vilhelm Sjöberg and Stephanie Weirich. "Programming up to congruence". In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2015, pp. 369–382 (cited on pages 5, 16).

[21] Jesper Cockx. "Type theory unchained: Extending agda with user-defined rewrite rules". In: *25th International Conference on Types for Proofs and Programs, TYPES 2019*. Schloss Dagstuhl-Leibniz-Zentrum fur Informatik GmbH, Dagstuhl Publishing. 2020, p. 2 (cited on pages 5, 16).

[22] Conor McBride. "Dependently typed functional programs and their proofs". In: (2000) (cited on page 6).

[23] Jason J Hickey. "Formal objects in type theory using very dependent types". In: *Foundations of Object Oriented Languages* 3 (1996), pp. 117–170 (cited on page 7).

[24] Thorsten Altenkirch et al. “The Münchhausen Method in Type Theory”. In: *28th International Conference on Types for Proofs and Programs 2022*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2023, p. 10 (cited on page 7).

[25] Alberto Martelli and Ugo Montanari. “An efficient unification algorithm”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4.2 (1982), pp. 258–282 (cited on page 7).

[26] Conor McBride. “First-order unification by structural recursion”. In: *Journal of functional programming* 13.6 (2003), pp. 1061–1075 (cited on page 9).

[27] Jesper Cockx, Nicolas Tabareau, and Théo Winterhalter. “The taming of the rew: a type theory with computational assumptions”. In: *Proceedings of the ACM on Programming Languages* 5.POPL (2021), pp. 1–29 (cited on pages 9, 16).

[28] Anja Petković Komel. “Meta-analysis of type theories with an application to the design of formal proofs”. PhD thesis. Ph. D. Dissertation. University of Ljubljana, 2021 (cited on pages 9, 16).

[29] Daniel Selsam and Leonardo de Moura. “Congruence closure in intensional type theory”. In: *Automated Reasoning: 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27–July 2, 2016, Proceedings 8*. Springer. 2016, pp. 99–115 (cited on pages 10, 16).

[30] Hannes Saffrich, Peter Thiemann, and Marius Weidner. “Intrinsically Typed Syntax, a LOGICAL Relation, and the Scourge of the Transfer Lemma”. In: *Proceedings of the 9th ACM SIGPLAN International Workshop on Type-Driven Development*. 2024, pp. 2–15 (cited on page 10).

[31] Conor McBride. “Outrageous but meaningful coincidences: dependent type-safe syntax and evaluation”. In: *Proceedings of the 6th ACM SIGPLAN Workshop on Generic Programming*. 2010, pp. 1–12 (cited on page 10).

[32] Various Contributors. *Views and the "with" rule*. 2023 (cited on page 12).

[33] Various Contributors. *Relation.Binary.PropositionalEquality*. 2024. (Visited on 01/20/2025) (cited on page 12).

[34] Conor McBride. *W-types: good news and bad news*. 2010. URL: `https://mazzo.li/epilogue/index.html%3Fp=324.html` (visited on 01/21/2025) (cited on page 14).

[35] Martin Sulzmann et al. “System F with type equality coercions”. In: *Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*. 2007, pp. 53–66 (cited on page 15).

[36] Sam Lindley and Conor McBride. “Hasochism: the pleasure and pain of dependently typed haskell programming”. In: *SIGPLAN Not.* 48.12 (Sept. 2013), pp. 81–92. DOI: `10.1145/2578854.2503786` (cited on page 15).

[37] Richard A Eisenberg. “Stitch: the sound type-indexed type checker (Functional Pearl)”. In: *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell*. 2020, pp. 39–53 (cited on page 15).

[38] Csongor Kiss et al. “Higher-order type-level programming in Haskell”. In: *Proceedings of the ACM on Programming Languages* 3.ICFP (2019), pp. 1–26 (cited on page 15).

[39] Olle Fredriksson. *Sixty*. 2019. (Visited on 01/21/2025) (cited on page 16).

[40] Ali Assaf et al. “Dedukti: a logical framework based on the $\lambda\Pi$-calculus modulo theory”. In: (2016) (cited on page 16).

[41] Yann Leray et al. “The Rewster: Type Preserving Rewrite Rules for the Coq Proof Assistant”. In: (2024) (cited on page 16).

[42] Guillaume Genestier. “SizeChangeTool: A termination checker for rewriting dependent types”. In: *HOR 2019-10th International Workshop on Higher-Order Rewriting*. 2019, pp. 14–19 (cited on page 16).

[43] Richard A Eisenberg. “System FC, as implemented in GHC”. In: (2015) (cited on page 16).

[44] Edwin Brady. *Yaffle: A New Core for Idris 2*. 2024. URL: `https://www.youtube.com/watch?v=_ApsEm2t6UY` (cited on page 16).

[45] Sebastian Andreas Ullrich. “An Extensible Theorem Proving Frontend”. PhD thesis. Dissertation, Karlsruhe, Karlsruher Institut für Technologie (KIT), 2023, 2023 (cited on page 16).

[46] Jesper Cockx. *Agda Core: The Dream and the Reality*. 2023. URL: `https://jesper.cx/posts/agda-core.html` (visited on 01/21/2024) (cited on page 16).

[47] Théo Winterhalter, Matthieu Sozeau, and Nicolas Tabareau. “Eliminating reflection from type theory”. In: *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 2019, pp. 91–103 (cited on page 17).

[48] Valentin Blot et al. “From Rewrite Rules to Axioms in the $\lambda\,\Pi$-Calculus Modulo Theory”. In: *International Conference on Foundations of Software Science and Computation Structures*. Springer. 2024, pp. 3–23 (cited on page 17).

[49] Nicolaas Govert De Bruijn. “Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem”. In: *Indagationes mathematicae (proceedings)*. Vol. 75. 5. Elsevier. 1972, pp. 381–392 (cited on page 17).

[50] Daniel J Dougherty and Ramesh Subrahmanyam. “Equality between functionals in the presence of coproducts”. In: *Information and Computation* 157.1-2 (2000), pp. 52–83 (cited on pages 17, 23).

[51] Thorsten Altenkirch et al. “Normalization by evaluation for typed lambda calculus with coproducts”. In: *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*. IEEE. 2001, pp. 303–310 (cited on pages 17, 19, 22, 24).

[52] Kenji Maillard. “Splitting Booleans with Normalization-by-Evaluation”. In: *30th International Conference on Types for Proofs and Programs TYPES 2024–Abstracts*. 2024, p. 121 (cited on page 19).

[53] Sam Lindley. “Extensional rewriting with sums”. In: *International Conference on Typed Lambda Calculi and Applications*. Springer. 2007, pp. 255–271 (cited on page 19).

[54] Thorsten Altenkirch and Tarmo Uustalu. “Normalization by evaluation for $\lambda\!\rightarrow 2$”. In: *Functional and Logic Programming: 7th International Symposium, FLOPS 2004, Nara, Japan, April 7-9, 2004. Proceedings 7*. Springer. 2004, pp. 260–275 (cited on page 19).

[55] Thorsten Altenkirch and James Chapman. “Big-step normalisation”. In: *Journal of Functional Programming* 19.3-4 (2009), pp. 311–333 (cited on page 21).

[56] Andreas Abel and Thierry Coquand. “Failure of normalization in impredicative type theory with proof-irrelevant propositional equality”. In: *Logical Methods in Computer Science* 16 (2020) (cited on page 21).

[57] Andreas Abel, Thierry Coquand, and Bassel Mannaa. “On the decidability of conversion in type theory”. In: *22nd International Conference on Types for Proofs and Programs, TYPES*. 2016, pp. 23–26 (cited on page 21).

[58] Thorsten Altenkirch and Colin Geniet. “Big Step Normalisation for Type Theory”. In: *25th International Conference on Types for Proofs and Programs (TYPES 2019)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik. 2020 (cited on page 21).

[59] Charles Gregory Nelson. *Techniques for program verification*. Stanford University, 1980 (cited on page 22).

[60] Max Willsey et al. “Egg: Fast and extensible equality saturation”. In: *Proceedings of the ACM on Programming Languages* 5.POPL (2021), pp. 1–29 (cited on page 22).

[61] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. “Categorical reconstruction of a reduction free normalization proof”. In: *Category Theory and Computer Science: 6th International Conference, CTCS’95 Cambridge, United Kingdom, August 7–11, 1995 Proceedings 6*. Springer. 1995, pp. 182–199 (cited on page 22).

[62] Thorsten Altenkirch and Ambrus Kaposi. “Normalisation by evaluation for type theory, in type theory”. In: *Logical methods in computer science* 13 (2017) (cited on page 22).

[63] Jonathan Sterling and Carlo Angiuli. “Normalization for cubical type theory”. In: *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE. 2021, pp. 1–15 (cited on page 22).

[64] András Kovács. *StrongNorm.agda*. Aug. 2020. URL: `https://github.com/AndrasKovacs/misc-stuff/blob/master/agda/STLCStrongNorm/StrongNorm.agda` (visited on 01/16/2025) (cited on page 23).

[65] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types*. Vol. 7. Cambridge university press Cambridge, 1989 (cited on page 23).

[66] Pierre-Marie Pédrot. “Debunking Sheaves”. In: *Unpublished manuscript. Feb* 8 (2021), p. 76 (cited on page 24).