IMPERIAL

MEng Individual Project

Department of Computing

Imperial College of Science, Technology and Medicine

Local Rewriting in Dependent Type Theory

Author:
Nathaniel Burke

Supervisor:
Dr. Steffen van Bakel

Second Marker:
Dr. Herbert Wiklicky

June 13, 2025

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

Abstract

Type theory provides a foundation for mechanised mathematics and cutting-edge programming languages. Unfortunately, to avoid ambiguity and retain decidable typechecking, many computer implementations support only a primitive notion of equality, forcing tedious manual work onto the user. For example, users of proof assistants often resort to doing equational reasoning "by hand", not only specifying which equational lemmas a result relies on, but also exactly how and where to apply them.

With the aim of resolving this tedium, this report studies type theories with a built-in notion of local equational assumptions. When designing these, a balance must be struck between automation, predictability and decidability. Ultimately, we strike such a balance, proving normalisation, and consequently, decidability of typechecking for a language that we name $\text{SC}^{\textsc{Def}}$. We argue this language could serve as a basis for future proof assistant development.

Acknowledgements

I would like to thank:

- Steffen van Bakel, for agreeing to supervise this project, and trying his best to warn me about leaving the report-writing until the last minute.
- David Davies, for giving useful writing advice and super detailed feedback on drafts of this report.
- Thorsten Altenkirch and Philip Wadler, for giving me a huge confidence boost by inviting me to collaborate on "Substitution Without Copy and Paste" after I answered just a couple questions on the Agda Zulip.
- Honestly, pretty-much the whole type theory community, for being so open to sharing work in progress and answering questions across Zulip, Discord, Mastodon, StackExchange, mailing lists etc. A few people helped answer questions directly relevant to this project, including Guillaume Allais, who taught me the "don't mash the potato" principle; Reed Mullanix, who suggested I look into extension types and Raphaël Bocquet, who resolved my confusions with stabilised neutrals. I also want to shout-out András Kovács, Conor McBride and Amélia Liao for producing some phenomenal learning resources. I should also thank Anja Petković Komel, Loïc Pujet and Théo Winterhalter for inspiring me to copy their use of the kaobook LaTeX template!
- My friends, for somehow putting up with my incessant rambling about dependent types for the past few years. I think Daniel, Iurii, Jacob, Jyry, Sophia and Robin probably faced the brunt of it. I also want to especially thank Alona; our late-night conversations about PL design during 2nd and 3rd year are a huge part of why I fell in love with this field.
- My family, for not only having to also put up with my type theory obsession, but also immediately supporting my pivot in future plans towards academia. Those plans are admittedly looking a little shaky right now, but I extremely grateful that I can count on your encouragement regardless.

Contents

Introduction 1

Dependent type theory provides a foundation for mechanised mathematics and cutting-edge programming languages, in which the proof writer/programmer can say what they mean with such precision that their claims (“this theorem follows from these lemmas”, or “this array access will never be out of bounds”) can be unambiguously checked by a computer implementation.

A significant boon of type-theory-based proof assistants is their generality, being capable of scaling to modern mathematics [1, 2], and metamathematics, including the study of new type theories [3, 4]. However, this generality comes with a curse: using proof assistants often entails a significant amount of tedium and boilerplate [5].

[1]: Escardó et al. (2025), *TypeTopology*
[2]: Buzzard et al. (2025), *FLT*
[3]: Pujet et al. (2022), *Observational equality: now for good*
[4]: Abel et al. (2023), *A Graded Modal Dependent Type Theory with a Universe and Erasure, Formalized*
[5]: Shi et al. (2025), *QED in Context: An Observation Study of Proof Assistant Users*

One significant pain-point in proof assistants is having to do equational reasoning manually[1] . With a rich literature of techniques designed to automatically decide equational theories (the *word problem*), including term rewriting [8], and E-Graphs [9, 10], it is perhaps surprising that the capabilities of many proof assistants are quite limited in this area. For example, Agda and Rocq only have only recently gained global rewrite rules [11, 12]. One possible underlying reason for this state of affairs is that modern proof assistants based specifically on intensional type theory (ITT) rely on the built-in (*definitional*) equality obeying some quite strong properties, including decidability (so it can actually be automated), transitivity and congruence. Extending definitional equality without losing any of these properties is challenging.

[6]: Saffrich et al. (2024), *Intrinsically Typed Syntax, a Logical Relation, and the Scourge of the Transfer Lemma*
[7]: Kaposi et al. (2025), *Type Theory in Type Theory Using a Strictified Syntax*
[8]: Baader et al. (1998), *Term Rewriting and All That*
[9]: Nelson (1980), *Techniques for program verification*
[10]: Willsey et al. (2021), *egg: Fast and extensible equality saturation*
[11]: Cockx (2019), *Type Theory Unchained: Extending Agda with User-Defined Rewrite Rules*
[12]: Leray et al. (2024), *The Rewster: Type Preserving Rewrite Rules for the Coq Proof Assistant*

For an example of where equational reasoning “by hand” gets tedious, consider the below proof by cases that for any Boolean function, f : 𝔹 → 𝔹, f **tt** = f (f (f **tt**)) (we can of course also prove f **ff** = f (f (f **ff**)), by an analagous argument).

```
𝔹-split : Π b → (b = tt → A) → (b = ff → A) → A
f3 : Π (f : 𝔹 → 𝔹) → f tt = f (f (f tt))
f3 f
  ≔ 𝔹-split (f tt)
    -- f tt = tt
    (λ p → f tt
             = by  cong f (sym p)
           f (f tt)
             = by  cong (λ □ → f (f □)) (sym p)
           f (f (f tt)) ■)
    -- f tt = ff
    (λ p → 𝔹-split (f ff)
      -- f ff = tt
      (λ q → f tt
               = by  cong f (sym q)
             f (f ff)
               = by  cong (λ □ → f (f □)) (sym p)
             f (f (f tt)) ■)
```

This example is originally from the Agda mailing list [13].

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

[1] This gets especially egregious when proving properties of functions which themselves rely on manual equational reasoning. In such situations (often referred to as “transport hell”), we must employ *coherence* lemmas to deal with seemingly entirely bureaucratic details, unrelated to the underlying function we actually care about e.g. showing that coercions (or *transports*) can be pushed under function applications [6, 7].

```
-- f ff = ff
(λ q → f tt
         = by p
        ff
         = by sym q
        f ff
         = by cong f (sym q)
        f (f ff)
         = by cong (λ □ → f (f □)) (sym p)
        f (f (f tt)) ■))
```

We could shorten the Agda proof here by "code golfing". For example, we could swap the readable (but verbose) equational reasoning combinators provided by the Agda standard library [14], for direct appeals to transitivity of equality (_•_). Specifically, this final case could be written as p • sym (cong f (cong f p • q) • q) - still pretty convoluted!

[14]: Various Contributors (2024), *Relation.Binary.EqReasoning*

In the proof-of-concept dependent typechecker written during this project, the same theorem can be proved successfully with just the following proof term.

```
\f. sif (f tt) then Rfl else (sif (f ff) then Rfl else Rfl)
```

Note that along with being much shorter, the proof is also conceptually simpler. We merely split on the result of f **tt** and f **ff**, and the rest is automated.

This example highlights a strong connection between local equational assumptions and pattern matching[2]: every branch of a case split gives rise to an equation between the "scrutinee" (the thing being split on) and the "pattern", which the programmer might wish to take advantage of. Connecting these two ideas is not novel - Altenkirch et al. first investigated such a construct during the development of ΠΣ [15, 16], naming it **smart case** [17]. However, this work was never published, ΠΣ eventually moved away from **smart case**, and both completeness and decidability (among other metatheoretical properties) remain open.

[2] Which we will analyse in more detail in Remark 3.2.2

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

[16]: Altenkirch et al. (2010), *PiSigma: Dependent Types without the Sugar*

[17]: Altenkirch (2011), *The case of the smart case*

This project then, can be seen as an attempt at continuing this work. At risk of spoiling the conclusion early: our final type theory, SC^{Def}, will actually move away from truly local equations, showing that we can recover most of the *expressivity* of **smart case** while only introducing new equations at the level of *definitions*, and most of the *convenience* via *elaboration*. We argue that SC^{Def} has potential to serve as a basis for an implementation of **smart case** in e.g. Agda.

On the path towards this conclusion though, we will first investigate the problem of deciding equivalence of simply-typed lambda calculus (STLC) terms modulo equations, and also spend time studying a minimal dependent type theory featuring "full" **smart case** for Booleans, named SC^{Bool}. Concretely, our contributions include:

- A proof of decidability of STLC modulo β-conversion, plus a set of Boolean equations (specifically, equations between $\mathbb{B}$-typed terms and closed Booleans tt/ff): Section 4.1.
- A specification of a minimal dependently-typed language with **smart case** named SC^{Bool}, including an appropriately-generalised notion of substitution to account for contexts containing equational assumptions: Section 5.1.
- Soundness of SC^{Bool}, by constructing a model in Agda: Section 5.2.
- A typechecking algorithm for SC^{Bool}, including a proof-of-concept implementation written in Haskell: Section 5.4.
- A specification of an alternative dependently-typed language supporting equational assumptions, but this time at the level of global *definitions*, SC^{Def}, along with another soundness proof: Section 6.1.
- Decidability of conversion for SC^{Def}, leveraging the technique of normalisation by evaluation (NbE): Section 6.2.
- An elaboration algorithm from a surface language with **smart case** to SC^{Def} (compared to "native" **smart case**, we lose only congruence of conversion over case splits): Section 6.3.

The formal results of this project are *mostly* mechanised in Agda. Some holes (corresponding to boring congruence cases) remain in the soundness proofs, and the NbE proofs skip over some bureaucratic details pertaining to e.g. naturality of substitution.

Background 2

2.1 Agda-as-a-Metatheory

In this report, the ambient metatheory in which we will define languages and write proofs will itself be a type theory (Agda [18, 19], specifically).

This poses a bit of a conundrum for the task of providing background: to formally introduce type theory, we must first formally introduce type theory. We shall take the compromise of first informally explaining the syntax/semantics of our metatheory (not too dissimilarly to how one might work in an "intuitive" set theory without being given the ZFC axioms), and then look at how to model type theories mathematically.

Readers already familiar with (dependent) type theory and Agda syntax[1] may wish to skip ahead to .

When working inside a particular type theory, we directly write both *terms* (typically denoted with the letters t, u and v) and *types* (denoted with the letters A, B, C, etc.). Under the Curry-Howard correspondence [20], type theories can be seen to semantically represent *logics* or *programming languages* with terms as *programs* or *proofs*, and types as *specifications* or *theorems*. Because of type theory's ability to act as a *logic*, we must carefully distinguish between *internal* and *external* judgements: internal judgements are objects of the type theory itself, arising from regarding terms as *proofs*, while external judgements are those made in a metatheory (one level up) about objects of the type theory. One example of an "external" judgement is typing: every term, t, is associated with a single type, A[2], called the type of t. We denote this relationship between types and terms with the ":" operator, so "|t| has type |A|" is written t : A.

Variables and Binding

Central to intuitive notation for type theory are the notions of *variables* and *binding*. Effectively, variables provide a way to name "placeholders" which stand for possible terms (we call the name of a variable its *identifier*). Terms embed variables, but a particular variable can only be used after it has been bound, which involves declaring its type (syntactically, we reuse the "**:**" operator for denoting the types of variables at their binding-sites, so e.g. x : A denotes that the variable x is bound with type A).

Variables in type theory closely mirror their functionality in other *programming languages*. From a *logic* perspective, we can view variables as a way of labelling *assumptions*.

Keeping track of which variables are *in-scope* (having been bound earlier) and their types is the *context*: a list of variable identifiers paired with their types. Contexts are extended via binding, but can also be more generally mapped between using *substitutions*: maps from the variables of one context to terms inside another.

When writing *programs*/*proofs* internally to a type theory, we usually do not write contexts or substitutions directly but, when giving examples, it will sometimes be useful to have some notation for these concepts. We denote contexts with the letters Γ, Δ, Θ and write them as (comma-separated) lists of bindings x : A , y : B , z : C , ... We denote substitutions with the letters δ, σ, γ and write them as lists of "/"-separated terms and variables, e.g. "t / x , u / y" denotes a substitution where x is mapped to t and y is mapped to u. Substitutions can applied to types or terms, replacing all embedded variables with their respective substitutes. We denote the action of substitution postfix,

[18]: Norell (2007), *Towards a practical programming language based on dependent type theory*
[19]: Agda Team (2024), *Agda*

[20]: Howard (1980), *The Formulae-as-Types Notion of Construction*

[1] For the benefit of readers who *are* Agda-proficient, we also note that this entire report is written in literate Agda, though we take some liberties with the typesetting to appear closer to on-paper presentations, writing ∀s as **Π**s and swapping the equality symbols _**=**_ and _≡_ to align with their conventional meanings in on-paper type theory

[2] It is of course possible to write down a string of symbols that appears syntactically-similar to a term, but is not type-correct. We do not consider such strings to be valid terms.

with the substitution itself enclosed in "[]"s, i.e. $t\ [\ u\ /\ x\ ]$ denotes the result of replacing all xs in t with u.

Functions

Aside from variables, terms and types are made up of so-called term and type *formers*[3]. Arguably, the most important type former is the Π-type. $\Pi\ (x\ :\ A)\ \to\ B$, where $x\ :\ A$ is bound inside B. Semantically, $\Pi\ (x\ :\ A)\ \to\ B$ represents *functions* or *implications* from A to B.

Term formers can be divided into introduction and elimination rules which express how to construct and use terms of that type, respectively. Functions are introduced with λ-abstraction: $\lambda\ (x\ :\ A)\ \to\ t : \Pi\ (x\ :\ A)\ \to\ B$ given $t : B$, and eliminated with application, denoted by juxtaposition $t\ u : B\ [\ u\ /\ x\ ]$[4] given $t : \Pi\ (x\ :\ A)\ \to\ B$ and $u : A$. Intuitively, abstraction $(\lambda\ (x\ :\ A)\ \to\ t)$ corresponds to binding the *parameters* of a function (here, just x), and application (t u) to applying a given function (t) to an *argument* (u).

For clarity and convenience, *programs*/*proofs* in our metatheory (Agda) can be broken up into definitions: typed identifiers which stand for specific terms (*subprograms*/*lemmas*), allowing their reuse. Syntactically, we declare definitions to have particular types with ":" and "implement" them with "≔". For example, assuming the existence of a *base*[5] type former standing for natural numbers "ℕ", we can write the identity function on ℕ, named idℕ as:

```
idℕ : Π (x : ℕ) → ℕ
idℕ ≔ λ (x : ℕ) → x
```

Note that while the x-variable in the type and body of idℕ do semantically refer to the same ℕ-typed parameter, they can be given different identifiers. The **Π** binds x in the codomain of the function type, and the **λ** separately binds x in the body of the abstraction. We consider types/terms modulo renaming of variables to be identical - *α-equivalence.*

Definitions are similar to variables, but they always stand for a single concrete term (i.e. substitutions cannot substitute them for other terms). When implementing definitions of function type, we can equivalently bind variables on the LHS of the ≔, such as

```
idℕ′ : Π (x : ℕ) → ℕ
idℕ′ x ≔ x
```

which evokes the usual syntax for defining functions in mathematics and programming, $f\ (x)\ =\ x$ (just with the parenthesis optional and using a different equality symbol to convey directness).

Mixfix Notation Some functions, such as addition of natural numbers (+) are more intuitively written *infix* between the two arguments (+ x y vs $x\ +\ y$). Agda supports using such notational conventions by naming definitions with underscores ("_") to stand for the locations of arguments. For example we can declare addition of natural numbers with $\_+\_\ :\ ℕ\ \to\ ℕ\ \to\ ℕ$, and afterwards use it infix ($x\ +\ y : ℕ$ when $x : ℕ$ and $y : ℕ$), prefix (_ + _ x y) or even partially apply it to just the LHS or RHS ((x +_) y or (_+ y) x).

Types can be quite descriptive, and so it is often the case that types and terms alike are unambiguously specified by the surrounding context (*inferable*). Taking advantage of this, we make use of a lot of *syntactic sugar.* We write _ to stand for a term or type that is inferable. e.g. assuming existence of type formers ℕ and Vec n where ℕ denotes the type of natural numbers, and Vec n the type of vectors of length n (where n a term of type ℕ), we can write:

```
zeros : Π (n : ℕ) → Vec n

origin : Vec 3
origin ≔ zeros _
```

Of course, overusing _s can get confusing. They are mostly useful in situations where we are working with relations or predicates, and the details with how the underlying objects are manipulated are somewhat irrelevant.

[3] Sometimes, in other work, these are also collectively called *constructors.* We reserve the term *constructor* only for introduction rules associated with inductive datatypes.

[4] Recall that $B\ [\ u\ /\ x\ ]$ denotes the result of substituting x for u in B.

[5] That is, not *parameterised* by other types.

Given the argument to zeros here clearly ought to be 3. Π-types and λ-abstractions with inferable domains `Π (x : _) → B`, `λ (x : _) → t` can also be written without the annotation on the bound variable `Π x → B`, `λ x → t`. Functions where the codomain does not depend on the domain (like e.g. `idℕ : Π (x : ℕ) → ℕ` above), can also be written more simply by dropping the **Π**, `idℕ : ℕ → ℕ`.

Writing many _s can still get tiresome, so we also allow Π-types to implicitly bind parameters, denoted with "{ }"s, `Π {x : A} → B`. Implicit Π-types can still be introduced and eliminated explicitly with `λ {x : A} → t` and `t {x ≔ u}`[6].

```
idVec : Π {n} → Vec n → Vec n
idVec xs ≔ xs

origin′ : Vec 3
origin′ ≔ idVec (zeros _)
```

Finally, writing **Π**s explicitly can also get unwieldy, so we sometimes rewrite type signatures with seemingly unbound (*free*) variables, which the assumption being that they should be implicit parameters of appropriate type.

```
idVec′ : Vec n → Vec n
idVec′ xs ≔ xs
```

Computation and Uniqueness

Critical to type theory is the notion of computation. Elimination and introduction forms compute when adjacent in so-called β-rules. For example, function applications compute by replacing the bound variable with the argument in the body. More formally, the β-rule for Π-types is written

```
(λ x → t) u ≡ t [ u / x ]
```

Dual to computation rules are *uniqueness* or *extensionality* laws which we call η-rules. Agda features η for Π-types, which tells us that all Π-typed terms, `t : Π (x : A) → B`, are equivalent to terms formed by λ-abstracting a fresh variable and applying it to t

```
t ≡ λ x → t x
```

Some η-rules are a lot trickier to decide than others. A general rule-of-thumb is that η-laws for *negative* type formers (e.g. functions (Π), pairs (Σ), unit (𝟙)) [7]. η-laws for *positive* type formers on the other hand (e.g. Booleans (𝔹), coproducts (_ + _), natural numbers (ℕ), propositional equality (_ = _)) are more subtle, either requiring significantly a more complicated normalisation algorithms [23] or being outright undecidable (we discuss this in more detail in Section 3.5).

[6] Note we specify the name of the parameter we instantiate here, (x). `t { u }` is also a valid elimination-form, but only applies u to the very first implicitly-bound parameter, which is somewhat restrictive.

[7] Specifically, we can delay checking of these η laws until after β-reduction, or alternatively can deal with them directly via NbE (Section 2.4) by specialising unquoting appropriately.
Note that if term-equivalence is not type-directed (e.g. in efficient implementations, or some formalisations), η-rules for negative type formers can still cause trouble [21, 22])

[21]: Lennon-Bertrand (2022), *Á Bas L'η*
[22]: Kovács (2025), *Eta conversion for the unit type*

[23]: Altenkirch et al. (2001), *Normalization by Evaluation for Typed Lambda Calculus with Coproducts*

Universes

In Agda, types are also *first-class* (types are themselves embedded in the syntax of terms)[8]. This means that we have a "type of types", named **Type** and therefore can recover polymorphism (á la System F [24–26]) by implicitly quantifying over **Type**-typed variables. E.g. the polymorphic identity function can be typed as

$$\mathsf{id} : \Pi\,\{\mathsf{A} : \mathbf{Type}\} \to \mathsf{A} \to \mathsf{A}$$

To avoid Russell-style paradoxes, type theories which embed types in terms in this fashion often employ the concept of a universe hierarchy (we call types of types in general *universes*). The term **Type** itself needs type, but **Type** : **Type** is unsound [27]. Instead we have **Type** : $\mathbf{Type}_1$, and $\mathbf{Type}_1$: $\mathbf{Type}_2$ etc... We refer to the Agda documentation [28] for details of how their implementation of universes works.

Equality

Equality in (intensional) type theory is quite subtle. The _≡_ above refers to so-called *definitional* equality (or *conversion*) which the typechecker automatically decides; types are always considered equal up-to-conversion. We sometimes need to refer to equations that the typechecker cannot automate, and for this we use a new type former $\mathsf{x} = \mathsf{y}$, called *propositional* equality. We discuss the intricacies of definitional and propositional equality in more depth in Section 18.

As with Π-types, propositional equality has associated introduction and elimination rules. Specifically, _=_ is introduced with reflexivity, **refl** : $\mathsf{x} = \mathsf{x}$ (x is equal to itself) and can be eliminated using the principle of *indiscernibility-of-identicals* [9] (if $\mathsf{x} = \mathsf{y}$, intuitively we should be able to use terms of type P x in all places where P y is expected, as long as we specify an appropriate $\mathsf{P} : \mathsf{A} \to \mathbf{Type}$ and proof of $\mathsf{x} = \mathsf{y}$).

$$\mathbf{transp} : \Pi\,(\mathsf{P} : \mathsf{A} \to \mathbf{Type}) \to \mathsf{x} = \mathsf{y} \to \mathsf{P}\,\mathsf{x} \to \mathsf{P}\,\mathsf{y}|$$

transp here stands for *transport*, evoking the idea of "transporting" objects of type P x along equalities $\mathsf{x} = \mathsf{y}$. Of course, _=_ must also have a β-rule, **transp** P **refl** x ≡ x. However, we do not assume the corresponding definitional (or *strict*) η-law as this makes conversion (and therefore typechecking) undecidable [29].

We will, however, sometimes take advantage of propositional *uniqueness of identity proofs* (UIP). That is, we will consider all =-typed terms to themselves be propositionally equal, e.g. witnessed with

$$\mathsf{uip} : \Pi\,(\mathsf{p} : \mathsf{x} = \mathsf{x}) \to \mathsf{p} = \mathbf{refl}$$

Assuming UIP globally is incompatible with some type theories (e.g. □TT), but is very convenient when working only with set-based structures.

Remark 2.1.1 (Curry Howard Breaks Down, Slightly)
While the Curry-Howard correspondence can be useful for intuition when learning type theory, some types are much better understood as *logical propositions* and others as *classes of data*. E.g. the natural numbers are a very boring *proposition* given their inhabitation can be trivially proved with ze : ℕ. Conversely, in most type theories =-typed *programs* always return **refl** eventually, and so cannot do much meaningful computation[10].

[8] Note that first-class types are not essential for a type theory to be *dependent* (types can depend on terms via type formers which embed terms). In fact, the type theories we shall study in this project will not support first-class types, or even feature type variables, as the subtleties of such systems are generally orthogonal to the problems we shall consider.

[24]: Fenstad (1971), *Une Extension De L'Interpretation De Gödel a L'analyse, Et Son Application a L'Elimination Des Coupures Dans L'Analyse Et La Theorie Des Types*
[25]: Reynolds (1974), *Towards a theory of type structure*
[26]: Girard (1986), *The System F of Variable Types, Fifteen Years Later*
[27]: Hurkens (1995), *A Simplification of Girard's Paradox*
[28]: Agda Team (2024), *Universe Levels*

[9] The full elimination rule for identity types (named axiom-J or *path induction*) allows the *motive* P to also quantify over the identity proof itself: = -elim : $\Pi\,(\mathsf{P} : \Pi\,\mathsf{y} \to \mathsf{x} = \mathsf{y} \to \mathbf{Type})\ (\mathsf{p} : \mathsf{x} = \mathsf{y}) \to \mathsf{P}\,\mathsf{x}\,\mathbf{refl} \to \mathsf{P}\,\mathsf{y}\,\mathsf{p}$, but **transp** can be derived from this.

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

[10] Actually, computational interpretations of Homotopy Type Theory (HoTT) [30] such as Cubical Type Theory (□TT) [31] propose an alternative perspective, where transporting over the identity type (renamed the *path* type) is a non-trivial operation. For example, paths between types are generalised to isomorphisms (technically, *equivalences*).

[30]: Univalent Foundations Program (2013), *Homotopy Type Theory: Univalent Foundations of Mathematics*
[31]: Cohen et al. (2015), *Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom*

Inductive Types

Agda also contains a scheme for defining types inductively. We declare new inductive type formers with the **data** keyword, and then inside a **where** block, provide the introduction rules.

```
data 𝔹 : Type where                data ℕ : Type where
  tt : 𝔹                             ze : ℕ
  ff : 𝔹                             su : ℕ → ℕ
```

As well as plain inductive datatypes like these, Agda also supports defining parametric inductive types and inductive families, along with forward declarations to enable mutual interleaving. We refer to the documentation for the details on what is supported and the conditions that ensure inductive types are well-founded (namely, strict-positivity) [32].

[32]: Agda Team (2024), *Data Types*

Note we do not need to explicitly give an elimination rule. Inductive types (being *positive* type formers) are fully characterised by their introduction rules (constructors).

Intuitively, eliminators correspond with induction principles, and can be derived mechanically by taking the displayed algebra [33] over the inductive type[11]. For example, the displayed algebra over 𝔹 is a pairing of the *motive* P : 𝔹 → **Type** along with *methods* P tt and P ff, so the eliminator is written as

[33]: Kovács (2023), *Type-theoretic signatures for algebraic theories and inductive types*

```
𝔹-elim : Π (P : 𝔹 → Type) b → P tt → P ff → P b
```

Slightly unusually (e.g. compared to more Spartan type theories like MLTT [35] or CIC [36], or even other type theories implemented by proof assistants like Lean [37] or Rocq [38]), Agda does not actually build-in these elimination principles as primitive. Instead, Agda's basic notion to eliminate inductive datatypes is pattern matching, which is syntactically restricted to the left-hand-side of function definitions.

[34]: Kovács (2023), *What are the complex induction patterns supported by Agda?*

[35]: Martin-Löf (1975), *An Intuitionistic Theory of Types: Predicative Part*

[36]: Pfenning et al. (1989), *Inductively Defined Types in the Calculus of Constructions*

[37]: Moura et al. (2021), *The Lean 4 Theorem Prover and Programming Language*

[38]: The Rocq Team (2025), *The Rocq Reference Manual – Release 9.0*

```
not    : 𝔹 → 𝔹
not tt ≔ ff
not ff ≔ tt
```

Note that traditional eliminators can be defined in terms of pattern matching.

```
𝔹-elim P tt Ptt Pff ≔ Ptt
𝔹-elim P ff Ptt Pff ≔ Pff
```

Equivalence Relations, Quotients and Setoids

Many types have some associated notions of equivalence which are not merely syntactic. For example we might define the integers as any number of applications of successor/predecessor to zero.

```
data ℤ : Type where
  ze : ℤ
  su : ℤ → ℤ
  pr : ℤ → ℤ
```

Of course, we could pick a more "canonical" encoding of the integers, which does support syntactic equality. For example, a natural number magnitude paired with a Boolean sign (being careful to avoid doubling-up on zero, e.g. by considering negative integers to start at -1.).

Sticking only to structures where equality is syntactic is ultimately untenable though. The canonical encoding of some type might be more painful to work with in practice, or might not even exist.

Syntactic equality on this datatype does not quite line up with how we might want this type to behave. E.g. we have ¬ pr (su ze) = ze.

Remark 2.1.2 (Proving Constructor Disjointness)
Agda can automatically rule-out "impossible" pattern matches (i.e. when no construc-

[11] At least for simple inductive types. When one starts defining inductive types mutually with each-other along with recursive functions, quotienting, mixing in coinduction etc... matters admittedly get more complicated [34].

tor is valid) and allows us to write *absurd patterns*, “()”, without a RHS. This syntax effectively corresponds to using the principle of explosion, and relies on Agda’s unification machinery building-in a notion of constructor disjointness.

```
pr-ze-disj : ¬ pr x = ze
pr-ze-disj ()
```

This feature is merely for convenience though. In general, we can prove disjointness of constructors by writing functions that distinguish them, returning 𝟙 or 𝟘. Then under the assumption of equality between the two constructors, we can apply the distinguishing function to both sides and then transport across the resulting proof of 𝟘 = 𝟙 to prove 𝟘 from ⟨⟩.

```
is-pr : ℤ → Type
is-pr (pr _) ≔ 𝟙
is-pr ze     ≔ 𝟘
is-pr (su _) ≔ 𝟘
pr-ze-disj′ : ¬ pr x = ze
pr-ze-disj′ p ≔ transp is-pr p ⟨⟩
```

Being able to pattern match on a term and return a **Type** relies on a feature known as *large elimination*. In type theories with universes, this arises naturally from allowing the motive of an elimination rule/return type of a pattern matching definition to lie in an arbitrary universe.

```
prsu-ze : ¬ pr (su ze) = ze
prsu-ze ≔ pr-ze-disj
```

This situation can be rectified by quotienting. Quotient inductive types allow us to define datatypes mutually with equations we expect to hold, e.g.

```
data Qℤ : Type where
  ze   : Qℤ
  su   : Qℤ → Qℤ
  pr   : Qℤ → Qℤ
  prsu : pr (su x) = x
  supr : su (pr x) = x
```

When pattern matching on quotient types, we are forced to mutually show that our definition is *sound* (i.e. it preserves congruence of propositional equality). Syntactically, each pattern matching definition f defined on Qℤ must include cases for each propositional equation p : x = y (in the case of Qℤ, prsu and supr), returning a proof of f x = f y. For example, we can define doubling on integers doubleQℤ : Qℤ → Qℤ, accounting for prsu and supr like so:

```
doubleQℤ ze     ≔ ze
doubleQℤ (su x) ≔ su (su (doubleℤ x))
doubleQℤ (pr x) ≔ pr (pr (doubleℤ x))
doubleQℤ prsu   ≔
  doubleQℤ (pr (su x))
   = by refl
  pr (pr (su (su (doubleQℤ x))))
   = by  cong pr prsu
  pr (su (doubleQℤ x))
   = by  prsu
  doubleQℤ x ■
doubleQℤ supr   ≔
  doubleQℤ (su (pr x))
   = by refl
  su (su (pr (pr (doubleQℤ x))))
   = by  cong su supr
  su (pr (doubleQℤ x))
  = by supr
  doubleQℤ x ■
```

For technical reasons[12], in the actual Agda mechanisation for this project, we do not use quotients. We can simulate working with quotient types (at the cost of significant boilerplate) by working explicit inductively-defined equivalence relations. E.g. for ℤ

```
data _~ℤ_ : ℤ → ℤ → Type where
  -- Equivalence
  rfl~   : x ~ℤ x
  sym~  : x₁ ~ℤ x₂ → x₂ ~ℤ x₁
  _•~_  : x₁ ~ℤ x₂ → x₂ ~ℤ x₃ → x₁ ~ℤ x₃
  -- Congruence
  su : x₁ ~ℤ x₂ → su x₁ ~ℤ su x₂
  pr : x₁ ~ℤ x₂ → pr x₁ ~ℤ pr x₂
  -- Computation
  prsu  : pr (su x) ~ℤ x
  supr  : su (pr x) ~ℤ x
```

Using this relation, we can implement operations on ℤ, such as doubling, as ordinary pattern matching definitions, and separately write proofs that they respect the equivalence.

```
doubleℤ : ℤ → ℤ
doubleℤ ze      ≔ ze
doubleℤ (su x) ≔ su (su (doubleℤ x))
doubleℤ (pr x) ≔ pr (pr (doubleℤ x))

doubleℤ~ : x₁ ~ℤ x₂ → doubleℤ x₁ ~ℤ doubleℤ x₂
doubleℤ~ rfl~              ≔ rfl~
doubleℤ~ (sym~ x~)         ≔ sym~ (doubleℤ~ x~)
doubleℤ~ (x~₁ •~ x~₂)      ≔ doubleℤ~ x~₁ •~ doubleℤ~ x~₂
doubleℤ~ (su x~)           ≔ su (su (doubleℤ~ x~))
doubleℤ~ (pr x~)           ≔ pr (pr (doubleℤ~ x~))
doubleℤ~ prsu              ≔ pr prsu •~ prsu
doubleℤ~ supr              ≔ su supr •~ supr
```

Note that unlike matching on QITs, we have to explicitly account for cases corresponding to equivalence and congruence[13].

Furthermore, when writing definitions/abstractions parametric over types, when relevant, we must explicitly account for whether each type has an associated equivalence relation. A general *design pattern* arises here: to pair types with their equivalence relations in bundles called *setoids*. The result is sometimes described as "setoid hell" [39] but for smaller mechanisations that stay as concrete as possible, it can be managed.

Setoids can be turned into isomorphic QITs (in theories which support them) by quotienting by the equivalence relation.

```
data _/_ (A : Type) (_~_ : A → A → Type) : Type where
  ⌜_⌝  : A → A / _~_
  quot : Π {x y} → x ~ y → ⌜ x ⌝ = ⌜ y ⌝
```

Translating from QITs to setoids has been explored as part of the work justifying Observational Type Theory (OTT), a type theory that natively supports quotient types and UIP [3, 40, 41]. We will detail the small additional complications when translating types indexed by QITs into setoid *fibrations* (applied to the concrete example of a syntax for dependent type theory) in Section 2.3.3.

[12] In short: Agda currently only supports quotient types as a special cases of higher-inductive type (HIT)s when using the cubical extension, is incompatible with UIP and lacks some useful pattern matching automation. Sometimes, HITs also have to be made significantly more complicated to account for [#7602].

Furthermore, sometimes it is actually useful to be able to temporarily reason about the syntactic structure of objects, even if all operations we might define should ultimately respect the equivalence. For example, when working with *reduction*, Section 2.2.4.

[#7602]: Xie (2025), *Transport in HIT not strictly positive*

[13] It is not too difficult to abstract over rfl~/sym~/_•~_ when mapping between equivalence relations, but this is not really achievable with congruence, which is arguably the worse problem given the number of congruence cases grows with the number of datatype constructors

[39]: Altenkirch (2017), *From setoid hell to homotopy heaven?*

[3]: Pujet et al. (2022), *Observational equality: now for good*
[40]: Altenkirch (1999), *Extensional Equality in Intensional Type Theory*
[41]: Altenkirch et al. (2019), *Setoid Type Theory - A Syntactic Translation*

2.2 Simply Typed Lambda Calculus

Section 1

Having established our metatheory informally, it is time to start studying type theory rigorously. As a warm-up, we begin by covering the theory of simply-typed lambda calculus (STLC), and then will later cover the extensions necessary to support dependent types.

2.2.1 Syntax

> There is no such thing as a free variable. There are only variables bound in the context.
>
> *Conor McBride [42]*

In this report, we will present type theories following the *intrinsically-typed* discipline. That is, rather than first defining a grammar of terms and then separately, the typing relation (i.e. inference rules), we will define terms as an inductive family such that only well-typed terms can be constructed.

Remark 2.2.1 (Syntax-Directed Typing)
Intrinsic typing enforces a one-to-one correspondence between term formers and typing rules (in the language of separate syntax and typing judgements, our inference rules must all be *syntax-directed*). However, features that appear in conflict with this restriction (such as subtyping or implicit coercions) can still be formalised via *elaboration*: that is, in the core type theory, all coercions must be explicit, but this does not prevent defining also an untyped surface language without coercions along with a partial mapping into core terms (the *elaborator*).

In STLC, the only necessary validity criteria on types and contexts is syntactic in nature, so we define these as usual. We include type formers for functions A → B, pairs A × B, sums A + B, unit 𝟙 and the empty type 𝟘, and define contexts as backwards lists of types.

```
data Ty : Type where
  _→_ : Ty → Ty → Ty
  _×_  : Ty → Ty → Ty
  _+_  : Ty → Ty → Ty
  𝟙    : Ty
  𝟘    : Ty
data Ctx : Type where
  •    : Ctx
  _▷_  : Ctx → Ty → Ctx
```

Variables can be viewed as (computationally-relevant[14]) proofs that a particular type occurs in the context. Trivially, the type A occurs in the context Γ ▷ A, and recursively if the type B occurs in context Γ, then the type B also occurs in the context Γ ▷ A.

```
data Var : Ctx → Ty → Type where
  vz : Var (Γ ▷ A) A
  vs : Var Γ B → Var (Γ ▷ A) B
```

After erasing the indexing, we are effectively left with de Bruijn variables [43]; natural numbers counting the number of binders between the use of a variable and the location it was bound.

[14] Note that contexts can contain multiple variables of the same type, and it is important to distinguish these.

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

We avoid named representations of variables in order to dodge complications arising from variable capture and α-equivalence. For legibility and convenience, when writing example programs internal to a particular type theory, we will still use named variables and elide explicit weakening, assuming the existence of a scope-checking/renaming algorithm which can translate to de Bruijn style. When writing such examples we will also separate binding(s) and body with "." rather than "**→**" (along with a non-bolded λ-symbol) as to more clearly distinguish object-level abstraction from that of the meta.

Terms embed variables, and are otherwise comprised of the standard introduction and elimination rules for _ → _, _ × _, _ + _, 𝟙.

In this report, when notation for various constructs is potentially ambiguous between the metatheory and object language, we ensure the meta-level version is denoted in bold. We also denote object-level function application with the binary operator _·_ as opposed plain juxtaposition (though when giving example programs, we will sometimes elide it).

```
data Tm : Ctx → Ty → Type where
  `_   : Var Γ A → Tm Γ A
  λ_   : Tm (Γ ▷ A) B → Tm Γ (A → B)
  _·_  : Tm Γ (A → B) → Tm Γ A → Tm Γ B
  _,_  : Tm Γ A → Tm Γ B → Tm Γ (A × B)
  π₁   : Tm Γ (A × B) → Tm Γ A
  π₂   : Tm Γ (A × B) → Tm Γ B
  in₁  : Tm Γ A → Tm Γ (A + B)
  in₂  : Tm Γ B → Tm Γ (A + B)
  case : Tm Γ (A + B) → Tm (Γ ▷ A) C
       → Tm (Γ ▷ B) C → Tm Γ C
  ⟨⟩   : Tm Γ 𝟙
```

2.2.2 Substitution and Renaming

We define parallel renaming and substitution operations by recursion on our syntax. Following [44], we avoid duplication between renaming (the subset of substitutions where variables can only be substituted for other variables) and substitutions by factoring via a boolean algebra of Sorts, valued either V : Sort or T : Sort with V < T (V standing for "Variable" and T for "Term"). We will skip over most of the details of how to encode this in Agda but define explicitly Sort-parameterised terms:

[44]: Altenkirch et al. (2025), *Substitution without copy and paste*

```
Tm[_] : Sort → Ctx → Ty → Type
Tm[ V ] ≔ Var
Tm[ T ] ≔ Tm
```

and lists of terms (parameterised by the sort of the terms, the context they exist in, and the list of types of each of the terms themselves):

```
data Tms[_] : Sort → Ctx → Ctx → Type where
  ε   : Tms[ q ] Δ •
  _,_ : Tms[ q ] Δ Γ → Tm[ q ] Δ A → Tms[ q ] Δ (Γ ▷ A)
```

We regard lists of variables as renamings, Ren ≔ Tms[V] and lists of terms as full substitutions Sub ≔ Tms[T]. The action of both is witnessed by the following recursively defined substitution operation:

```
_[_] : Tm[ q ] Γ A → Tms[ r ] Δ Γ → Tm[ q ⊔ r ] Δ A
```

Note that ε : Sub Δ • gives us the substitution that weakens a term defined in the empty context into Δ, and _,_ : Sub Δ Γ → Tm Δ A → Tms Δ (Γ ▷ A) expresses the principle that to map from a term in a context Γ extended with A into a context Δ, we need a term in Δ to substitute the zero de Bruijn variable for, Tm Δ A, and a substitution to recursively apply to all variables greater than zero, Sub Δ Γ.

To implement the computational behaviour of substitution, we need to be able to coerce up the sort of terms (terms are functorial over sort ordering, _≤_) and lift substitutions over context extension (substitutions are functorial over context extension[15]):

```
tm≤  : q ≤ r → Tm[ q ] Γ A → Tm[ r ] Γ A
_^_  : Tms[ q ] Δ Γ → Π A → Tms[ q ] (Δ ▷ A) (Γ ▷ A)

vz         [ δ , t ] ≔ t
vs i       [ δ , t ] ≔ i [ δ ]
(` i)      [ δ ]     ≔ tm≤ ≤T (i [ δ ])
(t · u)    [ δ ]     ≔ (t [ δ ]) · (u [ δ ])
(λ t)      [ δ ]     ≔ λ (t [ δ ^ _ ])
⟨⟩         [ δ ]     ≔ ⟨⟩
in₁ B t    [ δ ]     ≔ in₁ B (t [ δ ])
in₂ A t    [ δ ]     ≔ in₂ A (t [ δ ])
case t u v [ δ ]     ≔ case (t [ δ ]) (u [ δ ^ _ ]) (v [ δ ^ _ ])
π₁ t       [ δ ]     ≔ π₁ (t [ δ ])
π₂ t       [ δ ]     ≔ π₂ (t [ δ ])
(t , u)    [ δ ]     ≔ (t [ δ ]) , (u [ δ ])
```

We also use a number of recursively-defined operations to build and manipulate renamings/substitutions, including construction of identity substitutions id (backwards lists of increasing variables), composition _;_, single weakenings wk and single substitutions <_>.

```
id     : Tms[ q ] Γ Γ
_⁺_    : Tms[ q ] Δ Γ → Π A → Tms[ q ] (Δ ▷ A) Γ
suc[_] : Π q → Tm[ q ] Γ B → Tm[ q ] (Γ ▷ A) B
_;_    : Tms[ q ] Δ Γ → Tms[ r ] Θ Δ → Tms[ q ⊔ r ] Θ Γ
wk     : Ren (Γ ▷ A) Γ
<_>    : Tm[ q ] Γ A → Tms[ q ] Γ (Γ ▷ A)

id {Γ ≔ •}     ≔ ε
id {Γ ≔ Γ ▷ A} ≔ id ^ A

suc[ V ] ≔ vs
suc[ T ] ≔ _[ id {q ≔ V} ⁺ _ ]

ε       ⁺ A ≔ ε
(δ , t) ⁺ A ≔ (δ ⁺ A) , suc[ _ ] t

δ ^ A ≔ (δ ⁺ A) , tm≤ V≤ vz

ε       ; σ ≔ ε
(δ , t) ; σ ≔ (δ ; σ) , (t [ σ ])

wk    ≔ id ⁺ _
< t > ≔ id , t
```

Single-weakening of terms via suc[_] and its fold over substitutions _⁺_ can be regarded ultimately as implementation details in service of ensuring our definitions stay structurally well-founded.

[15] Concretely, we can take the category of context extension as dual to the category of weakenings Wk : Ctx → Ctx → **Type** where ε : Wk • • and _⁺_ : Wk Δ Γ → Π A → Wk (Δ ▷ A) Γ.

2.2.3 Soundness

To show how we can prove properties of type theories from our syntax, we will now embark on a proof of *soundness* for STLC.

Definition 2.2.1 (Soundness of a Type Theory)
A type theory with an empty type 𝟘 is sound if there are no 𝟘-typed terms in the empty context.

```
stlc-sound : ¬ Tm • 𝟘
```

Soundness expresses that STLC as a *logic* is *consistent* relative to our metatheory (Agda). From a *PL* perspective, constructing the *standard model* effectively entails writing *interpreter/evaluator* for STLC *programs*, and soundness is strongly related to STLC being a *total* programming language - it does not admit *general recursion* or *unchecked exceptions*.

Our strategy to prove this will be based on giving denotational semantics to STLC: we will interpret STLC constructs as objects in some other theory (i.e. construct a model). A natural choice is to interpret into corresponding objects in our metatheory (Agda), developing what is known as the ‘standard model’ or ‘meta-circular interpretation’.

In the standard model, we interpret object-theory types into their counterparts in **Type**. We call the inhabitants of these interpreted types *values* - i.e. $⟦ A ⟧^{Ty}$ is the type of A-typed closed values.

```
⟦Ty⟧ : Type₁
⟦Ty⟧ ≔ Type

⟦_⟧ᵀʸ : Ty → ⟦Ty⟧
⟦ A → B ⟧ᵀʸ ≔ ⟦ A ⟧ᵀʸ → ⟦ B ⟧ᵀʸ
⟦ A × B ⟧ᵀʸ ≔ ⟦ A ⟧ᵀʸ × ⟦ B ⟧ᵀʸ
⟦ A + B ⟧ᵀʸ ≔ ⟦ A ⟧ᵀʸ + ⟦ B ⟧ᵀʸ
⟦ 𝟙     ⟧ᵀʸ ≔ 𝟙
⟦ 𝟘     ⟧ᵀʸ ≔ 𝟘
```

Contexts are interpreted as nested pairs of values. We call inhabitants of these nested pairs *environments* - i.e. $⟦ Γ ⟧^{Ctx}$ is the type of environments at type Γ.

```
⟦Ctx⟧ : Type₁
⟦Ctx⟧ ≔ Type

⟦_⟧ᶜᵗˣ : Ctx → ⟦Ctx⟧
⟦ •     ⟧ᶜᵗˣ ≔ 𝟙
⟦ Γ ▷ A ⟧ᶜᵗˣ ≔ ⟦ Γ ⟧ᶜᵗˣ × ⟦ A ⟧ᵀʸ
```

Terms are then interpreted as functions from environments to values, so in non-empty contexts, variables project out their associated values. In other words, we can *evaluate* a term of type A in context Γ into a closed value of type A, $⟦ A ⟧^{Ty}$, given an environment $ρ : ⟦ Γ ⟧^{Ctx}$. Application directly applies values using application of the metatheory and abstraction extends environments with new values, using abstraction of the meta. Given we are working inside of a constructive type theory, meta-functions are computable-by-construction and so well-foundedness is ensured by structural recursion on our syntax.

```
⟦Tm⟧ : ⟦Ctx⟧ → ⟦Ty⟧ → Type
⟦Tm⟧ ⟦Γ⟧ ⟦A⟧ ≔ ⟦Γ⟧ → ⟦A⟧

⟦Var⟧ ≔ ⟦Tm⟧

lookup : Var Γ A → ⟦Var⟧ ⟦ Γ ⟧ᶜᵗˣ ⟦ A ⟧ᵀʸ
lookup vz     (ρ , tⱽ) ≔ tⱽ
lookup (vs i) (ρ , tⱽ) ≔ lookup i ρ

⟦_⟧ᵀᵐ : Tm Γ A → ⟦Tm⟧ ⟦ Γ ⟧ᶜᵗˣ ⟦ A ⟧ᵀʸ
⟦ ` i ⟧ᵀᵐ ρ ≔ lookup i ρ
⟦ ƛ t ⟧ᵀᵐ ρ ≔ λ x → ⟦ t ⟧ᵀᵐ (ρ , x)
```

$$
\begin{array}{l}
\llbracket\, t \cdot u \,\rrbracket^{\mathsf{Tm}}\ \rho \ \coloneqq\ (\llbracket\, t \,\rrbracket^{\mathsf{Tm}}\ \rho)\ (\llbracket\, u \,\rrbracket^{\mathsf{Tm}}\ \rho)\\
\llbracket\, t\ ,\ u \,\rrbracket^{\mathsf{Tm}}\ \rho \ \coloneqq\ (\llbracket\, t \,\rrbracket^{\mathsf{Tm}}\ \rho)\ ,\ (\llbracket\, u \,\rrbracket^{\mathsf{Tm}}\ \rho)\\
\llbracket\, \pi_1\ t \,\rrbracket^{\mathsf{Tm}}\ \rho \ \coloneqq\ \llbracket\, t \,\rrbracket^{\mathsf{Tm}}\ \rho\ .\boldsymbol{\pi}_1\\
\llbracket\, \pi_2\ t \,\rrbracket^{\mathsf{Tm}}\ \rho \ \coloneqq\ \llbracket\, t \,\rrbracket^{\mathsf{Tm}}\ \rho\ .\boldsymbol{\pi}_2\\
\llbracket\, \mathsf{in}_1\ B\ t \,\rrbracket^{\mathsf{Tm}}\ \rho \ \coloneqq\ \mathbf{in}_1\ (\llbracket\, t \,\rrbracket^{\mathsf{Tm}}\ \rho)\\
\llbracket\, \mathsf{in}_2\ A\ t \,\rrbracket^{\mathsf{Tm}}\ \rho \ \coloneqq\ \mathbf{in}_2\ (\llbracket\, t \,\rrbracket^{\mathsf{Tm}}\ \rho)\\
\llbracket\, \mathsf{case}\ t\ u\ v \,\rrbracket^{\mathsf{Tm}}\ \rho\ \mathbf{with}\ \llbracket\, t \,\rrbracket^{\mathsf{Tm}}\ \rho\\
\dots\ \mid\ \mathbf{in}_1\ t^{\mathsf{V}}\ \coloneqq\ \llbracket\, u \,\rrbracket^{\mathsf{Tm}}\ (\rho\ ,\ t^{\mathsf{V}})\\
\dots\ \mid\ \mathbf{in}_2\ t^{\mathsf{V}}\ \coloneqq\ \llbracket\, v \,\rrbracket^{\mathsf{Tm}}\ (\rho\ ,\ t^{\mathsf{V}})\\
\llbracket\, \langle\rangle \,\rrbracket^{\mathsf{Tm}}\ \rho \ \coloneqq\ \langle\rangle
\end{array}
$$

Soundness of STLC can now be proved by evaluating the $\mathbb{0}$-typed program in the empty context.

$$
\mathsf{stlc\text{-}sound}\ t\ \coloneqq\ \llbracket\, t \,\rrbracket^{\mathsf{Tm}}\ \langle\rangle
$$

The standard model is useful for more than just soundness. Note that after interpreting, computationally-equivalent closed terms become definitionally equal.

$$
\begin{array}{l}
\beta\text{-}\mathsf{example}\ :\ \llbracket\, (\lambda\ \grave{}\ \mathsf{vz}) \cdot \langle\rangle \,\rrbracket^{\mathsf{Tm}}\ \equiv\ \llbracket\, \langle\rangle\ \{\Gamma\ \coloneqq\ \bullet\} \,\rrbracket^{\mathsf{Tm}}\\
\beta\text{-}\mathsf{example}\ \coloneqq\ \mathbf{refl}
\end{array}
$$

This makes sense, given the definitional equality of our metatheory (Agda) encompasses β-equality. Computationally-equivalent terms in general can be described as those which are propositionally equal after interpreting. E.g.

$$
\llbracket\beta\rrbracket\ :\ \llbracket\, (\lambda\ t) \cdot u \,\rrbracket^{\mathsf{Tm}}\ \equiv\ \llbracket\, t\ [\ <\ u\ >\ ]\,\rrbracket^{\mathsf{Tm}}
$$

Though, to prove $\llbracket\beta\rrbracket$, we need to show that substitution is preserved appropriately by the standard model - i.e. substitution is sound w.r.t. our denotational semantics.

Definition 2.2.2 (Soundness with Respect to a Semantics)
An operation $f\ :\ A\ \rightarrow\ B$ is sound w.r.t. some semantics on A and B if its action respects those semantics.
The nature of this respect depends somewhat on the semantics in question: for soundness w.r.t. a model, we show that the model admits an analogous operation $\llbracket f\rrbracket$ such that the following diagram *commutes*[16]

$$
\begin{tikzcd}
x \arrow[r, "{\llbracket\_\rrbracket^{A}}"] \arrow[d, "f"'] & {\llbracket\, x \,\rrbracket^{A}} \arrow[d, "{\llbracket f\rrbracket}"] \\
f\ x \arrow[r, "{\llbracket\_\rrbracket^{V}}"'] & {\llbracket\, f\ x \,\rrbracket^{V}\ =\ \llbracket f\rrbracket\ \llbracket\, x \,\rrbracket^{A}}
\end{tikzcd}
$$

Given an equational semantics (Section 2.2.4), we instead must show that f preserves the equivalence, and in the case of operational semantics, reduction should be stable under f.

Soundness of Substitution

Substitutions that map terms from context Γ to context Δ can be interpreted as functions from Δ-environments to Γ-environments.

$$
\begin{array}{l}
\llbracket\mathsf{Sub}\rrbracket\ :\ \llbracket\mathsf{Ctx}\rrbracket\ \rightarrow\ \llbracket\mathsf{Ctx}\rrbracket\ \rightarrow\ \mathbf{Type}\\
\llbracket\mathsf{Sub}\rrbracket\ \llbracket\Delta\rrbracket\ \llbracket\Gamma\rrbracket\ \coloneqq\ \llbracket\Delta\rrbracket\ \rightarrow\ \llbracket\Gamma\rrbracket\\
\llbracket\_\rrbracket^{\mathsf{Sub}}\ :\ \mathsf{Sub}\ \Delta\ \Gamma\ \rightarrow\ \llbracket\mathsf{Sub}\rrbracket\ \llbracket\, \Delta \,\rrbracket^{\mathsf{Ctx}}\ \llbracket\, \Gamma \,\rrbracket^{\mathsf{Ctx}}\\
\llbracket\, \varepsilon \,\rrbracket^{\mathsf{Sub}}\ \rho\ \coloneqq\ \langle\rangle\\
\llbracket\, \delta\ ,\ t \,\rrbracket^{\mathsf{Sub}}\ \rho\ \coloneqq\ \llbracket\, \delta \,\rrbracket^{\mathsf{Sub}}\ \rho\ ,\ \llbracket\, t \,\rrbracket^{\mathsf{Tm}}\ \rho
\end{array}
$$

[16] We say a diagram commutes if all paths (compositions of the arrows) that begin and end at the same pair of points are equivalent (e.g. applying the associated actions in sequence always produces the same end-result). In the case of commuting squares like this, we therefore require that going right and then down is equivalent to going down and then right.

The contravariant ordering of Sub's indices is now justified! Γ-terms being interpreted as functions from Γ-environments makes them contravariant w.r.t. environment mappings. The semantic action of substitution (i.e. substitution inside the model) is just function composition.

```
⟦[]⟧ : ⟦Tm⟧ ⟦Γ⟧ ⟦A⟧ → ⟦Sub⟧ ⟦Δ⟧ ⟦Γ⟧ → ⟦Tm⟧ ⟦Δ⟧ ⟦A⟧
⟦[]⟧ ⟦t⟧ ⟦δ⟧ ρ ≔ ⟦t⟧ (⟦δ⟧ ρ)
```

Soundness of _[_] w.r.t. the standard model can now be stated as:

```
[]-sound : ⟦ t [ δ ] ⟧ᵀᵐ ≡ ⟦[]⟧ ⟦ t ⟧ᵀᵐ ⟦ δ ⟧ˢᵘᵇ
```

The case for e.g. t ≔ ⟨⟩ is trivial ([]-sound {t ≔ ⟨⟩} ≔ **refl**), but to prove this law in general, we also need to implement semantic versions of our other recursive substitution operations (i.e. _ˆ_, _⁺_, etc...) and mutually show preservation of all them.

After all of this work, we can finally prove ⟦β⟧ using []-sound and also preservation of <_>, <>-sound.

```
<>-sound : ⟦ < t > ⟧ˢᵘᵇ ≡ ⟦<>⟧ ⟦ t ⟧ᵀᵐ


⟦β⟧ {t ≔ t} {u ≔ u} ≔
  ⟦ (λ t) · u ⟧ᵀᵐ
  ≡⟨ by refl ⟩
  (λ ρ → ⟦ t ⟧ᵀᵐ (ρ , ⟦ u ⟧ᵀᵐ ρ))
  ≡⟨ by refl ⟩
  ⟦[]⟧ ⟦ t ⟧ᵀᵐ (⟦<>⟧ ⟦ u ⟧ᵀᵐ)
  ≡⟨ by cong (⟦[]⟧ ⟦ t ⟧ᵀᵐ) (sym (<>-sound {t ≔ u})) ⟩
  ⟦[]⟧ ⟦ t ⟧ᵀᵐ ⟦ < u > ⟧ˢᵘᵇ
  ≡⟨ by sym ([]-sound {t ≔ t}) ⟩
  ⟦ t [ < u > ] ⟧ᵀᵐ ∎
```

2.2.4 Reduction and Conversion

Constructing a model is not the only way to give a semantics to a type theory. We can also give *operational* and *equational* semantics to STLC using inductive relations named *reduction* and *conversion* respectively.

We arrive at (strong) one-step β-reduction by taking the smallest monotonic relation on terms which includes our computation rules:

```
data _>β_ : Tm Γ A → Tm Γ A → Type where
  -- Computation
  →β  : (λ t) · u            >β t [ < u > ]
  +β₁  : case (in₁ B t) u v >β u [ < t > ]
  +β₂  : case (in₂ A t) u v >β v [ < t > ]
  *β₁  : π₁ (t , u)         >β t
  *β₂  : π₂ (t , u)         >β u

  -- Monotonicity
  λ_     : t₁ >β t₂ → λ t₁       >β λ t₂
  l·     : t₁ >β t₂ → t₁ · u     >β t₂ · u
  ·r     : u₁ >β u₂ → t · u₁     >β t · u₂
  in₁    : t₁ >β t₂ → in₁ B t₁   >β in₁ B t₂
  in₂    : t₁ >β t₂ → in₂ A t₁   >β in₂ A t₂
  case₁  : t₁ >β t₂ → case t₁ u v >β case t₂ u v
  case₂  : u₁ >β u₂ → case t u₁ v >β case t u₂ v
  case₃  : v₁ >β v₂ → case t u v₁ >β case t u v₂
  ,₁     : t₁ >β t₂ → t₁ , u      >β t₂ , u
  ,₂     : u₁ >β u₂ → t , u₁      >β t , u₂
  π₁     : t₁ >β t₂ → π₁ t₁       >β π₁ t₂
  π₂     : t₁ >β t₂ → π₂ t₁       >β π₂ t₂
```

We say a term t_1 reduces to its reduct, t_2, if $t_1 >_\beta{}^* t_2$ (where `_>β*_ : Tm Γ A → Tm Γ A → Type` is the reflexive-transitive closure of `_>β_`). Using this relation, we define terms to be equivalent w.r.t. reduction (*algorithmic conversion*) if they have a common reduct.

```
record _<~>_ (t₁ t₂ : Tm Γ A) : Type where field
  {common} : Tm Γ A
  reduces₁  : t₁ >β* common
  reduces₂  : t₂ >β* common
```

Reduction as a concept becomes much more useful when the relation is well-founded. For a full one-step reduction relation that proceeds under λ-abstractions, we call this property *strong normalisation*, because we can define an algorithm which takes a term t and, by induction on the well-founded order, produces an equivalent (w.r.t. algorithmic conversion) but irreducible term t^{Nf}, t's *normal form*[17] (we show how to do this explicitly in Section 2.4.1).

Definition 2.2.3 (Normalisation)
In this report, we define normalisation algorithms as sound and complete mappings from some type, A, to a type of *normal forms*, Nf^A, with decidable equality.
Soundness here is defined as usual (i.e. the mapping preserves equivalence), while we define completeness as the converse property: that equal normal forms implies equivalence of the objects we started with.
In the formal definition, we assume A is quotiented by equivalence, and so soundness is ensured by the definition of quotient types. Completeness still needs to ensured with a side-condition though.

[17]: Technically, if reduction is not confluent, it might be possible to reduce a term t to multiple distinct normal forms. In principle, we can still explore all possible reduction chains in parallel and compare sets of irreducible terms to decide algorithmic conversion. In this scenario, we can consider the sets of irreducible terms themselves to be the normal forms (with equivalence defined by whether any pair of terms in the Cartesian product are equal syntactically).

Note that we do not enforce that normal forms are subset of the original type, which is sometimes useful flexibility - see e.g. [23].

If we do have an embedding `⌜_⌝ : Nfᴬ → A`, then completeness is equivalent to the property `⌜ norm x ⌝ = x`: if we assume `norm x = norm y`, then by congruence `⌜ norm x ⌝ = ⌜ norm y ⌝`, which simplifies to `x = y`.

[23]: Altenkirch et al. (2001), *Normalization by Evaluation for Typed Lambda Calculus with Coproducts*

```
record Norm : Type where
  field
    norm  : A → Nfᴬ
    compl : norm x ≡ norm y → x ≡ y
```

From normalisation and decidable equality of normal forms _ =$^{\mathrm{Nf}}$?_, we can easily decide equality on A.

```
_ =ᴺᶠ ?_ : Π (xᴺᶠ yᴺᶠ : Nfᴬ) → xᴺᶠ ≡ yᴺᶠ + ¬ xᴺᶠ ≡ yᴺᶠ


_ = ?_ : Π (x y : A) → x ≡ y + ¬ x ≡ y
x  = ? y with norm x  =ᴺᶠ ? norm y
... | in₁ p ≔ in₁ (compl p)
... | in₂ p ≔ in₂ λ q → p (cong norm q)
```

Perhaps uninventively, for theories which already have decidable equivalence, we are forced to accept the identity mapping as a valid (if trivial) implementation of normalisation. We can make sense of this situation by viewing normalisation as a strategy/technique for deciding equivalence, rather than an objective in-and-of-itself.

If we instead take the smallest congruent equivalence relation which includes the computation rules, we arrive at *declarative β*-conversion.

```
data _~_ : Tm Γ A → Tm Γ A → Type where
  -- Equivalence
  rfl~ : t ~ t
  sym~ : t₁ ~ t₂ → t₂ ~ t₁
  _•~_ : t₁ ~ t₂ → t₂ ~ t₃ → t₁ ~ t₃
  -- Computation
   →β : (λ t) · u             ~ t [ < u > ]
  +β₁  : case (in₁ B t) u v ~ u [ < t > ]
  +β₂  : case (in₂ A t) u v ~ v [ < t > ]
  *β₁  : π₁ (t , u)          ~ t
  *β₂  : π₂ (t , u)          ~ u
  -- Congruence
  λ_   : t₁ ~ t₂ → λ t₁ ~ λ t₂
  _·_  : t₁ ~ t₂ → u₁ ~ u₂ → t₁ · u₁ ~ t₂ · u₂
  in₁  : t₁ ~ t₂ → in₁ B t₁ ~ in₁ B t₂
  in₂  : t₁ ~ t₂ → in₂ A t₁ ~ in₂ A t₂
  case : t₁ ~ t₂ → u₁ ~ u₂ → v₁ ~ v₂ → case t₁ u₁ v₁ ~ case t₂ u₂ v₂
  _,_  : t₁ ~ t₂ → u₁ ~ u₂ → t₁ , u₁ ~ t₂ , u₂
  π₁   : t₁ ~ t₂ → π₁ t₁ ~ π₁ t₂
  π₂   : t₁ ~ t₂ → π₂ t₁ ~ π₂ t₂
```

We now have three distinct semantics-derived equivalence relations on terms.

Algorithmic and declarative β-conversion (as we have defined them here) are themselves equivalent notions. $t_1 \sim t_2 \to t_1$ <~> t_2 requires proving transitivity of _<~>_, which follows from confluence (which itself can be proved by via *parallel reduction* [45]). The converse, t_1 <~> t_2 follows from $\_>_\beta\_$ being contained inside _~_ ($t_1 >_\beta t_2 \to t_1 \sim t_2$).

We can also prove that the standard model preserves _~_, but it turns equality in the standard model is not equivalent to conversion as we have defined it. The model also validates various η equalities (inherited from the metatheory), including

```
⟦𝟙η⟧ : Π {t : Tm Γ 𝟙} → ⟦ t ⟧ᵀᵐ ≡ ⟦ ⟨⟩ ⟧ᵀᵐ
⟦𝟙η⟧ ≔ refl
```

and

```
⟦→η⟧ : Π {t : Tm Γ (A → B)}
       → ⟦ t ⟧ᵀᵐ ≡ ⟦ λ ((t [ wk ]) · (` vz)) ⟧ᵀᵐ
```

[45]: Takahashi (1995), *Parallel Reductions in lambda-Calculus*

(though the latter requires an inductive proof).

Declaring a $\beta\eta$-conversion relation which validates such equations is easy (we can just add the relevant laws as cases), but doing the same for reduction (while retaining normalisation and confluence) is tricky [46, 47].

[46]: Ghani (1995), *Adjoint Rewriting*
[47]: Lindley (2007), *Extensional Rewriting with Sums*

These interactions motivate taking declarative conversion as the "default" specification of the semantics when defining type theories from now on. Of course, poorly-designed conversion relations might be undecidable or equate "morally" distinct terms (e.g. **tt** $\sim$ **ff** is likely undesirable). We therefore should aim to justify our definitions of declarative conversion by e.g. constructing models which preserve the equivalence and proving desirable *metatheoretic* properties of the theory, such as normalisation. Given most operations on terms ought to respect conversion, it can be quite convenient to quotient (Section 1.1 - Equivalence Relations, Quotients and Setoids) terms by the relation (of course, up to conversion, reduction is a somewhat ill-defined concept).

2.2.5 Explicit Substitutions

For STLC, we were able to get away with first defining terms inductively, and then substitutions later as a recursive operation (and conversion after that, as an inductive relation). It is unclear how to do the same for dependent type theory (specifically, ITT) given types (with embedded terms) must be considered equal up to at least β-conversion (and β-conversion at Π-types inevitably refers to substitution.) One might hope to find a way to define a dependently-typed syntax mutually with a recursive substitution operation, but unfortunately it is currently unclear how to make this work in practice [7].

If one takes untyped terms as primitive and then defines typing relations explicitly, recursive substitution for dependent types is achievable directly [48], but this approach requires many tedious proofs that e.g. substitution preserves typing.

[48]: Abel et al. (2018), *Decidability of conversion for type theory in type theory*
[7]: Kaposi et al. (2025), *Type Theory in Type Theory Using a Strictified Syntax*

We therefore give an explicit substitution syntax for STLC, based on categories with families (CwFs)[49, 50], which can be more easily adapted to the setting of dependent types.

[49]: Dybjer (1995), *Internal Type Theory*
[50]: Castellan et al. (2019), *Categories with Families: Unityped, Simply Typed, and Dependently Typed*

Unlike our previous syntax, our explicit substitution calculus only contains four main sorts: contexts, types, terms and substitutions but no variables. Without variables, we no longer parameterise substitutions by whether they are renamings or "full" substitutions.

$$\begin{aligned}
&\mathsf{Ctx} &&: \textbf{Type}\\
&\mathsf{Ty} &&: \textbf{Type}\\
&\mathsf{Tm} &&: \mathsf{Ctx} \rightarrow \mathsf{Ty} \rightarrow \textbf{Type}\\
&\mathsf{Tms} &&: \mathsf{Ctx} \rightarrow \mathsf{Ctx} \rightarrow \textbf{Type}
\end{aligned}$$

We start with some properties of substitutions. Substitutions should form a category with contexts as objects (i.e. there is an identity substitution, and they can be composed).

Definition 2.2.4 (Category)
A type of objects Ob and family of morphisms Hom : Ob → Ob → **Type** forms a category if Hom includes identity and composition.

```
record Category : Type where field
  id   :  Hom x x
  _∘_  :  Hom x y → Hom y z
          → Hom x z
  id∘  :  id ∘ f = f
  ∘id  :  f ∘ id = f
  ∘∘   :  (f ∘ g) ∘ h = f ∘ (g ∘ h)
```

We denote flipped composition with _;_, which is convenient for substitutions as it aligns with their action on terms being denoted postfix.

We quotient by substitution laws here, but of course we could work up to some equivalence relation instead. By quotienting by the substitution laws, but not β/η, we can obtain a syntax that is isomorphic (w.r.t. propositional equality) to the recursive substitution approach (the proof of this is given in detail in [44]).

[44]: Altenkirch et al. (2025), *Substitution without copy and paste*

```
id   : Tms Γ Γ
_;_  : Tms Δ Γ → Tms Θ Δ → Tms Θ Γ
id;  : id ; δ  = δ
;id  : δ  ; id = δ
;;   : (δ ; σ) ; γ = δ ; (σ ; γ)
```

The category of substitutions features a terminal object (the empty context). The unique morphism ε applied to terms will correspond to weakening from the empty context.

```
•   : Ctx
ε   : Tms Δ •
•η  : δ = ε
```

Definition 2.2.5 (Terminal Object)
An object 𝟙 : Ob in a category C with a family of morphisms Hom is terminal if there is a unique morphism from every other object in the category, x : Ob, to 𝟙, ! : Hom x 𝟙.

```
record Terminal : Type where field
  !     : Hom x 𝟙
  uniq  : f = !
```

Terms are a presheaf on substitutions. That is, there is a (contravariantly) functorial action that applies substitutions to terms.

```
_[_]  : Tm Γ A → Tms Δ Γ → Tm Δ A
[id]  : t [ id ] = t
[][]  : t [ δ ] [ σ ] = t [ δ ; σ ]
```

Definition 2.2.6 (Presheaf)
We call family of types F : A → **Type** a presheaf on a category C (with Ob :≡ A and a family of morphisms Hom) if it is a contravariant functor on C. More concretely, there should exist a functorial action which "lift"s morphisms in C, Hom y x, to functions, F x → F y.

```
record Presheaf : Type where field
  map     : Hom y x → F x → F y
  map-id  : map id xs = xs
  map-∘   : map (f ∘ g) xs
          = map f (map g xs)
```

To support binding, we must equip our CwF with *context comprehension*, including a context extension operation _▷_ : Ctx → Ty → Ctx, and an associated way to extend substitutions a fresh term to replace the new variable with.

```
_▷_  : Ctx → Ty → Ctx
_,_  : Tms Δ Γ → Tm Δ A → Tms Δ (Γ ▷ A)
,;   : (δ , t) ; σ = (δ ; σ) , (t [ σ ])
```

We call laws like ",;" which cover how the various constructs of type theory interact with the functor operations, *naturality* laws. We can express these laws as commutative diagrams, e.g.

$$\begin{tikzcd}
\delta \arrow[r, "\_;\sigma"] \arrow[d, "\_,\mathsf{t}"'] & \delta ; \sigma \arrow[d, "\_,(\mathsf{t}\ [\ \sigma\ ])"] \\
\delta , \mathsf{t} \arrow[r, "\_;\sigma"'] & (\delta , \mathsf{t}) ; \sigma \ =\ (\delta ; \sigma) , (\mathsf{t}\ [\ \sigma\ ])
\end{tikzcd}$$

Given our intuition of parallel substitutions as lists of terms, we should expect a (natural) isomorphism:

```
Tms Δ (Γ ▷ A) ≈ Tms Δ Γ × Tm Δ A
```

This can be witnessed either directly with projection operations, or we can take single-weakening and the zero de Bruijn variable as primitive (wk :≡ π_1 id and vz :≡ π_2 id, or π_1 δ :≡ wk ; δ and π_2 δ :≡ vz [δ]) [50].

[50]: Castellan et al. (2019), *Categories with Families: Unityped, Simply Typed, and Dependently Typed*

```
π₁   : Tms Δ (Γ ▷ A) → Tms Δ Γ
π₂   : Tms Δ (Γ ▷ A) → Tm Δ A        wk   : Tms (Γ ▷ A) Γ
▷η   : δ = π₁ δ , π₂ δ               vz   : Tm (Γ ▷ A) A
π₁,  : π₁ (δ , t) = δ                wk;  : wk ; (δ , t) = δ
π₂,  : π₂ (δ , t) = t                vz[] : vz [ δ , t ] = t
π₁;  : π₁ (δ ; σ) = π₁ δ ; σ         id▷  : id {Γ ≔ Γ ▷ A} = wk , vz
π₂;  : π₂ (δ ; σ) = π₂ δ [ σ ]
```

From these primitives, we can derive single substitutions <_> and functoriality of context extension _ˆ_. The former just substitutes the zero de Bruijn variable for the given term, while acting as identity everywhere else. The latter is obtained by first weakening all terms in the substitution (to account for the new variable) and then mapping the new zero variable to itself.

```
<_> : Tm Γ A → Tms Γ (Γ ▷ A)
< t > ≔ id , t

_ˆ_ : Π (δ : Tms Δ Γ) A → Tms (Δ ▷ A) (Γ ▷ A)
δ ˆ A ≔ (δ ; wk) , vz
```

We can extend this syntax with functions by adding the relevant type former and introduction/elimination rules. Rather than the usual rule for application, it is convenient in explicit substitution syntaxes to take a more *pointfree* combinator as primitive, which directly inverts λ_.

```
_ → _ : Ty → Ty → Ty
λ_    : Tm (Γ ▷ A) B → Tm Γ (A → B)
λ⁻¹_  : Tm Γ (A → B) → Tm (Γ ▷ A) B

λ[]   : (λ t) [ δ ] = λ (t [ δ ˆ A ])
λ⁻¹[] : (λ⁻¹ t) [ δ ˆ A ] = λ⁻¹ (t [ δ ])
```

Semantically, λ^{-1}_ can be understood as the action of weakening the given function, and then applying it to the fresh zero variable. We can derive the more standard rule for application by following this up with a single-substitution:

```
_·_ : Tm Γ (A → B) → Tm Γ A → Tm Γ B
t · u ≔ (λ⁻¹ t) [ < u > ]
```

The advantages of λ^{-1}_ should hopefully be evident from now super-concise statement of the β/η equations for →-types.

```
→β : λ⁻¹ λ t ~ t
→η : t ~ λ λ⁻¹ t
```

For other type formers, using an explicit syntax does not change much, so we will stop here.

2.3 Dependently Typed Lambda Calculus

We will define an intensional type theory. See Section 18 for discussion on alternatives.

2.3.1 Syntax

As with our explicit STLC syntax, we define all four sorts mutually.

$$\begin{aligned}
\mathsf{Ctx} &: \mathbf{Type} \\
\mathsf{Ty} &: \mathsf{Ctx} \to \mathbf{Type} \\
\mathsf{Tm} &: \mathbf{\Pi}\,\Gamma \to \mathsf{Ty}\,\Gamma \to \mathbf{Type} \\
\mathsf{Tms} &: \mathsf{Ctx} \to \mathsf{Ctx} \to \mathbf{Type}
\end{aligned}$$

We start with substitutions. As with STLC, these must form a category. Again, we quotient our syntax, but this time, we will go a bit further and even quotient by some β/η laws to account for definitional equality (in ITT, types should always be considered equivalent up to computation).

As we will detail in Section 2.3.3, it is possible to split the quotienting into a separate equivalence relation, but in the setting of dependent types, the details get a bit more complicated because the indexing of types, terms and substitutions then needs to account for this equivalence (note that substitutions and computation will now occur inside types, so type-equivalence is no longer syntactic).

$$\begin{aligned}
\mathsf{id} &: \mathsf{Tms}\;\Gamma\;\Gamma \\
\_;\_ &: \mathsf{Tms}\;\Delta\;\Gamma \to \mathsf{Tms}\;\Theta\;\Delta \to \mathsf{Tms}\;\Theta\;\Gamma \\
\mathsf{id}; &: \mathsf{id}\;;\delta = \delta \\
;\mathsf{id} &: \delta\;;\mathsf{id} = \delta \\
;; &: (\delta\;;\sigma)\;;\gamma = \delta\;;(\sigma\;;\gamma)
\end{aligned}$$

The category of substitutions features a terminal object (the empty context).

$$\begin{aligned}
\bullet &: \mathsf{Ctx} \\
\varepsilon &: \mathsf{Tms}\;\Delta\;\bullet \\
\bullet\eta &: \delta = \varepsilon
\end{aligned}$$

In dependent type theory, types are a presheaf on substitutions, and terms are a displayed presheaf.

$$\begin{aligned}
\_[\_]_{\mathsf{Ty}} &: \mathsf{Ty}\;\Gamma \to \mathsf{Tms}\;\Delta\;\Gamma \to \mathsf{Ty}\;\Delta \\
[\mathsf{id}]\mathsf{Ty} &: \mathsf{A}\;[\;\mathsf{id}\;]_{\mathsf{Ty}} = \mathsf{A} \\
[][]\mathsf{Ty} &: \mathsf{A}\;[\;\delta\;]_{\mathsf{Ty}}\;[\;\sigma\;]_{\mathsf{Ty}} = \mathsf{A}\;[\;\delta\;;\sigma\;]_{\mathsf{Ty}} \\
\_[\_] &: \mathsf{Tm}\;\Gamma\;\mathsf{A} \to \mathbf{\Pi}\;(\delta\;:\;\mathsf{Tms}\;\Delta\;\Gamma) \to \mathsf{Tm}\;\Delta\;(\mathsf{A}\;[\;\delta\;]_{\mathsf{Ty}}) \\
[\mathsf{id}] &: \mathsf{t}\;[\;\mathsf{id}\;] =^{\mathit{cong}\;(\mathit{Tm}\;\Gamma)\;\mathit{[id]Ty}}\;\mathsf{t} \\
[][] &: \mathsf{t}\;[\;\delta\;]\;[\;\sigma\;] =^{\mathit{cong}\;(\mathit{Tm}\;\Theta)\;\mathit{[][]Ty}}\;\mathsf{t}\;[\;\delta\;;\sigma\;]
\end{aligned}$$

To support binding, we must support a (now dependent) context extension operation $\_\triangleright\_\;:\;\mathbf{\Pi}\;\Gamma \to \mathsf{Ty}\;\Gamma \to \mathsf{Ctx}$.

$$\begin{aligned}
\_\triangleright\_ &: \mathbf{\Pi}\;\Gamma \to \mathsf{Ty}\;\Gamma \to \mathsf{Ctx} \\
\_,\_ &: \mathbf{\Pi}\;(\delta\;:\;\mathsf{Tms}\;\Delta\;\Gamma) \to \mathsf{Tm}\;\Delta\;(\mathsf{A}\;[\;\delta\;]_{\mathsf{Ty}}) \to \mathsf{Tms}\;\Delta\;(\Gamma \triangleright \mathsf{A}) \\
,; &: (\delta\;,\;\mathsf{t})\;;\sigma = (\delta\;;\sigma)\;,\;\mathbf{transp}\;(\mathsf{Tm}\;\Theta)\;[][]\mathsf{Ty}\;(\mathsf{t}\;[\;\sigma\;])
\end{aligned}$$

Like in STLC, we can witness the isomorphism

$$\mathsf{Tms}\;\Delta\;(\Gamma \triangleright \mathsf{A}) \approx (\,\delta : \mathsf{Tms}\;\Delta\;\Gamma\,) \times \mathsf{Tm}\;\Delta\;(\mathsf{A}\;[\;\delta\;]_{\mathsf{Ty}})$$

either by adding projection operations or by taking single-weakening and the zero de Bruijn variable as primitive.

$$\begin{array}{lll}
\pi_1 & : & \text{Tms}\ \Delta\ (\Gamma \rhd \text{A}) \rightarrow \text{Tms}\ \Delta\ \Gamma \\
\pi_2 & : & \mathbf{\Pi}\ (\delta : \text{Tms}\ \Delta\ (\Gamma \rhd \text{A})) \rightarrow \text{Tm}\ \Delta\ (\text{A}\ [\ \pi_1\ \delta\ ]_{\text{Ty}}) \\
\rhd\eta & : & \delta = \pi_1\ \delta\ ,\ \pi_2\ \delta \\
\pi_1, & : & \pi_1\ (\delta\ ,\ \text{t}) = \delta \\
\pi_2, & : & \pi_2\ (\delta\ ,\ \text{t}) =^{\mathit{Tm_=}\ \boldsymbol{\mathit{refl}}\ (\mathit{cong}\ (\mathit{A}\ [\_]\mathit{Ty})\ \pi_1,)}\ \text{t} \\
\pi_1; & : & \pi_1\ (\delta\ ;\ \sigma) = \pi_1\ \delta\ ;\ \sigma \\
\pi_2; & : & \pi_2\ (\delta\ ;\ \sigma) \\
 & & =^{\mathit{Tm_=}\ \boldsymbol{\mathit{refl}}\ (\mathit{cong}\ (\mathit{A}\ [\_]\mathit{Ty})\ \pi_1;\ \bullet\ \mathit{sym}\ [][]\mathit{Ty})}\ \pi_2\ \delta\ [\ \sigma\ ]
\end{array}$$

$$\begin{array}{lll}
\text{wk} & : & \text{Tms}\ (\Gamma \rhd \text{A})\ \Gamma \\
\text{vz} & : & \text{Tm}\ (\Gamma \rhd \text{A})\ (\text{A}\ [\ \text{wk}\ ]_{\text{Ty}}) \\
\text{wk;} & : & \text{wk}\ ;\ (\delta\ ,\ \text{t}) = \delta \\
\text{vz[]} & : & \text{vz}\ [\ \delta\ ,\ \text{t}\ ] =^{\mathit{Tm_=}\ \boldsymbol{\mathit{refl}}\ ([][]\mathit{Ty}\ \bullet\ \mathit{cong}\ (\mathit{A}\ [\_]\mathit{Ty})\ \mathit{wk};)}\ \text{t} \\
\text{id}\rhd & : & \text{id}\ \{\Gamma \coloneqq \Gamma \rhd \text{A}\} = \text{wk}\ ,\ \text{vz}
\end{array}$$

We derive single substitutions <_> and functoriality of context extension _^_ as usual. Note we need to transport the term in both cases to account for the functor laws holding only propositionally.

$$\begin{array}{l}
<\_>\ :\ \text{Tm}\ \Gamma\ \text{A} \rightarrow \text{Tms}\ \Gamma\ (\Gamma \rhd \text{A}) \\
<\text{t}>\ \coloneqq\ \text{id}\ ,\ \textbf{transp}\ (\text{Tm}\ \_)\ (\text{sym [id]Ty})\ \text{t} \\
\_\hat{}\_\ :\ \mathbf{\Pi}\ (\delta : \text{Tms}\ \Delta\ \Gamma)\ \text{A} \rightarrow \text{Tms}\ (\Delta \rhd (\text{A}\ [\ \delta\ ]_{\text{Ty}}))\ (\Gamma \rhd \text{A}) \\
\delta\ \hat{}\ \text{A}\ \coloneqq\ (\delta\ ;\ \text{wk})\ ,\ \textbf{transp}\ (\text{Tm}\ \_)\ [][]\text{Ty vz}
\end{array}$$

We can also prove some derived substitution lemmas, such as how single-substitution commutes with functoriality of context extension.

$$<>\text{-comm}\ :\ (\delta\ \hat{}\ \text{A})\ ;\ <\text{t}\ [\ \delta\ ]> = <\text{t}>\ ;\ \delta$$

We extend our syntax with dependent function types by adding the relevant type former, introduction and elimination rules. We take pointfree/categorical application as primitive.

$$\begin{array}{lll}
\Pi & : & \mathbf{\Pi}\ \text{A} \rightarrow \text{Ty}\ (\Gamma \rhd \text{A}) \rightarrow \text{Ty}\ \Gamma \\
\lambda\_ & : & \text{Tm}\ (\Gamma \rhd \text{A})\ \text{B} \rightarrow \text{Tm}\ \Gamma\ (\Pi\ \text{A B}) \\
\lambda^{-1}\_ & : & \text{Tm}\ \Gamma\ (\Pi\ \text{A B}) \rightarrow \text{Tm}\ (\Gamma \rhd \text{A})\ \text{B} \\
\Pi[] & : & \Pi\ \text{A B}\ [\ \delta\ ]_{\text{Ty}} = \Pi\ (\text{A}\ [\ \delta\ ]_{\text{Ty}})\ (\text{B}\ [\ \delta\ \hat{}\ \text{A}\ ]_{\text{Ty}}) \\
\lambda[] & : & (\lambda\ \text{t})\ [\ \delta\ ] =^{\mathit{Tm_=}\ \boldsymbol{\mathit{refl}}\ \Pi\mathit{[]}}\ \lambda\ (\text{t}\ [\ \delta\ \hat{}\ \text{A}\ ]) \\
\Pi\beta & : & \lambda^{-1}\ \lambda\ \text{t} = \text{t} \\
\Pi\eta & : & \text{t} = \lambda\ \lambda^{-1}\ \text{t}
\end{array}$$

We can derive the more standard rule for application as usual. Interestingly, we can also derive the substitution law for λ^{-1} from $\lambda[]$, $\Pi\beta$ and $\Pi\eta$. For explicit STLC quotiented by β/η equations, we can write essentially the same proof, but of course do not need to worry about accounting for transporting of the term over $\Pi[]$.

$$\begin{array}{l}
\_\cdot\_\ :\ \text{Tm}\ \Gamma\ (\Pi\ \text{A B}) \rightarrow \mathbf{\Pi}\ (\text{u} : \text{Tm}\ \Gamma\ \text{A}) \rightarrow \text{Tm}\ \Gamma\ (\text{B}\ [\ <\text{u}>\ ]_{\text{Ty}}) \\
\text{t} \cdot \text{u}\ \coloneqq\ (\lambda^{-1}\ \text{t})\ [\ <\text{u}>\ ] \\
\lambda^{-1}[]\ :\ (\lambda^{-1}\ \text{t})\ [\ \delta\ \hat{}\ \text{A}\ ] = \lambda^{-1}\ (\textbf{transp}\ (\text{Tm}\ \Delta)\ \Pi[]\ (\text{t}\ [\ \delta\ ])) \\
\lambda^{-1}[]\ \{\text{A} \coloneqq \text{A}\}\ \{\text{t} \coloneqq \text{t}\}\ \{\delta \coloneqq \delta\}\ \coloneqq \\
\quad (\lambda^{-1}\ \text{t})\ [\ \delta\ \hat{}\ \text{A}\ ] \\
\quad = \mathit{by}\ \ \mathit{sym}\ \Pi\beta \\
\quad \lambda^{-1}\ (\lambda\ ((\lambda^{-1}\ \text{t})\ [\ \delta\ \hat{}\ \text{A}\ ])) \\
\quad = \mathit{by}\ \ \mathit{cong}\ \lambda^{-1}\_\ (\mathit{sym[]}\ \lambda\mathit{[]}) \\
\quad \lambda^{-1}\ \textbf{transp}\ (\text{Tm}\ \_)\ \Pi[]\ ((\lambda\ (\lambda^{-1}\ \text{t}))\ [\ \delta\ ]) \\
\quad = \mathit{by}\ \ \mathit{cong}\ (\boldsymbol{\lambda}\ \square \rightarrow \lambda^{-1}\ \boldsymbol{\mathit{transp}}\ (\mathit{Tm}\ \_)\ \Pi\mathit{[]}\ (\square\ [\ \delta\ ]))\ (\mathit{sym}\ \Pi\eta) \\
\quad \lambda^{-1}\ \textbf{transp}\ (\text{Tm}\ \_)\ \Pi[]\ (\text{t}\ [\ \delta\ ])\ \blacksquare
\end{array}$$

We also show how to extend our syntax with Booleans and their dependent elimination rule.

Given the term if A t u v, we call A the *motive* and t the *scrutinee*.

$$
\begin{array}{lll}
\mathbb{B} & : & \mathsf{Ty}\ \Gamma\\
\mathbb{B}[] & : & \mathbb{B}\ [\ \delta\ ]_{\mathsf{Ty}}\ =\ \mathbb{B}\\
\mathsf{tt} & : & \mathsf{Tm}\ \Gamma\ \mathbb{B}\\
\mathsf{ff} & : & \mathsf{Tm}\ \Gamma\ \mathbb{B}\\
\mathsf{if} & : & \mathbf{\Pi}\ (\mathsf{A}\ :\ \mathsf{Ty}\ (\Gamma \rhd \mathbb{B}))\ (\mathsf{t}\ :\ \mathsf{Tm}\ \Gamma\ \mathbb{B})\\
 & & \rightarrow\ \mathsf{Tm}\ \Gamma\ (\mathsf{A}\ [\ <\mathsf{tt}>\ ]_{\mathsf{Ty}})\ \rightarrow\ \mathsf{Tm}\ \Gamma\ (\mathsf{A}\ [\ <\mathsf{ff}>\ ]_{\mathsf{Ty}})\\
 & & \rightarrow\ \mathsf{Tm}\ \Gamma\ (\mathsf{A}\ [\ <\mathsf{t}>\ ]_{\mathsf{Ty}})\\
\mathsf{tt}[] & : & \mathsf{tt}\ [\ \delta\ ]\ =^{\mathit{Tm}_{=}\ \boldsymbol{refl}\ \mathbb{B}[]}\ \mathsf{tt}\\
\mathsf{ff}[] & : & \mathsf{ff}\ [\ \delta\ ]\ =^{\mathit{Tm}_{=}\ \boldsymbol{refl}\ \mathbb{B}[]}\ \mathsf{ff}\\
\mathsf{if}[] & : & \mathsf{if}\ \mathsf{A}\ \mathsf{t}\ \mathsf{u}\ \mathsf{v}\ [\ \delta\ ]\\
 & & =^{\mathit{Tm}_{=}\ \boldsymbol{refl}\ (\mathit{sym}\ \mathit{<>\text{-}commTy} \bullet \mathit{[][]coh}\ \{\mathit{q}\ \coloneqq\ \boldsymbol{refl}\})}\\
 & & \quad \mathsf{if}\ (\mathsf{A}\ [\ \mathbf{transp}\ (\boldsymbol{\lambda}\ \square\ \rightarrow\ \mathsf{Tms}\ (\Delta \rhd \square)\ (\Gamma \rhd \mathbb{B}))\ \mathbb{B}[]\ (\delta\ \hat{}\ \mathbb{B})\ ]_{\mathsf{Ty}})\\
 & & \quad\quad (\mathbf{transp}\ (\mathsf{Tm}\ \Delta)\ \mathbb{B}[]\ (\mathsf{t}\ [\ \delta\ ]))\\
 & & \quad\quad (\mathbf{transp}\ (\mathsf{Tm}\ \Delta)\ (\mathsf{sym}\ \mathsf{<>\text{-}commTy} \bullet \mathsf{[][]coh}\ \{\mathsf{q}\ \coloneqq\ \mathsf{tt}[]\})\ (\mathsf{u}\ [\ \delta\ ]))\\
 & & \quad\quad (\mathbf{transp}\ (\mathsf{Tm}\ \Delta)\ (\mathsf{sym}\ \mathsf{<>\text{-}commTy} \bullet \mathsf{[][]coh}\ \{\mathsf{q}\ \coloneqq\ \mathsf{ff}[]\})\ (\mathsf{v}\ [\ \delta\ ]))\\
\mathbb{B}\beta_1 & : & \mathsf{if}\ \mathsf{A}\ \mathsf{tt}\ \mathsf{u}\ \mathsf{v}\ =\ \mathsf{u}\\
\mathbb{B}\beta_2 & : & \mathsf{if}\ \mathsf{A}\ \mathsf{ff}\ \mathsf{u}\ \mathsf{v}\ =\ \mathsf{v}
\end{array}
$$

So far, while types have been declared to depend on terms, we have no type formers which explicitly rely on this dependency. In my opinion, this set-up makes it a little too easy to "cheat" when writing e.g. normalisation proofs, as such theories can ultimately be compiled into weaker type systems without type-term dependencies [51].

A common way to account for this without adding much complexity [52, 53] is to add universes. Minimally, we can add one type former standing for a universe $\mathsf{U}\ :\ \mathsf{Ty}\ \Gamma$ and embed U-typed terms in Ty Γ with $\mathsf{El}\ :\ \mathsf{Tm}\ \Gamma\ \mathsf{U}\ \rightarrow\ \mathsf{Ty}\ \Gamma$. However, because U cannot contain Π-types (to ensure predicativity[18]), minimised type theories like this are something of a special case. Specifically, in this setting, it is possible to statically compute the "spine" of Πs associated with each type, and use this to (in proofs) justify taking the inductive step from e.g. Π A B to $\mathsf{B}\ [\ <\mathsf{u}>\ ]_{\mathsf{Ty}}$ [52] (i.e. $\mathsf{B}\ [\ <\mathsf{u}>\ ]_{\mathsf{Ty}}$'s spine is guaranteed to be smaller than Π A Bs).

For the type theories that form the basis of modern proof assistants (e.g. Agda), this technique does not work due to the presence of large elimination (recall from Remark 2.1.2 that this is the feature that allows us to generically prove constructor disjointness, among other things). To ensure our proofs generalise to such theories, we therefore add a primitive large elimination rule for Booleans - i.e. type-level "if" expressions.

$$
\begin{array}{lll}
\mathsf{IF} & : & \mathsf{Tm}\ \Gamma\ \mathbb{B}\ \rightarrow\ \mathsf{Ty}\ \Gamma\ \rightarrow\ \mathsf{Ty}\ \Gamma\ \rightarrow\ \mathsf{Ty}\ \Gamma\\
\mathsf{IF}[] & : & \mathsf{IF}\ \mathsf{t}\ \mathsf{A}\ \mathsf{B}\ [\ \delta\ ]_{\mathsf{Ty}}\\
 & = & \mathsf{IF}\ (\mathbf{transp}\ (\mathsf{Tm}\ \Delta)\ \mathbb{B}[]\ (\mathsf{t}\ [\ \delta\ ]))\ (\mathsf{A}\ [\ \delta\ ]_{\mathsf{Ty}})\ (\mathsf{B}\ [\ \delta\ ]_{\mathsf{Ty}})\\
\mathsf{IF\text{-}tt} & : & \mathsf{IF}\ \mathsf{tt}\ \mathsf{A}\ \mathsf{B}\ =\ \mathsf{A}\\
\mathsf{IF\text{-}ff} & : & \mathsf{IF}\ \mathsf{ff}\ \mathsf{A}\ \mathsf{B}\ =\ \mathsf{B}
\end{array}
$$

We could go further, and add a recursive large elimination rule e.g. for ℕs, but I think IF provides a nice balance between forcing us to demonstrate how to account for large elimination without adding too much extra complexity.

We also show how extend the syntax with a propositional identity type $\mathsf{Id}\ \mathsf{A}\ \mathsf{t}_1\ \mathsf{t}_2$. Elements of this type are introduced with reflexivity and eliminated with the J rule (*path induction*).

$$
\begin{array}{lll}
\mathsf{Id} & : & \mathbf{\Pi}\ \mathsf{A}\ \rightarrow\ \mathsf{Tm}\ \Gamma\ \mathsf{A}\ \rightarrow\ \mathsf{Tm}\ \Gamma\ \mathsf{A}\ \rightarrow\ \mathsf{Ty}\ \Gamma\\
\mathsf{refl} & : & \mathsf{Tm}\ \Gamma\ (\mathsf{Id}\ \mathsf{A}\ \mathsf{t}\ \mathsf{t})
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{Id}[] & : & \mathsf{Id}\ \mathsf{A}\ \mathsf{t}_1\ \mathsf{t}_2\ [\ \delta\ ]_{\mathsf{Ty}}\ =\ \mathsf{Id}\ (\mathsf{A}\ [\ \delta\ ]_{\mathsf{Ty}})\ (\mathsf{t}_1\ [\ \delta\ ])\ (\mathsf{t}_2\ [\ \delta\ ])\\
\mathbf{refl}[] & : & \mathsf{refl}\ \{\mathsf{t}\ \coloneqq\ \mathsf{t}\}\ [\ \delta\ ]\ =^{\mathit{Tm}_{=}\ \boldsymbol{refl}\ \mathit{Id}[]}\ \mathsf{refl}
\end{array}
$$

[51]: Barras et al. (1997), *Coq in Coq*

[52]: Danielsson (2006), *A Formalisation of a Dependently Typed Language as an Inductive-Recursive Family*

[53]: Altenkirch et al. (2016), *Type theory in type theory using quotient inductive types*

[18] 18: To prevent Russel's paradox, it is important that Π-types always be placed in larger universes than their domain or range.

In a type theory with a hierarchy of universes, we could implement dependent and large elimination with the same primitive by generalising the motive of "if" to a type of any universe level.

```
J    : Π (B : Ty (Γ ▷ A ▷ Id (A [ wk ]Ty) (t₁ [ wk ]) vz))
          (p : Tm Γ (Id A t₁ t₂))
       → Tm Γ (B [ < t₁ > , transp (Tm Γ) wkvz<>Id refl ]Ty)
       → Tm Γ (B [ < t₂ > , transp (Tm Γ) wkvz<>Id p ]Ty)
Idβ  : J B refl t = t
```

```
J[] : J {Γ ≔ Γ} {A ≔ A} {t₁ ≔ u₁} {t₂ ≔ u₂} B p t [ δ ]
    = Tm₌ refl <>,-comm
      J (B [ transp (λ □ → Tms (Δ ▷ _ ▷ □) _) wk-commId
        ((δ ^ A) ^ Id (A [ wk ]Ty) (u₁ [ wk ]) vz) ]Ty)
        (transp (Tm Δ) Id[] (p [ δ ]))
        (transp (Tm Δ) <>,-comm′ (t [ δ ]))
```

Given the term J B p t, we call B the *motive* and p the *scrutinee.*

We can recover transporting (i.e. *indiscernibility-of-identicals*) from J by weakening the motive.

```
transp : Π (B : Ty (Γ ▷ A)) → Tm Γ (Id A t₁ t₂)
         → Tm Γ (B [ < t₁ > ]Ty) → Tm Γ (B [ < t₂ > ]Ty)
transp B p t
  ≔ transp (Tm _) wk;Ty (J (B [ wk ]Ty) p (transp (Tm _) (sym wk;Ty) t))
```

Equality in Type Theory

Both our metatheory (Agda) and the syntax-so-far are examples of *intensional* type theory (ITT). Equality judgements are divided into *definitional* (in Agda, denoted with _≡_) and *propositional* (in Agda, denoted by _=_). As we have quotiented our syntax by conversion, definitional equality in our object theory corresponds to propositional equality in the meta, _=_, while propositional equality is represented with the Id type former.

The key idea behind this division is that deciding propositional equality in general requires arbitrary proof search (and so is undecidable), so definitional equality carves out a decidable subset of propositional equality which the typechecker can feasibly automate.

While ITT is the foundation of many modern proof assistants/dependently typed PLs, including Rocq [38], Lean [37] and Idris [54] as well as Agda, it is not the only option. Our type theory can be turned into an extensional type theory (ETT) by adding the *equality reflection* rule:

```
reflect : Tm Γ (Id A t₁ t₂) → t₁ = t₂
```

[38]: The Rocq Team (2025), *The Rocq Reference Manual – Release 9.0*

[37]: Moura et al. (2021), *The Lean 4 Theorem Prover and Programming Language*

[54]: Brady (2021), *Idris 2: Quantitative Type Theory in Practice*

It is perhaps interesting to note that equality reflection is exactly the converse of the introduction rule for Id (up to _=_):

```
rfl′ : t₁ = t₂ → Tm Γ (Id A t₁ t₂)
rfl′ refl ≔ refl
```

So, both of these rules together have the effect of making propositional and definitional equality equivalent.

ETT loses decidable typechecking, but practical proof assistants can still in theory be built upon it by allowing the user to explicitly write out typing/conversion derivations.

On the other end of the spectrum is weak type theory (WTT) [55], where definitional equality is left as pure syntactic equality and β/η laws are dealt with via primitive operations returning propositional equalities.

Even within ITT, there is still quite a large design-space in how to treat equality. For example:

- Whether definitional equality only encompasses β laws or if certain η laws are admitted also [22, 56].

[55]: Winterhalter (2020), *Formalisation and meta-theory of type theory*

[22]: Kovács (2025), *Eta conversion for the unit type*

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

- Whether propositional uniqueness-of-identity-proofs (UIP) holds

  ```
  uip : Π (p : Tm Γ (Id A t t)) → Tm Γ (Id (Id A t t) p refl)
  ```

  Or equivalently, as *axiom K*

  ```
  K : Π (B : Ty (Γ ▷ Id A t t)) (p : Tm Γ (Id A t t))
    → Tm Γ (B [ < refl > ]Ty) → Tm Γ (B [ < p > ]Ty)
  ```

- Whether (propositional) function extensionality is supported

  ```
  funext : Tm (Γ ▷ A) (Id B (λ⁻¹ t₁) (λ⁻¹ t₂))
         → Tm Γ (Id (Π A B) t₁ t₂)
  ```

  as in OTT and □TT.
- Whether equality at the level of types (i.e. in a type theory with universes) is relaxed to that of *equivalences* (and is therefore computationally relevant, contradicting UIP) as in □TT.

etc...

2.3.2 Soundness

Soundness of dependent type theory can be shown very similarly to STLC - we construct the standard model. Rather than adding a dedicated empty type, we show that Tm • (Id 𝔹 tt ff) is uninhabited.

```
sound : ¬ Tm • (Id 𝔹 tt ff)
```

The main differences are:

- Types are now interpreted as functions from environments to **Type** (so terms become dependent functions)
- We need to mutually show soundness of interpretation w.r.t. conversion. Conveniently, all conversion equations hold definitionally in the model (:≡ **refl**) so we skip over them in the below presentation.

```
⟦Ctx⟧ : Type₁
⟦Ctx⟧ :≡ Type

⟦Ty⟧ : ⟦Ctx⟧ → Type₁
⟦Ty⟧ Γ :≡ Γ → Type

⟦Tm⟧ : Π Γ → ⟦Ty⟧ Γ → Type
⟦Tm⟧ Γ A :≡ Π ρ → A ρ

⟦Tms⟧ : ⟦Ctx⟧ → ⟦Ctx⟧ → Type
⟦Tms⟧ Δ Γ :≡ Δ → Γ
```

```
⟦_⟧Ctx : Ctx → ⟦Ctx⟧
⟦_⟧Ty  : Ty Γ → ⟦Ty⟧ ⟦ Γ ⟧Ctx
⟦_⟧Tm  : Tm Γ A → ⟦Tm⟧ ⟦ Γ ⟧Ctx ⟦ A ⟧Ty
⟦_⟧Tms : Tms Δ Γ → ⟦Tms⟧ ⟦ Δ ⟧Ctx ⟦ Γ ⟧Ctx
```

Note that for type-level (large) IF, we can use 𝔹's recursor, while for term-level (dependent) "if", we need to use the dependent eliminator.

```
⟦ •     ⟧Ctx :≡ 𝟙
⟦ Γ ▷ A ⟧Ctx :≡ Σ ⟦ Γ ⟧Ctx ⟦ A ⟧Ty

⟦ 𝔹         ⟧Ty :≡ λ ρ → 𝔹
⟦ Id A t₁ t₂ ⟧Ty :≡ λ ρ → ⟦ t₁ ⟧Tm ρ = ⟦ t₂ ⟧Tm ρ
```

```
⟦ Π A B      ⟧Ty ≔ λ ρ → Π uⱽ → ⟦ B ⟧Ty (ρ , uⱽ)
⟦ A [ δ ]Ty ⟧Ty ≔ λ ρ → ⟦ A ⟧Ty (⟦ δ ⟧Tms ρ)
⟦ IF t A B   ⟧Ty ≔ λ ρ → 𝔹-rec (⟦ t ⟧Tm ρ) (⟦ A ⟧Ty ρ) (⟦ B ⟧Ty ρ)

⟦ π₁ δ  ⟧Tms ≔ λ ρ → ⟦ δ ⟧Tms ρ .π₁
⟦ id    ⟧Tms ≔ λ ρ → ρ
⟦ ε     ⟧Tms ≔ λ ρ → ⟨⟩
⟦ δ , t ⟧Tms ≔ λ ρ → ⟦ δ ⟧Tms ρ , ⟦ t ⟧Tm ρ
⟦ δ ; σ ⟧Tms ≔ λ ρ → ⟦ δ ⟧Tms (⟦ σ ⟧Tms ρ)

⟦ λ t       ⟧Tm ≔ λ ρ   uⱽ   → ⟦ t ⟧Tm (ρ , uⱽ)
⟦ λ⁻¹ t     ⟧Tm ≔ λ (ρ , uⱽ) → ⟦ t ⟧Tm ρ uⱽ
⟦ tt        ⟧Tm ≔ λ ρ        → tt
⟦ ff        ⟧Tm ≔ λ ρ        → ff
⟦ t [ δ ]   ⟧Tm ≔ λ ρ        → ⟦ t ⟧Tm (⟦ δ ⟧Tms ρ)
⟦ π₂ δ      ⟧Tm ≔ λ ρ        → ⟦ δ ⟧Tms ρ .π₂
⟦ if A t u v ⟧Tm
  ≔ λ ρ → 𝔹-elim (λ b → ⟦ A ⟧Ty (ρ , b)) (⟦ t ⟧Tm ρ)
                 (⟦ u ⟧Tm ρ) (⟦ v ⟧Tm ρ)
⟦ J B p t   ⟧Tm
  ≔ λ ρ → =-elim (λ ⟦u⟧ ⟦p⟧ → ⟦ B ⟧Ty ((ρ , ⟦u⟧) , ⟦p⟧))
                 (⟦ p ⟧Tm ρ) (⟦ t ⟧Tm ρ)
```

```
tt/ff-disj : ¬ tt = ff
tt/ff-disj ()

sound t ≔ tt/ff-disj (⟦ t ⟧Tm ⟨⟩)
```

2.3.3 From Quotients to Setoids

As previously mentioned in Section 1.1 - Equivalence Relations, Quotients and Setoids, support for quotient types in modern proof assistants is somewhat hit-or-miss. Quotienting by conversion also prevents us from performing more fine-grained "intensional" analysis on terms [59] or using more "syntactic" proof techniques such as reduction. Therefore, when mechanising in Agda, we prefer to work with setoids rather than QIITs directly.

In a two-level metatheory [57] it is possible to simultaneously work with quotients up to equivalence when convenient and then go down to a raw syntactic level when required. The key idea behind 2LTT is to have both an *inner* and *outer* propositional equality, which differ in their degrees of extensionality. Indeed some exploration has been done on using this framework to formalise *elaboration* [58], a somewhat inherently syntactic algorithm.

2LTT also comes with some restrictions on eliminators mapping between the two levels though, which I expect to be problematic in proving e.g. strong normalisation. A pertinent question arises here: why not just scrap intrinsically-typed syntax and use inductive typing relations on untyped terms? Perhaps if our *only* aim was proving strong normalisation, this would be a sensible course of action.

We follow essentially the translation as outlined in [59]. Contexts become a setoid, types become a setoid fibration on contexts, substitutions become a setoid fibration on pairs of contexts and terms become a setoid fibration on types paired with their contexts.

We start by declaring the equivalence relations. We place these in a universe of strict propositions **Prop** for convenience.

```
data Ctx~  : Ctx → Ctx → Prop
data Ty~   : Ctx~ Γ₁ Γ₂ → Ty Γ₁ → Ty Γ₂ → Prop
data Tm~   : Π Γ~ → Ty~ Γ~ A₁ A₂ → Tm Γ₁ A₁ → Tm Γ₂ A₂
             → Prop
data Tms~  : Ctx~ Δ₁ Δ₂ → Ctx~ Γ₁ Γ₂ → Tms Δ₁ Γ₁ → Tms Δ₂ Γ₂
             → Prop
```

We add constructors to these relations corresponding to equivalence, congruence and computation (the latter of correspond to the propositional equations that we explicitly quotient by in a QIIT syntax).

```
data Ty~ where
  -- Equivalence
  rfl~ : Ty~ rfl~ A A
  sym~ : Ty~ Γ~ A₁ A₂ → Ty~ (sym~ Γ~) A₂ A₁
```

[57]: Annenkov et al. (2023), *Two-level type theory and applications*
[58]: Kovács (2024), *Basic setup for formalizing elaboration*
[59]: Kovács (2022), *Staged compilation with two-level type theory*

```
_•~_  : Ty~ Γ₁₂~ A₁ A₂ → Ty~ Γ₂₃~ A₂ A₃ → Ty~ (Γ₁₂~ •~ Γ₂₃~) A₁ A₃
-- Congruence
𝔹~    : Ty~ Γ~ 𝔹 𝔹
Π~    : Π A~ → Ty~ (Γ~ ▷~ A~) B₁ B₂ → Ty~ Γ~ (Π A₁ B₁) (Π A₂ B₂)
_[_]~ : Π (A~ : Ty~ Γ~ A₁ A₂) (δ~ : Tms~ Δ~ Γ~ δ₁ δ₂)
        → Ty~ Δ~ (A₁ [ δ₁ ]Ty) (A₂ [ δ₂ ]Ty)
IF~   : Tm~ Γ~ 𝔹~ t₁ t₂ → Ty~ Γ~ A₁ A₂ → Ty~ Γ~ B₁ B₂
        → Ty~ Γ~ (IF t₁ A₁ B₁) (IF t₂ A₂ B₂)
-- Computation
IF-TT~ : Ty~ rfl~ (IF tt A B) A
IF-FF~ : Ty~ rfl~ (IF ff A B) B
𝔹[]~   : Ty~ rfl~ (𝔹 [ δ ]Ty) 𝔹
Π[]~   : Ty~ rfl~ (Π A B [ δ ]Ty) (Π (A [ δ ]Ty) (B [ δ ^ A ]Ty))
[id]~  : Ty~ rfl~ (A [ id ]Ty) A
[][]~  : Ty~ rfl~ (A [ δ ]Ty [ σ ]Ty) (A [ δ ; σ ]Ty)
```

We are missing the computation rule for substitutions applied to IF:

```
IF[] : IF t A B [ δ ]Ty
     = IF (transp (Tm Δ) 𝔹[] (t [ δ ])) (A [ δ ]Ty) (B [ δ ]Ty)
```

The transport here is essential. t [δ] only has type 𝔹 [δ]$_{\mathsf{Ty}}$, but IF requires a term of type 𝔹. Typeability in dependent type theory must account for conversion. We can achieve this by adding constructors to each indexed sort (Ty, Tm and Tms) corresponding to coercion over the equivalence:

```
coeTy  : Ctx~ Γ₁ Γ₂ → Ty Γ₁ → Ty Γ₂
coeTm  : Π Γ~ → Ty~ Γ~ A₁ A₂ → Tm Γ₁ A₁ → Tm Γ₂ A₂
coeTms : Ctx~ Δ₁ Δ₂ → Ctx~ Γ₁ Γ₂ → Tms Δ₁ Γ₁ → Tms Δ₂ Γ₂
```

IF[]~ can now be written with an explicit coercion on the scrutinee:

```
if[]~ : Ty~ rfl~ (IF t A B [ δ ]Ty)
                 (IF (coeTm rfl~ 𝔹[]~ (t [ δ ])) (A [ δ ]Ty) (B [ δ ]Ty))
```

The final ingredient to make this work is *coherence*: coercion must respect the equivalence.

```
cohTy  : Ty~  Γ~ A (coeTy Γ~ A)
cohTms : Tms~ Δ~ Γ~ δ (coeTms Δ~ Γ~ δ)
cohTm  : Tm~  Γ~ A~ t (coeTm Γ~ A~ t)
```

2.3.4 Strictification

Whether quotiented or based on setoids, explicit-substitution syntaxes can be painful to work with in practice. We have already seen how many of the substitution laws for terms require manual coercion over the corresponding laws for types, e.g.

```
if[] : if A t u v [ δ ]
     = Tm= refl (sym <>-commTy • [][]coh {q ≔ refl})
       if (A [ transp (λ □ → Tms (Δ ▷ □) (Γ ▷ 𝔹)) 𝔹[] (δ ^ 𝔹) ]Ty)
          (transp (Tm Δ) 𝔹[] (t [ δ ]))
          (transp (Tm Δ) (sym <>-commTy • [][]coh {q ≔ tt[]}) (u [ δ ]))
          (transp (Tm Δ) (sym <>-commTy • [][]coh {q ≔ ff[]}) (v [ δ ]))
```

If substitution instead computed recursively, $\mathbb{B}[]\ :\ \mathbb{B}\ [\ \delta\ ]_{\mathsf{Ty}}\ =\ \mathbb{B}$, $\mathsf{tt}[]\ :\ \mathsf{tt}\ [\ \delta\ ]\ =\ \mathsf{tt}$ and $\mathsf{ff}[]\ :\ \mathsf{ff}\ [\ \delta\ ]\ =\ \mathsf{ff}$ would hold definitionally, enabling the substantially simpler

```
if[] : if A t u v [ δ ]
     = Tm= refl (sym (<>-commTy { B ≔ A }))
       if (A [ δ ^ 𝔹 ]Ty) (t [ δ ])
          (transp (Tm Δ) (sym (<>-commTy {B ≔ A})) (u [ δ ]))
          (transp (Tm Δ) (sym (<>-commTy {B ≔ A})) (v [ δ ]))
```

Of course, the rule still requires some transport to account for commuting of substitutions

```
<>-commTy : B [ δ ^ A ]Ty [ < t [ δ ] > ]Ty = B [ < t > ]Ty [ δ ]Ty
```

which does not hold by mere computation. If somehow this law were made strict as well, we could write the substitution law for "if" as

```
if[] :  if A t u v [ δ ]
     = if (A [ δ ^ 𝔹 ]Ty) (t [ δ ]) (u [ δ ])) (v [ δ ]))
```

This excessive transporting can get especially painful when constructing displayed models of syntax[19], e.g. when proving properties like canonicity or normalisation. Issues of this sort were severe enough that the Agda mechanisation of [60] was never fully finished.

Luckily, there has been some significant progress recently towards taking a well-understood explicit substitution syntax as primitive and then *strictifying* various substitution equations, as to construct something easier to work with. [61] illustrates one strategy towards achieving this, where operations intended to compute are redefined recursively and then a new induction principle is derived which refers to these recursive operations.

Unfortunately, while this approach can make substitution equations arising from direct computation such as $\mathbb{B}\ [\ \delta\ ]_{\mathsf{Ty}}\ =\ \mathbb{B}$ definitional, the functor laws remain propositional. [7] presents a much more involved construction based on presheaves, in which all substitution laws, except the η law for context extension $\triangleright\eta\ :\ \delta\ =\ \pi_1\ \delta\ ,\ \pi_2\ \delta$ / $\mathsf{id}\triangleright\ :\ \mathsf{id}\ \{\Gamma\ \coloneqq\ \Gamma \triangleright \mathsf{A}\}\ =\ \mathsf{wk}\ ,\ \mathsf{vz}$, are eventually strictified. When implemented in Agda, both approaches only allow induction via explicit eliminators, rather than pattern matching.

Some proof assistants also support reflecting a subset propositional equations into definitional ones via global REWRITE rules (e.g. Dedukti [62], Agda [11] and Rocq [12]). Global rewrite rules can be though of a restricted version of equality reflection from extensional type theory (in which transports/coercions are fully relegated to the typing derivations), and [63–65] show that ETT is ultimately conservative over ITT.

So, if we start with a QIIT definition of type theory, we have few possible routes towards strictifying equations. There remain problems though:

- ► Strictification can produce a more convenient induction principle for the syntax, but this is still just an induction principle. Directly-encoded inductive-recursive types in Agda allow for pattern matching, which is often more convenient (e.g. when pattern matching, we do not have to explicitly give cases for how to interpret the recursive operations).
- ► As mentioned in the previous section, Agda's support for quotient types is somewhat unsatisfactory, so we would rather use setoids. Rewriting via setoid equations is unsound (because setoid constructors are still provably disjoint w.r.t. propositional equality).
- ► Rewrite rules as implemented in Agda struggle somewhat with indexed types [#7602].

[19] In other words, *inducting* on syntax rather than merely *recursing*.

[60]: Altenkirch et al. (2017), *Normalisation by Evaluation for Type Theory, in Type Theory*

[61]: Kaposi (2023), *Towards quotient inductive-inductive-recursive types*

[7]: Kaposi et al. (2025), *Type Theory in Type Theory Using a Strictified Syntax*

[62]: Assaf et al. (2023), *Dedukti: a Logical Framework based on the λΠ-Calculus Modulo Theory*

[11]: Cockx (2019), *Type Theory Unchained: Extending Agda with User-Defined Rewrite Rules*

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

[63]: Hofmann (1995), *Conservativity of Equality Reflection over Intensional Type Theory*

[64]: Oury (2005), *Extensionality in the Calculus of Constructions*

[65]: Winterhalter et al. (2019), *Eliminating reflection from type theory*

[#7602]: Burke (2024), *Associativity of vector concatenation REWRITE sometimes doesn't apply*

The ultimate goal of this project is to explore new type theories with local equational assumptions, not to provide a watertight Agda mechanisation. Therefore, in the proofs of normalisation, where, frankly, we need all the help we can get, I axiomatise *strict*, implicit-substitution syntaxes, using a combination of POSTULATEs, REWRITE rules, NON_TERMINATING and NON_COVERING definitions, and even a new flag which re-enables [#6643] (these are of course very unsafe features, but the idea is to simulate working in a "nicer" metatheory where "transport-hell" is less of an issue). Critically, while substitution is strict, we still deal with β/η convertibility via an explicit equivalence relation, so the syntax remains setoid-based.

[#6643]: Liao (2023), *#6643: Rewrite rules are allowed in implicit mutual blocks*

For presentation in the report, going over the entire syntax of dependent type theory again, switching _ = _ signs to _ ≡ _ is probably not a super valuable use of anyone's time. I will quickly given the definition of variables though, given these are new to the strict presentation (though very similar to STLC).

```
data Var where
  coeVar : Π Γ~ → Ty~ Γ~ A₁ A₂ → Var Γ₁ A₁ → Var Γ₂ A₂
  vz : Var (Γ , A) (A [ wk ]Ty)
  vs : Var Γ B → Var (Γ , A) (B [ wk ]Ty)
```

We also return to *pointful* application:

```
_·_ : Tm Γ (Π A B) → Π (u : Tm Γ A) → Tm Γ (B [ < u > ]Ty)
```

2.4 Normalisation by Evaluation

Normalisation by Evaluation (NbE) [66, 67] is a normalisation algorithm for lambda calculus terms, which operates by first evaluating terms into a semantic domain (specifically, the *presheaf model*), and then inverting the evaluation function to *quote* back normal forms. It can be motivated from multiple directions:

[66]: Berger et al. (1991), *An Inverse of the Evaluation Functional for Typed lambda-calculus*
[67]: Altenkirch et al. (1995), *Categorical Reconstruction of a Reduction Free Normalization Proof*

- **No reliance on small-step reductions:** NbE is structurally recursive, and is therefore not reliant on a separate strong normalisation result to justify termination. This can be especially useful in settings where a strongly normalising set of small-step reductions is difficult to identify (e.g. dealing with η-expansion).
- **Applicability to quotiented syntax:** Following on from the first point, unlike term-rewriting-based approaches to normalisation, NbE does not rely on distinguishing $\beta\eta$-convertible terms (the algorithm can be structured in such a way as to simply map families of convertible terms to values [60]).
- **Efficiency:** NbE avoids repeated expensive single-substitutions (which need to traverse the whole syntax tree each time to possibly replace variables with the substitute) [68]. Instead, the mappings between variables and semantic values are tracked in a persistent map (the *environment*), such that variables can be looked up exactly when they are evaluated.

[60]: Altenkirch et al. (2017), *Normalisation by Evaluation for Type Theory, in Type Theory*

[68]: Kovács (2023), *smalltt*

This all means that NbE is useful both as a technique to prove normalisation for type theory, and as an algorithm in typechecker implementations for deciding convertibility of types. We will use NbE for both purposes in this project, and will discuss the shortcuts we can take when implementing NbE in a partial programming language (specifically Haskell) in (Section 5.4).

To introduce NbE, we will begin by deriving the algorithm for the the recursive substitution STLC syntax given in Section 2.2.1, and sketch how to prove its correctness. We will then extend the technique to dependent type theory following [60].

[60]: Altenkirch et al. (2017), *Normalisation by Evaluation for Type Theory, in Type Theory*

2.4.1 Naive Normalisation

As a warm-up to NbE, we will start by implementing "naive" normalisation, i.e. recursing on a term, contracting β-redexes where possible by applying single-substitutions. Using this approach, termination can only be justified by a separate strong normalisation result.

We first define our goal: β-normal forms, Nf Γ A, inductively (mutually recursively with stuck, neutral terms, Ne Γ A) along with the obvious injections back into ordinary terms, ⌜_⌝, ⌜_⌝ne.

To enforce η-normality for →, × and 𝟙, we could restrict embedded neutrals in Nf to only those of positive types, 𝟘 and +. $\beta\eta$-normal forms accounting for positive types more complicated [69] (and actually $\beta\eta$ normalisation for STLC with positive inductive types like ℕ is undecidable).

[69]: Scherer (2017), *Deciding equivalence with sums and the empty type*

```
data Ne : Ctx → Ty → Type
data Nf : Ctx → Ty → Type
data Ne where
  `_   : Var Γ A → Ne Γ A
  _·_  : Ne Γ (A → B) → Nf Γ A → Ne Γ B
  π₁   : Ne Γ (A × B) → Ne Γ A
  π₂   : Ne Γ (A × B) → Ne Γ B
  case : Ne Γ (A + B) → Nf (Γ ▷ A) C → Nf (Γ ▷ B) C → Ne Γ C
data Nf where
  ne   : Ne Γ A → Nf Γ A
  λ_   : Nf (Γ ▷ A) B → Nf Γ (A → B)
  _,_  : Nf Γ A → Nf Γ B → Nf Γ (A × B)
  ⟨⟩   : Nf Γ 𝟙
  in₁  : Π B → Nf Γ A → Nf Γ (A + B)
  in₂  : Π A → Nf Γ B → Nf Γ (A + B)
```

```
⌜_⌝Nf  :  Nf Γ A  →  Tm Γ A
⌜_⌝Ne  :  Ne Γ A  →  Tm Γ A
```

We can now attempt to define normalisation by direct recursion on terms, relying on substitution to contract β-redexes. For the rest of this section, we will restrict our attention to the cases for _ → _ types, for simplicity.

```
norm  :  Tm Γ A  →  Nf Γ A
nf-app  :  Nf Γ (A  →  B)  →  Nf Γ A  →  Nf Γ B

norm (λ t)     ≔  λ (norm t)
norm (t · u)  ≔  nf-app (norm t) (norm u)

nf-app (ne t) u  ≔  ne (t · u)
nf-app (λ t)   u  ≔  norm (⌜ t ⌝Nf [ < ⌜ u ⌝Nf > ])
```

In a partial language, when applied to normalising terms, this definition is works! The single substitutions are less efficient on terms with multiple β-redexes than the NbE approach of tracking all variable mappings in a single environment, but with effort, it can be optimised (e.g. we could annotate subterms with the sets of variables that are actually used, to avoid unnecessary traversals during substitution).

Note that normal forms are not stable under substitution (i.e. substitution can create new β-redexes), so we cannot easily define substitution on normal forms to resolve this. It is perhaps worth mentioning though, that if one is more careful with the representation of neutral spines (among other things), pushing in this direction can lead to another structurally recursive normalisation algorithm known as *hereditary substitution* [70]. Unfortunately, it is currently unknown whether this technique scales to dependent types.

In a total setting, unfortunately, naive normalisation is clearly not well-founded by structural recursion. ⌜ norm t ⌝Nf [< ⌜ norm u ⌝Nf >] is not structurally smaller than t · u.

Making naive normalisation total relies on a strong normalisation result: we need to know that β-reduction, _>$_\beta$_, is well-founded. Actually, we will make use of the accessibility of typed terms w.r.t. interleaved structural ordering, _>s_, and β-reduction, but luckily obtaining this from traditional strong normalisation is not too difficult [71]. Note that _>$_\beta$_ commutes with _>s_ in the sense that

[70]: Keller et al. (2010), *Hereditary Substitutions for Simple Types, Formalized*

```
|t >s u  →  u >β v  →  (w : Tm Γ) × t >β w × w >s v|
```

or as a diagram:

$$\begin{tikzcd}
t \arrow[r, "\_{>}s\_"] \arrow[d, dashed, "\_{>}\beta\_"'] & u \arrow[d, "\_{>}\beta\_"] \\
w \arrow[r, dashed, "\_{>}s\_"'] & v
\end{tikzcd}$$

Definition 2.4.1 (Accessibility) Classically, strong normalisation can be defined as there existing no infinite chains of reductions. To induct w.r.t. reduction order constructively, we instead use accessibility predicates.

```
data Acc (_>_  :  A  →  A  →  Type)
         (x  :  A)  :  Type where
  acc  :  (Π {y}  →  x > y
            →  Acc _>_ y)
          →  Acc _>_ x
```

Intuitively, Acc _>_ x can be thought of as the type of finite-depth trees starting at x, with branches corresponding to single steps along _>_ and minimal elements w.r.t. _>_ at the leaves.

We use SN as a synonym for Acc when the ordering is a small-step reduction relation that proceeds underneath abstractions.

We therefore skip ahead to defining a single _>βs_ relation on terms encompassing both structural and reduction orderings, and assume we have a proof that this combined order is well-founded.

```
data _>s_  :  Tm Γ A  →  Tm Δ B  →  Type where
  l·>  :  t · u >s t
  ·r>  :  t · u >s u
  λ>   :  λ t >s t

data _>βs_  :  BTm  →  BTm  →  Type where
  β>  :  t >β u  →  ⟪ t ⟫ >βs ⟪ u ⟫
  s>  :  t >s u  →  ⟪ t ⟫ >βs ⟪ u ⟫

-- All terms are strongly normalisable w.r.t. _>βs_
wf  :  Π (t  :  Tm Γ A)  →  SN _>βs_ ⟪ t ⟫
```

Normalisation can then be made total by consistently returning evidence that there exists a (possibly empty) chain of reductions _>$_\beta{}^*$_ to go from the input term to the resulting normal form.

We denote the transitive closure and reflexive-transitive closures of orders with $^+$ and * respectively.

```
Nf> : Π Γ A → Tm Γ A → Type
Nf> Γ A t ≔ (t^Nf : Nf Γ A) × (t >_β* ⌜ t^Nf ⌝Nf)
```

Actually using our accessibility predicate to justify naive normalisation gets quite cluttered, but the main idea is to ensure that we are always making progress with respect to _>βs_.

```
norm   : Π (t : Tm Γ A) → SN _>βs⁺_ ⟪ t ⟫ → Nf> Γ A t
nf-app : Π (t^Nf : Nf Γ (A → B)) (u^Nf : Nf Γ A)
         → SN _>βs⁺_ ⟪ t · u ⟫ → t · u >_β* ⌜ t^Nf ⌝Nf · ⌜ u^Nf ⌝Nf
         → Nf> Γ B (t · u)

norm (` i)   a ≔ ne (` i) , rfl*
norm (λ t)   (acc a)
  using t^Nf  , t>t^Nf ≔ norm t (a ⟪ s> λ> ⟫)
  ≔ (λ t^Nf) , λ× t>t^Nf
norm (t · u) (acc a)
  using t^Nf  , t>t^Nf  ≔ norm t (a ⟪ s> l·> ⟫)
     |  u^Nf , u>u^Nf  ≔ norm u (a ⟪ s> ·r> ⟫)
  ≔ nf-app t^Nf u^Nf (acc a) (t>t^Nf ·* u>u^Nf)

nf-app (ne t) u _ tu>tu^Nf
  ≔ ne (t · u) , tu>tu^Nf
nf-app (λ t)  u (acc a) rfl*
  using tu^Nf , tu>tu^Nf ≔ norm (⌜ t ⌝Nf [ < ⌜ u ⌝Nf > ]) (a ⟪ β>  →β ⟫)
  ≔ tu^Nf , →β :: tu>tu^Nf
nf-app (λ t)  u (acc a) (p :: q)
  using tu^Nf , tu>tu^Nf ≔ norm (⌜ t ⌝Nf [ < ⌜ u ⌝Nf > ])
                                (a (β> p ::⁺ (map* _ β> q ∘* ⟪ β>  →β ⟫*)))
  ≔ tu^Nf , (p :: q ∘* ⟪  →β ⟫* ∘* tu>tu^Nf)

normalise : Tm Γ A → Nf Γ A
normalise t ≔ norm t (sn⁺ (wf t)) .π₁
```

Soundness and completeness of normalise follows from equivalence between algorithmic and declarative conversion (completeness relies on confluence of reduction).

From the Standard Model to Presheaves

To derive a structurally-recursive normalisation algorithm, our attention be focused on the case for application. Recall that when aiming to produce Nf Γ As directly by recursion on our syntax, we failed to derive a structurally recursive algorithm because there is no analogue of _·_ : Tm Γ (A → B) → Tm Γ A → Tm Γ B on normal forms.

For inspiration on how to solve this, we recall the definition of the standard model. There, we were able to write a structurally-recursive interpreter for closed terms by interpreting object-level functions, abstractions and applications into their meta-level counterparts. E.g. we implemented application in the model merely with meta-level application (plus threading of environments.)

$$\llbracket t \cdot u \rrbracket^{Tm}\ \rho \ \coloneqq\ (\llbracket t \rrbracket^{Tm}\ \rho)\ (\llbracket u \rrbracket^{Tm}\ \rho)$$

We cannot recover normalisation from the standard model, however. Without an environment of closed values to evaluate with respect to, we cannot hope to inspect the structure of evaluated terms (i.e. meta-level functions like $\llbracket\ \Gamma\ \rrbracket^{Ctx} \rightarrow \llbracket\ A\ \rrbracket^{Ty}$ are opaque). Similarly, even with an environment, we cannot inspect the structure of interpreted →-typed values beyond testing their behaviour on particular inputs given these are again opaque meta-language functions. The “problem” we are encountering is that our values have no first-order representation of variables.

It turns out, by carefully defining a similar model, based on presheaves, we can embed stuck, first-order variables into values[20] , implement evaluation in open contexts and, critically, *invert* evaluation, *quoting* back into normalised first-order terms (i.e. our normal forms). This *evaluation* followed by *quoting* is exactly normalisation by evaluation.

2.4.2 The Presheaf Model

Central to the presheaf model (perhaps unsurprisingly) is the concept of a presheaf: contravariant functors into **Type** (Definition 2.2.6). We actually have a choice about which category to take presheaves over, with the key restrictions being that it must be a subset of substitutions, normal/neutral forms must be stable w.r.t. it and it must include the single-weakening $\mathsf{wk} : \mathsf{Tms}\ (\Gamma \rhd A)\ \Gamma$ (we will see why these latter two restrictions are important later). The two standard choices are renamings $\mathsf{Ren}\ \Delta\ \Gamma$, which we have seen already, and thinnings, $\mathsf{Thin}\ \Delta\ \Gamma$. We will use thinnings (also known as order-preserving embeddings) because type theories we will consider later in this report will actually not feature renaming-stable normal/neutral forms (Remark 6.2.1).

We define thinnings concretely as

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{data}\ \mathsf{Thin} : \mathsf{Ctx} \to \mathsf{Ctx} \to \textbf{Type where}} \\
\quad \varepsilon & : & \mathsf{Thin}\ \bullet\ \bullet \\
\quad \_{}^{\wedge\mathsf{Th}}\_ & : & \mathsf{Thin}\ \Delta\ \Gamma \to \Pi\ \mathsf{A} \to \mathsf{Thin}\ (\Delta \rhd \mathsf{A})\ (\Gamma \rhd \mathsf{A}) \\
\quad \_{}^{+\mathsf{Th}}\_ & : & \mathsf{Thin}\ \Delta\ \Gamma \to \Pi\ \mathsf{A} \to \mathsf{Thin}\ (\Delta \rhd \mathsf{A})\ \Gamma
\end{array}
$$

We can show these are indeed a category by deriving the identity thinning and composition, and proving the relevant laws

$$
\begin{array}{lll}
\mathsf{id}^{\mathsf{Th}} & : & \mathsf{Thin}\ \Gamma\ \Gamma \\
\_;^{\mathsf{Th}}\_ & : & \mathsf{Thin}\ \Delta\ \Gamma \to \mathsf{Thin}\ \Theta\ \Delta \to \mathsf{Thin}\ \Theta\ \Gamma \\
\mathsf{id};^{\mathsf{Th}} & : & \mathsf{id}^{\mathsf{Th}} ;^{\mathsf{Th}} \delta^{\mathsf{Th}} = \delta^{\mathsf{Th}} \\
;\mathsf{id}^{\mathsf{Th}} & : & \delta^{\mathsf{Th}} ;^{\mathsf{Th}} \mathsf{id}^{\mathsf{Th}} = \delta^{\mathsf{Th}} \\
;;^{\mathsf{Th}} & : & (\delta^{\mathsf{Th}} ;^{\mathsf{Th}} \sigma^{\mathsf{Th}}) ;^{\mathsf{Th}} \gamma^{\mathsf{Th}} = \delta^{\mathsf{Th}} ;^{\mathsf{Th}} (\sigma^{\mathsf{Th}} ;^{\mathsf{Th}} \gamma^{\mathsf{Th}})
\end{array}
$$

And indeed thinning encompass weakenings

$$
\begin{array}{lll}
\mathsf{wk}^{\mathsf{Th}} & : & \mathsf{Thin}\ (\Gamma \rhd \mathsf{A})\ \Gamma \\
\mathsf{wk}^{\mathsf{Th}} & :\equiv & \mathsf{id}^{\mathsf{Th}}\ {}^{+\mathsf{Th}}\ \_
\end{array}
$$

For their action, we can take a shortcut for now and rely on their embedding into renamings.

$$
\ulcorner\_\urcorner\mathsf{Th} : \mathsf{Thin}\ \Delta\ \Gamma \to \mathsf{Ren}\ \Delta\ \Gamma
$$

The standard model can be seen as interpreting object-level types into the corresponding objects in the category **Type** (where the objects are **Type**s and the morphisms are functions). In the presheaf model, we instead interpret into corresponding objects in the category of presheaves (where the objects are presheaves, and the morphisms are natural transformations).

For example, the unit presheaf (that is, the terminal object in the category of presheaves) is simply $\mathbb{1}^{\mathsf{Psh}} :\equiv \boldsymbol{\lambda}\ \Gamma \to \mathbb{1}$. Similarly, the products in the category of presheaves can be constructed as $\mathsf{F} \times^{\mathsf{Psh}} \mathsf{G} :\equiv \boldsymbol{\lambda}\ \Gamma \to \mathsf{F}\ \Gamma \times \mathsf{G}\ \Gamma$.

The exponential object in the category of presheaves is a bit more subtle. We might try to follow the pattern and define $\mathsf{F} \to^{\mathsf{Psh}} \mathsf{G} :\equiv \boldsymbol{\lambda}\ \Gamma \to \mathsf{F}\ \Gamma \to \mathsf{G}\ \Gamma$ but this doesn't quite work. When trying to implement
$\mathsf{thin} : \mathsf{Thin}\ \Delta\ \Gamma \to (\mathsf{F} \to^{\mathsf{Psh}} \mathsf{G})\ \Gamma \to (\mathsf{F} \to^{\mathsf{Psh}} \mathsf{G})\ \Delta$ we only have access to an

[20] In fact, we are forced to include general, stuck neutral terms to support application where the LHS is a variable.

F Δ and a function which accepts F Γs[21]. The solution is to quantify over thinnings, i.e. $\text{F} \to^{\text{Psh}} \text{G} \coloneqq \boldsymbol{\lambda}\ \Gamma \to \Pi\ \{\Delta\} \to \text{Thin}\ \Delta\ \Gamma \to \text{F}\ \Delta \to \text{G}\ \Delta$.

These are (almost) all the ingredients we need to define NbE values. Types in a context Γ are merely interpreted as the corresponding constructs in the category of presheaves. The presheaf action $\_[\_]^{\text{Psh}}$ is defined by recursion on types.

$$\begin{aligned}
&[\![\_]\!]^{\text{Psh}} : \text{Ty} \to \text{Ctx} \to \textbf{Type}\\
&[\![\ \text{A} \to \text{B}\ ]\!]^{\text{Psh}}\ \Gamma \coloneqq \Pi\ \{\Delta\} \to \text{Thin}\ \Delta\ \Gamma \to [\![\ \text{A}\ ]\!]^{\text{Psh}}\ \Delta \to [\![\ \text{B}\ ]\!]^{\text{Psh}}\ \Delta\\
&[\![\ \text{A} \times \text{B}\ ]\!]^{\text{Psh}}\ \Gamma \coloneqq [\![\ \text{A}\ ]\!]^{\text{Psh}}\ \Gamma \times [\![\ \text{B}\ ]\!]^{\text{Psh}}\ \Gamma\\
&[\![\ \text{A} + \text{B}\ ]\!]^{\text{Psh}}\ \Gamma \coloneqq [\![\ \text{A}\ ]\!]^{\text{Psh}}\ \Gamma + [\![\ \text{B}\ ]\!]^{\text{Psh}}\ \Gamma\\
&[\![\ \mathbb{1}\ ]\!]^{\text{Psh}}\ \Gamma \coloneqq \mathbb{1}\\
&[\![\ \mathbb{0}\ ]\!]^{\text{Psh}}\ \Gamma \coloneqq \mathbb{O}
\end{aligned}$$

$$\_[\_]^{\text{Psh}} : [\![\ \text{A}\ ]\!]^{\text{Psh}}\ \Gamma \to \text{Thin}\ \Delta\ \Gamma \to [\![\ \text{A}\ ]\!]^{\text{Psh}}\ \Delta$$

Remark 2.4.1 (Naturality of Presheaf Exponentials)
Technically, our presheaf exponentials are still not quite right here. We also need a naturality condition [72]: thinning the argument should be equivalent to thinning the result of the application.

$$\begin{aligned}
&[\![\ \text{A} \to \text{B}\ ]\!]^{\text{Psh}}\ \Gamma\\
&\quad \coloneqq (\text{f} : (\Pi\ \{\Delta\} \to \text{Thin}\ \Delta\ \Gamma \to [\![\ \text{A}\ ]\!]^{\text{Psh}}\ \Delta \to [\![\ \text{B}\ ]\!]^{\text{Psh}}\ \Delta)\\
&\qquad) \times (\Pi\ \{\Delta\ \Theta\}\ \text{u}^{\text{V}}\ (\delta^{\text{Th}} : \text{Thin}\ \Delta\ \Gamma)\ (\sigma^{\text{Th}} : \text{Thin}\ \Theta\ \Delta)\\
&\qquad\qquad \to \text{f}\ (\delta^{\text{Th}}\ ;^{\text{Th}}\ \sigma^{\text{Th}})\ (\text{u}^{\text{V}}\ [\ \sigma^{\text{Th}}\ ]\text{Psh}) = (\text{f}\ \delta^{\text{Th}}\ \text{u}^{\text{V}})\ [\ \sigma^{\text{Th}}\ ]\text{Psh})
\end{aligned}$$

To merely implement NbE algorithm for (unquotiented) STLC, allowing unnatural →-typed values does not cause any trouble. However, when proving soundness, this refinement is essential [73] (specifically, when showing preservation of substitution). For simplicity, we will ignore the naturality condition for now.

A final subtlety arises with the *positive* type formers _ + _ and 𝟘. E.g. While $\boldsymbol{\lambda}\ \Gamma \to \mathbb{O}$ does satisfy all the necessary laws of an initial object, and terms of type 𝟘 can only occur inside empty contexts (i.e. contexts containing 𝟘), when it comes to evaluating a variable of type 𝟘, we cannot hope to produce a proof of 𝕆 (i.e. the context containing the empty type does not mean evaluation can give up - normalisation requires evaluating under all contexts).

To solve this, we must embed neutrals into the model. E.g. we could interpret 𝟘 as $\boldsymbol{\lambda}\ \Gamma \to \text{Ne}\ \Gamma\ \mathbb{0}$. $\boldsymbol{\lambda}\ \Gamma \to \text{Ne}\ \Gamma\ \mathbb{0}$ is obviously not an initial object in the category of presheaves, so by doing this we have slightly broken the model, but it turns out that only the η laws for 𝟘 are actually lost (which lines up exactly with the consequences of embedding neutrals in Nf). We are aiming only to β-normalise terms for now, and will therefore actually take a more extreme option, embedding neutrals of all types as to line up more closely with our β-normal forms.

$$\begin{aligned}
&\text{Val} : \text{Ctx} \to \text{Ty} \to \textbf{Type}\\
&\text{PshVal} : \text{Ctx} \to \text{Ty} \to \textbf{Type}\\
&\text{Val}\ \Gamma\ \text{A} \coloneqq \text{PshVal}\ \Gamma\ \text{A} + \text{Ne}\ \Gamma\ \text{A}\\
&\text{PshVal}\ \Gamma\ (\text{A} \to \text{B}) \coloneqq \Pi\ \{\Delta\} \to \text{Thin}\ \Delta\ \Gamma \to \text{Val}\ \Delta\ \text{A} \to \text{Val}\ \Delta\ \text{B}\\
&\text{PshVal}\ \Gamma\ (\text{A} \times \text{B}) \coloneqq \text{Val}\ \Gamma\ \text{A} \times \text{Val}\ \Gamma\ \text{B}\\
&\text{PshVal}\ \Gamma\ (\text{A} + \text{B}) \coloneqq \text{Val}\ \Gamma\ \text{A} + \text{Val}\ \Gamma\ \text{B}\\
&\text{PshVal}\ \Gamma\ \mathbb{1} \coloneqq \mathbb{1}\\
&\text{PshVal}\ \Gamma\ \mathbb{0} \coloneqq \mathbb{O}
\end{aligned}$$

Note that although we are mixing inductively (i.e. Ne) and recursively (i.e. PshVal) defined type families here, the combination remains well-founded.

[21] Note the Thin Δ Γ thinning can only transform F Γs into F Δs, not the other way around.

[72]: Liao (2025), *Exponential objects in presheaf categories*

[73]: Kovács (2017), *A machine-checked correctness proof of normalization by evaluation for simply typed lambda calculus*

[23] explores NbE using model based on sheaves (instead of presheaves) to fix this more elegantly and in doing so decides η (as well as β) equivalence for sums, We discuss the costs of doing this in in Section 3.5.

[23]: Altenkirch et al. (2001), *Normalization by Evaluation for Typed Lambda Calculus with Coproducts*

Thinning can now be implemented for PshVal Γ A by recursion on the type A. For thinning of values in general, we can delegate to thinning on PshVal Γ As and Ne Γ As as appropriate.

```
_[_]Val      : Val Γ A → Thin Δ Γ → Val Δ A
thinPshVal   : Π A → Thin Δ Γ → PshVal Γ A → PshVal Δ A

in₁ t^V  [ δ^Th ]Val ≔ in₁ (thinPshVal _ δ^Th t^V)
in₂ t^Ne [ δ^Th ]Val ≔ in₂ (t^Ne [ δ^Th ]Ne)

thinPshVal (A → B) δ^Th t^V         ≔ λ σ^Th u^V → t^V (δ^Th ;^Th σ^Th) u^V
thinPshVal (A × B) δ^Th (t^V , u^V) ≔ t^V [ δ^Th ]Val , u^V [ δ^Th ]Val
thinPshVal (A + B) δ^Th (in₁ t^V)   ≔ in₁ (t^V [ δ^Th ]Val)
thinPshVal (A + B) δ^Th (in₂ t^V)   ≔ in₂ (t^V [ δ^Th ]Val)
thinPshVal 𝟙 δ^Th ⟨⟩                 ≔ ⟨⟩
```

To implement NbE, we need to define both evaluation from terms to values and *quotation* from values to normal forms.

```
data Env : Ctx → Ctx → Type

qval : Π A → Val Γ A → Nf Γ A
eval : Tm Γ A → Env Δ Γ → Val Δ A
```

We start with evaluation, which is quite similar to $[\![\_]\!]^{\mathsf{Tm}}$ in the standard model, but needs to deal with the cases for stuck neutrals appropriately. Evaluation is done w.r.t. an environment, which unlike the standard model is now parameterised by two contexts, similarly to Thin/Tms: first, the context each of the values exist in and second the list of types of the values themselves.

```
data Env where
  ε   : Env Δ •
  _,_ : Env Δ Γ → Val Δ A → Env Δ (Γ ▷ A)
```

Note that environments can be thinned by simply folding _[_]$_{\mathsf{Val}}$, and identity environments can be constructed by lifting over context extension and embedding variables by composing `_ : Var Γ A → Ne Γ A and **in**$_2$: Ne Γ A → Val Γ A.

```
_[_]Env   : Env Δ Γ → Thin Θ Δ → Env Θ Γ
_^^Env_   : Env Δ Γ → Π A → Env (Δ ▷ A) (Γ ▷ A)
id^Env    : Env Γ Γ
```

Evaluation then proceeds by recursion on the target term. The main subtlety is in application of values, where the LHS is neutral. In this case we need to turn quote the RHS back to an Nf via qval to apply _·_ : Ne Γ (A → B) → Nf Γ A → Ne Γ B (i.e. evaluation actually depends on quotation).

```
lookupVal : Var Γ A → Env Δ Γ → Val Δ A
lookupVal vz     (ρ , t^V) ≔ t^V
lookupVal (vs i) (ρ , t^V) ≔ lookupVal i ρ

appVal : Val Γ (A → B) → Val Γ A → Val Γ B
appVal (in₁  t^V ) u^V ≔ t^V id^Th u^V
appVal (in₂  t^Ne) u^V ≔ in₂ (t^Ne · qval _ u^V)

π₁Val : Val Γ (A × B) → Val Γ A
π₁Val (in₁ (t^V , u^V)) ≔ t^V
π₁Val (in₂ t^Ne)        ≔ in₂ (π₁ t^Ne)

π₂Val : Val Γ (A × B) → Val Γ B
π₂Val (in₁ (t^V , u^V)) ≔ u^V
π₂Val (in₂ t^Ne)        ≔ in₂ (π₂ t^Ne)
```

```
caseVal : Val Γ (A + B)
        → (Val Γ A → Val Γ C) → Nf (Γ ▷ A) C
        → (Val Γ B → Val Γ C) → Nf (Γ ▷ B) C
        → Val Γ C
caseVal (in₁ (in₁ t^V)) u^V u^Nf v^V v^Nf ≔ u^V t^V
caseVal (in₁ (in₂ t^V)) u^V u^Nf v^V v^Nf ≔ v^V t^V
caseVal (in₂ t^Ne)      u^V u^Nf v^V v^Nf ≔ in₂ (case t^Ne u^Nf v^Nf)

eval (` i)          ρ ≔ lookupVal i ρ
eval (λ t)          ρ ≔ in₁ λ δ^Th u^V → eval t ((ρ [ δ^Th ]_Env) , u^V)
eval (t · u)        ρ ≔ appVal (eval t ρ) (eval u ρ)
eval (t , u)        ρ ≔ in₁ (eval t ρ , eval u ρ)
eval (π₁ t)         ρ ≔ π₁Val (eval t ρ)
eval (π₂ t)         ρ ≔ π₂Val (eval t ρ)
eval (in₁ B t)      ρ ≔ in₁ (in₁ (eval t ρ))
eval (in₂ A t)      ρ ≔ in₁ (in₂ (eval t ρ))
eval ⟨⟩             ρ ≔ in₁ ⟨⟩
eval (case t u v) ρ
  ≔ caseVal (eval t ρ)
            (λ t^V → eval u (ρ , t^V)) (qval _ (eval u (ρ ^^Env _)))
            (λ t^V → eval v (ρ , t^V)) (qval _ (eval v (ρ ^^Env _)))
```

To implement qval, we instead proceed by recursion on types. Being able to weaken values is critical to quoting back →-typed values, where to inspect their structure, we need to be able to apply them to a fresh variable vz.

```
qval A       (in₂ t)         ≔ ne t
qval (A → B) (in₁ f)         ≔ λ qval B (f wk^Th (in₂ (` vz)))
qval (A × B) (in₁ (t , u))   ≔ qval A t , qval B u
qval (A + B) (in₁ (in₁ t))   ≔ in₁ B (qval A t)
qval (A + B) (in₁ (in₂ t))   ≔ in₂ A (qval B t)
qval 𝟙       (in₁ ⟨⟩)        ≔ ⟨⟩
```

Normalisation can now be implemented by evaluation followed by quoting.

```
nbe : Tm Γ A → Nf Γ A
nbe t ≔ qval _ (eval t id^Env)
```

We are done! Of course, to verify our normalisation algorithm is correct (actually *prove* normalisation for STLC), we need to do more work, checking soundness and completeness as defined in Definition 2.2.3. We refer to [73] for the details, but in short, we can prove:

- **Soundness** by proving that eval preserves conversion by induction on terms, which in turn requires proving preservation of substitution (and to do this, we also need to enforce naturality of →-typed values as mentioned in Remark 2.4.1).
- **Completeness** by defining a logical relation between terms and values by induction on types, showing t [δ] and eval t ρ are logically related given the terms in δ are all logically related to the values in ρ and finally proving that qval preserves the logical relation.

[73]: Kovács (2017), *A machine-checked correctness proof of normalization by evaluation for simply typed lambda calculus*

2.4.3 NbE for Dependent Types

When applying NbE for dependent types, we need to deal with terms embedded inside types. As a first approximation, we might try and keep a similar type for Val and construct identity environments to evaluate embedded terms in on demand:

```
Val : Π Γ → Ty Γ → Type
Val Γ (if t A B) with eval t id^Env
... | tt     ≔ Val Γ A
... | ff     ≔ Val Γ B
... | ne t^Ne ≔ Ne Γ (if t A B)
```

However, this definition poses difficulties for the case of Π-types, where we need to recurse at types A [δ] and B [δ , u].

```
Val Γ (Π A B)
  ≔ Π {Δ δ} (δ^Th : Thin Δ Γ δ) (u^V : Val Δ (A [ δ ]))
  → Val Δ (B [ δ , u ])
```

Unfortunately, multiple things go wrong here:

- A [δ] and B [δ , u] are not structurally smaller than Π A B, so it is not obvious that Val as defined above is well-founded. The case for A can be fixed by relying on how thinnings do not structurally alter (substitution-normal) types in a meaningful way. However, B [δ , u] is harder In the presence of large elimination Remark 2.1.2, there is no easy structurally-derived order on types which is also stable w.r.t. substitution[22]
- It turns out that some of the cases in qval/uval depend on completeness of the NbE algorithm. We could attempt to mutually prove correctness, but this does not appear to work in practice, as explained in [60].

To solve the latter issue, we need to pair NbE values with the correctness proofs (fusing the presheaf model with the logical relation), and therefore index values by the term which we are evaluating (and environments by the list of terms they contain values of). To solve the former, we can additionally parameterise types by a substitution, and the corresponding environment in which to evaluate embedded terms.

```
Env : Π Δ Γ → Tms Δ Γ → Type
Val : Π Γ A Δ δ → Env Δ Γ δ → Tm Δ (A [ δ ]Ty) → Type
```

Evaluating both terms and substitutions can then be specified like so:

```
eval  : Π (t : Tm Γ A) (ρ : Env Δ Γ δ) → Val Γ A Δ δ ρ (t [ δ ])
eval* : Π δ (ρ : Env Θ Δ σ) → Env Θ Γ (δ ; σ)
```

Given we are indexing values by the evaluated term, it is convenient to also index normal forms by the normalised term (ultimately, working up to conversion, any term which happens to be convertible to the normal form).

```
data Ne : Π Γ A → Tm Γ A → Type
data Nf : Π Γ A → Tm Γ A → Type

data Ne where
  `_  : Π i → Ne Γ A (` i)
  _·_ : Ne Γ (Π A B) t → Nf Γ A u → Ne Γ (B [ < u > ]Ty) (t · u)
  if  : Π A {t u v}
      → Ne Γ 𝔹 t → Nf Γ (A [ < tt > ]Ty) u → Nf Γ (A [ < ff > ]Ty) v
      → Ne Γ (A [ < t > ]Ty) (if A t u v)

data Nf where
```

[22] Consider e.g. recursing on a natural number to build an iterated Π-types, as is sometimes done in dependently-typed languages to achieve arity-polymorphism.

[60]: Altenkirch et al. (2017), *Normalisation by Evaluation for Type Theory, in Type Theory*

```
  ne𝔹  : Ne Γ 𝔹 t → Nf Γ 𝔹 t
  neIF : Ne Γ 𝔹 u → Ne Γ (IF u A B) t → Nf Γ (IF u A B) t
  λ_   : Nf (Γ ▷ A) B t → Nf Γ (Π A B) (λ t)
  tt   : Nf Γ 𝔹 tt
  ff   : Nf Γ 𝔹 ff
```

Of course, if we are using a setoid-based model of syntax, we also need coercion operations

```
coeNe~ : Π Γ~ A~ → Tm~ Γ~ A~ t₁ t₂ → Ne Γ₁ A₁ t₁ → Ne Γ₂ A₂ t₂
coeNf~ : Π Γ~ A~ → Tm~ Γ~ A~ t₁ t₂ → Nf Γ₁ A₁ t₁ → Nf Γ₂ A₂ t₂
```

We will elide these coercions (and cases pertaining to them) from now on because dealing with coercions is ultimately very mechanical.

We also index thinnings by equivalent substitutions

```
data Thin : Π Δ Γ → Tms Δ Γ → Type where
  ε      : Thin • • ε
  _^ᵀʰ_  : Thin Δ Γ δ → Π A → Thin (Δ ▷ (A [ δ ]Ty)) (Γ ▷ A) (δ ^ A)
  _⁺ᵀʰ_  : Thin Δ Γ δ → Π A → Thin (Δ ▷ A) Γ (δ ; wk)

_[_]Nf   : Nf Γ A t → Thin Δ Γ δ → Nf Δ (A [ δ ]Ty) (t [ δ ])
_[_]Ne   : Ne Γ A t → Thin Δ Γ δ → Ne Δ (A [ δ ]Ty) (t [ δ ])
```

We can now define environments by recursion on contexts. An inductive definition like we had for STLC would still be well-founded, but causes some subtle technical issues later on

```
Env Δ •         δ ≔ 𝟙
Env Δ (Γ ▷ A) δ ≔ (ρ : Env Δ Γ (π₁ δ)) × Val Γ A Δ (π₁ δ) ρ (π₂ δ)
```

Values are a bit more complicated. Again, the key idea is interpret types into the category of presheaves, but dealing with large elimination requires evaluating the embedded Boolean term.

As in STLC (Remark 2.4.1), we technically should enforce naturality of Π-typed values here. To keep the presentation simpler, we again skip this.

```
if-Val : Π Γ A B Δ δ (ρ : Env Δ Γ δ) {u[]}
         → Tm Δ (IF u[] (A [ δ ]Ty) (B [ δ ]Ty))
         → Nf Δ 𝔹 u[] → Type
if-Val Γ A B Δ δ ρ t tt
  ≔ Val Γ A Δ δ ρ (coe~ rfl~ IF-tt t)
if-Val Γ A B Δ δ ρ t ff
  ≔ Val Γ B Δ δ ρ (coe~ rfl~ IF-ff t)
if-Val Γ A B Δ δ ρ {u[]} t (ne𝔹 _)
  ≔ Ne Δ (IF u[] (A [ δ ]Ty) (B [ δ ]Ty)) t

Val Γ 𝔹          Δ δ ρ t ≔ Nf Δ 𝔹 t
Val Γ (IF b A B) Δ δ ρ t ≔ if-Val Γ A B Δ δ ρ t (eval b ρ)
Val Γ (Π A B)    Δ δ ρ t
  ≔ Π {Θ γ} (γᵀʰ : Thin Θ Δ γ)
        {u} (uⱽ : Val Γ A Θ (δ ; γ) (ρ [ γᵀʰ ]Env) u)
  → Val (Γ ▷ A) B Θ ((δ ; γ) , u) ((ρ [ γᵀʰ ]Env) , uⱽ) ((t [ γ ]) · u)
```

We also enforce η-equality of functions this time by embedding neutrals only at 𝔹 and stuck IF types. This will slightly simplify the case in the fundamental theorem for function application, at the cost of making the embedding of neutrals into values more complicated. We call this embedding operation *unquoting*, and define it mutually with qval.

```
uval : Π A {t} → Ne Δ (A [ δ ]Ty) t → Val Γ A Δ δ ρ t
qval : Π A {t} → Val Γ A Δ δ ρ t → Nf Δ (A [ δ ]Ty) t
```

Evaluation of variables looks up the corresponding value in the environment as usual. Evaluation of abstractions relies on coercing the value at term `t [ (δ ; γ) , u ]` to `(λ (t [ (δ ; γ) ^ A ]) · u`

```
lookupᴱⁿᵛ : Π (i : Var Γ A) (ρ : Env Δ Γ δ) → Val Γ A Δ δ ρ (lookup i δ)
```

```
eval (` i) ρ ≔ lookupᴱⁿᵛ i ρ
eval tt    ρ ≔ tt
eval ff    ρ ≔ ff
eval (λ t) ρ {γ ≔ γ} γᵀʰ {u ≔ u} uⱽ
  ≔ coeⱽᵃˡ rfl~ (sym~ (Πβ {t ≔ t [ (_ ; _) ^ _ ]} {u ≔ u}))
          (eval {δ ≔ (_ ; _) , _} t ((ρ [ γᵀʰ ]Env) , uⱽ))
```

Dealing with the elimination rules (application and “if”-expressions) is a bit trickier. We want evaluate `t · u` in `ρ` by evaluating each term independently and directly applying them with the identity thinning, `eval t ρ idᵀʰ (eval u ρ)` but hit two different type errors:

- First of all, `eval t ρ idᵀʰ` expects a value in the environment `ρ [ idᵀʰ ]Env`, rather than `ρ`. We can separately prove the identity law for thinning of values and environments to account for this discrepancy.
- The overall type of the application ends up as

  ```
  Val (Γ ▷ A) B Δ (δ , (u [ δ ])) (ρ , eval u ρ) ((t [ δ ]) · (u [ δ ]))
  ```

  but the inductive hypothesis requires

  ```
  Val Γ (B [ < u > ]Ty) Δ δ ρ ((t [ δ ]) · (u [ δ ]))
  ```

  We seemingly need to “shift” substitutions onto and off of the type `(δ , (u [ δ ]) ≡ < u > ; δ)`.

  ```
  shiftVal[] : Val Δ (A [ δ ]Ty) Θ σ ρ t = Val Γ A Θ (δ ; σ) (eval* δ ρ) t
  ```

We can get a better picture of the latter puzzle here by concretely writing out the motives of the displayed (presheaf plus logical relation) model we are implicitly constructing via evaluation. The motives for Ctx, Ty, Var, Tm and Tms are:

```
record Motives : Type₂ where field
  PCtx  : Ctx → Type₁
  PTy   : PCtx Γ → Ty Γ → Type₁
  PVar  : Π (Γᴾ : PCtx Γ) → PTy Γᴾ A → Var Γ A → Type
  PTm   : Π (Γᴾ : PCtx Γ) → PTy Γᴾ A → Tm Γ A → Type
  PTms  : Π (Δᴾ : PCtx Δ) (Γᴾ : PCtx Γ) → Tms Δ Γ → Type
```

and in the case of evaluation, we instantiate these as follows

```
NbE : Motives
NbE .PCtx Γ         ≔ Π Δ → Tms Δ Γ → Type
NbE .PTy  Γᴾ A      ≔ Π Δ δ → Γᴾ Δ δ → Tm Δ (A [ δ ]Ty) → Type
NbE .PVar Γᴾ Aᴾ i   ≔ Π Δ δ (ρ : Γᴾ Δ δ) → Aᴾ Δ δ ρ (lookup i δ)
NbE .PTm  Γᴾ Aᴾ t   ≔ Π Δ δ (ρ : Γᴾ Δ δ) → Aᴾ Δ δ ρ (t [ δ ])
NbE .PTms Δᴾ Γᴾ δ   ≔ Π Θ σ (ρ : Δᴾ Θ σ) → Γᴾ Θ (δ ; σ)
```

such that, modulo reordering of arguments, these match the types of Env, Val, eval and eval*

```
elimCtx  : Π Γ  → PCtx Γ
elimTy   : Π A  → PTy (elimCtx Γ) A
elimVar  : Π i  → PVar (elimCtx Γ) (elimTy A) i
elimTm   : Π t  → PTm (elimCtx Γ) (elimTy A) t
elimTms  : Π δ  → PTms (elimCtx Δ) (elimCtx Γ) δ

elimCtx  Γ Δ δ     :≡ Env Δ Γ δ
elimTy   A Δ δ ρ t :≡ Val _ A Δ δ ρ t
elimVar  i  Δ δ ρ  :≡ lookup^Env i ρ
elimTm   t  Δ δ ρ  :≡ eval t ρ
elimTms δ Θ σ ρ    :≡ eval* δ ρ
```

From this perspective, we can see that the law we need corresponds exactly to preservation of type substitution in the model:

```
_[_]^P_Ty : PTy Γ^P A → PTms Δ^P Γ^P δ → PTy Δ^P (A [ δ ]_Ty)
A^P [ δ^P ]^P_Ty :≡ λ Θ σ ρ t → A^P Θ _ (δ^P Θ σ ρ) t

elim-[]Ty : Π {δ : Tms Δ Γ}
          → elimTy (A [ δ ]_Ty) = elimTy A [ elimTms δ ]^P_Ty

shiftVal[] {ρ :≡ ρ} {t :≡ t} :≡
  cong-app (cong-app (cong-app (cong-app elim-[]Ty _) _) ρ) t
```

It turns out we will also rely on preservation of id and wk:

```
_,^P_ : Π Γ^P → PTy Γ^P A → PCtx (Γ ▷ A)
Γ^P ,^P A^P :≡ λ Δ δ → (ρ : Γ^P Δ (wk ; δ)) × A^P Δ (wk ; δ) ρ ((` vz) [ δ ])

wk^P : Π {A^P : PTy Γ^P A} → PTms (Γ^P ,^P A^P) Γ^P (wk {A :≡ A})
wk^P :≡ λ θ σ ρ → ρ .π₁

id^P : PTms Γ^P Γ^P id
id^P :≡ λ θ σ ρ → ρ

elim-id  : elimTms (id {Γ :≡ Γ}) = id^P
elim-wk  : elimTms (wk {A :≡ A}) = wk^P {A^P :≡ elimTy A}
```

These laws are why we decided to implement Env recursively. In an inductive definition of Env, we would only get isomorphisms here.

From now on, we assume both the functor laws for $\_[\_]_{\mathsf{Env}}$ and the above preservation equations hold definitionally. Of course, we will need to prove these properties propositionally later.

With elim-[]Ty holding definitionally, evaluation of substitutions is merely of fold of eval over the list of terms.

```
eval* ε       ρ :≡ ⟨⟩
eval* (δ , t) ρ :≡ eval* δ ρ , eval t ρ
```

Finally, we return to dealing with the eliminator cases of eval. Evaluation of application just applies the left and right-hand-side values, while evaluation of "if"-expressions splits on the scrutinee. In the tt and ff cases, we just select the appropriate value, while if the scrutinee is a stuck neutral, we build a neutral "if" expression and embed it into Val by unquoting.

```
eval-if : Π A {t u v} (t^Nf : Nf Δ 𝔹 t)
        → Val (Γ ▷ 𝔹) A Δ (δ , tt) (ρ , tt) u
        → Val (Γ ▷ 𝔹) A Δ (δ , ff) (ρ , ff) v
        → Val (Γ ▷ 𝔹) A Δ (δ , t) (ρ , t^Nf) (if (A [ δ ^ 𝔹 ]_Ty) t u v)
eval-if {δ :≡ δ} A tt u^V v^V
  :≡ coe^Val (rfl~ {A :≡ A}) (sym~ (𝔹β₁ (A [ δ ^ 𝔹 ]_Ty))) u^V
eval-if {δ :≡ δ} A ff u^V v^V
  :≡ coe^Val (rfl~ {A :≡ A}) (sym~ (𝔹β₂ (A [ δ ^ 𝔹 ]_Ty))) v^V
```

```
eval-if {δ ≔ δ} A (ne𝔹 t^Ne) u^V v^V
  ≔ uval A (if (A [ δ ^ 𝔹 ]_Ty) t^Ne (qval A u^V) (qval A v^V))


eval (t · u)       ρ ≔ eval t ρ id^Th (eval u ρ)
eval (if A t u v) ρ ≔ eval-if A (eval t ρ) (eval u ρ) (eval v ρ)
```

We must also check in both Val and eval that β (and η in the case of Π-typed terms) equations are preserved. IF-tt and IF-ff are preserved up to coherence ($\text{Val}\ \Gamma\ (\text{IF tt A B})\ \Delta\ \delta\ \rho\ \text{t} \equiv \text{Val}\ \Gamma\ \text{A}\ \Delta\ \delta\ \rho\ (\text{coe\~{}}\ \_\ \_\ \text{t})$. IF$\beta_1$ and IFβ_2 are conserved similarly $\text{eval (if A tt u v)}\ \rho \equiv \text{coe}^{\text{Val}}\ \_\ \_\ (\text{eval u}\ \rho)$.

$\Pi\beta$ and $\Pi\eta$ are more subtle. We have

```
eval ((λ t) · u) ρ ≡ coe^Val _ _ (eval t (ρ , eval u ρ))
```

and

```
eval (λ ((t [ wk ]) · (` vz))) ρ
  ≡ λ γ^Th {u} u^V → coe^Val _ _ (eval (t [ wk ])
                                  ((ρ [ γ^Th ]_Env) , u^V) id^Th u^V)
```

But this does not get us quite far enough in either case. We need preservation of term substitution.

```
_[_]^P : Π {Γ^P : PCtx Γ} {Δ^P : PCtx Δ} {A^P : PTy Γ^P A}
       → PTm Γ^P A^P t → Π (δ^P : PTms Δ^P Γ^P δ)
       → PTm Δ^P (A^P [ δ^P ]^P_Ty) (t [ δ ])
t^P [ δ^P ]^P ≔ λ Δ σ ρ → t^P Δ (_ ; σ) (δ^P Δ σ ρ)


elim-[] : elimTm (t [ δ ]) = elimTm t [ elimTms δ ]^P
```

Finally, we can proceed to the definitions of quoting and unquoting. These functions are mutually recursive on types, with much of the complexity coming from dealing with large IF.

```
uval-if : Π A B {u[] t} (u^Nf : Nf Δ 𝔹 u[])
        → Ne Δ (IF u[] (A [ δ ]_Ty) (B [ δ ]_Ty)) t
        → if-Val Γ A B Δ δ ρ t u^Nf
qval-if : Π A B {u[] t} (u^Nf : Nf Δ 𝔹 u[])
        → if-Val Γ A B Δ δ ρ t u^Nf
        → Nf Δ (IF u[] (A [ δ ]_Ty) (B [ δ ]_Ty)) t


uval 𝔹         t^Ne              ≔ ne𝔹 t^Ne
uval (Π A B)   t^Ne γ^Th {u} u^V ≔ uval B ((t^Ne [ γ^Th ]Ne) · qval A u^V)
uval (IF b A B) t^Ne             ≔ uval-if A B (eval b _) t^Ne

uval-if A B tt          t^Ne ≔ uval A (coeNe~ rfl~ IF-tt coh t^Ne)
uval-if A B ff          t^Ne ≔ uval B (coeNe~ rfl~ IF-ff coh t^Ne)
uval-if A B (ne𝔹 u^Ne) t^Ne ≔ t^Ne

qval 𝔹          t^V ≔ t^V
qval (IF b A B) t^V ≔ qval-if A B (eval b _) t^V
qval (Π A B)    t^V ≔ coeNf~ rfl~ rfl~ (sym~ Πη) t^Nf
  where vz^V ≔ uval {δ ≔ _ ; wk {A ≔ (A [ _ ]_Ty)}} A (` vz)
        t^Nf ≔ λ qval B (t^V wk^Th vz^V)

qval-if A B tt t^V
  ≔ coeNf~ rfl~ (sym~ IF-tt) (sym~ coh) (qval A t^V)
qval-if A B ff t^V
```

```
    ≔ coeNf~ rfl~ (sym~ IF-ff) (sym~ coh) (qval B t^V)
qval-if A B (ne𝔹 u^Ne) t^V ≔ neIF u^Ne t^V
```

Again, we need to ensure IF-tt and IF-ff are preserved by uval and qval, and indeed they are (up to coherence), so finally, we obtain normalisation:

```
nbe : Π t → Nf Γ A t
nbe t ≔ qval {δ ≔ id} _ (eval t id^Env)
```

We have checked soundness throughout the development of the algorithm. Completeness instead follows from a simple inductive proof (on normal forms) that for t^{Nf} : Nf Γ A t, we have t = ⌜ t^{Nf} ⌝Nf.

We should also technically check preservation of substitution operations and the functor laws for thinning of values/environments. Functor laws for thinnings follow by induction on types/contexts and eventually (in the 𝔹 base case) on normal/neutral forms.

Preservation of substitution operations requires checking the associated naturality laws (which in-turn requires ensuring naturality of Π-typed values are natural). Staying well-founded is a little tricky: assuming substitution operations all respect some well-founded order, we could in principle induct w.r.t. that, though in Agda (as we saw in Section 2.4.1), well-founded induction gets quite ugly. We could also pivot to explicit eliminators, via which preservation laws would hold definitionally (see e.g. the canonicity proof given in [7]), but we would still have to check all naturality equations, and we would lose the conciseness of pattern matching. Ultimately I argue these technical details are not fundamental to the algorithm/proof.

[7]: Kaposi et al. (2025), *Type Theory in Type Theory Using a Strictified Syntax*

Related Work 3

3.1 Dependent Pattern Matching

Pattern matching in simply-typed languages (assuming a structural restriction on recursive calls) can be viewed as syntax sugar for using recursion principles. For example, addition of natural numbers can be defined alternatively by pattern matching or ℕ's recursor, ℕ-rec:

```
_ +ℕ _ : ℕ → ℕ → ℕ               ℕ-rec : ℕ → A → (A → A) → A
ze   +ℕ m ≔ m                    _ +ℕ _ : ℕ → ℕ → ℕ
su n +ℕ m ≔ su (n +ℕ m)          n +ℕ m ≔ ℕ-rec n m su
```

In dependently-typed languages, we are not limited to only recursion principles though. Dependently-typed eliminators can perform *induction*, enabling, for example, the inductive proof that ze is a right identity of $\_ +_{\mathbb{N}} \_$.

```
ℕ-elim : Π (P : ℕ → Type) (n : ℕ)
         → P ze → (Π {k} → P k → P (su k)) → P n
+ze : n +ℕ ze = n
+ze {n ≔ n} ≔ ℕ-elim (λ n′ → n′ +ℕ ze = n′) n refl (cong su)
```

Dependent pattern matching is the extension of pattern matching to dependently-typed programming languages [74, 75], supporting such inductive definitions. The key idea is, in the bodies of matches, to substitute each matched-on variable ("scrutinee") for the corresponding pattern everywhere in the typing context. For example, we can again prove +ze, this time by pattern matching:

```
+ze : n +ℕ ze = n
+ze {n ≔ ze}   ≔ refl
+ze {n ≔ su n} ≔ cong su (+ze {n ≔ n})
```

In the n ≔ ze case, the substitution ze / n is applied everywhere, including in the goal type $n +_{\mathbb{N}} ze = n$ to produce the *refined* goal $ze +_{\mathbb{N}} ze = ze$, at which **refl** typechecks successfully ($ze +_{\mathbb{N}} ze$ reduces to ze definitionally). A similar process makes the n ≔ su n case work out.

A limitation of dependent pattern matching, defined in this way, is that matching anything other than individual variables is hard to justify. Substitutions can only target variables. Many functional programming languages (e.g. Haskell [76]) support case_of_ *expressions* on the RHS of definitions, where the scrutinee can be any appropriately-typed term.

Some dependently-typed languages feature **with**-abstractions, enabling similar matching on intermediary expressions on the LHS. However, as we will explain in Section 3.1.2, this feature has some significant drawbacks.

3.1.1 Indexed Pattern Matching

Another important aspect of pattern matching in dependently-typed languages is dealing with indexed types. For example, the type, Fin n, of natural numbers bounded by n : ℕ.

[74]: Coquand (1992), *Pattern matching with dependent types*
[75]: Cockx (2017), *Dependent Pattern Matching and Proof-Relevant Unification*

[76]: Marlow (2010), *Haskell 2010 Language Report*

```
data Fin : ℕ → Type where
  fz : Fin (su n)
  fs : Fin n → Fin (su n)
```

“Fording” [77] shows us how to reduce *indexed* inductive types to *parameterised* inductive types, assuming the existence of a propositional identity type

“Any natural number the programmer wants, as long as it’s su n!”

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

```
data Finℱ (m : ℕ) : Type where
  fzℱ : m = su n → Finℱ m
  fsℱ : m = su n → Finℱ n → Finℱ m
```

but this does not immediately solve the puzzle of how to support “convenient” pattern matching. Without a way to match on the inductive propositional equality type x = y, we are forced into heavily (ab)using manual transport. To give an example, let us define the datatype of length-indexed vectors (again in ordinary and “Forded” style)

```
data Vec (A : Type) : ℕ → Type where          data Vecℱ (A : Type) (m : ℕ) : Type where
  []  : Vec A ze                                 []ℱ : m = ze → Vecℱ A m
  _::_ : A → Vec A n → Vec A (su n)              ::ℱ : m = su n → A → Vecℱ A n → Vecℱ A m
```

for which we will attempt to implement a total vector lookup operation. Under the “Forded” approach (without being able to match on _=_), we must use manual equational reasoning (including relying on a proof of injectivity of su) to get the indices to align in the recursive case, and we need to explicitly appeal to constructor disjointness to demonstrate that out-of-bounds accesses are impossible.

```
su-inj      : su n = su m → n = m
ze/su-disj  : ¬ ze = su n

vlookupℱ : Finℱ n → Vecℱ A n → A
vlookupℱ (fzℱ p)   ([]ℱ q)       ≔ 𝟘-elim (ze/su-disj (sym q ∙ p))
vlookupℱ (fsℱ p i) ([]ℱ q)       ≔ 𝟘-elim (ze/su-disj (sym q ∙ p))
vlookupℱ (fzℱ p)   (::ℱ q x xs)  ≔ x
vlookupℱ (fsℱ p i) (::ℱ q x xs)
  ≔ vlookupℱ (transp Finℱ (su-inj (sym p ∙ q)) i) xs
```

With Agda’s built-in support for indexed pattern-matching, we can instead simply write:

```
vlookup : Fin n → Vec A n → A
vlookup fz     (x :: xs) ≔ x
vlookup (fs i) (x :: xs) ≔ vlookup i xs
```

Behind the scenes, vlookup is elaborated to simultaneously match on the ℕ-typed variable, n. We do not need to give cases for n ≡ ze because Agda builds-in constructor disjointness, and in the recursive case, we get that the n in i : Fin n and in xs : Vec A n are equal from built-in constructor injectivity.

A key idea here is *forced patterns* (also called *dot patterns*) [75]. Variables, i, can be matched with arbitrary expressions, t, if the equation between the variable and expression (i ≡ t) is forced by simultaneous matches on indexed types.

[75]: Cockx (2017), *Dependent Pattern Matching and Proof-Relevant Unification*

In Agda, we can explicitly write forced patterns by prefixing the expression with a “ . ”. Note that below, we match on the n : ℕ argument to _::_ with the existing variable m (bound from matching on the Fin n index), rather than introducing a fresh variable. We are only able to do this because Agda internalises the fact that su is injective (so there is ultimately no other option).

```
vlookup : Fin n → Vec A n → A
vlookup {n ≔ . (su m)} (fz {n ≔ m})  (_::_ {n ≔ .m} x xs) ≔ x
vlookup {n ≔ . (su m)} (fs {n ≔ m} i) (_::_ {n ≔ .m} x xs) ≔ vlookup i xs
```

Indexed pattern matching makes it possible to reflect a subset of propositional equations (specifically those where the LHS or RHS is a single variable). For example, consider this (slightly intimidating) law stating that transports can be pushed underneath (dependent) function applications.

```
subst-application′ : Π {A : Type} (B : A → Type) {C : A → Type}
                       {x₁ x₂ : A} {y : B x₁}
                       (g : Π x → B x → C x) (p : x₁ = x₂)
                     → transp C p (g x₁ y) = g x₂ (transp B p y)
```

In Agda, we can prove this just by matching on p : x_1 = x_2 with **refl**, simultaneously forcing the match x_2 ≔ .x_1. It remains to prove **transp** C **refl** (g x_1 y) = g x_1 (**transp** B **refl** y), which reduces to g x_1 y = g x_1 y, at which point **refl** typechecks successfully.

```
subst-application′ B {x₁ ≔ x₁} {x₂ ≔ .x₁} g refl ≔ refl
```

3.1.2 With Abstraction

Both Agda and Idris 2 support matching on intermediary expression to a limited extent via **with**-abstractions (originally named "views") [78–80]. The key idea is to apply a one-off rewrite to the context, replacing every occurrence 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 "core" program only contains definitions that match on individual variables).

[78]: McBride et al. (2004), *The view from the left*
[79]: Agda Team (2024), *With-Abstraction*
[80]: 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 **tt** = f (f (f **tt**)) proof from the introduction (Chapter 1), taking advantage of this feature, the goal only gets partially simplified:

```
f3 : Π (f : 𝔹 → 𝔹) → f tt = f (f (f tt))
f3 f with f tt
f3 f | tt ≔ ?0
```

At ?0, Agda has replaced every occurrence of f **tt** in the context with **tt** exactly once, and so now expects a proof of **tt** = f (f **tt**), but it is not obvious how to prove this. We could match on f **tt** again, but Agda will force us to cover both the **tt** and **ff** cases, with no memory of the prior 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 between the scrutinee and pattern) to the new variable p.

This feature can also be simulated without special syntax via the "inspect" idiom [81].

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

```
f3 : Π (f : 𝔹 → 𝔹) → f tt = f (f (f tt))
f3 f with f tt in p
f3 f | tt ≔ tt
              = by sym p
             f tt
              = by cong f (sym p)
             f (f tt) ■
f3 f | ff with f ff in q
f3 f | ff | tt ≔ sym p
f3 f | ff | ff ≔ sym q
```

We could also avoid all manual equational reasoning by repeating previous matches, forced, by simultaneously matching on the propositional equality.

```
f3 : Π (f : 𝔹 → 𝔹) → f tt = f (f (f tt))
f3 f with f tt in p
f3 f | tt with f tt | p
... | .true | refl with f tt | p
... | .true | refl ≔ refl
f3 f | ff with f ff in q
f3 f | ff | tt with f tt | p
... | .false | refl ≔ refl
f3 f | ff | ff with f ff | q
... | .false | refl ≔ refl
```

Agda contains yet another piece of syntactic sugar to make this pattern neater: **rewrite** takes a propositional equality, and applies a one-off rewrite to the context by implicitly **with**-abstracting over the LHS.

```
f3 : Π (f : 𝔹 → 𝔹) → f tt = f (f (f tt))
f3 f with f tt in p
f3 f | tt rewrite p
              rewrite p
    ≔ refl
f3 f | ff with f ff in q
f3 f | ff | tt rewrite p
    ≔ refl
f3 f | ff | ff rewrite q
    ≔ refl
```

But by now we are up to four distinct syntactic constructs, and the proof is still significantly more verbose than with **smart if**:

```
\f. sif (f tt) then Rfl else (sif (f ff) then Rfl else Rfl)
```

with-abstractions also have a critical issue that **smart case** intends to solve: 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 must definitionally be of that neutral form, and replacing it with some pattern, 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 3.1.1 (Ill-typed **with**-abstraction Involving Fin)
In the following code snippet, we attempt a forced match on n $+_{\mathbb{N}}$ m, as this expression occurs in the index of i : Fin (n $+_{\mathbb{N}}$ m). Unfortunately, after rewriting n $+_{\mathbb{N}}$ m to su k, we are left with i : Fin (su k) and Pred n m i (which relies on i having type Fin n $+_{\mathbb{N}}$ m) is no longer typeable.

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

Agda cannot do better here than just throwing an error:

```
[UnequalTerms]
w != n +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, and contributes to the well-known problem of "green slime" [82] (the general term for pain arising from indexing types by neutral expressions, like $n +_{\mathbb{N}} m$ as above). A common issue is that a **with** abstraction works just fine when implementing some operation on an indexed type, but when attempting to later prove properties about this operation, repeating the same **with** abstraction suddenly fails.

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

3.2 Local Equational Assumptions

As mentioned in the introduction, this work is largely inspired by Altenkirch et al.'s work on **smart case** [17]. Following the dependently-typed syntax introduced in Section 2.3, we can add a **smart case** rule for Booleans (we name this **smart if** for short), assuming a way to extend contexts with equational assumptions (_▷_~_) and an associated weakening operator (wk~) as follows:

[17]: Altenkirch (2011), *The case of the smart case*

```
_▷_~_ : Π Γ {A} → Tm Γ A → Tm Γ A → Ctx
wk~   : Tms (Γ ▷ t₁ ~ t₂) Γ

if : Π (t : Tm Γ 𝔹)
   → Tm (Γ ▷ t ~ tt) (A [ wk~ ]Ty)
   → Tm (Γ ▷ t ~ ff) (A [ wk~ ]Ty)
   → Tm Γ A
```

We explore a type theory using a similar typing rule for "if" in Chapter 5. To give a small taste of what makes this theory tricky metatheoretically, we introduce the notions of *definitional inconsistency* and *equality collapse*.

Definition 3.2.1 (Definitional Context Inconsistency)
We define contexts to be definitionally inconsistent if tt and ff are convertible under that context.

```
incon : Ctx → Type
incon Γ ≔ _=_ {A ≔ Tm Γ 𝔹} tt ff
```

In ITT, definitionally identifying non-neutral terms is dangerous as it can lead to equality collapse [83].

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

Definition 3.2.2 (Equality Collapse) We define equality collapse as the state when all terms/types are convertible. Equality collapse specifically at the level of types is very dangerous, as we shall see shortly.

```
collapse : Ctx → Type
collapse Γ ≔ Π (A B : Ty Γ) → A = B
```

Remark 3.2.1 (Definitional Inconsistency Implies Equality Collapse)
Assuming congruence of conversion (which is highly desirable for definitional equality to behave intuitively), and large elimination of Booleans, we can derive equality collapse (A = B for arbitrary types A and B) from definitional inconsistency (tt = ff).

```
incon-collapse : incon Γ → collapse Γ
```

```
incon-collapse Γ! A B ≔
  A
   = by  sym IF-tt
  IF tt A B
   = by  cong (λ □  →  IF □ A B) Γ!
  IF ff A B
   = by  IF-ff
  B ■
```

Assuming β-rules for Booleans, we can also also derive that definitionally inconsistent contexts collapse the term equality, using a similar argument.

Convertibility of all types is dangerous, as we can perform self-application, and define terms that loop w.r.t β-reduction.

Example 3.2.1 (Equality Collapse Enables Self-Application)
Under definitional equality of all types, we have that, e.g. $A \rightarrow A \equiv A$, which means we can type self-application.

```
_[_]! : incon Γ  →  Tms Δ Γ  →  incon Δ


self-app : incon Γ  →  Tm Γ (Π A (A [ wk ]Ty))
self-app Γ!
  ≔ λ transp (Tm _) wk<>Ty
              (transp (Tm _) (incon-collapse (Γ! [ wk ]!) _ _) vz · vz)
```

To jump from here to truly looping terms such as Ω ((λ x. x x) (λ x. x x)) we only need to repeat the construction.

Of course, if a particular context is definitionally inconsistent, conversion is trivially decidable (any two terms must be convertible, assuming a β-law for Booleans). However, if definitional inconsistency is not decidable, then the above example means we also lose normalisation/decidable conversion in open contexts, and therefore in the setting of dependent types (specifically ITT) decidability of typechecking is lost.

In SC^{Bool}, collapsing the definitional equality is easy. We can just case split on a closed Boolean (or some term that is convertible to a closed Boolean). Then, one of the contexts, of one of the "if" branches, most contain the definitionally-inconsistent assumption tt ~ ff (or reversed).

Normalising the scrutinee before checking the branches of "if" (to see if it reduces to a closed Boolean) is not enough to detect definitional inconsistency. For example, consider the program (in a context where b : 𝔹 and not ≔ λ b. if b ff tt)

```
if ( not  b) (if b ?0 ?1) ?2
```

When checking the inner "if" expression (in the left branch of the outer "if"), the scrutinee, b, is is normal form (the assumption not b ~ tt is not enough to derive b ≡ ff by pure equational reasoning). However, in ?0, the context becomes definitionally inconsistent (b ~ tt and the β-rule for Booleans implies not b ≡ not tt ≡ ff, so not b ~ tt enables deriving ff ≡ tt).

Possible solutions here include:

- Iterating over the set of equations reducing LHSs[1] w.r.t. all other equations. We repeat this until either a fixed point is reached, and we are left with a confluent term rewriting system (TRS), or a definitional inconsistency is detected. This technique is named *completion*, and on ground first-order equations it is known to terminate [8].

[1] For more general equations where the RHS might be reducible, we can reduce both sides, and use a notion of a well-founded term ordering to orient them appropriately.

[8]: Baader et al. (1998), *Term Rewriting and All That*

- Placing syntactic restrictions on the equations which can be introduced (i.e. the scrutinees of **smart if** expressions) to try and prevent situations like this early (for example, perhaps we could require that all LHSs are irreducible from the start).

We will consider both of these strategies over the course of Chapter 4, Chapter 5 and Chapter 6.

A more direct use of local equational assumptions is *local equality reflection.*

3.2.1 Local Equality Reflection

Recall the equality reflection rule from ETT

$$\mathsf{reflectETT} : \mathsf{Tm}\ \Gamma\ (\mathsf{Id}\ \mathsf{A}\ \mathsf{t}_1\ \mathsf{t}_2) \to \mathsf{t}_1 = \mathsf{t}_2$$

If we turn this from a meta-level judgement to an object-level one, we arrive at a syntactic construct we call "local equality reflection" (assuming some way of extending contexts with local equational assumptions)

$$\begin{aligned}&\mathsf{reflect} : \mathsf{Tm}\ \Gamma\ (\mathsf{Id}\ \mathsf{A}\ \mathsf{t}_1\ \mathsf{t}_2) \to \mathsf{Tm}\ (\Gamma \rhd \mathsf{t}_1 \sim \mathsf{t}_2)\ (\mathsf{B}\ [\ \mathsf{wk}{\sim}\ ]_{\mathsf{Ty}})\\ &\quad \to \mathsf{Tm}\ \Gamma\ \mathsf{B}\end{aligned}$$

reflect is significantly less powerful than "full" ETT equality reflection (reflectETT); the programmer must specify every equality proof they want to reflect, rather than assuming the existence of an oracle which can do proof search during typechecking[2] . The utility over transport comes from not requiring the programmer to also specify where to apply each equation (we assume definitional equality is congruent).

Perhaps surprisingly then, typechecking dependent types with this local reflection rule is still undecidable, as shown in [84]. They present the example of reflecting the definition of the Collatz function (in a context where $\mathsf{f} : \mathbb{N} \to \mathbb{B}$ is a variable).

$$\mathsf{Id}\ (\mathbb{N} \to \mathbb{B})\ \mathsf{f}\ (\lambda\ \mathsf{n}.\ \mathsf{if\ even?\ n\ then\ f}\ (\mathsf{n}\ /_{\mathbb{N}}\ 2)\ \mathsf{else\ su}\ (3\ \times_{\mathbb{N}}\ \mathsf{n}))$$

If we accept the new definitional equality, f had better halt on all $\mathbb{N}$-typed inputs or β-reduction might run into a loop (e.g. deciding $\mathsf{f}\ \mathsf{k} \equiv \mathbf{tt}$ for $\mathsf{k} : \mathbb{N}$). At least in the context of "obviously" definitionally inconsistent Remark 3.2.1 equations such as $\mathsf{Id}\ \mathbb{B}\ \mathsf{tt}\ \mathsf{ff}$, we can skip conversion-checking (all terms must be convertible). For equations like the above though, we cannot assume inconsistency: without a counter-example to the Collatz conjecture, we have no way of deriving a contradiction from its assumption.

For another example, imagine the programmer reflects a propositional equation between two arbitrary closed functions from $\mathbb{N}$s to $\mathbb{B}$s, $\mathsf{Id}\ (\mathbb{N} \to \mathbb{B})\ \mathsf{f}\ \mathsf{g}$. Assuming our type theory is not anti-classical, assuming identity between pointwise-equal functions is reasonable (even if we do not build-in function extensionality). However, if we reflect $\mathsf{f} \equiv \mathsf{g}$ for a f and g for which there exists a closed natural number $\mathsf{n} : \mathbb{N}$ such that $\mathsf{f}\ \mathsf{n} \equiv \mathsf{tt}$ and $\mathsf{g}\ \mathsf{n} \equiv \mathsf{ff}$, then by congruence we are in a definitionally inconsistent context, and self-application is typeable. We have no hope of catching this in a typechecker, as the problem of deciding whether two functions with infinitary domains are equal on all inputs (for any reasonably expressive theory[3]) is undecidable.

Local equality reflection and **smart case** are not ultimately so different.

Remark 3.2.2 (Smart Case is Local Equality Reflection)
Assuming indexed matching (via forced patterns) and ordinary eliminators, an unrestricted **smart case** is exactly as powerful as reflect. To reflect a propositional equality, $\mathsf{p} : \mathsf{Id}\ \mathsf{A}\ \mathsf{u}\ \mathsf{v}$, with **smart case**, we can simultaneously match on p with **refl** and the term $\mathsf{u} : \mathsf{A}$ with the forced pattern .v. To go the other direction, we can apply the associated splitter for the type, and then in each branch, reflect the provided

[2] This is perhaps a slightly unfair interpretation of reflectETT given the system is not expected to have decidable typechecking.

[84]: Sjöberg et al. (2015), *Programming up to Congruence*

[3] E.g. is capable for formalising Peano arithmetic.

propositional equality.
As a corollary, typechecking unrestricted **smart case** is undecidable! Therefore, when justifying a language featuring **smart case** or local equality reflection, we must pay specific attention to identifying restrictions on the class of equations which can be reflected, so decidability can be maintained.
Generally in this project, we focus on using **smart case**-style syntax to introduce local equations, as we argue it often makes examples cleaner. Furthermore, in the absence of indexed pattern matching/forced patterns, **smart case** suggests some nice potential restrictions on equations (e.g. **smart if** can only introduce equations of the form t ~ tt and t ~ ff).

3.2.2 Existing Systems with Local Equations

GHC Haskell may not be a full dependently-typed language (it is instead based on a System F_C core theory) but the surface language does include many quite sophisticated features, including automation of its type-level equality constraints [85] (implemented in the *constraint solving* typechecking phase). Combined with type families, which enable real computation at the type level, we can actually “prove”[4] our standard f True = f (f (f True)) example.

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

Example 3.2.2 (f b = f (f (f b)), in Haskell)

```
type data Bool ≔ True | False
type SBool :: Bool → Type
data SBool b where
  STrue :: SBool True
  SFalse :: SBool False
type F :: Bool → Bool
type family F b where
boolLemma :: (forall b. SBool b → SBool (F b))
           → F True :~: F (F (F True))
boolLemma f ≔ case f STrue of
  STrue → Refl
  SFalse → case f SFalse of
    STrue → 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 3.2.3 (Conversion is not Congruent in GHC Haskell)
In GHC 9.12.2, 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
```

[4] 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 *singleton* encodings [86, 87].
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 [88]. 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.

[86]: Lindley et al. (2013), *Hasochism: the pleasure and pain of dependently typed haskell programming*
[87]: Eisenberg (2020), *Stitch: the sound type-indexed type checker (functional pearl)*
[88]: Kiss et al. (2019), *Higher-order type-level programming in Haskell*

Haskell is not the only language to support a **smart case**-like feature. The dependently-typed language "Zombie" builds congruence closure into the definitional equality of the surface language and implements **smart case** in full, while retaining decidable typechecking [84]. The sacrifice is β-conversion: Zombie does not automatically apply computation rules, requiring manual assistance to unfold definitions during typechecking.

[84]: 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 generally assume β-equality.

One could view traditional definitional equality as a hack, carefully defining an equational theory that happens to be a decidable subset of propositional equality, and building it into the typechecker, but it is undeniably effective.

The Lean proof assistant features a tactic for automatically discharging equality proofs following from congruence closure [89], but their algorithm is not capable of interleaving congruence and reduction (which is required in our setting to ensure transitivity of conversion).

[89]: Selsam et al. (2016), *Congruence Closure in Intensional Type Theory*

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

Sixty [90] 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.

Andromeda 2 [91] is a proof assistant that supports local equational assumptions via rewriting with the goal of supporting user-specified type theories. The system goes beyond the class of equations we consider here, supporting also rewrite rules that themselves quantify over variables (standing for all appropriately-typed terms). In this report, we refer to such contextual equations that only refer to prior-bound variables as *ground*, and therefore view this work as accounting also for *non-ground* equations[5]. They focus primarily on proving soundness of their equality checking algorithm, and leave confluence/termination checking and completeness results for future work.

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

[92] also deals with non-ground equations, following work on controlling unfolding in type theory [93]. In their setting, equations cannot refer directly to local bound variables as **smart case** requires.

[92]: Winterhalter (2025), *Controlling computation in type theory, locally*

[93]: Gratzer et al. (2022), *Controlling unfolding in type theory*

3.3 Global Equational Assumptions

There has been a significant body of work examining type theories extended with global (non-ground) rewrite rules, plus implementations in Dedukti [62], Agda [11] and Rocq [12]. Work in the area has examined automatic (albeit necessarily conservative) confluence [94] and termination [95] checking of these rewrites. Agda's implementation of REWRITE rules specifically checks confluence, but not termination.

A key difference between these works and **smart case** is that global equations cannot refer to local variables bound inside terms/definitions. We also cannot ever disable global rewrites which earlier definitions might depend on without endangering subject reduction, which becomes problematic when building larger developments. For example, two different modules might rely on distinct sets of global rewrites that are individually confluent and terminating, but together are not. It is now impossible to safely import code from both of these modules.

[62]: Assaf et al. (2023), *Dedukti: a Logical Framework based on the λΠ-Calculus Modulo Theory*

[11]: Cockx (2019), *Type Theory Unchained: Extending Agda with User-Defined Rewrite Rules*

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

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

[95]: Genestier (2019), *SizeChangeTool: A Termination Checker for Rewriting Dependent Types*

3.4 Elaboration

A principled and increasingly popular way to design and implement programming languages [96–98] is by *elaboration* into a minimal core syntax. A significant benefit of this approach is modularity [99]: multiple extensions to the surface language can be

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

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

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

[99]: Cockx (2024), *Agda Core: The Dream and the Reality*

[5] 5: We justify this terminology by noting that, in a fixed context, variables essentially act like constants. Of course, unlike ordinary ground term rewriting, we do need to worry about what happens when these bound variables are substituted for other terms.

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 justified, core theory.

[65, 100] investigate elaborating ETT and ITT plus global rewrite rules into 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 (such as Homotopy Type Theory)[6].

[65]: Winterhalter et al. (2019), *Eliminating reflection from type theory*
[100]: Blot et al. (2024), *From Rewrite Rules to Axioms in the $\lambda\Pi$-Calculus Modulo Theory*

In this report, we do not consider the problem of similarly elaborating **smart case** to an ordinary intensional type theory, without contextual equational assumptions (one could consider this mostly covered by the above-cited prior work). Instead, in Chapter 6 we leverage a quite simple elaboration algorithm based on lambda-lifting to give the appearance of **smart case** while maintaining a more well-behaved core theory than SC$^{\text{Bool}}$.

3.5 Strict η for Coproducts

Another use-case for tracking equational assumptions is to decide conversion in the presence of strict η for Booleans or, more generally, coproducts. For example, [101] and [23] introduce collections of equations between $(\mathsf{A} + \mathsf{B})$-typed neutrals and terms of the form $\mathsf{in}_1\ \mathsf{i}$ or $\mathsf{in}_2\ \mathsf{i}$ (where i is a variable), the latter naming these “neutral constrained environments”.

[101]: Dougherty et al. (2000), *Equality between Functionals in the Presence of Coproducts*
[23]: Altenkirch et al. (2001), *Normalization by Evaluation for Typed Lambda Calculus with Coproducts*

We formally define the simply-typed η-law for Booleans, following the syntax introduced in Section 1 (assuming fully strict substitution laws, and propositional quotienting by conversion).

Definition 3.5.1 (η For Booleans)
η-conversion for Booleans can be stated as

```
𝔹η : u [ < t > ] = if t (u [ < tt > ]) (u [ < ff > ])
```

In words: every term containing a boolean-typed sub-expression, t : Tm Γ 𝔹, can be expanded into an “if” expression, with t replaced by tt in the tt branch and ff in the ff branch.
In dependent type theory, we can prove this law internally by induction on Booleans (even if our theory, like Agda, does not implement η for such types definitionally).

```
𝔹-η : Π (f : 𝔹 → A) b
      → f b = 𝔹-rec b (f tt) (f ff)
𝔹-η f tt ≔ refl
𝔹-η f ff ≔ refl
```

η for Booleans is quite powerful. For example, it enables deriving *commuting conversions*.

Example 3.5.1 (Commuting Conversions)
Commuting conversions express the principle that case-splits on inductive types can be lifted upwards (towards the root of the term) as long as the variables occurring in the scrutinee remain in scope. i.e.

```
𝔹-comm : f [ < if t u v > ] = if t (f [ < u > ]) (f [ < v > ])
```

This follows from $\mathbb{B}\eta$ and $\mathbb{B}\beta_1/\mathbb{B}\beta_2$ as follows

```
𝔹-comm {f ≔ f} {t ≔ t} {u ≔ u} {v ≔ v} ≔
  (f [ < if t u v > ])
   = by 𝔹η {u ≔ f [ wk ^ _ ] [ < if (` vz) (u [ wk ]) (v [ wk ]) > ]}
```

[6] Note that implicit transporting along equivalences between completely distinct types (such as ℕ and ℤ) could be used to implement coercions/subtyping, so automating equational reasoning on types with proof-relevant equality could still be useful if there is a distinguished “default” mapping.
Such use-cases appear 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.

```
    if t (f [ < if tt u v > ]) (f [ < if ff u v > ])
     = by  cong₂ (λ □₁ □₂  →  if t (f [ < □₁ > ]) (f [ < □₂ > ])) 𝔹β₁ 𝔹β₂
    if t (f [ < u > ]) (f [ < v > ]) ■
```

Again, we can prove an analagous propositional law internally, using 𝔹-η.

```
Bool-comm  :  Π (f  :  A  →  B) (b  :  𝔹) (x y  :  A)
              →  f (𝔹-rec b x y)  =  𝔹-rec b (f x) (f y)
Bool-comm f b x y  ≔  𝔹-η (λ b  →  f (𝔹-rec b x y)) b
```

In a system with strict η for functions and another type A, definitional equality of functions on A is observational[7].

Theorem 3.5.1 (Strict η for Functions and Booleans Implies Definitional Observational Equality of Boolean Functions)
Assuming f · tt = g · tt and f · ff = g · ff, we can derive f = g from →η and 𝔹η.

```
𝔹⇒  :  Π {f g  :  Tm Γ (𝔹  →  𝔹)}
       →  f · tt  =  g · tt  →  f · ff  =  g · ff
       →  f  =  g
𝔹⇒ {f  ≔  f} {g  ≔  g} tt≡ ff≡  ≔
  f
   = by  →η
  λ f′ · ` vz
   = by  cong (λ □  →  λ f′ · □) (𝔹η {u  ≔  ` vz})
  λ f′ · if (` vz) tt ff
   = by  cong (λ_) (𝔹-comm {f  ≔  f′ [ wk ] · ` vz})
  λ (if (` vz) (f′ · tt) (f′ · ff))
   = by  cong₂ (λ □₁ □₂  →  λ (if (` vz) □₁ □₂)) tt≡′ ff≡′
  λ if (` vz) (g′ · tt) (g′ · ff)
   = by  cong (λ_) (sym (𝔹-comm {f  ≔  g′ [ wk ] · ` vz}))
  λ g′ · if (` vz) tt ff
   = by  cong (λ □  →  λ g′ · □) (sym (𝔹η {u  ≔  ` vz}))
  λ g′ · ` vz
   = by  sym  →η
  g ■
  where f′    ≔  f [ wk ]
        g′    ≔  g [ wk ]
        tt≡′  ≔  cong _[ wk ] tt≡
        ff≡′  ≔  cong _[ wk ] ff≡
```

Subtly, propositional, observational equality of Boolean functions (f **tt** = g **tt** → f **ff** = g **ff** → f = g) is not provable internally using the with propositional 𝔹-η unless we also assume function extensionality to get our hands on a 𝔹-typed term to pass as b.
This is to be expected, given we have seen that propositional η-laws for inductive types can be proven merely by induction, but observational equality of functions (called "function extensionality" in the general case) is not provable in intensional MLTT [29].

It is perhaps also worth noting that in a dependently-typed setting, η for A + B binary coproducts can be obtained merely with η for booleans, Σ types and large elimination, via the encoding A + B ≔ Σ 𝔹 (**λ** b → if b A B) [103].

As mentioned in Section 1.1 - Computation and Uniqueness, while η rules for positive types (such as Booleans or coproducts), can be useful, they do have some downsides.

- 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 [46, 47] or sheaf [23] theory. Normalisation for dependent type theory

[7] 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 [102].
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).

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

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

[103]: Kovács (2022), *Strong eta-rules for functions on sum types*

[46]: Ghani (1995), *Adjoint Rewriting*
[47]: Lindley (2007), *Extensional Rewriting with Sums*
[23]: Altenkirch et al. (2001), *Normalization by Evaluation for Typed Lambda Calculus with Coproducts*

with boolean η remains open (though some progress on this front has been made recently [56]).

- Second, efficient implementation appears challenging. Algorithms such as [23] aggressively introduce case-splits on all neutral subterms of coproduct-type and lifts the splits as high as possible, in an effort to prevent the build-up of stuck terms. In the worst-case, this requires re-normalising twice for every distinct coproduct-typed neutral subterm. [56] proposes a similar algorithm for type-checking dependent types with strict boolean η, using a monadic interpreter with an effectful splitting operation. [104] is even more extreme: when a variable f of type 𝔹 → 𝔹 is bound, for example, case splits are generated on f **tt** and f **ff** (regardless of whether such terms actually occur anywhere in the body), in essence enumerating over all possible implementations of f.

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

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

The (current) lack of normalisation result for dependent types with strict Boolean η prevents justifying **smart case** merely by piggy-backing on existing work. The problem we examine in this report is further distinguished from η-equality due to its potential to target a wider variety of equations than is allowed in the "neutral constrained environments" of Dougherty [101] or Altenkirch [23]. Specifically, we are also interested in equations where both sides are neutral, or equations between infinitary-typed terms (ℕ, List A, Tree A, etc..., for which η-equality is undecidable).

[101]: Dougherty et al. (2000), *Equality between Functionals in the Presence of Coproducts*

3.6 Extension Types

In retrospect, the machinery we introduce in SC^{Bool} and SC^{Def} to extend contexts with convertibility assumptions and generalise substitutions appropriately can be seen as a subset of extension types [105, 106].

[105]: Riehl et al. (2017), *A type theory for synthetic ∞-categories*
[106]: Zhang (2023), *Three non-cubical applications of extension types*

Extension types assume the existence of a sort of propositions F that we can extend contexts with

```
F   : Ctx → Type
_▷F_ : Π Γ → F Γ → Ctx
```

Extension types, A | ϕ ⤳ u, encode terms that are convertible u under the assumption of ϕ.

```
_|_⤳_ : (A : Ty Γ) (ϕ : F Γ) → Tm (Γ ▷F ϕ) (A [ wkF ]Ty)
   → Ty Γ
```

```
inS    : Π (t : Tm Γ A) → Tm Γ (A | ϕ ⤳ (t [ wkF ]))
outS   : Tm Γ (A | ϕ ⤳ u) → Tm Γ A
out⤳   : Π {t : Tm Γ (A | ϕ ⤳ u)} → outS t [ wkF ] = u
extβ   : outS (inS {ϕ ≔ ϕ} t) = t
```

The introduction rule inS is often written as

```
inS′ : Π (t : Tm Γ A)
    → t [ wkF ] = u
    → Tm Γ (A | ϕ ⤳ u)
```

making explicit that t needs to be convertible to u under the assumption ϕ. Assuming a quotiented syntax, these two rules are equivalent (inS′ is just the "Forded" version of inS).

Assuming a universe of types, U, and an F Γ which includes 𝔹-typed convertibility assumptions, we can give the following elimination rule for Booleans.

```
U   : Ty Γ
El  : Tm Γ U → Ty Γ
_~_ : Tm Γ 𝔹 → Tm Γ 𝔹 → F Γ

ext-if : Π {A : Tm Γ U} (t : Tm Γ 𝔹)
         (Att : Tm Γ (U | (t ~ tt) ⤳ (A [ wkF ])))
         (Aff : Tm Γ (U | (t ~ ff) ⤳ (A [ wkF ])))
       → Tm Γ (El (outS Att))
       → Tm Γ (El (outS Aff))
       → Tm Γ (El A)
```

In the context of Cubical type theory, extension types with propositions F Γ corresponding to interval expressions that must definitionally equal i1 are are also called Cubical subtypes ([107]).

[107]: Agda Team (2024), *Cubical*

This bears some resemblance with **smart if**: the LHS and RHS branches of the if expression can differ in type up to replacing the scrutinee with tt/ff. Unlike the typing rule for **smart case** suggested in [17], the LHS and RHS branch are still typed in context Γ, which could make the metatheory much easier.

Unfortunately, this construct is more limited than we would like. The concise proof of f **tt** = f (f (f **tt**)) from the introduction (Chapter 1) cannot be replicated with ext-if. If we make an attempt (working internally, for convenience)

```
f3 : Id 𝔹 (f tt) (f (f (f tt)))
f3 ≔ ext-if (f tt) (inS (Id 𝔹 tt tt)) (inS (Id 𝔹 ff (f (f ff))))
             refl
             ext-if (f ff) (inS (Id 𝔹 ff (f tt))) (inS (Id 𝔹 ff ff))
                    ?0 refl
```

we get stuck in the case labelled ?0. The problem is that, as with **with**-abstraction, ext-if does not have "memory" of the prior case splits. ext-if still does manage a better job than **with**-abstraction, being able to apply the equation to the type multiple times (e.g. simplifying f (f (f tt)) all the way to tt in the left branch of the split on f tt). However, in ?0, we need to reuse the fact that f tt = ff, and no longer have access to it.

I therefore argue that **smart case** truly does need to type the branches of the split in a context extended with the appropriate equation. Therefore, it appears that the existing theory of extension types is not directly applicable to this use-case.

Type inference also appears to be trickier for ext-if, than full **smart if** hence the explicitly annotated for the LHS and RHS types. **smart if** (as defined in Section 3.2) can check the LHS and RHS branches at the same type as the entire if expression, A : Ty Γ, only weakened to account for the new equation. ext-if, on the other hand, requires coming up with types in Γ for the LHS and RHS branches with the constraint that they are convertible to A after weakening (the choices here are not unique, because distinct types can be made convertible after introducing an equation).

[17]: Altenkirch (2011), *The case of the smart case*

STLC Modulo Equations 4

In this chapter, we prove decidability of conversion for STLC modulo a fixed (global) set of Boolean equations via rewriting to completion. We end by discussing the challenges in adapting this proof to a setting where these equations can be introduced locally.

4.1 STLC with Boolean Equations

We begin our exploration of **smart case**/local equality reflection by studying convertibility of STLC terms modulo equations. We will focus on equations of a restricted form: $t \equiv b$, where t is a 𝔹-typed term and b is a closed Boolean.

We use an intrinsically-typed syntax with recursive substitutions following Section 2.2.1, containing → and 𝔹 type formers, with their standard introduction and elimination rules. Note that simply-typed "if"-expressions require the left and right branches to have exactly the same type.

```
if : Tm Γ 𝔹 → Tm Γ A → Tm Γ A → Tm Γ A
```

The computation rules then just select the appropriate branch.

```
𝔹β₁ : if tt u v ~ u
𝔹β₂ : if ff u v ~ v
```

We will package the set of equations with which we decide conversion modulo into *equational contexts*. For our restricted class of equations, these take the form of lists of pairs of 𝔹-typed terms and closed Booleans.

```
data Eqs (Γ : Ctx) : Type where
  •         : Eqs Γ
  _▷_ ⤳ _ : Eqs Γ → Tm Γ 𝔹 → 𝔹 → Eqs Γ
```

Substituting equational contexts folds substitution over the LHS terms.

```
_[_]Eq : Eqs Γ → Tms[ q ] Δ Γ → Eqs Δ
•               [ δ ]Eq ≔ •
(Ξ ▷ t ⤳ b) [ δ ]Eq ≔ (Ξ [ δ ]Eq) ▷ (t [ δ ]) ⤳ b
```

Conversion relative to a set of in-scope equations can then be defined inductively. Our starting point is to copy over the definition of β-conversion given in Section 2.2.4 (specialised to our pair of type formers).

```
data _⊢_~_ (Ξ : Eqs Γ) : Tm Γ A → Tm Γ A → Type where
  -- Equivalence
  rfl~ : Ξ ⊢ t ~ t
  sym~ : Ξ ⊢ t₁ ~ t₂ → Ξ ⊢ t₂ ~ t₁
  _•~_ : Ξ ⊢ t₁ ~ t₂ → Ξ ⊢ t₂ ~ t₃ → Ξ ⊢ t₁ ~ t₃
  -- Congruence
  λ_   : Ξ [ wk ]Eq ⊢ t₁ ~ t₂ → Ξ ⊢ λ t₁ ~ λ t₂
  _·_  : Ξ ⊢ t₁ ~ t₂ → Ξ ⊢ u₁ ~ u₂ → Ξ ⊢ t₁ · u₁ ~ t₂ · u₂
  if   : Ξ ⊢ t₁ ~ t₂ → Ξ ⊢ u₁ ~ u₂ → Ξ ⊢ v₁ ~ v₂
       → Ξ ⊢ if t₁ u₁ v₁ ~ if t₂ u₂ v₂
  -- Computation
  →β  : Ξ ⊢ (λ t) · u ~ t [ < u > ]
  𝔹β₁  : Ξ ⊢ if tt u v  ~ u
  𝔹β₂  : Ξ ⊢ if ff u v  ~ v
```

We account for local equations by defining a type of evidence that a particular equation, t ⤳ b, occurs in an equational context, Ξ: EqVar Ξ t b.

```
data EqVar : Eqs Γ → Tm Γ 𝔹 → 𝔹 → Type where
  ez : EqVar (Ξ ▷ t ⤳ b) t b
  es : EqVar Ξ t b₁ → EqVar (Ξ ▷ u ⤳ b₂) t b₁


eq : EqVar Ξ t b → Ξ ⊢ t ~ ⌜ b ⌝𝔹
```

Note that the congruence rule for "if" here is not **smart** in the sense of **smart case**: we do not introduce equations on the scrutinee in the branches.

```
if : Ξ ⊢ t₁ ~ t₂ → Ξ ▷ t₁ ⤳ tt ⊢ u₁ ~ u₂ → Ξ ▷ t₁ ⤳ ff ⊢ v₁ ~ v₂
  → Ξ ⊢ if t₁ u₁ v₁ ~ if t₂ u₂ v₂
```

We will study the effect of locally introducing equations with rules like this later in section Section 4.4.

Before moving on, we give a couple important definitions.

Definition 4.1.1 (Definitional Inconsistency)
We define definitionally inconsistent equational contexts identically to the dependently typed setting (Remark 3.2.1). That is, contexts in which tt and ff are convertible.

```
def-incon : Eqs Γ → Type
def-incon Ξ ≔ Ξ ⊢ tt ~ ff
```

Again, under definitionally-inconsistent contexts, all terms are convertible.

```
collapse : def-incon Ξ → Ξ ⊢ u ~ v
collapse {u ≔ u} {v ≔ v} tf~ ≔
  u
  ~by sym~ 𝔹β₁
  if tt u v
  ~by if tf~ rfl~ rfl~
  if ff u v
  ~by 𝔹β₂
  v ∎
```

However, because of the lack of computation at the level of types in STLC (that is, the absence of large elimination), we do not get a type-level equality collapse. Definitional inconsistency is therefore a bit less dangerous in the setting of STLC, but we must still keep the consequences it in mind when deciding conversion.

Definition 4.1.2 (Equational Context Equivalence)
We define equivalence of equational contexts observationally: two equational contexts Ξ_1 and Ξ_2 are equivalent if they equate the same sets of terms via conversion _⊢_~_.

```
record _~ᴱᑫˢ_ (Ξ₁ Ξ₂ : Eqs Γ) : Type where field
  to   : Ξ₁ ⊢ t₁ ~ t₂ → Ξ₂ ⊢ t₁ ~ t₂
  from : Ξ₂ ⊢ t₁ ~ t₂ → Ξ₁ ⊢ t₁ ~ t₂
```

4.1.1 Difficulties with Reduction

Rewriting gives a nice intuition for the operational behaviour of these equations (in the context $\Gamma \triangleright \mathrm{t} \rightsquigarrow \mathbf{tt}$, t should reduce to tt), but declarative conversion being an equivalence by definition makes it perhaps more powerful than we might initially expect.

For example, if we try to directly translate this definition of conversion into a small-step reduction relation

```
data _⊢_>_ (Ξ : Eqs Γ) : Tm Γ A → Tm Γ A → Type where
  -- Computation
   →β  : Ξ ⊢ (λ t) · u > t [ < u > ]
  𝔹β₁  : Ξ ⊢ if tt u v  > u
  𝔹β₂  : Ξ ⊢ if ff u v  > v

  -- Rewriting
  rw : EqVar Ξ t b → Ξ ⊢ t > ⌜ b ⌝𝔹

  -- Monotonicity
  λ_   : Ξ [ wk ]Eq ⊢ t₁ > t₂ → Ξ ⊢ λ t₁      > λ t₂
  l·   : Ξ          ⊢ t₁ > t₂ → Ξ ⊢ t₁ · u    > t₂ · u
  ·r   : Ξ          ⊢ u₁ > u₂ → Ξ ⊢ t · u₁    > t · u₂
  if₁  : Ξ          ⊢ t₁ > t₂ → Ξ ⊢ if t₁ u v > if t₂ u v
  if₂  : Ξ          ⊢ u₁ > u₂ → Ξ ⊢ if t u₁ v > if t u₂ v
  if₃  : Ξ          ⊢ v₁ > v₂ → Ξ ⊢ if t u v₁ > if t u v₂
```

while we do at least stay conservative over conversion

```
pres> : Ξ ⊢ t₁ > t₂ → Ξ ⊢ t₁ ~ t₂
```

we find that the induced notion of algorithmic convertibility is much weaker than our declarative specification. Problems arise from how the LHS terms in contextual equations need not themselves be irreducible, so e.g. in the equational context $\bullet \triangleright \mathrm{if\ tt\ tt\ v} \rightsquigarrow \mathbf{ff}$, we can derive tt ~ ff, but not $\mathrm{tt} >^* \mathrm{ff}$ (or $\mathrm{ff} >^* \mathrm{tt}$)

```
ex1 : • ▷ if tt tt v ⇝ ff ⊢ tt ~ ff
ex1 {v ≔ v} ≔
  tt
  ~by sym~ 𝔹β₁
  if tt tt v
  ~by eq ez
  ff ∎

ex2 : ¬ • ▷ if tt ff v ⇝ tt ⊢ tt >* ff
ex2 (rw (es ()) :> _)
```

This reduction relation has other problems as well. In the context $\bullet \triangleright \mathrm{tt} \rightsquigarrow \mathbf{tt}$, reduction is not well-founded[1] and in the context $\bullet \triangleright \mathrm{tt} \rightsquigarrow \mathbf{ff}$, reduction is non-confluent[2].

The situation is slightly improved by explicitly preventing rewriting of terms that are syntactically equal to closed Booleans:

```
𝔹? : Tm Γ A → 𝔹
𝔹? tt ≔ tt
𝔹? ff ≔ tt
𝔹? _ ≔ ff


rw : ¬is 𝔹? t → EqVar Ξ t b → Ξ ⊢ t > ⌜ b ⌝𝔹
```

[1] There is an infinite chain of reduction tt > tt > tt > ....

[2] We can pick two terms u and v such that ¬ u > v, e.g. the Church Booleans u ≔ λ x y. x and v ≔ λ x y. y, and then start with the term if tt u v. We can either reduce with $\beta\mathbb{B}_1$ directly and get if tt u v > u or we can apply the rewrite and follow up with $\beta\mathbb{B}_2$, obtaining if tt u v > if ff u v > v.

_⊢_>_ is now even weaker, and is still non-confluent, but as it turns out, it is strongly normalising! More significantly, we will show that this reduction stays strongly normalising even without the EqVar Ξ t b pre-condition on rw . Intuitively, closed Booleans are irreducible, so reduction chains which collapse the entire 𝔹-typed term to a closed Boolean with rw must terminate at that point, but of course replacing subterms in some large expression with tt or ff can unlock new reductions, so well-foundedness is not completely trivial.

Removing this pre-condition is equivalent to being allowed to “swap” the equational context after every reduction.

```
_>Swap_ : Tm Γ A → Tm Γ A
        → Type
_>Swap_ {Γ ≔ Γ} t₁ t₂
  ≔ (Ξ : Eqs Γ) × Ξ ⊢ t₁ > t₂
```

Intuitively, this is a useful property, because it allows us to freely modify the equational context while performing well-founded induction.

4.2 Normalisation via Completion

In the prior section, we ended by gesturing at a reduction relation similar to _⊢_>_, but without a pre-condition on Boolean rewriting (beyond the LHS not already being a closed Boolean). We will now make this notion concrete, and name it *spontaneous reduction* (𝔹-typed terms may “spontaneously” collapse to tt or ff).

Recall that ¬is 𝔹? here ensures that t is not already a closed Boolean, preventing reductions like tt >! tt.

```
data _>!_ : Tm Γ A → Tm Γ A → Type where
  -- Computation
   →β  : (λ t) · u >! t [ < u > ]
  𝔹β₁  : if tt u v  >! u
  𝔹β₂  : if ff u v  >! v

  -- Spontaneous rewriting
  rw : ¬is 𝔹? t → t >! ⌜ b ⌝𝔹

  -- Monotonicity
  λ_   : t₁ >! t₂ → λ t₁      >! λ t₂
  l·   : t₁ >! t₂ → t₁ · u    >! t₂ · u
  ·r   : u₁ >! u₂ → t · u₁    >! t · u₂
  if₁  : t₁ >! t₂ → if t₁ u v >! if t₂ u v
  if₂  : u₁ >! u₂ → if t u₁ v >! if t u₂ v
  if₃  : v₁ >! v₂ → if t u v₁ >! if t u v₂
```

In Section 4.3 we will prove that _>!_ is strongly normalising. Before we dive into that proof though, we will show how to derive a normalisation algorithm using this result.

The key idea here will be *completion.* We call equational contexts where every LHS is irreducible w.r.t. all other equations *complete*[3].

```
Stk : Eqs Γ → Tm Γ A → Type
Stk Ξ t ≔ Π u → ¬ Ξ ⊢ t > u

_-_ : Π (Ξ : Eqs Γ) → EqVar Ξ t b → Eqs Γ
(Ξ ▷ t ⇝ b)  - ez   ≔ Ξ
(Ξ ▷ u ⇝ b′) - es e ≔ (Ξ - e) ▷ u ⇝ b′

data AllStk (Ξ : Eqs Γ) : Eqs Γ → Type where
  •   : AllStk Ξ •
  _▷_ : AllStk Ξ Ψ
        → Π (e : EqVar Ξ t b) → ¬is 𝔹? t
        → Stk (Ξ - e) t → AllStk Ξ (Ψ ▷ t ⇝ b)

Complete : Eqs Γ → Type
Complete Ξ ≔ AllStk Ξ Ξ
```

Under complete equational contexts Ξ, there are no critical pairs w.r.t. Ξ ⊢_>_ (LHSs cannot overlap). Therefore, we can prove that reduction is confluent (ordinary β-reduction cases are dealt with by switching to parallel reduction [45] - we know the new rw case can only apply if the term is otherwise irreducible from Stk (Ξ - e) t).

[45]: Takahashi (1995), *Parallel Reductions in lambda-Calculus*

```
compl-confl : Complete Ξ → Ξ ⊢ t >* u → Ξ ⊢ t >* v
              → (w : Tm Γ A) × (Ξ ⊢ u >* w × Ξ ⊢ v >* w)
```

We can define algorithmic conversion and, via confluence, prove that declarative conversion is preserved.

```
record _⊢_<~>_ (Ξ : Eqs Γ) (t₁ t₂ : Tm Γ A) : Type where
  constructor _|_
  field
    {common} : Tm Γ A
```

[3] Slightly confusingly, equational contexts being *complete* is required to prove *soundness* of normalisation (to ensure we appropriately identify all convertible terms and do not miss any reductions), rather than completeness (which will ultimately be provable by Ξ ⊢_>_ being contained in Ξ ⊢_~_).

```
    reduces₁    : Ξ ⊢ t₁ >* common
    reduces₂    : Ξ ⊢ t₂ >* common
```

```
<~>-trans : Complete Ξ → Ξ ⊢ t₁ <~> t₂ → Ξ ⊢ t₂ <~> t₃ → Ξ ⊢ t₁ <~> t₃
<~>-trans Ξᶜ (t₁> | t₂>) (t₂>′ | t₃>)
  using w , t₁>′ , t₃>′ ≔ compl-confl Ξᶜ t₂> t₂>′
  ≔ (t₁> ∘* t₁>′) | (t₃> ∘* t₃>′)

<~>-pres : Complete Ξ → Ξ ⊢ t₁ ~ t₂ → Ξ ⊢ t₁ <~> t₂
```

Algorithmic convertibility of stuck terms implies syntactic equality (Stk<~>), so we can further derive uniqueness of normal forms (stuck terms under complete equational context reduction).

```
Stk>* : Stk Ξ t₁ → Ξ ⊢ t₁ >* t₂ → t₁ = t₂
Stk>* ¬t₁> rfl*             ≔ refl
Stk>* ¬t₁> (t₁> :> t₁>*)  ≔ 𝟘-elim (¬t₁> _ t₁>)

Stk<~> : Stk Ξ t₁ → Stk Ξ t₂ → Ξ ⊢ t₁ <~> t₂ → t₁ = t₂
Stk<~> ¬t₁> ¬t₂> (t₁>* | t₂>*) ≔ Stk>* ¬t₁> t₁>* • sym (Stk>* ¬t₂> t₂>*)

nf-uniq : Complete Ξ → Stk Ξ t₁ → Stk Ξ t₂ → Ξ ⊢ t₁ ~ t₂ → t₁ = t₂
nf-uniq Ξᶜ ¬t₁> ¬t₂> t~ ≔ Stk<~> ¬t₁> ¬t₂> (<~>-pres Ξᶜ t~)
```

We now specify the completion algorithm as a function that completes equational contexts while preserving equivalence.

```
complete        : Eqs Γ → Eqs Γ
complete-pres   : Ξ ~ᴱᑫˢ complete Ξ
complete-compl  : Complete (complete Ξ)
```

Under complete equational contexts Ξ, we have shown that algorithmic conversion induced by Ξ ⊢_>eq_ is equivalent to declarative conversion Ξ ⊢_~_. Therefore, we can obtain a sound and complete normalisation algorithm from completion and the existence of a function which fully reduces terms w.r.t. Ξ ⊢_>eq_.

Decidability of convertibility normal forms (terms which are Stk w.r.t. Complete equational contexts) follows from decidability of syntactic equality on first-order datatypes.

“reduce” fully reduces terms w.r.t. _⊢_>_.

```
reduce          : Eqs Γ → Tm Γ A → Tm Γ A
reduce-reduces  : Ξ ⊢ t >* reduce Ξ t
reduce-Stk      : Stk Ξ (reduce Ξ t)
```

```
norm : Eqs Γ → Tm Γ A → Tm Γ A
norm Ξ t ≔ reduce (complete Ξ) t

reduce-pres : Ξ ⊢ t ~ reduce Ξ t
reduce-pres ≔ pres>* reduce-reduces

norm-sound : Ξ ⊢ t₁ ~ t₂ → norm Ξ t₁ = norm Ξ t₂
norm-sound {Ξ ≔ Ξ} {t₁ ≔ t₁} {t₂ ≔ t₂} t~
  ≔ nf-uniq complete-compl reduce-Stk reduce-Stk (
      norm Ξ t₁
      ~by sym~ reduce-pres
      t₁
      ~by complete-pres .to t~
      t₂
      ~by reduce-pres
      norm Ξ t₂ ■)

norm-pres : Ξ ⊢ t ~ norm Ξ t
norm-pres ≔ complete-pres .from reduce-pres

norm-compl : norm Ξ t₁ = norm Ξ t₂ → Ξ ⊢ t₁ ~ t₂
norm-compl {Ξ ≔ Ξ} {t₁ ≔ t₁} {t₂ ≔ t₂} t₌ ≔
```

```
  t₁
  ~by norm-pres
  norm Ξ t₁
  ~by ≡~ t₌
  norm Ξ t₂
  ~by sym~ norm-pres
  t₂ ■
```

There is a remaining subtlety: completion as specified cannot be implemented on definitionally inconsistent contexts. Specifically, it is provable that in all equational contexts satisfying Complete, deriving Ξ ⊢ tt ~ ff is impossible, so clearly completion cannot preserve context equivalence in these cases.

```
complete-not-incon : Complete Ξ → ¬ Ξ ⊢ tt ~ ff

contradiction : 𝟘
contradiction
  ≔ complete-not-incon (complete-compl {Ξ ≔ Ξ⊥}) (complete-pres .to (eq ez))
  where Ξ⊥ ≔ • ▷ tt {Γ ≔ •} ⇝ ff
```

It follows that completion in our setting should be *partial*. We will either complete an equational environment, or discover a syntactically inconsistent equation like tt ⇝ **ff** and conclude that it is definitionally inconsistent.

Our corrected specification of completion is (we fuse the correctness conditions with the definition to simplify the spec)

```
data Complete? (Ξ : Eqs Γ) : Type where
  compl : Π Ξ′ → Ξ ~Eqs Ξ′ → Complete Ξ′ → Complete? Ξ
  !!    : def-incon Ξ → Complete? Ξ
complete : Π (Ξ : Eqs Γ) → Complete? Ξ
```

We also have to update our definition of normal forms. In definitionally inconsistent contexts, all terms are convertible, so our normal forms be characterised by the unit type.

```
Nf : Π Γ (Ξ : Eqs Γ) → Ty → Complete? Ξ → Type
Nf Γ Ξ A (compl Ξ′ _ _) ≔ (t : Tm Γ A) × Stk Ξ′ t
Nf Γ Ξ A (!! _)          ≔ 𝟙
norm : Π (Ξ : Eqs Γ) → Tm Γ A → Nf Γ Ξ A (complete Ξ)

norm-sound    : Ξ ⊢ t₁ ~ t₂ → norm Ξ t₁ = norm Ξ t₂
norm-complete : norm Ξ t₁ = norm Ξ t₂ → Ξ ⊢ t₁ ~ t₂
```

Note that these normal forms do not cleanly embed back into the STLC terms (all information about the structure of the term is lost in the case of inconsistent contexts) but we can still decide equality by first completing the context, and then either syntactically comparing stuck terms (the Stk part is proof-irrelevant and so can be ignored) or immediately returning reflexivity.

Normalisation can then be implemented as before in the case completion succeeds (i.e. returns compl ...) or otherwise can just return ⟨⟩.

```
norm Ξ t with complete Ξ
... | compl Ξ′ _ _ ≔ reduce Ξ′ t , reduce-Stk
... | !! _         ≔ ⟨⟩
```

Of course, this normalisation function is only actually implementable if we can define complete and reduce with all appropriate correctness conditions. Given well-foundedness of _>!_, reduce can be defined very similarly to naive normalisation as in Section 2.4.1 (recurse over the term, contracting redexes where possible, now additionally checking for rewrites by syntactically comparing subterms to LHSs in the equational context). complete then can be implemented by repeatedly reducing LHS terms, with termination justified by extending _>!_ lexicographically over the equational context.

4.3 Strong Normalisation of Spontaneous Reduction

All that remains then is strong normalisation of $\_>_{!}\_$. We will prove this in two steps, using an intermediary notion of "non-deterministic" reduction, $\_>_{\text{N.D.}}\_$: a slightly generalised version of β-reduction, where "if"-expressions can be non-deterministically collapsed to the LHS or RHS branch irrespective of the scrutinee.

- First we will prove that strong normalisability w.r.t. non-deterministic reduction, SN $\_>_{\text{N.D.}}\_$ t, implies SN w.r.t. spontaneous reduction, SN $\_>_{!}\_$ t. We will actually show this on untyped terms (generalising $\_>_{!}\_$ appropriately) to simplify the presentation.
- Then we will show strong normalisation of typed terms w.r.t. $\_>_{\text{N.D.}}\_$ by the technique of computability/(unary) logical relations.

4.3.1 An Untyped Reduction Proof

In this section, we show that the untyped terms which are strongly-normalising w.r.t. non-deterministic reduction are also strongly-normalising w.r.t. spontaneous reduction.

We define untyped terms indexed by the number of variables in the context ("intrinsically well-scoped"). Note that in this section, the symbols Γ, Δ, Θ denote untyped contexts (i.e. natural numbers) rather than lists of types.

```
vz  : Var (su Γ)
vs  : Var Γ → Var (su Γ)

`_  : Var Γ → Tm Γ
_·_ : Tm Γ → Tm Γ → Tm Γ
λ_  : Tm (su Γ) → Tm Γ
tt  : Tm Γ
ff  : Tm Γ
if  : Tm Γ → Tm Γ → Tm Γ → Tm Γ
```

In this section, we will be dealing with quite a few distinct reduction relations at a fine-grained level of detail. To assist with this, we define generically the monotonic closure of term relations, _[_]>_. This lets us lift term relations _>_ over our various term formers.

```
_[_]>_ : Tm Γ → (Π {Γ} → Tm Γ → Tm Γ → Type)
         → Tm Γ → Type
```

```
⟪_⟫ : t₁ > t₂ → t₁ [ _>_ ]> t₂
l·  : t₁ [ _>_ ]> t₂ → t₁ · u [ _>_ ]> t₂ · u
·r  : u₁ [ _>_ ]> u₂ → t · u₁ [ _>_ ]> t · u₂
λ_  : t₁ [ _>_ ]> t₂ → λ t₁ [ _>_ ]> λ t₂
if₁ : t₁ [ _>_ ]> t₂ → if t₁ u v [ _>_ ]> if t₂ u v
if₂ : u₁ [ _>_ ]> u₂ → if t u₁ v [ _>_ ]> if t u₂ v
if₃ : v₁ [ _>_ ]> v₂ → if t u v₁ [ _>_ ]> if t u v₂
```

Monotonic closure is functorial over mappings between the closed-over reduction relations.

```
map> : (Π {Γ} {t u : Tm Γ} → t >₁ u → t >₂ u)
       → t [ _>₁_ ]> u → t [ _>₂_ ]> u
```

We can now define our reduction relations as a “step” relation containing the interesting cases, lifted using _[_]>. Ordinary β-reduction can then just be defined as the monotonic closure of the computation rules for → and 𝔹:

```
data β-step : Tm Γ → Tm Γ → Type where
   →β  : β-step ((λ t) · u) (t [ < u > ])
  𝔹β1  : β-step (if tt u v) u
  𝔹β2  : β-step (if ff u v) v
_>β_ : Tm Γ → Tm Γ → Type
_>β_ ≔ _[ β-step ]>_
```

Spontaneous reduction _$>_{!}$_ in this section refers only to the relation which rewrites terms to closed Booleans (as long as the terms not already syntactically equal to tt or ff); we do not, by default, include β-reductions as well. We also do not require the LHS term to have Boolean type, which we are somewhat forced into given we are working with untyped terms. We therefore will end up proving strong normalisation of a larger relation than our concrete goal of *typed* spontaneous (plus β) reduction.

```
data !-step : Tm Γ → Tm Γ → Type where
  !TT : ¬is 𝔹? t → !-step t tt
  !FF : ¬is 𝔹? t → !-step t ff
_>!_ : Tm Γ → Tm Γ → Type
_>!_ ≔ _[ !-step ]>_
```

Non-deterministic reduction treats “if”-expressions like non-deterministic choices, ignoring the scrutinee.

```
data N.D.-step : Tm Γ → Tm Γ → Type where
   →β : N.D.-step ((λ t) · u) (t [ < u > ])
  ndl : N.D.-step (if t u v) u
  ndr : N.D.-step (if t u v) v
_>N.D._ : Tm Γ → Tm Γ → Type
_>N.D._ ≔ _[ N.D.-step ]>_
```

We need one more monotonic relation on terms, _$>_{\mathbb{B}}$_, where t $>_{\mathbb{B}}$ u holds when u is syntactically equal to t except for replacing a single arbitrary subterm with a closed Boolean (tt or ff).

```
_>𝔹_ : Tm Γ → Tm Γ → Type
_>𝔹_ ≔ _[ (λ _ u → is 𝔹? u) ]>_
```

Our overall goal is to prove that all terms which are strongly-normalising w.r.t. non-deterministic reduction are also strongly-normalising w.r.t. spontaneous reduction plus β rules, _$>_{\beta !}$_.

```
_>β!_ : Tm Γ → Tm Γ → Type
_>β!_ ≔ _[ _>β_ ∣ _>!_ ]_
snN.D.-snβ! : SN _>N.D._ t → SN _>β!_ t
```

The key lemma we need to show is that _$>_{\mathbb{B}}{}^{*}$_ (i.e. the relation defined by replacements of arbitrary subterms of the LHS term with closed Booleans) commutes with non-deterministic reduction:

```
𝔹*/nd-comm> : t >𝔹* u → u >N.D. v → (w : Tm Γ) × t >N.D. w × w >𝔹* v
```

Note that _$>_{\text{N.D.}}$_ does not commute with _$>_{!}$_ in the same way. _$>_{\text{N.D.}}$_ includes the β-rule for functions, and so any reduction relation which commutes with _$>_{\text{N.D.}}$_ must be stable under substitution. Spontaneous reduction is not stable under substitution, because e.g. we can rewrite ` i >! tt, but if we apply the substitution ff / i to both sides then we are left with ff >! tt which is impossible (the LHS of _>!_ cannot be tt or ff).

Luckily, _>𝔹*_ does not face the same issue: tt >𝔹 ff and ff >𝔹 tt are valid. We can prove 𝔹*/nd-comm> by checking all the cases for individual N.D.-steps/single Boolean rewrites (_>𝔹_) and then extending over the monotonic closure of N.D.-step and transitive closure of _>𝔹_. The relevant cases are:

- When the N.D.-step is a →β contraction, then the Boolean rewrite (_>𝔹_) must have occurred inside the lambda body or the argument, and so we can first β-reduce and then rewrite (multiple times, if the rewrite took place inside the argument specifically[4]) to get back to the same term.
- When the step is a non-deterministic choice, the Boolean rewrite must have occurred inside the scrutinee, LHS, or RHS, of the "if" expression. We can instead perform the non-deterministic choice before the rewrite, and then either get back to the term immediately (if the rewrite was wither inside the scrutinee or the dropped branch of the "if"), or apply the rewrite again to the retained branch.

```
data _>Tms𝔹*_ : Tms Δ Γ → Tms Δ Γ → Type where
  refl : δ >Tms𝔹* δ
  _,_ : δ >Tms𝔹* σ → t >𝔹* u → (δ ▷ t) >Tms𝔹* (σ ▷ u)
_[_]𝔹>* : t >𝔹* u → δ >Tms𝔹* σ → t [ δ ] >𝔹* u [ σ ]
𝔹/N.D.-comm : t >𝔹 u → N.D.-step u v → (w : Tm Γ) × N.D.-step t w × w >𝔹* v
𝔹/N.D.-comm (l· (λ p))          →β
  ≔ _ , →β , ⟪ p ⟫* [ refl ]𝔹>*
𝔹/N.D.-comm (·r {t ≔ λ t} p) →β
  ≔ _ , →β , rfl* {x ≔ t} [ refl {δ ≔ id} , (p :: rfl*) ]𝔹>*
𝔹/N.D.-comm (if₁ p) ndl ≔ _ , ndl , rfl*
𝔹/N.D.-comm (if₂ p) ndl ≔ _ , ndl , ⟪ p ⟫*
𝔹/N.D.-comm (if₃ p) ndl ≔ _ , ndl , rfl*
𝔹/N.D.-comm (if₁ p) ndr ≔ _ , ndr , rfl*
𝔹/N.D.-comm (if₂ p) ndr ≔ _ , ndr , rfl*
𝔹/N.D.-comm (if₃ p) ndr ≔ _ , ndr , ⟪ p ⟫*
```

_[_]𝔹>* here witnesses a generalisation of _>𝔹*_ being stable under substitution. Specifically, we allow the substitute terms to also be reduced via _>𝔹*_.

We can also prove that spontaneous reduction alone is strongly normalising by structural induction on terms (once we rewrite a term to a Boolean, it cannot reduce further).

```
sn! : Π (t : Tm Γ) → SN _>!_ t
```

Using our commuting lemma to ensure we keep making progress w.r.t. non-deterministic reduction, we can prove by mutual well-founded induction on non-deterministic and spontaneous reduction that the strongly normalising terms w.r.t. _>N.D._ are exactly those which are strongly normalising w.r.t. _>N.D.!_ (interleaved _>N.D._ and _>!_).

```
_>N.D.!_ : Tm Γ → Tm Γ → Type
_>N.D.!_ ≔ _[ _>N.D._ | _>!_ ]_
!𝔹> : t >! u → t >𝔹 u
snN.D.!  : t >𝔹* u → SN _>N.D._ t → SN _>!_ u → SN _>N.D.!_ u
snN.D.!> : t >𝔹* u → SN _>N.D._ t → SN _>!_ u → u >N.D.! v
           → SN _>N.D.!_ v
snN.D.! p N.D.^Acc !^Acc ≔ acc (snN.D.!> p N.D.^Acc !^Acc)
snN.D.!> p (acc N.D.^Acc) !^Acc        (in₁ q)
  using v , q′ , p′ ≔ 𝔹*/nd-comm> p q
  ≔ snN.D.! p′ (N.D.^Acc q′) (sn! _)
snN.D.!> p N.D.^Acc         (acc !^Acc) (in₂ q)
  ≔ snN.D.! (p <: !𝔹> q) N.D.^Acc (!^Acc q)
```

Note that we generalise the inductive hypothesis over _>𝔹*_ here to account for subterms getting rewritten to Booleans. We name the lemma that spontaneous reduction is included in _>𝔹_, !𝔹>, and prove it by considering !-step cases and lifting with map>.

[4] E.g. given u >𝔹 u′, then we can get from (λ x. f x x) u to f u′ u′ by first β-contracting to get f u u and then applying the rewrite twice.

```
snN.D.-snN.D.! : SN _>N.D._ t → SN _>N.D.!_ t
snN.D.-snN.D.! N.D.Acc ≔ snN.D.! rfl* N.D.Acc (sn! _)
```

Finally, we recover our original goal

```
snN.D.-snβ! : SN _>N.D._ t → SN _>β!_ t
```

from _>β!_ reduction being included inside _>N.D.!_.

```
β-N.D. : β-step t u → N.D.-step t u
β-N.D. →β ≔ →β
β-N.D. 𝔹β1  ≔ ndl
β-N.D. 𝔹β2  ≔ ndr
snN.D.-snβ! N.D.Acc
  ≔ accessible (map+ (map> β-N.D.) (λ p → p)) (snN.D.-snN.D.! N.D.Acc)
```

4.3.2 Strong Normalisation of Non-Deterministic Reduction

We now return to the world of simply typed terms in order to prove that all such terms are strongly normalising w.r.t. non-deterministic reduction. For this, we will use the technique of logical relations (also known as computability [108] or reducibility candidates). The specific proof we attempt is based on Girard's proof of strong normalisation for STLC in chapter 6 of [109], translated into Agda by András Kovács [110].

[108]: Tait (1967), *Intensional Interpretations of Functionals of Finite Type I*

[109]: Girard et al. (1989), *Proofs and Types*

[110]: Kovács (2020), *StrongNorm.agda*

To simplify the proof, we will assume all substitution equations hold definitionally. For STLC, we can prove these equations by induction on the syntax following [44], so to justify this decision, we merely need to reflect these propositional equations as definitional ones (by conservativity of ETT over ITT [63, 65] we, in principle, lose nothing by simplifying the presentation in this way).

[44]: Altenkirch et al. (2025), *Substitution without copy and paste*

[63]: Hofmann (1995), *Conservativity of Equality Reflection over Intensional Type Theory*

[65]: Winterhalter et al. (2019), *Eliminating reflection from type theory*

We recall the definition of non-deterministic reduction.

```
data _>N.D._ : Tm Γ A → Tm Γ A → Type where
  -- Computation
  →β  : (λ t) · u >N.D. t [ < u > ]
  ndl  : if t u v  >N.D. u
  ndr  : if t u v  >N.D. v
  -- Monotonicity
  λ_   : t1 >N.D. t2  → λ t1      >N.D. λ t2
  l·   : t1 >N.D. t2  → t1 · u    >N.D. t2 · u
  ·r   : u1 >N.D. u2  → t · u1    >N.D. t · u2
  if1  : t1 >N.D. t2  → if t1 u v >N.D. if t2 u v
  if2  : u1 >N.D. u2  → if t u1 v >N.D. if t u2 v
  if3  : v1 >N.D. v2  → if t u v1 >N.D. if t u v2
```

We define computability (i.e. the logical relation) as follows

```
P : Π Γ A → Tm Γ A → Type
P Γ 𝔹 t ≔ SN _>N.D._ t
P Γ (A → B) t
  ≔ Π {Δ} (δ : Ren Δ Γ) {u} → P Δ A u → P Δ B ((t [ δ ]) · u)
```

The resemblance to Val in NbE (Section 2.4) should not be so surprising. If we naively attempt to prove strong normalisation by direct structural induction on terms, we will again get stuck in the case of application, where the LHS and RHS being strongly normalising does not imply that their application is.

Like in NbE, we can parameterise function computability over renamings or thinnings, corresponding to the presheaf exponential over the category of renamings or the category of thinnings. We choose renamings here only for convenience.

Our analogue of NbE environments is evidence of computability of each of the terms we will substitute every variable for.

```
data Ps (Δ : Ctx) : Π Γ → Sub Δ Γ → Type where
  ε : Ps Δ • ε
  _,_ : Ps Δ Γ δ → P Δ A t → Ps Δ (Γ ▷ A) (δ , t)
```

We can prove that non-deterministic reduction is stable under substitutions and inverting renamings.

```
_[_]>    : t1 >N.D. t2 → (δ : Tms[ q ] Δ Γ) → t1 [ δ ] >N.D. t2 [ δ ]
[_]>^-1_ : Π (δ : Ren Δ Γ) → t [ δ ] >N.D. t[]′
           → (t′ : Tm Γ A) × (t >N.D. t′ × t′ [ δ ] = t[]′)
```

These stability properties follow pretty directly from induction on the definition of reduction (plus definitional substitution equations). E.g. for the case of applying a substitution to $\rightarrow\beta$, we need $((\lambda\ t) \cdot u)\ [\ \delta\ ] >_{\text{N.D.}} t\ [\ <u>\ ]\ [\ \delta\ ]$, which is satisfied immediately with $\rightarrow\beta$ because

```
((λ t) · u) [ δ ] ≡ (λ (t [ δ ^ A ])) · (u [ δ ])
```

and

```
t [ < u > ] [ δ ] ≡ t [ δ , (u [ δ ]) ] ≡ t [ δ ^ A ] [ < u [ δ ] > ]
```

From stability of reduction under inverted renamings, we can show that SN is stable under (forwards) renaming, and therefore computability is also. Note that we needed stability w.r.t. inverted renamings to show this because the reduction itself appears in contravariant position (i.e. left of arrow) in the type of acc (intuitively, we are transforming reduction chains starting from t [δ] into reduction chains starting from t).

```
_[_]SN : SN _>N.D._ t → Π (δ : Ren Δ Γ) → SN _>N.D._ (t [ δ ])
acc t^Acc [ δ ]SN
  ≔ acc λ p → let t′ , p′ , q ≔ [ δ ]>^-1 p
               in transp (SN _>N.D._) q (t^Acc p′ [ δ ]SN)
```

```
_[_]P  : P Γ A t → Π (δ : Ren Δ Γ) → P Δ A (t [ δ ])
_[_]Ps : Ps Δ Γ δ → Π (σ : Ren Θ Δ) → Ps Θ Γ (δ ; σ)
```

By analogous reasoning, strong normalisation is also stable under inverting substitution.

```
[_]SN^-1_ : Π (δ : Tms[ q ] Δ Γ) → SN _>N.D._ (t [ δ ]) → SN _>N.D._ t
[ δ ]SN^-1 (acc t[]^Acc)
  ≔ acc λ p → [ δ ]SN^-1 (t[]^Acc (p [ δ ]>))
```

We are now ready to prove the fundamental theorem: t [δ] is computable as long as all terms in δ are.

```
fndThm : Π (t : Tm Γ A) → Ps Δ Γ δ → P Δ A (t [ δ ])
```

To prove the fundamental theorem, we need a couple of lemmas. Specifically, that it is possible to derive strong normalisation from computability (P-SN) and that if all immediate reducts of a term (not headed by λ-abstraction) are computable, then the original term must be also (P<). These lemmas resemble quoting and unquoting in NbE.

```
λ? : Tm Γ A → 𝔹
λ? (λ _) ≔ tt
λ? _ ≔ ff

P-SN : P Γ A t → SN _>N.D._ t
P<   : ¬is λ? t → (Π {t′} → t >N.D. t′ → P Γ A t′) → P Γ A t
```

The fundamental theorem is proved by induction on terms, similarly to evaluation in NbE. We use the fact that tt, ff and fresh variables are trivially computable (there are no reductions with these constructs on the LHS). The cases for λ-abstraction and “if” are more complicated, so we will cover these separately.

```
lookupP : Π (i : Var Γ A) → Ps Δ Γ δ → P Δ A (i [ δ ])

tt-sn : SN _>N.D._ (tt {Γ ≔ Γ})
tt-sn ≔ acc λ ()

ff-sn : SN _>N.D._ (ff {Γ ≔ Γ})
ff-sn ≔ acc λ ()

`P : P Γ A (` i)
`P ≔ P< ⟨⟩ λ ()

fndThm-λ : (Π {u} → P Γ A u → P Γ B (t [ < u > ]))
           → SN _>N.D._ t → P Γ A u → SN _>N.D._ u → P Γ B ((λ t) · u)
fndThm-if : SN _>N.D._ t → P Γ A u → SN _>N.D._ u → P Γ A v → SN _>N.D._ v
           → P Γ A (if t u v)

fndThm (` i)     ρ ≔ lookupP i ρ
fndThm (λ t)     ρ
  ≔ λ σ uᴾ → fndThm-λ (λ uᴾ′ → fndThm t ((ρ [ σ ]Ps) , uᴾ′))
                      (P-SN (fndThm t ((ρ [ σ ⁺ _ ]Ps) , `P)))
                      uᴾ (P-SN uᴾ)
fndThm (t · u)   ρ ≔ fndThm t ρ id (fndThm u ρ)
fndThm tt        ρ ≔ tt-sn
fndThm ff        ρ ≔ ff-sn
fndThm (if t u v) ρ
  ≔ fndThm-if (fndThm t ρ) uᴾ (P-SN uᴾ) vᴾ (P-SN vᴾ)
  where uᴾ ≔ fndThm u ρ
        vᴾ ≔ fndThm v ρ
```

To prove the fundamental theorem in the case of λ-abstractions and “if” expressions, we repeatedly appeal to P< to step along the chain of reductions, and rely on SN of subterms to induct w.r.t. reduction order in the cases where a subterm is reduced. When we finally hit $\rightarrow\beta$ or ndl/ndr, we return computability of the reduct. To carry along computability evidence until this point, we also need that computability is stable under reduction, P>.

```
P> : t₁ >N.D. t₂ → P Γ A t₁ → P Γ A t₂

fndThm-λ> : (Π {u} → P Γ A u → P Γ B (t [ < u > ]))
  → SN _>N.D._ t → P Γ A u → SN _>N.D._ u
  → (λ t) · u >N.D. t′ → P Γ B t′

fndThm-λ tᴾ tᴬᶜᶜ uᴾ uᴬᶜᶜ ≔ P< ⟨⟩ (fndThm-λ> tᴾ tᴬᶜᶜ uᴾ uᴬᶜᶜ)

fndThm-λ> tᴾ tᴬᶜᶜ     uᴾ uᴬᶜᶜ        →β
  ≔ tᴾ uᴾ
fndThm-λ> tᴾ tᴬᶜᶜ     uᴾ (acc uᴬᶜᶜ) (·r u>)
  ≔ fndThm-λ tᴾ tᴬᶜᶜ (P> u> uᴾ) (uᴬᶜᶜ u>)
```

```
fndThm-λ> t^P (acc t^Acc) u^P u^Acc         (l· (λ t>))
  ≔ fndThm-λ (λ {u ≔ u′} u^P′ → P> (t> [ < u′ > ]>) (t^P u^P′))
             (t^Acc t>) u^P u^Acc

fndThm-if> : SN _>N.D._ t → P Γ A u → SN _>N.D._ u
           → P Γ A v → SN _>N.D._ v
           → if t u v >N.D. t′ → P Γ A t′
fndThm-if t^Acc u^P u^Acc v^P v^Acc ≔ P< ⟨⟩ (fndThm-if> t^Acc u^P u^Acc v^P v^Acc)
fndThm-if> t^Acc       u^P u^Acc       v^P v^Acc       ndl ≔ u^P
fndThm-if> t^Acc       u^P u^Acc       v^P v^Acc       ndr ≔ v^P
fndThm-if> (acc t^Acc) u^P u^Acc       v^P v^Acc       (if_1 t>)
  ≔ fndThm-if (t^Acc t>) u^P u^Acc v^P v^Acc
fndThm-if> t^Acc       u^P (acc u^Acc) v^P v^Acc       (if_2 u>)
  ≔ fndThm-if t^Acc (P> u> u^P) (u^Acc u>) v^P v^Acc
fndThm-if> t^Acc       u^P u^Acc       v^P (acc v^Acc) (if_3 v>)
  ≔ fndThm-if t^Acc u^P u^Acc (P> v> v^P) (v^Acc v>)
```

We now prove the remaining lemmas by recursion on types.

- Stability of computability under reduction is proved by considering larger and larger spines (always applying the reduction to the LHS) until we reach base 𝔹 type.
- Mapping from computability to strong normalisation is achieved by repeatedly applying computability of →-typed terms to a fresh variable, and then taking advantage of how strong normalisability is stable under taking subterms and renaming to get back to SN of the original →-typed term.

```
P> {A ≔ 𝔹}         t> (acc t^Acc) ≔ t^Acc t>
P> {A ≔ A → B} t> t^P ≔ λ δ u^P → P> (l· (t> [ δ ]>)) (t^P δ u^P)

SN-l· : SN _>N.D._ (t · u) → SN _>N.D._ t
SN-l· (acc tu^Acc) ≔ acc λ p → SN-l· (tu^Acc (l· p))

P-SN {A ≔ 𝔹}         t^Acc ≔ t^Acc
P-SN {A ≔ A → B} t^P
  ≔ [ wk ]SN^-1 SN-l· (P-SN (t^P wk (`P {i ≔ vz})))
```

P< (all reducts of a term being computable implies the term itself is) is a bit more complicated.

We mutually prove a more specialised version for the case of applications, P<·, where we have direct computability of the RHS and know every term the LHS reduces to is computable. We prove this by appealing to P<: if the reduction occurs in the LHS, we can obtain computability of the application immediately by combining computability info, while if the reduction occurs in the RHS, we proceed by induction w.r.t. reduction order (computability of the RHS implies it is strongly normalising). We avoid needing to consider the case where the overall application contracts with →β by assuming the LHS is not λ-abstraction headed.

Then to prove P< itself at →-type, we take advantage of having access to computability of the argument to apply P<·.

```
P<·  : ¬is λ? t → (Π {t′} → t >N.D. t′ → P Γ (A → B) t′)
     → P Γ A u → SN _>N.D._ u → P Γ B ((t · u))
P<·> : ¬is λ? t → (Π {t′} → t >N.D. t′ → P Γ (A → B) t′)
     → P Γ A u → SN _>N.D._ u → (t · u) >N.D. t′ → P Γ B t′
P<· ¬λ t^P u^P u^Acc ≔ P< ⟨⟩ (P<·> ¬λ t^P u^P u^Acc)
P<·> ¬λ t^P u^P u^Acc         (l· t>) ≔ t^P t> id u^P
P<·> ¬λ t^P u^P (acc u^Acc) (·r u>) ≔ P<· ¬λ t^P (P> u> u^P) (u^Acc u>)
P< {A ≔ 𝔹}                    ¬λ t^P ≔ acc λ p → t^P p
```

```
P< {A ≔ A → B} {t ≔ t} ¬λ tᴾ
  ≔ 𝛌 δ uᴾ → P<· (¬λ [ δ ]¬ƛ)
    (𝛌 p σ uᴾ′ → let _ , p′ , q ≔ [ δ ]>⁻¹ p
                 in transp (𝛌 □ → P _ B ((□ [ σ ]) · _)) q
                      (tᴾ p′ (δ ; σ) uᴾ′))
    uᴾ (P-SN uᴾ)
```

Now to obtain strong normalisation, we merely need to derive computability of all variables in the identity substitution, so we can apply the fundamental theorem and follow it up with P-SN.

```
idᴾ : Ps Γ Γ id
idᴾ {Γ ≔ •}       ≔ ε
idᴾ {Γ ≔ Γ ▷ A} ≔ idᴾ [ wk ]Ps , `P

sn : SN _>N.D._ t
sn {t ≔ t} ≔ P-SN (fndThm t idᴾ)
```

4.4 Locally Introducing Equations

Back in Section 4.1, we discussed potentially enhancing our notion of conversion-modulo-equations by introducing new equations on the scrutinee inside the LHS and RHS branches of "if"-expressions.

$$\begin{aligned}&\mathsf{if} \;\mathbf{:}\; \Xi \vdash \mathsf{t}_1 \sim \mathsf{t}_2 \;\mathbf{\rightarrow}\; \Xi \rhd \mathsf{t}_1 \;\rightsquigarrow\; \mathbf{tt} \vdash \mathsf{u}_1 \sim \mathsf{u}_2 \;\mathbf{\rightarrow}\; \Xi \rhd \mathsf{t}_1 \;\rightsquigarrow\; \mathbf{ff} \vdash \mathsf{v}_1 \sim \mathsf{v}_2 \\ &\quad \mathbf{\rightarrow}\; \Xi \vdash \mathsf{if}\ \mathsf{t}_1\ \mathsf{u}_1\ \mathsf{v}_1 \sim \mathsf{if}\ \mathsf{t}_2\ \mathsf{u}_2\ \mathsf{v}_2\end{aligned}$$

I would argue this rule is about as close as we can get to a simply-typed analogue of Boolean **smart case**. Of course, typeability of STLC is independent of conversion, so this rule does not expand the expressivity of the language (i.e. make more terms typeable) like dependent **smart case** does, but it does still expand our notion of convertibility to somewhere between pure β-equivalence and adding in full η-equality of Booleans.

For example, in the empty equational context, we can now obtain if t t t ~ if t tt ff, but unlike Boolean η, we cannot simplify further to just t. We also cannot derive commuting conversions (Example 3.5.1). We argue that this more limited notion of conversion can be advantageous though. As mentioned in Section 3.5, known algorithms which can decide η-equality for positive types are quite inefficient (e.g. renormalising terms 2^n times where n is the number of distinct neutral Boolean subterms). I claim that we can take a smarter approach with _⊢_~_. Specifically, we can split on Boolean neutrals only at stuck "if"-expressions, and normalise the left and right branch just once under the corresponding equation (e.g. given if t u v, we must normalise u under t ~ tt, but not t ~ ff).

Justifying normalisation for this theory is quite subtle, however. Retaining our strategy of first completing the equational context and then reducing seems promising, but we now hit a new case in reduction where, after recursing into the LHS or RHS of an "if" expression, we must call back into completion again. Even though "if"-expressions only add equations one-at-a-time, completion might have to run for many iterations (i.e. if the new equation unblocks existing neutral LHSs) so the termination metric here is non-obvious (if indeed this algorithm does actually terminate).

Mutually calling into completion during normalisation when recursing under stuck "if"-expressions is exactly the approach I am employing in my SC$^{\text{Bool}}$ typechecker (Section 5.4).

Remark 4.4.1 (Adding One Equation to a Complete Equational Context can Trigger an Arbitrary Number of Completion Iterations)
To show that we cannot meaningfully take advantage of prior completion evidence and the fact we only introduce one equation at a time, we construct an example pair of Boolean equations which requires an arbitrarily high number of iterations to complete.
Concretely, let us first consider the equation x ⇝ tt (in a context where x : 𝔹). Clearly the equational context containing only this equation is complete. If we then add the equation if x y i $>_{\mathsf{Rw}}$ tt (in this example, the letters x, y, z and i, j, k all stand for distinct 𝔹-typed variables in the context), clearly, the LHS is reducible if x y i > if tt y i > y, so the completed set of equations becomes

x ⇝ tt
y ⇝ tt

Now let us instead consider the pair of equations

if (if x y i) x j ⇝ tt
if x y i ⇝ tt

The first equation's LHS is now reducible (to x), but then this returns us to the original equation pair:

x ⇝ tt
if x y i ⇝ tt

To clarify the pattern, we repeat it once more, now considering the pair of equations:

```
if (if x y i) x j  ⇝  tt
if (if (if x y i) x j) (if x y i) k  ⇝  tt
```

The second equation's LHS is now the immediately-reducible (to if x y i). The general construction we are employing here is to repeatedly replace the smaller LHS, u, with if t u v where t is the larger LHS and v is some arbitrary 𝔹-typed term. Given the other equation must be of the form t ⇝ tt, this new LHS must reduce down to u. The equational context containing only the smallest equation is always be complete, but to complete the extended equational context, completion must alternate between reducing each of the LHSs exactly as many times as we have repeated the construction.

I leave the question of decidability of _⊢_~_, with the local-equation-introducing "if" rule, open.

We also leave discussion of how to deal with more general classes of equations (e.g. at types other than 𝔹) for the coming chapters, as there is not too much insight to be gained by focusing on the special case of simple types (in some ways, STLC makes such extensions more challenging, as the expressivity of dependent types gives us ways to encode many type formers in terms of simpler ones - e.g. coproducts in terms of Booleans and large elimination).

A Minimal Language with Smart Case 5

In this chapter, we introduce and study a minimal dependently-typed language featuring a **smart case**-like elimination principle for Booleans. We name this language SC$^{\text{Bool}}$. Our large chunk of the chapter is dedicated to explaining why normalisation for this theory is so challenging, with examples. We will also detail the core ideas behind the Haskell typechecker for an extended version of this language.

5.1 Syntax

When moving from STLC with local equations to dependent types, we note while equations of course must depend on the context (i.e. the LHS or RHS terms can embed variables), it is also sometimes desirable for types in the context to depend on local equations. For example, in a context where we have x : 𝔹, y : IF x 𝔹 A and the (definitional) equation x ≡ tt holds, we have y : 𝔹 (congruence of definitional equality), and so ought to be able bind z : IF y B C.

To support this, we fuse the ordinary and equational context: contexts can now be extended either with types (introducing new variables) or definitional equations (expanding conversion).

We build upon our quotiented, explicit-substitution syntax laid out in Section 2.3. Again, we have four sorts:

```
Ctx  : Type
Ty   : Ctx → Type
Tm   : Π Γ → Ty Γ → Type
Tms  : Ctx → Ctx → Type
```

We carry over all the substitution laws, the existence of context extension and the term/type formers associated with Π and 𝔹 types, except term-level (dependent) "if". In SC$^{\text{Bool}}$, "if" will be "smart" in the sense that it will add equations to the context in the left and right branches, as opposed to requiring an explicit motive.

In principle, we could also make type-level, "large" IF "smart" in the same way, adding equations to the contexts the LHS and RHS types are defined in. We avoid considering this here only for simplicity.

We start by defining the obvious embedding of Booleans into SC$^{\text{Bool}}$, and prove the substitution law on embedded Booleans by cases.

```
⌜_⌝𝔹 : 𝔹 → Tm Γ 𝔹
⌜ tt ⌝𝔹 :≡ tt
⌜ ff ⌝𝔹 :≡ ff

⌜⌝𝔹[] : ⌜ b ⌝𝔹 [ δ ] =Tm= refl 𝔹[] ⌜ b ⌝𝔹
⌜⌝𝔹[] {b :≡ tt} :≡ tt[]
⌜⌝𝔹[] {b :≡ ff} :≡ ff[]
```

The key idea behind SC$^{\text{Bool}}$ is to allow extending contexts with Boolean equations which we expect to hold definitionally.

```
_▷_ ⤳ _ : Π Γ → Tm Γ 𝔹 → 𝔹 → Ctx
```

We need to define an analagous rule to _, _ for context extension by equations. Concretely, the question is this: given a substitution δ : Tms Δ Γ, what additional information do we need to map from Γ extended by a new equation, Γ ▷ t > eq b? Recall that local equations are used in terms/types to derive convertibility, so I claim that the appropriate notion here is that t and b are convertible in the new context Δ (i.e. with the substitution applied).

```
_,↝_ : Π (δ : Tms Δ Γ) → t [ δ ] =^{Tm₌ refl 𝔹[]} ⌜ b ⌝𝔹
        → Tms Δ (Γ ▷ t ↝ b)
```

Note that requiring convertibility evidence (as opposed to e.g. evidence of the substituted rewrite exactly occurs somewhere in Δ) enables removing rewrites from contexts when they become redundant.

We now give also the associated naturality laws and projections:

```
,;↝  : Π {δ : Tms Δ Γ} {σ : Tms Θ Δ} {t₌}
     → (δ ,↝ t₌) ; σ
     = (δ ; σ) ,↝ (transp (Tm Θ) 𝔹[] (t [ δ ; σ ])
                   = by cong (transp (Tm Θ) 𝔹[]) (sym [][])
                  transp (Tm Θ) 𝔹[] (transp (Tm Θ) [][]Ty (t [ δ ] [ σ ]))
                   = by coh[][] {p ≔ 𝔹[]}
                  transp (Tm Θ) 𝔹[] (transp (Tm Δ) 𝔹[] (t [ δ ]) [ σ ])
                   = by cong (transp (Tm Θ) 𝔹[]) (cong (_[ σ ]) t₌)
                  transp (Tm Θ) 𝔹[] (⌜ b ⌝𝔹 [ σ ])
                   = by ⌜⌝𝔹[]
                  ⌜ b ⌝𝔹 ■)
π₁↝   : Tms Δ (Γ ▷ t ↝ b) → Tms Δ Γ
π₂↝   : Π (δ : Tms Δ (Γ ▷ t ↝ b))
      → t [ π₁↝ δ ] =^{Tm₌ refl 𝔹[]} ⌜ b ⌝𝔹
▷η↝   : δ = π₁↝ δ ,↝ π₂↝ δ
π₁↝,  : Π {t₌ : t [ δ ] =^{Tm₌ (refl {x ≔ Δ}) 𝔹[]} ⌜ b ⌝𝔹}
      → π₁↝ (δ ,↝ t₌) = δ
π₁eq; : π₁↝ (δ ; σ) = π₁↝ δ ; σ
```

Note that $\pi_{2\rightsquigarrow}$ id allows us to make use of the most recently bound equation in the context as convertibility evidence.

We define derived notions of weakening contexts by assuming new equations, wk$_{\rightsquigarrow}$, instantiating contextual equations with evidence of convertibility, <_> ↝, and finally functoriality of context extension by equations, _ˆ_ ↝ _

```
wk↝ : Tms (Γ ▷ t ↝ b) Γ
wk↝ ≔ π₁↝ id

<_> ↝ : t = ⌜ b ⌝𝔹 → Tms Γ (Γ ▷ t ↝ b)
<_> ↝ {t ≔ t} {b ≔ b} t₌
  ≔ id
  ,↝ (transp (Tm _) 𝔹[] (t [ id ])
   = by cong (transp (Tm _) 𝔹[]) (shift [id])
  transp (Tm _) (sym [id]Ty • 𝔹[]) t
   = by cong (transp (Tm _) (sym [id]Ty • 𝔹[])) t₌
  transp (Tm _) (sym [id]Ty • 𝔹[]) ⌜ b ⌝𝔹
   = by coh⌜⌝𝔹 {A₌ ≔ sym [id]Ty • 𝔹[]}
  ⌜ b ⌝𝔹 ■)

_ˆ_ ↝ _ : Π (δ : Tms Δ Γ) t b
        → Tms (Δ ▷ transp (Tm Δ) 𝔹[] (t [ δ ]) ↝ b) (Γ ▷ t ↝ b)
δ ˆ t ↝ b
  ≔ (δ ; wk↝)
  ,↝ (transp (Tm _) 𝔹[] (t [ δ ; wk↝ ])
      = by cong (transp (Tm _) 𝔹[]) (sym [][])
     transp (Tm _) 𝔹[] (transp (Tm _) [][]Ty ((t [ δ ]) [ wk↝ ]))
      = by coh[][] {p ≔ 𝔹[]}
     transp (Tm _) 𝔹[] (transp (Tm _) 𝔹[] (t [ δ ]) [ π₁⇝ id ])
      = by π₂⇝ id
     ⌜ b ⌝𝔹 ∎)
```

We also prove some equations about how these new substitution operations commute. These are very similar to familiar laws pertaining to context extension by types, rather than equations: weakening commutes with lifting over the new equation, and weakening followed by instantiation is identity.

```
wk^⇝   : wk⇝ ; (δ ˆ t ⇝ b) = δ ; wk⇝
wk<>⇝  : Π {t₌ : t = ⌜ b ⌝𝔹} → wk⇝ ; < t₌ »eq = id {Γ ≔ Γ}
```

```
if : Π (t : Tm Γ 𝔹)
  → Tm (Γ ▷ t ⇝ tt) (A [ wk⇝ ]Ty)
  → Tm (Γ ▷ t ⇝ ff) (A [ wk⇝ ]Ty)
  → Tm Γ A
if[] : if t u v [ δ ]
    = if (transp (Tm Δ) 𝔹[] (t [ δ ]))
         (transp (Tm _) wk^Ty⇝ (u [ δ ˆ t ⇝ tt ]))
         (transp (Tm _) wk^Ty⇝ (v [ δ ˆ t ⇝ ff ]))
𝔹β₁ : if tt u v
    =Tm₌ refl (sym (wk<>Ty⇝ {t₌ ≔ refl})) u [ < refl »eq ]
𝔹β₂ : if ff u v
    =Tm₌ refl (sym (wk<>Ty⇝ {t₌ ≔ refl})) v [ < refl »eq ]
```

As with our simply-typed equational contexts, SC^{Bool} contexts can become definitionally inconsistent, and collapse the definitional equality.

Definition 5.1.1 (Definitional context inconsistency)

An SC^{Bool} context is considered def.-inconsistent iff under that context, tt and ff are convertible.

```
incon : Ctx → Type
incon Γ ≔ _=_ {A ≔ Tm Γ 𝔹} tt ff
```

Recall from Remark 3.2.1 that definitionally inconsistent contexts lead to equality collapse: are types become convertible (assuming large elimination of Booleans).

```
collapse : Ctx → Type
collapse Γ ≔ Π (A B : Ty Γ) → A = B
incon-collapse : incon Γ → collapse Γ
```

As an example of how the substitution calculus of SC^{Bool} works, we will prove also that definitional inconsistency implies the collapse of the term equality.

```
tm-collapse : Ctx → Type
tm-collapse Γ ≔ Π A (u v : Tm Γ A) → u = v
tm-incon-collapse : Π Γ → incon Γ → tm-collapse Γ
```

Note that, the u and v inside the “if” must be weakened to account for the new local equation, and contracting the “if” requires explicitly instantiating this equation with a substitution. Our wk<>⇝ lemma from earlier is exactly what we need to show that the composition of these two actions has no ultimate effect.

```
tm-incon-collapse Γ p A u v ≔
  u
   = by sym (subst-subst-sym wk<>⇝ᵀʸ)
  transp (Tm Γ) (sym (wk<>⇝ᵀʸ {t₌ ≔ refl}) ∙ wk<>⇝ᵀʸ {t₌ ≔ refl}) u
   = by cong (transp (Tm Γ) wk<>⇝ᵀʸ) (sym[] (wk<>eqTm {t₌ ≔ refl}))
  transp (Tm Γ) wk<>⇝ᵀʸ (u [ wk⇝ ] [ < refl »eq ])
   = by sym[] (𝔹β₁ {u ≔ u [ wk⇝ ]} {v ≔ v [ wk⇝ ]})
  if tt (u [ wk⇝ ]) (v [ wk⇝ ])
   = by cong (λ □ → if □ (u [ wk⇝ ]) (v [ wk⇝ ])) p
  if ff (u [ wk⇝ ]) (v [ wk⇝ ])
   = by shift 𝔹β₂
  transp (Tm Γ) wk<>⇝ᵀʸ (v [ wk⇝ ] [ < refl »eq ])
   = by cong (transp (Tm Γ) wk<>⇝ᵀʸ) (shift (wk<>eqTm {t₌ ≔ refl}))
transp (Tm Γ) (sym (wk<>⇝ᵀʸ {t₌ ≔ refl}) ∙ wk<>⇝ᵀʸ {t₌ ≔ refl}) v
   = by subst-subst-sym wk<>⇝ᵀʸ
  v ■
```

5.2 Soundness

We prove soundness of $\mathrm{SC}^{\mathrm{Bool}}$ by updating the standard model construction given in Section 2.3.2.

The model gets a little more interesting for $\mathrm{SC}^{\mathrm{Bool}}$ though. Our metatheory does not feature a "first-class" definitional equality, so we instead interpret definitional contextual equalities propositionally (i.e. ⟦ Γ ▷ t >eq b ⟧Ctx == ⟦ Γ ▷ Id 𝔹 t ⌜ b ⌝𝔹 ⟧Ctx).

```
⟦ Γ ▷ t ⇝ b ⟧Ctx ≔ (ρ : ⟦ Γ ⟧Ctx) × ⟦ t ⟧Tm ρ ≡ b
⟦ π₁⇝ δ ⟧Tms ≔ λ ρ → ⟦ δ ⟧Tms ρ .π₁
```

When interpreting _,⇝_, we split on the particular Boolean RHS so the substitution on it computes definitionally (slightly simplifying the equational reasoning, at the cost of having to repeat it).

```
⟦ _,⇝_ {t ≔ t} {b ≔ tt} δ t₌ ⟧Tms
  ≔ λ ρ → ⟦ δ ⟧Tms ρ
          , cong-app
          (⟦ t [ δ ] ⟧Tm
           = by sym (coh⟦⟧Tm {t ≔ t [ δ ]} {A₌ ≔ 𝔹[]} {⟦A⟧₌ ≔ refl})
          ⟦ transp (Tm _) 𝔹[] (t [ δ ]) ⟧Tm
           = by cong ⟦_⟧Tm t₌
          ⟦ tt ⟧Tm ■) ρ
⟦ _,⇝_ {t ≔ t} {b ≔ ff} δ t₌ ⟧Tms
  ≔ λ ρ → ⟦ δ ⟧Tms ρ
          , cong-app
          (⟦ t [ δ ] ⟧Tm
           = by sym (coh⟦⟧Tm {t ≔ t [ δ ]} {A₌ ≔ 𝔹[]} {⟦A⟧₌ ≔ refl})
          ⟦ transp (Tm _) 𝔹[] (t [ δ ]) ⟧Tm
           = by cong ⟦_⟧Tm t₌
          ⟦ ff ⟧Tm ■) ρ
```

To interpret **smart** "if", we define an analagous operation in our metatheory that takes a propositional equality instead: the Boolean "splitter".

```
𝔹-split : (b : 𝔹) → (b = tt → A) → (b = ff → A) → A
𝔹-split tt t f :≡ t refl
𝔹-split ff t f :≡ f refl


⟦ if t u v ⟧Tm :≡ λ ρ → 𝔹-split (⟦ t ⟧Tm ρ)
                                (λ t₌ → ⟦ u ⟧Tm (ρ , t₌))
                                (λ t₌ → ⟦ v ⟧Tm (ρ , t₌))
```

For all finite types A, the "splitter" and eliminator are equally powerful.
To derive the splitter, splitting on a scrutinee x : A, producing a type B, from the eliminator, we instantiate the motive P : A → **Type** with P y :≡ x = y → B. The eliminator's methods then exactly correspond to the splitter's cases, and passing **refl** : x = x to the result of eliminating gets back to type B.
To derive the eliminator from the splitter, we instead instantiate B :≡ P x, and transport the appropriate over the provided propositional equality in each case.
Of course, splitters cannot induct, so the splitter for infinitary types like ℕ is weaker than the associated eliminator.

Finally, to ensure soundness, we also need to show that conversion is preserved. The updated computation rules for "if" still hold definitionally in the meta, but the new $\pi_{2\leadsto}$ law does not. We need to manually project out the propositional equality from the substituted environment, but to do this, we need to get our hands on an environment to substitute (alternatively: evaluate the substitutes in). For this, we need function extensionality (also, we again split on the Boolean to simplify the reasoning):

```
⟦ π₂↝ {t :≡ t} {b :≡ tt} δ ⟧Tm :≡
  ⟦ transp (Tm _) 𝔹[] (t [ π₁↝ δ ]) ⟧Tm
  = by coh⟦⟧Tm {t :≡ t [ π₁↝ δ ]} {A₌ :≡ 𝔹[]} {⟦A⟧₌ :≡ refl}
  ⟦ t [ π₁↝ δ ] ⟧Tm
  = by funext (λ ρ → ⟦ δ ⟧Tms ρ .π₂)
  ⟦ tt ⟧Tm ∎
⟦ π₂↝ {t :≡ t} {b :≡ ff} δ ⟧Tm :≡
  ⟦ transp (Tm _) 𝔹[] (t [ π₁↝ δ ]) ⟧Tm
  = by coh⟦⟧Tm {t :≡ t [ π₁↝ δ ]} {A₌ :≡ 𝔹[]} {⟦A⟧₌ :≡ refl}
  ⟦ t [ π₁↝ δ ] ⟧Tm
  = by funext (λ ρ → ⟦ δ ⟧Tms ρ .π₂)
  ⟦ ff ⟧Tm ∎
```

Soundness itself can be proved as usual: by passing the empty environment to the interpreted proof of Id 𝔹 tt ff.

```
sound : ¬ Tm • (Id 𝔹 tt ff)
sound t :≡ tt/ff-disj (⟦ t ⟧Tm ⟨⟩)
```

5.3 Normalisation Challenges

Normalisation of $\mathrm{SC}^{\mathrm{Bool}}$ is tricky. From our simply-typed investigations in Chapter 4, we already know that completing sets of equations (that is, turning them into confluent, terminating, rewriting systems) is essential to decide equivalence under equational assumptions, and that when such equations can be introduced locally, normalisation and completion must be interleaved.

In $\mathrm{SC}^{\mathrm{Bool}}$, checking whether the equational context can be completed before reducing is even more important, given under definitionally inconsistent contexts, terms that loop w.r.t. β-reduction can be defined (Example 3.2.1). We need to be careful to only ever reduce terms after we have completed at least the set of equations that their typing directly depends on.

Example 5.3.1 ($\mathrm{SC}^{\mathrm{Bool}}$ Reduction W.R.T. a TRS Can Loop Even if the TRS Is Complete)

Consider following the $\mathrm{SC}^{\mathrm{Bool}}$ context (assuming support for neutral equations at

$\mathbb{B}$-type[1]).

$$\begin{aligned}
\Gamma \coloneqq\ & x : \mathbb{B} \\
, \ & y : \mathbb{B} \\
, \ & z : \mathsf{IF}\ x\ \mathbb{B}\ (\mathbb{B} \rightarrow \mathbb{B}) \\
, \ & x \sim y \\
, \ & (\mathsf{if}\ x\ (\lambda\ \_ .\ \mathsf{tt})\ z)\ (\mathsf{if}\ y\ z\ \mathsf{tt}) \rightsquigarrow \mathsf{tt} \\
, \ & x \sim \mathsf{ff} \\
, \ & y \sim \mathsf{tt}
\end{aligned}$$

The TRS $x >_{\mathsf{Rw}} \mathsf{FF}$, $y >_{\mathsf{Rw}} \mathsf{tt}$ is conservative over this equational context, and is complete at least in the sense that all LHSs are irreducible (with respect to the other TRS rewrite and β reduction). However, if (during completion of the full context) we reduce $(\mathsf{if}\ x\ (\lambda\ \_ .\ \mathsf{tt})\ z)\ (\mathsf{if}\ y\ z\ \mathsf{tt})$ w.r.t. this TRS, we get a self-application!

$$\begin{aligned}
& (\mathsf{if}\ x\ (\lambda\ \_ .\ \mathsf{tt})\ z)\ (\mathsf{if}\ y\ z\ \mathsf{tt}) \\
>\ & (\mathsf{if}\ \mathsf{ff}\ (\lambda\ \_ .\ \mathsf{tt})\ z)\ (\mathsf{if}\ \mathsf{tt}\ z\ \mathsf{tt}) \\
>\ & z\ z
\end{aligned}$$

The problem here is that $(\mathsf{if}\ x\ (\lambda\ \_ .\ \mathsf{tt})\ z)\ (\mathsf{if}\ y\ z\ \mathsf{tt})$ relies on the equation $x \sim y$ to typecheck (specifically, so that in the left branch of the second "if" expression, $z : \mathbb{B}$ as required). However, this equation is not validated by the TRS. Essentially, the context is definitionally inconsistent, but we failed to detect this.
If we instead required $x \sim y$ to be in the TRS when reducing $(\mathsf{if}\ x\ (\lambda\ \_ .\ \mathsf{tt})\ z)\ (\mathsf{if}\ y\ z\ \mathsf{tt})$, then completing the TRS with $x \sim \mathsf{ff}$ and $y \sim \mathsf{tt}$ also included will find the definitional inconsistency, so we can avoid blindly reducing.

Technically, self-application does not immediately imply the existence of reduction loops, but we can easily repeat this construction to obtain Ω (Example 3.2.1).

5.3.1 Type Theory Modulo (Boolean) Equations

A nice first-step towards normalisation for $\mathrm{SC}^{\textsc{Bool}}$ would be to attempt to prove decidability of conversion for for dependent types modulo a fixed (global) set of Boolean equations. We can arrive at an explicit syntax for this problem by just replacing $\mathrm{SC}^{\textsc{Bool}}$'s **smart if** with the ordinary dependent one[2].

Note that difficulties associated with completion can in principle be avoided by requiring the set of equations to satisfy completion criteria by construction. In this setting, our problem here is effectively a special case of [94]. Unfortunately, when moving to locally-introduced equations, relying on the LHSs all being mutually irreducible is not really feasible. As we will discuss in Section 5.3.2, any restrictions on equations must be stable under substitution, and irreducibility of LHSs does not satisfy this criteria.

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

A natural strategy here is to make an attempt at adapting our simply-typed result from Section 4.2. Unfortunately, it seems impossible to reuse the same techniques. For starters, non-deterministic reduction on dependent "if" does not preserve typing. Recall that in the definition of "if"

$$\begin{aligned}
\mathsf{if} : \ & \mathbf{\Pi}\ (A : \mathsf{Ty}\ (\Gamma \rhd \mathbb{B}))\ (t : \mathsf{Tm}\ \Gamma\ \mathbb{B}) \\
& \rightarrow \mathsf{Tm}\ \Gamma\ (A\ [\ {<}\ \mathsf{tt}\ {>}\ ]_{\mathsf{Ty}}) \rightarrow \mathsf{Tm}\ \Gamma\ (A\ [\ {<}\ \mathsf{ff}\ {>}\ ]_{\mathsf{Ty}}) \\
& \rightarrow \mathsf{Tm}\ \Gamma\ (A\ [\ {<}\ t\ {>}\ ]_{\mathsf{Ty}})
\end{aligned}$$

the LHS and RHS have type $A\ [\ {<}\ \mathsf{tt}\ {>}\ ]$ and $A\ [\ {<}\ \mathsf{ff}\ {>}\ ]_{\mathsf{Ty}}$ respectively, while the overall expression instead has type $A\ [\ {<}\ t\ {>}\ ]_{\mathsf{Ty}}$ (where t is the scrutinee).

Actually, the problem is more fundamental: we can construct terms in dependent type theory (with just with ordinary, dependent "if") that, after projecting out the untyped term, loop w.r.t. non-deterministic (or spontaneous) reduction. For example (working internally, in a context where $b : \mathbb{B}$ and $x : A$ for some type A), the following term is typeable with $\mathsf{IF}\ b\ A\ (A \rightarrow A) \rightarrow A$.

$$\begin{aligned}
& \lambda\ (y : \mathsf{IF}\ b\ A\ (A \rightarrow A))\ . \\
& \quad (\mathsf{if}\ [b'.\ \mathsf{IF}\ b'\ A\ (A \rightarrow A) \rightarrow A \rightarrow A]\ b\ (\lambda\ \_ .\ x)\ y) \\
& \quad (\mathsf{if}\ [b'.\ \mathsf{IF}\ b'\ A\ (A \rightarrow A) \rightarrow A] \qquad\quad b\ y \qquad\quad x)
\end{aligned}$$

The untyped projection of this term is just

$$\lambda\ y.\ (\mathsf{if}\ b\ (\lambda\ \_ .\ x)\ y)\ (\mathsf{if}\ b\ y\ x)$$

[1] We can avoid this assumption by replacing x and y with slightly more complicated β-neutral terms t and u that are convertible modulo a particular Boolean equation. E.g. $t \coloneqq \mathsf{if}\ x\ \mathsf{tt}\ \mathsf{ff}$ and $u \coloneqq \mathsf{if}\ x\ \mathsf{tt}\ \mathsf{tt}$ are equal assuming $x \rightsquigarrow \mathsf{tt}$.

[2] Note that in such a setting, we can consider a vastly restricted subset of $\mathrm{SC}^{\textsc{Bool}}$'s substitutions, where the region of the context up to the last equational assumption always remains constant, and no new equations can be added.

Under non-deterministic reduction, we can collapse the first “if” the right branch (y), and the second “if” the left branch (also, y) resulting in $\lambda\ \mathsf{y}.\ \mathsf{y}\ \mathsf{y}$ (and under spontaneous reduction, we can do the same thing in two steps, but first collapsing the scrutinee the the appropriate closed Boolean). We then just need to repeat the same construction, replacing A with $\mathsf{IF}\ \mathsf{b}\ \mathsf{A}\ (\mathsf{A} \rightarrow \mathsf{A}) \rightarrow \mathsf{A}$, and we are left with (after erasing types)

$$(\lambda\ \mathsf{y}.\ (\mathsf{if}\ \mathsf{b}\ (\lambda\ \_.\ \mathsf{x})\ \mathsf{y})\ (\mathsf{if}\ \mathsf{b}\ \mathsf{y}\ \mathsf{x}))\ (\lambda\ \mathsf{y}.\ (\mathsf{if}\ \mathsf{b}\ (\lambda\ \_.\ \mathsf{x})\ \mathsf{y})\ (\mathsf{if}\ \mathsf{b}\ \mathsf{y}\ \mathsf{x}))$$

which (spontaneously or non-deterministically) reduces down to Ω

$$(\lambda\ \mathsf{y}.\ \mathsf{y}\ \mathsf{y})\ (\lambda\ \mathsf{y}.\ \mathsf{y}\ \mathsf{y})$$

This puts us essentially back to square-one. We know to normalise SC$^{\textsc{Bool}}$, we need to do completion, but completion can only be justified by making progress w.r.t. to some well-founded order, and our best candidate from STLC does not work. Perhaps one potential route forwards could be to define a TRS for an SC$^{\textsc{Bool}}$ context as a list of Boolean rewrites, plus a small-step relation covering the steps of completion (reducing a LHS, removing a redundant equation, concluding definitional inconsistency from an inconsistent one).

Another difficulty with SC$^{\textsc{Bool}}$ is that we must account for weakening the context by equations when recursing on the syntax (specifically, when recursing into the LHS and RHS branches of **smart if**). Strong normalisation defined as just accessibility w.r.t. the reduction relation is clearly not stable under extending the context with new Boolean equations (the new equations can unlock new reductions)! One route forwards here could be take inspiration from the *positively-characterised* neutral forms of [93, 111]. There, neutral forms being unstable w.r.t. renamings was dealt with by pairing an inductively defined neutral with a function from evidence that the neutral becomes reducible (is *destabilised*) in some renamed context to a normal form. I think applying a similar idea to strong normalisation (parameterising accessibility over all thinned contexts) could assist a strong normalisation proof for SC$^{\textsc{Bool}}$ similarly.

[93]: Gratzer et al. (2022), *Controlling unfolding in type theory*
[111]: Sterling et al. (2021), *Normalization for Cubical Type Theory*

5.3.2 Beyond Booleans

On top of the prior-mentioned issues, SC$^{\textsc{Bool}}$’s scope is somewhat limited. Specifically, generalising the **smart case** construct more general local equality reflection, admitting a larger class of equations, appears impossible.

As covered in Remark 3.2.2, when we start generalising **smart case**, it is useful to view it through the lens of local equality reflection. A significant constraint with introducing equations locally, as in SC$^{\textsc{Bool}}$, is that the restrictions we enforce on reflected equations must be stable under substitution. This is a consequence of being able to introduce equations underneath λ-abstractions and definitional β-reduction: if the equation restriction is not stable under substitution, then β-reduction could take a well-typed term that reflects a valid equation, and reduce it to a term where the reflected equation is no longer valid.

Neutral Types

This has significant consequences. One class of equations we may aim to support are equations between arbitrary terms of neutral type. For example, in a context with variables $\mathsf{b} : \mathbb{B}$ and $\mathsf{x} : \mathsf{IF}\ \mathsf{b}\ \mathsf{A}\ \mathsf{B}$, such that the term $\mathsf{t} : \Pi\ \mathsf{b}.\ \mathsf{IF}\ \mathsf{b}\ \mathsf{A}\ \mathsf{B}$ is typeable (and $\mathsf{t}\ \mathsf{b}$ is neutral), we might want to reflect the equation $\mathsf{t}\ \mathsf{b} \equiv \mathsf{x}$. However, if we then applied the substitution $\mathsf{tt}\ /\ \mathsf{b}$, we get the new equation $\mathsf{t}\ \mathsf{tt} \equiv \mathsf{x}$, where both sides now have type A. This was possible for any type A and term t that blocks on its argument, so for example, we could make this example more concrete by setting $\mathsf{A} \coloneqq \mathbb{N} \rightarrow \mathbb{B}$ and $\mathsf{t} \coloneqq \lambda\ \mathsf{b}.\ \mathsf{if}\ [\mathsf{b}'.\ \mathsf{IF}\ \mathsf{b}'\ (\mathbb{N} \rightarrow \mathbb{B})\ \mathsf{B}]\ \mathsf{b}\ \mathsf{u}\ \mathsf{v}$. Now, $\mathsf{t}\ \mathsf{tt}$ β-reduces to u, an arbitrary

$\mathbb{N} \to \mathbb{B}$-typed term. As mentioned in Section 3.2.1, reflecting equations higher-order-typed equations like this quickly leads to undecidability. Therefore, we must prevent $\mathsf{u} \equiv \mathsf{x}$, and so to ensure stability under substitution, we must also reject the original $\mathsf{t}\ \mathsf{b} \equiv \mathsf{x}$ equation. In practice, I argue this example is repeatable for pretty-much all equations of neutral type[3].

In our minimal type theory, where the only neutral types can be constructed out of large IF, perhaps this does not seem so important. One should consider though, that in theories with type variables, equations of neutral type are extremely common. Consider, for example, code that is generic over a functor, $\mathsf{F} : \textbf{Type} \to \textbf{Type}$. Note the functor laws such as $\mathsf{fmap\text{-}id} : \mathsf{fmap}\ \mathsf{id}\ \mathsf{xs} = \mathsf{xs}$ are all at neutral type. While our focus on manual equality reflection of individual propositional equalities does not aim for quite the same convenience as building the functor equations into the typechecker (i.e. we still need to manually instantiate the particular functor law with our particular $\mathsf{F}\ \mathsf{A}$-typed term), we argue that this kind of automation would still go a long way towards resolving the issues that motivated work such as [112]. Being restricted only to Boolean equations is unnacceptable!

A more skeptical reader might wonder whether our desire here - i.e. reflecting neutral equations - is at all realistic. I reply "yes, because I have found a way to do it!" and "skip ahead to the next chapter to see how!"

[112]: Allais et al. (2013), *New equations for neutral terms: a sound and complete decision procedure, formalized*

Finitary Types

Going a little beyond our Boolean equations appears to be achievable to some extent though. The first obvious equality-reflection-motivated generalisation is to allow $\mathbb{B}$-typed equations where the RHS is not restricted to be a closed Boolean. Assuming an irreducible LHS and RHS, the new equations here are all between neutral terms of $\mathbb{B}$-type, and can be dealt with either directly via completion (using a term ordering on neutrals to orient them appropriately) or (as exchanging neutral terms cannot unblock β-reductions) conversion modulo these equations can be delayed until after β-reduction (and then decided using any approach we like - perhaps ground term rewriting, perhaps E-Graphs).

Note that the subject reduction issues we encountered with equations of neutral type do not crop up here, because while substitutions might unblock the LHS or RHS and allow it to reduce, this reduction can only ever produce another $\mathbb{B}$-typed neutral term or a closed Boolean (though we still would have to repeat completion after substitution).

By extending our language with dependent pairs (Σ-types) with strict η, we also get sum/coproduct-typed equations "for free" via a similar argument to [103]. Specifically, we can define sums/coproducts with Boolean large elimination like so

[103]: Kovács (2022), *Strong eta-rules for functions on sum types*

$$\mathsf{A} + \mathsf{B} \coloneqq \Sigma\,(\mathsf{b} : \mathbb{B})\ .\ \mathsf{IF}\ \mathsf{b}\ \mathsf{A}\ \mathsf{B}$$

$$\mathsf{in}_1\ \mathsf{t} \coloneqq \mathsf{tt}\ ,\ \mathsf{t}$$
$$\mathsf{in}_2\ \mathsf{t} \coloneqq \mathsf{ff}\ ,\ \mathsf{t}$$

Equations of the form $\mathsf{t} \equiv \mathsf{in}_1\ \mathsf{u}$ at type $\mathsf{A} + \mathsf{B}$ can now be decomposed into a Boolean equation $\pi_1\ \mathsf{t} \equiv \textbf{tt}$ and an A-typed equation $\pi_2\ \mathsf{t} \equiv \mathsf{u}$. Of course, this approach only works if the A-typed equation is itself valid.

Example 5.3.2 (Decomposing Coproduct Equations)
I find it is interesting that taking this encoding can deal with rewrites that otherwise appear like they should inevitably loop. For example, consider the equations

$$\mathsf{t} \rightsquigarrow \mathsf{in}_1\ (\mathsf{case}\ \mathsf{t}\ \mathsf{tt}\ \mathsf{ff})$$

where t is some neutral term of type $\mathbb{B} + \mathbb{B}$. Without decomposing using the above encoding, we appear to be stuck. The rewrite must be oriented towards $\mathsf{in}_1\ (\mathsf{case}\ \mathsf{t}\ \mathsf{tt}\ \mathsf{ff})$ or we risk missing β-reductions for case expressions blocking on t, but because t is also a subterm of the RHS, this rewriting process appears to have no end.
If we decompose this using the encoding above, and η-expand the RHS, we get

$$(\pi_1\ \mathsf{t}\ ,\ \pi_2\ \mathsf{t}) \rightsquigarrow (\mathsf{tt}\ ,\ \pi_2\ (\mathsf{case}\ \mathsf{t}\ \mathsf{tt}\ \mathsf{ff}))$$

[3] The main exception that I can think of is equations typed with large $\mathbb{O}$-elim . As $\mathbb{O}$-elim has no computation rules, it cannot possibly reduce down to a higher order type. Technically, I suppose one could also look into the branches of IF to see if all branches return first-order types, and design a restriction based on that.

This can be decomposed into the Boolean equation π_1 t ⇝ tt and the neutral equation at 𝔹-type π_2 t ~ π_2 (case t tt ff). Under a reasonable term ordering (that is, one which is a monotonic simplification ordering), we would probably expect the latter equation to be oriented π_2 (case t tt ff) ⇝ π_2 t, but given both sides are neutral, this reorienting is fine!

Equations between neutrals t ≡ u of type A + B are unfortunately a bit more problematic: the first π_1 t ≡ π_2 u component is fine, assuming validity of neutral Boolean equations, but π_2 t ≡ π_2 u has type if b A B - this is a neutral equation of neutral type, which as explained above, is hard to justify.

Infinitary Types

We can attempt to use a similar generic encoding to deal with infinitary types such as natural numbers, ℕ. By considering the underlying functor, we can decompose inductive types into a fixpoint operation.

```
fixℕ   : 𝟙 + ℕ → ℕ
unfixℕ : ℕ → 𝟙 + ℕ
```

Using this decomposition, and assuming a definitional η rule n ≡ fixℕ (unfixℕ n), the equation n ~ su m is equivalent to

-- *Original*

n ~ su m

-- *η-expanding fix*

fixℕ (unfixℕ n) ~ fixℕ (in$_2$ m)

-- *η-expanding Σ*

fixℕ (π_1 (unfixℕ n) , π_2 (unfixℕ n)) ~ fixℕ (**ff** , m)

which can be decomposed into the two equations

π_1 (unfixℕ n) ⇝ **ff**

π_2 (unfixℕ n) ~ m

However, with infinitary types, we do need to be a bit more careful, as this decomposition process can end up producing infinitely-many equations.

n ~ su n

-- *η-expanding fix and +*

fixℕ (π_1 (unfixℕ n) , π_2 (unfixℕ n)) ~ fixℕ (**ff** , n)

-- *Decomposing*

π_1 (unfixℕ n) ⇝ **ff**

π_2 (unfixℕ n) ~ n

-- *But now if we η-expand n on the RHS...*

π_2 (unfixℕ n) ~ fixℕ (π_1 (unfixℕ n) , π_2 (unfixℕ n))

-- *...the first decomposed rewrite applies!*

π_2 (unfixℕ n) ~ fixℕ (**ff** , π_2 (unfixℕ n))

-- *And we are left with the same structure of equation as we got from initially*

-- *η-expanding*

Intuitively, the problematic cases here all arise when one side of the equation occurs as a subterm of the other. We might hope to do a sort of "occurs check" to explicitly prevent this, but we again hit issues with stability substitution. n ~ su x might pass the occurs check, but after applying the substitution n / x it certainly does not.

5.4 Typechecking Smart Case

We end this section with a short description of the SC$^{\text{Bool}}$ typechecker implemented in Haskell as a component of this project. As explained previously in Section 5.3, I do not know how to prove normalisation of SC$^{\text{Bool}}$, and therefore do not claim that this typechecker is complete. In practice though, it has handled the examples which I have thrown at it correctly, without getting stuck in loops.

The language we check is a slight extension of SC$^{\text{Bool}}$, including a single impredicative universe (**Type** : **Type**). This is technically unsound ([27]), but I argue that programs/proofs which might actually abuse this inconsistency are quite rare in practice (the **Type** : **Type** sledgehammer is also much simpler to implement than an actual universe hierarchy, and concerns with universes are pretty orthogonal to the new features of SC$^{\text{Bool}}$).

Idris2 also features **Type** : **Type**, though there are plans to add a universe hierarchy eventually.

[27]: Hurkens (1995), *A Simplification of Girard's Paradox*

Other than the extensions to specifically support **smart if**, the implementation of SC$^{\text{Bool}}$ is pretty standard. Following [113], it implements bidirectional typechecking in terms of mutually recursive "infer" and "check" functions, and decides convertibility of types using normalisation by evaluation (NbE). To guard against mistakes in the implementation, it also makes extensive use of GADTs (including singleton encodings [86]) to maintain invariants, including that terms are intrinsically well-scoped [87] (after we complete a scope-checking pass, turning names into well-scoped de Bruijn variables) and normal/neutral forms do not contain β-redexes.

[113]: Coquand (1996), *An Algorithm for Type-Checking Dependent Types*

We also use GADTs to explicitly maintain a slightly more unusual invariant: that terms do not contain "obviously ill-typed" β-redexes. That is, introduction rules in scrutinee position are always associated with the appropriate type former.

Assuming a correct implementation, this is completely reasonable (it is a subset of terms being well-typed in general), but alone it is not necessarily preserved over operations such as substitution or reduction. The compromise being struck here is essentially that Haskell's type system is not powerful enough to model full intrinsically-typed syntax, so I am encoding this weaker invariant and then coercing (technically unsafely) when necessary. It is somewhat unclear whether this was a good idea, and for the code snippets in the section, we prune away the details associated with this aspect.

[86]: Lindley et al. (2013), *Hasochism: the pleasure and pain of dependently typed haskell programming*

[87]: Eisenberg (2020), *Stitch: the sound type-indexed type checker (functional pearl)*

When implementing NbE in a partial language, we can take a couple shortcuts[4]:

- Rather than defining values as a type family on object-language types, and defining quoting and unquoting by recursion on types, we define values directly as a non-positive inductive datatype.
- Rather than always quoting before deciding conversion, we can decide conversion directly on values.

The novel part of the typechecker is dealing with the local equations. We explain the implementation of this aspect in more detail, starting with evaluation and then finishing with the actual typechecking.

To track equations, we store a map of rewrites, EqMap g, from neutrals to values, with the invariant that all neutral LHSs are stuck w.r.t. all other rewrites. We pair this map with a list of values associated with every variable in scope to form generalised environments, Env d g

```
type EqMap g ≔ [ (Ne g, Val g) ]
type Env d g ≔ (Vals d g, EqMap d)
```

Unquoting neutral terms during evaluation corresponds exactly to looking up the neutral in the map. In the case the lookup fails (no rewrite applies), we just embed the neutral into Val directly (for simplicity, we do not support η-conversion, though adding support for η of functions by tweaking equality testing of neutral/normal forms should be possible [21])

```
lookupNe :: EqMap g → Ne g → Val g
lookupNe es t ≔ fromMaybe (Ne t) (lookup t es)
appVal :: EqMap g → Val g → Val g → UnkVal g
appVal es (Ne t) u ≔ lookupNe es (App t u)
```

[21]: Lennon-Bertrand (2022), *Á Bas L'η*

To support extending the context with new equations, we must interleave evaluation and completion. For example, to evaluate **smart if**, we add the relevant equation (with addEq) between the scrutinee and tt/ff to the environment in which we evaluate each branch.

[4] The optimisations I decided to make here were generally motivated by simplicity rather than performance. There is certainly a lot of potential to optimise further, e.g. by using a more efficient variable representation than unary de Bruijn indices, using de Bruijn levels in values, switching from metalanguage closures to first-order ones, eliminating the overheads associated with singleton encodings by unsafeCoerce-ing more often, using more efficient data structures, unboxing etc.

```
evalOrAbsrd :: Maybe (Env d g) → Model g → PresVal d
eval        :: Env d g → Tm g → Val d
smrtIfVal   :: Env d g → Maybe (Val d) → Val d
              → Tm g → Tm g
              → Val d

addEq :: Env d g → Val d → Val d
        → Maybe (Env d g)

smrtIfVal r _ tt       u _ ≔ eval r u
smrtIfVal r _ ff       _ v ≔ eval r v
smrtIfVal r m (Ne t) u v
  | rT  ← addEq r (Ne t) tt
  , rF  ← addEq r (Ne t) ff
  , u'  ← evalOrAbsrd rT u
  , v'  ← evalOrAbsrd rF v
  ≔ lookupNe (eqs r) (SmrtIf m t u' v')

eval r (SmrtIf m t u v)
  ≔ smrtIfVal r m' t' u v
  where m' ≔ fmap (eval r) m
        t' ≔ eval r t
```

Note that addEq is partial in order to account for the context possibly becoming definitionally inconsistent (Nothing means "def. inconsistent"). To guard against the danger of evaluating under such contexts, and make the behaviour of the typechecker more predictable, we introduce dedicated syntax for "absurd" terms in def. inconsistent contests ("!" or Absrd). We regard using any term other than ! in a def. inconsistent to be a type error[5]. We always check typeability of terms before evaluating them, so evaluation should never encounter this case.

Dedicated absurd syntax is partially inspired by Agda's impossible patterns.

```
evalOrAbsrd (Just r) t      ≔ eval r t
evalOrAbsrd Nothing Absrd   ≔ Absrd
evalOrAbsrd Nothing _       ≔ __IMPOSSIBLE__
```

Adding equations to the environment calls completion, which itself operates by repeatedly iterating over the set of equations, evaluating LHSs w.r.t. all other equations, until a fixed point is reached (as mentioned in Section 5.3, we need to be careful here to not evaluate under sets of rewrites that might be definitionally inconsistent).

```
complete :: Env g g → Maybe (Env g g)
complete r ≔ iterMaybeFix complStep r

addEq (vs, es) t u ≔ do
  r'  ← addRw (mkEq t u) (idVals, es)
  r'' ← complete r'
  pure (evalVals r'' vs, eqs r'')
```

addRw and mkEq together add the new equation t ∼ u to the set of rewrites, after ensuring it is not already "obviously inconsistent" (that is, literally of the form tt ∼ ff or ff ∼ tt. We also slightly optimise the case of equations on variables, replacing the value in the value environment rather than tracking an equation.
evalVals r" vs re-evaluates the value environment w.r.t. the new completed equations.

Similarly to evaluation, inference for **smart if** adds new equations to the environment before recursively typechecking the branches. We of course must check that terms in def. inconsistent branches are absurd, though unlike evaluation though, failing this check throws a human-readable type error (as opposed to raising an internal exception).

```
checkMaybeAbsurd :: Ctx g → Maybe (Env g g) → Ty g → Tm g → TCM ()
checkMaybeAbsurd g (Just r) a t       ≔ check g r a t
checkMaybeAbsurd _ Nothing _ Absrd    ≔ pure ()
checkMaybeAbsurd _ Nothing _ t        ≔ throw
  ("Body in inconsistent contexts must be absurd, but was instead "
  <> show t)

infer g r (SmrtIf (Just m) t u v) ≔ do
```

[5] We also regard usage of ! in def. consistent contexts to be a type error.

```
    check g r U m
    check g r B t
    let t'  ≔ eval r t
    let rT ≔ addEq r t' tt
    let rF ≔ addEq r t' ff
    checkMaybeAbsurd g rT m u
    checkMaybeAbsurd g rF m v
    let m' ≔ eval r m
    pure m'
```

Note that while type inference for **smart if** does not require a motive parameterised over the scrutinee, it still does require an expected type to check at (Just m above). We support optionally annotating **smart if** expressions with their return type, but to take advantage of the type we are checking at if it is known, check adds annotations before calling “infer”

```
  check g r a t ≔ case t of
    SmrtIf Nothing t' u' v' → do
      _ ← infer g r (SmrtIf (Just a) t' u' v')
      pure ()
```

In retrospect, having “infer” erase annotations and call into “check” for the actual typechecking logic would have probably been a bit neater, but this approach also works.

Elaborating Smart Case 6

In this chapter, we define a new type theory, named SC^{Def}, which introduces equational assumptions at the level of global definitions. We motivate SC^{Def} with the insight that the challenges presented in the prior chapter (Section 5.3) vanish when giving up just a couple conversion rules. Removing these conversion rules outright leaves us with a poorly behaved theory, but it turns out that global definitions, by remaining opaque until the scrutinee they block on reduces, enable us to achieve a similar effect while retaining e.g. congruence of conversion. We prove normalisation (by evaluation), and describe an elaboration algorithm to turn local **smart case**-like splits into top-level definitions.

6.1 A New Core Language

To recap the findings of the previous chapter, locally-introduced equations caused two main issues:

- Any restrictions on equations (enforced in order to retain decidability) must be stable under substitution (to support introducing equations under λ-abstractions without losing subject reduction).
- Any proofs by induction on the syntax must account for weakening the context with new equations. This is problematic for normalisation proofs, because neutral terms are not stable under introducing equations.

The latter of these issues is, in principle, solved if we give up congruence of conversion over **smart if** (or in general, whatever piece of syntax happens to introduce equations). Specifically, if we give up

```
if~  :  Π (t~  :  Tm~ Γ~ 𝔹 t₁ t₂)
     →  Tm~ (Γ~ ▷ t~  ⇝) (A~ [ wkeq~ t~ ]) u₁ u₂
     →  Tm~ (Γ~ ▷ t~  ⇝) (A~ [ wkeq~ t~ ]) v₁ v₂
     →  Tm~ Γ~ A~ (if t₁ u₁ v₁) (if t₂ u₂ v₂)
```

then normalisation no longer needs to recurse into the LHS/RHS branches of "if" expressions until the scrutinee actually reduces to tt or ff.

The first issue can also be fixed by carefully relaxing the substitution law for "if", if[].

```
if[]  :  Tm~ rfl~ rfl~ (if t u v [ δ ])
                       (if (coe~ rfl~ 𝔹[] (t [ δ ]))
                       (coe~ rfl~ wk^eq (u [ δ ^eq t ]))
                       (coe~ rfl~ wk^eq (v [ δ ^eq t ])))
```

Intuitively, we want substitutions to apply recursively to the scrutinee (so we check if it reduces to tt or ff), but stack up on the LHS/RHS (so we do not invalidate the equation in each branch). One way we can achieve this is by outright throwing away if[], and generalising the β-laws $\mathbb{B}\beta_1$ and $\mathbb{B}\beta_2$

```
wk,Ty  :  Ty~ rfl~ (A [ δ ]) (A [ wk⇝ ] [ δ ,⇝  t~ ])
𝔹β₁  :  Π (t~  :  Tm~ rfl~ 𝔹[] (t [ δ ]) tt)
       →  Tm~ rfl~ wk,Ty (if t u v [ δ ]) (u [ δ ,⇝  t~ ])
𝔹β₂  :  Π (t~  :  Tm~ rfl~ 𝔹[] (t [ δ ]) ff)
       →  Tm~ rfl~ wk,Ty (if t u v [ δ ]) (v [ δ ,⇝  t~ ])
```

Using these new laws, the equational theory for "if" somewhat resembles that of a weak-head reduction strategy. That is, normalisation may halt as soon as it hits a stuck "if" expression, instead of recursing into the branches.

This seems like an exciting route forwards. In practice, losing congruence of definitional equality over case splits is not a huge deal, as the proof in question can always just repeat the same case split, proving the desired equation in each branch separately. Unfortunately, from a metatheoretical standpoint, non-congruent conversion is somewhat hard to justify. QIIT and GAT signatures, for example, bake-in congruence of the equational theory (we used an explicit conversion relation, Tm~, above for a reason).

The key insight in solving this comes in the form of *lambda lifting*. For context, Agda's core language only supports pattern-matching at the level of definitions, but it can still support **with**-abstractions [79] and pattern-matching lambdas [114] via elaboration: new top-level definitions are created for every "local" pattern-match. Because definitions are *generative*, from the perspective of the surface language, Agda also loses congruence of conversion (actually, even reflexivity of conversion) for pattern-matching lambdas. For example, consider the equation between these two seemingly-identical implementations of Boolean negation.

[79]: Agda Team (2024), *With-Abstraction*

[114]: Agda Team (2024), *Lambda Abstraction*

```
not-eq : _≡_ {A ≔ 𝔹 → 𝔹}
  (λ where tt → ff
           ff → tt)
  (λ where tt → ff
           ff → tt)
```

Attempting to prove not-eq with reflexivity (**refl**) returns the error:

```
(λ {tt → ff; ff → tt}) x !=
(λ {tt → ff; ff → tt}) x of type 𝔹
Because they are distinct extended lambdas: one is defined at
  /home/nathaniel/agda/fyp/Report/Final/c6-1_scdef.lagda:110.15-111.37
and the other at
  /home/nathaniel/agda/fyp/Report/Final/c6-1_scdef.lagda:112.15-113.37,
so they have different internal representations.
```

In SC$^{\text{Def}}$, we pull essentially the same trick. We can rigorously study a core type theory which introduces equations via top-level definitions (proving soundness and normalisation), and then describe an *elaboration* algorithm to take a surface language with a **smart case**-like construct, and compile it into core SC$^{\text{Def}}$ terms (by lifting **smart** case-splits into top-level definitions).

6.1.1 Syntax

To support global definitions, we introduce an additional sort: *signatures* (Sig). Signatures are similar to contexts in that they effectively store lists of terms that we can reuse, but unlike contexts, signatures also store the concrete implementation of every definition, and do not allow for arbitrary substitution.

```
Sig : Type
Ctx : Sig → Type
```

We associate with Sig a set of morphisms, Wk, forming a category of signature weakenings. Ctx is a presheaf on this category, and substitutions (Tms) are appropriately generalised to map between contexts paired with their signature (we will embed signature weakenings into Tms).

$$\begin{aligned}
&\mathsf{Ty} &&: \mathsf{Ctx}\ \Xi \rightarrow \textbf{Type}\\
&\mathsf{Tm} &&: \mathbf{\Pi}\ (\Gamma : \mathsf{Ctx}\ \Xi) \rightarrow \mathsf{Ty}\ \Gamma \rightarrow \textbf{Type}\\
&\mathsf{Wk} &&: \mathsf{Sig} \rightarrow \mathsf{Sig} \rightarrow \textbf{Type}\\
&\mathsf{Tms} &&: \mathsf{Ctx}\ \Phi \rightarrow \mathsf{Ctx}\ \Psi \rightarrow \textbf{Type}
\end{aligned}$$

We consider all signature weakenings to be equal (i.e. every morphism $\mathsf{Wk}\ \Phi\ \Psi$ is unique; signature weakenings form a *thin category*[1]).

[1] There is a slightly confusing clash of terminology here. The category of thinnings is not a thin category, because e.g. the context $\bullet \rhd \mathbb{B}$ can be thinned to $\bullet \rhd \mathbb{B} \rhd \mathbb{B}$ either by inserting a new $\mathbb{B}$ typed variable at the end or the start of the context. Wk does not have this same freedom: it can only append definitions to the end of the signature.

Remark 6.1.1 (Specialised Substitutions)
We could alternatively build a syntax taking non-generalised (or "specialised") substitutions as primitive (enforcing that the signatures contextualising the domain and range contexts must be the same, $\mathsf{Tms} : \mathsf{Ctx}\ \Xi \rightarrow \mathsf{Ctx}\ \Xi \rightarrow \textbf{Type}$). If we committed to this approach, we would have to add two distinct presheaf actions to Ty and Tm (one for Wk and one for Tms), and also ensure Tms itself is a displayed presheaf over signature weakenings. Our category of generalised substitutions can then be derived by pairing $\phi : \mathsf{Wk}\ \Phi\ \Psi$ and $\delta : \mathsf{Tms}\ \Delta\ \Gamma$ morphisms, with the overall effect of on terms being to take them from context Γ to context $\Delta\ [\ \phi\ ]$.
We will take exactly this approach in the strictified syntax, where it is desirable for signature weakenings embedded in generalised substitutions to compute automatically. For the explicit substitution presentation though, defining generalised substitutions directly leads to a more concise specification.

We give the standard categorical combinators (substitution operations), and context extension (as in Section 2.3), eliding projections and equations for brevity.

$$\begin{aligned}
&\mathsf{id}^{\mathsf{Wk}} &&: \mathsf{Wk}\ \Psi\ \Psi\\
&\_;^{\mathsf{Wk}}\_ &&: \mathsf{Wk}\ \Phi\ \Psi \rightarrow \mathsf{Wk}\ \Xi\ \Phi \rightarrow \mathsf{Wk}\ \Xi\ \Psi
\end{aligned}$$

$$\begin{aligned}
&\mathsf{id} &&: \mathsf{Tms}\ \Gamma\ \Gamma\\
&\_;\_ &&: \mathsf{Tms}\ \Delta\ \Gamma \rightarrow \mathsf{Tms}\ \Theta\ \Delta \rightarrow \mathsf{Tms}\ \Theta\ \Gamma
\end{aligned}$$

$$\begin{aligned}
&\_[\_]^{\mathsf{Wk}}_{\mathsf{Ctx}} &&: \mathsf{Ctx}\ \Psi \rightarrow \mathsf{Wk}\ \Phi\ \Psi \rightarrow \mathsf{Ctx}\ \Phi\\
&\_[\_]_{\mathsf{Ty}} &&: \mathsf{Ty}\ \Gamma \rightarrow \mathsf{Tms}\ \Delta\ \Gamma \rightarrow \mathsf{Ty}\ \Delta\\
&\_[\_] &&: \mathsf{Tm}\ \Gamma\ \mathsf{A} \rightarrow \mathbf{\Pi}\ (\delta : \mathsf{Tms}\ \Delta\ \Gamma) \rightarrow \mathsf{Tm}\ \Delta\ (\mathsf{A}\ [\ \delta\ ]_{\mathsf{Ty}})
\end{aligned}$$

$$\begin{aligned}
&\bullet &&: \mathsf{Ctx}\ \Xi\\
&\_\rhd\_ &&: \mathbf{\Pi}\ (\Gamma : \mathsf{Ctx}\ \Xi) \rightarrow \mathsf{Ty}\ \Gamma \rightarrow \mathsf{Ctx}\ \Xi
\end{aligned}$$

$$\begin{aligned}
&\varepsilon &&: \mathsf{Tms}\ \{\Phi \coloneqq \Xi\}\ \{\Psi \coloneqq \Xi\}\ \Delta\ \bullet\\
&\_,\_ &&: \mathbf{\Pi}\ (\delta : \mathsf{Tms}\ \Delta\ \Gamma) \rightarrow \mathsf{Tm}\ \Delta\ (\mathsf{A}\ [\ \delta\ ]_{\mathsf{Ty}}) \rightarrow \mathsf{Tms}\ \Delta\ (\Gamma \rhd \mathsf{A})
\end{aligned}$$

Signatures are simply lists of definitions. Our first approximation for these definitions is a bundle containing a *telescope* of argument types $\Gamma : \mathsf{Ctx}\ \Xi$ (recall that contexts are just lists of types), a return type $\mathsf{A} : \mathsf{Ty}\ \Gamma$, and a body $\mathsf{Tm}\ \Gamma\ \mathsf{A}$.

$$\begin{aligned}
&\bullet^{\mathsf{Sig}} &&: \mathsf{Sig}\\
&\_\rhd\_\rightarrow\_{:=}\_ &&: \mathbf{\Pi}\ \Xi\ (\Gamma : \mathsf{Ctx}\ \Xi)\ \mathsf{A} \rightarrow \mathsf{Tm}\ \Gamma\ \mathsf{A} \rightarrow \mathsf{Sig}
\end{aligned}$$

Note that $\bullet^{\mathsf{Sig}}$ is the terminal object in the category of signature weakenings. After we define single-weakenings, we can derive the associated morphism $\mathsf{Wk}\ \Phi\ \bullet^{\mathsf{Sig}}$ by composing them.

Intuitively, to call a definition with argument telescope Γ while in a context Δ, we must provide an appropriate list of arguments, specifically a list Δ-terms matching each type in Γ. This is exactly a substitution ($\mathsf{Tms}\ \Delta\ \Gamma$).

Of course, we also want to be able to put equational assumptions in contexts, as in $\mathrm{SC}^{\textsc{Bool}}$.

$$\begin{aligned}
&\_\rhd\_\sim\_ &&: \mathbf{\Pi}\ (\Gamma : \mathsf{Ctx}\ \Xi)\ \{\mathsf{A}\} \rightarrow \mathsf{Tm}\ \Gamma\ \mathsf{A} \rightarrow \mathsf{Tm}\ \Gamma\ \mathsf{A} \rightarrow \mathsf{Ctx}\ \Xi\\
&\_,\!\rightsquigarrow\_ &&: \mathbf{\Pi}\ (\delta : \mathsf{Tms}\ \Delta\ \Gamma) \rightarrow \mathsf{t}_1\ [\ \delta\ ] = \mathsf{t}_2\ [\ \delta\ ]\\
& &&\rightarrow \mathsf{Tms}\ \Delta\ (\Gamma \rhd \mathsf{t}_1 \sim \mathsf{t}_2)
\end{aligned}$$

Rather than shying away from this generalisation, and defining specific argument telescope/argument list types, we commit fully to our extended notions of context and substitution, and take advantage of the flexibility.

Specifically, placing equations in argument telescopes gives us a way to preserve definitional equalities across definition-boundaries. Intuitively, to call a definition that asks for a definitional equality between t_1 and t_2 (its argument telescope contains $t_1 \sim t_2$), the caller must provide evidence that $t_1\ [\ \delta\ ]\ \equiv\ t_2\ [\ \delta\ ]$ (where δ is the list of arguments prior to the equation). In other words, to call a function that asks for a definitional equality, that equation must also hold definitionally at the call-site.

With that said, by only preserving equations (not reflecting new ones) our definitions are still more limited than we need for SC$^{\textsc{Def}}$. Analogously to let-bindings, we could inline the body of every definition and retain a well-typed program (so their only possible application as-currently-defined, like let-bindings, would be to factor out code reuse). We support equality reflection local to each definition by allowing them to each block on one propositional equality.

$$\begin{aligned}
&\mathsf{Id}\ :\ \Pi\ \mathsf{A}\ \rightarrow\ \mathsf{Tm}\ \Gamma\ \mathsf{A}\ \rightarrow\ \mathsf{Tm}\ \Gamma\ \mathsf{A}\ \rightarrow\ \mathsf{Ty}\ \Gamma\\
&\_\triangleright\_\rightarrow\_\mathsf{reflects}\_{:=}\_\ :\ \Pi\ \Xi\ (\Gamma\ :\ \mathsf{Ctx}\ \Xi)\ \mathsf{A}\ \{\mathsf{B}\ t_1\ t_2\}\ \rightarrow\ \mathsf{Tm}\ \Gamma\ (\mathsf{Id}\ \mathsf{B}\ t_1\ t_2)\\
&\quad\rightarrow\ \mathsf{Tm}\ (\Gamma \triangleright t_1 \sim t_2)\ (\mathsf{A}\ [\ \mathsf{wk}_{\rightsquigarrow}\ ]_{\mathsf{Ty}})
\end{aligned}$$

Note that the return type of the definition, A, must still be valid without the equational assumption, and therefore weakened while typing the body. If this were not the case, the result of calling definitions could be ill-typed ($t_1\ [\ \delta\ ]\ \equiv\ t_2\ [\ \delta\ ]$ may not hold at the call-site).

Note that while each individual definition can only reflect one equation at a time, definitions can depend on each other linearly, and preserve previous reflected equations (by asking for them in their argument telescopes), thus nesting multiple equality reflections.

Returning to Booleans

For closer comparison with SC$^{\textsc{Bool}}$, and frankly, to simplify the coming normalisation proof, we return to only supporting Boolean equations.

$$\begin{aligned}
&\_\triangleright\_\rightsquigarrow\_\ &&:\ \Pi\ (\Gamma\ :\ \mathsf{Ctx}\ \Xi)\ \rightarrow\ \mathsf{Tm}\ \Gamma\ \mathbb{B}\ \rightarrow\ \mathbb{B}\ \rightarrow\ \mathsf{Ctx}\ \Xi\\
&\_,\!\rightsquigarrow\_ &&:\ \Pi\ (\delta\ :\ \mathsf{Tms}\ \Delta\ \Gamma)\ \rightarrow\ t\ [\ \delta\ ]\ =^{\mathit{Tm}_{=}\ \boldsymbol{\mathit{refl}}\ \mathbb{B}[]}\ \ulcorner\ b\ \urcorner\mathbb{B}\\
& &&\rightarrow\ \mathsf{Tms}\ \Delta\ (\Gamma \triangleright t\ \rightsquigarrow\ b)
\end{aligned}$$

We could retain the existing $\_\triangleright\_\ \rightarrow\ \_\mathsf{reflects}\_{:=}\_$-style definition by adding the appropriate restriction the RHS term (it needs to be a closed Boolean). Together with the ordinary dependent “if”, we can recover **smart if** by splitting on the scrutinee $t\ :\ \mathsf{Tm}\ \Gamma\ \mathbb{B}$ and calling the appropriate definition with the propositional evidence $\mathsf{Tm}\ \Gamma\ (\mathsf{Id}\ \mathbb{B}\ t\ tt)\}\ /\ \{\mathsf{Tm}\ \Gamma\ (\mathsf{Id}\ \mathbb{B}\ t\ FF)$ in each branch.

For simplicity though, we instead fuse this notion of case-splitting into the signature definitions. Instead of blocking on a propositional equation, definitions now block on a $\mathbb{B}$-typed scrutinee, and reduce to the LHS or RHS when the substituted scrutinee becomes convertible to tt or ff.

$$\begin{aligned}
&\_\triangleright\_\rightarrow\_\mathsf{if}\_{:=}\_|\_\ :\ \Pi\ \Xi\ (\Gamma\ :\ \mathsf{Ctx}\ \Xi)\ \mathsf{A}\ (t\ :\ \mathsf{Tm}\ \Gamma\ \mathbb{B})\\
&\qquad\rightarrow\ \mathsf{Tm}\ (\Gamma \triangleright t\ \rightsquigarrow\ \mathbf{tt})\ (\mathsf{A}\ [\ \mathsf{wk}_{\rightsquigarrow}\ ]_{\mathsf{Ty}})\\
&\qquad\rightarrow\ \mathsf{Tm}\ (\Gamma \triangleright t\ \rightsquigarrow\ \mathbf{ff})\ (\mathsf{A}\ [\ \mathsf{wk}_{\rightsquigarrow}\ ]_{\mathsf{Ty}})\\
&\qquad\rightarrow\ \mathsf{Sig}
\end{aligned}$$

As well as cutting down on the number of term formers, this removes our dependence on having a propositional equality type.

We now define single signature weakenings, and the embedding of signature weakenings into substitutions

$$\begin{aligned}
&\mathsf{wk}^{\mathsf{Wk}} && : \mathsf{Wk}\ (\Psi \rhd \Gamma\ \rightarrow\ \mathsf{A\ if\ t} \coloneqq \mathsf{u} \mid \mathsf{v})\ \Psi\\
&\mathsf{wk}^{\mathsf{Sig}} && : \mathsf{Tms}\ (\Gamma\ [\ \mathsf{wk}^{\mathsf{Wk}}\ \{\mathsf{t}\ \coloneqq\equiv\ \mathsf{t}\}\ \{\mathsf{u}\ \coloneqq\equiv\ \mathsf{u}\}\ \{\mathsf{v}\ \coloneqq\equiv\ \mathsf{v}\}\ ]^{\mathsf{Wk}}_{\mathsf{Ctx}})\ \Gamma
\end{aligned}$$

$$\begin{aligned}
&\ulcorner\_\urcorner^{\mathsf{Wk}} && : \boldsymbol{\Pi}\ (\phi\ :\ \mathsf{Wk}\ \Phi\ \Psi)\ \boldsymbol{\rightarrow}\ \mathsf{Tms}\ (\Gamma\ [\ \phi\ ]^{\mathsf{Wk}}_{\mathsf{Ctx}})\ \Gamma\\
&\ulcorner\ \mathsf{id}\ \ \urcorner^{\mathsf{Wk}} && \coloneqq\equiv\ \mathbf{transp}\ (\boldsymbol{\lambda}\ \square\ \boldsymbol{\rightarrow}\ \mathsf{Tms}\ \square\ \_)\ (\mathsf{sym}\ [\mathsf{id}])\ \mathsf{id}\\
&\ulcorner\ \phi\ ;\ \psi\ \urcorner^{\mathsf{Wk}} && \coloneqq\equiv\ \mathbf{transp}\ (\boldsymbol{\lambda}\ \square\ \boldsymbol{\rightarrow}\ \mathsf{Tms}\ \square\ \_)\ [][]\ (\ulcorner\ \phi\ \urcorner^{\mathsf{Wk}}\ ;\ \ulcorner\ \psi\ \urcorner^{\mathsf{Wk}})\\
&\ulcorner\ \mathsf{wk}^{\mathsf{Sig}}\ \urcorner^{\mathsf{Wk}} && \coloneqq\equiv\ \mathsf{wk}^{\mathsf{Sig}}
\end{aligned}$$

Finally, we give the term former for function calls. Because terms are a presheaf on signature weakenings, we only need to handle the case where the called definition is the last one in the signature (in the strictified syntax, we instead use first-order de Bruijn variables).

$$\mathsf{call}\ :\ \mathsf{Tm}\ \{\Xi\ \coloneqq\equiv\ \Xi \rhd \Gamma\ \rightarrow\ \mathsf{A\ if\ t} \coloneqq \mathsf{u} \mid \mathsf{v}\}\ (\Gamma\ [\ \mathsf{wk}^{\mathsf{Wk}}\ ]^{\mathsf{Wk}}_{\mathsf{Ctx}})\ (\mathsf{A}\ [\ \mathsf{wk}^{\mathsf{Sig}}\ ]_{\mathsf{Ty}})$$

Note that we also do not ask for a list of arguments here. Explicit substitutions handle this for us.

Of course, the β-laws for call must account for the list of arguments, and so target substituted call expressions.

$$\begin{aligned}
\mathsf{call\text{-}tt}\ :\ \ & \boldsymbol{\Pi}\ (\mathsf{t}_=\ :\ \mathsf{t}\ [\ \mathsf{wk}^{\mathsf{Sig}}\ ;\ \delta\ ]\ \boldsymbol{=}^{\mathit{Tm_=}\ \boldsymbol{\mathit{refl}}\ \mathbb{B}\mathit{[]}}\ \ \mathsf{tt})\\
& \boldsymbol{\rightarrow}\ \mathsf{call}\ \{\mathsf{t}\ \coloneqq\equiv\ \mathsf{t}\}\ \{\mathsf{u}\ \coloneqq\equiv\ \mathsf{u}\}\ [\ \delta\ ]\\
& \boldsymbol{=}^{\mathit{Tm_=}\ \boldsymbol{\mathit{refl}}\ (\mathit{sym\ wk}{\rightsquigarrow}^{\mathit{Sig}}\mathit{,Ty})}\\
& \quad \mathsf{u}\ [\ \mathsf{wk}^{\mathsf{Sig}}\ ;\ \mathbf{transp}\ (\mathsf{Tms}\ \_)\ (\mathsf{sym}\ {\rhd}{>}\mathsf{eq}[])\\
& \qquad (\delta\ ,_{\rightsquigarrow}\ (\mathsf{sym}[]\ \{\mathsf{p}\ \coloneqq\equiv\ \mathsf{Tm}_=\ \mathbf{refl}\ (\mathsf{sym}\ \mathbb{B}[])\}\ \mathsf{wk}^{\mathsf{Sig}};\mathsf{Tm}\ \bullet\ \mathsf{t}_=))\ ]\\
\mathsf{call\text{-}ff}\ :\ \ & \boldsymbol{\Pi}\ (\mathsf{t}_=\ :\ \mathsf{t}\ [\ \mathsf{wk}^{\mathsf{Sig}}\ ;\ \delta\ ]\ \boldsymbol{=}^{\mathit{Tm_=}\ \boldsymbol{\mathit{refl}}\ \mathbb{B}\mathit{[]}}\ \ \mathsf{ff})\\
& \boldsymbol{\rightarrow}\ \mathsf{call}\ \{\mathsf{t}\ \coloneqq\equiv\ \mathsf{t}\}\ \{\mathsf{v}\ \coloneqq\equiv\ \mathsf{v}\}\ [\ \delta\ ]\\
& \boldsymbol{=}^{\mathit{Tm_=}\ \boldsymbol{\mathit{refl}}\ (\mathit{sym\ wk}{\rightsquigarrow}^{\mathit{Sig}}\mathit{,Ty})}\\
& \quad \mathsf{v}\ [\ \mathsf{wk}^{\mathsf{Sig}}\ ;\ \mathbf{transp}\ (\mathsf{Tms}\ \_)\ (\mathsf{sym}\ {\rhd}{>}\mathsf{eq}[])\\
& \qquad (\delta\ ,_{\rightsquigarrow}\ (\mathsf{sym}[]\ \{\mathsf{p}\ \coloneqq\equiv\ \mathsf{Tm}_=\ \mathbf{refl}\ (\mathsf{sym}\ \mathbb{B}[])\}\ \mathsf{wk}^{\mathsf{Sig}};\mathsf{Tm}\ \bullet\ \mathsf{t}_=))\ ]
\end{aligned}$$

Dealing with explicit substitutions here gets a little messy, but the key idea is just that if the scrutinee is convertible to tt or ff after substituting in the arguments, then the call should reduce to the appropriate branch. We have made use of the following two commuting lemmas.

$$\begin{aligned}
\mathsf{wk}^{\mathsf{Sig}};\mathsf{Tm}\ :\ & \mathsf{t}\ [\ \mathsf{wk}^{\mathsf{Sig}}\ ;\ \delta\ ]\\
& \boldsymbol{=}^{\mathit{Tm_=}\ \boldsymbol{\mathit{refl}}\ (\mathbb{B}\mathit{[]}\ \bullet\ \mathit{sym}\ \mathbb{B}\mathit{[]})}\\
& \ \mathbf{transp}\ (\mathsf{Tm}\ (\Gamma\ [\ \mathsf{wk}^{\mathsf{Wk}}\ ]^{\mathsf{Wk}}_{\mathsf{Ctx}}))\ \mathbb{B}[]\ (\mathsf{t}\ [\ \ulcorner\ \mathsf{wk}^{\mathsf{Wk}}\ \urcorner^{\mathsf{Wk}}\ ])\ [\ \delta\ ]\\
\mathsf{wk}{\rightsquigarrow}^{\mathsf{Sig}},\mathsf{Ty}\ :\ & \boldsymbol{\Pi}\ \{\mathsf{t}_=\ :\ \mathsf{t}\ [\ \mathsf{wk}^{\mathsf{Sig}}\ ;\ \delta\ ]\ \boldsymbol{=}^{\mathit{Tm_=}\ \boldsymbol{\mathit{refl}}\ \mathbb{B}\mathit{[]}}\ \ \mathsf{tt}\}\\
& \boldsymbol{\rightarrow}\ \mathsf{A}\ [\ \mathsf{wk}_{\rightsquigarrow}\ ]_{\mathsf{Ty}}\\
& \qquad [\ \mathsf{wk}^{\mathsf{Sig}}\ ;\ \mathbf{transp}\ (\mathsf{Tms}\ \_)\ (\mathsf{sym}\ {\rhd}{>}\mathsf{eq}[])\\
& \qquad\ \ (\delta\ ,_{\rightsquigarrow}\ (\mathsf{sym}[]\ \{\mathsf{p}\ \coloneqq\equiv\ \mathsf{Tm}_=\ \mathbf{refl}\ (\mathsf{sym}\ \mathbb{B}[])\}\ \mathsf{wk}^{\mathsf{Sig}};\mathsf{Tm}\ \bullet\ \mathsf{t}_=))\ ]_{\mathsf{Ty}}\\
& \boldsymbol{=}\ \mathsf{A}\ \ [\ \mathsf{wk}^{\mathsf{Sig}}\ ]_{\mathsf{Ty}}\ [\ \delta\ ]_{\mathsf{Ty}}
\end{aligned}$$

We do not need any other substitution laws for call. The composition functor law is already enough for additional substitutions to recursively apply to the argument list (by composing the substitutions).

$$\mathsf{call}\ [\ \delta\ ]\ [\ \sigma\ ]\ \boldsymbol{=}\ \mathsf{call}\ [\ \delta\ ;\ \sigma\ ]$$

6.1.2 Soundness

We prove soundness of SC$^{\textsc{Def}}$ by constructing a model. Our model contains two notions of environments: one relating to signatures (we denote signature environments with "χ") and one to local contexts (we denote context environments with "ρ" as usual). Signature weakenings can be interpreted as functions between signature environments, while generalised substitutions become pairs of signature environment and context environment mappings.

$$\begin{aligned}
&\llbracket \mathsf{Sig} \rrbracket \ : \ \mathbf{Type}_1 \\
&\llbracket \mathsf{Sig} \rrbracket \ :\equiv \ \mathbf{Type}
\end{aligned}$$

$$\begin{aligned}
&\llbracket \mathsf{Ctx} \rrbracket \ : \ \llbracket \mathsf{Sig} \rrbracket \ \rightarrow \ \mathbf{Type}_1 \\
&\llbracket \mathsf{Ctx} \rrbracket \ \llbracket \Psi \rrbracket \ :\equiv \ \llbracket \Psi \rrbracket \ \rightarrow \ \mathbf{Type} \\
&\llbracket \mathsf{Ty} \rrbracket \ : \ \llbracket \mathsf{Ctx} \rrbracket \ \llbracket \Psi \rrbracket \ \rightarrow \ \mathbf{Type}_1 \\
&\llbracket \mathsf{Ty} \rrbracket \ \llbracket \Gamma \rrbracket \ :\equiv \ \mathbf{\Pi}\ \chi \ \rightarrow \ \llbracket \Gamma \rrbracket \ \chi \ \rightarrow \ \mathbf{Type} \\
&\llbracket \mathsf{Tm} \rrbracket \ : \ \mathbf{\Pi}\ (\llbracket \Gamma \rrbracket \ : \ \llbracket \mathsf{Ctx} \rrbracket \ \llbracket \Psi \rrbracket) \ \rightarrow \ \llbracket \mathsf{Ty} \rrbracket \ \llbracket \Gamma \rrbracket \ \rightarrow \ \mathbf{Type} \\
&\llbracket \mathsf{Tm} \rrbracket \ \llbracket \Gamma \rrbracket \ \llbracket \mathsf{A} \rrbracket \ :\equiv \ \mathbf{\Pi}\ \chi\ \rho \ \rightarrow \ \llbracket \mathsf{A} \rrbracket \ \chi\ \rho \\
&\llbracket \mathsf{Wk} \rrbracket \ : \ \llbracket \mathsf{Sig} \rrbracket \ \rightarrow \ \llbracket \mathsf{Sig} \rrbracket \ \rightarrow \ \mathbf{Type} \\
&\llbracket \mathsf{Wk} \rrbracket \ \llbracket \Phi \rrbracket \ \llbracket \Psi \rrbracket \ :\equiv \ \llbracket \Phi \rrbracket \ \rightarrow \ \llbracket \Psi \rrbracket \\
&\llbracket [\,]_{\mathsf{Ctx}} \rrbracket \ : \ \llbracket \mathsf{Ctx} \rrbracket \ \llbracket \Psi \rrbracket \ \rightarrow \ \llbracket \mathsf{Wk} \rrbracket \ \llbracket \Phi \rrbracket \ \llbracket \Psi \rrbracket \ \rightarrow \ \llbracket \mathsf{Ctx} \rrbracket \ \llbracket \Phi \rrbracket \\
&\llbracket [\,]_{\mathsf{Ctx}} \rrbracket \ \llbracket \Gamma \rrbracket \ \llbracket \delta \rrbracket \ :\equiv \ \boldsymbol{\lambda}\ \chi \ \rightarrow \ \llbracket \Gamma \rrbracket \ (\llbracket \delta \rrbracket \ \chi) \\
&\llbracket \mathsf{Tms} \rrbracket \ : \ \llbracket \mathsf{Ctx} \rrbracket \ \llbracket \Phi \rrbracket \ \rightarrow \ \llbracket \mathsf{Ctx} \rrbracket \ \llbracket \Psi \rrbracket \ \rightarrow \ \mathbf{Type} \\
&\llbracket \mathsf{Tms} \rrbracket \ \{\llbracket \Phi \rrbracket \ :\equiv \ \llbracket \Phi \rrbracket\} \ \{\llbracket \Psi \rrbracket \ :\equiv \ \llbracket \Psi \rrbracket\} \ \llbracket \Delta \rrbracket \ \llbracket \Gamma \rrbracket \\
&\quad :\equiv \ (\llbracket \delta \rrbracket : \llbracket \mathsf{Wk} \rrbracket \ \llbracket \Phi \rrbracket \ \llbracket \Psi \rrbracket) \ \times \ (\mathbf{\Pi}\ \{\chi\} \ \rightarrow \ \llbracket \Delta \rrbracket \ \chi \ \rightarrow \ \llbracket [\,]_{\mathsf{Ctx}} \rrbracket \ \llbracket \Gamma \rrbracket \ \llbracket \delta \rrbracket \ \chi) \\
&\llbracket\_\rrbracket\mathsf{Sig} \quad : \ \mathsf{Sig} \ \rightarrow \ \llbracket \mathsf{Sig} \rrbracket \\
&\llbracket\_\rrbracket\mathsf{Ctx} \quad : \ \mathsf{Ctx}\ \Psi \ \rightarrow \ \llbracket \mathsf{Ctx} \rrbracket \ \llbracket\, \Psi \,\rrbracket\mathsf{Sig} \\
&\llbracket\_\rrbracket\mathsf{Ty} \quad : \ \mathsf{Ty}\ \Gamma \ \rightarrow \ \llbracket \mathsf{Ty} \rrbracket \ \llbracket\, \Gamma \,\rrbracket\mathsf{Ctx} \\
&\llbracket\_\rrbracket\mathsf{Tm} \quad : \ \mathsf{Tm}\ \Gamma\ \mathsf{A} \ \rightarrow \ \llbracket \mathsf{Tm} \rrbracket \ \llbracket\, \Gamma \,\rrbracket\mathsf{Ctx} \ \llbracket\, \mathsf{A} \,\rrbracket\mathsf{Ty} \\
&\llbracket\_\rrbracket\mathsf{Wk} \quad : \ \mathsf{Wk}\ \Phi\ \Psi \ \rightarrow \ \llbracket \mathsf{Wk} \rrbracket \ \llbracket\, \Phi \,\rrbracket\mathsf{Sig} \ \llbracket\, \Psi \,\rrbracket\mathsf{Sig} \\
&\llbracket\_\rrbracket\mathsf{Tms} \ : \ \mathsf{Tms}\ \Delta\ \Gamma \ \rightarrow \ \llbracket \mathsf{Tms} \rrbracket \ \llbracket\, \Delta \,\rrbracket\mathsf{Ctx} \ \llbracket\, \Gamma \,\rrbracket\mathsf{Ctx}
\end{aligned}$$

The interpretations of ordinary constructs from dependently-typed lambda calculus are mostly unchanged in this new model, except for having to account for both environments. E.g. Π-types are now interpreted as

$$\llbracket\, \Pi\ \mathsf{A}\ \mathsf{B} \,\rrbracket\mathsf{Ty} \ :\equiv \ \boldsymbol{\lambda}\ \chi\ \rho \ \rightarrow \ \mathbf{\Pi}\ \mathsf{t}^{\mathsf{V}} \ \rightarrow \ \llbracket\, \mathsf{B} \,\rrbracket\mathsf{Ty}\ \chi\ (\rho\ ,\ \mathsf{t}^{\mathsf{V}})$$

We therefore focus on the new cases. Local equations are interpreted as propositional equations, as in SC$^{\textsc{Bool}}$ (Section 5.2) and the new presheaf action on contexts is just function composition.

$$\begin{aligned}
&\llbracket\, \Gamma \rhd \mathsf{t} \ \rightsquigarrow \ \mathsf{b} \,\rrbracket\mathsf{Ctx} \ :\equiv \ \boldsymbol{\lambda}\ \chi \ \rightarrow \ (\rho : \llbracket\, \Gamma \,\rrbracket\mathsf{Ctx}\ \chi) \ \times \ \llbracket\, \mathsf{t} \,\rrbracket\mathsf{Tm}\ \chi\ \rho \ = \ \mathsf{b} \\
&\llbracket\, \Gamma\ [\ \delta\ ]_{\mathsf{Ctx}} \quad \rrbracket\mathsf{Ctx} \ :\equiv \ \boldsymbol{\lambda}\ \chi \ \rightarrow \ \llbracket\, \Gamma \,\rrbracket\mathsf{Ctx}\ (\llbracket\, \delta \,\rrbracket\mathsf{Wk}\ \chi)
\end{aligned}$$

As previously mentioned, we interpret signatures as environments. Our Boolean-splitting definitions are interpreted with a single body, plus equations it evaluates evaluate to the appropriate branch depending on which closed Boolean the scrutinee reduces to.

$$\begin{aligned}
&\llbracket\, \bullet^{\mathsf{Sig}} \qquad\qquad\qquad \rrbracket\mathsf{Sig} \ :\equiv \ \mathbb{1} \\
&\llbracket\, \Xi \rhd \Gamma \ \longrightarrow \ \mathsf{A}\ \mathsf{if}\ \mathsf{t} := \mathsf{u} \mid \mathsf{v} \,\rrbracket\mathsf{Sig} \\
&\quad :\equiv \ (\chi : \llbracket\, \Xi \,\rrbracket\mathsf{Sig}) \ \times \\
&\qquad (\mathsf{f} : (\mathbf{\Pi}\ \rho \ \rightarrow \ \llbracket\, \mathsf{A} \,\rrbracket\mathsf{Ty}\ \chi\ \rho)) \ \times \\
&\qquad\ (\mathbf{\Pi}\ \rho\ (\mathsf{t}_{=} \ : \ \llbracket\, \mathsf{t} \,\rrbracket\mathsf{Tm}\ \chi\ \rho \ = \ \mathbf{tt}) \\
&\qquad\quad \rightarrow \ \mathsf{f}\ \rho \ = \ \llbracket\, \mathsf{u} \,\rrbracket\mathsf{Tm}\ \chi\ (\rho\ ,\ \mathsf{t}_{=})) \ \times
\end{aligned}$$

```
      (Π ρ (t₌ : ⟦ t ⟧Tm χ ρ ≡ ff)
      → f ρ ≡ ⟦ v ⟧Tm χ (ρ , t₌))
```

Single signature weakenings are interpreted as projections:

```
  ⟦ wkᵂᵏ ⟧Wk  ≔ π₁
  ⟦ wkˢⁱᵍ ⟧Tms ≔ π₁ , λ ρ → ρ
```

and calls to definitions merely project out the body

```
  ⟦ call ⟧Tm (χ , f , f-tt , f-ff) ρ
    ≔ f ρ
```

The only non-trivial equations arise from $\pi_{2\rightsquigarrow}$ and callTT/callFF. We can account for the former of these using the equation in the environment and function extensionality, as in SC$^{\text{Bool}}$. The computation laws for call also require function extensionality; depending on whether the scrutinee reduces to tt or ff, we apply the relevant equation in the signature environment.

6.2 Normalisation

In the below section, we switch to use a strictified SC$^{\text{Def}}$ syntax. Compared to the presentation with explicit substitutions, the main differences (beyond substitution equations holding definitionally) are as follows:

- Tms Δ Γ now refers only to specialised substitutions (Remark 6.1.1).
- We have dedicated types for representing indexing into signatures (DefVar Ξ Γ A) and picking out equations from the context (EqVar Γ t b).

```
data EqVar : Π (Γ : Ctx Ξ) {A} → Tm Γ A → 𝔹 → Type where
  ez   : EqVar (Γ ▷ t ⇝ b) (t [ wk_⇝ ]) b
  es   : EqVar Γ t b → EqVar (Γ ▷ A) (t [ wk ]) b
  eseq : EqVar Γ t b_1 → EqVar (Γ ▷ u ⇝ b_2) (t [ wk_⇝ ]) b_1
data DefVar where
  fz : DefVar (Ξ ▷ Γ → A if t := u | v) (Γ [ wk^Wk ]^Wk_Ctx) (A [ wk^Wk ]^Wk_Ty)
  fs : DefVar Ξ Γ A → DefVar (Ξ ▷ Δ → B if t := u | v)
                             (Γ [ wk^Wk ]^Wk_Ctx) (A [ wk^Wk ]^Wk_Ty)
```

These datatypes also need coe constructors, corresponding to their role as setoid fibrations.

- DefVars have an associated lookup$^{\text{Sig}}$ operation.

```
record Def Ξ (Γ : Ctx Ξ) (A : Ty Γ) : Type where
  constructor if
  field
    scrut : Tm Γ 𝔹
    lhs   : Tm (Γ ▷ scrut ⇝ tt) (A [ wk_⇝ ]_Ty)
    rhs   : Tm (Γ ▷ scrut ⇝ ff) (A [ wk_⇝ ]_Ty)
lookup^Sig : Π Ξ {Γ A} → DefVar Ξ Γ A → Def Ξ Γ A
```

- "call"s are now explicitly bundled with their list of arguments.

```
call : Π (f : DefVar Ξ Γ A) (δ : Tms Δ Γ)
  → Tm Δ (A [ δ ]_Ty)
```

6.2.1 Conversion and Coherence

When presenting NbE for dependent types in Section 2.4.3, we were able to preserve the conversion relation at every step. This justified us playing quite "fast and loose" with details relating to coercion/coherence: using setoids was ultimately just an implementation detail and we could have achieved the same result using a quotiented syntax instead [60].

[60]: Altenkirch et al. (2017), *Normalisation by Evaluation for Type Theory, in Type Theory*

In SC$^{\text{Def}}$, the situation gets a bit trickier. I do not know how to deal with contextual equations other than via term rewriting, but rewriting is an inherently very syntactic procedure.

Luckily, setoids give us a framework for working with multiple distinct equivalence relations. Indexing of the syntax itself must still be up to conversion in order to account for definitional equality, but this does not stop us from writing functions that e.g. project out raw untyped terms. I will sometimes refer to equality *up-to-coherence* merely referring to the smallest congruence relation including coh. Applied to the syntax of type theory, this aligns exactly with syntactic equality of untyped projections.

For simplicity of the presentation in the report, we still try to avoid getting too bogged-down in encoding details associated with these different equivalence relations, but it is important to keep in mind that some portions of the below algorithm (especially those parts which directly refer to term rewriting concepts) do not respect conversion alone.

6.2.2 Normal and Neutral Forms

We define SC$^{\text{Def}}$ normal forms as usual, assuming some appropriate definition of neutrals. Like in Section 2.4.3, normal forms form a setoid fibration on conversion, so the term we index by only needs to be convertible to the normal form.

```
data Nf : Π Γ A → Tm {Ξ ≔ Ξ} Γ A → Type
Ne      : Π Γ A → Tm {Ξ ≔ Ξ} Γ A → Type

data Nf where
  coe~ : Π Γ~ A~ → Tm~ Γ~ A~ t_1 t_2 → Nf Γ_1 A_1 t_1 → Nf Γ_2 A_2 t_2
  ne𝔹  : Ne Γ 𝔹 t → Nf Γ 𝔹 t
  neIF : Ne Γ 𝔹 u → Ne Γ (IF u A B) t → Nf Γ (IF u A B) t
  λ_   : Nf (Γ ▷ A) B t → Nf Γ (Π A B) (λ t)
  tt   : Nf Γ 𝔹 tt
  ff   : Nf Γ 𝔹 ff
```

SC$^{\text{Def}}$ neutrals are a little more tricky. Boolean equations mean we can no longer define these purely inductively, as modulo contextual equations, any 𝔹-typed term can in principle be convertible to tt or ff (which are of course non-neutral - tt and ff do not block β-reduction). We start, therefore, by defining *pre-neutrals* as β-neutral terms where all subterms are fully normal/neutral.

```
data PreNe : Π Γ A → Tm {Ξ ≔ Ξ} Γ A → Type where
  coe~ : Π Γ~ A~ → Tm~ Γ~ A~ t_1 t_2 → PreNe Γ_1 A_1 t_1 → PreNe Γ_2 A_2 t_2
  `_   : Π i → PreNe Γ A (` i)
  _·_  : Ne Γ (Π A B) t → Nf Γ A u
       → PreNe Γ (B [ < u > ]Ty) (t · u)
  callNe : Ne Δ 𝔹 (lookup^Sig Ψ f .scrut [ δ ])
         → PreNe Δ (A [ δ ]Ty) (call {A ≔ A} f δ)
```

We then define the “true” neutrals by pairing the pre-neutral term with explicit evidence that it is not convertible to a closed Boolean.

```
predNe : Π Γ A → Tm {Ξ ≔ Ξ} Γ A → Type
predNe Γ A t ≔ Π {Γ′} b Γ~ A~ → ¬ Tm~ {Γ_2 ≔ Γ′} Γ~ A~ t ⌜ b ⌝𝔹
Ne Γ A t ≔ PreNe Γ A t × predNe Γ A t
```

As conversion (Tm~ Γ~ A~ t_1 t_2) lies in **Prop**, we normally would need to “box” the proof here. To hide these encoding details, we rely on cumulativity, which includes **Prop** <: **Type** subtyping.

Remark 6.2.1 (Stability Under Thinnings vs Renamings)
These neutral forms are not stable under arbitrary renamings. For example, in the context Γ ≔ x : 𝔹 , y : 𝔹 , x ⇝ **tt**, the variable y is neutral. However, if we apply the renaming y / x, the context becomes Γ [y / x] ≡ y : 𝔹 , y ⇝ **tt**, and y is now convertible to a closed Boolean. We therefore make sure to take presheaves over the category of thinnings (which does not encounter this problem) when proving normalisation.

Remark 6.2.2 (Beyond Booleans)
This definition relies heavily on the fact that all of our equations are of the form t ~ ⌜ b ⌝𝔹. If equations e.g. between neutral terms were to be allowed, then these normal forms would no longer be unique (up to coherence).
As in Section 5.3.2 - Finitary Types I think there are at least two possible solutions to here:

- We could keep the same definition of neutrals as above, and give up on uniqueness of normal forms. Instead, equivalence of neutrals can be defined modulo a set of neutral equations. Note that rewriting neutral subterms to other neutrals cannot unblock β-reductions (the whole motivation for neutral terms is that they block reduction), so NbE still makes progress (it fully decides the β-equality). To actually decide equality of normal forms, we then can use standard term rewriting approaches such as ground completion or E-Graphs (the equational theory on β-normal forms is, up to coherence, a ground TRS).

- Alternatively, we could attempt to fully normalise terms during NbE, by integrating ground completion directly. Specifically, we can define a term ordering on β-normal/neutral terms such that tt and ff are minimal, and then generalise predNe to the non-existence of normal forms (of the same term) smaller than the given neutral.
  To avoid getting bogged down in accounting for conversion/coherence, we concretely define the term ordering on untyped terms.

```
UTm  : Type
βNe  : Π Γ A  →  Tm {Ξ  ≔  Ξ} Γ A  →  Type
βNf  : Π Γ A  →  Tm {Ξ  ≔  Ξ} Γ A  →  Type

projβNe  :  βNe Γ A t  →  UTm
projβNf  :  βNf Γ A t  →  UTm

_>UTm_  :  UTm  →  UTm  →  Type

predNe  : Π Γ A t  →  βNe {Ξ  ≔  Ξ} Γ A t  →  Type
predNe Γ A t tᴺᵉ
    ≔  Π (tᴺᶠ  :  βNf Γ A t)  →  ¬ projβNe tᴺᵉ >UTm projβNf tᴺᶠ
```

We will stick with t ~ ⌜ b ⌝𝔹 equations for simplicity. In either of the above approaches, I suspect the extra difficulties will primarily relate to needing to be careful with exactly which types/relations are setoid fibrations on either coherence or conversion.

When I refer to β-equality/β-normality here, I am also implicitly including η for Π types. Actually, accounting for η equality in the second approach is a little subtle: we rely on the fact that the result of η-expanding any neutral is never considered *smaller* than the original. I argue this is a pretty reasonable expectation (e.g. it follows from monotonicity), but alternatively, we could just require that t^{Ne} not be larger than any alternative β-neutral ($\mathsf{t}^{\mathsf{Ne}\prime}$: βNe Γ A t) and combine this with the statement that t is also not convertible to a closed Boolean given prior.

Note that all the definitions of normal/neutrals forms presented here are assuming definitionally consistent contexts. In definitionally inconsistent contexts, we can collapse all terms to 𝟙 as in Section 4.2.

6.2.3 Sound and Complete TRSs

Justifying *completion* with a well-founded order (also taking reduction into account) is hard[2]. Luckily, because stability under substitution is no longer a requirement, we have a lot more freedom in how to restrict equations such that completion is not necessary. For example, we could require that all Boolean equation LHSs are mutually irreducible (and check this syntactically), ensuring that our equation set is completed by definition.

We delay the actual details of this syntactic check and recovering the required semantic properties for Section 6.3.1. For now, we specify the semantic requirement on completed contexts only: either the context should be definitionally inconsistent, or there must be a completed TRS, equivalent to the equational context.

Raw TRSs are just lists of paired pre-neutral LHSs and Boolean RHSs.

```
data TRS (Γ  :  Ctx Ψ)  :  Type where
  •           :  TRS Γ
  _▷_>Rw_  :  TRS Γ  →  PreNe Γ 𝔹 t  →  𝔹  →  TRS Γ
```

We then define TRSs to be valid (for a particular context) if rewrites imply convertibility and vice versa on pre-neutral terms. This is similar in spirit to the observational equivalence property of equational contexts in Section 4.2, but instead of between contexts, we define the equivalence between the SC$^{\text{Def}}$ context (which induces a declarative notion of conversion) and a concrete set of rewrites (where the induced notion of conversion is operational).

[2] Recall from Section 5.3 that our trick involving *spontaneous reduction* Section 4.2 does not extend to dependent types).

```
data RwVar : TRS Γ → PreNe Γ 𝔹 t → 𝔹 → Type where
  rz : RwVar (Γ^{TRS} ▷ t^{PreNe} >_{Rw} b) t^{PreNe} b
  rs : RwVar Γ^{TRS} t^{PreNe} b_1 → RwVar (Γ^{TRS} ▷ u^{PreNe} >_{Rw} b_2) t^{PreNe} b_1
record ValidTRS (Γ : Ctx Ξ) : Type where field
  trs  : TRS Γ
  to   : Tm~ rfl~ rfl~ t ⌜ b ⌝𝔹 → Π (t^{PreNe} : PreNe Γ 𝔹 t)
       → RwVar trs t^{PreNe} b
  from : RwVar {t ≔ t} trs t^{PreNe} b → Tm~ rfl~ rfl~ t ⌜ b ⌝𝔹
def-incon : Ctx Ξ → Prop
def-incon Γ ≔ Tm~ (rfl~ {Γ ≔ Γ}) rfl~ tt ff
data TRS? (Γ : Ctx Ξ) : Type where
  compl : ValidTRS Γ → TRS? Γ
  !!    : def-incon Γ → TRS? Γ
```

Technically, RwVars here should be defined up to coherence-equivalence. To account for this, we must to index by pre-neutrals of arbitrary type, A (rather than 𝔹) and then generalise "from" and "to" appropriately. In "from" specifically, we need to specify the coherence equation CohTy~ _ A 𝔹 to satisfy the indexing of Tm~ (either at the declaration, or when applying it). We can either index RwVar directly by the coherence equation or project out the proof by recursion.

Remark 6.2.3 (Alternative Definition of TRS "to")
Note that the "to" condition above is equivalent to

```
to′ : Π (Γ^{TRS} : TRS Γ) → EqVar Γ t b
    → Π (t^{PreNe} : PreNe Γ 𝔹 t) → RwVar Γ^{TRS} t^{PreNe} b
```

given the following lemma, which should be provable by introducing reduction and algorithmic conversion, showing the equivalence with declarative conversion (via confluence of reduction) and then taking advantage of how the only possible reduction which can apply to a pre-neutral term is a rewrite targetting the whole thing (recall that all subterms of pre-neutrals are fully neutral/normal).

```
inv-lemma : PreNe Γ A t → Tm~ Γ~ A~ t ⌜ b ⌝𝔹 → EqVar Γ (coe~ Γ~ A~ t) b
```

We rely on this lemma in Section 6.3.1, however, this is a lot of work for a small and quite technical result, so we will not prove this in detail. Finding an easier way to prove this (or avoid relying on it entirely) could be interesting future work.

6.2.4 Normalisation by Evaluation

We now extend normalisation by evaluation for dependent types (as initially presented in Section 2.4.3 to SC$^{\text{Def}}$.

As before, the core of the normalisation argument will hinge on neutral/normal forms being presheaves on a category of thinnings[3]. To account for local equational assumptions in contexts, we extend thinnings with lifting over contexts extended by equations (i.e. so it is still possible to construct identity thinnings) but critically do not include equation-weakenings (Thin (Δ ▷ t ⤳ b) Γ (δ ; wk$_{\rightsquigarrow}$)), which destabilise neutral terms (and destroy completion evidence).

```
data Thin {Ξ} : Π Δ Γ → Tms {Ξ ≔ Ξ} Δ Γ → Type where
  ε           : Thin • • ε
  _^{^Th}_     : Thin Δ Γ δ → Π A
              → Thin (Δ ▷ (A [ δ ]_{Ty})) (Γ ▷ A) (δ ^ A)
  _^{^Th}_ ⤳ _ : Thin Δ Γ δ → Π t b
              → Thin (Δ ▷ t [ δ ] ⤳ b) (Γ ▷ t ⤳ b) (δ ^ t ⤳ b)
  _^{+Th}_     : Thin Δ Γ δ
              → Π A → Thin (Δ ▷ A) Γ (δ ; wk)
```

When defining environments and values, we require a valid TRS associated with the target context (recall that normalisation in definitionally inconsistent contexts is trivial, so we focus only on the definitionally consistent case here). Throughout the normalisation

[3] We will also require stability of completion evidence w.r.t. thinning, which follows from applying the thinning pointwise to the underlying TRS, and then taking advantage of how thinnings can be inverted.

algorithm, we will never add new equations to the target context, so we can preserve the ValidTRS the whole way through.

$$
\begin{aligned}
&\mathsf{Env} && : \boldsymbol{\Pi}\ \Xi\ \Delta\ \Gamma\ \rightarrow\ \mathsf{ValidTRS}\ \Delta\ \rightarrow\ \mathsf{Tms}\ \{\Xi\ \coloneqq\ \Xi\}\ \Delta\ \Gamma\ \rightarrow\ \mathbf{Type}\\
&\mathsf{Val} && : \boldsymbol{\Pi}\ \Gamma\ \mathsf{A}\ \Delta\ \Delta^{\mathsf{C}}\ \delta\\
& && \quad \rightarrow\ \mathsf{Env}\ \Xi\ \Delta\ \Gamma\ \Delta^{\mathsf{C}}\ \delta\ \rightarrow\ \mathsf{Tm}\ \Delta\ (\mathsf{A}\ [\ \delta\ ]_{\mathsf{Ty}})\ \rightarrow\ \mathbf{Type}\\
&\mathsf{eval} && : \boldsymbol{\Pi}\ \Delta^{\mathsf{C}}\ (\mathsf{t}\ :\ \mathsf{Tm}\ \Gamma\ \mathsf{A})\ (\rho\ :\ \mathsf{Env}\ \Xi\ \Delta\ \Gamma\ \Delta^{\mathsf{C}}\ \delta)\\
& && \quad \rightarrow\ \mathsf{Val}\ \Gamma\ \mathsf{A}\ \Delta\ \Delta^{\mathsf{C}}\ \delta\ \rho\ (\mathsf{t}\ [\ \delta\ ])\\
&\mathsf{eval}^{*} && : \boldsymbol{\Pi}\ \Theta^{\mathsf{C}}\ \delta\ (\rho\ :\ \mathsf{Env}\ \Xi\ \Theta\ \Delta\ \Theta^{\mathsf{C}}\ \sigma)\ \rightarrow\ \mathsf{Env}\ \Xi\ \Theta\ \Gamma\ \Theta^{\mathsf{C}}\ (\delta\ ;\ \sigma)
\end{aligned}
$$

Perhaps surprisingly, and unlike when constructing the standard model, we do not associate an environment with the signature. We can get away with simply recursively evaluating definitions every time we hit a call.

We define a specialised version of unquoting on pre-neutrals, uvalpre. The intuition here is that uvalpre first syntactically compares the given neutral with all LHSs of the TRS to see if it can be reduced, and then if it is still stuck, delegates to uval, which unquotes as usual.

$$
\begin{aligned}
&\mathsf{uvalpre} && : \boldsymbol{\Pi}\ \mathsf{A}\ \{\mathsf{t}\}\ \rightarrow\ \mathsf{PreNe}\ \Delta\ (\mathsf{A}\ [\ \delta\ ]_{\mathsf{Ty}})\ \mathsf{t}\ \rightarrow\ \mathsf{Val}\ \Gamma\ \mathsf{A}\ \Delta\ \Delta^{\mathsf{C}}\ \delta\ \rho\ \mathsf{t}\\
&\mathsf{uval} && : \boldsymbol{\Pi}\ \mathsf{A}\ \{\mathsf{t}\}\ \rightarrow\ \mathsf{Ne}\ \Delta\ (\mathsf{A}\ [\ \delta\ ]_{\mathsf{Ty}})\ \mathsf{t}\ \rightarrow\ \mathsf{Val}\ \Gamma\ \mathsf{A}\ \Delta\ \Delta^{\mathsf{C}}\ \delta\ \rho\ \mathsf{t}\\
&\mathsf{qval} && : \boldsymbol{\Pi}\ \mathsf{A}\ \{\mathsf{t}\}\ \rightarrow\ \mathsf{Val}\ \Gamma\ \mathsf{A}\ \Delta\ \Delta^{\mathsf{C}}\ \delta\ \rho\ \mathsf{t}\ \rightarrow\ \mathsf{Nf}\ \Delta\ (\mathsf{A}\ [\ \delta\ ]_{\mathsf{Ty}})\ \mathsf{t}
\end{aligned}
$$

Like in Section 2.4.3, we will cheat a bit, and assume functor laws for thinning environments hold definitionally (to avoid excessive transport clutter). Actually, for these laws to typecheck, we now also need to assume functor laws for thinning completed TRSs.

$$
\begin{aligned}
&\Gamma^{\mathsf{C}}\ [\ \mathsf{id}^{\mathsf{Th}}\ ]_{\mathsf{C}}\ \equiv\ \Gamma^{\mathsf{C}}\\
&\Gamma^{\mathsf{C}}\ [\ \delta^{\mathsf{Th}}\ ]_{\mathsf{C}}\ [\ \sigma^{\mathsf{Th}}\ ]_{\mathsf{C}}\ \equiv\ \Gamma^{\mathsf{C}}\ [\ \delta^{\mathsf{Th}}\ ;^{\mathsf{Th}}\ \sigma^{\mathsf{Th}}\ ]_{\mathsf{C}}\\
&\rho\ [\ \mathsf{id}^{\mathsf{Th}}\ ]_{\mathsf{Env}}\ \equiv\ \rho\\
&\rho\ [\ \sigma^{\mathsf{Th}}\ ]_{\mathsf{Env}}\ [\ \gamma^{\mathsf{Th}}\ ]_{\mathsf{Env}}\ \equiv\ \rho\ [\ \sigma^{\mathsf{Th}}\ ;^{\mathsf{Th}}\ \gamma^{\mathsf{Th}}\ ]_{\mathsf{Env}}
\end{aligned}
$$

The definition of environments now needs to account for local equations. We take inspiration from the standard model constructions for $\mathsf{SC}^{\textsc{Bool}}$ and $\mathsf{SC}^{\textsc{Def}}$, and require environments to hold evidence of convertibility of the LHS and RHS values.

$$
\begin{aligned}
&\ulcorner\_\urcorner\mathbb{B}^{\mathsf{Nf}}\ :\ \boldsymbol{\Pi}\ \mathsf{b}\ \rightarrow\ \mathsf{Nf}\ \Gamma\ \mathbb{B}\ \ulcorner\ \mathsf{b}\ \urcorner\mathbb{B}\\
&\ulcorner\ \mathbf{tt}\ \urcorner\mathbb{B}^{\mathsf{Nf}}\ \coloneqq\ \mathsf{tt}\\
&\ulcorner\ \mathbf{ff}\ \urcorner\mathbb{B}^{\mathsf{Nf}}\ \coloneqq\ \mathsf{ff}\\[1ex]
&\mathsf{Env}\ \Xi\ \Delta\ (\Gamma\ \rhd\ \mathsf{t}\ \rightsquigarrow\ \mathsf{b})\ \Delta^{\mathsf{C}}\ \delta\\
&\quad \coloneqq\ (\rho : \mathsf{Env}\ \Xi\ \Delta\ \Gamma\ \Delta^{\mathsf{C}}\ (\pi_{1\rightsquigarrow}\ \delta))\ \times\\
&\qquad\quad \mathsf{Nf}{\sim}\ \mathsf{rfl}{\sim}\ \mathsf{rfl}{\sim}\ (\pi_{2\rightsquigarrow}\ \delta)\ (\mathsf{eval}\ \Delta^{\mathsf{C}}\ \mathsf{t}\ \rho)\ \ulcorner\ \mathsf{b}\ \urcorner\mathbb{B}^{\mathsf{Nf}}
\end{aligned}
$$

Values are defined entirely as usual. Evaluation of substitutions, eval*, now needs to produce the proof of normal-form equality. This is achievable via mutually proving soundness of evaluation.

For evaluation, we focus just on the new case for calls. We split on the evaluated scrutinee in a top-level helper, eval-call.

$$
\begin{aligned}
&\mathsf{eval\text{-}call}\ :\ \boldsymbol{\Pi}\ \{\mathsf{f}\ :\ \mathsf{DefVar}\ \Xi\ \Gamma\ \mathsf{A}\}\ (\rho\ :\ \mathsf{Env}\ \Xi\ \Delta\ \Gamma\ \Delta^{\mathsf{C}}\ \delta)\\
&\qquad\qquad\quad (\mathsf{t}^{\mathsf{V}}\ :\ \mathsf{Nf}\ \Delta\ \mathbb{B}\ \mathsf{t})\\
&\qquad\qquad\quad (\mathsf{t}{\sim}\ :\ \mathsf{Tm}{\sim}\ \mathsf{rfl}{\sim}\ \mathsf{rfl}{\sim}\ \mathsf{t}\ (\mathsf{lookup}^{\mathsf{Sig}}\ \Xi\ \mathsf{f}\ .\mathsf{scrut}\ [\ \delta\ ]))\\
&\qquad\quad \rightarrow\ (\boldsymbol{\Pi}\ \mathsf{t}{\sim}'\ \rightarrow\ \mathsf{Nf}{\sim}\ \mathsf{rfl}{\sim}\ \mathsf{rfl}{\sim}\ (\mathsf{t}{\sim}\ \bullet{\sim}\ \mathsf{t}{\sim}')\ \mathsf{t}^{\mathsf{V}}\ \mathsf{tt}\\
&\qquad\qquad\quad \rightarrow\ \mathsf{Val}\ \Gamma\ \mathsf{A}\ \Delta\ \Delta^{\mathsf{C}}\ \delta\ \rho\ (\mathsf{lookup}^{\mathsf{Sig}}\ \Xi\ \mathsf{f}\ .\mathsf{lhs}\ [\ \delta\ ,_{\rightsquigarrow}\ \mathsf{t}{\sim}'\ ]))\\
&\qquad\quad \rightarrow\ (\boldsymbol{\Pi}\ \mathsf{t}{\sim}'\ \rightarrow\ \mathsf{Nf}{\sim}\ \mathsf{rfl}{\sim}\ \mathsf{rfl}{\sim}\ (\mathsf{t}{\sim}\ \bullet{\sim}\ \mathsf{t}{\sim}')\ \mathsf{t}^{\mathsf{V}}\ \mathsf{ff}\\
&\qquad\qquad\quad \rightarrow\ \mathsf{Val}\ \Gamma\ \mathsf{A}\ \Delta\ \Delta^{\mathsf{C}}\ \delta\ \rho\ (\mathsf{lookup}^{\mathsf{Sig}}\ \Xi\ \mathsf{f}\ .\mathsf{rhs}\ [\ \delta\ ,_{\rightsquigarrow}\ \mathsf{t}{\sim}'\ ]))\\
&\qquad\quad \rightarrow\ \mathsf{Val}\ \Gamma\ \mathsf{A}\ \Delta\ \Delta^{\mathsf{C}}\ \delta\ \rho\ (\mathsf{call}\ \mathsf{f}\ \delta)
\end{aligned}
$$

```
eval-call {f ≔ f} ρ tt t~ u^V v^V
  ≔ coe^Val {ρ ≔ ρ} rfl~ (sym~ (call-tt {f ≔ f} (sym~ t~))) u^V′
  where u^V′ ≔ u^V (sym~ t~) rfl~
eval-call {f ≔ f} ρ ff t~ u^V v^V
  ≔ coe^Val {ρ ≔ ρ} rfl~ (sym~ (call-ff {f ≔ f} (sym~ t~))) v^V′
  where v^V′ ≔ v^V (sym~ t~) rfl~
eval-call {f ≔ f} ρ (ne𝔹 t^Ne) t~ u^V v^V
  ≔ uvalpre _ (callNe {f ≔ f} (coeNe~ rfl~ rfl~ t~ t^Ne))
```

Unlike evaluation of dependent “if” (eval-if in Section 2.4.3), we do not rely on quoting here. When producing stuck calls, we have no reason to the normalise the branches.

To actually make use of eval-call, we need to evaluate the scrutinee, and the LHS and RHS branch under the appropriate convertibility assumptions.

```
eval {δ ≔ σ} Δ^C (call f δ) ρ
  ≔ eval-call {f ≔ f} δ^V t^V (≡~ refl) u^V v^V
  where δ^V ≔ eval* Δ^C δ ρ
        t^V ≔ eval Δ^C (lookup^Sig _ f .scrut) δ^V
        u^V ≔ λ t~ t^Nf~ → eval {δ ≔ (δ ; σ) ,⤳ t~} Δ^C (lookup^Sig _ f .lhs)
                                (δ^V , t^Nf~)
        v^V ≔ λ t~ t^Nf~ → eval {δ ≔ (δ ; σ) ,⤳ t~} Δ^C (lookup^Sig _ f .rhs)
                                (δ^V , t^Nf~)
```

We can ensure this case of evaluation stays structurally recursive by “Fording”. For example, $\text{lookup}^{\text{Sig}}$ _ f .scrut is not obviously structurally smaller than call f δ, but if we “Ford” by adding an extra term parameter to call, t : Tm Γ 𝔹 and the propositional equation t = $\text{lookup}^{\text{Sig}}$ _ f .scrut, the induction here becomes structurally well-founded.

We should make sure to check soundness. call-tt and call-ff are preserved up-to-coherence just by computation of eval. $\pi_{2\leadsto}$ instead requires us to prove

```
Nf~ rfl~ rfl~ (π_{2⤳} δ [ rfl~ ]~)
    (eval Θ^C (t [ π_{1⤳} δ ]) ρ) (eval Θ^C ⌜ b ⌝𝔹 ρ)
```

This is why we had to embed equations into environments. After splitting on the Boolean, the RHS reduces to tt/ff, and if we project our the convertibility evidence the environment, specifically eval* Θ^C δ ρ (focusing on the tt case WLOG), we obtain

```
Tm~ rfl~ rfl~ (eval Θ^C t (eval* Θ^C δ ρ .π₁)) tt
```

So it remains to prove equality of eval Θ^C (t [$\pi_{1\leadsto}$ δ]) ρ and eval Θ^C t (eval* Θ^C δ ρ .π_1), which is just preservation of $\pi_{1\leadsto}$.

The core unquoting (uval) and quoting (qval) operations stay mostly unchanged from ordinary NbE for dependent types[4], but we do of course need to implement uvalpre.

We first define a procedure for checking if any TRS rewrites possibly apply to a given pre-neutral term.

```
data CheckRwResult (Γ^TRS : TRS Γ) : PreNe Γ A t → Type where
  rw  : RwVar Γ^TRS (coe~ rfl~ A~ coh t^PreNe) b → CheckRwResult Γ^TRS t^PreNe
  stk : (Π A~ b → ¬ RwVar Γ^TRS (coe~ rfl~ A~ coh t^PreNe) b)
      → CheckRwResult Γ^TRS t^PreNe
checkrw : Π (Γ^TRS : TRS Γ) (t^PreNe : PreNe Γ A t)
        → CheckRwResult Γ^TRS t^PreNe
```

Note that as we are working with plain TRSs here, we need to work with terms up to coherence rather than up to conversion. We can prove that overall conversion is preserved using the correctness criteria associated with ValidTRSs after we are done.

We then implement uvalpre by splitting on the result of checkrw, and either returning the closed Boolean, or the stuck neutral, depending on the result. We need the “from” and “to” properties of our TRS here to translate between evidence about the existence or lack of rewrites and convertibility.

```
uvalpre {Δ^C ≔ Δ^C} A t^PreNe with checkrw (Δ^C .trs) t^PreNe
... | rw {A~ ≔ A~} {b ≔ b} r
```

[4] I say “mostly” because technically we do need to call uvalpre rather than uval in a couple of places to build new stuck neutrals, but other than that, the definitions are identical.

```
    ≔ coe^Val′ (sym~ A~) (sym~ (Δ^C .from r) •~ sym~ coh) ⌜ b ⌝𝔹^Nf
  ... | stk ¬r
    ≔ uval A (t^PreNe , λ b Γ~ A~ t~ →
              ¬r (A~ •~ 𝔹 {Γ~ ≔ sym~ Γ~}) b
                (Δ^C .to (sym~ coh •~ t~ •~ ⌜⌝𝔹~ {Γ~ ≔ sym~ Γ~})
                      (coe~ _ _ _ t^PreNe)))
```

Soundness of uvalpre also follows from “from” and “to”, so we are done!

```
  nbe : ValidTRS Γ → Π t → Nf Γ A t
  nbe Γ^C t ≔ qval {δ ≔ id} _ (eval Γ^C t id^Env)
```

Of course, we can only call into nbe if we have a ValidTRS, so we move on to the topic of constructing these now.

6.3 Elaboration

We first consider the task of generating ValidTRSs from a set of equational assumptions in a context, and then move on to presenting an elaboration algorithm which can turn **smart case** into $\mathrm{SC}^{\mathrm{DEF}}$ calls.

6.3.1 Syntactic Restrictions for Generating TRSs

As mentioned in Section 6.2.3, justifying completion is hard (because finding a well-founded order is hard). Luckily, completion is also no longer necessary. In $\mathrm{SC}^{\mathrm{DEF}}$, we can place essentially arbitrary restrictions on equations, without endangering subject reduction (stability under substitutions is no longer necessary).

One such restriction, for example, could be to require that the LHS of every reflected equation is syntactically a variable, essentially recovering dependent pattern matching (Section 3.1). Checking equality of variables is easy, so we can iterate through the set of equations $i \rightsquigarrow b$ and in the case of overlaps, either remove the offending equation (if it is redundant - i.e. the RHSs are equal Booleans) or report a definitional inconsistency (if it is definitionally inconsistent - i.e. the RHSs are not equal). Of course, the resulting theory would not be super exciting, given dependent pattern matching that is restricted in this way is standard (and the limitations therein ultimately motivated this entire project).

A more interesting strategy would be iterate over the set of equations, normalising each LHS, $t : \mathsf{Tm}\ \Gamma\ \mathbb{B}$, w.r.t. the prior equation set, building a ValidTRS as we go. Before moving on to the next equation, we inspect the reduced LHS, t, and:

- If t is a closed Boolean, we compare it to the RHS and either remove the redundant equation or immediately report the definitional inconsistency.
- If t is a neutral term, we check that it does not occur as a subterm of any of the prior neutral LHSs. If it does (the new rewrite *destabilises* the TRS so-far) then we can just report an error and ask the programmer to rewrite their program (doing a better job here really does require completion).

To justify this approach is sensible, we need to actually derive the "from" and "to" conditions associated with the TRS we construct. Attempting these proofs formally in Agda gets extremely painful, so we will give just the main ideas:

(A) We say a neutral *destabilises* a TRS if it occurs as a subterm of (or equals) any of the LHSs of that TRS.

(B) Given a ValidTRS for a context Γ and a proof that a particular neutral $t^{\mathsf{Ne}} : \mathsf{Ne}\ \Gamma\ \mathbb{B}\ t$ does not destabilise the underlying TRS, and a proof that t^{Ne} does not occur as a subterm of (or equals) $u^{\mathsf{Ne}} : \mathsf{Ne}\ \Gamma\ \mathbb{B}\ u$, we can obtain an $\mathsf{Ne}\ (\Gamma \rhd t \rightsquigarrow b)\ \mathbb{B}\ (u\ [\ \mathsf{wk}_{\rightsquigarrow}\ ])$.

(C) Given t^{Ne} cannot occur as a subterm of any of t^{Ne}'s direct subterms, we can also obtain $\mathsf{PreNe}\ (\Gamma \rhd \ \rightsquigarrow b)\ \mathbb{B}\ (t\ [\ \mathsf{wk}_{\rightsquigarrow}\ ])$.

(D) (B) and (C) are sufficient to construct the TRS $(\Gamma \rhd t \rightsquigarrow b)$, including a rewrite corresponding to the new equation.

(E) "from" for this new TRS can be proven by cases. If the RwVar is rz (i.e. the rewrite makes use of the last rewrite in the TRS), then eq ez proves the required equivalence (the last rewrite in the TRS maps exactly from the $\mathsf{PreNe}\ (\Gamma \rhd \ \rightsquigarrow b)\ \mathbb{B}\ (t\ [\ \mathsf{wk}_{\rightsquigarrow}\ ])$ to b).

(F) If the RwVar instead is of the form rs r, then we know the LHS is some neutral that was already present in the TRS, so we can reuse the existing evidence of from.

(H) Finally, to prove "to", we assume some way of getting our hands on a concrete EqVar corresponding to the convertibility evidence (recall that we should be able obtain this, albeit painfully, via introducing reduction Remark 6.2.3). We then perform a similar case split: ez maps to rz and es e can be dealt with using the prior "to" result.

I leave a full Agda mechanisation of this proof for future work. Most of the pain arises from parts (F) and (H), where we need to invert the the weakening of neutrals to account for the new equation.

6.3.2 Elaborating Case Splits

We now quickly outline how to elaborate from an untyped surface language that appears to feature local **smart case**, to SC^{Def}. Concretely, we will work with an untyped syntax resembling SC^{Bool}, and write the algorithm in bidirectional style ([115]), with a mutually recursive check and infer (as in [113], and also my Haskell SC^{Bool} typechecker (Section 5.4).

[115]: Dunfield et al. (2022), *Bidirectional Typing*

[113]: Coquand (1996), *An Algorithm for Type-Checking Dependent Types*

To account for local case splits being turned into new top level definitions, we consistently return a signature weakening along with elaborated SC^{Def} term. To be able to normalise types and check conversion, we also require the existence of a ValidTRS associated with the given context.

We also assume the existence of a definition of *normal types* (NfTy) here. The only difference between these are ordinary SC^{Def} types (with strictified substitution) is that large IF must always be blocked on a neutral term.

```
data NfTy : Π Γ → Ty {Ξ ≔ Ξ} Γ → Type
record InfTm (Γ : Ctx Ξ) : Type where
  constructor inf
  pattern
  field
    {infSig} : Sig
    infWk    : Wk infSig Ξ
    infTy    : Ty (Γ [ infWk ]^{Wk}_{Ctx})
    infTy^{Nf} : NfTy (Γ [ infWk ]^{Wk}_{Ctx}) infTy
    infTm    : Tm (Γ [ infWk ]^{Wk}_{Ctx}) infTy
record ChkTm (Γ : Ctx Ξ) (A : Ty Γ) : Type where
  constructor chk
  pattern
  field
    {elabSig} : Sig
    elabWk    : Wk elabSig Ξ
    elabTm    : Tm (Γ [ elabWk ]^{Wk}_{Ctx}) (A [ elabWk ]^{Wk}_{Ty})
check : ValidTRS Γ → NfTy Γ A → PreTm → Maybe (ChkTm Γ A)
infer : ValidTRS Γ → PreTm → Maybe (InfTm Γ)
```

Because our input is untyped, check and infer can fail (if the term is not typeable with the given type, or the type of the term is not inferrable, respectively). We use **do**-notation [116] to avoid excessive boilerplate matching on the results of recursive calls (elaboration should fail if any recursive call fails).

[116]: Agda Team (2024), *Syntactic Sugar*

check and infer for ordinary lambda calculus constructs (application, abstraction, etc.) is relatively standard. We just need to make sure to account for new top-level definitions generated during elaboration of subterms by composing the returned signature weakenings.

Elaborated terms being parameterised by a signature weakening, and needing to compose these for every recursive call, also feels quite monadic in nature (though the relevant category is no longer **Type**). It would perhaps be interesting for future work to explore how to eliminate this boilerplate.

(Un-annotated) λ-abstractions are not inferrable

```
infer Γ^C (λ t) ≔ nothing
```

However, we can infer applications by first inferring the LHS, ensuring that the synthesised type of the LHS is headed with Π, and checking also that the argument has the appropriate type

Technically we should also account for the case where the synthesised type of t is headed with a coercion..

```
infer Γ^C (t · u) ≔ do
  inf ϕ_1 (Π A B) (Π A^Nf B^Nf) t′ ≔ infer Γ^C t
    where _ → nothing
  let Γ^C′     ≔ Γ^C [ ϕ_1 ]^Wk_C
  chk ϕ_2 u′ ≔ check Γ^C′ A^Nf u
  just (inf  (ϕ_1 ;^Wk ϕ_2)
             _
             (normTy (Γ^C′ [ ϕ_2 ]^Wk_C) ((B [ ϕ_2 ]^Wk_Ty) [ < u′ > ]_Ty))
             ((t′ [ ϕ_2 ]^Wk) · u′))
```

We can also check (un-annotated) λ-abstractions by checking the body has the expected type (in the context extended by the domain)

```
check Γ^C (Π A^Nf B^Nf) (λ t) ≔ do
  chk ϕ t′ ≔ check (Γ^C [ wk^Th ]_C) B^Nf t
  just (chk ϕ (λ t′))
```

Of course, λ-abstractions are only typeable at Π-types

```
check Γ^C _ (λ t) ≔ nothing
```

We can check applications by first inferring a type, and then checking it matches the expected one. Actually, all inferrable terms can be checked using this approach.

```
check {A ≔ A} Γ^C A^Nf (t · u) ≔ do
  inf ϕ A′ _ tu′ ≔ infer Γ^C (t · u)
  Γ~ , A~        ≔ convTy Γ^C A′ (A [ ϕ ]^Wk_Ty)
  just (chk ϕ (coe~ Γ~ A~ tu′))
```

The interesting case here is really elaboration of **smart if**. We first recursively check the subterms, then construct a new definition using these, and finally return a call expression which simply calls the definition.

Note that as we would expect for **smart if**, we do not need a motive! Instead the LHS and RHS terms are just checked at the same type of the overall "if" expression, with dependent elimination coming from the new equations added to the context.

```
check {A ≔ A} Γ^C A^Nf (if t u v) ≔ do
  chk ϕ_1 t′ ≔ check Γ^C 𝔹 t
  Γtt^C      ≔ complete ((_ [ ϕ_1 ]^Wk_Ctx) ▷ t′ ⇝ tt)
  Γff^C      ≔ complete ((_ [ ϕ_1 ]^Wk_Ctx) ▷ t′ ⇝ ff)
  chk ϕ_2 u′ ≔ check? Γtt^C (A [ ϕ_1 ]^Wk_Ty [ wk_⇝ ]_Ty) u
  chk ϕ_3 v′ ≔ check? (Γff^C [ ϕ_2 ]?^+) (A [ ϕ_1 ;^Wk ϕ_2 ]^Wk_Ty [ wk_⇝ ]_Ty) v
  let ϕ_123  ≔ ϕ_1 ;^Wk (ϕ_2 ;^Wk ϕ_3)
  let Ξ′     ≔ _ ▷ _ → (A [ ϕ_123 ]^Wk_Ty)
                 if (t′ [ ϕ_2 ]^Wk [ ϕ_3 ]^Wk) ≔ u′ [ ϕ_3 ]^Wk | v′
  just (chk (ϕ_123 ;^Wk wk^Wk) (call {Ξ ≔ Ξ′} fz id))
```

We rely on a few helpers here. complete is a partial implementation of completion (capable of either returning a ValidTRS, evidence of a definitional inconsistency of failing). We described some possible implementations of this in Section 6.3.1.

We also need a slightly generalised version of check, to account for (improved) implementations of complete that sometimes return evidence of definitional inconsistency.

```
check? : TRS? Γ → Ty Γ A → PreTm → Maybe (ChkTm Γ A)
```

In a definitionally inconsistent context, all types and terms are convertible, so we can arbitrarily elaborate everything to tt (the inhabitant of the unit type is perhaps more appropriate, but any term will ultimately do).

$$\begin{aligned}
&\mathsf{check?}\ (\mathsf{compl}\ \Gamma^{\mathsf{C}})\ \mathsf{A}\ \mathsf{t} && \coloneqq\ \mathsf{check}\ \Gamma^{\mathsf{C}}\ (\mathsf{normTy}\ \Gamma^{\mathsf{C}}\ \mathsf{A})\ \mathsf{t}\\
&\mathsf{check?}\ (!!\ \Gamma!)\qquad\ \ \mathsf{A}\ \mathsf{t} && \coloneqq\ \mathsf{just}\ (\mathsf{chk}\ \mathsf{id}^{\mathsf{Wk}}\ (\mathsf{coe\sim}\ \mathsf{rfl\sim}\ (\mathsf{collapse}\ \Gamma!)\ \mathsf{tt}))
\end{aligned}$$

By working with intrinsically-typed syntax, this algorithm must be *sound* in at least the sense that it only produces well-typed $\mathrm{SC}^{\mathrm{DEF}}$ terms. However, in principle, we would probably expect a stronger soundness condition on elaboration, expressing in some sense that the semantic meaning of the input PreTm is preserved[5]. Furthermore, we might also expect a completeness property, expressing that if a pre-term is sufficiently annotated and typeable, then elaboration should succeed. Ideas from [58] are likely to be highly relevant here. We leave the work of defining and checking such additional correctness criteria to future work.

[5] The first step here, naturally, would be to actually give some semantic meaning to untyped pre-terms.

[58]: Kovács (2024), *Basic setup for formalizing elaboration*

Evaluation and Future Work 7

The goals of this project were to build a proof-of-concept typechecker for **smart case**, and to make progress on the metatheory of type theories with local equations. Both of these were achieved to some extent, though there is still a significant amount of work remaining before a full implementation in a mainstream proof assistant can be justified.

SC$^{\text{Bool}}$

As demonstrated at the very start of this report (Chapter 1) The poof-of-concept SC$^{\text{Bool}}$ typechecker was successful in at least the sense that it admits vastly simpler proofs of some theorems (specifically those with a heavy reliance on Boolean case splitting, such as f **tt** = f (f (f **tt**))) than one can write in e.g. Agda. This demonstrates the practicality of **smart case** and its potential benefits to some extent. Unfortunately, the it is also very-much specialised to Boolean equations, and extending beyond this without hitting issues with subject reduction or undecidability appears challenging (Section 5.3). On the other hand, I argue that the analysis and counter-examples here are still a valuable contribution, especially given the lack prior work (recall that Altenkirch et al.'s work on **smart case** [17] was never published).

[17]: Altenkirch (2011), *The case of the smart case*

The typechecker is also written in somewhat "naive Haskell". De Bruijn variables are encoded in unary form, proofs using singletons will perform computation at runtime despite their output ultimately being irrelevant and the data structure choices are very sub-optimal (most glaringly, local equations are stored as a list of pairs of neutrals and values, rather than an efficient map data structure). Future work on optimising and exploring the overall performance impact of supporting **smart case** (e.g. comparing against ordinary **with** abstraction) would be a good idea before rushing to implement the feature in mainstream proof assistants.

Another interesting optimisation could be to leverage non-destructive term rewriting techniques such as E-Graphs [9, 10] for equations between neutrals. I think destructive rewriting is ultimately required for equations where one side is of introduction form (so later β-reductions get unblocked), but conversion checking modulo equations between neutrals can be delayed.

[9]: Nelson (1980), *Techniques for program verification*
[10]: Willsey et al. (2021), *egg: Fast and extensible equality saturation*

SC$^{\text{Def}}$

I feel positive about the potential for ideas from SC$^{\text{Def}}$ to form the basis for future proof assistant development. Being able to restrict equations with properties that are not stable under substitution gives a huge amount of flexibility, and normalisation not needing to interleave with completion vastly simplifies the metatheory. Another nice advantage of the SC$^{\text{Def}}$ approach is that fits nicely with the design of some existing proof assistants, including Agda[1].

An unexpected bonus feature of SC$^{\text{Def}}$ is that is suggests a way to enable preserving definitional equations across top-level definitions[2]. In Agda, sometimes abstracting over a repeated argument necessitates additional transport boilerplate, because definitional equations which hold in the concrete cases can only be stated propositionally in the abstract setting. To resolve this, it could be interesting to explore direct surface syntax for this feature, rather than leaving it as a mere detail of the encoding.

There is still a significant amount of remaining work on the metatheory of SC$^{\text{Def}}$. Our normalisation result only accounts for reflecting Boolean equations, and relies on the existence of a completed term-rewriting system (TRS) associated with the set of equations in the context. Section 6.3.1 describes a possible approach to generating these, but it restricts the set of acceptable equations in a quite significant way[3]. Leveraging

[1] Agda already elaborates **with**-abstractions to top-level definitions.

[2] Specifically, by using full substitutions, including those that instantiate contextual equations, as argument lists.

[3] Specifically, LHSs of later equations cannot occur as subterms in prior ones. In practice, this means that users of **smart case** would sometimes have to carefully order their case splits in order to avoid destabilising previous equations and getting an error.

completion to justify a wider set of equations could be exciting future work (this would require proving some sort of strong normalisation result).

Before integrating $\mathrm{SC}^{\mathrm{Def}}$ with the core type theories of existing proof assistant, there also needs to be extensive work on analysing the interactions between $\mathrm{SC}^{\mathrm{Def}}$ and a myriad of other modern proof assistant features (e.g. global rewrite rules [11], cubical identity [31], quotient types). $\mathrm{SC}^{\mathrm{Def}}$ definitions are also quite limited in the sense that they can only depend on prior ones - that is, (mutually) recursive definitions are not possible. Integrating $\mathrm{SC}^{\mathrm{Def}}$ with with work on justifying (structurally) recursive definitions [117], type-based termination [118, 119] or even going further and elaborating uses of induction into eliminators following [120, 121] would be important future work. It could be interesting to also examine going even further with elaboration, following work such as [63–65] to elaborate $\mathrm{SC}^{\mathrm{Def}}$ into a traditional intensional type theory (without equational assumptions).

[11]: Cockx (2019), *Type Theory Unchained: Extending Agda with User-Defined Rewrite Rules*

[31]: Cohen et al. (2015), *Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom*

[117]: Abel et al. (2002), *A predicative analysis of structural recursion*

[118]: Barthe et al. (2006), *$\widehat{\mathit{CIC}}$: Type-Based Termination of Recursive Definitions in the Calculus of Inductive Constructions*

[119]: Nisht (2024), *Type-Based Termination Checking in Agda*

[120]: Goguen et al. (2006), *Eliminating Dependent Pattern Matching*

[121]: Cockx et al. (2018), *Elaborating dependent (co)pattern matching*

[63]: Hofmann (1995), *Conservativity of Equality Reflection over Intensional Type Theory*

[64]: Oury (2005), *Extensionality in the Calculus of Constructions*

[65]: Winterhalter et al. (2019), *Eliminating reflection from type theory*

Because I had the idea for $\mathrm{SC}^{\mathrm{Def}}$ quite late into the project, I did not have time to write a typechecker implementation with which to directly demonstrate its utility. Beyond the elaboration of case splits (which I cover in detail in Section 6.3.2), I expect a similar implementation to $\mathrm{SC}^{\mathrm{Bool}}$ (tracking neutral to value mappings during NbE) to be feasible.

Mechanisation and Meta-Metatheory

Taking a more general perspective, this project can also be seen as an exploration in studying the metatheory of type theory from a perspective grounded in mechanisation. We have used the proof assistant Agda as our metatheory throughout. A hugely exciting possibility that arises from committing to this approach is the potential to build correct-by-construction, type theory implementations (i.e. verified typecheckers) [122]. With Section 6.3.2, a genuine Agda implementation of an $\mathrm{SC}^{\mathrm{Def}}$ typechecker does not seem completely out of reach, but of course actually going the distance here would require much more work fleshing out the details (fleshing out all the details of the NbE proof, defining normalisation of types, checking equality up-to-coherence of normal forms, etc.)

[122]: Chapman (2008), *Type Theory Should Eat Itself*

We also relied on many unsafe features to define strictified syntaxes. This was successful at avoiding a lot of the transport boilerplate while staying relatively concise (the full strictified $\mathrm{SC}^{\mathrm{Def}}$ syntax is slightly over 500 lines of Agda), but future work could increase the level of trust in these mechanised proofs by leveraging the construction of [7] to strictify substitution laws safely.

[7]: Kaposi et al. (2025), *Type Theory in Type Theory Using a Strictified Syntax*

Finally, I think it is also worth reflecting on whether the focus on (categorically-inspired) intrinsically-typed syntax (following e.g. [52, 53]), as opposed to the extrinsic approach (where typing relations are defined on untyped terms, used in e.g. [48]) was ultimately the right decision. I think the benefits of taking a more “semantic” [7] definition of type theory are in part demonstrated by the soundness proofs and the presentation of normalisation by evaluation for ordinary dependent type theory (Section 2.4.3), in which semantic equivalence of terms (conversion) is preserved throughout.

[52]: Danielsson (2006), *A Formalisation of a Dependently Typed Language as an Inductive-Recursive Family*

[53]: Altenkirch et al. (2016), *Type theory in type theory using quotient inductive types*

[48]: Abel et al. (2018), *Decidability of conversion for type theory in type theory*

However, in the case of normalisation for $\mathrm{SC}^{\mathrm{Def}}$ (Section 6.2), the story gets a little messier, with the term rewriting aspects heavily relying on syntactic analysis of pre-neutral terms. The overall normalisation algorithm is still sound, but individual steps do not appear to preserve conversion. Making this rigorous requires some quite ugly and repetitive setoid reasoning, which I have not gone through the full details of. Future work could aim to rectify this messiness by somehow adjusting the NbE model/algorithm such that conversion is fully preserved (though I am not sure how one could actually achieve this) or by translating the argument into a theory with direct support for working at different levels of abstraction (i.e. 2LTT [57, 58]).

[57]: Annenkov et al. (2023), *Two-level type theory and applications*

[58]: Kovács (2024), *Basic setup for formalizing elaboration*

Declarations 8

- **GenAI Use:** GenAI was not used at any point during this project. I remain unconvinced that GenAI, at least as it currently exists, has a place in formal methods research.
- **Ethical Considerations:** I do not believe there were any direct ethical risks associated with this project.
- **Sustainability:** I expect my refusal to use GenAI combined with the fact that this project did not involve any machine learning techniques immediately places its carbon footprint on the lower end of Imperial final year projects.
- **Artifact:** The source for the Haskell SC$^{\text{Bool}}$ typechecker, the various companion Agda mechanisations and the literate Agda comprising the report itself are all available at `https://github.com/NathanielB123/fyp/tree/main`.

Bibliography

[1] Martín H. Escardó and contributors. *TypeTopology*. Agda development. 2025. URL: https://github.com/martinescardo/TypeTopology (cited on page 1).

[2] Kevin Buzzard and contributors. *FLT. Ongoing Lean formalisation of the proof of Fermat's Last Theorem.* 2025. URL: https://github.com/ImperialCollegeLondon/FLT (cited on page 1).

[3] Loïc Pujet and Nicolas Tabareau. "Observational equality: now for good". In: *Proc. ACM Program. Lang.* 6.POPL (2022), pp. 1–27. DOI: 10.1145/3498693 (cited on pages 1, 9).

[4] Andreas Abel, Nils Anders Danielsson, and Oskar Eriksson. "A Graded Modal Dependent Type Theory with a Universe and Erasure, Formalized". In: *Proc. ACM Program. Lang.* 7.ICFP (2023), pp. 920–954. DOI: 10.1145/3607862 (cited on page 1).

[5] Jessica Shi, Cassia Torczon, Harrison Goldstein, Benjamin C. Pierce, and Andrew Head. "QED in Context: An Observation Study of Proof Assistant Users". In: *Proc. ACM Program. Lang.* 9.OOPSLA1 (2025), pp. 337–363. DOI: 10.1145/3720426 (cited on page 1).

[6] 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, TyDe 2024, Milan, Italy, 6 September 2024.* Ed. by Sandra Alves and Jesper Cockx. ACM, 2024, pp. 2–15. DOI: 10.1145/3678000.3678201 (cited on page 1).

[7] Ambrus Kaposi and Loïc Pujet. "Type Theory in Type Theory Using a Strictified Syntax". 2025. URL: https://pujet.fr/pdf/strictification_preprint.pdf (cited on pages 1, 18, 28, 42, 104).

[8] Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge university press, 1998 (cited on pages 1, 48).

[9] Charles Gregory Nelson. "Techniques for program verification". PhD thesis. Stanford University, 1980. URL: https://people.eecs.berkeley.edu/~necula/Papers/nelson-thesis.pdf (cited on pages 1, 103).

[10] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. "egg: Fast and extensible equality saturation". In: *Proc. ACM Program. Lang.* 5.POPL (2021), pp. 1–29. DOI: 10.1145/3434304 (cited on pages 1, 103).

[11] Jesper Cockx. "Type Theory Unchained: Extending Agda with User-Defined Rewrite Rules". In: *25th International Conference on Types for Proofs and Programs, TYPES 2019, June 11-14, 2019, Oslo, Norway.* Ed. by Marc Bezem and Assia Mahboubi. Vol. 175. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 2:1–2:27. DOI: 10.4230/LIPICS.TYPES.2019.2 (cited on pages 1, 28, 51, 104).

[12] Yann Leray, Gaëtan Gilbert, Nicolas Tabareau, and Théo Winterhalter. "The Rewster: Type Preserving Rewrite Rules for the Coq Proof Assistant". In: LIPIcs 309 (2024). Ed. by Yves Bertot, Temur Kutsia, and Michael Norrish, 26:1–26:18. DOI: 10.4230/LIPICS.ITP.2024.26 (cited on pages 1, 28, 51).

[13] Thorsten Altenkirch. *Smart Case [Re: [Agda] A puzzle with "with"]. The Agda Mailing List.* 2009. URL: https://lists.chalmers.se/pipermail/agda/2009/001106.html (cited on page 1).

[14] Various Contributors. *Relation.Binary.EqReasoning. The Agda Standard Library 2.1.1.* 2024. URL: https://agda.github.io/agda-stdlib/v2.1.1/Relation.Binary.Reasoning.Setoid.html (visited on 01/20/2025) (cited on page 2).

[15] Thorsten Altenkirch and Nicolas Oury. "ΠΣ: A Core Language for Dependently Typed Programming". 2008. URL: https://people.cs.nott.ac.uk/psztxa/publ/pisigma.pdf (cited on page 2).

[16] Thorsten Altenkirch, Nils Anders Danielsson, Andres Löh, and Nicolas Oury. "PiSigma: Dependent Types without the Sugar". In: *Functional and Logic Programming, 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings.* Ed. by Matthias Blume, Naoki Kobayashi, and Germán Vidal. Vol. 6009. Lecture Notes in Computer Science. Springer, 2010, pp. 40–55. DOI: 10.1007/978-3-642-12251-4_5 (cited on page 2).

[17] Thorsten Altenkirch. "The case of the smart case. How to implement conditional convertibility?" In: *Presented at NII Shonan Seminar 007.* 2011. URL: https://shonan.nii.ac.jp/archives/seminar/007/files/2011/09/altenkirch_slides.pdf (cited on pages 2, 47, 55, 103).

[18] Ulf Norell. “Towards a practical programming language based on dependent type theory”. PhD thesis. Chalmers University of Technology, 2007 (cited on page 3).

[19] The Agda Team. *Agda*. Version 2.7.0.1. 2024. URL: `https://agda.readthedocs.io/` (visited on 05/14/2025) (cited on page 3).

[20] William Alvin Howard. “The Formulae-as-Types Notion of Construction”. In: *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism.* Ed. by Haskell Curry, Hindley B., Seldin J. Roger, and P. Jonathan. Academic Press, 1980 (cited on page 3).

[21] Meven Lennon-Bertrand. “Á Bas L’η. Coq’s Troublesome η-Conversion”. In: *Presented at the Workshop on the Implementation of Type Systems (WITS) 2022.* 2022. URL: `https://www.meven.ac/documents/WITS-22.pdf` (cited on pages 5, 82).

[22] András Kovács. “Eta conversion for the unit type. (is still not that simple)”. In: *Presented at the Workshop on the Implementation of Type Systems (WITS) 2024.* 2025. URL: `https://andraskovacs.github.io/pdfs/wits25prez.pdf` (cited on pages 5, 24).

[23] Thorsten Altenkirch, Peter Dybjer, Martin Hofmann, and Philip J. Scott. “Normalization by Evaluation for Typed Lambda Calculus with Coproducts”. In: *16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, June 16-19, 2001, Proceedings.* IEEE Computer Society, 2001, pp. 303–310. DOI: `10.1109/LICS.2001.932506` (cited on pages 5, 16, 34, 52–54).

[24] “Une Extension De L’Interpretation De Gödel a L’analyse, Et Son Application a L’Elimination Des Coupures Dans L’Analyse Et La Theorie Des Types”. In: *Proceedings of the Second Scandinavian Logic Symposium.* Ed. by J.E. Fenstad. Vol. 63. Studies in Logic and the Foundations of Mathematics. Elsevier, 1971, pp. 63–92. DOI: `10.1016/S0049-237X(08)70843-7` (cited on page 6).

[25] John C. Reynolds. “Towards a theory of type structure”. In: *Programming Symposium, Proceedings Colloque sur la Programmation, Paris, France, April 9-11, 1974.* Ed. by Bernard J. Robinet. Vol. 19. Lecture Notes in Computer Science. Springer, 1974, pp. 408–423. DOI: `10.1007/3-540-06859-7\_148` (cited on page 6).

[26] Jean-Yves Girard. “The System F of Variable Types, Fifteen Years Later”. In: *Theor. Comput. Sci.* 45.2 (1986), pp. 159–192. DOI: `10.1016/0304-3975(86)90044-7` (cited on page 6).

[27] Antonius J. C. Hurkens. “A Simplification of Girard’s Paradox”. In: *Typed Lambda Calculi and Applications, Second International Conference on Typed Lambda Calculi and Applications, TLCA ’95, Edinburgh, UK, April 10-12, 1995, Proceedings.* Ed. by Mariangiola Dezani-Ciancaglini and Gordon D. Plotkin. Vol. 902. Lecture Notes in Computer Science. Springer, 1995, pp. 266–278. DOI: `10.1007/BFB0014058` (cited on pages 6, 82).

[28] The Agda Team. *Universe Levels. The Agda 2.7.0.1 User Manual.* 2024. URL: `https://agda.readthedocs.io/en/v2.7.0.1/language/universe-levels.html` (visited on 05/14/2025) (cited on page 6).

[29] Thomas Streicher. “Investigations into intensional type theory”. habilthesis. 1993, p. 57. URL: `https://ncatlab.org/nlab/files/Streicher-IntensionalTT.pdf` (cited on pages 6, 53).

[30] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics.* Institute for Advanced Study: `https://homotopytypetheory.org/book`, 2013 (cited on page 6).

[31] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. “Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom”. In: *21st International Conference on Types for Proofs and Programs, TYPES 2015, May 18-21, 2015, Tallinn, Estonia.* Ed. by Tarmo Uustalu. Vol. 69. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015, 5:1–5:34. DOI: `10.4230/LIPICS.TYPES.2015.5` (cited on pages 6, 104).

[32] The Agda Team. *Data Types. The Agda 2.7.0.1 User Manual.* 2024. URL: `https://agda.readthedocs.io/en/v2.7.0.1/language/data-types.html` (visited on 05/14/2025) (cited on page 7).

[33] András Kovács. “Type-theoretic signatures for algebraic theories and inductive types”. PhD thesis. Eötvös Loránd University, 2023. DOI: `10.48550/ARXIV.2302.08837` (cited on page 7).

[34] András Kovács. *What are the complex induction patterns supported by Agda? Proof Assistants StackExchange Answer.* 2023. URL: `https://proofassistants.stackexchange.com/a/2002` (visited on 05/14/2025) (cited on page 7).

[35] Per Martin-Löf. “An Intuitionistic Theory of Types: Predicative Part”. In: *Logic Colloquium ’73.* Ed. by H.E. Rose and J.C. Shepherdson. Vol. 80. Studies in Logic and the Foundations of Mathematics. Elsevier, 1975, pp. 73–118. DOI: `https://doi.org/10.1016/S0049-237X(08)71945-1` (cited on page 7).

[36] Frank Pfenning and Christine Paulin-Mohring. “Inductively Defined Types in the Calculus of Constructions”. In: *Mathematical Foundations of Programming Semantics, 5th International Conference, Tulane University, New Orleans, Louisiana, USA, March 29 - April 1, 1989, Proceedings.* Ed. by Michael G. Main, Austin Melton, Michael W. Mislove, and David A. Schmidt. Vol. 442. Lecture Notes in Computer Science. Springer, 1989, pp. 209–228. DOI: `10.1007/BFB0040259` (cited on page 7).

[37] 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.* Ed. by André Platzer and Geoff Sutcliffe. Vol. 12699. Lecture Notes in Computer Science. Springer, 2021, pp. 625–635. DOI: `10.1007/978-3-030-79876-5_37` (cited on pages 7, 24).

[38] The Rocq Team. *The Rocq Reference Manual – Release 9.0.* 2025. URL: `https://coq.inria.fr/doc/v9.0/refman/` (cited on pages 7, 24).

[#7602] Szumi Xie. *Transport in HIT not strictly positive. Agda GitHub issue.* 2025. URL: `https://github.com/agda/agda/issues/7905` (cited on page 9).

[39] Thorsten Altenkirch. “From setoid hell to homotopy heaven? The role of extensionality in Type Theory”. In: *Presented at 23rd International Conference on Types for Proofs and Programs TYPES 2017.* 2017. URL: `https://people.cs.nott.ac.uk/psztxa/talks/types-17-hell.pdf` (cited on page 9).

[40] Thorsten Altenkirch. “Extensional Equality in Intensional Type Theory”. In: *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999.* IEEE Computer Society, 1999, pp. 412–420. DOI: `10.1109/LICS.1999.782636` (cited on page 9).

[41] Thorsten Altenkirch, Simon Boulier, Ambrus Kaposi, and Nicolas Tabareau. “Setoid Type Theory - A Syntactic Translation”. In: *Mathematics of Program Construction - 13th International Conference, MPC 2019, Porto, Portugal, October 7-9, 2019, Proceedings.* Ed. by Graham Hutton. Vol. 11825. Lecture Notes in Computer Science. Springer, 2019, pp. 155–196. DOI: `10.1007/978-3-030-33636-3\_7` (cited on page 9).

[42] Conor McBride. *There is no such thing as a free variable. There are only variables bound in the context. Mastodon Post.* 2025. URL: `https://types.pl/@pigworker/114087501391646354` (visited on 03/07/2025) (cited on page 10).

[43] N.G de Bruijn. “Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem”. In: vol. 75. 5. 1972, pp. 381–392. DOI: `https://doi.org/10.1016/1385-7258(72)90034-0` (cited on page 10).

[44] Thorsten Altenkirch, Nathaniel Burke, and Philip Wadler. “Substitution without copy and paste”. 2025. URL: `https://github.com/txa/substitution/blob/main/lfmtp25-submission.pdf` (cited on pages 11, 19, 66).

[45] Masako Takahashi. “Parallel Reductions in lambda-Calculus”. In: *Inf. Comput.* 118.1 (1995), pp. 120–127. DOI: `10.1006/INCO.1995.1057` (cited on pages 17, 60).

[46] Neil Ghani. “Adjoint Rewriting”. PhD thesis. University of Edinburgh, 1995. URL: `https://era.ed.ac.uk/bitstream/handle/1842/404/ECS-LFCS-95-339.PDF` (cited on pages 18, 53).

[47] Sam Lindley. “Extensional Rewriting with Sums”. In: *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings.* Ed. by Simona Ronchi Della Rocca. Vol. 4583. Lecture Notes in Computer Science. Springer, 2007, pp. 255–271. DOI: `10.1007/978-3-540-73228-0_19` (cited on pages 18, 53).

[48] Andreas Abel, Joakim Öhman, and Andrea Vezzosi. “Decidability of conversion for type theory in type theory”. In: *Proc. ACM Program. Lang.* 2.POPL (2018), 23:1–23:29. DOI: `10.1145/3158111` (cited on pages 18, 104).

[49] Peter Dybjer. “Internal Type Theory”. In: *Types for Proofs and Programs, International Workshop TYPES’95, Torino, Italy, June 5-8, 1995, Selected Papers.* Ed. by Stefano Berardi and Mario Coppo. Vol. 1158. Lecture Notes in Computer Science. Springer, 1995, pp. 120–134. DOI: `10.1007/3-540-61780-9_66` (cited on page 18).

[50] Simon Castellan, Pierre Clairambault, and Peter Dybjer. “Categories with Families: Unityped, Simply Typed, and Dependently Typed”. In: *CoRR* abs/1904.00827 (2019). URL: `http://arxiv.org/abs/1904.00827` (cited on pages 18, 19).

[51] Bruno Barras and Benjamin Werner. “Coq in Coq”. 1997. URL: `https://www.lix.polytechnique.fr/Labo/Bruno.Barras/publi/coqincoq.pdf` (cited on page 23).

[52] Nils Anders Danielsson. “A Formalisation of a Dependently Typed Language as an Inductive-Recursive Family”. In: *Types for Proofs and Programs, International Workshop, TYPES 2006, Nottingham, UK, April 18-21, 2006, Revised Selected Papers*. Ed. by Thorsten Altenkirch and Conor McBride. Vol. 4502. Lecture Notes in Computer Science. Springer, 2006, pp. 93–109. doi: 10.1007/978-3-540-74464-1_7 (cited on pages 23, 104).

[53] Thorsten Altenkirch and Ambrus Kaposi. “Type theory in type theory using quotient inductive types”. In: (2016). Ed. by Rastislav Bodík and Rupak Majumdar, pp. 18–29. doi: 10.1145/2837614.2837638 (cited on pages 23, 104).

[54] Edwin C. Brady. “Idris 2: Quantitative Type Theory in Practice”. In: LIPIcs 194 (2021). Ed. by Anders Møller and Manu Sridharan, 9:1–9:26. doi: 10.4230/LIPICS.ECOOP.2021.9 (cited on page 24).

[55] Théo Winterhalter. “Formalisation and meta-theory of type theory”. PhD thesis. Université de Nantes, 2020. url: https://github.com/TheoWinterhalter/phd-thesis/releases/download/v1.2.1/TheoWinterhalter-PhD-v1.2.1.pdf (cited on page 24).

[56] Kenji Maillard. “Splitting Booleans with Normalization-by-Evaluation”. In: *Presented at 30th International Conference on Types for Proofs and Programs TYPES 2024*. 2024, p. 121. url: https://kenji.maillard.blue/Presentations/boolextTypes24.pdf (cited on pages 24, 54).

[57] Danil Annenkov, Paolo Capriotti, Nicolai Kraus, and Christian Sattler. “Two-level type theory and applications”. In: *Math. Struct. Comput. Sci.* 33.8 (2023), pp. 688–743. doi: 10.1017/S0960129523000130 (cited on pages 26, 104).

[58] András Kovács. *Basic setup for formalizing elaboration. GitHub Gist.* 2024. url: https://gist.github.com/AndrasKovacs/1758f83cced957afb00b1382a8974c92 (visited on 01/10/2025) (cited on pages 26, 102, 104).

[59] András Kovács. “Staged compilation with two-level type theory”. In: *Proc. ACM Program. Lang.* 6.ICFP (2022), pp. 540–569. doi: 10.1145/3547641 (cited on page 26).

[60] Thorsten Altenkirch and Ambrus Kaposi. “Normalisation by Evaluation for Type Theory, in Type Theory”. In: *Log. Methods Comput. Sci.* 13.4 (2017). doi: 10.23638/LMCS-13(4:1)2017 (cited on pages 28, 30, 37, 92).

[61] Ambrus Kaposi. “Towards quotient inductive-inductive-recursive types”. In: *Presented at 29th International Conference on Types for Proofs and Programs TYPES 2023*. 2023. url: https://akaposi.github.io/pres_types_2023.pdf (cited on page 28).

[62] Ali Assaf et al. “Dedukti: a Logical Framework based on the $\lambda\Pi$-Calculus Modulo Theory”. In: *CoRR* abs/2311.07185 (2023). doi: 10.48550/ARXIV.2311.07185 (cited on pages 28, 51).

[63] Martin Hofmann. “Conservativity of Equality Reflection over Intensional Type Theory”. In: *Types for Proofs and Programs, International Workshop TYPES’95, Torino, Italy, June 5-8, 1995, Selected Papers*. Ed. by Stefano Berardi and Mario Coppo. Vol. 1158. Lecture Notes in Computer Science. Springer, 1995, pp. 153–164. doi: 10.1007/3-540-61780-9_68 (cited on pages 28, 66, 104).

[64] Nicolas Oury. “Extensionality in the Calculus of Constructions”. In: *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*. Ed. by Joe Hurd and Thomas F. Melham. Vol. 3603. Lecture Notes in Computer Science. Springer, 2005, pp. 278–293. doi: 10.1007/11541868_18 (cited on pages 28, 104).

[65] 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, CPP 2019, Cascais, Portugal, January 14-15, 2019*. Ed. by Assia Mahboubi and Magnus O. Myreen. ACM, 2019, pp. 91–103. doi: 10.1145/3293880.3294095 (cited on pages 28, 52, 66, 104).

[#7602] Nathaniel Burke. *Associativity of vector concatenation REWRITE sometimes doesn’t apply. Agda GitHub issue.* 2024. url: https://github.com/agda/agda/issues/7602 (cited on page 28).

[#6643] Amélia Liao. *#6643: Rewrite rules are allowed in implicit mutual blocks. Agda GitHub issue.* 2023. url: https://github.com/agda/agda/issues/6643 (cited on page 29).

[66] Ulrich Berger and Helmut Schwichtenberg. “An Inverse of the Evaluation Functional for Typed lambda-calculus”. In: *Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS ’91), Amsterdam, The Netherlands, July 15-18, 1991.* IEEE Computer Society, 1991, pp. 203–211. doi: 10.1109/LICS.1991.151645 (cited on page 30).

[67] 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, UK, August 7-11, 1995, Proceedings*. Ed. by David H. Pitt, David E. Rydeheard, and Peter T. Johnstone. Vol. 953. Lecture Notes in Computer Science. Springer, 1995, pp. 182–199. DOI: 10.1007/3-540-60164-3_27 (cited on page 30).

[68] András Kovács. *smalltt*. 2023. URL: https://github.com/AndrasKovacs/smalltt (visited on 03/07/2025) (cited on page 30).

[69] Gabriel Scherer. “Deciding equivalence with sums and the empty type”. In: (2017). Ed. by Giuseppe Castagna and Andrew D. Gordon, pp. 374–386. DOI: 10.1145/3009837.3009901 (cited on page 30).

[70] Chantal Keller and Thorsten Altenkirch. “Hereditary Substitutions for Simple Types, Formalized”. In: *Proceedings of the 3rd ACM SIGPLAN Workshop on Mathematically Structured Functional Programming, MSFP@ICFP 2010, Baltimore, MD, USA, September 25, 2010*. Ed. by Venanzio Capretta and James Chapman. ACM, 2010, pp. 3–10. DOI: 10.1145/1863597.1863601 (cited on page 31).

[71] Allais Guillaume and Naïm Camille Favier. *Combining Accessibility Proofs and Structural Ordering. Discussion on the Agda Zulip Server*. 2024. URL: https://agda.zulipchat.com/#narrow/channel/238741-general/topic/Combining.20Accessibility.20Proofs.20and.20Structural.20Ordering (cited on page 31).

[72] Amélia Liao. *Exponential objects in presheaf categories. 1Lab*. 2025. URL: https://1lab.dev/Cat.Instances.Presheaf.Exponentials.html (visited on 04/05/2025) (cited on page 34).

[73] András Kovács. “A machine-checked correctness proof of normalization by evaluation for simply typed lambda calculus”. MA thesis. Eötvös Loránd University, Budapest, 2017. URL: https://andraskovacs.github.io/pdfs/mscthesis.pdf (cited on pages 34, 36).

[74] Thierry Coquand. “Pattern matching with dependent types”. In: *Informal proceedings of Logical Frameworks*. Vol. 92. 1992, pp. 66–79. URL: https://wonks.github.io/type-theory-reading-group/papers/proc92-coquand.pdf (cited on page 43).

[75] Jesper Cockx. “Dependent Pattern Matching and Proof-Relevant Unification”. PhD thesis. Katholieke Universiteit Leuven, Belgium, 2017. URL: https://lirias.kuleuven.be/handle/123456789/583556 (cited on pages 43, 44).

[76] “Haskell 2010 Language Report”. In: (2010). Ed. by Simon Marlow. URL: https://www.haskell.org/onlinereport/haskell2010/ (cited on page 43).

[77] Conor McBride. “Dependently typed functional programs and their proofs”. PhD thesis. University of Edinburgh, UK, 2000. URL: https://hdl.handle.net/1842/374 (cited on page 44).

[78] Conor McBride and James McKinna. “The view from the left”. In: *J. Funct. Program.* 14.1 (2004), pp. 69–111. DOI: 10.1017/S0956796803004829 (cited on page 45).

[79] The Agda Team. *With-Abstraction. The Agda 2.7.0.1 User Manual*. 2024. URL: https://agda.readthedocs.io/en/v2.7.0.1/language/with-abstraction.html (visited on 01/20/2025) (cited on pages 45, 86).

[80] Various Contributors. *Views and the "with" rule. A Crash Course in Idris 2*. 2023. URL: https://idris2.readthedocs.io/en/latest/tutorial/views.html (visited on 05/20/2025) (cited on page 45).

[81] Various Contributors. *Relation.Binary.PropositionalEquality. The Agda Standard Library 2.1.1*. 2024. URL: https://agda.github.io/agda-stdlib/v2.1.1/Relation.Binary.PropositionalEquality.html (visited on 01/20/2025) (cited on page 45).

[82] Conor McBride. “A polynomial testing principle”. 2012. URL: https://personal.cis.strath.ac.uk/conor.mcbride/PolyTest.pdf (cited on page 47).

[83] Conor McBride. *W-types: good news and bad news. Epilogue for Epigram*. 2010. URL: https://mazzo.li/epilogue/index.html%3Fp=324.html (visited on 01/21/2025) (cited on page 47).

[84] Vilhelm Sjöberg and Stephanie Weirich. “Programming up to Congruence”. In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. Ed. by Sriram K. Rajamani and David Walker. ACM, 2015, pp. 369–382. DOI: 10.1145/2676726.2676974 (cited on pages 49, 51).

[85] Martin Sulzmann, Manuel M. T. Chakravarty, Simon L. Peyton Jones, and Kevin Donnelly. “System F with type equality coercions”. In: *Proceedings of TLDI’07: 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Nice, France, January 16, 2007*. Ed. by François Pottier and George C. Necula. ACM, 2007, pp. 53–66. DOI: 10.1145/1190315.1190324 (cited on page 50).

[86] Sam Lindley and Conor McBride. “Hasochism: the pleasure and pain of dependently typed haskell programming”. In: *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell, Boston, MA, USA, September 23-24, 2013*. Ed. by Chung-chieh Shan. ACM, 2013, pp. 81–92. DOI: `10.1145/2503778.2503786` (cited on pages 50, 82).

[87] Richard A. Eisenberg. “Stitch: the sound type-indexed type checker (functional pearl)”. In: *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2020, Virtual Event, USA, August 7, 2020*. Ed. by Tom Schrijvers. ACM, 2020, pp. 39–53. DOI: `10.1145/3406088.3409015` (cited on pages 50, 82).

[88] Csongor Kiss, Tony Field, Susan Eisenbach, and Simon Peyton Jones. “Higher-order type-level programming in Haskell”. In: *Proc. ACM Program. Lang.* 3.ICFP (2019), 102:1–102:26. DOI: `10.1145/3341706` (cited on page 50).

[89] 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*. Ed. by Nicola Olivetti and Ashish Tiwari. Vol. 9706. Lecture Notes in Computer Science. Springer, 2016, pp. 99–115. DOI: `10.1007/978-3-319-40229-1_8` (cited on page 51).

[90] Olle Fredriksson. *Sixty*. 2019. URL: `https://github.com/ollef/sixty` (visited on 01/21/2025) (cited on page 51).

[91] Anja Petković Komel. “Meta-analysis of type theories with an application to the design of formal proofs”. PhD thesis. University of Ljubljana, 2021. URL: `https://anjapetkovic.com/img/doctoralThesis.pdf` (cited on page 51).

[92] Th’eo Winterhalter. “Controlling computation in type theory, locally”. In: *Presented at the EuroProofNet WG6 meeting 2025*. 2025. URL: `https://theowinterhalter.github.io/res/slides/local-comp-wg6-25.pdf` (cited on page 51).

[93] Daniel Gratzer, Jonathan Sterling, Carlo Angiuli, Thierry Coquand, and Lars Birkedal. “Controlling unfolding in type theory”. In: *CoRR* abs/2210.05420 (2022). DOI: `10.48550/ARXIV.2210.05420` (cited on pages 51, 79).

[94] Jesper Cockx, Nicolas Tabareau, and Théo Winterhalter. “The taming of the rew: a type theory with computational assumptions”. In: *Proc. ACM Program. Lang.* 5.POPL (2021), pp. 1–29. DOI: `10.1145/3434341` (cited on pages 51, 78).

[95] Guillaume Genestier. “SizeChangeTool: A Termination Checker for Rewriting Dependent Types”. In: *Joint Proceedings of HOR 2019 and IWC 2019*. Ed. by Mauricio Ayala-Rincón, Silvia Ghilezan, and Jakob Grue Simonsen. Joint Proceedings of HOR 2019 and IWC 2019. Dortmund, Germany, 2019, pp. 14–19. URL: `https://hal.science/hal-02442465v1/file/presentationSCT.pdf` (cited on page 51).

[96] Richard A Eisenberg. *System FC, as implemented in GHC*. Tech. rep. University of Pennsylvania, 2015. URL: `https://repository.brynmawr.edu/cgi/viewcontent.cgi?article=1015&context=compsci_pubs` (cited on page 51).

[97] Edwin Brady. *Yaffle: A New Core for Idris 2. Presented at the Workshop on the Implementation of Type Systems (WITS) 2024*. 2024. URL: `https://www.youtube.com/watch?v=_ApsEm2t6UY` (cited on page 51).

[98] Sebastian Ullrich. “An Extensible Theorem Proving Frontend”. PhD thesis. Karlsruhe Institute of Technology, Germany, 2023. DOI: `10.5445/IR/1000161074` (cited on page 51).

[99] Jesper Cockx. *Agda Core: The Dream and the Reality*. 2024. URL: `https://jesper.cx/posts/agda-core.html` (visited on 01/21/2024) (cited on page 51).

[100] Valentin Blot, Gilles Dowek, Thomas Traversié, and Théo Winterhalter. “From Rewrite Rules to Axioms in the $\lambda\Pi$-Calculus Modulo Theory”. In: *Foundations of Software Science and Computation Structures - 27th International Conference, FoSSaCS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part II*. Ed. by Naoki Kobayashi and James Worrell. Vol. 14575. Lecture Notes in Computer Science. Springer, 2024, pp. 3–23. DOI: `10.1007/978-3-031-57231-9_1` (cited on page 52).

[101] Daniel J. Dougherty and Ramesh Subrahmanyam. “Equality between Functionals in the Presence of Coproducts”. In: *Inf. Comput.* 157.1-2 (2000), pp. 52–83. DOI: `10.1006/INCO.1999.2833` (cited on pages 52, 54).

[102] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. “Observational equality, now!” In: *Proceedings of the ACM Workshop Programming Languages meets Program Verification, PLPV 2007, Freiburg, Germany, October 5, 2007*. Ed. by Aaron Stump and Hongwei Xi. ACM, 2007, pp. 57–68. DOI: `10.1145/1292597.1292608` (cited on page 53).

[103] András Kovács. *Strong eta-rules for functions on sum types. Proof Assistants StackExchange Answer.* 2022. URL: `https://proofassistants.stackexchange.com/a/1886` (visited on 06/06/2025) (cited on pages 53, 80).

[104] Thorsten Altenkirch and Tarmo Uustalu. "Normalization by evaluation for $\lambda^{\to 2}$". In: *Functional and Logic Programming, 7th International Symposium, FLOPS 2004, Nara, Japan, April 7-9, 2004, Proceedings.* Ed. by Yukiyoshi Kameyama and Peter J. Stuckey. Vol. 2998. Lecture Notes in Computer Science. Springer, 2004, pp. 260–275. DOI: `10.1007/978-3-540-24754-8_19` (cited on page 54).

[105] Emily Riehl and Michael Shulman. "A type theory for synthetic ∞-categories". In: *Higher Structures* 1 (1 2017), pp. 147–224. DOI: `10.21136/hs.2017.06` (cited on page 54).

[106] Tesla Zhang. "Three non-cubical applications of extension types". In: *CoRR* abs/2311.05658 (2023). DOI: `10.48550/ARXIV.2311.05658` (cited on page 54).

[107] The Agda Team. *Cubical. The Agda 2.7.0.1 User Manual.* 2024. URL: `https://agda.readthedocs.io/en/v2.7.0.1/language/cubical.html` (visited on 05/09/2025) (cited on page 54).

[108] William W. Tait. "Intensional Interpretations of Functionals of Finite Type I". In: *J. Symb. Log.* 32.2 (1967), pp. 198–212. DOI: `10.2307/2271658` (cited on page 66).

[109] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types.* Cambridge Tracts in Theoretical Computer Science 7. Cambridge University Press, 1989 (cited on page 66).

[110] András Kovács. *StrongNorm.agda. GitHub Gist.* 2020. URL: `https://github.com/AndrasKovacs/misc-stuff/blob/master/agda/STLCStrongNorm/StrongNorm.agda` (visited on 01/16/2025) (cited on page 66).

[111] Jonathan Sterling and Carlo Angiuli. "Normalization for Cubical Type Theory". In: *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021.* IEEE, 2021, pp. 1–15. DOI: `10.1109/LICS52264.2021.9470719` (cited on page 79).

[112] Guillaume Allais, Conor McBride, and Pierre Boutillier. "New equations for neutral terms: a sound and complete decision procedure, formalized". In: *Proceedings of the 2013 ACM SIGPLAN workshop on Dependently-typed programming, DTP@ICFP 2013, Boston, Massachusetts, USA, September 24, 2013.* Ed. by Stephanie Weirich. ACM, 2013, pp. 13–24. DOI: `10.1145/2502409.2502411` (cited on page 80).

[113] Thierry Coquand. "An Algorithm for Type-Checking Dependent Types". In: *Sci. Comput. Program.* 26.1-3 (1996), pp. 167–177. DOI: `10.1016/0167-6423(95)00021-6` (cited on pages 82, 100).

[114] The Agda Team. *Lambda Abstraction. The Agda 2.7.0.1 User Manual.* 2024. URL: `https://agda.readthedocs.io/en/v2.7.0.1/language/lambda-abstraction.html` (visited on 06/10/2025) (cited on page 86).

[115] Jana Dunfield and Neel Krishnaswami. "Bidirectional Typing". In: *ACM Comput. Surv.* 54.5 (2022), 98:1–98:38. DOI: `10.1145/3450952` (cited on page 100).

[116] The Agda Team. *Syntactic Sugar. The Agda 2.7.0.1 User Manual.* 2024. URL: `https://agda.readthedocs.io/en/v2.7.0.1/language/syntactic-sugar.html` (visited on 06/11/2025) (cited on page 100).

[117] Andreas Abel and Thorsten Altenkirch. "A predicative analysis of structural recursion". In: *J. Funct. Program.* 12.1 (2002), pp. 1–41. DOI: `10.1017/S0956796801004191` (cited on page 104).

[118] Gilles Barthe, Benjamin Grégoire, and Fernando Pastawski. "$\widehat{\text{CIC}}$: Type-Based Termination of Recursive Definitions in the Calculus of Inductive Constructions". In: *Logic for Programming, Artificial Intelligence, and Reasoning, 13th International Conference, LPAR 2006, Phnom Penh, Cambodia, November 13-17, 2006, Proceedings.* Ed. by Miki Hermann and Andrei Voronkov. Vol. 4246. Lecture Notes in Computer Science. Springer, 2006, pp. 257–271. DOI: `10.1007/11916277\_18` (cited on page 104).

[119] Kanstantin Nisht. "Type-Based Termination Checking in Agda". MA thesis. 2024. URL: `https://knisht.github.io/agda/msc.pdf` (cited on page 104).

[120] Healfdene Goguen, Conor McBride, and James McKinna. "Eliminating Dependent Pattern Matching". In: *Algebra, Meaning, and Computation, Essays Dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday.* Ed. by Kokichi Futatsugi, Jean-Pierre Jouannaud, and José Meseguer. Vol. 4060. Lecture Notes in Computer Science. Springer, 2006, pp. 521–540. DOI: `10.1007/11780274\_27` (cited on page 104).

[121] Jesper Cockx and Andreas Abel. "Elaborating dependent (co)pattern matching". In: *Proc. ACM Program. Lang.* 2.ICFP (2018), 75:1–75:30. DOI: `10.1145/3236770` (cited on page 104).

[122] James Chapman. "Type Theory Should Eat Itself". In: *Proceedings of the International Workshop on Logical Frameworks and Metalanguages: Theory and Practice, LFMTP@LICS 2008, Pittsburgh, PA, USA, June 23, 2008.* Ed. by Andreas Abel and Christian Urban. Vol. 228. Electronic Notes in Theoretical Computer Science. Elsevier, 2008, pp. 21–36. DOI: `10.1016/J.ENTCS.2008.12.114` (cited on page 104).