Decidability of typechecking in a dependently-typed programming language with sigma types

Wojciech J. Nawrocki
St Edmund's College

Supervisor: Dr. Neel Krishnaswami

UNIVERSITY OF CAMBRIDGE

A dissertation submitted to the University of Cambridge
in partial fulfilment of the requirements for the degree of
Master of Philosophy in Advanced Computer Science

University of Cambridge
Computer Laboratory
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom

Email: wjn26@cam.ac.uk

June 17, 2020

Declaration

I, Wojciech J. Nawrocki of St Edmund's College, being a candidate for the M.Phil in Advanced Computer Science, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed:

Date:

Abstract

Typechecking algorithms play an important role in bridging the gap between the theory and implementation of dependent types. A metatheoretic property which has historically been found to enable practical proof assistants is decidability of judgmental equality. Although it is widely believed that most mainstream theories such as the Calculus of Inductive Constructions enjoy decidable conversion, this is rarely proved beyond a sketch and almost never formally due to how heavyweight the proofs tend to be. Recently, Abel, Öhman and Vezzosi presented an efficient conversion checking algorithm based on weak-head normalisation together with a decidability proof using Kripke logical relations. The proof is mechanised in Agda, bootstrapping one type theory from another. While their object theory is already fairly extensive, it misses two important constructs – the terminal/unit type and dependent sum types. In this project, I add those and extend the algorithm as well as the formal proof to establish decidability for the new theory. The resulting language includes η-extensional Π and Σ types, the type of natural numbers, the unit type, the empty type and a Russell-style universe. The project is an exercise both in semantics of type theory and in working with an existing proof codebase. The proof methods used have relevance even for theories which do not enjoy decidability, because the result sits atop a mountain of widely applicable abstraction.

Total word count: 10,897

Contents

1 Introduction

1.1 Moving target

The original goal of this project was to investigate the metatheory of XTT [43, 44], a new type theory due to Sterling, Angiuli and Gratzer. In particular, to prove that type conversion in XTT is decidable, which the authors conjecture but don't establish. I discuss why decidability is useful in 1.2, but why for XTT specifically? XTT takes ideas from the worlds of Homotopy Type Theory [47], its cubical variants [17], as well as the focus on *observations* of Observational Type Theory [11, 13] and applies them back to the older ideal of computationally well-behaved, extensional identity types [32]. It achieves this in an arguably more foundational manner than other work – function extensionality is a consequence rather than a postulate of their setup.

We decided to carry out the proof by extending an Agda codebase [8] due to Abel, Öhman and Vezzosi (*AÖV*). While these kinds of proofs have historically been massive, AÖV take several steps towards decreasing the workload. Alas, there turned out not to be a clear path to cubical metatheory using our chosen tools in the available timeframe of 6 months or so. So, the target switched to Observational Type Theory, which is closer to being expressible as an extension of the theory that AÖV formalised. But even that proved to be so much work that I only managed to formalise a small step on the road to a type conversion algorithm for OTT – the unit type and dependent sums. The final contribution, despite being rather modest, required over 4,000 new lines of Agda code to express. I call the present type theory λ^{Σ} because it includes Σ types, among other things.

1.2 Do we even want decidability?

A valid counterargument to insisting that typechecking should be decidable is that in doing so we compromise on the expressivity of a type theory. By this I don't mean its proof theoretic strength but rather how easy it is to write down certain kinds of statements. The canonical example of a type-theoretic feature which breaks decidability is equality reflection. As the name suggests, it allows reflecting witnesses of identity of types into judgmental equalities:

$$\frac{\Gamma \vdash P : t =_A u}{\Gamma \vdash t \equiv u : A}$$

In principle, this is quite nice as users can convince the typechecker that two types are convertible by proving it. For example, since one can prove that $x, y : \mathbb{N} \vdash x + y =_{\mathbb{N}} y + x$, the typechecker has to recognize this and understand that, for example, $x, y : \mathbb{N} \vdash \mathsf{vec}\, \mathbb{N}\, (x+y) \equiv \mathsf{vec}\, \mathbb{N}\, (y+x)$, i.e. the type of lists of natural numbers of length $x+y$ and ditto of length $y+x$ are, in fact, the same type. This enables a user to freely exchange between them and use a term of type $\mathsf{vec}\, \mathbb{N}\, (x+y)$ where an element of the other type is expected. Of course, precisely because the typechecker must decide which identities are provable, this makes the algorithm undecidable. When our dependently-typed program is taking a long time to check, is the compiler just slow or is it trying to prove the Riemann hypothesis?

So for practical reasons, we insist on decidability. But the proof of decidability is relevant even to theories in which it doesn't hold because in developing it we build up a whole bunch of useful metatheory which will be discussed shortly.

1.3 Contributions

The primary contribution is a fully mechanised proof of decidability of type conversion for λ^{Σ}, achieved by extending the type theory and proof due to AÖV [8] with dependent sum types and a unit type. The contribution counts out to `71 files changed, 5297 insertions(+), 1323 deletions(-)` out of a total 15750 lines of code *after* the changes. The full Agda development is available at `https://github.com/Vtec234/logrel-mltt/tree/1290bd9eb6804a41b2fb1214c4e2f6a6cbf858ef` as well as the zipped source code archive.

A secondary contribution is this document. Since a detailed walkthrough of the proof is already available in the work of AÖV, instead of repeating it I attempt to intuitively explain the high-level structure of the proof. I also discuss some design choices in the formalisation as well as the principles behind them, explicitly writing down a number of folklore observations.

In chapter 2, I give an overview of how Kripke logical relations can be used to give meaning to the syntactic constructs of a formal system by considering strong normalization and decidability of $\beta\eta$-equivalence in the simply-typed lambda calculus. In chapter 3 I summarize the structure of AÖV's proof and some of their technical innovations. In chapter 3.1, I describe the addition of a contractible Unit type and how it propagates through the proof. In chapter 3.2, I do the same for Σ-types. Finally, I conclude. The full definition of, and conversion algorithm for, λ^{Σ} are in the appendix.

2 The logical relations model

To see how the semantics of dependent types which we are going to need for the proof is constructed, let us first consider some properties of a far simpler language, the simply-typed λ-calculus (*STLC*) with, say, a single base type $\mathbb{1}$ containing one constant, $\overline{\Gamma \vdash * : \mathbb{1}}$. STLC is generally well-behaved, and terms can easily be checked to have a certain type. Here's one way to do it in Agda:

```
_⊢_⇐_ : (Γ : Ctx) (t : Term) (A : Type) → Dec (Γ ⊢ t ∷ A)
Γ ⊢ star  ⇐ Unit    = yes starⱼ
Γ ⊢ star  ⇐ F ▹ G   = no λ ()
Γ ⊢ lam t ⇐ F ▹ G
  with Γ · F ⊢ t ⇐ G
... | yes P         = yes (lamⱼ P)
... | no ¬P         = no λ { (lamⱼ P) → ¬P P }
Γ ⊢ lam t ⇐ Unit    = no λ ()
Γ ⊢ var n ⇐ A
  with isVarInCtx n A Γ
... | yes P         = yes (varⱼ P)
... | no ¬P         = no λ { (varⱼ P) → ¬P P }
-- Applications are annotated with the argument type to avoid type inference.
Γ ⊢ [ F ] fn ∘ arg ⇐ G
  with Γ ⊢ fn ⇐ F ▹ G | Γ ⊢ arg ⇐ F
... | yes P | yes Q = yes (P ∘ⱼ Q)
... | no ¬P | _     = no λ { (P ∘ⱼ _) → ¬P P }
... | _     | no ¬P = no λ { (_ ∘ⱼ P) → ¬P P }
```

STLC is also strongly normalizing, meaning that the process of applying β-reduction steps to a term is eventually going to terminate with a normal form which cannot be further reduced. This, however, is trickier to prove. Pierce [36, ch. 12] explains the problem well, as does Crary [37, ch. 6] for the case of when we not only want to evaluate STLC terms but also compare them for equivalence. The family of proof techniques developed in response to this issue is known as either Tait's *computability* method [45], or Girard's *reducibility candidates* [27], or as *logical relations* [38] [39]. I will briefly summarize the general idea – for a full proof using this strategy, see for example POPLMark Reloaded [9].

Suppose we try to prove termination via induction on typing derivations. Writing the goal as "for all Γ, t, A, if $\Gamma \vdash t : A$ then t `terminates`", consider the application case:

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash e : A}{\Gamma \vdash f\ e : B}$$

By the induction hypothesis, we have that f `terminates` and e `terminates`, but how are we to obtain $f\ e$ `terminates`? The substitution of e into the body of f may well enable further reduction, so the goal doesn't follow. Other attempts at simple structural induction are going to run into roughly the same problem. Fundamentally, this is because function application can make terms arbitrarily larger *syntactically*. For example, $(\lambda f : \mathbb{1} \to \mathbb{1}.\ f\ (f\ *))$ `hugeTerm` reduces to `hugeTerm` (`hugeTerm` $*$), growing the overall expression in size.

To carry out the proof, we need a stronger induction hypothesis. We do this by constructing, for each type A, the set $[\![A]\!]$ of terminating terms of that type. Crucially, this construction shall be *logical* in the sense that elements of $[\![A]\!]$ (which we call *reducible*) must respect elimination forms for the type A. In the current variant of STLC, the only elimination form is function application. Then, we show by induction on types or typing derivations that each well-typed term $t : A$ is in $[\![A]\!]$ and therefore terminating. For each type A, $[\![A]\!]$ is a logical predicate (a unary logical relation):

$$\boxed{[\![A]\!]}$$

$$\begin{aligned}
[\![\mathbb{1}]\!] &= \{\, s \in \mathit{Term} \mid \diamond \vdash s : \mathbb{1}\ \wedge\ s\ \texttt{terminates} \,\} \\
[\![A \to B]\!] &= \{\, f \in \mathit{Term} \mid \diamond \vdash f : A \to B\ \wedge\ f\ \texttt{terminates}\ \wedge \\
&\qquad \forall e \in [\![A]\!].\ f\ e \in [\![B]\!] \,\}
\end{aligned}$$

The above isn't quite enough because it only handles closed terms ($\diamond$ is the empty context), but it already illustrates a couple of important points. As promised, the relation is logical in that reducible functions take reducible arguments to reducible values, so the elimination form for functions – function application – is respected. In fact, if we fix type theory as the metatheory, the relation $[\![A \to B]\!]$ corresponding to a function type is itself a type of functions under the Brouwer-Heyting-Kolmogorov interpretation of universal quantification. We have thus built a kind of denotational semantics. This is an important observation which carries over all the way up the lambda cube to dependent types. It also justifies referring to $[\![A]\!]$ as the *meaning* or *semantics* of A. In the present work (ch. 3.2), a reducible dependent pair is going to be exactly a dependent pair of reducible terms together with a whole bunch of extra predicates.

Moreover, observe that the relation is defined by structural recursion on STLC types since $[\![A \to B]\!]$ refers only to $[\![A]\!]$, $[\![B]\!]$. We therefore don't need to justify its well-foundedness beyond the validity of large elimination in the metatheory. This is notably not going to be the case in the dependent setting.

To complete the definition, we should handle variable contexts and make weakening of the relation admissible. The canonical solution is to make it *Kripke*, i.e. monotonic in the context extension relation (counterintuitively s.t. $\Gamma, A \leq \Gamma$). We only need to enforce monotonicity at function types – the rules for $\mathbb{1}$ don't bind variables, so $[\![\mathbb{1}]\!]$ can already be weakened. We define a logical predicate for each context and type:

$$\boxed{\Gamma \Vdash t \in [\![A]\!]}$$

$$\begin{aligned}
\Gamma \Vdash s \in [\![\mathbb{1}]\!] &\Leftrightarrow \Gamma \vdash s : \mathbb{1} \ \land\ s \ \texttt{terminates} \\
\Gamma \Vdash f \in [\![A \to B]\!] &\Leftrightarrow \Gamma \vdash f : A \to B \ \land\ f \ \texttt{terminates} \land \\
&\qquad \forall e \in \mathit{Term}.\ \forall \Gamma' \leq \Gamma.\ \Gamma' \Vdash e \in [\![A]\!] \Rightarrow \Gamma' \Vdash f\ e \in [\![B]\!]
\end{aligned}$$

Figure 2.1: Logical predicate for normalization of STLC

Now, since terms in the relation are terminating by definition, to show termination of well-typed terms it suffices to show completeness of the relation w.r.t. our typing rules, i.e. that $\Gamma \vdash t : A$ implies $\Gamma \Vdash t \in [\![A]\!]$. This statement often bears the name of *Fundamental Theorem* (of logical relations). Proving the Fundamental Theorem for dependent typing judgments entailed the vast majority of work in the present project.

2.1 Equational theory

With strong normalization in hand, we can develop the theory further. While comparing programs in a Turing-complete language for equivalence is undecidable, in STLC we can do it by simply evaluating the programs and comparing the results. Indeed, driving terms down to well-behaved normal forms and then comparing those is the key idea behind many decision procedures for equational theories. For example, the `ring` tactic [29] in Coq handles the theory of commutative semirings.

We could thus design an algorithm to decide a simple β-equivalence which equates STLC terms that evaluate to the same value. However this notion of equivalence is not very useful. What we really want is something to group terms into *observationally* equivalent classes, that is into classes whose members exhibit the same computational behaviour. For example, $(\lambda x.\ f\ x) : A \to B$ and $f : A \to B$ are extensionally the same function, but are not β-equivalent. A better notion is then one of $\beta\eta$-equivalence, given by the reflexive-symmetric-transitive closure of:

$$\textsc{fun-}\beta\ \frac{\Gamma, A \vdash t : B \qquad \Gamma \vdash e : A}{\Gamma \vdash (\lambda x.\ t)\ e \equiv t[e/x] : B} \qquad \textsc{fun-}\eta\ \frac{\Gamma \vdash f : A \to B}{\Gamma \vdash (\lambda x.\ f\ x) \equiv f : A \to B} \qquad \textsc{unit-}\eta\ \frac{\Gamma \vdash s : \mathbb{1}}{\Gamma \vdash s \equiv * : \mathbb{1}}$$

$$\textsc{abs-cong}\ \frac{\Gamma, x : A \vdash t \equiv u : B}{\Gamma \vdash (\lambda x.\ t) \equiv (\lambda x.\ u) : A \to B} \qquad \textsc{app-cong}\ \frac{\Gamma \vdash f \equiv g : A \to B \qquad \Gamma \vdash t \equiv u : A}{\Gamma \vdash f\ t \equiv g\ u : B}$$

Since β-reduction is not a normalization procedure w.r.t. this equivalence, other strategies have been developed. Prominent in the literature is Normalization by Evaluation (*NbE*) [15]. Roughly speaking, NbE works by repeatedly reflecting terms into their semantic correspondents and then reifying them back into the object language. Its operation is described well in Abel's habilitation thesis [1]. Unfortunately as an algorithm for deciding $\beta\eta$-equivalence, as well as type conversion in the dependent setting, NbE is wasteful. To see why, it suffices to know that NbE normalizes all subterms of a term in order to bring it to an η-long normal form. Consider STLC with an inductive type of natural numbers:

$$\overset{\textsc{zero}}{\frac{}{\Gamma \vdash 0 : \mathbb{N}}} \qquad \overset{\textsc{suc}}{\frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathsf{suc}\ n : \mathbb{N}}} \qquad \overset{\textsc{rec}}{\frac{\Gamma \vdash z : A \qquad \Gamma \vdash s : \mathbb{N} \to A \to A \qquad \Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathsf{rec}\ z\ s\ n : A}}$$

And a sufficiently expanded notion of equivalence:

$$\overset{\textsc{rec-zero}}{\frac{\Gamma \vdash z : A \qquad \Gamma \vdash s : \mathbb{N} \to A \to A}{\Gamma \vdash \mathsf{rec}\ z\ s\ 0 \equiv z : A}} \qquad \overset{\textsc{rec-suc}}{\frac{\Gamma \vdash z : A \qquad \Gamma \vdash s : \mathbb{N} \to A \to A \qquad \Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathsf{rec}\ z\ s\ (\mathsf{suc}\ n) \equiv s\ n\ (\mathsf{rec}\ z\ s\ n) : A}}$$

$$\overset{\textsc{suc-cong}}{\frac{\Gamma \vdash n \equiv m : \mathbb{N}}{\Gamma \vdash \mathsf{suc}\ n \equiv \mathsf{suc}\ m : \mathbb{N}}}$$

$$\overset{\textsc{rec-cong}}{\frac{\Gamma \vdash z_1 \equiv z_2 : A \qquad \Gamma \vdash s_1 \equiv s_2 : \mathbb{N} \to A \to A \qquad \Gamma \vdash n_1 \equiv n_2 : \mathbb{N}}{\Gamma \vdash \mathsf{rec}\ z_1\ s_1\ n_1 \equiv \mathsf{rec}\ z_2\ s_2\ n_2 : A}}$$

In this language, it is quite clear that $\mathsf{suc}\ t \not\equiv 0 : \mathbb{N}$ for all t. In type theory this is sometimes referred to as the *no-confusion* principle – distinct constructors cannot produce equivalent terms. An NbE-based decision procedure, however, will not exploit this observation when faced with the task of comparing $\mathsf{suc}\ \mathtt{hugeTerm}$ and 0. It will instead go on to normalize $\mathtt{hugeTerm}$ leading to major inefficiency. An alternative, useful especially for conversion in dependent type theory [18], is to use a lazier, *weak-head* form of reduction and compare terms in weak-head normal form (*WHNF*).

A term in WHNF ($\mathtt{Whnf}$) is either *neutral* ($\mathtt{Ne}$) – inert because a redex subterm is blocked by a variable – or an introduction form – such as an inductive type constructor – in head position with arbitrary terms which needn't be in WHNF as arguments. While the WHNFs of λ^{Σ} are characterized in the appendix, fig. 5.1, the following grammar characterizes WHNFs and neutral terms for STLC with natural numbers:

$$\begin{array}{lll} \mathtt{Term} \ni t, u, v & ::= & \bar{t} \mid t\ u \mid \mathsf{rec}\ t\ u\ v \\ \mathtt{Whnf} \ni \bar{t} & ::= & n \mid \lambda x.\ t \mid * \mid \mathsf{zero} \mid \mathsf{suc}\ t \\ \mathtt{Ne} \ \ni n & ::= & x \mid n\ t \mid \mathsf{rec}\ t\ u\ n \end{array}$$

Since both $\mathsf{suc}\ \mathtt{hugeTerm}$ and 0 are WHNFs, the algorithm can immediately compare their head constructors (suc and 0) and quickly decide the inequality.

One important consideration remains. Unlike NbE, weak-head reduction will not expand terms into η-long form, so the equal pair $(\lambda x.\ f\ x)$ and f looks different to any algorithm which only compares terms in WHNF syntactically (up to α-equivalence). To regain extensionality, we could extend weak-head reduction with an η-reduction rule: "$(\lambda x.\ f\ x) \rightsquigarrow f$ if x is not free in f". But then properties like "a term in WHNF does not reduce" would not hold since an abstraction is weak-head normal already, so we would need a more complicated notion of weak-head normality. A better alternative which simplifies the presentation is to modify the equivalence rules to be more *algorithm-friendly*. To do so, replace both $\textsc{fun-}\eta$ and

ABS-CONG with a single rule:

$$\frac{\Gamma, x : A \vdash f\,x \equiv g\,x : B}{\Gamma \vdash f \equiv g : A \to B}\ \textsc{fun-ext}$$

The reader can verify that, assuming inversion ($\Gamma \vdash t \equiv u : A$ implies $\Gamma \vdash t : A$ and $\Gamma \vdash u : A$), this rule implies and is implied by (FUN-η $\land$ ABS-CONG). It also directly tells us what to do, in a *type-directed* manner. When two terms of function type are to be compared for equivalence, simply apply both to a fresh variable and compare the outputs. The extensionality principle is made clear – two functions are equal if they behave the same on all inputs. More generally, two terms are equal if they look the same from all external perspectives, that is if any observations we can make about them are equal. Following this philosophy, in ch. 3.2 the equivalence rule for dependent sums is going to compare their behaviour at first and second projections.[1]. A similar modification can be made for the unit type:

$$\frac{\Gamma \vdash t : \mathbb{1} \qquad \Gamma \vdash u : \mathbb{1}}{\Gamma \vdash t \equiv u : \mathbb{1}}\ \textsc{unit-ext}$$

2.2 Decidability of $\beta\eta$-equivalence

With the right design principles in place, we can go on to write down a term conversion algorithm for STLC and outline a proof of its correctness. Here, I follow Crary's exposition [37, ch. 6], adapting it to be more in line with the target algorithm for λ^{Σ}. Any mistakes are mine. In order to set the scene for dependent sum types, we shall consider $\lambda_{\times}$ – an STLC with binary products[2] (and without natural numbers, as they introduce no novelty here).

The grammar of $\lambda_{\times}$:

$$\begin{array}{lll} \texttt{Term} \ni t, u & ::= & \bar{t} \mid t\,u \mid \mathsf{fst}\,t \mid \mathsf{snd}\,t \\ \texttt{Whnf} \ni \bar{t}, \bar{u} & ::= & n \mid \lambda x.\,t \mid * \mid \langle t, u \rangle \\ \texttt{Ne} \ni n, m & ::= & x \mid n\,t \mid \mathsf{fst}\,n \mid \mathsf{snd}\,n \end{array}$$

New typing and equivalence rules:

[1] This is something I got stuck on in the formal development, since I originally postulated the algorithm-unfriendly $p \equiv \langle \mathsf{fst}\,p, \mathsf{snd}\,p \rangle$.

[2] Which happens to complete the cartesian-closed structure, but this fact will not be used.

$\boxed{\Gamma \vdash t : A}$

$$\textsc{prod}\quad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : B}{\Gamma \vdash \langle t, u \rangle : A \times B} \qquad \textsc{fst}\quad \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \mathsf{fst}\, t : A} \qquad \textsc{snd}\quad \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \mathsf{snd}\, t : B}$$

$\boxed{\Gamma \vdash t \equiv u : A}$

$$\textsc{fst-}\beta\quad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : B}{\Gamma \vdash \mathsf{fst}\, \langle t, u \rangle \equiv t : A} \qquad \textsc{snd-}\beta\quad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : B}{\Gamma \vdash \mathsf{snd}\, \langle t, u \rangle \equiv u : B}$$

$$\textsc{prod-ext}\quad \frac{\Gamma \vdash \mathsf{fst}\, t \equiv \mathsf{fst}\, t : A \qquad \Gamma \vdash \mathsf{snd}\, t \equiv \mathsf{snd}\, t : B}{\Gamma \vdash t \equiv u : A \times B} \qquad \textsc{fst-cong}\quad \frac{\Gamma \vdash t \equiv u : A \times B}{\Gamma \vdash \mathsf{fst}\, t \equiv \mathsf{fst}\, u : A}$$

$$\textsc{snd-cong}\quad \frac{\Gamma \vdash t \equiv u : A \times B}{\Gamma \vdash \mathsf{snd}\, t \equiv \mathsf{snd}\, u : B}$$

As preordained, the extensionality principle PROD-EXT for products considers their observable behaviour by comparing the sides at both projections.

Then, the algorithm, given by the following rules and *not more* – we do not take any closures, transitive or otherwise. Its rules should be read bottom-to-top:

$\boxed{\Gamma \vdash t \mathrel{\hat{\Longleftrightarrow}} u : A}$

$$\frac{t \longrightarrow^* \bar{t} \qquad u \longrightarrow^* \bar{u} \qquad \Gamma \vdash \bar{t} \Longleftrightarrow \bar{u} : A}{\Gamma \vdash t \mathrel{\hat{\Longleftrightarrow}} u : A}$$

$\boxed{\Gamma \vdash \bar{t} \Longleftrightarrow \bar{u} : A}$

$$\frac{\Gamma \vdash n \longleftrightarrow m : A}{\Gamma \vdash n \Longleftrightarrow m : A}$$

$$\frac{\Gamma, x : A \vdash \bar{t}\, x \mathrel{\hat{\Longleftrightarrow}} \bar{u}\, x : B}{\Gamma \vdash \bar{t} \Longleftrightarrow \bar{u} : A \to B}$$

$$\frac{\Gamma \vdash \bar{t} : \mathbb{1} \qquad \Gamma \vdash \bar{u} : \mathbb{1}}{\Gamma \vdash \bar{t} \Longleftrightarrow \bar{u} : \mathbb{1}}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{fst}\, \bar{t} \mathrel{\hat{\Longleftrightarrow}} \mathsf{fst}\, \bar{u} : A \\ \Gamma \vdash \mathsf{snd}\, \bar{t} \mathrel{\hat{\Longleftrightarrow}} \mathsf{snd}\, \bar{u} : B\end{array}}{\Gamma \vdash \bar{t} \Longleftrightarrow \bar{u} : A \times B}$$

$\boxed{\Gamma \vdash n \longleftrightarrow m : A}$

$$\frac{x : A \in \Gamma}{\Gamma \vdash x \longleftrightarrow x : A}$$

$$\frac{\Gamma \vdash n \longleftrightarrow m : A \to B \qquad \Gamma \vdash t \mathrel{\hat{\Longleftrightarrow}} u : A}{\Gamma \vdash n\, t \longleftrightarrow m\, u : B}$$

$$\frac{\Gamma \vdash n \longleftrightarrow m : A \times B}{\Gamma \vdash \mathsf{fst}\, n \longleftrightarrow \mathsf{fst}\, m : A}$$

$$\frac{\Gamma \vdash n \longleftrightarrow m : A \times B}{\Gamma \vdash \mathsf{snd}\, n \longleftrightarrow \mathsf{snd}\, m : B}$$

The algorithm's entrypoint is the *algorithmic conversion* judgment $\Gamma \vdash t \mathrel{\hat{\Longleftrightarrow}} u : A$. Its only job is to weak-head reduce both terms and then invoke *weak-head algorithmic conversion* $\Gamma \vdash \bar{t} \Longleftrightarrow \bar{u} : A$.

Weak-head conversion is type-directed – application of fresh variable for functions, immediate equality for unit and comparison at projections for products. Observe that while introduction forms – $\lambda x.\, t$, $*$, $\langle t, u \rangle$ – are all WHNFs, there are no congruence or reflexivity rules for them in $\Longleftrightarrow$. This is because they are admissible as a consequence of the extensionality principles for their respective types! As seen before, at function types FUN-EXT implies both ABS-CONG and FUN-η and similar results hold for the other types. Thanks to this we keep the algorithm minimal.

If the terms to compare are neutral, weak-head conversion invokes conversion of neutral terms $\Gamma \vdash n \longleftrightarrow m : A$ which handles variables and congruence of elimination forms – function application and product projections. If neutral conversion has to compare function arguments, it invokes the top-level judgment again and *only then* are the arguments reduced. Note that since neutral terms are weak-head normal already, there is no distinct judgment like $\hat{\Longleftrightarrow}$ to reduce them.

AÖV's algorithm (extended to λ^{Σ} in 5.4) follows these principles into the dependent setting. Another pair of judgments – conversion of types $\Gamma \vdash A \hat{\Longleftrightarrow} B$ and weak-head normal types $\Gamma \vdash \bar{A} \Longleftrightarrow \bar{B}$ – is introduced which term-level rules have to be compatible with in the sense that if $\Gamma \vdash t \hat{\Longleftrightarrow} u : A$ and $\Gamma \vdash A \hat{\Longleftrightarrow} B$ then $\Gamma \vdash t \hat{\Longleftrightarrow} u : B$. Moreover, to deal with type-level operational semantics, the invocation of $\Gamma \vdash \bar{t} \Longleftrightarrow \bar{u} : \bar{A}$ by $\Gamma \vdash t \hat{\Longleftrightarrow} u : A$ also normalizes the type A. Analogous weak-head reduction has to be performed during type conversion as well as conversion of neutral terms since, while neutrals are in WHNF, their types may not be. A new $\hat{\longleftrightarrow}$ judgment is added to reduce the type of neutral terms.

Back to simple types. Correctness of the conversion algorithm is entailed by two properties:

- soundness – "$\Gamma \vdash t \hat{\Longleftrightarrow} u : A$ implies $\Gamma \vdash t \equiv u : A$";
- and completeness – "$\Gamma \vdash t \equiv u : A$ implies $\Gamma \vdash t \hat{\Longleftrightarrow} u : A$".

Unlike in the dependent case which requires a semantic argument at this point already, soundness for simple types is provable via straightforward inductive reasoning since most algorithmic rules look like their declarative counterparts.

Proving completeness is much harder. The trouble is essentially the same as in strong normalization proofs, and so it can be solved in the same way. As could be expected of something that makes a binary relation logical, the Kripke logical relation we construct is binary. It entails algorithmic conversion while respecting elimination forms:

$$\boxed{\Gamma \Vdash t \mathrel{\hat{\Longleftrightarrow}} u \in \llbracket A \rrbracket}$$

$$\begin{aligned}
\Gamma \Vdash t \mathrel{\hat{\Longleftrightarrow}} u \in \llbracket \mathbb{1} \rrbracket \text{ iff } \exists \bar{t}, \bar{u} \text{ s.t. } & t \longrightarrow^{*} \bar{t} \wedge u \longrightarrow^{*} \bar{u} \wedge \\
& \Gamma \vdash \bar{t} : \mathbb{1} \wedge \Gamma \vdash \bar{u} : \mathbb{1}
\end{aligned}$$

$$\begin{aligned}
\Gamma \Vdash f \mathrel{\hat{\Longleftrightarrow}} g \in \llbracket A \to B \rrbracket \text{ iff } \exists \bar{f}, \bar{g} \text{ s.t. } & f \longrightarrow^{*} \bar{f} \wedge g \longrightarrow^{*} \bar{g} \wedge \\
& \Gamma \vdash \bar{f} : A \to B \wedge \Gamma \vdash \bar{g} : A \to B \wedge \\
& \forall t, u \in \mathit{Term}.\ \forall \Gamma' \leq \Gamma.\ \text{if } \Gamma' \Vdash t \mathrel{\hat{\Longleftrightarrow}} u \in \llbracket A \rrbracket \\
& \qquad \text{then } \Gamma' \Vdash \bar{f}\ t \mathrel{\hat{\Longleftrightarrow}} \bar{g}\ u \in \llbracket B \rrbracket
\end{aligned}$$

$$\begin{aligned}
\Gamma \Vdash p \mathrel{\hat{\Longleftrightarrow}} r \in \llbracket A \times B \rrbracket \text{ iff } \exists \bar{p}, \bar{r} \text{ s.t. } & p \longrightarrow^{*} \bar{p} \wedge r \longrightarrow^{*} \bar{r} \wedge \\
& \Gamma \vdash \bar{p} : A \times B \wedge \Gamma \vdash \bar{r} : A \times B \wedge \\
& \Gamma \Vdash \mathtt{fst}\ \bar{p} \mathrel{\hat{\Longleftrightarrow}} \mathtt{fst}\ \bar{r} \in \llbracket A \rrbracket \wedge \\
& \Gamma \Vdash \mathsf{snd}\ \bar{p} \mathrel{\hat{\Longleftrightarrow}} \mathsf{snd}\ \bar{r} \in \llbracket B \rrbracket
\end{aligned}$$

The statement is a little complicated, so let's explore its main features. The somewhat vague `terminates` predicate from earlier is still present at every type, but now in an exact form which prescribes the existence of well-typed WHNFs to which both sides reduce. This formulation builds in weak-head normalisation and enforces weak-head expansion – "if $\Gamma \Vdash t' \mathrel{\hat{\Longleftrightarrow}} u' \in \llbracket A \rrbracket$ and $t \longrightarrow^{*} t'$ and $u \longrightarrow^{*} u'$ then $\Gamma \Vdash t \mathrel{\hat{\Longleftrightarrow}} u \in \llbracket A \rrbracket$" – by definition. As such, it is somewhat overkill compared to alternative presentations – using a slightly different algorithm, Crary [37, ch. 6] manages to avoid talking about reduction at all. However thanks to looking like it does, the above definition generalizes more or less directly to what the model of dependently-typed conversion is going to look like.

Like the predicate from fig. 2.1, our semantics models functions as functions. Moreover, since $\lambda_{\times}$ also includes products, the relation carries itself through their elimination forms. Under the Brouwer-Heyting-Kolmogorov interpretation of logic, the conjunction of "`fst` projections are reducibly equal" with "snd projections are reducibly equal" is indeed a pair, so the semantics of a pair is also a pair in the metathory.

Note also that we only require reducibility of product projections to hold under the same context rather than being necessarily monotonic like reducibility of function applications. This is sufficient since, similarly to $\mathbb{1}$, none of the rules for products bind variables, so we can prove monotonicity for them and hence for the entire relation without postulating it definitionally.

Using this definition we can prove completeness, in two parts:

1. If $\Gamma \vdash t \equiv u : A$ then $\Gamma \Vdash t \mathrel{\hat{\Longleftrightarrow}} u \in \llbracket A \rrbracket$ (fundamental theorem)
2. If $\Gamma \Vdash t \mathrel{\hat{\Longleftrightarrow}} u \in \llbracket A \rrbracket$ then $\Gamma \vdash t \mathrel{\hat{\Longleftrightarrow}} u : A$ (escape lemma)

Take note that reducible equality is implied by ordinary equivalence $t \equiv u$ but implies algorithmic conversion $t \mathrel{\hat{\Longleftrightarrow}} u$. There's a degree of asymmetry – the logical relation $\Gamma \Vdash t \mathrel{\hat{\Longleftrightarrow}} u \in \llbracket A \rrbracket$ is denoted by the relation which it *implies* rather than that which it is implied by. This will be important later.

Let us first consider escape. This is provable via structural induction on types with the double hypothesis:

- if $\Gamma \Vdash t \mathrel{\hat{\Longleftrightarrow}} u \in [\![A]\!]$ then $\Gamma \vdash t \mathrel{\hat{\Longleftrightarrow}} u : A$
- if $\Gamma \vdash n \longleftrightarrow m : A$ then $\Gamma \Vdash n \mathrel{\hat{\Longleftrightarrow}} m \in [\![A]\!]$

Where the second part is needed to generate $\Gamma, x : A \Vdash x \mathrel{\hat{\Longleftrightarrow}} x \in [\![A]\!]$ which we can then pass to the function semantics in order to get $\Gamma, x : A \Vdash f\ x \mathrel{\hat{\Longleftrightarrow}} g\ x \in [\![B]\!]$ and subsequently retrieve $\Gamma \vdash f \Longleftrightarrow g : A \to B$ via extensionality. Setting things up for this to work requires careful consideration of the relation's shape. AÖV take a different approach – simply add $\Gamma \vdash t \mathrel{\hat{\Longleftrightarrow}} u : A$ to every case of $\Gamma \Vdash t \mathrel{\hat{\Longleftrightarrow}} u \in [\![A]\!]$, making escape another property which follows by-definition.

As for the fundamental theorem, it is almost provable by induction on equivalence derivations thanks to the generality provided by our logical relation. Alas, a number of technicalities remain.

First, since the conversion algorithm avoids non-deterministic rules such as transitivity (otherwise, how would it come up with u in $t \equiv u \land u \equiv v$ when checking $t \equiv v$?) and symmetry, we must verify that conversion is a valid equivalence relation a posteriori. This is not too hard, but it forces the design of the algorithm to be obviously transitive and symmetric. That is, these properties must follow by syntactic rather than semantic arguments since we need them to build semantics in the first place. This isn't easily doable in the dependent case, so something different is going to happen.

Second, besides closure under elimination forms, our relation must also be closed under substitution. Otherwise, there is no way to validate equalities that use substitution – in $\lambda_{\times}$, that's reduction of abstractions (FUN-β) given by $(\lambda x.\ t)\ e \equiv t[e/x]$. The canonical solution is to introduce a notion of *reducibly equal substitutions* and close reducible equality under those.

Two simultaneous substitutions $\sigma, \sigma' : \Delta \to \Gamma$ are *reducibly equal*, denoted by $\Delta \Vdash^{s} \sigma \mathrel{\hat{\Longleftrightarrow}} \sigma' : \Gamma$, iff for all $x : A \in \Gamma$ we have that $\Delta \Vdash \sigma(x) \mathrel{\hat{\Longleftrightarrow}} \sigma'(x) \in [\![A]\!]$.

Reducible equality that's also closed under reducibly equal substitutions is then packaged by AÖV into a single *validity* judgment.

Two terms t, u of type A are in the validity relation, denoted by $\Gamma \Vdash^{v} t \mathrel{\hat{\Longleftrightarrow}} u \in [\![A]\!]$, iff for all simultaneous substitutions $\sigma, \sigma' : \Delta \to \Gamma$, if $\Delta \Vdash^{s} \sigma \mathrel{\hat{\Longleftrightarrow}} \sigma' : \Gamma$ then $\Delta \Vdash t[\sigma] \mathrel{\hat{\Longleftrightarrow}} u[\sigma'] \in [\![A]\!]$.

With these tools, we can prove an adjusted fundamental theorem:

- If $\Gamma \vdash t \equiv u : A$ then $\Gamma \Vdash^{v} t \mathrel{\hat{\Longleftrightarrow}} u \in [\![A]\!]$

and correctness follows.

Finally, to show decidability of equivalence we exhibit a concrete decision procedure in a constructive metatheory, for example by writing a function of type `Γ ⊢ t <≙> t ∷ A → Γ ⊢ u <≙> u ∷ A → Dec (Γ ⊢ t <≙> u ∷ A)` in Agda and transporting the result to `Γ ⊢ t ∷ A → Γ ⊢ u ∷ A → Dec (Γ ⊢ t ≡ u ∷ A)` using correctness.

2.3 Adding dependency

So far we have seen that $\beta\eta$-equivalence of STLC terms can be efficiently decided with a type-directed algorithm using weak-head reduction. The correctness of this algorithm can be proven using logical relation semantics.

The situation with dependent type theory is analogous in a sense, yet tremendously more complicated in practice. To recap, a Martin-Löf style type theory can be given by four judgments:

- $\Gamma \vdash A$ “A is a type in context Γ”
- $\Gamma \vdash A = B$ “A and B are equal types in context Γ”
- $\Gamma \vdash t : A$ “t is a term of type A in context Γ”
- $\Gamma \vdash t = u : A$ “t and u are equal terms of type A in context Γ”

These judgments refer to each other arbitrarily via rules such as conversion (“if $\Gamma \vdash A \equiv B$ and $\Gamma \vdash t : A$ then $\Gamma \vdash t : B$”), so their statements are mutually recursive. This stands in contrast to the STLC in which equivalence rules depend on typing but not vice-versa. We were able to exploit this fact and model algorithmic conversion with a single relation, $\Gamma \Vdash t \mathrel{\hat{\Longleftrightarrow}} u \in [\![A]\!]$. Unfortunately we cannot do that here – what’s necessary instead is to construct a logical relation for each of the type-theoretic judgments $\Gamma \vdash \mathfrak{J}$. The relations will be denoted by $\Gamma \Vdash \mathfrak{J}$ and the substitution validity wrappers by $\Gamma \Vdash^{v} \mathfrak{J}$.

Just stating this construction can be difficult. Crary and Harper remark that in their work on polymorphic and recursive types [20], establishing the relation’s existence was the most difficult step, making all other proofs comparatively easy. Consider what (non-Kripke) reducibility might look like for dependent functions:

$$\Gamma \Vdash f : \Pi A \triangleright B \text{ iff } \Gamma \vdash f : \Pi A \triangleright B \ \wedge\ f \ \texttt{terminates} \ \wedge\ \forall t \in \mathit{Term}. \text{ if } \Gamma \Vdash t : A \text{ then } \Gamma \Vdash f\ t : B[t]$$

Well-foundedness of the above definition is in trouble because reducibility at $\Pi A \triangleright B$ refers to reducibility at B with t substituted into it, which is a larger term! This is precisely the ghost of function application from earlier coming back to haunt us at the type level. One way to resolve the conflict is to justify the relation’s existence as the least fixpoint of a monotone operator on a complete pointed partial order (a lattice, more or less) of semantic objects. Harper [30] carries out this construction for a variant of Martin-Löf’s type theory [33] with Π/Σ/identity types and a cumulative hierarchy of universes.

An alternative, arguably better approach taken by AÖV is to use a metatheory in which this kind of construction is justified a priori. To achieve this, they employ Agda’s support for induction-recursion [22] to define the logical relations by induction on types and recursion on *reducibility derivations*. While alleviating the need for an elaborate fixpoint construction, the scheme does not come without its own issues. For one, as AÖV observe, the gap in proof-theoretic strength between the object theory and the inductive-recursive metatheory is perhaps wider than one might like [23]. This isn’t so much a disadvantage over Harper’s work since the metatheory there is also quite strong, but one might imagine that the semantics required for decidability could be expressible in weaker systems, too. More immediately, indexing by reducibility derivations makes for one of the worst pain points when working with the formal codebase.

The Kripke relations split as follows (ignoring universes for now). For typehood, we still have $\Gamma \Vdash A$, read as “A is a reducible type in context Γ”. For other judgments, let $[A]$ be a derivation of $\Gamma \Vdash A$. We then have:

- $\Gamma \Vdash A = B \,/\, [A]$ “given $[A]$, A and B are reducibly equal types in context Γ”

- $\Gamma \Vdash t : A \,/\, [A]$ "given $[A]$, t is a reducible term of type A in context Γ"
- $\Gamma \Vdash t = u : A \,/\, [A]$ "given $[A]$, t and u are reducibly equal terms of type A in context Γ"

AÖV immediately prove irrelevance lemmas which state that these relations are really the same regardless of which derivation $[A]$ is. Still, in a theorem prover one finds a need to apply irrelevance rather often. As such, the definition is one of many causes of the proofs' verbosity. Under the hood, this is yet another instance of the dependent indexing problem hinted at in ch. 1.2. We hope that a solution to this particular case that doesn't require radically redefining the relation should be possible.

One idea would be to develop tactic-based tools which apply irrelevance automatically. Unfortunately the rudimentary state of metaprogramming in Agda makes this somewhat difficult. Another would be to work with existential types $\sum_{[A]} . \Gamma \Vdash \mathfrak{J} \,/\, [A]$, however AÖV claim that this also causes some issues. Finally, one could try to place $\Gamma \Vdash A$ in `Prop`, the proof-irrelevant universe of strict propositions which has recently been added [25] to Agda. Consequently being proofs of a strict proposition, all $[A]$s would become definitionally equal. But there are conceptual obstacles to how such a construction could work. Restrictions on dependent elimination of `Prop` proofs would likely force all propositions in sight including completeness, soundness and decidability to inhabit `Prop` as well. Moreover, if all proofs of $[A]$ are considered the same then recursion on their derivations does not seem to make sense. Finally, is it even possible to model a type theory describing data in a universe hierarchy of pure propositions? With the preceding statements made, it has to be noted that they are only vague intuitions which have not been investigated formally, so it may well be that something to the effect of `Prop`-ifying the relation is possible.

Further difficulties occur due to the dynamic behaviour of types – in dependent type theory, types have operational semantics similar to that of terms and can take arbitrary syntactic forms. Even with induction-recursion, we need additional insight to tame this behaviour. The canonical solution is to state the logical relations on types in WHNF only and then use weak-head expansion properties (roughly that if $\Gamma \Vdash \mathfrak{J}(u)$ and $t \longrightarrow^* u$, then $\Gamma \Vdash \mathfrak{J}(t)$) to extend this definition to all types.

Finally, the existence of two universes necessitates defining logical families under an induction on the universe level such that the relation at U, a large type, can refer to reducible small types as its elements. The resulting semantics are then of the form $\Gamma \Vdash_\ell \mathfrak{J}$ for $\ell = 0, 1$.

The last step towards establishing the right form of reducibility is to update our understanding of the various parts of the decidability proof. Term and type equality depend on each other, so we show decidability of both together. The proof's three components at term equality adapted to dependent types could look like so:

1. soundness – "if $\Gamma \vdash t \mathrel{\hat{\Longleftrightarrow}} u : A$ then $\Gamma \vdash t = u : A$"
2. completeness – "if $\Gamma \vdash t = u : A$ then $\Gamma \vdash t \mathrel{\hat{\Longleftrightarrow}} u : A$"
3. decidability – "if $\Gamma \vdash t \mathrel{\hat{\Longleftrightarrow}} t : A$ and $\Gamma \vdash u \mathrel{\hat{\Longleftrightarrow}} u : A$ then we can decide whether $\Gamma \vdash t \mathrel{\hat{\Longleftrightarrow}} u : A$"

and analogously for equality of types.

Exhibiting a decision procedure is not very hard, so let us focus on 1 and 2. From the study of $\lambda_\times$ we know that a fundamental theorem will be needed to prove completeness. Recall that since reducible equality implies convertibility, we would expect to prove something like “if $\Gamma \vdash t = u : A$ then exists $[A]$ s.t. $\Gamma \Vdash^{v} t \mathrel{\hat{\Longleftrightarrow}} u : A \,/\, [A]$”.

But what about soundness? Unfortunately dependency presents another stumbling block here. While soundness for STLC was easily provable, proving it for the formulation of λ^{Σ} in 5.2 requires a healthy amount of semantics. In fact, *another* fundamental theorem is needed: “if $\Gamma \vdash t = u : A$ then exists $[A]$ s.t. $\Gamma \Vdash^{v} t = u : A \,/\, [A]$”. Here we are showing a slightly different kind of reducible equality, one which models ordinary definitional equality rather than algorithmic conversion. This of course requires defining another logical relation.

At this point it is worthwhile to consider whether this relation (called the *first* because it comes before completeness) is really necessary. As a model, it provides rather strong results – canonicity and logical consistency of the calculus. While important, they are completely tangential to the goal of demonstrating decidable conversion and are not used in the proof. This pecularity begs the question of whether there might be an easier path to the prerequisites of soundness.

One prerequisite is syntactic validity – for type equality, “if $\Gamma \vdash A = B$ then $\Gamma \vdash A$ and $\Gamma \vdash B$”; for term equality, “if $\Gamma \vdash t = u : A$ then $\Gamma \vdash A$ and $\Gamma \vdash t : A$ and $\Gamma \vdash u : A$”; and correspondingly for well-typedness judgments. As noted by Sterling [41], syntactic validity can be made to be provable syntactically by building the required presuppositions into the typing rules. For example, congruence of the second projection could be enriched as follows:

$$\frac{\Gamma \vdash t = u : \Sigma F{\triangleright}G \qquad \begin{array}{c}\Gamma \vdash F \qquad \Gamma, F \vdash G \\ \Gamma \vdash G[\mathtt{fst}\ t]\end{array} \qquad \Gamma \vdash \mathsf{snd}\ t : G[\mathtt{fst}\ t] \qquad \Gamma \vdash \mathsf{snd}\ u : G[\mathtt{fst}\ t]}{\Gamma \vdash \mathsf{snd}\ t = \mathsf{snd}\ u : G[\mathtt{fst}\ t]}$$

While quite heavyweight syntactically, this process does eliminate the necessity of a semantic argument. It is not inconceivable then that with similar tricks we might be able to prove other prerequisites such as unique typing of neutral terms and get to soundness via syntactic means. Does this mean that conversion-escaping semantics (the *second* logical relation) could be sufficient on its own? Sadly not – we cannot immediately get rid of the first logical relation. This is because its consequences are also necessary to prove the second fundamental theorem.

Recall that certain rules like symmetry and transitivity are omitted from algorithmic conversion due to their non-deterministic nature and that we must show these properties a posteriori to validate the corresponding definitional equalities. Consider algorithmic conversion of neutral second projections:

$$\frac{\Gamma \vdash n \;\longleftrightarrow\; m : \Sigma F{\triangleright}G}{\Gamma \vdash \mathsf{snd}\ n \mathrel{\hat{\longleftrightarrow}} \mathsf{snd}\ m : \mathrm{G}[\mathtt{fst}\ n]}$$

Flipping the conclusion into $\Gamma \vdash \mathsf{snd}\ m \mathrel{\hat{\longleftrightarrow}} \mathsf{snd}\ n : G[\mathtt{fst}\ n]$ to achieve symmetry requires nontrivial theorems about the validity of substitution using the equality $\mathtt{fst}\ n \equiv \mathtt{fst}\ m$, so it isn’t something we can do without semantics. And injecting additional typing assumptions as above to make symmetry hold by fiat seems like a strange thing to do for an algorithm that should be kept minimal. Even worse, transitivity seems to require weak-head normalization. Despite these observations, I believe golfing the proof could be an interesting avenue of future exploration. For now, though, we stick with two fundamental theorems.

As AÖV observe, and my own experience echoes this statement, proving even a single fundamental theorem is a significant undertaking:

> Especially the proof of the fundamental theorem of logical relations is substantial (5.000 lines). The congruence rule for the recursor for natural numbers alone requires a lemma that stretches over more than 500 lines. It is not that the proof is mathematically deep, once the right definition of the logical relation and the right formulation of the fundamental theorem are in place–it is just that a formalization requires us to get all the technicalities right. In research articles with pen and paper proofs only, the proof of the fundamental theorem is often skipped or reduced to the single sentence "proof by induction on the typing and equality derivations". Yet checking that each case of the induction goes through would require a reviewer many hours of disciplined technical reasoning. Written out, the proof would stretch over many pages. [8]

To the existing 5,000 lines, in this project I added just short of another thousand. It would be very painful to carry out the proof twice. To avoid doing this, AÖV make the relation parametric over a *generic equality.*

3 Generic equality and typed reduction

Generic equality is one of two technical devices which AÖV introduce to simplify the formal development. I will briefly recount them here and then summarize the structure of their proof.

1. A set of predicates on a family of binary relations is defined (in 5.5). Any family which meets these predicates is called an *instance of generic equality.* Such an instance is denoted by $\Gamma \vdash A \cong B$ (type equality), $\Gamma \vdash t \cong u : A$ (term equality) and $\Gamma \vdash t \sim u : A$ (neutral term equality). The definition is generic enough that both definitional equality and algorithmic conversion are instances, but specific enough to be usable in a model of the type theory. An instance is symmetric and transitive but irreflexive in general – reflexivity is only required at introduction forms[3]. Besides respecting the typing rules of λ^{Σ}, it must admit type conversion, weakening, weak-head expansion and be compatible with definitional equality.

   On top of any such instance, a family $\Gamma \Vdash t \cong u : A \,/\, [A]$, $\Gamma \Vdash A \cong B \,/\, [A]$ of generic logical relations is defined. We are then able to show:

   - if $\Gamma \vdash t \equiv u : A$ then exists $[A]$ s.t. $\Gamma \Vdash^{v} t \cong u : A \,/\, [A]$ (generic fundamental theorem)
   - if $\Gamma \Vdash t \cong u : A \,/\, [A]$ then $\Gamma \vdash t \cong u : A$ (generic escape)
   - if $\Gamma \Vdash t \cong u : A \,/\, [A]$ then $\Gamma \vdash t \equiv u : A$ (compatibility with definitional equality)

   and correspondingly for type equality.

   As motivated earlier, the abstraction of models over definitional and algorithmic equality makes a massive difference in the formalisation effort. On the other hand, a disadvantage of this approach is that declarative equality rules and algorithmic conversion rules must match very closely. Freedom of design is restricted by a necessity to specify

[3] Although in general the semantics isn't quite Partial Equivalence Relation-based, we could refer to reflexive terms as being *in the PER.*

algorithmic conversion rules close in form to their corresponding definitional equalities. This means that conversion can be lighter on typing premises, but it is difficult to make more fundamental changes such as using algorithm-friendly (ch. 2.2) rules in the algorithm and generating equations such as $f = (\lambda x.\ f\ x)$ in the static type theory.

2. Following the work of Abel, Coquand and Mannaa [7], AÖV use a typed weak-head reduction relation $\Gamma \vdash t \longrightarrow u : A$ as opposed to the usual untyped judgment $t \longrightarrow u$. The construction could be seen as moving from Curry-style towards a Church-style presentation. It has the useful effect of making the inclusion of reduction in equivalence – that $\Gamma \vdash t \longrightarrow u : A$ implies $\Gamma \vdash t = u : A$ – provable early-on via a purely syntactic argument. I found this to be a good idea, although it does force certain non-obvious design choices. For example, the second projection reduction for dependent sums must read

   $$\frac{\Gamma \vdash t : F \qquad \Gamma \vdash u : G[t]}{\Gamma \vdash \mathsf{snd}\ \langle t, u\rangle \longrightarrow u : G[\mathtt{fst}\ \langle t, u\rangle]} \quad \text{rather than} \quad \frac{\Gamma \vdash t : F \qquad \Gamma \vdash u : G[t]}{\Gamma \vdash \mathsf{snd}\ \langle t, u\rangle \longrightarrow u : G[t]}$$

   in order for the syntactic proofs to go through. The latter version is admissible as a consequence of the fundamental theorem.

   This happens because well-typedness of the redex is eventually needed to show weak-head expansion. To get at the correct form of typed reduction, we suggest constructing a typing derivation for the left side, which in this case would be:

   $$\frac{\dfrac{\Gamma \vdash t : F \qquad \Gamma \vdash u : G[t]}{\Gamma \vdash \langle t, u\rangle : \Sigma F \blacktriangleright G}}{\Gamma \vdash \mathit{snd}\ \langle t, u\rangle : G[\mathtt{fst}\ \langle t, u\rangle]}$$

To sum up, the high-level structure of proof is as follows:

1. Establish syntactic properties of typing, definitional equality and reduction rules. Admissibility of weakening, inclusion of reduction in definitional equality, strong confluence, properties of WHNFs.
2. Establish a number of properties of the logical relation over a generic equality culminating in the generic fundamental theorem.
3. Show that definitional equality is an instance of generic equality. Apply the fundamental theorem to derive properties such as injectivity of Π/Σ types, canonicity, consistency, weak-head normalization and admissibility of substitution.
4. Use these properties to prove soundness of algorithmic conversion and show that it is an instance of generic equality.
5. Derive completeness of the algorithm from the generic fundamental theorem.
6. Exhibit the decision procedure and conclude the proof.

3.1 Adding Unit

Having described the high-level proof structure, I will now explain at which points changes had to be made to add the unit type. As a fairly straightforward addition, it clearly demonstrates parts of the codebase.

3.1.1 Type

As we've seen, reducible typehood $\Gamma \Vdash A$ is given by an inductive predicate with the other three judgments defined by recursion on top of it. In code, extending the relational model with a new type has the effect of a new constructor for the predicate and a new case to handle it in each recursive definition:

```
data _⊩¹_ (Γ : Con Term) : Term → Set where
  Unitᵣ : ∀ {A} → Γ ⊩Unit A → Γ ⊩¹ A
  -- other constructors

_⊩¹_≡_/_ : (Γ : Con Term) (A B : Term) → Γ ⊩¹ A → Set
Γ ⊩¹ A ≡ B / Unitᵣ D = Γ ⊩Unit A ≡ B
-- other cases

_⊩¹_::_/_ : (Γ : Con Term) (t A : Term) → Γ ⊩¹ A → Set
Γ ⊩¹ t :: A / Unitᵣ D = Γ ⊩Unit t ::Unit
-- other cases

_⊩¹_≡_::_/_ : (Γ : Con Term) (t u A : Term) → Γ ⊩¹ A → Set
Γ ⊩¹ t ≡ u :: A / Unitᵣ D = Γ ⊩Unit t ≡ u ::Unit
-- other cases
```

As the symbol $\mathbb{1}$ itself is just a single constant, its typehood and type equality are the closures of this constant under weak-head expansion:

```
_⊩Unit_ : (Γ : Con Term) (A : Term) → Set
Γ ⊩Unit A = Γ ⊢ A :⇒*: Unit

_⊩Unit_≡_ : (Γ : Con Term) (A B : Term) → Set
Γ ⊩Unit A ≡ B = Γ ⊢ B ⇒* Unit
```

(The judgment `Γ ⊢ A :⇒*: B` packages together `Γ ⊢ A`, `Γ ⊢ B` and `Γ ⊢ A ⇒* B`. Similarly, `Γ ⊢ t :⇒*: u :: A` entails `Γ ⊢ t :: A`, `Γ ⊢ u :: A` and `Γ ⊢ t ⇒* u :: A`.)

3.1.2 Terms

The unit type contains one canonical term – the star $*$. There is no computational behaviour associated with terms. The present type theory isn't linear, so no eliminators are needed either. Beyond the design of extensionality principles discussed earlier, there isn't anything surprising about the typing rules (fully specified in the appendix):

$$\frac{\vdash \Gamma}{\Gamma \vdash * : \mathbb{1}} \qquad\qquad \frac{\Gamma \vdash t : \mathbb{1} \qquad \Gamma \vdash u : \mathbb{1}}{\Gamma \vdash t = u : \mathbb{1}}$$

Generic equality is modified accordingly. Note that reflexivity of $*$ is derivable from extensionality, so is not added:

$$\frac{\Gamma \vdash t : \mathbb{1} \qquad \Gamma \vdash u : \mathbb{1}}{\Gamma \vdash t \cong u : \mathbb{1}}$$

As for term reducibility at $\mathbb{1}$, after several iterations I was able to cut it down to a bare minimum. In an earlier version, it was specified as for a general positive type [35] by describing reducible WHNFs – the star and reducible neutral terms (below) and taking their closure under weak-head expansion.

```
-- The property of being a reducible unit term in WHNF.
data Unit-prop (Γ : Con Term) : (n : Term) → Set where
  starᵣ : Unit-prop Γ star
  ne    : ∀ {n} → Γ ⊩neNf n ∷ Unit → Unit-prop Γ n
```

However this turned out to be unnecessary, more or less thanks to unit's contractibility[4]. The final definition of a reducible term at 𝟙 is then exactly the same as it would be for STLC, simply postulating that the term reduce to any WHNF at all, as long as it has the right type:

```
record _⊩Unit_∷Unit (Γ : Con Term) (t : Term) : Set where
  inductive
  constructor Unitₜ
  field
    n : Term
    d : Γ ⊢ t :⇒*: n ∷ Unit
    prop : Whnf n
```

To finish the logical relation, we model reducible equality by any pair of terms whatsoever, as long as they have the right type. This postulate is sufficient (again due to contractibility) to prove all needed properties. In contrast with the relation for conversion of $\lambda_\times$, not even weak-head normalisation is required, morally because the termhood relation (which wasn't necessary and so wasn't defined at all for $\lambda_\times$ conversion) already implies it. There are no elimination forms to respect and hence no logicality conditions.

```
record _⊩Unit_≡_∷Unit (Γ : Con Term) (t u : Term) : Set where
  constructor Unitₜ₌
  field
    ⊢t : Γ ⊢ t ∷ Unit
    ⊢u : Γ ⊢ u ∷ Unit
```

3.1.3 Fundamental theorem

Cases of the fundamental theorem for 𝟙 are similarly straightforward. For example, η-unitᵛ below validates extensionality by proving that any two reducible (and valid) terms of unit type are reducibly equal (and stay equal under all substitutions). Some irrelevance has to be applied to move between reducible typehood derivation indices:

```
η-unitᵛ : ∀ {Γ l e e'} ([Γ] : ⊩ᵛ Γ)
          ([Unit] : Γ ⊩ᵛ⟨ l ⟩ Unit / [Γ])
          ([e] : Γ ⊩ᵛ⟨ l ⟩ e ∷ Unit / [Γ] / [Unit])
          ([e'] : Γ ⊩ᵛ⟨ l ⟩ e' ∷ Unit / [Γ] / [Unit])
          → Γ ⊩ᵛ⟨ l ⟩ e ≡ e' ∷ Unit / [Γ] / [Unit]
η-unitᵛ {Γ} {l} {e} {e'} [Γ] [Unit] [e] [e'] {Δ} {σ} ⊢Δ [σ] =
  let J = proj₁ ([Unit] ⊢Δ [σ])
      [σe] = proj₁ ([e] ⊢Δ [σ])
      [σe'] = proj₁ ([e'] ⊢Δ [σ])
      UnitJ : Δ ⊩⟨ l ⟩ Unit
      UnitJ = Unitᵣ (idRed:*: (Unitⱼ ⊢Δ))
      [σe] = irrelevanceTerm J UnitJ [σe]
      [σe'] = irrelevanceTerm J UnitJ [σe']
      ⊢σe = escapeTerm UnitJ [σe]
      ⊢σe' = escapeTerm UnitJ [σe']
  in  irrelevanceEqTerm UnitJ J (Unitₜ₌ ⊢σe ⊢σe')
```

[4] The property that there exists a term of this type s.t. all elements of the type are equal to it.

(A detail of the above definition which I entirely glossed over is that besides being indexed by derivations of typehood, validity judgments also have to be indexed by *context validity* derivations $\Vdash^v \Gamma$. In fact, valid typehood itself is indexed by these. So given a derivation $[\Gamma]$ of $\Vdash^v \Gamma$, as well as a derivation $[A]$ of $\Gamma \Vdash^v A \,/\, [\Gamma]$, we get term validity $\Gamma \Vdash^v t : A \,/\, [\Gamma] \,/\, [A]$ and so forth. The reason for this extra index is essentially the same as for $[A]$ – it is there to bootstrap the inductive-recursive definition of reducibly equal substitutions (ch. 2.2). While being yet another detail to keep in mind, it is not conceptually very important.)

3.1.4 Conversion

For $\mathbb{1}$, we extend the algorithm with a single extensionality principle shown below. A few cases (in appendix) are also added to reflexivity and compatibility rules to ensure that $\mathbb{1} \Longleftrightarrow \mathbb{1}$, that neutral conversion at $\mathbb{1}$ implies algorithmic conversion at $\mathbb{1}$ and so on.

$$\frac{\Gamma \vdash \bar{t} : \mathbb{1} \qquad \Gamma \vdash \bar{u} : \mathbb{1}}{\Gamma \vdash \bar{t} \Longleftrightarrow \bar{u} : \mathbb{1}}$$

Note that the algorithm I procure is actually somewhat inefficient at comparing unitary terms. Recall that the purpose of $\hat{\Longleftrightarrow}$ is to reduce both sides before invoking $\Longleftrightarrow$. When tasked with checking $\Gamma \vdash t \mathrel{\hat{\Longleftrightarrow}} u : A$ where $\Gamma \vdash A \longrightarrow^* \mathbb{1}$, the procedure is going to unnecessarily reduce all terms to WHNF even though it could just reduce the type, observe that it is $\mathbb{1}$ and then immediately appeal to contractibility. After reduction, this appeal is finally made by $\Longleftrightarrow$.

I do not carry out this kind of microptimization because it would likely require nontrivial changes to the structure of the formal codebase and conversion judgments. An interesting idea would be to postulate that *every* term of type $\mathbb{1}$ is in WHNF in order to completely banish reduction. But again, implementing the idea here would require a substantial amount of effort either in making WHNFhood typed or in some other change which preserves necessary provabilities. As such, it is left as future work. A similar idea (to my best knowledge not discussed in literature, although the implementation of the Lean theorem prover [21], whose conjectured undecidability [2] could also potentially be fixed by banishing proof reduction, seems to do it) could work for converting proofs in a proof-irrelevant universe.

Going back to the present code, a bit of notation – algorithmic conversion judgments are expressed in Agda as follows:

Judgment	Agda representation
$\Gamma \vdash A \mathrel{\hat{\Longleftrightarrow}} B$	`Γ ⊢ A [conv↑] B`
$\Gamma \vdash A \Longleftrightarrow B$	`Γ ⊢ A [conv↓] B`
$\Gamma \vdash t \mathrel{\hat{\Longleftrightarrow}} u : A$	`Γ ⊢ t [conv↑] u ∷ A`
$\Gamma \vdash t \Longleftrightarrow u : A$	`Γ ⊢ t [conv↓] u ∷ A`
$\Gamma \vdash t \mathrel{\hat{\longleftrightarrow}} u : A$	`Γ ⊢ t ~ u ↑ A`
$\Gamma \vdash t \longleftrightarrow u : A$	`Γ ⊢ t ~ u ↓ A`

And in the decision procedure, thanks to unit's contractibility and utilizing some consequences of the first fundamental theorem (`syntacticEqTerm` is syntactic validity) we can just answer `yes`:

```
decConv↓Term-Unit : ∀ {t u Γ}
                  → Γ ⊢ t [conv↓] t ∷ Unit
                  → Γ ⊢ u [conv↓] u ∷ Unit
                  → Dec (Γ ⊢ t [conv↓] u ∷ Unit)
```

```
decConv↓Term-Unit tConv uConv =
  let t≡t = soundnessConv↓Term tConv
      u≡u = soundnessConv↓Term uConv
      _ , [t] , _ = syntacticEqTerm t≡t
      _ , [u] , _ = syntacticEqTerm u≡u
      _ , tWhnf , _ = whnfConv↓Term tConv
      _ , uWhnf , _ = whnfConv↓Term uConv
  in  yes (η-unit [t] [u] tWhnf uWhnf)
```

3.2 Adding Sigma

While 𝟙 was a walk in the park, dependent sums engage the entire machinery developed so far.

3.2.1 Types

In an attempt to minimize duplication while defining the type family, I generalized the rules for Π types to talk about $\mathfrak{B}$*inding types* – Π and Σ – instead.

$$\frac{\Gamma \vdash F \qquad \Gamma, F \vdash G}{\Gamma \vdash \mathfrak{B}F \blacktriangleright G} \text{ (for } \mathfrak{B} = \Pi, \Sigma\text{)}$$

$$\frac{\Gamma \vdash F \qquad \Gamma \vdash F = H \qquad \Gamma, F \vdash G = E}{\Gamma \vdash \mathfrak{B}F \blacktriangleright G = \mathfrak{B}H \blacktriangleright E} \text{ (for } \mathfrak{B} = \Pi, \Sigma\text{)}$$

In code, the symbols are denoted by two constants `BΠ, BΣ : BindingType` and the types by a generic type `⟦_⟧ F ▸ G`:

```
⟦_⟧_▹_ : BindingType → Term → Term → Term
⟦ BΠ ⟧ F ▹ G = Π F ▹ G
⟦ BΣ ⟧ F ▹ G = Σ F ▹ G
```

Wherever possible, mechanised proofs are also generic over the two.

Primarily due to the presence of variable binding, reducibility of a $\mathfrak{B}$-type already has to postulate a number of requirements:

```
record _⊩¹B⟨_⟩_ (Γ : Con Term) (W : BindingType) (A : Term) : Set where
  inductive
  constructor Bᵣ
  field
    F : Term
    G : Term
    D : Γ ⊢ A :⇒*: ⟦ W ⟧ F ▹ G
    ⊢F : Γ ⊢ F
    ⊢G : Γ ∙ F ⊢ G
    A≡A : Γ ⊢ ⟦ W ⟧ F ▹ G ≅ ⟦ W ⟧ F ▹ G
    [F] : ∀ {ρ Δ} → ρ ∷ Δ ⊆ Γ → ⊢ Δ → Δ ⊩¹ U.wk ρ F
    [G] : ∀ {ρ Δ a}
        → ([ρ] : ρ ∷ Δ ⊆ Γ) (⊢Δ : ⊢ Δ)
        → Δ ⊩¹ a ∷ U.wk ρ F / [F] [ρ] ⊢Δ
        → Δ ⊩¹ U.wk (lift ρ) G [ a ]
    G-ext : ∀ {ρ Δ a b}
          → ([ρ] : ρ ∷ Δ ⊆ Γ) (⊢Δ : ⊢ Δ)
          → ([a] : Δ ⊩¹ a ∷ U.wk ρ F / [F] [ρ] ⊢Δ)
          → ([b] : Δ ⊩¹ b ∷ U.wk ρ F / [F] [ρ] ⊢Δ)
```

```
      → Δ ⊩¹ a ≡ b ∷ U.wk ρ F / [F] [ρ] ⊢Δ
      → Δ ⊩¹ U.wk (lift ρ) G [ a ] ≡ U.wk (lift ρ) G [ b ] / [G] [ρ] ⊢Δ [a]
```

These are generalized from reducibility of Π types without modification, so I will only briefly sum them up here. Kripke monotonicity generates a fair bit of syntactic noise and tedium – let's attempt to tease out underlying properties which entail A being a reducible $\mathfrak{B}$-type:

- The existence of a WHNF of the right shape to which A reduces (also achieving weak-head expansion closure) is postulated in `F`, `G` and `D`. Well-typedness of the WHNF's components is enforced in `⊢F` and `⊢G`, which also force syntactic validity of Σ (if $\Gamma \vdash \Sigma F \triangleright G$ then $\Gamma \vdash F$ and $\Gamma, F \vdash G$). The WHNFs must be reflexive w.r.t. generic equality, or *in the generic PER*, says `A≡A`.

- `[F]` and `[G]` are componentwise reducibility postulates. The latter is a logicality condition – `G` is effectively a type-level reducible function in that it takes reducible arguments to reducible types.

- `G-ext` is logicality w.r.t. reducible term equality – reducibly equal reducible terms substitute into `G` to yield reducibly equal types.

Reducible equality of $\mathfrak{B}$-types is similar. $\Gamma \Vdash \mathfrak{B}F \triangleright G \cong \mathfrak{B}F' \triangleright G' \mathbin{/} [A]$ breaks down into componentwise reducible equalities. `[F≡F′]` is just Kripkefied $\Gamma \Vdash F \cong F' \mathbin{/} [F]$, while `[G≡G′]` could be seen as $\Gamma, F \Vdash G \cong G' \mathbin{/} [G]$ – a reducible equality of type families. As such, it generates another logicality condition which ensures the types stay equal under substitutions of reducible terms. The two equalities `[F≡F′]` and `[G≡G′]` also have the effect of enforcing injectivity of Π and Σ types.

```
record _⊩¹B⟨_⟩_≡_/_ (Γ : Con Term) (W : BindingType) (A B : Term) ([A] : Γ ⊩¹B⟨ W ⟩ A) : Set
↪ where
  inductive
  constructor B₌
  -- Access implications of the reducibility derivation [A] we are indexed by.
  open _⊩¹B⟨_⟩_ [A]
  field
    F′     : Term
    G′     : Term
    D′     : Γ ⊢ B ⇒* ⟦ W ⟧ F′ ▹ G′
    A≡B    : Γ ⊢ ⟦ W ⟧ F ▹ G ≅ ⟦ W ⟧ F′ ▹ G′
    [F≡F′] : ∀ {ρ Δ}
           → ([ρ] : ρ ∷ Δ ⊆ Γ) (⊢Δ : ⊢ Δ)
           → Δ ⊩¹ U.wk ρ F ≡ U.wk ρ F′ / [F] [ρ] ⊢Δ
    [G≡G′] : ∀ {ρ Δ a}
           → ([ρ] : ρ ∷ Δ ⊆ Γ) (⊢Δ : ⊢ Δ)
           → ([a] : Δ ⊩¹ a ∷ U.wk ρ F / [F] [ρ] ⊢Δ)
           → Δ ⊩¹ U.wk (lift ρ) G [ a ] ≡ U.wk (lift ρ) G′ [ a ] / [G] [ρ] ⊢Δ [a]
```

Like for $\mathbb{1}$, reducible typing and reducible type equality are added to the general relations:

```
data _⊩¹_ (Γ : Con Term) : Term → Set where
  Bᵣ  : ∀ {A} W → Γ ⊩¹B⟨ W ⟩ A → Γ ⊩¹ A
  -- other constructors

_⊩¹_≡_/_ : (Γ : Con Term) (A B : Term) → Γ ⊩¹ A → Set
Γ ⊩¹ A ≡ B / Bᵣ W [A] = Γ ⊩¹B⟨ W ⟩ A ≡ B / [A]
-- other cases
```

3.2.2 Terms

The term-level rules for dependent pairs are below. Their operational semantics of typed reduction (in appendix 5.3) are exactly the equalities except extensionality, read left-to-right.

$$\frac{\Gamma \vdash F \qquad \Gamma, F \vdash G \qquad \Gamma \vdash t : F \qquad \Gamma \vdash u : G[t]}{\Gamma \vdash \langle t, u \rangle : \Sigma F \blacktriangleright G}$$

$$\frac{\Gamma \vdash F \qquad \Gamma, F \vdash G \qquad \Gamma \vdash t : \Sigma F \blacktriangleright G}{\Gamma \vdash \mathsf{fst}\, t : F} \qquad \frac{\Gamma \vdash F \qquad \Gamma, F \vdash G \qquad \Gamma \vdash t : \Sigma F \blacktriangleright G}{\Gamma \vdash \mathsf{snd}\, t : G[\mathsf{fst}\, t]}$$

$$\frac{\Gamma \vdash F \qquad \Gamma, F \vdash G \qquad \Gamma \vdash t = u : \Sigma F \blacktriangleright G}{\Gamma \vdash \mathsf{fst}\, t = \mathsf{fst}\, u : F} \qquad \frac{\Gamma \vdash F \qquad \Gamma, F \vdash G \qquad \Gamma \vdash t = u : \Sigma F \blacktriangleright G}{\Gamma \vdash \mathsf{snd}\, t = \mathsf{snd}\, u : G[\mathsf{fst}\, t]}$$

$$\frac{\Gamma \vdash F \qquad \Gamma, F \vdash G \qquad \Gamma \vdash t : F \qquad \Gamma \vdash u : G[t]}{\Gamma \vdash \mathsf{fst}\, \langle t, u \rangle = t : F}$$

$$\frac{\Gamma \vdash F \qquad \Gamma, F \vdash G \qquad \Gamma \vdash t : F \qquad \Gamma \vdash u : G[t]}{\Gamma \vdash \mathsf{snd}\, \langle t, u \rangle = u : G[\mathsf{fst}\, \langle t, u \rangle]}$$

$$\frac{\begin{array}{c} \Gamma \vdash F \qquad \Gamma, F \vdash G \\ \Gamma \vdash t : \Sigma F \blacktriangleright G \qquad \Gamma \vdash u : \Sigma F \blacktriangleright G \qquad \Gamma \vdash \mathsf{fst}\, t = \mathsf{fst}\, u : F \qquad \Gamma \vdash \mathsf{snd}\, t = \mathsf{snd}\, u : G[\mathsf{fst}\, t] \end{array}}{\Gamma \vdash t = u : \Sigma F \blacktriangleright G}$$

The design of these rules follows principles laid out in the preceding discussion. The extensionality principle looks at all observations we can make about a pair. It implies congruence of the introduction form $\langle t, u \rangle$, so we do not postulate it. On the other hand, we *do* need explicit congruence at elimination forms. The rule for reduction of snd has a peculiar type arrived at via typing the left-hand side.

Note also the following – every single rule has $\Gamma \vdash F$ and $\Gamma, F \vdash G$, the propositions from which $\Gamma \vdash \Sigma F \blacktriangleright G$ follows, in its premises. Why is this? Recall the discussion from ch. 2.3 on proving syntactic validity without semantics. This design choice is meant to achieve that on a smaller scale – we don't get validity for everything, but we get enough premises from inversion to simplify some proofs. The downside is merely some syntactic noise which I maintain is worth it for the extra workload it reduces.

Interestingly, we don't add operational rules to generic equality (below) – they are going to be admissible via the semantic argument. Consequently, we can view generic equality as closer to algorithmic conversion.

$$\frac{\begin{array}{c} \Gamma \vdash F \qquad \Gamma, F \vdash G \\ \Gamma \vdash \bar{t} : \Sigma F \blacktriangleright G \qquad \Gamma \vdash \bar{u} : \Sigma F \blacktriangleright G \qquad \Gamma \vdash \mathsf{fst}\, \bar{t} \cong \mathsf{fst}\, \bar{u} : F \qquad \Gamma \vdash \mathsf{snd}\, \bar{t} \cong \mathsf{snd}\, \bar{u} : G[\mathsf{fst}\, \bar{t}] \end{array}}{\Gamma \vdash t \cong u : \Sigma F \blacktriangleright G}$$

$$\frac{\Gamma \vdash F \qquad \Gamma, F \vdash G \qquad \Gamma \vdash n \sim m : \Sigma F \blacktriangleright G}{\Gamma \vdash \mathsf{fst}\, n \sim \mathsf{fst}\, m : F} \qquad \frac{\Gamma \vdash F \qquad \Gamma, F \vdash G \qquad \Gamma \vdash n \sim m : \Sigma F \blacktriangleright G}{\Gamma \vdash \mathsf{snd}\, n \sim \mathsf{snd}\, m : G[\mathsf{fst}\, n]}$$

Now for the semantics. At term level, the logical relations for Π and Σ are very different, so split into type-specific definitions:

```
_⊩¹_∷_/_ : (Γ : Con Term) (t A : Term) → Γ ⊩¹ A → Set
Γ ⊩¹ t ∷ A / Bᵣ BΠ [A]  = Γ ⊩¹Π t ∷ A / [A]
Γ ⊩¹ t ∷ A / Bᵣ BΣ [A]  = Γ ⊩¹Σ t ∷ A / [A]
-- other cases

_⊩¹_≡_∷_/_ : (Γ : Con Term) (t u A : Term) → Γ ⊩¹ A → Set
Γ ⊩¹ t ≡ u ∷ A / Bᵣ BΠ [A] = Γ ⊩¹Π t ≡ u ∷ A / [A]
Γ ⊩¹ t ≡ u ∷ A / Bᵣ BΣ [A]  = Γ ⊩¹Σ t ≡ u ∷ A / [A]
-- other cases
```

To rephrase the discussion from ch. 2 – in type-theoretic semantics of type theory, the object modelling a type mirrors its structure. So we model dependent pairs by dependent pairs. The Σ types are presented as *negative* [34] – reducibility of pairs specifies their meaning in terms of their behaviour under elimination forms (projections).

While it might be possible to formulate something like `Product-prop` similar to `Unit-prop` which holds for reducibly neutral terms and for pairs of reducible terms, when attempting to do so I was unable to convince Agda's termination checker that the resulting part of the inductive-recursive definition of logical relations was well-formed. Perhaps the lesson here is that negative formulations are simply preferrable for this form of metatheory.

Dependent pair reducibility (below) includes the usual layer of reduction to WHNF (`Product` is a WHNFhood predicate for Σ) and generic reflexivity. We then construct a metatheoretic dependent pair modelling the object theory pair:

- reducibility under `fst` at `F`
- reducibility under `snd` at `G[ fst p ]` given `Γ ⊩ fst p ∷ F / [F]`, i.e. the first component

Like in the relation for $\lambda_\times$, monotonicity isn't built-in for these as the term-level rules for Σ don't involve binders.

```
_⊩¹Σ_∷_/_ : (Γ : Con Term) (t A : Term) ([A] : Γ ⊩¹B⟨ BΣ ⟩ A) → Set
Γ ⊩¹Σ t ∷ A / [A]@(Bᵣ F G D ⊢F ⊢G A≡A [F] [G] G-ext) =
  ∃ λ p → Γ ⊢ t :⇒*: p ∷ Σ F ▹ G
        × Product p
        × Γ ⊢ p ≅ p ∷ Σ F ▹ G
        -- Weakening by identity (a noop) is necessary for technical reasons.
        × (Σ (Γ ⊩¹ fst p ∷ U.wk id F / [F] id (wf ⊢F)) λ [fst]
            → Γ ⊩¹ snd p ∷ U.wk (lift id) G [ fst p ] / [G] id (wf ⊢F) [fst])
```

Reducible term equality is similar and includes the expected logicality conditions. This relation is somewhat excessive – it pulls in reducibility of `t` using `Γ ⊩¹Σ t ∷ A / [A]` and ditto for `u`. It also postulates reducibility of both `fst p` and `fst r` to achieve symmetry by fiat. It could be simplified, but the returns from each successive refactoring are increasingly diminishing.

```
_⊩¹Σ_≡_∷_/_ : (Γ : Con Term) (t u A : Term) ([A] : Γ ⊩¹B⟨ BΣ ⟩ A) → Set
Γ ⊩¹Σ t ≡ u ∷ A / [A]@(Bᵣ F G D ⊢F ⊢G A≡A [F] [G] G-ext) =
  ∃₂ λ p r → Γ ⊢ t :⇒*: p ∷ Σ F ▹ G
           × Γ ⊢ u :⇒*: r ∷ Σ F ▹ G
           × Product p
           × Product r
           × Γ ⊢ p ≅ r ∷ Σ F ▹ G
           × Γ ⊩¹Σ t ∷ A / [A]
           × Γ ⊩¹Σ u ∷ A / [A]
           × (Σ (Γ ⊩¹ fst p ∷ U.wk id F / [F] id (wf ⊢F)) λ [fstp]
               → Γ ⊩¹ fst r ∷ U.wk id F / [F] id (wf ⊢F)
               × Γ ⊩¹ fst p ≡ fst r ∷ U.wk id F / [F] id (wf ⊢F)
               × Γ ⊩¹ snd p ≡ snd r ∷ U.wk (lift id) G [ fst p ] / [G] id (wf ⊢F) [fstp])
```

3.2.3 Fundamental theorem

Unfortunately reinforcing AÖV's sentiment, the proof of the fundamental theorem is not incredibly illuminating. All interesting choices sit in the design of the relational semantics, generic equality and the algorithm, while the present proof consists of large amounts of mostly mechanical manipulations of terms under reduction, substitution and so on. The proof's value is dynamic – success or failure of carrying it through informs the design of the specifications. In fact, this kind of trial-and-error is how I arrived at the correct definitions presented here after a fair amount of iterations whose reasons for failure I try to synthesize as principles in the preceding chapters.

3.2.4 Conversion

And finally, the conversion algorithm:

$$\frac{\Gamma \vdash n \;\longleftrightarrow\; m : \Sigma F \triangleright G}{\Gamma \vdash \mathsf{fst}\, n \mathrel{\hat{\longleftrightarrow}} \mathsf{fst}\, m : F} \qquad \frac{\Gamma \vdash n \;\longleftrightarrow\; m : \Sigma F \triangleright G}{\Gamma \vdash \mathsf{snd}\, n \mathrel{\hat{\longleftrightarrow}} \mathsf{snd}\, m : G[\mathsf{fst}\, n]}$$

$$\frac{\begin{array}{c}\Gamma \vdash \bar{p} : \Sigma F \triangleright G \\ \Gamma \vdash \bar{r} : \Sigma F \triangleright G \qquad \Gamma \vdash \mathsf{fst}\, \bar{p} \mathrel{\hat{\Longleftrightarrow}} \mathsf{fst}\, \bar{r} : F \qquad \Gamma \vdash \mathsf{snd}\, \bar{p} \mathrel{\hat{\Longleftrightarrow}} \mathsf{snd}\, \bar{r} : G[\mathsf{fst}\, \bar{p}]\end{array}}{\Gamma \vdash \bar{p} \;\Longleftrightarrow\; \bar{r} : \Sigma F \triangleright G}$$

The only weak-head conversion rule is extensionality. During neutral conversion, we inspect pairs which projections are applied to. Eliminators are never in non-neutral WHNF, so they do not have to be checked by $\Longleftrightarrow$. Correspondingly, introduction forms are never neutral so $\longleftrightarrow$ does not need an extensionality rule.

This concludes the presentation.

4 Conclusions

I presented the extension of a practical type conversion algorithm for dependent type theory by a contractible unit type and dependent sum types. Philosophical and practical aspects of the theory, the algorithm, and their semantics were discussed, first on the example of simply-typed lambda calculi and then the type theory proper. Decidability of conversion for the extended theory using the algorithm has been formally proven in the Agda theorem prover via an argument using Kripke logical relations.

4.1 Related work

The primary piece of literature to mention is of course the work of Abel, Öhman and Vezzosi (AÖV) [8] on which this project is based. Their work, in turn, came from a line of research going back to Coquand's '91 paper [18] and focusing on efficient weak-head conversion algorithms. The closest one is probably that exhibited by Abel and Scherer [3] for a type theory with an irrelevance modality. Its authors had to use two logical relations which AÖV subsequently merged into the generic one seen in ch. 3. Also important is a note due to Abel, Coquand and Mannaa [7] introducing typed reduction. A slightly more distant relative due to Harper and Pfenning [31] checks conversion in the LF logical framework but crucially relies on the absence of large elimination resulting in an ability to erase type dependencies and prove injectivity of Π-types early-on.

A related strand of work explores the use of Normalization by Evaluation (NbE) [15, 1] in type-checking and conversion algorithms. Together with many collaborators, Abel investigates a series of such algorithms. First with Aehlig and Dybjer [4] for a type theory with Π types, one universe and an untyped notion of conversion. Then, with Coquand and Dybjer [5] for an earlier version of $\lambda^{\Pi UN}$, the type theory considered by AÖV which λ^{Σ} extends. Later, with Coquand and Pagano [6] for types including proof-irrelevant propositions. To my best knowledge the only work in this area which has been formalised, and even then only partially so, is that of Altenkirch and Kaposi [10] on decidability of equality in a type theory without large elimination. They use Agda for the mechanisation as done here. Recently, Gratzer, Sterling and Birkedal [28] presented a type checking algorithm for a type theory including a necessity modality and close in many respects to λ^{Σ}. Being NbE-based, their algorithm is a very interesting point of comparison in the design space.

4.2 Future work

The obvious direction for future work is to carry on, attempting to exhibit and verify a weak-head conversion procedure for extensional identity types, be they in OTT form [13], XTT form [44] or perhaps taken from the recent setoid type theory of Altenkirch and collaborators [14]. But with experience gained from this project, the predicted workload is absolutely massive. Even without mechanisation, paper proofs could reasonably reach a hundred pages or more. Can we do better? In informal communication and certain literature (e.g. Gratzer et al. [28]), members of the community hint at the categorical viewpoint as something to investigate in search for better proof methods. With sufficient squinting, rather general structure can be seen to emerge from proofs by logical relation. The methods of *categorical gluing* and semantic normalization are one way to codify this intuition. Papers to look at in this area include the work of Altenkirch, Hofmann and Streicher [12] and of Fiore [24] on normalization of the simply-typed lambda calculus. The work of Altenkirch and Kaposi [10] also employs categorical methods to some extent. Coquand [19] presents a proof sketch for normalization in a type theory with a cumulative universe hierarchy. Gluing in particular is discussed in the work of Sterling and Spitters [42] as well as Sterling [40]. The general semantic framework of Uemura [46] can be hoped to unify many of these results. Lastly, Brunerie [16] has been working on a (to my best knowledge unpublished) formalised proof of the initiality conjecture.

5 Appendix A

The type theory λ^{Σ}, its weak-head operational semantics, as well as the type and term conversion algorithm for it, are fully defined below. Black text represents rules due to AÖV [8], which are reproduced from their work with small modifications. Blue text represents rules for the empty type added by Gaëtan Gilbert in a GitHub pull request [26]. To my best knowledge this has not been written up outside of Agda code. Finally, red text represents rules added by me in the present work. Taken together, coloured annotations visualize contributions of other people to, and the expansion of, the proof since published by AÖV.

5.1 Grammar

$$\begin{array}{llll}
\mathtt{ℕ} & \ni x & & \\
\mathtt{Exp} & \ni t,u,v,A,B & ::= & \bar{t} \mid t\ u \mid \mathtt{fst}\ t \mid \mathtt{snd}\ t \mid \mathtt{natrec}\ A\ t\ u\ v \mid \mathtt{Emptyrec}\ A\ t \\
\mathtt{Whnf} & \ni \bar{t} & ::= & n \mid \lambda t \mid \langle t,u \rangle \mid \mathtt{zero} \mid \mathtt{suc}\ t \mid * \mid \\
 & & & U \mid \Pi A \blacktriangleright B \mid \Sigma A \blacktriangleright B \mid \mathbb{N} \mid \mathbb{1} \mid \mathbb{0} \\
\mathtt{Ne} & \ni n,m,N,M & ::= & i_x \mid n\ t \mid \mathtt{fst}\ n \mid \mathtt{snd}\ n \mid \mathtt{natrec}\ A\ t\ u\ n \mid \mathtt{Emptyrec}\ A\ n \\
\mathtt{Cxt} & \ni \Gamma,\Delta & ::= & \varepsilon \mid \Gamma, A \\
\mathtt{Wk} & \ni \rho & ::= & \mathtt{id} \mid \uparrow \rho \mid \Uparrow \rho \mid \rho \circ \rho' \\
\mathtt{Subst} & \ni \sigma & ::= & \rho \mid \uparrow \sigma \mid \Uparrow \sigma \mid \sigma \circ \sigma' \mid \sigma, t
\end{array}$$

Figure 5.1: Grammar of λ^{Σ}

5.2 Terms and types

Well-formedness of types $\boxed{\Gamma \vdash A}$

$$\frac{\Gamma \vdash F \qquad \Gamma, F \vdash G}{\Gamma \vdash \mathfrak{B} F \blacktriangleright G}\ (\text{for } \mathfrak{B} = \Pi, \Sigma) \qquad \frac{\vdash \Gamma}{\Gamma \vdash A}\ (\text{for } A = U, \mathbb{N}, \mathbb{1}, \mathbb{0}) \qquad \frac{\Gamma \vdash A : U}{\Gamma \vdash A}$$

Equality of types $\boxed{\Gamma \vdash A = B}$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A = A} \qquad \frac{\Gamma \vdash A = B}{\Gamma \vdash B = A} \qquad \frac{\Gamma \vdash A = B \qquad \Gamma \vdash B = C}{\Gamma \vdash A = C}$$

$$\frac{\Gamma \vdash F \qquad \Gamma \vdash F = H \qquad \Gamma, F \vdash G = E}{\Gamma \vdash \mathfrak{B} F \blacktriangleright G = \mathfrak{B} H \blacktriangleright E}\ (\text{for } \mathfrak{B} = \Pi, \Sigma) \qquad \frac{\Gamma \vdash A = B : U}{\Gamma \vdash A = B}$$

Well-formedness of terms $\boxed{\Gamma \vdash t : A}$

$$\frac{\vdash \Gamma}{\Gamma \vdash A : U}\ (\text{for } A = \mathbb{N}, \mathbb{1}, \mathbb{0}) \qquad \frac{\Gamma \vdash F : U \qquad \Gamma, F \vdash G : U}{\Gamma \vdash \mathfrak{B} F \triangleright G : U}\ (\text{for } \mathfrak{B} = \Pi, \Sigma)$$

$$\frac{\vdash \Gamma \qquad i : A \in \Gamma}{\Gamma \vdash i : A} \qquad \frac{\Gamma \vdash F \qquad \Gamma, F \vdash t : G}{\Gamma \vdash \lambda t : \Pi F \triangleright G} \qquad \frac{\Gamma \vdash g : \Pi F \triangleright G \qquad \Gamma \vdash a : F}{\Gamma \vdash g\ a : G[a]}$$

$$\frac{\Gamma \vdash F \qquad \Gamma, F \vdash G \qquad \Gamma \vdash t : F \qquad \Gamma \vdash u : G[t]}{\Gamma \vdash \langle t, u \rangle : \Sigma F \triangleright G}$$

$$\frac{\Gamma \vdash F \qquad \Gamma, F \vdash G \qquad \Gamma \vdash p : \Sigma F \triangleright G}{\Gamma \vdash \mathsf{fst}\ p : F} \qquad \frac{\Gamma \vdash F \qquad \Gamma, F \vdash G \qquad \Gamma \vdash p : \Sigma F \triangleright G}{\Gamma \vdash \mathsf{snd}\ p : G[\mathsf{fst}\ p]}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{zero} : \mathbb{N}} \qquad \frac{\Gamma \vdash t : \mathbb{N}}{\Gamma \vdash \mathsf{suc}\ t : \mathbb{N}}$$

$$\frac{\Gamma, \mathbb{N} \vdash G \qquad \Gamma \vdash z : G[\mathsf{zero}] \qquad \Gamma \vdash s : \Pi \mathbb{N} \triangleright (G \to G[\uparrow \mathsf{id}, \mathsf{suc}\ i_0]) \qquad \Gamma \vdash t : \mathbb{N}}{\Gamma \vdash \mathsf{natrec}\ G\ z\ s\ t : G[t]}$$

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash t : \mathbb{0}}{\Gamma \vdash \texttt{Emptyrec}\ A\ t : A} \qquad \frac{\vdash \Gamma}{\Gamma \vdash * : \mathbb{1}} \qquad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash A = B}{\Gamma \vdash t : B}$$

Equality of terms $\boxed{\Gamma \vdash t = u : A}$

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash t = t : A} \qquad \frac{\Gamma \vdash t = u : A}{\Gamma \vdash u = t : A} \qquad \frac{\Gamma \vdash t_1 = t_2 : A \qquad \Gamma \vdash t_2 = t_3 : A}{\Gamma \vdash t_1 = t_3 : A}$$

$$\frac{\Gamma \vdash t = u : A \qquad \Gamma \vdash A = B}{\Gamma \vdash t = u : B}$$

$$\frac{\Gamma \vdash F \qquad \Gamma \vdash F = H : U \qquad \Gamma, F \vdash G = E : U}{\Gamma \vdash \mathfrak{B} F \triangleright G = \mathfrak{B} H \triangleright E : U} \text{ (for } \mathfrak{B} = \Pi, \Sigma\text{)}$$

$$\frac{\Gamma \vdash f = g : \Pi F \triangleright G \qquad \Gamma \vdash a = b : F}{\Gamma \vdash f\ a = g\ b : G[a]} \qquad \frac{\Gamma \vdash F \qquad \Gamma, F \vdash t : G \qquad \Gamma \vdash a : F}{\Gamma \vdash (\lambda t)\ a = t[a] : G[a]}$$

$$\frac{\Gamma \vdash f : \Pi F \triangleright G \qquad \Gamma \vdash g : \Pi F \triangleright G \qquad \Gamma, F \vdash f[\uparrow \mathsf{id}]\ i_0 = g[\uparrow \mathsf{id}]\ i_0 : G}{\Gamma \vdash f = g : \Pi F \triangleright G}$$

$$\frac{\Gamma \vdash F \qquad \Gamma, F \vdash G \qquad \Gamma \vdash t = u : \Sigma F \triangleright G}{\Gamma \vdash \mathsf{fst}\ t = \mathsf{fst}\ u : F} \qquad \frac{\Gamma \vdash F \qquad \Gamma, F \vdash G \qquad \Gamma \vdash t = u : \Sigma F \triangleright G}{\Gamma \vdash \mathsf{snd}\ t = \mathsf{snd}\ u : G[\mathsf{fst}\ t]}$$

$$\frac{\Gamma \vdash F \qquad \Gamma, F \vdash G \qquad \Gamma \vdash t : F \qquad \Gamma \vdash u : G[t]}{\Gamma \vdash \mathsf{fst}\ \langle t, u \rangle = t : F}$$

$$\frac{\Gamma \vdash F \qquad \Gamma, F \vdash G \qquad \Gamma \vdash t : F \qquad \Gamma \vdash u : G[t]}{\Gamma \vdash \mathsf{snd}\ \langle t, u \rangle = u : G[\mathsf{fst}\ \langle t, u \rangle]}$$

$$\frac{\begin{array}{c} \Gamma \vdash F \qquad \Gamma, F \vdash G \\ \Gamma \vdash t : \Sigma F \triangleright G \qquad \Gamma \vdash u : \Sigma F \triangleright G \qquad \Gamma \vdash \mathsf{fst}\ t = \mathsf{fst}\ u : F \qquad \Gamma \vdash \mathsf{snd}\ t = \mathsf{snd}\ u : G[\mathsf{fst}\ t] \end{array}}{\Gamma \vdash t = u : \Sigma F \triangleright G}$$

$$\frac{\Gamma \vdash t = u : \mathbb{N}}{\Gamma \vdash \mathsf{suc}\ t = \mathsf{suc}\ u : \mathbb{N}}$$

$$\frac{\begin{array}{c} \Gamma, \mathbb{N} \vdash G_1 = G_2 \\ \Gamma \vdash z_1 = z_2 : G_1[\mathsf{zero}] \qquad \Gamma \vdash s_1 = s_2 : \Pi \mathbb{N} \triangleright (G_1 \to G_1[\uparrow \mathsf{id}, \mathsf{suc}\ i_0]) \qquad \Gamma \vdash t_1 = t_2 : \mathbb{N} \end{array}}{\Gamma \vdash \mathsf{natrec}\ G_1\ z_1\ s_1\ t_1 = \mathsf{natrec}\ G_2\ z_2\ s_2\ t_2 : G_1[t_1]}$$

$$\frac{\Gamma, \mathbb{N} \vdash G \qquad \Gamma \vdash z : G[\mathsf{zero}] \qquad \Gamma \vdash s : \Pi \mathbb{N} \triangleright (G \to G[\uparrow \mathsf{id}, \mathsf{suc}\ i_0])}{\Gamma \vdash \mathsf{natrec}\ G\ z\ s\ \mathsf{zero} = z : G[\mathsf{zero}]}$$

$$\frac{\Gamma, \mathbb{N} \vdash G \qquad \Gamma \vdash z : G[\mathsf{zero}] \qquad \Gamma \vdash s : \Pi \mathbb{N} \triangleright (G \to G[\uparrow \mathsf{id}, \mathsf{suc}\ i_0]) \qquad \Gamma \vdash t : \mathbb{N}}{\Gamma \vdash \mathsf{natrec}\ G\ z\ s\ (\mathsf{suc}\ t) = (s\ t)\ (\mathsf{natrec}\ G\ z\ s\ t) : G[\mathsf{suc}\ t]}$$

$$\frac{\Gamma \vdash A = B \qquad \Gamma \vdash t = u : \mathbb{1}}{\Gamma \vdash \mathtt{Emptyrec}\ A\ t = \mathtt{Emptyrec}\ B\ u : A} \qquad \frac{\Gamma \vdash t : \mathbb{1} \qquad \Gamma \vdash u : \mathbb{1}}{\Gamma \vdash t = u : \mathbb{1}}$$

5.3 Operational semantics

Weak-head reduction $\boxed{\Gamma \vdash A \longrightarrow B}$ and $\boxed{\Gamma \vdash t \longrightarrow u : A}$

$$\frac{\Gamma \vdash A \longrightarrow B : U}{\Gamma \vdash A \longrightarrow B} \qquad \frac{\Gamma \vdash t \longrightarrow u : A \qquad \Gamma \vdash A = B}{\Gamma \vdash t \longrightarrow u : B}$$

$$\frac{\Gamma \vdash f \longrightarrow g : \Pi F \blacktriangleright G \qquad \Gamma \vdash a : F}{\Gamma \vdash f\ a \longrightarrow g\ a : G[a]} \qquad \frac{\Gamma \vdash F \qquad \Gamma, F \vdash t : G \qquad \Gamma \vdash a : F}{\Gamma \vdash (\lambda t)\ a \longrightarrow t[a] : G[a]}$$

$$\frac{\Gamma \vdash F \qquad \Gamma, F \vdash G \qquad \Gamma \vdash t \longrightarrow u : \Sigma F \blacktriangleright G}{\Gamma \vdash \mathsf{fst}\ t \longrightarrow \mathsf{fst}\ u : F}$$

$$\frac{\Gamma \vdash F \qquad \Gamma, F \vdash G \qquad \Gamma \vdash t \longrightarrow u : \Sigma F \blacktriangleright G}{\Gamma \vdash \mathsf{snd}\ t \longrightarrow \mathsf{snd}\ u : G[\mathsf{fst}\ t]}$$

$$\frac{\Gamma \vdash F \qquad \Gamma, F \vdash G \qquad \Gamma \vdash t : F \qquad \Gamma \vdash u : G[t]}{\Gamma \vdash \mathsf{fst}\ \langle t, u \rangle \longrightarrow t : F}$$

$$\frac{\Gamma \vdash F \qquad \Gamma, F \vdash G \qquad \Gamma \vdash t : F \qquad \Gamma \vdash u : G[t]}{\Gamma \vdash \mathsf{snd}\ \langle t, u \rangle \longrightarrow u : G[\mathsf{fst}\ \langle t, u \rangle]}$$

$$\frac{\Gamma, \mathbb{N} \vdash G \qquad \Gamma \vdash z : G[\mathsf{zero}] \qquad \Gamma \vdash s : \Pi \mathbb{N} \blacktriangleright (G \to G[\uparrow \mathsf{id}, \mathsf{suc}\ i_0]) \qquad \Gamma \vdash t \longrightarrow u : \mathbb{N}}{\Gamma \vdash \mathsf{natrec}\ G\ z\ s\ t \longrightarrow \mathsf{natrec}\ G\ z\ s\ u : G[t]}$$

$$\frac{\Gamma, \mathbb{N} \vdash G \qquad \Gamma \vdash z : G[\mathsf{zero}] \qquad \Gamma \vdash s : \Pi \mathbb{N} \blacktriangleright (G \to G[\uparrow \mathsf{id}, \mathsf{suc}\ i_0])}{\Gamma \vdash \mathsf{natrec}\ G\ z\ s\ \mathsf{zero} \longrightarrow z : G[\mathsf{zero}]}$$

$$\frac{\Gamma, \mathbb{N} \vdash G \qquad \Gamma \vdash z : G[\mathsf{zero}] \qquad \Gamma \vdash s : \Pi \mathbb{N} \blacktriangleright (G \to G[\uparrow \mathsf{id}, \mathsf{suc}\ i_0]) \qquad \Gamma \vdash t : \mathbb{N}}{\Gamma \vdash \mathsf{natrec}\ G\ z\ s\ (\mathsf{suc}\ t) \longrightarrow (s\ t)\ (\mathsf{natrec}\ G\ z\ s\ t) : G[\mathsf{suc}\ t]}$$

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash t \longrightarrow u : \mathbb{0}}{\Gamma \vdash \mathtt{Emptyrec}\ A\ t \longrightarrow \mathtt{Emptyrec}\ A\ u : A}$$

5.4 Type conversion algorithm

Conversion of neutral terms $\boxed{\Gamma \vdash n \longleftrightarrow m : \bar{A}}$

$$\frac{\Gamma \vdash A \longrightarrow^{*} \bar{A} \qquad \Gamma \vdash n \mathrel{\hat{\Longleftrightarrow}} m : A}{\Gamma \vdash n \longleftrightarrow m : \bar{A}} \qquad \frac{\Gamma \vdash i_x : A \qquad x \equiv y}{\Gamma \vdash i_x \mathrel{\hat{\longleftrightarrow}} i_y : A}$$

$$\frac{\Gamma \vdash n \longleftrightarrow m : \Pi F \blacktriangleright G \qquad \Gamma \vdash t \mathrel{\hat{\Longleftrightarrow}} u : F}{\Gamma \vdash n\, t \mathrel{\hat{\longleftrightarrow}} m\, u : G[t]} \qquad \frac{\Gamma \vdash n \longleftrightarrow m : \Sigma F \blacktriangleright G}{\Gamma \vdash \mathtt{fst}\, n \mathrel{\hat{\longleftrightarrow}} \mathtt{fst}\, m : F}$$

$$\frac{\Gamma \vdash n \longleftrightarrow m : \Sigma F \blacktriangleright G}{\Gamma \vdash \mathtt{snd}\, n \mathrel{\hat{\longleftrightarrow}} \mathtt{snd}\, m : G[\mathtt{fst}\, n]}$$

$$\frac{\begin{array}{c}\Gamma, \mathbb{N} \vdash F \mathrel{\hat{\Longleftrightarrow}} G \\ \Gamma \vdash z \mathrel{\hat{\Longleftrightarrow}} z' : F[\mathtt{zero}] \qquad \Gamma \vdash s \mathrel{\hat{\Longleftrightarrow}} s' : \Pi \mathbb{N} \blacktriangleright (F \to F[\uparrow \mathtt{id}, \mathtt{suc}\ i_0]) \qquad \Gamma \vdash n \longleftrightarrow m : \mathbb{N}\end{array}}{\Gamma \vdash \mathtt{natrec}\, F\, z\, s\, n \mathrel{\hat{\longleftrightarrow}} \mathtt{natrec}\, G\, z'\, s'\, m : F[n]}$$

$$\frac{\Gamma \vdash F \mathrel{\hat{\Longleftrightarrow}} G \qquad \Gamma \vdash n \longleftrightarrow m : \mathbb{0}}{\Gamma \vdash \mathtt{Emptyrec}\, F\, n \mathrel{\hat{\longleftrightarrow}} \mathtt{Emptyrec}\, G\, m : F}$$

Conversion of types $\boxed{\Gamma \vdash \bar{A} \Longleftrightarrow \bar{B}}$

$$\frac{\Gamma \vdash A \longrightarrow^{*} \bar{A} \qquad \Gamma \vdash B \longrightarrow^{*} \bar{B} \qquad \Gamma \vdash \bar{A} \Longleftrightarrow \bar{B}}{\Gamma \vdash A \mathrel{\hat{\Longleftrightarrow}} B} \qquad \frac{\Gamma \vdash N \longleftrightarrow M : U}{\Gamma \vdash N \Longleftrightarrow M}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \bar{A} \Longleftrightarrow \bar{A}} \text{ (for } \bar{A} = U, \mathbb{N}, \mathbb{1}, \mathbb{0}\text{)} \qquad \frac{\Gamma \vdash F \mathrel{\hat{\Longleftrightarrow}} H \qquad \Gamma, F \vdash G \mathrel{\hat{\Longleftrightarrow}} E}{\Gamma \vdash \mathfrak{B} F \blacktriangleright G \Longleftrightarrow \mathfrak{B} H \blacktriangleright E} \text{ (for } \mathfrak{B} = \Pi, \Sigma\text{)}$$

Conversion of terms $\boxed{\Gamma \vdash \bar{t} \Longleftrightarrow \bar{u} : \bar{A}}$

$$\frac{\Gamma \vdash A \longrightarrow^* \bar{A} \qquad \Gamma \vdash t \longrightarrow^* \bar{t} : \bar{A} \qquad \Gamma \vdash u \longrightarrow^* \bar{u} : \bar{A} \qquad \Gamma \vdash \bar{t} \Longleftrightarrow \bar{u} : \bar{A}}{\Gamma \vdash t \mathrel{\widehat{\Leftrightarrow}} u : A}$$

$$\frac{\Gamma \vdash n \longleftrightarrow m : \bar{A}}{\Gamma \vdash n \Longleftrightarrow m : \bar{A}} \text{ (for } \bar{A} = \mathbb{N}, \mathbb{1}, \mathbb{0}\text{)} \qquad \frac{\Gamma \vdash n : \bar{A} \qquad \Gamma \vdash m : \bar{A} \qquad \Gamma \vdash n \longleftrightarrow m : \bar{B}}{\Gamma \vdash n \Longleftrightarrow m : \bar{A}}$$

$$\frac{\Gamma \vdash \bar{A} : U \qquad \Gamma \vdash \bar{B} : U \qquad \Gamma \vdash \bar{A} \Longleftrightarrow \bar{B}}{\Gamma \vdash \bar{A} \Longleftrightarrow \bar{B} : U}$$

$$\frac{\Gamma \vdash \bar{f} : \Pi F \triangleright G \qquad \Gamma \vdash \bar{g} : \Pi F \triangleright G \qquad \Gamma, F \vdash \bar{f}[\uparrow \mathsf{id}]\ i_0 \mathrel{\widehat{\Leftrightarrow}} \bar{g}[\uparrow \mathsf{id}]\ i_0 : G}{\Gamma \vdash \bar{f} \Longleftrightarrow \bar{g} : \Pi F \triangleright G}$$

$$\frac{\begin{array}{c}\Gamma \vdash \bar{p} : \Sigma F \triangleright G \\ \Gamma \vdash \bar{r} : \Sigma F \triangleright G \qquad \Gamma \vdash \mathsf{fst}\ \bar{p} \mathrel{\widehat{\Leftrightarrow}} \mathsf{fst}\ \bar{r} : F \qquad \Gamma \vdash \mathsf{snd}\ \bar{p} \mathrel{\widehat{\Leftrightarrow}} \mathsf{snd}\ \bar{r} : G[\mathsf{fst}\ \bar{p}]\end{array}}{\Gamma \vdash \bar{p} \Longleftrightarrow \bar{r} : \Sigma F \triangleright G}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{zero} \Longleftrightarrow \mathsf{zero} : \mathbb{N}} \qquad \frac{\Gamma \vdash t \mathrel{\widehat{\Leftrightarrow}} u : \mathbb{N}}{\Gamma \vdash \mathsf{suc}\ t \Longleftrightarrow \mathsf{suc}\ u : \mathbb{N}} \qquad \frac{\Gamma \vdash \bar{t} : \mathbb{1} \qquad \Gamma \vdash \bar{u} : \mathbb{1}}{\Gamma \vdash \bar{t} \Longleftrightarrow \bar{u} : \mathbb{1}}$$

5.5 Generic equality

Generic type equality $\boxed{\Gamma \vdash A \cong B}$

$$\frac{\Gamma \vdash A \cong B}{\Gamma \vdash A \equiv B} \qquad \frac{\Gamma \vdash A \cong B : U}{\Gamma \vdash A \cong B} \qquad \frac{\Gamma \vdash A \cong B}{\Gamma \vdash B \cong A} \qquad \frac{\Gamma \vdash A \cong B \qquad \Gamma \vdash B \cong C}{\Gamma \vdash A \cong C}$$

$$\frac{\vdash \Delta \qquad \rho : \Delta \leq \Gamma \qquad \Gamma \vdash A \cong B}{\Delta \vdash A[\rho] \cong B[\rho]} \qquad \frac{\Gamma \vdash A \longrightarrow^* \bar{A} \qquad \Gamma \vdash B \longrightarrow^* \bar{B} \qquad \Gamma \vdash \bar{A} \cong \bar{B}}{\Gamma \vdash A \cong B}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash A \cong A} \text{ (for } A = U, \mathbb{N}, \mathbb{1}, \mathbb{0}\text{)} \qquad \frac{\Gamma \vdash F \qquad \Gamma \vdash F \cong H \qquad \Gamma, F \vdash G \cong E}{\Gamma \vdash \mathfrak{B} F \triangleright G \cong \mathfrak{B} G \triangleright E} \text{ (for } \mathfrak{B} = \Pi, \Sigma\text{)}$$

Generic term equality $\boxed{\Gamma \vdash t \cong u : A}$

$$\frac{\Gamma \vdash t \sim u : A}{\Gamma \vdash t \cong u : A} \qquad \frac{\Gamma \vdash t \cong u : A}{\Gamma \vdash t \equiv u : A} \qquad \frac{\Gamma \vdash t \cong u : A}{\Gamma \vdash u \cong t : A} \qquad \frac{\Gamma \vdash t \cong u : A \qquad \Gamma \vdash u \cong v : A}{\Gamma \vdash t \cong v : A}$$

$$\frac{\Gamma \vdash t \cong u : A \qquad \Gamma \vdash A \equiv B}{\Gamma \vdash t \cong u : B} \qquad \frac{\vdash \Delta \qquad \rho : \Delta \leq \Gamma \qquad \Gamma \vdash t \cong u : A}{\Delta \vdash t[\rho] \cong u[\rho] : A[\rho]}$$

$$\frac{\Gamma \vdash A \longrightarrow^{*} \bar{A} \qquad \Gamma \vdash t \longrightarrow^{*} \bar{t} : \bar{A} \qquad \Gamma \vdash u \longrightarrow^{*} \bar{u} : \bar{A} \qquad \Gamma \vdash \bar{t} \cong \bar{u} : \bar{A}}{\Gamma \vdash t \cong u : A}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash A \cong A : U} \text{ (for } A = \mathbb{N}, \mathbb{1}, \mathbb{0}\text{)}$$

$$\frac{\Gamma \vdash F \qquad \Gamma \vdash F \cong H : U \qquad \Gamma, F \vdash G \cong E : U}{\Gamma \vdash \mathfrak{B} F \triangleright G \cong \mathfrak{B} G \triangleright E : U} \text{ (for } \mathfrak{B} = \Pi, \Sigma\text{)}$$

$$\frac{\Gamma \vdash F \qquad \Gamma \vdash \bar{f} : \Pi F \triangleright G \qquad \Gamma \vdash \bar{g} : \Pi F \triangleright G \qquad \Gamma, F \vdash \bar{f}[\uparrow \texttt{id}]\ i_0 \cong \bar{g}[\uparrow \texttt{id}]\ i_0 : G}{\Gamma \vdash f \cong g : \Pi F \triangleright G}$$

$$\frac{\Gamma \vdash \bar{t} : \Sigma F \triangleright G \qquad \Gamma \vdash \bar{u} : \Sigma F \triangleright G \qquad \begin{array}{c} \Gamma \vdash F \qquad \Gamma, F \vdash G \\ \Gamma \vdash \mathsf{fst}\ \bar{t} \cong \mathsf{fst}\ \bar{u} : F \end{array} \qquad \Gamma \vdash \mathsf{snd}\ \bar{t} \cong \mathsf{snd}\ \bar{u} : G[\mathsf{fst}\ \bar{t}]}{\Gamma \vdash t \cong u : \Sigma F \triangleright G}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{zero} \cong \mathsf{zero} : \mathbb{N}} \qquad \frac{\Gamma \vdash t \cong u : \mathbb{N}}{\Gamma \vdash \mathsf{suc}\ t \cong \mathsf{suc}\ u : \mathbb{N}} \qquad \frac{\Gamma \vdash t : \mathbb{1} \qquad \Gamma \vdash u : \mathbb{1}}{\Gamma \vdash t \cong u : \mathbb{1}}$$

Generic neutral term equality $\boxed{\Gamma \vdash n \sim m : A}$

$$\frac{\Gamma \vdash n \sim m : A}{\Gamma \vdash m \sim n : A} \qquad \frac{\Gamma \vdash n \sim m : A \qquad \Gamma \vdash m \sim k : A}{\Gamma \vdash n \sim k : A} \qquad \frac{\Gamma \vdash n \sim m : A \qquad \Gamma \vdash A \equiv B}{\Gamma \vdash n \sim m : B}$$

$$\frac{\vdash \Delta \qquad \rho : \Delta \leq \Gamma \qquad \Gamma \vdash n \sim m : A}{\Delta \vdash n[\rho] \sim m[\rho] : A[\rho]} \qquad \frac{\Gamma \vdash x : A}{\Gamma \vdash x \sim x : A}$$

$$\frac{\Gamma \vdash n \sim m : \Pi F \triangleright G \qquad \Gamma \vdash t \cong u : F}{\Gamma \vdash n\ t \sim m\ u : G[t]} \qquad \frac{\Gamma \vdash F \qquad \Gamma, F \vdash G \qquad \Gamma \vdash n \sim m : \Sigma F \triangleright G}{\Gamma \vdash \mathsf{fst}\ n \sim \mathsf{fst}\ m : F}$$

$$\frac{\Gamma \vdash F \qquad \Gamma, F \vdash G \qquad \Gamma \vdash n \sim m : \Sigma F \triangleright G}{\Gamma \vdash \mathsf{snd}\ n \sim \mathsf{snd}\ m : G[\mathsf{fst}\ n]}$$

$$\frac{\begin{array}{c}\Gamma, \mathbb{N} \vdash F \cong F' \\ \Gamma \vdash z \cong z' : F[\mathtt{zero}] \qquad \Gamma \vdash s \cong s' : \Pi \mathbb{N} \triangleright (F \to F[\uparrow \mathtt{id}, \mathtt{suc}\ i_0]) \qquad \Gamma \vdash n \sim n' : \mathbb{N}\end{array}}{\Gamma \vdash \mathtt{natrec}\ F\ z\ s\ n \sim \mathtt{natrec}\ F'\ z'\ s'\ n' : F[n]}$$

$$\frac{\Gamma \vdash F \cong G \qquad \Gamma \vdash n \sim m : \mathbb{0}}{\Gamma \vdash \mathtt{Emptyrec}\ F\ n \sim \mathtt{Emptyrec}\ G\ m : F}$$

References

[1] Andreas Abel. *Normalization by Evaluation: Dependent Types and Impredicativity.* PhD thesis, Ludwig-Maximilians-Universität München, 2013.

[2] Andreas Abel and Thierry Coquand. Failure of normalization in impredicative type theory with proof-irrelevant propositional equality. *CoRR*, abs/1911.08174, 2019. URL http://arxiv.org/abs/1911.08174.

[3] Andreas Abel and Gabriel Scherer. On irrelevance and algorithmic equality in predicative type theory. *Logical Methods in Computer Science*, 8(1), 2012. doi: 10.2168/LMCS-8(1:29)2012. URL https://doi.org/10.2168/LMCS-8(1:29)2012.

[4] Andreas Abel, Klaus Aehlig, and Peter Dybjer. Normalization by evaluation for martin-löf type theory with one universe. In Marcelo Fiore, editor, *Proceedings of the 23rd Conference on the Mathematical Foundations of Programming Semantics, MFPS 2007, New Orleans, LA, USA, April 11-14, 2007*, volume 173 of *Electronic Notes in Theoretical Computer Science*, pages 17–39. Elsevier, 2007. doi: 10.1016/j.entcs.2007.02.025. URL https://doi.org/10.1016/j.entcs.2007.02.025.

[5] Andreas Abel, Thierry Coquand, and Peter Dybjer. Normalization by evaluation for martin-lof type theory with typed equality judgements. In *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-12 July 2007, Wroclaw, Poland, Proceedings*, pages 3–12. IEEE Computer Society, 2007. ISBN 0-7695-2908-9. doi: 10.1109/LICS.2007.33. URL https://doi.org/10.1109/LICS.2007.33.

[6] Andreas Abel, Thierry Coquand, and Miguel Pagano. A modular type-checking algorithm for type theory with singleton types and proof irrelevance. *Logical Methods in Computer Science*, 7(2), 2011. doi: 10.2168/LMCS-7(2:4)2011. URL https://doi.org/10.2168/LMCS-7(2:4)2011.

[7] Andreas Abel, Thierry Coquand, and Bassel Mannaa. On the decidability of conversion in type theory. 05 2016.

[8] Andreas Abel, Joakim Öhman, and Andrea Vezzosi. Decidability of conversion for type theory in type theory. *PACMPL*, 2(POPL):23:1–23:29, 2018. doi: 10.1145/3158111. URL https://doi.org/10.1145/3158111.

[9] Andreas Abel, Guillaume Allais, Aliya Hameer, Brigitte Pientka, Alberto Momigliano, Steven Schäfer, and Kathrin Stark. Poplmark reloaded: Mechanizing proofs by logical relations. *J. Funct. Program.*, 29:e19, 2019. doi: 10.1017/S0956796819000170. URL https://doi.org/10.1017/S0956796819000170.

[10] Thorsten Altenkirch and Ambrus Kaposi. Normalisation by evaluation for type theory, in type theory. *Logical Methods in Computer Science*, 13(4), 2017. doi: 10.23638/LMCS-13(4:1)2017. URL https://doi.org/10.23638/LMCS-13(4:1)2017.

[11] Thorsten Altenkirch and Conor McBride. Towards observational type theory. 2006.

[12] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Categorical reconstruction of a reduction free normalization proof. In David H. Pitt, David E. Rydeheard, and Peter T. Johnstone, editors, *Category Theory and Computer Science, 6th International Conference, CTCS '95, Cambridge, UK, August 7-11, 1995, Proceedings*, volume 953 of *Lecture Notes*

in Computer Science, pages 182–199. Springer, 1995. ISBN 3-540-60164-3. doi: 10.1007/3-540-60164-3_27. URL https://doi.org/10.1007/3-540-60164-3_27.

[13] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In Aaron Stump and Hongwei Xi, editors, *Proceedings of the ACM Workshop Programming Languages meets Program Verification, PLPV 2007, Freiburg, Germany, October 5, 2007*, pages 57–68. ACM, 2007. ISBN 978-1-59593-677-6. doi: 10.1145/1292597.1292608. URL https://doi.org/10.1145/1292597.1292608.

[14] Thorsten Altenkirch, Simon Boulier, Ambrus Kaposi, and Nicolas Tabareau. Setoid type theory - A syntactic translation. In Graham Hutton, editor, *Mathematics of Program Construction - 13th International Conference, MPC 2019, Porto, Portugal, October 7-9, 2019, Proceedings*, volume 11825 of *Lecture Notes in Computer Science*, pages 155–196. Springer, 2019. ISBN 978-3-030-33635-6. doi: 10.1007/978-3-030-33636-3_7. URL https://doi.org/10.1007/978-3-030-33636-3_7.

[15] 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*, pages 203–211. IEEE Computer Society, 1991. ISBN 0-8186-2230-X. doi: 10.1109/LICS.1991.151645. URL https://doi.org/10.1109/LICS.1991.151645.

[16] Guillaume Brunerie. A formalized proof of a version of the initiality conjecture. https://github.com/guillaumebrunerie/initiality, 2020.

[17] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: A constructive interpretation of the univalence axiom. In Tarmo Uustalu, editor, *21st International Conference on Types for Proofs and Programs, TYPES 2015, May 18-21, 2015, Tallinn, Estonia*, volume 69 of *LIPIcs*, pages 5:1–5:34. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015. ISBN 978-3-95977-030-9. doi: 10.4230/LIPIcs.TYPES.2015.5. URL https://doi.org/10.4230/LIPIcs.TYPES.2015.5.

[18] Thierry Coquand. *An algorithm for testing conversion in type theory*, page 255–279. Cambridge University Press, 1991. doi: 10.1017/CBO9780511569807.011.

[19] Thierry Coquand. Canonicity and normalization for dependent type theory. *CoRR*, 2018. URL http://arxiv.org/abs/1810.09367v1.

[20] Karl Crary and Robert Harper. Syntactic logical relations for polymorphic and recursive types. *Electron. Notes Theor. Comput. Sci.*, 172:259–299, 2007. doi: 10.1016/j.entcs.2007.02.010. URL https://doi.org/10.1016/j.entcs.2007.02.010.

[21] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, volume 9195 of *Lecture Notes in Computer Science*, pages 378–388. Springer, 2015. ISBN 978-3-319-21400-9. doi: 10.1007/978-3-319-21401-6_26. URL https://doi.org/10.1007/978-3-319-21401-6_26.

[22] Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *J. Symb. Log.*, 65(2):525–549, 2000. doi: 10.2307/2586554. URL https://doi.org/10.2307/2586554.

[23] Peter Dybjer and Anton Setzer. Induction-recursion and initial algebras. *Ann. Pure Appl. Log.*, 124(1-3):1–47, 2003. doi: 10.1016/S0168-0072(02)00096-9. URL https://doi.org/10.1016/S0168-0072(02)00096-9.

[24] Marcelo P. Fiore. Semantic analysis of normalisation by evaluation for typed lambda calculus. In *Proceedings of the 4th international ACM SIGPLAN conference on Principles and practice of declarative programming, October 6-8, 2002, Pittsburgh, PA, USA (Affiliated with PLI 2002)*, pages 26–37. ACM, 2002. doi: 10.1145/571157.571161. URL https://doi.org/10.1145/571157.571161.

[25] Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. Definitional proof-irrelevance without K. *Proc. ACM Program. Lang.*, 3(POPL):3:1–3:28, 2019. doi: 10.1145/3290316. URL https://doi.org/10.1145/3290316.

[26] Gaëtan Gilbert. Add an empty type. https://github.com/mr-ohman/logrel-mltt/pull/3, 2018.

[27] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types.* Cambridge University Press, 1989. ISBN 0521371813.

[28] Daniel Gratzer, Jonathan Sterling, and Lars Birkedal. Implementing a modal dependent type theory. *PACMPL*, 3(ICFP):107:1–107:29, 2019. doi: 10.1145/3341711. URL https://doi.org/10.1145/3341711.

[29] Benjamin Grégoire and Assia Mahboubi. Proving equalities in a commutative ring done right in coq. In Joe Hurd and Thomas F. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, volume 3603 of *Lecture Notes in Computer Science*, pages 98–113. Springer, 2005. ISBN 3-540-28372-2. doi: 10.1007/11541868_7. URL https://doi.org/10.1007/11541868_7.

[30] Robert Harper. Constructing type systems over an operational semantics. *J. Symb. Comput.*, 14(1):71–84, 1992. doi: 10.1016/0747-7171(92)90026-Z. URL https://doi.org/10.1016/0747-7171(92)90026-Z.

[31] Robert Harper and Frank Pfenning. On equivalence and canonical forms in the LF type theory. *ACM Trans. Comput. Log.*, 6(1):61–101, 2005. doi: 10.1145/1042038.1042041. URL https://doi.org/10.1145/1042038.1042041.

[32] Martin Hofmann. *Extensional constructs in intensional type theory.* CPHC/BCS distinguished dissertations. Springer, 1997. ISBN 978-3-540-76121-1.

[33] Per Martin-Löf. Constructive mathematics and computer programming [and discussion]. volume 312, pages 501–518. The Royal Society, 1984. URL http://www.jstor.org/stable/37448.

[34] nLab authors. negative type. http://ncatlab.org/nlab/show/negative%20type, June 2020. Revision 6.

[35] nLab authors. positive type. http://ncatlab.org/nlab/show/positive%20type, June 2020. Revision 6.

[36] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, 2002. ISBN 978-0-262-16209-8.

[37] Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press. MIT Press, 2005. ISBN 978-0-262-16228-9. URL `https://books.google.pl/books?id=A5ic1MPTvVsC`.

[38] Gordon Plotkin. Lambda-definability and logical relations. 1973.

[39] Richard Statman. λ-definable functionals and $\beta\eta$ conversion. *Arch. Math. Log.*, 23(1): 21–26, 1983. doi: 10.1007/BF02023009. URL `https://doi.org/10.1007/BF02023009`.

[40] Jonathan Sterling. Algebraic type theory and universe hierarchies. *CoRR*, 2019. URL `http://arxiv.org/abs/1902.08848v1`.

[41] Jonathan Sterling. An ok version of type theory. 2020. URL `http://www.jonmsterling.com/pdfs/ok-type-theory.pdf`.

[42] Jonathan Sterling and Bas Spitters. Normalization by gluing for free λ-theories. *CoRR*, 2018. URL `http://arxiv.org/abs/1809.08646v1`.

[43] Jonathan Sterling, Carlo Angiuli, and Daniel Gratzer. Cubical syntax for reflection-free extensional equality. In Herman Geuvers, editor, *4th International Conference on Formal Structures for Computation and Deduction, FSCD 2019, June 24-30, 2019, Dortmund, Germany*, volume 131 of *LIPIcs*, pages 31:1–31:25. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. ISBN 978-3-95977-107-8. doi: 10.4230/LIPIcs.FSCD.2019.31. URL `https://doi.org/10.4230/LIPIcs.FSCD.2019.31`.

[44] Jonathan Sterling, Carlo Angiuli, and Daniel Gratzer. A cubical language for bishop sets. *CoRR*, abs/2003.01491, 2020. URL `https://arxiv.org/abs/2003.01491`.

[45] William W. Tait. Intensional interpretations of functionals of finite type I. *J. Symb. Log.*, 32(2):198–212, 1967. doi: 10.2307/2271658. URL `https://doi.org/10.2307/2271658`.

[46] Taichi Uemura. A general framework for the semantics of type theory. *CoRR*, abs/1904.04097, 2019. URL `http://arxiv.org/abs/1904.04097`.

[47] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `https://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.