Verifying a Semantic $\beta\eta$-Conversion Test for Martin-Löf Type Theory (extended version)

Andreas Abel,[1] Thierry Coquand,[2] and Peter Dybjer[2]*

[1] Department of Computer Science, Ludwig-Maximilians-University, Munich
abel@tcs.ifi.lmu.de
[2] Department of Computer Science, Chalmers University of Technology
coquand,peterd@cs.chalmers.se

Abstract. Type-checking algorithms for dependent type theories often rely on the interpretation of terms in some semantic domain of *values* when checking equalities. Here we analyze a version of Coquand's algorithm for checking the $\beta\eta$-equality of such semantic values in a theory with a predicative universe hierarchy and large elimination rules. Although this algorithm does not rely on normalization by evaluation explicitly, we show that similar ideas can be employed for its verification. In particular, our proof uses the new notions of *contextual* reification and *strong semantic equality*.

The algorithm is part of a bi-directional type checking algorithm which checks whether a normal term has a certain semantic type, a technique used in the proof assistants Agda and Epigram. We work with an abstract notion of semantic domain in order to accommodate a variety of possible implementation techniques, such as normal forms, weak head normal forms, closures, and compiled code. Our aim is to get closer than previous work to verifying the type-checking algorithms which are actually used in practice.

1 Introduction

Proof assistants based on dependent type theory have now been around for about 25 years. The most prominent representative, Coq [INR07], has become a mature system. It can now be used for larger scale program development and verification, as Leroy's ongoing implementation of a verified compiler shows [Ler06]. Functional programmers have also become more and more interested in using dependent types to ensure program and data structure invariants. New functional languages with dependent types such as Agda 2 [Nor07] and Epigram 2 [CAM07] enjoy increasing popularity.

Although many questions about properties of dependent type theories have been settled in the 1990s, some problems are still waiting for a satisfactory solution. One example is the treatment of equality in implementations of proof assistants. When we check that a dependently typed program is well-typed,

* Research partially supported by the EU coordination action *TYPES* (510996) and the project *TLCA* of Vetenskapsrådet.

we may need to test whether two types are definitionally equal (convertible). Although it is of course impossible for a system to recognize all semantically equal types, a user will feel more comfortable if it can recognize as many as possible. Whenever it fails the user has to resort to *proving* manually that the types are equal. This has the additional drawback of introducing proof-objects for these equalities. In recent years there has therefore been a move from β-equality (computational equality) to the stronger $\beta\eta$-equality (computational and extensional equality).

Recently, algorithms for testing $\beta\eta$-equality have been formulated and verified by the authors both for an untyped notion of conversion [AAD07] and for typed equality judgements [ACD07]. These algorithms use the technique of normalization by evaluation (NbE). However, the algorithms used by proof assistants such as Agda and Epigram [CAM07], use Coquand's $\beta\eta$-conversion test for semantic "values", and do not employ the NbE-technique of the above-mentioned papers. Moreover, there is a gap between algorithms on paper and their actual implementation. Proofs on paper are often informal about the treatment of variable names, and they tend to represent values as pieces of abstract syntax. Besides Pollack's [Pol94], Coquand's algorithm [Coq96] is a notable exception: values are represented as closures, and the algorithm explicitly deals with α-equivalence by replacing variables by numbers (de Bruijn levels).

We here continue the work of the second author and verify an implementation of the $\beta\eta$-conversion test close to the one used in practice. In particular:

- Equality is checked incrementally, and not by full normalization followed by a test for syntactical identity.
- The representation of values is abstract. We only require that they form a syntactical applicative structure. In this way, several possible implementations, such as normal forms, closures, and abstract machine code, are covered by our verification.
- The verification approach is extensible: Although we only spell out the proofs for a core of type theory with predicative universes, our development extends to richer languages. We can for example include a unit type, Σ types, proof irrelevance, and inductive types with large eliminations.

Overview. In Sec. 2 we present an abstract type and equality checking algorithm, which only assumes that the domain of values forms a *syntactical applicative structure*. In Sec. 3 inference rules for typing and type equality are given for a version of Martin-Löf type theory with explicit substitutions. An outline of the verification is given in Sec. 4 together with a definition of contextual reification, our main tool for verification. Using contextual reification, an alternative equality test can be formulated, which is shown complete in Sec. 5, by construction of a Kripke model, and proven sound in Sec. 6 via a Kripke logical relation. Completeness of the original algorithm then follows easily in Sec. 7. For soundness, we have to introduce a Kripke logical relation and the concept of strong semantic equality in Sec. 8. In Sec. 9 we discuss the problem of termination of the equality algorithm, which remains open.

2 Semantic Type and Equality Checking

We consider *dependently typed programs* p to be given as lists of the form

$$\begin{array}{lll} x_0 & : V_0 & = v_0 \\ & \vdots & \\ x_{n-1} & : V_{n-1} & = v_{n-1} \end{array}$$

where x_i is a identifier, V_i its type, and v_i the definition of x_i for $i < n$. (Typically x_i will be a function identifier and v_i the function definition.) The identifiers are not defined simultaneously (which would correspond to mutual recursion), but one-after-another. Hence, V_i and v_i may only refer to previously defined identifiers x_j for $j < i$. A program is *type correct* if each V_i is a well-formed type and each v_i is a term of type V_i. To establish this, the type and definition of the previously defined identifiers may be used. It is reasonable to assume and easy to check that the global identifiers x_i are all distinct; global declarations should not be shadowed. However, local shadowing is allowed; the identifiers x_i may be reused in some of the v_j or V_j. Our type checking algorithm handles shadowing correctly without any informal use of α-conversion.

Dependent types need to be evaluated during type checking. Thus it is common to store them in evaluated form. Without going into further details now, let $v\rho$ ($V\rho$) denote the *evaluation* of term v (type V) in environment ρ. The *environment* maps already checked identifiers to their values. A typing *context* Δ maps already checked identifiers to their types (in evaluated form). Checking a program starts in an empty environment ρ_0 and an empty typing context Δ_0. For $i = 0, \ldots, n-1$, we execute the following steps:

1. Check that V_i is a well-formed type in the current context Δ_i.
2. Evaluate V_i in the current environment: $X_i = V_i\rho_i$.
3. Check that v_i is of type X_i in the current context. This test is written $\Delta_i \vdash v_i\ \delta_{\mathsf{id}} \Uparrow X_i$, where δ_{id} is the identity map on names.
4. Evaluate v_i and extend the current environment by binding x_i to the result: $\rho_{i+1} = (\rho_i, x_i = v_i\rho_i)$.
5. Extend the current context: $\Delta_{i+1} = \Delta, x_i\!:\!X_i$.

The details of type checking depend on the language. We here show how to verify an algorithm for a core language with dependent function types and predicative universes. This language is enriched with natural numbers and primitive recursion. However, the algorithms and proofs in this work directly extend to dependent tuples (Σ and unit type).

2.1 Syntax

Expressions r, s, t are formed from variables x and constants c by application $r\,s$ and function abstraction λxt. The types V_i and terms v_i of a program $p \equiv (x_i :$

$V_i = v_i)_i$ must be in normal form.

$\mathsf{Var} \ni x, y, z$	$::= \dots, \mathsf{x}_1, \mathsf{x}_2, \dots$	variables
$\mathsf{Const} \ni c$	$::= \mathsf{N} \mid \mathsf{z} \mid \mathsf{s} \mid \mathsf{rec} \mid \mathsf{Fun} \mid \mathsf{Set}_i$	constants ($i \in \mathbb{N}$)
$\mathsf{Exp} \ni r, s, t$	$::= c \mid x \mid \lambda x t \mid r\, s$	expressions
$\mathsf{Nf} \ni v, w, V, W$	$::= u \mid \lambda x v \mid \mathsf{N} \mid \mathsf{z} \mid \mathsf{s}\, v \mid \mathsf{Fun}\, V\, \lambda x W \mid \mathsf{Set}_i$	$\beta\iota$-normal expressions
$\mathsf{Ne} \ni u$	$::= x \mid u\, v \mid \mathsf{rec}\, V\, v_z\, v_s\, u$	neutral expressions

The set Var of variable identifiers contains, among others, the special variables $\mathsf{x}_1, \mathsf{x}_2, \dots$ which are called *de Bruijn levels*. To aid the reader, we use the letters A, B, C, V, W for expressions which are to be understood as types and r, s, t, u, v, w for terms. Dependent function types, usually written $\Pi x : A. B$, are written $\mathsf{Fun}\, A\, \lambda x B$. When B does not depend on x we have a non-dependent function type and write $A \to B$.

An expression like $\mathsf{Fun}\, A\, \lambda x B$ is parsed as $(\mathsf{Fun}\, A)\, (\lambda x B)$; application is left-associative. To save on parentheses, we introduce the notation $\lambda x. t$ where the dot opens a parenthesis which closes as far to the right as syntactically possible. For instance $\lambda x. r\, s$ is short for $\lambda x\, (r\, s)$, whereas $\lambda x r\, s$ means $(\lambda x r)\, s$.

We use the abbreviation $\mathsf{Rec}\,(C)$ for the type of the primitive recursion combinator $\mathsf{rec}\,(C)$ with the result type $C\, n$ depending on $n : \mathsf{N}$.

$$\mathsf{Rec}\,(C) = C\, \mathsf{z} \to (\mathsf{Fun}\ \mathsf{N}\ (\lambda n. C\, n \to C\, (\mathsf{s}\, n))) \to \mathsf{Fun}\, \mathsf{N}\, C.$$

The *hello world* program of dependent types, the polymorphic identity, becomes in our notation

$$\mathsf{id} : \mathsf{Fun}\ \mathsf{Set}_0\ \lambda A. A \to A = \lambda A \lambda a\, a.$$

The predicative universes Set_i are types of types. A well-formed type $V : \mathsf{Set}_i$ lives in universe i and above. A universe Set_i lives in higher universes Set_j, $j > i$.

2.2 Values

In implementations of dependently typed languages, different representations of values have been chosen: Twelf [PS99] uses de Bruijn terms with explicit substitutions; Agda 2 [Nor07] de Bruijn terms in normal form; and Epigram 2 [CAM07] higher-order abstract syntax. Furthermore, the second author has suggested to use closures [Coq96]. In this article, we abstract over several possible representations, by considering a *syntactical applicative structure with atoms.*

Applicative structure with atoms. This is an applicative structure $(\mathsf{D}, \_ \cdot \_)$ which includes all variables and constants as atoms ($\mathsf{Var} \cup \mathsf{Const} \subseteq \mathsf{D}$). Elements of D are denoted by d, e, f, X, Y, Z, E, F and called *values* or *objects. Neutral values* are given inductively by $e, E ::= x \mid e \cdot d \mid \mathsf{rec} \cdot F \cdot d_z \cdot d_s \cdot e$. Neutral application is injective: If $e \cdot d = e' \cdot d'$, then $e = e'$ and $d = d'$. Similarly, if $\mathsf{rec} \cdot F \cdot d_z \cdot d_s \cdot e = \mathsf{rec} \cdot F' \cdot d'_z \cdot d'_s \cdot e'$ then $F = F'$, $d_z = d'_z$, $d_s = d'_s$, and $e = e'$. Neutral values are distinct, for instance, $x \cdot \boldsymbol{d} \neq y \cdot \boldsymbol{d'} \neq \mathsf{rec} \cdot F \cdot d_z \cdot d_s \cdot e$. We will sometimes write application as juxtaposition, especially in neutral and constructed values.

The constants Set_i, N, and z are constructors of arity 0, s is a constructor of arity 1 and Fun is a constructor of arity 2. Constructors are injective, thus, $\mathsf{Fun}\,X\,F = \mathsf{Fun}\,X'\,F'$ implies $X = X'$ and $F = F'$. *Constructed values*, i. e., of the form $c\,\boldsymbol{d}$, are distinguished from each other and from neutral values, e.g., $\mathsf{z} \neq \mathsf{s}\,d \neq x\,d'$. The constant rec implements primitive recursion via the laws:

$$\begin{array}{ll} \textsc{app-rec-z} & \mathsf{rec}\,F\,d_z\,d_s \cdot \mathsf{z} = d_z \\ \textsc{app-rec-s} & \mathsf{rec}\,F\,d_z\,d_s \cdot (\mathsf{s}\,d) = d_s \cdot d \cdot (\mathsf{rec}\,F\,d_z\,d_s \cdot d) \end{array}$$

It is decidable whether an object is neutral, constructed, or neither. If an object is neutral, we can extract the head variable and the arguments, and similar for constructed objects.

Syntactical applicative structure with atoms. We enrich the applicative structure with an evaluation operation $t\rho \in \mathsf{D}$ for expressions $t \in \mathsf{Exp}$ in an environment $\rho : \mathsf{Var} \rightharpoonup \mathsf{D}$. Let ρ_{id} denote the identity environment. The following axioms must hold for evaluation.

$$\begin{array}{ll} \textsc{eval-c} & c\rho = c \\ \textsc{eval-var} & x\rho = \rho(x) \\ \textsc{eval-fun-e} & (r\,s)\rho = r\rho \cdot s\rho \\ \textsc{app-fun} & (\lambda xt)\rho \cdot d = t(\rho, x{=}d) \end{array}$$

An applicative structure which satisfies these equations is called a *syntactical applicative structure* $(\mathsf{D}, \_\cdot\_, \_\_)$. Barendregt [Bar84, 5.3.1.] adds a sanity condition that the evaluation of an expression may only depend on the valuations of its free variables, but we will not require it. All syntactical λ-models [BL84] [Bar84, 5.3.2.(ii)] are instances of syntactical applicative structures, yet they must additionally fulfill weak extensionality (ξ).

An instance: closures. There are applicative structures which are neither λ-models nor combinatory algebras, for example the representation of values by *closures*. Values are given by the grammar

$$\begin{aligned} \mathsf{D} \ni d &::= [\lambda xt]\rho \mid e \\ e &::= c \mid x \mid e\,d \end{aligned}$$

where $[\lambda xt]\rho$ is a closure such that ρ provides a value for each free variable in λxt. Evaluation does not proceed under binders; it is given by the above axioms plus:

$$\begin{array}{ll} \textsc{eval-fun-i} & (\lambda xt)\rho = [\lambda xt]\rho \\ \textsc{app-ne} & e \cdot d = e\,d \end{array}$$

Closures are a standard tool for building interpreters for λ-calculi; the second author has used them to implement a type checker [Coq96]. While for the soundness proof he requires weak extensionality in D, we will not; instead, extensionality of functions in Type Theory is proven via a Kripke model (Section 5).

2.3 Type Checking

In this section, we present a bidirectional type-checking algorithm [PT98] which *checks* a normal term against a type and *infers* the type of a normal expression. In the dependently typed setting, where types may contain computations, the principled approach is to keep types in evaluated form. During the course of type-checking we will have to evaluate terms (see rule INF-FUN-E below). To avoid non-termination, it is crucial to *only evaluate terms which have already been type checked*.

We are ready to present the semantic type checking algorithm, where "semantic" refers to the fact that types are values in D and not type expressions. As usual we describe it using inference rules. These can be read as the clauses of logic programs and we specify the modes (input and output). Note that the modes are not part of the mathematical definition of the inductive judgements—they only describe how the judgements should be executed. Since the rules are deterministic, they also describe a functional implementation of type checking, which can be obtained mechanically from the rules.

In the following definitions δ ranges over special environments, *renamings*, which are finite maps from variables to de Bruijn levels. As before, $t\delta$ denotes the evaluation of t in environment δ.

Semantic (typing) contexts are given by the grammar $\Delta ::= \diamond \mid \Delta, x : X$, where $x \notin \mathsf{dom}(\Delta)$. If $(x : X) \in \Delta$ then $\Delta(x) = X$. We write x_Δ for the first de Bruijn level which is not used in Δ, $\mathsf{x}_{\Delta+1}$ for the next one, etc.

Type checking algorithm. We define bidirectional type checking of normal terms and well-formedness checking of normal types by the following three judgements. Herein, $\Delta \in \mathsf{SemCxt}$, $u \in \mathsf{Ne}$, $v, V \in \mathsf{Nf}$, $\delta \in \mathsf{Var} \rightharpoonup \mathsf{Var}$, $X \in \mathsf{D}$ and $i \in \mathbb{N}$.

$\Delta \vdash u\ \delta \Downarrow X$	the type of neutral u is inferred as X
$\Delta \vdash v\ \delta \Uparrow X$	normal v checks against type X
$\Delta \vdash V\ \delta \Uparrow \mathsf{Set} \rightsquigarrow i$	V is a well-formed type of inferred level i.

In all judgements we maintain the invariant that Δ assigns types to the free variables in X and $\Delta \circ \delta$ to the free variables in u, v, V.

Type inference $\Delta \vdash u\ \delta \Downarrow X$. (Inputs: Δ, u, δ. Output: type X of u or fail.)

$$\textsc{inf-var}\ \frac{}{\Delta \vdash x\ \delta \Downarrow \Delta(x\delta)}$$

$$\textsc{inf-fun-e}\ \frac{\Delta \vdash u\ \delta \Downarrow \mathsf{Fun}\, X F \qquad \Delta \vdash v\ \delta \Uparrow X}{\Delta \vdash (u\, v)\ \delta \Downarrow F \cdot v\delta}$$

The type of a variable x under renaming δ is just looked up in the context. When computing the type of an application $u\, v$ from the type $\mathsf{Fun}\, X F$ of the function part we need to apply F to the evaluated argument part $v\delta$ (dependent function application). At this point, it is crucial that we have type-checked v already, otherwise the application $F \cdot v\delta$ could diverge.

Type checking $\Delta \vdash v\ \delta \Uparrow X$. (Inputs: Δ, v, δ, X. Output: succeed or fail.)

$$\text{CHK-FUN-I}\ \frac{\Delta, \mathsf{x}_\Delta \!:\! X \vdash v\ (\delta, x\!=\!\mathsf{x}_\Delta) \Uparrow F \cdot \mathsf{x}_\Delta}{\Delta \vdash (\lambda x v)\ \delta \Uparrow \mathsf{Fun}\, X F}$$

$$\text{CHK-SET}\ \frac{\Delta \vdash V\ \delta \Uparrow \mathsf{Set} \rightsquigarrow i}{\Delta \vdash V\ \delta \Uparrow \mathsf{Set}_j}\ i \leq j$$

$$\text{CHK-INF}\ \frac{\Delta \vdash u\ \delta \Downarrow X \qquad \Delta \vdash X = X' \Uparrow \mathsf{Set} \rightsquigarrow i}{\Delta \vdash u\ \delta \Uparrow X'}$$

When checking an abstraction $\lambda x v$ against a dependent function type value $\mathsf{Fun}\, X F$, we rename x to the next free de Bruijn level x_Δ and check the abstraction body v against $F \cdot \mathsf{x}_\Delta$ in the extended context which binds the abstracted variable to the domain X. To check a neutral term u against X', we infer the type X of u and compare X and X'. The implementation and verification of this comparison will occupy our attention for the remainder of this article.

Type well-formedness $\Delta \vdash V\ \delta \Uparrow \mathsf{Set} \rightsquigarrow i$. (Inputs: Δ, V, δ. Output: universe level i of V or fail.)

$$\text{CHK-INF-F}\ \frac{\Delta \vdash V\ \delta \Downarrow \mathsf{Set}_i}{\Delta \vdash V\ \delta \Uparrow \mathsf{Set} \rightsquigarrow i} \qquad \text{CHK-SET-F}\ \frac{}{\Delta \vdash \mathsf{Set}_i\ \delta \Uparrow \mathsf{Set} \rightsquigarrow i+1}$$

$$\text{CHK-FUN-F}\ \frac{\Delta \vdash V\ \delta \Uparrow \mathsf{Set} \rightsquigarrow i \qquad \Delta,\ \mathsf{x}_\Delta \!:\! V\delta \vdash W\ (\delta, y\!=\!\mathsf{x}_\Delta) \Uparrow \mathsf{Set} \rightsquigarrow j}{\Delta \vdash (\mathsf{Fun}\, V\, \lambda y W)\ \delta \Uparrow \mathsf{Set} \rightsquigarrow \max(i,j)}$$

This judgement checks that V is a well-formed type and additionally infers the lowest universe level i this type lives in. The type Set_i is well-formed and lives in level $i+1$. A neutral type is well-formed if its type is computed as Set_i for some i. A function type is well-formed if both domain and codomain are, and it lives in any level both components live in.

Comparison to [Coq96]. The second author has presented a similar type checking algorithm before [Coq96] for unstratified universes $\mathsf{Set} : \mathsf{Set}$. The main difference is that in rule CHK-INF he uses an untyped β-conversion check $X \sim X'$ instead of typed $\beta\eta$-conversion $\Delta \vdash X = X' \Uparrow \mathsf{Set} \rightsquigarrow i$ which we will describe in the following section. A minor difference is that in $\Delta \vdash v\ \delta \Uparrow X$ he uses Δ to assign types to the free variables of term v whereas we use $\Delta \circ \delta$. Consequently, the free variables of X would live in context $\Delta \circ \delta^{-1}$ in his case, however, this is problematic in principle since δ may not be invertible, e. g., in case of shadowing. Since he uses untyped conversion, this is irrelevant, because he never needs to look at the types of free variables in X. In our case, it is crucial.

2.4 Checking Equality

Checking the type of a neutral expression against X' while its type has been inferred as X requires testing the types X and X' for equality. Since types depend on objects, we will also have to compare objects. The principal method to check η-equality is a *type-directed* algorithm. In the following we present such a type-directed algorithm for comparing *values*.

Analogously to type checking, we define three inductive judgements. Herein, $d, d', e, e', X, X' \in \mathsf{D}$, $\Delta \in \mathsf{SemCxt}$, and $i \in \mathbb{N}$.

$\Delta \vdash e = e' \Downarrow X$	neutral e and e' are equal, inferring type X
$\Delta \vdash d = d' \Uparrow X$	d and d' are equal, checked at type X
$\Delta \vdash X = X' \Uparrow \mathsf{Set} \rightsquigarrow i$	X and X' are equal types, inferring universe level i

Inference mode $\Delta \vdash e = e' \Downarrow X$ (inputs: Δ,e,e', output: X or fail). In inference mode, neutral values e, e' are checked for equality, and their type is inferred.

$$\text{AQ-VAR}\ \frac{}{\Delta \vdash x = x \Downarrow \Delta(x)}$$

$$\text{AQ-FUN}\ \frac{\Delta \vdash e = e' \Downarrow \mathsf{Fun}\, X F \qquad \Delta \vdash d = d' \Uparrow X}{\Delta \vdash e\, d = e'\, d' \Downarrow F \cdot d}$$

$$\text{AQ-N-E}\ \frac{\begin{array}{c}\Delta, \mathsf{x}_\Delta \!:\! \mathsf{N}, \mathsf{x}_{\Delta+1} \!:\! F \cdot \mathsf{x}_\Delta \vdash d_s \cdot \mathsf{x}_\Delta \cdot \mathsf{x}_{\Delta+1} = d'_s \cdot \mathsf{x}_\Delta \cdot \mathsf{x}_{\Delta+1} : F \cdot (\mathsf{s}\, \mathsf{x}_\Delta) \\ \Delta, \mathsf{x}_\Delta \!:\! \mathsf{N} \vdash F \cdot \mathsf{x}_\Delta = F' \cdot \mathsf{x}_\Delta \Uparrow \mathsf{Set} \rightsquigarrow i \qquad \Delta \vdash d_z = d'_z \Uparrow F \cdot z \qquad \Delta \vdash e = e' \Downarrow \mathsf{N}\end{array}}{\Delta \vdash \mathsf{rec}\, F\, d_z\, d_s\, e = \mathsf{rec}\, F'\, d'_z\, d'_s\, e' : F \cdot e}$$

A variable is only equal to itself; its type is read from the context. A neutral application $e\, d$ is only equal to another neutral application $e'\, d'$ and the function parts must be equal, as well as the argument parts. The type of e must be a function type $\mathsf{Fun}\, X F$, whose domain X is used to check d and d' for equality.

The type of the application $e\, d$ is computed as $F \cdot d$. We could equally well have chosen to return $F \cdot d'$. That both choices amount to the same follows from the correctness of the equality check; yet we cannot rely on it before we have established correctness. This will be an issue in the correctness proof (Sec. 5); Harper and Pfenning [HP05] have avoided these complications by using simply-typed skeletons to direct the equality algorithm. Their method relies on dependency erasure which works for LF but not for type theories with large eliminations.

Checking mode $\Delta \vdash d = d' \Uparrow X$ (inputs: Δ, d, d', X, output: succeed or fail).

$$\text{AQ-NE-F}\ \frac{\Delta \vdash e = e' \Downarrow E_1 \qquad \Delta \vdash E_1 = E_2 \Downarrow \mathsf{Set}_i}{\Delta \vdash e = e' \Uparrow E_2}$$

$$\text{AQ-EXT}\ \frac{\Delta, \mathsf{x}_\Delta \!:\! X \vdash f \cdot \mathsf{x}_\Delta = f' \cdot \mathsf{x}_\Delta \Uparrow F \cdot \mathsf{x}_\Delta}{\Delta \vdash f = f' \Uparrow \mathsf{Fun}\, X F}$$

$$\text{AQ-TY}\ \frac{\Delta \vdash X = X' \Uparrow \mathsf{Set} \rightsquigarrow i}{\Delta \vdash X = X' \Uparrow \mathsf{Set}_j}\ i \leq j$$

$$\text{AQ-N-I-Z}\ \frac{}{\Delta \vdash \mathsf{z} = \mathsf{z} \Uparrow \mathsf{N}} \qquad \text{AQ-N-I-S}\ \frac{\Delta \vdash d = d' \Uparrow \mathsf{N}}{\Delta \vdash \mathsf{s}\, d = \mathsf{s}\, d' \Uparrow \mathsf{N}}$$

Neutral values e, e' can only be of neutral type E_i, they are passed to inference mode. The check $\Delta \vdash E_1 = E_2 \Downarrow \mathsf{Set}_i$ should actually not be necessary, because we already now that e and e' are well-typed, and types are only present to guide the equality algorithm. However, currently we do not know how to show soundness without it. Such redundant checks are present in other works as well [HP05, p. 77].

Two values f, f' of functional type are equal if applying them to a fresh variable x_Δ makes them equal. This is extensional equality, provided we can substitute arbitrary values for the variable. Had we formulated the algorithm on terms instead of values, this would be a standard substitution theorem. However, in our case it is more difficult. We deal with this issue in Sec. 8.

Values X, X' of type Set_j must be types, we check their equality in the type mode. The inferred universe i must be at most j, otherwise X, X' are not well-typed.

Type mode $\Delta \vdash X = X' \Uparrow \mathsf{Set} \rightsquigarrow i$ (inputs: Δ, X, X', output: i or fail).

$$\textsc{aq-ty-ne}\ \frac{\Delta \vdash E = E' \Downarrow \mathsf{Set}_i}{\Delta \vdash E = E' \Uparrow \mathsf{Set} \rightsquigarrow i}$$

$$\textsc{aq-ty-n}\ \frac{}{\Delta \vdash \mathsf{N} = \mathsf{N} \Uparrow \mathsf{Set} \rightsquigarrow 0} \qquad \textsc{aq-ty-set}\ \frac{}{\Delta \vdash \mathsf{Set}_i = \mathsf{Set}_i \Uparrow \mathsf{Set} \rightsquigarrow i+1}$$

$$\textsc{aq-ty-fun}\ \frac{\Delta \vdash X = X' \Uparrow \mathsf{Set} \rightsquigarrow i \qquad \Delta, \mathsf{x}_\Delta : X \vdash F \cdot \mathsf{x}_\Delta = F' \cdot \mathsf{x}_\Delta \Uparrow \mathsf{Set} \rightsquigarrow j}{\Delta \vdash \mathsf{Fun}\, X F = \mathsf{Fun}\, X' F' \Uparrow \mathsf{Set} \rightsquigarrow \max(i, j)}$$

A neutral type E can only be equal to another neutral type E', we delegate the test to the inference mode. A universe Set_i is only equal to itself. Function types $\mathsf{Fun}\, X F$ and $\mathsf{Fun}\, X' F'$ are equal if their domains and codomains coincide. For checking the codomains we introduce the fresh variable x_Δ of type X into the context. Again arbitrarily; we could have chosen X' instead. This is another source of asymmetry which complicates the correctness proof; for LF, it can be avoided by considering simply-typed contexts [HP05].

This algorithm is called *semantic* since it compares values. It is part of the core of Agda and Epigram 2. Since it is of practical relevance, it is a worth-while effort to verify it. Correctness of type checking is then a consequence of the correctness of algorithmic equality.

3 Specification: Typing with Explicit Substitutions

We want to prove that our algorithmic equality is correct, so we should say in which sense and provide a specification. There are different ways to present type theory. We choose a formulation with explicit substitutions [ML92] because the typing and equality rules can then be validated directed in any (Kripke) PER model over *any syntactical applicative structure* (see Thm. 1). Altenkirch and Chapman [AC08] exploit this fact for their closure-based definition of values. A formulation with non-explicit (deep) substitution directly validates the inference rules only for PER models over syntactical applicative structures *with extra properties*, e. g., λ-algebras [AC07], or combinatory algebras [ACD07].

We extend the expression syntax by explicit substitutions and introduce syntactical typing contexts:

$$
\begin{array}{llll}
\mathsf{Exp} & \ni r, s, t & ::= \cdots \mid t\sigma & \text{expressions} \\
\mathsf{Subst} & \ni \sigma, \tau & ::= (\sigma, x{=}t) \mid \sigma_{\mathsf{id}} \mid \sigma \circ \tau & \text{substitutions} \\
\mathsf{Cxt} & \ni \Gamma & ::= \diamond \mid \Gamma, x{:}A & \text{typing contexts}
\end{array}
$$

We identify expressions up to α-conversion and adopt the convention that in contexts Γ all variables must be distinct. Hence, we can view Γ as a map from variables to types with finite domain $\mathsf{dom}(\Gamma)$ and let $\Gamma(x) = A$ iff $(x{:}A) \in \Gamma$. In context extensions $\Gamma, x : A$ we assume $x \notin \mathsf{dom}(\Gamma)$. As usual $\mathsf{FV}(t)$ is the set of free variables of t. We let $\mathsf{FV}(t_1, \ldots, t_n) = \mathsf{FV}(t_1) \cup \cdots \cup \mathsf{FV}(t_n)$ and $\mathsf{FV}(\Gamma) = \bigcup_{x \in \mathsf{dom}(\Gamma)} \mathsf{FV}(\Gamma(x))$.

We have extended the language by explicit substitutions so we need to extend the notion of syntactical applicative structure, to ensure substitutions are evaluated reasonably:

$$
\begin{array}{lr}
\text{EVAL-SUBST-ID} & \sigma_{\mathsf{id}}\rho = \rho \\
\text{EVAL-SUBST-COMP} & (\sigma \circ \sigma')\rho = \sigma(\sigma'\rho) \\
\text{EVAL-SUBST-EXT} & (\sigma, x{=}s)\rho = (\sigma\rho, x{=}s\rho) \\
\text{EVAL-ESUBST} & (t\sigma)\rho = t(\sigma\rho)
\end{array}
$$

Herein, $\sigma\rho$ is defined by $x(\sigma\rho) = (x\sigma)\rho$.

Judgements. Our type theory with explicit substitutions has the following forms of judgement:

$$
\begin{array}{ll}
\Gamma \vdash & \Gamma \text{ is a well-formed context} \\
\Gamma \vdash A & A \text{ is a well-formed type in } \Gamma \\
\Gamma \vdash t : A & t \text{ has type } A \text{ in } \Gamma \\
\Gamma \vdash \sigma : \Gamma' & \sigma \text{ is a well-formed substitution in } \Gamma \\
\Gamma \vdash A = A' & A \text{ and } A' \text{ are equal types in } \Gamma \\
\Gamma \vdash t = t' : A & t \text{ and } t' \text{ are equal terms of type } A \text{ in } \Gamma \\
\Gamma \vdash \sigma = \sigma' : \Gamma' & \sigma \text{ and } \sigma' \text{ are equal substitutions in } \Gamma
\end{array}
$$

For an arbitrary judgement, we write $\Gamma \vdash J$, where J is a collection of syntactic entities (terms, contexts, substitutions) to the right of $\vdash$ in a judgement. $\mathsf{FV}(J)$ is the union of the free variable sets of all entities in J. Exceptions are $\mathsf{FV}(\sigma : \Delta)$, which is defined as $\bigcup_{x \in \mathsf{dom}(\Delta)} \mathsf{FV}(\Delta(x), \sigma(x))$, and $\mathsf{FV}(\sigma = \sigma' : \Delta) = \bigcup_{x \in \mathsf{dom}(\Delta)} \mathsf{FV}(\Delta(x), \sigma(x), \sigma'(x))$.

The judgements on types can be defined in terms of the judgement on terms.

$$
\begin{array}{llll}
\Gamma \vdash A & \iff & \Gamma \vdash A : \mathsf{Set}_i & \text{for some } i \\
\Gamma \vdash A = A' & \iff & \Gamma \vdash A = A' : \mathsf{Set}_i & \text{for some } i
\end{array}
$$

The *inference rules* for the other judgements are given in figures 2, 3, 4, and 5. They are inspired by categorical presentations of type theory, in particular, categories with families [Dyb96], which have been inspired by Martin-Löf's substitution calculus [ML92].

Remark 1. Martin-Löf has the following more general version of EQ-SUBST-EXT-η (see Figure 4).

$$\frac{\Gamma \vdash \sigma : \Gamma', x\!:\!A}{\Gamma \vdash \sigma = (\sigma, x\!=\!x\sigma) : \Gamma', x\!:\!A}$$

However, this rule is derivable in our calculus by EQ-SUBST-EXT-β.

Rule CONST relies on an auxiliary judgement $\Sigma \vdash c : A$ meaning constant c can be assigned type A. It is defined in Figure 1.

$$\text{N-F}\ \frac{}{\Sigma \vdash \mathsf{N} : \mathsf{Set}_0} \qquad \text{SET-F}\ \frac{}{\Sigma \vdash \mathsf{Set}_i : \mathsf{Set}_{i+1}}$$

$$\text{N-I-Z}\ \frac{}{\Sigma \vdash \mathsf{z} : \mathsf{N}} \qquad \text{N-I-S}\ \frac{}{\Sigma \vdash \mathsf{s} : \mathsf{N} \to \mathsf{N}}$$

$$\text{N-E}\ \frac{}{\Sigma \vdash \mathsf{rec} : \mathsf{Fun}\ (\mathsf{N} \to \mathsf{Set}_i)\ \lambda C.\, C \to (\mathsf{Fun}\ \mathsf{N}\ \lambda n.\, C\, n \to C\, (\mathsf{s}\, n)) \to \mathsf{Fun}\, \mathsf{N}\, C}$$

Fig. 1. Signature $\Sigma \vdash c : A$.

Rule SUB implements a minimum of subtyping: The types at level i are contained in higher levels $j \leq i$. Stronger subtyping, as results from the usual contravariant rule for function spaces, would be also valid in the Kripke model (Sec 5).

The judgements enjoy some standard properties, like weakening, inversion of typing, and well-formedness of contexts, types and terms (syntactic validity).

The following lemmata have easy but tedious proofs (many cases to consider).

Lemma 1 (Basic properties).

1. *(Scope:) If* $\Gamma \vdash J$ *then* $\mathsf{FV}(J) \subseteq \mathsf{dom}(\Gamma)$.
2. *(Context well-formedness:) If* $\Gamma, x\!:\!A, \Gamma' \vdash J$*, then* $\Gamma \vdash A$.
3. *(Weakening:) If* $\Gamma, \Gamma' \vdash J$ *and both* $\Gamma \vdash A$ *and* $x \notin \mathsf{dom}(\Gamma, \Gamma')$*, then* $\Gamma, x\!:\!A, \Gamma' \vdash J$.
4. *(Context conversion:) Let* $\Gamma \vdash B = A$ *and* $\Gamma \vdash B$*. If* $\Gamma, x\!:\!A, \Gamma' \vdash J$ *then* $\Gamma, x\!:\!B, \Gamma' \vdash J$.

Lemma 2 (Inversion).

Well-formed contexts $\Gamma \vdash$.

$$\text{CXT-EMPTY}\ \frac{}{\diamond \vdash} \qquad \text{CXT-EXT}\ \frac{\Gamma \vdash A}{\Gamma, x\!:\!A \vdash}$$

Well-typed terms $\Gamma \vdash t : A$.

$$\text{CONST}\ \frac{\Gamma \vdash \qquad \Sigma \vdash c : A}{\Gamma \vdash c : A} \qquad \text{HYP}\ \frac{\Gamma \vdash \qquad (x\!:\!A) \in \Gamma}{\Gamma \vdash x : A}$$

$$\text{CONV}\ \frac{\Gamma \vdash t : A \qquad \Gamma \vdash A = A'}{\Gamma \vdash t : A'} \qquad \text{SUB}\ \frac{\Gamma \vdash A : \mathsf{Set}_i}{\Gamma \vdash A : \mathsf{Set}_j}\ i \leq j$$

$$\text{FUN-F}\ \frac{\Gamma \vdash A : \mathsf{Set}_i \qquad \Gamma, x\!:\!A \vdash B : \mathsf{Set}_i}{\Gamma \vdash \mathsf{Fun}\, A\, \lambda x B : \mathsf{Set}_i}$$

$$\text{FUN-I}\ \frac{\Gamma, x\!:\!A \vdash t : B}{\Gamma \vdash \lambda x t : \mathsf{Fun}\, A\, \lambda x B} \qquad \text{FUN-E}\ \frac{\Gamma \vdash r : \mathsf{Fun}\, A\, \lambda x B \qquad \Gamma \vdash s : A}{\Gamma \vdash r\, s : B(\sigma_{\mathsf{id}}, x\!=\!s)}$$

$$\text{ESUBST-F}\ \frac{\Gamma \vdash \sigma : \Gamma' \qquad \Gamma' \vdash t : A}{\Gamma \vdash t\sigma : A\sigma}$$

Well-formed substitutions $\Gamma \vdash \sigma : \Gamma'$.

$$\text{SUBST-EXT}\ \frac{\Gamma \vdash \sigma : \Gamma' \qquad \Gamma' \vdash A \qquad \Gamma \vdash s : A\sigma}{\Gamma \vdash (\sigma, x\!=\!s) : (\Gamma', x\!:\!A)}$$

$$\text{SUBST-ID}\ \frac{\Gamma \vdash}{\Gamma \vdash \sigma_{\mathsf{id}} : \Gamma} \qquad \text{SUBST-COMP}\ \frac{\Gamma_2 \vdash \sigma : \Gamma_3 \qquad \Gamma_1 \vdash \tau : \Gamma_2}{\Gamma_1 \vdash \sigma \circ \tau : \Gamma_3}$$

$$\text{SUBST-WEAK}\ \frac{\Gamma \vdash \sigma : \Gamma', x\!:\!A, \Gamma''}{\Gamma \vdash \sigma : \Gamma', \Gamma''}$$

Fig. 2. Rules for contexts, types, and terms.

Equality $\Gamma \vdash t = t' : A$. Equivalence, hypotheses, conversion.

$$\text{EQ-REFL}\ \frac{\Gamma \vdash t : A}{\Gamma \vdash t = t : A} \qquad \text{EQ-SYM}\ \frac{\Gamma \vdash t = t' : A}{\Gamma \vdash t' = t : A}$$

$$\text{EQ-TRANS}\ \frac{\Gamma \vdash t = t' : A \qquad \Gamma \vdash t' = t'' : A}{\Gamma \vdash t = t'' : A}$$

$$\text{EQ-CONST}\ \frac{\Gamma \vdash \qquad \Sigma \vdash c : A}{\Gamma \vdash c = c : A} \qquad \text{EQ-HYP}\ \frac{\Gamma \vdash \qquad (x\!:\!A) \in \Gamma}{\Gamma \vdash x = x : A}$$

$$\text{EQ-CONV}\ \frac{\Gamma \vdash t = t' : A \qquad \Gamma \vdash A = A'}{\Gamma \vdash t = t' : A'} \qquad \text{EQ-SUB}\ \frac{\Gamma \vdash A = A' : \mathsf{Set}_i}{\Gamma \vdash A = A' : \mathsf{Set}_j}\ i \leq j$$

Dependent functions.

$$\text{EQ-FUN-F}\ \frac{\Gamma \vdash A = A' : \mathsf{Set}_i \qquad \Gamma, x\!:\!A \vdash B = B' : \mathsf{Set}_i}{\Gamma \vdash \mathsf{Fun}\, A\, \lambda x B = \mathsf{Fun}\, A'\, \lambda x B' : \mathsf{Set}_i}$$

$$\text{EQ-FUN-I}\ \frac{\Gamma, x\!:\!A \vdash t = t' : B}{\Gamma \vdash \lambda x t = \lambda x t' : \mathsf{Fun}\, A\, \lambda x B}$$

$$\text{EQ-FUN-E}\ \frac{\Gamma \vdash r = r' : \mathsf{Fun}\, A\, \lambda x B \qquad \Gamma \vdash s = s' : A}{\Gamma \vdash r\, s = r'\, s' : B(\sigma_{\mathsf{id}}, x\!=\!s)}$$

$$\text{EQ-FUN-}\beta\ \frac{\Gamma, x\!:\!A \vdash t : B \qquad \Gamma \vdash s : A}{\Gamma \vdash (\lambda x t)\, s = t(\sigma_{\mathsf{id}}, x\!=\!s) : B(\sigma_{\mathsf{id}}, x\!=\!s)}$$

$$\text{EQ-FUN-}\eta\ \frac{\Gamma \vdash t : \mathsf{Fun}\, A\, \lambda x B}{\Gamma \vdash (\lambda x.\, t\, x) = t : \mathsf{Fun}\, A\, \lambda x B}\ x \notin \mathsf{dom}(\Gamma)$$

Natural numbers.

$$\text{EQ-N-}\iota\text{-Z}\ \frac{\Gamma \vdash C : \mathsf{N} \to \mathsf{Set}_i \qquad \Gamma \vdash z : C\, \mathsf{z} \qquad \Gamma \vdash s : \mathsf{Fun}\, \mathsf{N}\, \lambda n.\, C\, n \to C\, (\mathsf{s}\, n)}{\Gamma \vdash \mathsf{rec}\, C\, z\, s\, \mathsf{z} = z : C\, \mathsf{z}}$$

$$\text{EQ-N-}\iota\text{-S}\ \frac{\Gamma \vdash C : \mathsf{N} \to \mathsf{Set}_i \qquad \Gamma \vdash z : C\, \mathsf{z} \qquad \Gamma \vdash s : \mathsf{Fun}\, \mathsf{N}\, \lambda n.\, C\, n \to C\, (\mathsf{s}\, n) \qquad \Gamma \vdash n : \mathsf{N}}{\Gamma \vdash \mathsf{rec}\, C\, z\, s\, (\mathsf{s}\, n) = s\, n\, (\mathsf{rec}\, C\, z\, s\, n) : C\, (\mathsf{s}\, n)}$$

Fig. 3. Equality rules.

Equivalence rules and weakening.

$$\textsc{eq-subst-refl}\ \frac{\mathit{\Gamma} \vdash \sigma : \mathit{\Gamma}'}{\mathit{\Gamma} \vdash \sigma = \sigma : \mathit{\Gamma}'} \qquad \textsc{eq-subst-sym}\ \frac{\mathit{\Gamma} \vdash \sigma = \sigma' : \mathit{\Gamma}'}{\mathit{\Gamma} \vdash \sigma' = \sigma : \mathit{\Gamma}'}$$

$$\textsc{eq-subst-trans}\ \frac{\mathit{\Gamma}_1 \vdash \sigma = \sigma' : \mathit{\Gamma}_2 \qquad \mathit{\Gamma}_2 \vdash \sigma' = \sigma'' : \mathit{\Gamma}_3}{\mathit{\Gamma}_1 \vdash \sigma = \sigma'' : \mathit{\Gamma}_3}$$

$$\textsc{eq-subst-weak}\ \frac{\mathit{\Gamma} \vdash \sigma = \sigma' : \mathit{\Gamma}', x{:}A, \mathit{\Gamma}''}{\mathit{\Gamma} \vdash \sigma = \sigma' : \mathit{\Gamma}', \mathit{\Gamma}''}$$

Rules of the category of contexts and substitutions.

$$\textsc{eq-subst-id-l}\ \frac{\mathit{\Gamma} \vdash \sigma : \mathit{\Gamma}'}{\mathit{\Gamma} \vdash \sigma_{\mathsf{id}} \circ \sigma = \sigma : \mathit{\Gamma}'} \qquad \textsc{eq-subst-id-r}\ \frac{\mathit{\Gamma} \vdash \sigma : \mathit{\Gamma}'}{\mathit{\Gamma} \vdash \sigma \circ \sigma_{\mathsf{id}} = \sigma : \mathit{\Gamma}'}$$

$$\textsc{eq-subst-assoc}\ \frac{\mathit{\Gamma}_3 \vdash \sigma : \mathit{\Gamma}_4 \qquad \mathit{\Gamma}_2 \vdash \sigma' : \mathit{\Gamma}_3 \qquad \mathit{\Gamma}_1 \vdash \sigma'' : \mathit{\Gamma}_2}{\mathit{\Gamma}_1 \vdash (\sigma \circ \sigma') \circ \sigma'' = \sigma \circ (\sigma' \circ \sigma'') : \mathit{\Gamma}_4}$$

Rules for the empty substitution and substitution extension.

$$\textsc{eq-subst-empty-}\eta\ \frac{\mathit{\Gamma} \vdash \sigma : \diamond \qquad \mathit{\Gamma} \vdash \sigma' : \diamond}{\mathit{\Gamma} \vdash \sigma = \sigma' : \diamond}$$

$$\textsc{eq-subst-ext-}\beta\ \frac{\mathit{\Gamma}_2 \vdash \sigma : \mathit{\Gamma}_3 \qquad \mathit{\Gamma}_3 \vdash A \qquad \mathit{\Gamma}_2 \vdash s : A\sigma \qquad \mathit{\Gamma}_1 \vdash \tau : \mathit{\Gamma}_2}{\mathit{\Gamma}_1 \vdash (\sigma, x{=}s) \circ \tau = (\sigma \circ \tau, x{=}s\tau) : \mathit{\Gamma}_3, x{:}A}$$

$$\textsc{eq-subst-ext-}\eta\ \frac{\mathit{\Gamma}, x{:}A \vdash}{\mathit{\Gamma}, x{:}A \vdash (\sigma_{\mathsf{id}}, x{=}x) = \sigma_{\mathsf{id}} : \mathit{\Gamma}, x{:}A}$$

$$\textsc{eq-subst-ext-weak}\ \frac{\mathit{\Gamma} \vdash \sigma : \mathit{\Gamma}' \qquad \mathit{\Gamma}', x{:}A \vdash \qquad \mathit{\Gamma} \vdash t : A\sigma}{\mathit{\Gamma} \vdash (\sigma, x{=}t) = \sigma : \mathit{\Gamma}'}$$

Congruence rules.

$$\textsc{eq-subst-ext}\ \frac{\mathit{\Gamma} \vdash \sigma = \sigma' : \mathit{\Gamma}' \qquad \mathit{\Gamma}' \vdash A \qquad \mathit{\Gamma} \vdash s = s' : A\sigma}{\mathit{\Gamma} \vdash (\sigma, x{=}s) = (\sigma', x{=}s') : (\mathit{\Gamma}', x{:}A)}$$

$$\textsc{eq-subst-comp}\ \frac{\mathit{\Gamma}_2 \vdash \sigma = \sigma' : \mathit{\Gamma}_3 \qquad \mathit{\Gamma}_1 \vdash \tau = \tau' : \mathit{\Gamma}_2}{\mathit{\Gamma}_1 \vdash \sigma \circ \tau = \sigma' \circ \tau' : \mathit{\Gamma}_3}$$

Fig. 4. Equality rules for substitutions $\mathit{\Gamma} \vdash \sigma = \sigma' : \mathit{\Delta}$.

$$\text{EQ-ESUBST-F}\ \frac{\Gamma \vdash \sigma = \sigma' : \Delta \qquad \Delta \vdash t = t' : A}{\Gamma \vdash t\sigma = t'\sigma' : A\sigma}$$

$$\text{EQ-ESUBST-ID}\ \frac{\Gamma \vdash t : A}{\Gamma \vdash t\sigma_{\mathsf{id}} = t : A}$$

$$\text{EQ-ESUBST-COMP}\ \frac{\Gamma \vdash \tau : \Gamma' \qquad \Gamma' \vdash \sigma : \Gamma'' \qquad \Gamma'' \vdash t : A}{\Gamma \vdash t(\sigma \circ \tau) = (t\sigma)\tau : A(\sigma \circ \tau)}$$

$$\text{EQ-ESUBST-C}\ \frac{\Gamma \vdash \sigma : \Gamma' \qquad \Sigma \vdash c : A}{\Gamma \vdash c\sigma = c : A}$$

$$\text{EQ-ESUBST-VAR}\ \frac{\Gamma \vdash \sigma : \Gamma' \qquad \Gamma', x\!:\!A \vdash \qquad \Gamma \vdash t : A\sigma}{\Gamma \vdash x(\sigma, x\!=\!t) = t : A\sigma}$$

$$\text{EQ-ESUBST-FUN-F}\ \frac{\Gamma \vdash \sigma : \Gamma' \qquad \Gamma' \vdash A : \mathsf{Set}_i \qquad \Gamma', x\!:\!A \vdash B : \mathsf{Set}_i}{\Gamma \vdash (\mathsf{Fun}\, A\, \lambda x B)\sigma = \mathsf{Fun}\, (A\sigma)\, (\lambda x B)\sigma : \mathsf{Set}_i}\ x \notin \mathsf{dom}(\Gamma)$$

$$\text{EQ-ESUBST-FUN-I}\ \frac{\Gamma \vdash \sigma : \Gamma' \qquad \Gamma', x\!:\!A \vdash t : B}{\Gamma \vdash (\lambda x t)\sigma = \lambda x.\, t(\sigma, x\!=\!x) : (\mathsf{Fun}\, A\, \lambda x B)\sigma}\ x \notin \mathsf{dom}(\Gamma)$$

$$\text{EQ-ESUBST-FUN-E}\ \frac{\Gamma \vdash \sigma : \Gamma' \qquad \Gamma' \vdash r : \mathsf{Fun}\, A\, \lambda x B \qquad \Gamma' \vdash s : A}{\Gamma \vdash (r\, s)\sigma = r\sigma\, s\sigma : B(\sigma, x\!=\!s\sigma)}$$

Fig. 5. Equality rules for explicit substitutions.

1. *If* $\Gamma \vdash c : C$ *then* $\Sigma \vdash c : C'$ *for some* C' *with* $\Gamma \vdash C = C'$.
2. *If* $\Gamma \vdash x : C$ *then* $\Gamma \vdash C = \Gamma(x)$.
3. *If* $\Gamma \vdash \lambda x t : C$ *then* $\Gamma \vdash C = \mathsf{Fun}\, A\, \lambda x B$ *and* $\Gamma, x{:}A \vdash t : B$.
4. *If* $\Gamma \vdash r\, s : C$ *then* $\Gamma \vdash r : \mathsf{Fun}\, A\, \lambda x B$ *with* $\Gamma \vdash s : A$ *and* $\Gamma \vdash C = B(\sigma_{\mathsf{id}}, x{=}s)$.
5. *If* $\Gamma \vdash \mathsf{Fun}\, A\, \lambda x B : C$ *then* $\Gamma \vdash C = \mathsf{Set}_i$ *for some* i *and* $\Gamma \vdash A : \mathsf{Set}_i$ *and* $\Gamma, x{:}A \vdash B : \mathsf{Set}_i$.

It is tempting to treat Fun as ordinary constant of type $\mathsf{Fun}\, \mathsf{Set}_i\, \lambda A.(A \to \mathsf{Set}_i) \to \mathsf{Set}_i$ for each i, and dispose of FUN-F. However, then inversion for function types (Lemma 2.5) requires injectivity, with the consequence that the following lemma cannot be proven syntactically anymore.

Lemma 3 (Syntactic validity).

1. *Typing: If* $\Gamma \vdash t : A$ *then* $\Gamma \vdash A$.
2. *Typing of substitutions: If* $\Gamma \vdash \sigma : \Gamma'$ *then* $\Gamma' \vdash$.
3. *Equality: If* $\Gamma \vdash t = t' : A$ *then* $\Gamma \vdash A$ *and both* $\Gamma \vdash t : A$ *and* $\Gamma \vdash t' : A$.
4. *Equality of substitutions: If* $\Gamma \vdash \sigma = \sigma' : \Gamma'$ *then* $\Gamma' \vdash$ *and both* $\Gamma \vdash \sigma : \Gamma'$ *and* $\Gamma \vdash \sigma' : \Gamma'$.

4 Verification Plan and Contextual Reification

A standard method to show completeness of the algorithmic equality would be the following [HP05].

1. Define a Kripke logical relation $\Delta \vdash d = d' : X$ on a semantic context Δ and values d, d' by induction on the type X. We will call this relation *Kripke model*. For base types X, let the relation coincide with algorithmic equality $\Delta \vdash d = d' \Uparrow X$, for function types do the usual functional construction: $\Delta \vdash f = f' : \mathsf{Fun}\, X F$ iff $\Delta' \vdash d = d' : X$ implies $\Delta' \vdash f \cdot d = f' \cdot d' : F \cdot d$ for all d, d' and all extensions Δ' of Δ.
2. Show that if two values are related in the model, then the algorithm relates them as well. Following Schürmann and Sarnat [SS08] we call this the *escape* lemma, since it allows to "get out of the logical relation".
3. Finally show validity of the inference rules w. r. t. the model, i.e., if two terms t, t' are equal of type A, then for each well-formed environment ρ, we have $t\rho = t'\rho : A\rho$ in the model, which implies that the algorithm accepts t, t' as equal.

In particular, the relation $\Delta \vdash \_ = \_ : X$ needs to be a partial equivalence, in order to validate symmetry and transitivity rules. But due to the asymmetric nature of algorithmic equality (rules AQ-FUN and AQ-TY-FUN), this can only be shown if we have soundness, which at this point we cannot obtain.

Normalization-by-evaluation (NbE) [ML75,BS91,Coq94,Dan99] to the rescue! There already are equality algorithms for dependent types with large eliminations which are based on semantics [AAD07,ACD07]. Two semantic values are considered equal if they reify to the same expression. *Reification* at a type X converts a value to a term, η-expanding it on the fly according to type X. It turns out that we can verify the algorithmic equality by relating it to NbE.

We will verify algorithmic equality according to this plan:

1. Define normalization-by-evaluation for our setting. This amounts to defining *contextual reification* $\Delta \vdash d \searrow v \Uparrow X$ which converts value d of type X in context Δ to normal form v.
2. Show completeness of NbE (Sec. 5), meaning that if one takes two judgmentally equal terms t, t', evaluates and reifies them, one arrives at the same normal form v (Cor. 1). To this end, we construct a Kripke model based on reification, meaning that two values are equal at base type if they reify to the same normal form.
3. Show soundness of NbE (Sec. 6), meaning that if we take a term t, evaluate and reify it, we arrive at a normal form v judgmentally equal to t (Cor. 2). The main tool is a Kripke logical relation between well-typed terms t and semantic objects d, which for base types states that d reifies to a normal form v which is judgmentally equal to t.
4. Show completeness of the algorithmic equality (Sec. 7). This is a corollary of completeness of NbE, since we will see that if two values reify to the same normal form, then the algorithm will accept them as equal (Lemma 17).

5. Show soundness of the algorithm (Sec. 8). The direct approach, showing that two algorithmically equal values reify to the same normal form, fails due to the asymmetry of the algorithm. We introduce the concept of strong semantic typing and equality (a super Kripke model, so to say) and prove that the algorithm is sound for strong semantic equality. By establishing that the inference rules are valid in the super Kripke model (Cor. 4), and that equality in the super Kripke model entails equality in the Kripke model, we finally show that well-typed terms whose values are algorithmically equal reify to the same normal form, thus, are judgmentally equal (Thm. 3).

What remains open is termination of algorithmic equality.

4.1 Contextual Reification

Reification [BS91] converts a semantic value to a syntactic term, η-expanding it on the fly. It is defined by recursion on the type of the value. In previous NbE approaches [BS91,ACD07] the semantics of base types has been defined as a set of η-long neutral terms, thus, reification at base type is simply the identity. In our approach, the semantics of base types is a set of neutral values, which are not η-expanded and need to be reified recursively. This can only happen if reification has access to the types of the free variables of a neutral value. For example, to reify $x\,d\,d'$ at base type we need to retrieve the type $\mathsf{Fun}\,X\,F$ of x, recursively reify d at type X, compute $F \cdot d = \mathsf{Fun}\,X'\,F'$ and recursively reify d' at type X'. For this task, we introduce a new form of reification which is parameterized by a semantic typing context Δ, hence the name *contextual reification*.

We simultaneously define three inductive judgements by the rules to follow. Herein, $\Delta \in \mathsf{SemCxt}$, $d, e, X \in \mathsf{D}$, $u \in \mathsf{Ne}$, $v, V \in \mathsf{Nf}$ and $i \in \mathbb{N}$.

$\Delta \vdash e \searrow u \Downarrow X$	neutral value e reifies to u, inferring its type X
$\Delta \vdash d \searrow v \Uparrow X$	value d reifies to normal form v at type X
$\Delta \vdash X \searrow V \Uparrow \mathsf{Set} \rightsquigarrow i$	type value X reifies to V, inferring its level i.

Inference mode $\Delta \vdash e \searrow u \Downarrow X$ (inputs: Δ, e, outputs: u, X or fail).

$$\textsc{reify-var}\ \frac{}{\Delta \vdash x \searrow x \Downarrow \Delta(x)}$$

$$\textsc{reify-fun-e}\ \frac{\Delta \vdash e \searrow u \Downarrow \mathsf{Fun}\,X F \qquad \Delta \vdash d \searrow v \Uparrow X}{\Delta \vdash e\,d \searrow u\,v \Downarrow F \cdot d}$$

$$\textsc{reify-n-e}\ \frac{\Delta \vdash e \searrow u \Downarrow \mathsf{N} \qquad \Delta \vdash d_z \searrow v_z \Uparrow F \cdot \mathsf{z} \qquad \Delta, x\!:\!\mathsf{N} \vdash F \cdot x \searrow V \Uparrow \mathsf{Set} \rightsquigarrow i \qquad \Delta, x\!:\!\mathsf{N}, y\!:\!F \cdot x \vdash d_s \cdot x \cdot y \searrow v_s \Uparrow F \cdot (\mathsf{s}\,x)}{\Delta \vdash \mathsf{rec}\,F\,d_z\,d_s\,e \searrow \mathsf{rec}\,(\lambda x V)\,v_z\,(\lambda x \lambda y.\,v_s)\,u \Downarrow F \cdot e}$$

Variables reify to themselves and neutral applications to neutral applications. The type information flows out of the context Δ and is used in REIFY-FUN-E to reify d of type X in checking mode.

Checking mode $\Delta \vdash d \searrow v \Uparrow X$ (inputs: Δ, d, X, output: v or fail).

$$\textsc{reify-ne}\ \frac{\Delta \vdash e \searrow u \Downarrow E_1 \qquad \Delta \vdash E_1 \searrow u' \Downarrow \mathsf{Set}_i \qquad \Delta \vdash E_2 \searrow u' \Downarrow \mathsf{Set}_i}{\Delta \vdash e \searrow u \Uparrow E_2}$$

$$\textsc{reify-ext}\ \frac{\Delta, x\!:\!X \vdash f \cdot x \searrow v \Uparrow F \cdot x}{\Delta \vdash f \searrow \lambda x v \Uparrow \mathsf{Fun}\, X F}$$

$$\textsc{reify-n-i-z}\ \frac{}{\Delta \vdash \mathsf{z} \searrow \mathsf{z} \Uparrow \mathsf{N}} \qquad \textsc{reify-n-i-s}\ \frac{\Delta \vdash d \searrow v \Uparrow \mathsf{N}}{\Delta \vdash \mathsf{s}\, d \searrow \mathsf{s}\, v \Uparrow \mathsf{N}}$$

$$\textsc{reify-ty}\ \frac{\Delta \vdash X \searrow V \Uparrow \mathsf{Set} \rightsquigarrow i}{\Delta \vdash X \searrow V \Uparrow \mathsf{Set}_j}\ i \leq j$$

Any value f of functional type is reified by applying it to a fresh variable x. Note that this can trigger further evaluation, e. g., in the λ-model where functional values are just weak head normal forms or closures. The result of reifying a functional value is always a λ-abstraction, which means that reification returns η-long forms.

At neutral type E_2, objects e need to be neutral and are reified in inference mode. The inferred type E_1 needs to be equal to E_2—this is checked by reifying both types, expecting the same normal form.

Type mode $\Delta \vdash X \searrow V \Uparrow \mathsf{Set} \rightsquigarrow i$ (inputs: Δ, X, outputs: V, i or fail).

$$\textsc{reify-ty-ne}\ \frac{\Delta \vdash e \searrow u \Downarrow \mathsf{Set}_i}{\Delta \vdash e \searrow u \Uparrow \mathsf{Set} \rightsquigarrow i}$$

$$\textsc{reify-ty-n}\ \frac{}{\Delta \vdash \mathsf{N} \searrow \mathsf{N} \Uparrow \mathsf{Set} \rightsquigarrow 0} \qquad \textsc{reify-ty-set}\ \frac{}{\Delta \vdash \mathsf{Set}_i \searrow \mathsf{Set}_i \Uparrow \mathsf{Set} \rightsquigarrow i+1}$$

$$\textsc{reify-ty-fun}\ \frac{\Delta \vdash X \searrow V \Uparrow \mathsf{Set} \rightsquigarrow i \qquad \Delta, x\!:\!X \vdash F \cdot x \searrow W \Uparrow \mathsf{Set} \rightsquigarrow j}{\Delta \vdash \mathsf{Fun}\, X F \searrow \mathsf{Fun}\, V\, \lambda x W \Uparrow \mathsf{Set} \rightsquigarrow \max(i, j)}$$

Function type values reify to function type expressions in long normal form, universes to universes and neutral type values to neutral type expressions.

We write $\Delta \vdash e, e' \searrow u \Downarrow X$ for $\Delta \vdash e \searrow u \Downarrow X$ and $\Delta \vdash e' \searrow u \Downarrow X$. This gives a basic equality on neutral objects (which is used in rule reify-ne, for instance).

We say Δ' *extends* Δ, written $\Delta' \leq \Delta$, if $\Delta'(x) = \Delta(x)$ for all $x \in \mathsf{dom}(\Delta)$. (The direction of $\leq$ is as in subtyping.) Reification is closed under weakening of contexts, i.e., reifying in an extended context produces an α-equivalent normal form.

Lemma 4 (Weakening). *Let* $\Delta' \leq \Delta$.

1. *If* $\Delta \vdash e \searrow u \Downarrow X$ *then* $\Delta' \vdash e \searrow u \Downarrow X$.
2. *If* $\Delta \vdash d \searrow v \Uparrow X$ *then* $\Delta' \vdash d \searrow v \Uparrow X$.

3. If $\Delta \vdash X \searrow V \Uparrow \mathsf{Set} \rightsquigarrow i$ *then* $\Delta' \vdash X \searrow V \Uparrow \mathsf{Set} \rightsquigarrow i$.

Proof. Simultaneously by induction on the derivation.

Reification provides us with a normalization function: given a closed term $t : A$, the normal form v is obtained by $\diamond \vdash t\rho_{\mathsf{id}} \searrow v \Uparrow A\rho_{\mathsf{id}}$.

5 Kripke Model and Completeness of NbE

Dependent types complicate the definition of a logical relation, because one cannot use structural induction on the type expression. Instead one needs to simultaneously define the "good" type values X by induction and their denotation, a relation on objects, by recursion [Dyb00]. We spell out this construction for our Kripke model in sections 5.1–5.3. In Section 5.4 we prove that it models our inference rules.

5.1 An induction measure

We write $\mathcal{X}, \mathcal{Y}$ for subsets of D. If $\mathcal{X} \subseteq \mathsf{D}$ and $\mathcal{F}(d) \subseteq \mathsf{D}$ for each $d \in \mathcal{X}$, then the dependent function space

$$\Pi\, \mathcal{X}\, \mathcal{F} = \{ f \mid \forall d \in \mathcal{X}.\, f \cdot d \in \mathcal{F}(d) \}$$

is another subset of D. For $i = 0, 1, \dots$ we define the sets $\mathcal{T}_i \subseteq \mathsf{D} \times \mathcal{P}(\mathsf{D})$ inductively as follows:

$$\frac{}{(E, \mathsf{D}) \in \mathcal{T}_i} \qquad \frac{}{(\mathsf{Set}_j, |\mathcal{T}_j|) \in \mathcal{T}_i}\; j < i$$

$$\frac{(X, \mathcal{X}) \in \mathcal{T}_i \qquad \forall d \in \mathcal{X}.\, (F \cdot d, \mathcal{F}(d)) \in \mathcal{T}_i}{(\mathsf{Fun}\, X F,\, \Pi\, \mathcal{X}\, \mathcal{F}) \in \mathcal{T}_i}$$

Herein, $|\mathcal{T}_i| = \{ X \mid \exists \mathcal{X}.\, (X, \mathcal{X}) \in \mathcal{T}_i \}$. We define the relation $: \subseteq \mathsf{D} \times \mathsf{D}$ by

$$d : X \iff \exists \mathcal{X}, i.\, (X, \mathcal{X}) \in \mathcal{T}_i \text{ and } d \in \mathcal{X}$$

Lemma 5.

$$X : \mathsf{Set}_i \iff X \in |\mathcal{T}_i|$$

Proof.

$$\begin{aligned} & X : \mathsf{Set}_i \\ \iff & \exists j, \mathcal{X}. (\mathsf{Set}_i, \mathcal{X}) \in \mathcal{T}_j \text{ and } X \in \mathcal{X} \\ \iff & (\mathsf{Set}_i, |\mathcal{T}_i|) \in \mathcal{T}_{i+1} \text{ and } X \in |\mathcal{T}_i| \\ \iff & X \in |\mathcal{T}_i| \end{aligned}$$

We will use the derivation of membership in $\mathcal{T}_i$ as induction measure, quoted as "induction on $X : \mathsf{Set}_i$".

5.2 Construction of the Kripke model

It is tempting to define $\Delta \vdash d = d' : X$ for base types directly as "$\Delta \vdash d \searrow v \Uparrow X$ and $\Delta \vdash d' \searrow v \Uparrow X$ for some v". However, then the proof of the escape lemma will fail, because during reification of function types, their domains flow into the context. Reifying two function types will soon take place in different contexts. We need to be more general and relate semantic objects at a priori different types in a priori different contexts. Thus, in the following we define a relation $\Delta \vdash d : X \circledS \Delta' \vdash d' : X'$ for the purpose of proving the escape lemma, and we obtain $\Delta \vdash d = d' : X$ as a special case in Sec. 5.3.

By lexicographic induction on i and $X' : \mathsf{Set}_i$ we define the relations:

$$\_ \vdash \_ : \_ \circledS \_ \vdash X' : \mathsf{Set}_i$$
$$\_ \vdash \_ : \_ \circledS \_ \vdash \_ : X'$$

– *Case* $E' : \mathsf{Set}_i$.

$$\begin{aligned}&\Delta \vdash X : Z \circledS \Delta' \vdash E' : \mathsf{Set}_i\\ &\quad\iff X = E \text{ neutral and } Z = \mathsf{Set}_i \text{ and}\\ &\qquad\ \Delta \vdash E \searrow U \Downarrow \mathsf{Set}_j \text{ and } \Delta' \vdash E' \searrow U \Downarrow \mathsf{Set}_{j'} \text{ for some } U \text{ and } j, j' \leq i\end{aligned}$$

$$\begin{aligned}&\Delta \vdash d : X \circledS \Delta' \vdash d' : E'\\ &\quad\iff X = E \text{ neutral and } \Delta \vdash d \searrow u \Downarrow \hat{E} \text{ and } \Delta' \vdash d' \searrow u \Downarrow \hat{E}' \text{ and}\\ &\qquad\ \Delta \vdash E, \hat{E} \searrow U \Downarrow \mathsf{Set}_j \text{ and } \Delta' \vdash E', \hat{E}' \searrow U \Downarrow \mathsf{Set}_{j'}\\ &\qquad\ \text{for some } E, \hat{E}, \hat{E}', u, U, \text{ and } j, j' \leq i\end{aligned}$$

– *Case* $\mathsf{Set}_j : \mathsf{Set}_i$ for $j < i$.

$$\begin{aligned}&\Delta \vdash X : Z \circledS \Delta' \vdash \mathsf{Set}_j : \mathsf{Set}_i \iff X = \mathsf{Set}_j \text{ and } Z = \mathsf{Set}_i\\ &\Delta \vdash d : X \circledS \Delta' \vdash d' : \mathsf{Set}_j \quad \text{has already been defined}\end{aligned}$$

– *Case* $\mathsf{Fun}\, X'\, F' : \mathsf{Set}_i$ where $X' : \mathsf{Set}_i$ and $F' \cdot d : \mathsf{Set}_i$ for all $d : X'$.

$$\begin{aligned}&\Delta \vdash Y : Z \circledS \Delta' \vdash \mathsf{Fun}\, X' F' : \mathsf{Set}_i\\ &\quad\iff Y = \mathsf{Fun}\, X F \text{ for some } X, F \text{ and } Z = \mathsf{Set}_i \text{ and}\\ &\qquad\ \Delta \vdash X : \mathsf{Set}_i \circledS \Delta' \vdash X' : \mathsf{Set}_i \text{ and}\\ &\qquad\ \text{for all } \hat{\Delta} \leq \Delta, \hat{\Delta}' \leq \Delta', d, d',\ \hat{\Delta} \vdash d : X \circledS \hat{\Delta}' \vdash d' : X'\\ &\qquad\qquad \text{implies } \hat{\Delta} \vdash F \cdot d : \mathsf{Set}_i \circledS \hat{\Delta}' \vdash F' \cdot d' : \mathsf{Set}_i\end{aligned}$$

$$\begin{aligned}&\Delta \vdash f : Y \circledS \Delta' \vdash f : \mathsf{Fun}\, X' F'\\ &\quad\iff Y = \mathsf{Fun}\, X F \text{ for some } X, F \text{ and}\\ &\qquad\ \text{for all } \hat{\Delta} \leq \Delta, \hat{\Delta}' \leq \Delta', d, d',\ \hat{\Delta} \vdash d : X \circledS \hat{\Delta}' \vdash d' : X'\\ &\qquad\qquad \text{implies } \hat{\Delta} \vdash f \cdot d : F \cdot d \circledS \hat{\Delta}' \vdash f' \cdot d' : F' \cdot d'\end{aligned}$$

Lemma 6. $\circledS$ *is symmetric and transitive.*

Lemma 7 (Type conversion). *Let* $X' : \mathsf{Set}_i$*. If* $\Delta \vdash d : X \circledS \Delta' \vdash d' : X'$ *and* $\Delta' \vdash X' : \mathsf{Set}_i \circledS \Delta' \vdash X'' : \mathsf{Set}_i$ *then* $\Delta \vdash d : X \circledS \Delta' \vdash d' : X''$*.*

Proof. By induction on $X' : \mathsf{Set}_i$.

Lemma 8 (Into and out of the model / escape lemma). *Let* $X' : \mathsf{Set}_i$. *Then*

1. *(In:) If* $\Delta \vdash e \searrow u \Downarrow X$ *and* $\Delta' \vdash e' \searrow u \Downarrow X'$ *then* $\Delta \vdash e : X \circledS \Delta' \vdash e' : X'$.
2. *(Out:) If* $\Delta \vdash d : X \circledS \Delta' \vdash d' : X'$ *then* $\Delta \vdash d \searrow v \Uparrow X$ *and* $\Delta' \vdash d' \searrow v \Uparrow X'$ *for some* v.
3. *(Out-Type:) If* $\Delta \vdash X : \mathsf{Set}_i \circledS \Delta' \vdash X' : \mathsf{Set}_i$ $\Delta \vdash X \searrow V \Uparrow \mathsf{Set} \rightsquigarrow j$ *and* $\Delta' \vdash X' \searrow V \Uparrow \mathsf{Set} \rightsquigarrow j'$ *for some* V *and* $j, j' \leq i$.

Proof. Simultaneously by induction on $X' : \mathsf{Set}_i$.

– *Case* $\mathsf{Fun}\, X'F' : \mathsf{Set}_i$ where $X' : \mathsf{Set}_i$ and $F' \cdot d : \mathsf{Set}_i$ for all $d : X'$.

  1. (In:) Show $\Delta \vdash e : \mathsf{Fun}\, X F \circledS \Delta' \vdash e' : \mathsf{Fun}\, X'F'$. Assume $\hat{\Delta} \leq \Delta, \hat{\Delta}' \leq \Delta'$ and $\hat{\Delta} \vdash d : X \circledS \hat{\Delta}' \vdash d' : X'$. By second induction hypothesis, $\hat{\Delta} \vdash d \searrow v \Uparrow X$ and $\hat{\Delta}' \vdash d' \searrow v \Uparrow X'$. By weakening, $\hat{\Delta} \vdash e \searrow u \Downarrow \mathsf{Fun}\, X F$ and $\hat{\Delta}' \vdash e' \searrow u \Downarrow \mathsf{Fun}\, X'F'$, hence, $\hat{\Delta} \vdash e\, d \searrow u\, v \Downarrow F \cdot d$ and $\hat{\Delta}' \vdash e'\, d' \searrow u\, v \Downarrow F' \cdot d'$. By first induction hypothesis, $\hat{\Delta} \vdash e\, d : F \cdot d \circledS \hat{\Delta}' \vdash e'\, d' : F' \cdot d$, and, since $\hat{\Delta}$, $\hat{\Delta}'$, d, and d' were arbitrary, the goal follows.
  2. (Out:) Show $\Delta \vdash f \searrow \lambda x v \Uparrow \mathsf{Fun}\, X F$ and $\Delta' \vdash f' \searrow \lambda x v \Uparrow \mathsf{Fun}\, X'F'$ for some x and v. It suffices to show $\Delta, x : X \vdash f \cdot x \searrow v \Uparrow F \cdot x$ and $\Delta', x : X' \vdash f' \cdot x \searrow v \Uparrow F' \cdot x$ for some $x \notin \mathsf{dom}(\Delta, \Delta')$. By first induction hypothesis, $\Delta, x : X \vdash x : X \circledS \Delta', x : X'| \vdash x : X'$, hence $\Delta, x : X \vdash f \cdot x : F \cdot x \circledS \Delta', x : X' \vdash f' \cdot x : F' \cdot x$. By the second induction hypothesis we get $\Delta, x : X \vdash f \cdot x \searrow v \Uparrow F \cdot x$ and $\Delta', x {:} X' \vdash f' \cdot x \searrow v \Uparrow F' \cdot x$.
  3. (Out-type:) Show $\Delta \vdash \mathsf{Fun}\, X F \searrow \mathsf{Fun}\, V\, \lambda x W \Uparrow \mathsf{Set} \rightsquigarrow j$ and $\Delta' \vdash \mathsf{Fun}\, X'F' \searrow \mathsf{Fun}\, V\, \lambda x W \Uparrow \mathsf{Set} \rightsquigarrow j'$ for $j, j' \leq i$. By induction hypothesis $\Delta \vdash X \searrow V \Uparrow \mathsf{Set} \rightsquigarrow j_1$ and $\Delta' \vdash X' \searrow V \Uparrow \mathsf{Set} \rightsquigarrow j_1'$ for $j_1, j_1' \leq i$. Since by the first induction hypothesis, $\Delta, x {:} X \vdash x : X \circledS \Delta', x {:} X' \vdash x : X'$, we have $\Delta, x : X \vdash F \cdot x : \mathsf{Set}_i \circledS \Delta', x : X' \vdash F' \cdot x : \mathsf{Set}_i$ and, by induction hypothesis, $\Delta, x : X \vdash F \cdot x \searrow W \Uparrow \mathsf{Set} \rightsquigarrow j_2$ and $\Delta', x {:} X' \vdash F' \cdot x \searrow W \Uparrow \mathsf{Set} \rightsquigarrow j_2'$ for $j_2, j_2' \leq i$. This entails the goal for $j := \max(j_1, j_2)$ and $j' := \max(j_1', j_2')$.

5.3 The Kripke model

We now define

$$\Delta \vdash d = d' : X \iff \Delta \vdash d : X \circledS \Delta \vdash d' : X.$$

We can view the relation $\Delta \vdash d = d' : X$ as inductively generated by the following rules:

$$\frac{}{\Delta \vdash \mathsf{Set}_i = \mathsf{Set}_i : \mathsf{Set}_j}\ i < j$$

$$\frac{\Delta \vdash E \searrow u \Downarrow \mathsf{Set}_i \qquad \Delta \vdash E' \searrow u \Downarrow \mathsf{Set}_i}{\Delta \vdash E = E' : \mathsf{Set}_j}\ i \leq j$$

$$\frac{\Delta \vdash e \searrow u \Downarrow E_1 \qquad \Delta \vdash e' \searrow u \Downarrow E_2 \qquad \Delta \vdash E_0, E_1, E_2 \searrow u' \Downarrow \mathsf{Set}_i}{\Delta \vdash e = e' : E_0}$$

$$\frac{\begin{array}{c}\Delta \vdash X = X' : \mathsf{Set}_i \\ \Delta' \vdash F \cdot d = F' \cdot d' : \mathsf{Set}_i \text{ for all } \Delta' \leq \Delta \text{ and } \Delta' \vdash d = d' : X\end{array}}{\Delta \vdash \mathsf{Fun}\, X F = \mathsf{Fun}\, X' F' : \mathsf{Set}_i}$$

$$\frac{\Delta' \vdash f \cdot d = f' \cdot d' : F \cdot d \text{ for all } \Delta' \leq \Delta \text{ and } \Delta' \vdash d = d' : X}{\Delta \vdash f = f' : \mathsf{Fun}\, X F}$$

$$\frac{}{\Delta \vdash \mathsf{N} = \mathsf{N} : \mathsf{Set}_i} \qquad \frac{\Delta \vdash e \searrow u \Downarrow \mathsf{N} \qquad \Delta \vdash e' \searrow u \Downarrow \mathsf{N}}{\Delta \vdash e = e' : \mathsf{N}}$$

$$\frac{}{\Delta \vdash \mathsf{z} = \mathsf{z} : \mathsf{N}} \qquad \frac{\Delta \vdash d = d' : \mathsf{N}}{\Delta \vdash \mathsf{s}\, d = \mathsf{s}\, d' : \mathsf{N}}$$

We let $\Delta \vdash X = X'$ iff there exists an i such that $\Delta \vdash X = X' : \mathsf{Set}_i$. We write $\Delta \vdash d : X$ for $\Delta \vdash d = d : X$.

5.4 Validity of Syntactic Typing

Now we can show that all the rules for our typing and equality judgements are valid in the model. As a byproduct, we get completeness of NbE.

Let

$$\Delta \vdash \rho = \rho' : \Gamma \iff \forall x \in \mathsf{dom}(\Gamma).\, \Delta \vdash \rho(x) = \rho'(x) : \Gamma(x)\rho$$

Lemma 9 (Environment extension). *If $\Delta \vdash \rho = \rho' : \Gamma$ and $\Delta \vdash d = d' : A\rho$ and $x \notin \mathsf{dom}(\Gamma)$ then $\Delta \vdash (\rho, x{=}d) = (\rho', x{=}d') : \Gamma, x{:}A$.*

Define $\Gamma \Vdash J$, meaning that $\Gamma \vdash J$ is valid in the Kripke model, as follows:

$$\begin{array}{lll}
\diamond \Vdash & :\iff & \text{true} \\
\Gamma, x{:}A \Vdash & :\iff & \Gamma \Vdash A \\
\Gamma \Vdash A & :\iff & \text{either } A = \mathsf{Set}_i \text{ and } \Gamma \Vdash \text{ or } \Gamma \Vdash A : \mathsf{Set}_i \text{ for some } i \\
\Gamma \Vdash t : A & :\iff & \Gamma \Vdash t = t : A \\
\Gamma \Vdash t = t' : A & :\iff & \Gamma \Vdash A \text{ and } \forall \Delta \vdash \rho = \rho' : \Gamma.\ \Delta \vdash t\rho = t'\rho' : A\rho \\
\Gamma \Vdash \sigma : \Gamma' & :\iff & \Gamma \Vdash \sigma = \sigma : \Gamma' \\
\Gamma \Vdash \sigma = \sigma' : \Gamma' & :\iff & \Gamma \Vdash \text{ and } \Gamma' \Vdash \text{ and } \forall \Delta \vdash \rho = \rho' : \Gamma.\ \Delta \vdash \sigma\rho = \sigma'\rho' : \Gamma'
\end{array}$$

Each inference rule can now be proven to be valid, for instance:

Lemma 10 (Validity of EQ-ESUBST-FUN-I).

$$\frac{\Gamma \Vdash \sigma : \Gamma' \qquad \Gamma', x\!:\!A \Vdash t : B}{\Gamma \Vdash (\lambda x t)\sigma = \lambda x.\, t(\sigma, x\!=\!x) : (\mathsf{Fun}\, A\, \lambda x B)\sigma}\ x \notin \mathsf{dom}(\Gamma)$$

Proof.

$\Gamma' \Vdash A$	by def.
$\Gamma', x\!:\!A \Vdash B$	by def.
$\Delta \vdash \rho = \rho' : \Gamma$	assumption
$\Delta' \leq \Delta$	assumption
$\Delta' \vdash d = d' : (A\sigma)\rho$	assumption
$\Delta' \vdash \rho = (\rho', x\!=\!d') : \Gamma$	since $x \notin \mathsf{dom}(\Gamma)$
$\Delta' \vdash \sigma\rho = \sigma(\rho', x\!=\!d') : \Gamma'$	from $\Gamma \Vdash \sigma : \Gamma'$
$\Delta' \vdash (\sigma\rho, x\!=\!d) = (\sigma(\rho', x\!=\!d'), x\!=\!d') : \Gamma', x\!:\!A$	Lemma 9
$\Delta' \vdash t(\sigma\rho, x\!=\!d) = t(\sigma(\rho', x\!=\!d'), x\!=\!d') : B(\sigma\rho, x\!=\!d)$	from $\Gamma', x\!:\!A \Vdash t : B$
$\Delta' \vdash t(\sigma\rho, x\!=\!d) = (t(\sigma, x\!=\!x))(\rho', x\!=\!d') : B(\sigma\rho, x\!=\!d)$	equalities in D
$\Delta' \vdash (\lambda x t)\sigma\rho \cdot d = (\lambda x.\, t(\sigma, x\!=\!x))\rho' \cdot d' : (\lambda x.B)\sigma\rho \cdot d$	by APP-FUN
$\Delta \vdash ((\lambda x t)\sigma)\rho = (\lambda x.\, t(\sigma, x\!=\!x))\rho' : \mathsf{Fun}\,(A\sigma\rho)\,(\lambda x B)\sigma\rho$	goal

Theorem 1 (Soundness of the inference rules). *If $\Gamma \vdash J$ then $\Gamma \Vdash J$.*

Proof. By induction on $\Gamma \vdash J$. (Tedious, but easy.)

Lemma 11. $\Gamma\rho_{\mathsf{id}} \vdash \rho_{\mathsf{id}} = \rho_{\mathsf{id}} : \Gamma$.

Proof. For each $x \in \mathsf{dom}(\Gamma)$, we have $\Gamma\rho_{\mathsf{id}} \vdash x\rho_{\mathsf{id}} = x\rho_{\mathsf{id}} : \Gamma(x)\rho_{\mathsf{id}}$.

Corollary 1 (Completeness of NbE). *If $\Gamma \vdash t = t' : A$ then $\Gamma\rho_{\mathsf{id}} \vdash t\rho_{\mathsf{id}} \searrow v \Uparrow A\rho_{\mathsf{id}}$ and $\Gamma\rho_{\mathsf{id}} \vdash t'\rho_{\mathsf{id}} \searrow v \Uparrow A\rho_{\mathsf{id}}$ for some v.*

6 Kripke Logical Relation and Soundness of NbE

In the previous section we have defined a logical relation $\Delta \vdash d : X$ Ⓢ $\Delta' \vdash d' : X'$ between two semantic objects in their typing environments. Now we will define a logical relation between a well-typed expression $\Gamma \vdash t : A$ and a value d in its typing environment $\Delta \vdash X$. The relation shall imply that d at type X reifies in Δ to a normal form v which is judgmentally equal to t at type A in context Γ (Lemma 15). The construction is similar to [ACD07] but has Δ as additional parameter.

We write $\Gamma' \leq \Gamma$ if Γ' is a well-formed extension of Γ, meaning $\Gamma' = \Gamma, \Gamma_0$ for some Γ_0 and $\Gamma' \vdash$. By induction on $X : \mathsf{Set}_i$ we define the relation

$$\Gamma \vdash t : C \circledR \Delta \vdash d : X$$

between well-typed terms $\Gamma \vdash t : C$ and semantic objects $\Delta \vdash d : X$.

$$\begin{array}{l}
\Gamma \vdash r : C \circledR \Delta \vdash f : \mathsf{Fun}\, X F \iff \\
\quad \Gamma \vdash C = \mathsf{Fun}\, A\, \lambda x B : \mathsf{Set}_i \text{ for some } A, B \text{ and} \\
\quad \Gamma \vdash A : \mathsf{Set}_i \circledR \Delta \vdash X : \mathsf{Set}_i \text{ and} \\
\quad \Gamma' \vdash r\, s : B(\sigma_{\mathsf{id}}, x{=}s) \circledR \Delta' \vdash f\, d : F\, d \text{ for all } \Gamma' \leq \Gamma, \Delta' \leq \Delta \\
\qquad \text{and } \Gamma' \vdash s : A \circledR \Delta' \vdash d : X
\end{array}$$

$$\begin{array}{l}
\Gamma \vdash r : C \circledR \Delta \vdash \mathsf{Fun}\, X F : \mathsf{Set}_i \iff \\
\quad \Gamma \vdash C = \mathsf{Set}_i : \mathsf{Set}_{i+1} \\
\quad \Gamma \vdash r = \mathsf{Fun}\, A\, \lambda x B : \mathsf{Set}_i \text{ for some } A, B \text{ and} \\
\quad \Gamma \vdash A : \mathsf{Set}_i \circledR \Delta \vdash X : \mathsf{Set}_i \text{ and} \\
\quad \Gamma' \vdash B(\sigma_{\mathsf{id}}, x{=}s) : \mathsf{Set}_i \circledR \Delta' \vdash F\, d : \mathsf{Set}_i \text{ for all } \Gamma' \leq \Gamma, \Delta' \leq \Delta \\
\qquad \text{and } \Gamma \vdash s : A \circledR \Delta \vdash d : X
\end{array}$$

$$\begin{array}{l}
\Gamma \vdash r : C \circledR \Delta \vdash d : X \iff \Delta \vdash d \searrow v \Uparrow X \text{ and } \Gamma \vdash r = v : C \\
\quad \text{for } X \in \{e, \mathsf{N}\} \text{ or } X = \mathsf{Set}_i \text{ and } d \neq \mathsf{Fun}\, Y F
\end{array}$$

The logical relation is closed under weakening of contexts (both the syntactic, Γ, and the semantic one, Δ) and under judgmental and Kripke model equality, meaning one can always trade expressions and values for equals without violating the relation.

Lemma 12 (Weakening). *If $\Gamma \vdash r : C \circledR \Delta \vdash d : X$ and $\Gamma' \leq \Gamma$, $\Delta' \leq \Delta$, then $\Gamma' \vdash r : C \circledR \Delta' \vdash d : X$.*

Lemma 13. *Let $\Gamma \vdash C = C' : \mathsf{Set}_i$ and $\Gamma \vdash r = r' : C$. If $\Gamma \vdash r : C \circledR \Delta \vdash d : X$ then $\Gamma \vdash r' : C' \circledR \Delta \vdash d : X$.*

Proof. By induction on $\Delta \vdash X$.

Lemma 14. *Let $\Delta \vdash X = X' : \mathsf{Set}_i$ and $\Delta \vdash d = d' : X$. If $\Gamma \vdash r : C \circledR \Delta \vdash d : X$ then $\Gamma \vdash r : C \circledR \Delta \vdash d' : X'$.*

Proof. By induction on $\Delta \vdash X = X' : \mathsf{Set}_i$.

– *Case* $X = E_2$ and $X' = E_2'$ and

$$\frac{\Delta \vdash E_2, E_2' \searrow u' \Downarrow \mathsf{Set}_{i'}}{\Delta \vdash E_2 = E_2' : \mathsf{Set}_i}\; i' \leq i$$

By definition of $\Gamma \vdash r : C \circledR \Delta \vdash d : X$ we have $d = e$ neutral, $\Gamma \vdash r = u : C$ and

$$\frac{\Delta \vdash e \searrow u \Downarrow E_1 \qquad \Delta \vdash E_1, E_2 \searrow u'' \Downarrow \mathsf{Set}_{i''}}{\Delta \vdash e \searrow u \Uparrow E_2}$$

Since reification is deterministic, $u'' = u'$ and $i'' = i'$. By inversion on $\Delta \vdash d = d' : E_2$, $d' = e'$ for some e' and both $\Delta \vdash e' \searrow u \Downarrow E_1'$ and $\Delta \vdash E_1', E_2 \searrow u' \Downarrow \mathsf{Set}_{i'}$. Hence $\Delta \vdash E_1', E_2' \searrow u' \Downarrow \mathsf{Set}_{i'}$. Thus, $\Gamma \vdash r : C \circledR \Delta \vdash d' : X'$.

Lemma 15 (Into and out of the logical relation / escape lemma). *Let* $\Gamma \vdash C : \mathsf{Set}_i \circledR \Delta \vdash X : \mathsf{Set}_i$.

1. *(In:) If* $\Gamma \vdash u : C$ *and* $\Delta \vdash e \searrow u \Downarrow X$ *then* $\Gamma \vdash u : C \circledR \Delta \vdash e : X$.
2. *(Out:) If* $\Gamma \vdash r : C \circledR \Delta \vdash d : X$ *then* $\Delta \vdash d \searrow v \Uparrow X$ *for some* v *with* $\Gamma \vdash r = v : C$.
3. *(Out-type:)* $\Delta \vdash X \searrow V \Uparrow \mathsf{Set} \rightsquigarrow j$ *for* $j \leq i$ *and* $\Gamma \vdash C = V : \mathsf{Set}_i$.

Proof. By induction on $X : \mathsf{Set}_i$.

Fundamental theorem. We relate substitutions $\Gamma' \vdash \sigma : \Gamma$ to environments $\Delta \vdash \rho : \Gamma$ by the following definition:

$$\begin{aligned} \Gamma' \vdash \sigma \circledR \Delta \vdash \rho :: \Gamma \iff & \text{ for all } x \in \mathsf{dom}(\Gamma), \\ & \Gamma' \vdash x\sigma : \Gamma(x)\sigma \circledR \Delta \vdash \rho(x) : \Gamma(x)\rho \end{aligned}$$

Theorem 2 (Fundamental theorem of logical relations). *If* $\Gamma \vdash t : A$ *then* $\Gamma' \vdash t\sigma : A\sigma \circledR \Delta \vdash t\rho : A\rho$ *for all* $\Gamma' \vdash \sigma \circledR \Delta \vdash \rho :: \Gamma$.

Proof. By induction on $\Gamma \vdash t : A$.

Lemma 16. $\Gamma \vdash \sigma_{\mathsf{id}} \circledR (\Gamma\rho_{\mathsf{id}}) \vdash \rho_{\mathsf{id}} :: \Gamma$.

Proof. We have to show $\Gamma \vdash x : \Gamma(x) \circledR \Gamma\rho_{\mathsf{id}} \vdash x : \Gamma(x)\rho_{\mathsf{id}}$ for all $x \in \mathsf{dom}(\Gamma)$. This holds by Lemma 15 since $\Gamma\rho_{\mathsf{id}} \vdash x \searrow x \Downarrow (\Gamma\rho_{\mathsf{id}})(x)$.

Corollary 2 (Soundness of NbE). *If* $\Gamma \vdash t : A$ *then there is some* v *such that* $\Gamma\rho_{\mathsf{id}} \vdash t\rho_{\mathsf{id}} \searrow v \Uparrow A\rho_{\mathsf{id}}$ *and* $\Gamma \vdash t = v : A$.

7 Completeness of Algorithmic Equality

In this section, we conclude the completeness of the equality algorithm from the completeness of NbE. In particular, if two values reify to the same normal form, they are algorithmically equal. Some care has to be paid to the case of functional values. It could be that f reifies to λxv since $f \cdot x$ reifies to v and f' reifies to $\lambda x'v' =_\alpha \lambda xv$ since $f' \cdot x'$ reifies to v'. Now we want to conclude that f and f' are algorithmically equal, which requires $f \cdot \mathsf{x}_\Delta$ equal to $f' \cdot \mathsf{x}_\Delta$. But the induction hypothesis is not applicable since x, x', x_Δ might be different variables, hence, $f \cdot x$ and $f \cdot \mathsf{x}_\Delta$ are different objects. If our values are actually normal *expressions*, we could use plain old α-conversion and rename the variables. However, we are dealing with values in an arbitrary syntactical applicative structure D!

We restrict the class of possible models to those that admit renaming of free variables. This is a sensible restriction since values should be *parametric* in the names of their free variables—case distinction on the name of a identifier is not a desirable program behavior.

Renamings. Let π be a bijective map from variables to variables. We assume a renaming operations $d\pi$ on values $d \in \mathsf{D}$ with the following properties:

$$
\begin{array}{lll}
\textsc{ren-c} & c\pi & = c \\
\textsc{ren-var} & x\pi & = \pi(x) \\
\textsc{ren-app} & (f \cdot d)\pi & = f\pi \cdot d\pi \\
\textsc{ren-eval} & (t\rho)\pi & = t(\rho\pi)
\end{array}
$$

Renaming can be defined for many syntactical applicative structures D: term models, explicit substitutions, closures, even Scott models which evaluate an abstraction $(\lambda xt)\rho$ to an actual function $h : \mathsf{D} \to \mathsf{D}$ with $h(d) = t(\rho, x = d)$. Setting $(h\pi)(d) = (h(d\pi^{-1}))\pi$ we have $(h(d))\pi = (h\pi)(d\pi)$ [Pit06].

We define renaming of contexts $\pi(\Delta)$ by

$$
\begin{array}{lll}
\textsc{ren-cxt-empty} & \pi(\diamond) & = \diamond \\
\textsc{ren-cxt-ext} & \pi(\Delta, x \!:\! X) & = \pi(\Delta), \pi(x) \!:\! X\pi
\end{array}
$$

Remark 2. This is not to be confused with the operation $\Delta\pi$ which is composition defined by $(\Delta\pi)(x) = \Delta(x)\pi$. We have $\pi(\Delta)(x\pi) = \Delta(x)\pi$. Thus, $\pi(\Delta) = \pi^{-1}\Delta\pi$ is a conjugation.

Lemma 17 (Completeness of algorithmic equality w. r. t. reification). *Let $\Delta = x_1 \!:\! X_1, \ldots x_n \!:\! X_n$ be a semantic context and π a permutation of names assigning the ith de Bruijn level to variable x_i ($\pi(x_i) = \mathsf{x}_i$ for $i = 1..n$).*

1. *If $\Delta \vdash e \searrow u \Downarrow X$ and $\Delta' \vdash e' \searrow u \Downarrow X'$ then $\pi(\Delta) \vdash e\pi = e'\pi \Downarrow X\pi$.*
2. *If $\Delta \vdash d \searrow v \Uparrow X$ and $\Delta' \vdash d' \searrow v \Uparrow X'$ then $\pi(\Delta) \vdash d\pi = d'\pi \Uparrow X\pi$.*
3. *If $\Delta \vdash X \searrow V \Uparrow \mathsf{Set} \rightsquigarrow i$ and $\Delta' \vdash X' \searrow V \Uparrow \mathsf{Set} \rightsquigarrow i'$ then $\pi(\Delta) \vdash X\pi = X'\pi \Uparrow \mathsf{Set} \rightsquigarrow \max(i, i')$.*

Proof. Simultaneously by induction on the first derivation.

– *Case*

$$\frac{\Delta \vdash e \searrow u \Downarrow e_1 \qquad \Delta \vdash e_1 \searrow u' \Downarrow \mathsf{Set}_i \qquad \Delta \vdash e_2 \searrow u' \Downarrow \mathsf{Set}_j}{\Delta \vdash e \searrow u \Uparrow e_2}$$

By inversion on the second derivation we get $\Delta' \vdash e' \searrow u \Downarrow \_$. By induction hypotheses, $\pi(\Delta) \vdash e\pi = e'\pi \Downarrow e_1\pi$ and $\pi(\Delta) \vdash e_1\pi = e_2\pi \Downarrow \mathsf{Set}_i$. Thus, $\pi(\Delta) \vdash e\pi = e'\pi \Uparrow e_2\pi$.

– *Case*

$$\frac{\Delta, x \!:\! X \vdash f \cdot x \searrow v \Uparrow F \cdot x}{\Delta \vdash f \searrow \lambda xv \Uparrow \mathsf{Fun}\, X F} \; x \notin \mathsf{dom}(\Delta)$$

Let π' be a permutation satisfying $\pi'(x) = \mathsf{x}_\Delta$ and $\pi'(y) = \pi(y)$ for $y \in \mathsf{dom}(\Delta)$. By induction hypothesis, $\pi(\Delta, x : X) \vdash (f \cdot x)\pi = (f' \cdot x)\pi \Uparrow (F \cdot x)\pi$, which means $\pi(\Delta), \mathsf{x}_\Delta \!:\! X\pi \vdash f\pi \cdot \mathsf{x}_\Delta = f'\pi \cdot \mathsf{x}_\Delta \Uparrow F\pi \cdot \mathsf{x}_\Delta$. Hence, $\pi(\Delta) \vdash f\pi = f'\pi \Uparrow (\mathsf{Fun}\, X\, F)\pi$.

A name permutation π can be viewed as an environment, taking a variable x to $\pi(x)$. This explains the notation $t\pi$, and $\pi(\Gamma)$ which satisfies $\pi(\Gamma)(x\pi) = \Gamma(x)\pi$. Let $\pi_{(x_1{:}A_1,\dots x_n{:}A_n)}(x_i) = \mathsf{x}_i$ for $i = 1..n$.

Corollary 3 (Completeness of algorithmic equality). *If* $\Gamma \vdash t = t' : A$ *then* $\pi_\Gamma(\Gamma) \vdash t\pi_\Gamma = t'\pi_\Gamma \Uparrow A\pi_\Gamma$.

Proof. We have $\Gamma\rho_{\mathsf{id}} \vdash t\rho_{\mathsf{id}}, t'\rho_{\mathsf{id}} \searrow \_ \Uparrow A\rho_{\mathsf{id}}$ by Cor. 1, thus, the corollary follows from Lemma 17 with $\rho_{\mathsf{id}}\pi_\Gamma = \pi_\Gamma$.

8 Strong Semantic Typing and Soundness of Algorithmic Equality

In this section, we tackle the second problem: soundness of the equality algorithm. We show that if two values are algorithmically equal, then they reify to the same normal form. Rule AQ-FUN gives us a hard time:

$$\text{AQ-FUN}\ \frac{\Delta \vdash e = e' \Downarrow \mathsf{Fun}\, X F \qquad \Delta \vdash d = d' \Uparrow X}{\Delta \vdash e\, d = e'\, d' \Downarrow F \cdot d}$$

Using the induction hypothesis, we can show $\Delta \vdash e'\, d' \searrow u\, v \Downarrow F \cdot d'$ but we need $F \cdot d$! The fact that d and d' reify to the same normal form does not help us here, we need a stronger induction hypothesis.

Again, soundness of an algorithmic equality on *syntax* is usually trivial: one simply shows that each algorithmic rule is an instance of an inference rule. When trying to apply this intuition to our semantic typing and equality, the Kripke model, one realizes that it lacks a substitution principle: *Judgements remain valid when substituting a term of the right type for a variable.* In the following we will equip our semantics with a substitution principle by brute force! We define *strong* semantic judgements (a *super* Kripke model) to hold if the weak semantic judgements (Kripke model) hold for all well-typed valuations of free variables (cf. hypothetical judgements [CPT05]). As precondition, we need a model in which we can reevaluate values in a new environment.

Reevaluation. For the following, impose more conditions on the model D. It must be equipped with a reevaluation function $d\theta$ of value d in environment θ, satisfying the following laws:

REEVAL-ID	$d\,\rho_{\mathsf{id}}$	$= d$
REEVAL-C	$c\theta$	$= c$
REEVAL-VAR	$x\theta$	$= \theta(x)$
REEVAL-FUN-E	$(f \cdot d)\theta$	$= f\theta \cdot d\theta$
REEVAL-ESUBST	$(t\rho)\theta$	$= t(\rho\theta)$

Reevaluation is easy to define on the syntactical applicative structure of closures; simply add $([\lambda x t]\rho)\theta = [\lambda x t](\rho\theta)$ and $(e\, d)\theta = e\theta \cdot d\theta$ to the above laws.

For Scott models, it is a bit more problematic. If $f \in \mathsf{D} \to \mathsf{D}$, we can set $(f\theta)(d) = (f(x))((\theta, x{=}d)$ for a fresh variable x. Freshness can be defined if we construct D as a nominal set [Shi05]. However, not all continuous functions f will fulfill REEVAL-FUN-E: for $(f(d))\theta = (f(x))(\theta, x{=}d\theta)$ to hold f must treat variables parametrically. For example, the function $f(d) = d$ if d not a variable and $f(x) = \mathsf{Set}_0$ for x a variable is not parametric and, thus, not compatible with reevaluation. The formulation of suitable parametricity conditions and the proof of parametricity for all values in the Kripke model remains open.

8.1 Super Kripke model

Let Θ range over semantic contexts and let

$$\Delta \vdash \rho = \rho' : \Theta \iff \forall x \in \mathsf{dom}(\Theta).\ \Delta \vdash x\rho = x\rho' : \Theta(x)\rho$$

We write $\Delta \vdash \rho : \Theta$ iff $\Delta \vdash \rho = \rho : \Theta$.

Lemma 18. *If $\Delta \vdash \rho : \Theta$ and $x \notin \mathsf{dom}(\Theta)$ then $\Delta \vdash \rho = (\rho, x{=}d) : \Theta$.*

We define strong semantic equality by

$$\Theta \models d = d' : X \iff \forall \Delta \vdash \rho = \rho' : \Theta.\ \Delta \vdash d\rho = d'\rho' : X\rho$$

Let $\Theta \models X = X'$ iff $\Theta \models X = X' : \mathsf{Set}_i$ for some i. We write $\Theta \models X$ for $\Theta \models X = X$ and $\Theta \models d : X$ for $\Theta \models d = d : X$, and say d is semantically strongly typed of type X in context Θ.

Lemma 19 (Weakening).

1. *If $\Delta' \leq \Delta$, $\Delta \vdash \rho = \rho' : \Theta$, and $\Theta \leq \Theta'$ then $\Delta' \vdash \rho = \rho' : \Theta'$.*
2. *If $\Theta \leq \Theta'$ and $\Theta' \models d = d' : X$ then $\Theta \models d = d' : X$.*

Proof. For 2., Assume $\Delta \vdash \rho = \rho' : \Theta$. By 1, $\Delta \vdash \rho = \rho' : \Theta'$, thus, $\Delta \vdash d\rho = d'\rho' : X\rho$.

Lemma 20 (Valuation extension). *Let $\Theta \models X$ and $x \notin \mathsf{dom}(\Theta)$. If $\Delta \vdash \rho = \rho' : \Theta$ and $\Delta \vdash d = d' : X\rho$ then $\Delta \vdash (\rho, x{=}d) = (\rho', x{=}d') : \Theta, x{:}X$.*

Proof. The first goal $\Delta \vdash (\rho, x{=}d) = (\rho', x{=}d') : \Theta$ follows since $x \notin \mathsf{dom}(\Theta)$. The second goal, $\Delta \vdash d = d' : X(\rho, x{=}d)$ follows from $\Delta \vdash X\rho = X(\rho, x{=}d)$ which in turn is a consequence of $\Theta \models X$.

Since $\Delta \vdash \rho_{\mathsf{id}} = \rho_{\mathsf{id}} : \Delta$, strong semantic equality $\Delta \models d = d' : X$ implies weak semantic equality $\Delta \vdash d = d' : X$.

Lemma 21 (Admissible rules). *The following implications, written as rules, hold for strong semantic equality.*

$$\frac{}{\Theta \models x = x : \Theta(x)}\ x \in \mathsf{dom}(\Theta)$$

$$\frac{\Theta \models f = f' : \mathsf{Fun}\, X F \qquad \Theta \models d = d' : X}{\Theta \models f \cdot d = f' \cdot d' : F \cdot d}$$

$$\frac{\begin{array}{c}\Theta \models \mathsf{Fun}\, X F \qquad \Theta \models f : \mathsf{Fun}\, X F \qquad \Theta \models f' : \mathsf{Fun}\, X F \\ \Theta, x\!:\!X \models f \cdot x = f' \cdot x : F \cdot x\end{array}}{\Theta \models f = f' : \mathsf{Fun}\, X F}$$

$$\frac{\begin{array}{c}\Theta \models \mathsf{Fun}\, X F,\ \mathsf{Fun}\, X' F' : \mathsf{Set}_i \\ \Theta \models X = X' : \mathsf{Set}_i \qquad \Theta, x\!:\!X \models F \cdot x = F' \cdot x : \mathsf{Set}_i\end{array}}{\Theta \models \mathsf{Fun}\, X F = \mathsf{Fun}\, X' F' : \mathsf{Set}_i}$$

$$\frac{\Theta \models d = d' : X \qquad \Theta \models X = X'}{\Theta \models d = d' : X'}$$

Proof. The soundness of the application rule relies on distributivity of valuation over application, $(f \cdot d)\rho = f\rho \cdot d\rho$.

We verify the extensionality rule: Assume $\Delta \vdash \rho = \rho' : \Theta$ and show $\Delta \vdash f\rho = f'\rho' : \mathsf{Fun}\,(X\rho)\,(F\rho)$. To this end assume $\Delta' \leq \Delta$ and $\Delta' \vdash d = d' : X\rho$ and show $\Delta' \vdash f\rho \cdot d = f'\rho' \cdot d' : F\rho \cdot d$. By Lemma 20 we have $\Delta' \vdash (\rho, x = d) = (\rho', x = d') : \Theta, x\!:\!X$. Now the main hypothesis gives us $\Delta' \vdash (f \cdot x)(\rho, x = d) = (f' \cdot x)(\rho', x = d') : (F \cdot x)(\rho, x = d)$, which simplifies to $\Delta' \vdash f(\rho, x = d) \cdot d = f'(\rho', x = d') \cdot d' : F(\rho, x = d) \cdot d$. From hypothesis $\Theta \models \mathsf{Fun}\, X F$ we infer $\Delta' \vdash (\mathsf{Fun}\, X F)\rho = (\mathsf{Fun}\, X F)(\rho, x = d)$, which entails $\Delta' \vdash F\rho \cdot d = F(\rho, x = d) \cdot d$. Similarly, we obtain $\Delta' \vdash f\rho \cdot d = f(\rho, x = d) \cdot d : F \cdot d$ and $\Delta' \vdash f'\rho \cdot d = f'(\rho, x = d) \cdot d : F \cdot d$. Thus, we conclude $\Delta' \vdash f\rho \cdot d = f'\rho' \cdot d' : F\rho \cdot d$.

Semantic context equality $\models \Theta = \Theta'$ is given inductively by

$$\frac{}{\models \diamond = \diamond} \qquad \frac{\models \Theta = \Theta' \qquad \Theta \models X = X' : \mathsf{Set}_i}{\models \Theta, x\!:\!X = \Theta', x\!:\!X'}\, x \notin \mathsf{dom}(\Theta)$$

We write $\models \Theta$ for $\models \Theta = \Theta$.

Lemma 22 (Soundness of algorithmic equality). *Let* $\models \Delta$.

1. *If* $\Delta \vdash e = e' \Downarrow X$ *then* $\Delta \models X$ *and* $\Delta \models e = e' : X$.
2. *If* $\Delta \vdash d = d' \Uparrow X$ *and* $\Delta \models X$ *and* $\Delta \models d, d' : X$ *then* $\Delta \models d = d' : X$.
3. *If* $\Delta \vdash X = X' \Uparrow \mathsf{Set} \rightsquigarrow i$ *and* $\Delta \models X, X' : \mathsf{Set}_i$ *then* $\Delta \vdash X = X' : \mathsf{Set}_i$.

Proof. Simultaneously by induction on the derivation of algorithmic equality using the admissible rules.

– *Case*
$$\frac{\Delta \vdash e = e' \Downarrow E_1 \qquad \Delta \vdash E_1 = E_2 \Downarrow \mathsf{Set}_i}{\Delta \vdash e = e' \Uparrow E_2}$$

By induction hypotheses, $\Delta \models e = e' : E_1$ and $\Delta \models E_1 = E_2 : \mathsf{Set}_i$. Thus, $\Delta \models e = e' : E_2$.

8.2 Strong Validity of Syntactic Typing

In the following, we establish that our inference rules are also valid in the super Kripke model. However, no new inductive proof is needed. We have already shown that the rules are valid in the weak semantics (Kripke model) under all weakly typed environments. This is actually equivalent to being valid in the strong semantics (super Kripke model) under all strongly typed environments.

We define strongly typed environments by

$$\Delta \models \rho = \rho' : \Gamma \iff \forall x \in \mathsf{dom}(\Gamma).\, \Delta \models \rho(x) = \rho'(x) : \Gamma(x)\rho.$$

Lemma 23 (Composing strongly and weakly typed environments). *If $\Theta \models \rho = \rho' : \Gamma$ and $\Delta \vdash \theta = \theta' : \Theta$ then $\Delta \vdash \rho\theta = \rho'\theta' : \Gamma$.*

Proof. Show for each $x \in \mathsf{dom}(\Gamma)$ that $\Delta \vdash x\rho\theta = x\rho'\theta' : \Gamma(x)\rho\theta$. By assumption $\Theta \models x\rho = x\rho' : \Gamma(x)\rho$, which implies the goal.

Define $\Gamma \models\!\!\models J$, meaning that $\Gamma \vdash J$ is valid in the super Kripke model, as follows:

$$\begin{array}{lll}
\diamond \models\!\!\models & :\iff & \text{true} \\
\Gamma, x\!:\!A \models\!\!\models & :\iff & \Gamma \models\!\!\models A \\
\Gamma \models\!\!\models A & :\iff & \text{either } A = \mathsf{Set}_i \text{ and } \Gamma \models\!\!\models \text{ or } \Gamma \models\!\!\models A : \mathsf{Set}_i \text{ for some } i \\
\Gamma \models\!\!\models t : A & :\iff & \Gamma \models\!\!\models t = t : A \\
\Gamma \models\!\!\models t = t' : A & :\iff & \Gamma \models\!\!\models A \text{ and } \forall \Delta \models \rho = \rho' : \Gamma.\ \Delta \models t\rho = t'\rho' : A\rho \\
\Gamma \models\!\!\models \sigma : \Gamma_0 & :\iff & \Gamma \models\!\!\models \sigma = \sigma : \Gamma_0 \\
\Gamma \models\!\!\models \sigma = \sigma' : \Gamma_0 & :\iff & \Gamma \models\!\!\models \text{ and } \Gamma_0 \models\!\!\models \text{ and} \\
 & & \quad \forall \Delta \models \rho = \rho' : \Gamma.\ \Delta \models \sigma\rho = \sigma'\rho' : \Gamma_0
\end{array}$$

Lemma 24 (Validity: weak implies strong). *If $\Gamma \Vdash J$ then $\Gamma \models\!\!\models J$.*

Proof. The hypothesis is $\forall \Delta \vdash \rho = \rho' : \Gamma.\ \Delta \vdash t\rho = t'\rho' : A\rho$. Assume $\Theta \models \rho = \rho' : \Gamma$ and $\Delta \vdash \theta = \theta' : \Theta$ and show $\Delta \vdash t\rho\theta = t'\rho'\theta' : A\rho\theta$. By Lemma 23 $\Delta \vdash \rho\theta = \rho'\theta' : \Gamma$, hence our goal follows by assumption.

Corollary 4. *If $\Gamma \vdash J$ then $\Gamma \models\!\!\models J$.*

Lemma 25. *$\Gamma\rho_{\mathsf{id}} \models \rho_{\mathsf{id}} = \rho_{\mathsf{id}} : \Gamma$.*

Corollary 5. *If* $\Gamma \vdash t : A$ *then* $\Gamma\rho_{\mathsf{id}} \models t\rho_{\mathsf{id}} : A\rho_{\mathsf{id}}$.

Putting things together:

Theorem 3 (Soundness of algorithmic equality). *If* $\Gamma \vdash t, t' : A$ *and* $\Gamma\rho_{\mathsf{id}} \vdash t\rho_{\mathsf{id}} = t'\rho_{\mathsf{id}} \Uparrow A\rho_{\mathsf{id}}$ *then* $\Gamma \vdash t = t' : A$.

Proof. We have $\Gamma \vdash t : A$ Ⓡ $\Gamma\rho_{\mathsf{id}} \vdash t\rho_{\mathsf{id}} : A\rho_{\mathsf{id}}$ and $\Gamma \vdash t' : A$ Ⓡ $\Gamma\rho_{\mathsf{id}} \vdash t'\rho_{\mathsf{id}} : A\rho_{\mathsf{id}}$. Also $\models \Gamma\rho_{\mathsf{id}}$, $\Gamma\rho_{\mathsf{id}} \models A\rho_{\mathsf{id}}$, $\Gamma\rho_{\mathsf{id}} \models t\rho_{\mathsf{id}} : A\rho_{\mathsf{id}}$, and $\Gamma\rho_{\mathsf{id}} \models t'\rho_{\mathsf{id}} : A\rho_{\mathsf{id}}$. By semantic soundness of the algorithm, $\Gamma\rho_{\mathsf{id}} \models t\rho_{\mathsf{id}} = t'\rho_{\mathsf{id}} : A\rho_{\mathsf{id}}$ which implies $\Gamma\rho_{\mathsf{id}} \vdash t\rho_{\mathsf{id}} = t'\rho_{\mathsf{id}} : A\rho_{\mathsf{id}}$, hence $\Gamma \vdash t : A$ Ⓡ $\Gamma\rho_{\mathsf{id}} \vdash t'\rho_{\mathsf{id}} : A\rho_{\mathsf{id}}$. This entails $\Gamma \vdash t = t' : A$.

9 On Termination

What remains to show is termination of the algorithmic equality: Given two terms of the same type, the algorithm terminates on their values. We have already seen that the values of well-typed terms are reifiable, hence the NbE-algorithm, which compares the results of reification, is terminating. We would like to extend this result to algorithmic equality, which performs reification incrementally. One would expect a statement similar to Lemma 17:

> If $\Delta \vdash d \searrow \_ \Uparrow X$ and $\Delta \vdash d' \searrow \_ \Uparrow X$ then the query $\Delta \vdash d = d' \Uparrow X$ terminates.

Generalizing this statement to the mutually defined notions of reification and algorithmic equality ($\Downarrow$ and $\Uparrow \mathsf{Set}$), we see that the proof fails since during reification of d', context and type diverge from Δ and X. (Similar considerations lead us to the definition of Ⓢ in Section 5.2.)

The skeleton of the algorithmic derivation is already determined by the derivation of $\Delta \vdash d \searrow \_ \Uparrow X$. Yet we have to show that the application $f' \cdot \mathsf{x}_\Delta$ is terminating in AQ-EXT. Somehow we have to exploit that d' originates from a well-typed term, thus, $\Delta \vdash d' : X$ and even $\Delta \models d : X$ both hold. But it is unclear how to make use of these facts in a termination proof.

10 Conclusion and Related Work

We have presented a bidirectional incremental $\beta\eta$-equality algorithm for a dependent type theory with predicative universes and verified it using NbE-techniques. The algorithm is formulated with respect to an abstract representation of values which supports several implementations. In Sec. 8 we had to exclude the representation via higher order abstract syntax, which was used in an early version of Agdalight, a predecessor of Agda 2. In the future we want to explore how to generalize our proof so that we also cover this implementation technique. Furthermore, we want to close the gap and prove termination of the algorithm as well as soundness and completeness of type checking.

Some complications in the verification vanish if we restrict to a concrete, term-like implementation of values, for instance, closures [Coq96,AC08]. Closures are a special case of explicit substitutions, hence, soundness of algorithmic equality is trivial: each algorithmic rule can be replaced by a sequence of declarative rules. Termination also becomes apparent: algorithmic equality works only on well-typed terms, which are normalizing.

Related work. The current proof uses similar techniques to those in our previous work on NbE [ACD07]. However, there are several differences, in order to deal with contextual reification we here need a Kripke model instead of a plain PER model.

Goguen [Gog94] proves decidability of UTT using typed operational semantics. He treats η, universes, and even inductive types and a impredicative universe of propositions. Showing soundness and completeness of his syntactic Kripke model he establishes subject reduction, confluence, and strong normalization, which imply decidability. However, he is not concerned about particular algorithms. Since his approach is based on η-reduction instead of η-expansion, it is not clear whether it scales to a unit type with extensional equality.

Harper and Pfenning [HP05] present an incremental bidirectional $\beta\eta$-equality algorithm for LF using erasure of dependencies; this does not extend to large eliminations. Chapman, Altenkirch, and McBride [CAM07] share their algorithm with us. They describe an implementation, but no verification.

Grégoire and Leroy [GL02] have implemented an incremental β-conversion test for the Calculus of Inductive Constructions based on "normalization by execution". Values are computed by compiling (open!) expression to Caml byte code. The result of executing the code is then *read back* to a β-normal expression. The soundness of this efficient form of evaluation has been formally verified in Coq. We expect that our and their work can be combined to obtain an efficient $\beta\eta$-conversion test. To this end, one needs to instantiate the syntactical applicative structure D to machine code and define contextual reification of code, similar to Grégoire and Leroy's *read back* function.

Acknowledgments. Thanks to the anonymous referees who gave insightful comments and pointed to the problem of termination.

A Proofs of Syntactical Properties

Lemma 26 (Derivable rules).

$$\textsc{eq-subst-empty-}\beta\ \frac{\Gamma \vdash \sigma : \Gamma'}{\Gamma \vdash () \circ \sigma = () : \diamond} \qquad \textsc{eq-subst-empty}\ \frac{\Gamma \vdash}{\Gamma \vdash () = () : \diamond}$$

$$\textsc{eq-esubst-weak}\ \frac{\Gamma \vdash \sigma : \Gamma' \qquad \Gamma', x{:}A \vdash \qquad \Gamma \vdash t : A\sigma \qquad \Gamma' \vdash y : C}{\Gamma \vdash y(\sigma, x{=}t) = y\sigma : C\sigma}$$

Proof. – EQ-ESUBST-WEAK

$$\begin{array}{lr} \Gamma \vdash (\sigma, x{=}t) = \sigma : \Gamma' & \text{EQ-SUBST-EXT-WEAK} \\ \Gamma \vdash \sigma = (\sigma, x{=}t) : \Gamma' & \text{EQ-SUBST-SYM} \\ \Gamma \vdash y\sigma = y(\sigma, x{=}t) : C\sigma & \text{ESUBST-F} \\ \Gamma \vdash y(\sigma, x{=}t) = y\sigma : C\sigma & \text{EQ-SYM} \end{array}$$

Lemma 27 (Admissible rules).

1. *(Function type formation:) If $\Gamma \vdash A$ and $\Gamma, x{:}A \vdash B$ then $\Gamma \vdash \mathsf{Fun}\, A\, \lambda x B$.*

Lemma 28 (Scope). *If $\Gamma \vdash J$ then $\mathsf{FV}(J) \subseteq \mathsf{dom}(\Gamma)$. If $\Gamma, x{:}A, \Gamma' \vdash J$ then $\mathsf{FV}(A) \subseteq \mathsf{dom}(\Gamma)$.*

Proof. Simultaneously by induction on the derivation.

Lemma 29 (Context well-formedness).

1. *If $\Gamma, \Gamma' \vdash J$ then $\Gamma \vdash$.*
2. *If $\Gamma, x{:}A, \Gamma' \vdash J$, then $\Gamma \vdash A$.*

Lemma 30 (Weakening). *If $\Gamma, \Gamma' \vdash J$ and both $\Gamma \vdash A$ and $x \notin \mathsf{dom}(\Gamma, \Gamma')$, then $\Gamma, x{:}A, \Gamma' \vdash J$.*

Proof. – *Case*

$$\text{EQ-SUBST-EXT-}\eta\ \frac{\Gamma, \Gamma' \vdash B}{\Gamma, \Gamma', y{:}B \vdash (\sigma_{\mathsf{id}}, y{=}y) = \sigma_{\mathsf{id}} : \Gamma, \Gamma', y{:}B}\ y \notin \mathsf{dom}(\Gamma, \Gamma')$$

By ind.hyp., $\Gamma, x : A, \Gamma' \vdash B$, hence, $\Gamma, x : A, \Gamma', y : B \vdash (\sigma_{\mathsf{id}}, y = y) = \sigma_{\mathsf{id}} : \Gamma, x : A, \Gamma', y : B$ by EQ-SUBST-EXT-η. With EQ-SUBST-WEAK we conclude $\Gamma, x{:}A, \Gamma', y{:}B \vdash (\sigma_{\mathsf{id}}, y{=}y) = \sigma_{\mathsf{id}} : \Gamma, \Gamma', y{:}B$.

Corollary 6 (Iterative weakening). *If $\Gamma \vdash J$ and $\Gamma, \Gamma' \vdash J'$ then $\Gamma, \Gamma' \vdash J$.*

Proof. By induction on Γ', using the two previous lemmata.

Lemma 31 (Context conversion). *Let $\Gamma \vdash B = A$ and $\Gamma \vdash B$. If $\mathcal{D} :: \Gamma, x{:}A, \Gamma' \vdash J$ then $\Gamma, x{:}B, \Gamma' \vdash J$.*

Proof. By induction on $\mathcal{D}$. Does not require any of the previous lemmata.

Lemma 32 (Inversion of Typing).

1. *If $\mathcal{D} :: \Gamma \vdash c : C$ then $\Sigma \vdash c : C'$ for some C' with $\Gamma \vdash C = C'$.*
2. *If $\mathcal{D} :: \Gamma \vdash x : C$ then $\Gamma \vdash C = \Gamma(x)$.*
3. *If $\mathcal{D} :: \Gamma \vdash \lambda x t : C$ then $\Gamma \vdash C = \mathsf{Fun}\, A\, \lambda x B$ and $\Gamma, x{:}A \vdash t : B$.*
4. *If $\mathcal{D} :: \Gamma \vdash r\, s : C$ then $\Gamma \vdash r : \mathsf{Fun}\, A\, \lambda x B$ with $\Gamma \vdash s : A$ and $\Gamma \vdash C = B(\sigma_{\mathsf{id}}, x{=}s)$.*

5. *If* $\mathcal{D} :: \Gamma \vdash \mathsf{Fun}\, A\, \lambda x B : C$ *then* $\Gamma \vdash C = \mathsf{Set}_i$ *for some* i *and* $\Gamma \vdash A : \mathsf{Set}_i$ *and* $\Gamma, x\!:\!A \vdash B : \mathsf{Set}_i$.

Proof. Each by induction on $\mathcal{D}$, treating rules CONV and SUBST until one gets to CONST, HYP, FUN-I, FUN-E, or FUN-F, resp. This lemma can be proven in isolation.

Lemma 33 (Derived substitution rules). *Let* $\Gamma \vdash s : A$.

1. *If* $\Gamma, x\!:\!A \vdash B$ *then* $\Gamma \vdash B(\sigma_{\mathsf{id}}, x\!=\!s)$.
2. *If* $\Gamma, x\!:\!A \vdash t : B$ *then* $\Gamma \vdash t(\sigma_{\mathsf{id}}, x\!=\!s) : B(\sigma_{\mathsf{id}}, x\!=\!s)$.

Lemma 34 (Derived functionality rules). *Let* $\Gamma \vdash s = s' : A$.

1. *If* $\Gamma, x\!:\!A \vdash B$ *then* $\Gamma \vdash B(\sigma_{\mathsf{id}}, x\!=\!s) = B(\sigma_{\mathsf{id}}, x\!=\!s')$.
2. *If* $\Gamma, x\!:\!A \vdash t : B$ *then* $\Gamma \vdash t(\sigma_{\mathsf{id}}, x\!=\!s) = t(\sigma_{\mathsf{id}}, x\!=\!s') : B(\sigma_{\mathsf{id}}, x\!=\!s)$.

Lemma 35 (Syntactic validity).

1. *Typing: If* $\mathcal{D} :: \Gamma \vdash t : A$ *then* $\Gamma \vdash A$.
2. *Typing of substitutions: If* $\mathcal{D} :: \Gamma \vdash \sigma : \Gamma'$ *then* $\Gamma' \vdash$.
3. *Equality: If* $\mathcal{D} :: \Gamma \vdash t = t' : A$ *then* $\Gamma \vdash A$ *and both* $\Gamma \vdash t : A$ *and* $\Gamma \vdash t' : A$.
4. *Equality of substitutions: If* $\mathcal{D} :: \Gamma \vdash \sigma = \sigma' : \Gamma'$ *then* $\Gamma' \vdash$ *and both* $\Gamma \vdash \sigma : \Gamma'$ *and* $\Gamma \vdash \sigma' : \Gamma'$.

Proof. Simultaneously by induction on $\mathcal{D}$. Inversion for function types (Lemma 32) is needed.

In case HYP we conclude with context well-formedness (Lemma 29). In case FUN-E, we have to show $\Gamma \vdash B(\sigma_{\mathsf{id}}, x = s)$. By induction hypothesis, $\Gamma \vdash \mathsf{Fun}\, A\, \lambda x B$, thus, $\Gamma, x : A \vdash B$ by inversion. By substitution (Lemma 33), $\Gamma \vdash B(\sigma_{\mathsf{id}}, x\!=\!s)$. In case EQ-FUN-F, we need context conversion (Lemma 31). In case EQ-FUN-E, the goal $\Gamma \vdash r'\, s' : B(\sigma_{\mathsf{id}}, x = s)$ is interesting. By induction hypothesis, $\Gamma \vdash r' : \mathsf{Fun}\, A\, \lambda x B$ and $\Gamma \vdash s' : A$, which entail $\Gamma \vdash r'\, s' : B(\sigma_{\mathsf{id}}, x = s')$. From the induction hypothesis $\Gamma \vdash \mathsf{Fun}\, A\, \lambda x B$ we get by inversion (Lemma 2) $\Gamma, x : A \vdash B$, hence, by functionality (Lemma 34), $\Gamma \vdash B(\sigma_{\mathsf{id}}, x\!=\!s) = B(\sigma_{\mathsf{id}}, x\!=\!s')$. Thus, by conversion, $\Gamma \vdash r'\, s' : B(\sigma_{\mathsf{id}}, x\!=\!s')$.

– *Case*

$$\textsc{eq-esubst-fun-i}\ \frac{\Gamma \vdash \sigma : \Gamma' \qquad \Gamma', x\!:\!A \vdash t : B}{\Gamma \vdash (\lambda x t)\sigma = \lambda x.\, t(\sigma, x\!=\!x) : (\mathsf{Fun}\, A\, \lambda x B)\sigma}\, x \notin \mathsf{dom}(\Gamma)$$

$\Gamma' \vdash A$ by context well-formedness
$\Gamma', x{:}A \vdash B$ by induction hypothesis
$\Gamma' \vdash \mathsf{Fun}\, A\, \lambda x B$ by Lemma 27.1
$\Gamma \vdash (\mathsf{Fun}\, A\, \lambda x B)\sigma$ by ESUBST-F

$\Gamma' \vdash \lambda x t : \mathsf{Fun}\, A\, \lambda x B$ by FUN-I
$\Gamma \vdash (\lambda x t)\sigma : (\mathsf{Fun}\, A\, \lambda x B)\sigma$ by ESUBST-F

$\Gamma \vdash A\sigma$ by ESUBST-F
$\Gamma, x{:}A\sigma \vdash \sigma : \Gamma'$ by weakening (Lemma 1.3)
$\Gamma, x{:}A\sigma \vdash x : A\sigma$ by HYP
$\Gamma, x{:}A\sigma \vdash (\sigma, x{=}x) : \Gamma', x{:}A$ by SUBST-EXT
$\Gamma, x{:}A\sigma \vdash t(\sigma, x{=}x) : B(\sigma, x{=}x)$ by ESUBST-F
$\Gamma \vdash \lambda x.\, t(\sigma, x{=}x) : \mathsf{Fun}\,(A\sigma)\,(\lambda x. B(\sigma, x{=}x))$ by FUN-I
$\Gamma \vdash (\mathsf{Fun}\, A\, \lambda x B)\sigma = \mathsf{Fun}\,(A\sigma)(\lambda x.\, B(\sigma, x{=}x))$ by EQ-ESUBST-FUN-F
$\Gamma \vdash \lambda x.\, t(\sigma, x{=}x) : (\mathsf{Fun}\, A\, \lambda x B)\sigma$ by EQ-SYM, CONV

– *Case*

$$\text{EQ-SUBST-EXT-WEAK}\ \frac{\Gamma \vdash \sigma : \Gamma' \qquad \Gamma' \vdash A \qquad \Gamma \vdash t : A\sigma}{\Gamma \vdash (\sigma, x{=}t) = \sigma : \Gamma'}\ x \notin \mathsf{dom}(\Gamma')$$

$\Gamma \vdash (\sigma, x{=}t) : \Gamma', x{:}A$ SUBST-EXT
$\Gamma \vdash (\sigma, x{=}t) : \Gamma'$ SUBST-WEAK

References

AAD07. Andreas Abel, Klaus Aehlig, and Peter Dybjer. Normalization by evaluation for Martin-Löf type theory with one universe. In Marcelo Fiore, editor, *Proc. of the 23rd Conf. on the Mathematical Foundations of Programming Semantics (MFPS XXIII)*, volume 173 of *Electr. Notes in Theor. Comp. Sci.*, pages 17–39. Elsevier, 2007.

AC07. Andreas Abel and Thierry Coquand. Untyped algorithmic equality for Martin-Löf's logical framework with surjective pairs. *Fundam. Inform.*, 77(4):345–395, 2007. TLCA'05 special issue.

AC08. Thorsten Altenkirch and James Chapman. Big step normalisation. Draft, available on the authors' homepages., 2008.

ACD07. Andreas Abel, Thierry Coquand, and Peter Dybjer. Normalization by evaluation for Martin-Löf Type Theory with typed equality judgements. In *Proc. of the 22nd IEEE Symp. on Logic in Computer Science (LICS 2007)*, pages 3–12. IEEE Computer Soc. Press, 2007.

Bar84. Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North Holland, Amsterdam, 1984.

BL84. Kim B. Bruce and Giuseppe Longo. On combinatory algebras and their expansions. *Theor. Comput. Sci.*, 31:31–40, 1984.

BS91. Ulrich Berger and Helmut Schwichtenberg. An inverse to the evaluation functional for typed λ-calculus. In *Proceedings of the 6th Annual IEEE Symposium on Logic in Computer Science, Amsterdam*, pages 203–211, July 1991.

CAM07. James Chapman, Thorsten Altenkirch, and Conor McBride. Epigram reloaded: a standalone typechecker for ETT. In Marko C. J. D. van Eekelen, editor, *Revised Selected Papers from the 6th Symp. on Trends in Functional Programming, TFP 2005*, volume 6 of *Trends in Functional Programming*, pages 79–94. Intellect, 2007.

Coq94. Catarina Coquand. From semantics to rules: A machine assisted analysis. In Egon Börger, Yuri Gurevich, and Karl Meinke, editors, *Proc. of the 7th Wksh. on Computer Science Logic, CSL '93*, volume 832 of *Lect. Notes in Comput. Sci.*, pages 91–105. Springer-Verlag, 1994.

Coq96. Thierry Coquand. An algorithm for type-checking dependent types. In *Mathematics of Program Construction. Selected Papers from the Third International Conference on the Mathematics of Program Construction (July 17–21, 1995, Kloster Irsee, Germany)*, volume 26 of *Science of Computer Programming*, pages 167–177. Elsevier Science, May 1996.

CPT05. Thierry Coquand, Randy Pollack, and Makoto Takeyama. A logical framework with dependently typed records. *Fundam. Inform.*, 65(1-2):113–134, 2005.

Dan99. Olivier Danvy. Type-directed partial evaluation. In John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors, *Partial Evaluation – Practice and Theory, DIKU 1998 International Summer School, Copenhagen, Denmark, June 29 - July 10, 1998*, volume 1706 of *Lect. Notes in Comput. Sci.*, pages 367–411. Springer-Verlag, 1999.

Dyb96. Peter Dybjer. Internal type theory. In Stefano Berardi and Mario Coppo, editors, *Types for Proofs and Programs (TYPES'95), Torino, Italy*, number 1158 in Lect. Notes in Comput. Sci., pages 120–134. Springer-Verlag, 1996.

Dyb00. Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *The Journal of Symbolic Logic*, 65(2):525–549, 2000.

GL02. Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In *Proc. of the 7th ACM SIGPLAN Int. Conf. on Functional Programming (ICFP '02)*, volume 37 of *SIGPLAN Notices*, pages 235–246. ACM Press, September 2002.

Gog94. Healfdene Goguen. *A Typed Operational Semantics for Type Theory*. PhD thesis, University of Edinburgh, August 1994. Available as LFCS Report ECS-LFCS-94-304.

HP05. Robert Harper and Frank Pfenning. On equivalence and canonical forms in the LF type theory. *ACM Transactions on Computational Logic*, 6(1):61–101, 2005.

INR07. INRIA. *The Coq Proof Assistant, Version 8.1*. INRIA, 2007. http://coq.inria.fr/.

Ler06. Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *Proc. of the 33rd ACM Symp. on Principles of Programming Languages, POPL 2006*, pages 42–54. ACM Press, 2006.

ML75. Per Martin-Löf. About models for intuitionistic type theories and the notion of definitional equality. In S. Kanger, editor, *Proceedings of the 3rd Scandinavian Logic Symposium*, pages 81–109, 1975.

ML92. Per Martin-Löf. Substitution calculus. Unpublished notes from a lecture in Göteborg, November 1992.

Nor07. Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-41296 Göteborg, Sweden, September 2007.

Pit06. A. M. Pitts. Alpha-structural recursion and induction. *Journal of the ACM*, 53:459–506, 2006.

Pol94. Robert Pollack. Closure under alpha-conversion. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs: International Workshop, Nijmegen, May 1993, Selected Papers*, number 806 in LNCS, pages 313–332. Springer-Verlag, 1994.

PS99. Frank Pfenning and Carsten Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, volume 1632 of *Lect. Notes in Art. Intell.*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag.

PT98. Benjamin C. Pierce and David N. Turner. Local type inference. In *POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Diego, California, 1998.

Shi05. Mark Shinwell. *The Fresh Approach: Functional Programming with Names and Binders*. PhD thesis, University of Cambridge, 2005.

SS08. Carsten Schürmann and Jeffrey Sarnat. Structural logical relations. In Frank Pfenning, editor, *Proc. of the 23nd IEEE Symp. on Logic in Computer Science (LICS 2008)*, 2008.