Logical Methods in Computer Science
Vol. 7 (2:4) 2011, pp. 1–57
www.lmcs-online.org

Submitted Nov. 15, 2009
Published May 4, 2011

A MODULAR TYPE-CHECKING ALGORITHM FOR TYPE THEORY WITH SINGLETON TYPES AND PROOF IRRELEVANCE

ANDREAS ABEL [a], THIERRY COQUAND [b], AND MIGUEL PAGANO [c]

[a] Ludwig-Maximilians-Universität München
e-mail address: andreas.abel@ifi.lmu.de

[b] Göteborg University
e-mail address: coquand@chalmers.se

[c] Universidad Nacional de Córdoba
e-mail address: pagano@famaf.unc.edu.ar

Abstract. We define a logical framework with singleton types and one universe of small types. We give the semantics using a PER model; it is used for constructing a normalisation-by-evaluation algorithm. We prove completeness and soundness of the algorithm; and get as a corollary the injectivity of type constructors. Then we give the definition of a correct and complete type-checking algorithm for terms in normal form. We extend the results to proof-irrelevant propositions.

1. Introduction and Related Work

One of the raisons d'être of proof-checkers like Agda [46], Coq [33], and Epigram [40] is to decide if a given term has some type (either checking for a given type or inferring one); i.e., if a term corresponds to a proof of a proposition [32]. Hence, the convenience of such a system is, in part, determined by the types for which the system can check membership. We extend the decidability of type-checking done in previous works [2, 3] for Martin-Löf type theories [38, 45] by considering singleton types and proof-irrelevant propositions.

We consider a type theory with a *universe*, which allows large eliminations, i.e., types defined by recursion on natural numbers. The universe of small types was introduced by Martin-Löf [37] for formalising category theory. Martin-Löf presents universes in two different styles [38]: *à la Russell* (the one considered here), and *à la Tarski*.

1998 ACM Subject Classification: F.4.1.

Key words and phrases: type theory, type-checking, normalisation-by-evaluation, singleton types, proof-irrelevance.

[a] Supported by INRIA as guest researcher in the PI.R2 team, PPS, Paris, France, from October 2009 to March 2010.

[c] Partially supported by CONICET, Argentina.

Singleton types were introduced by Aspinall [10] in the context of specification languages. An important use of singletons is as definitions by abbreviations (see [10, 21]); they were also used to model translucent sums in the formalisation of SML [34]. It is interesting to consider singleton types because beta-eta phase separation fails: one cannot do eta-expansion before beta-normalisation of types because the shape of the types at which to eta-expand is still unknown at this point; and one cannot postpone eta-expansion after beta-normalisation, because eta-expansion at singleton type can trigger new beta-reductions. Stone and Harper [54] decide type checking in a logical framework (LF) with singleton types and subtyping. Yet it is not clear whether their method extends to computation on the type level. As far as we know, our work is the first where singleton types are considered together with a universe.

De Bruijn proposed the concept of *irrelevance of proofs* [18], for reducing the burden in the formalisation of mathematics. As shown by Werner [56], the use of proof-irrelevance types together with sigma types is one way to get subset types à la PVS [51] in type-theories having the eta rule. This style of subset types was also explored by Sozeau [53, Sec. 3.3]; for another presentation of subset types in Martin-Löf type-theory see [50]. Berardi conjectured that (impredicative) type-theory with proof-irrelevance is equivalent to constructive mathematics [14].

Checking dependent types relies on checking types for equality. To this end, we compute η-long normal forms using *normalisation by evaluation* (NbE) [39]. Syntactic expressions are evaluated into a semantic domain and then *reified* back to expressions in normal form. To handle functional and open expressions, the semantic domain has to be equipped with variables; a major challenge in rigorous treatments of NbE has been the problem to generate fresh identifiers. Solutions include term families [16], liftable de Bruijn terms [8], or Kripke semantics [5]. In this work we present a novel formulation of NbE which avoids the problem completely: reification is split into an η-expansion phase ($\downarrow$) in the semantics, followed by a read back function (R) into the syntax which is indexed by the number of already used variables. This way, a standard PER model is sufficient, and technical difficulties are avoided.

Outline. In Section 2, we first present λ^{Sing}, Martin-Löf's logical framework with one universe and singleton types, as a generalized algebraic theory [19]. Secondly, we introduce λ^{Irr}, Martin-Löf type theory with natural numbers, sigma types, and proof-irrelevant propositions. In Section 3, we show some examples using singleton types and proof-irrelevant types. In Section 4, we present briefly NbE for untyped and simply typed lambda calculi; in particular we illustrate our novel approach to generate fresh identifiers. In Section 5, we define the semantics of the type theories by a PER model and prove the soundness of the inference rules. We use this model to introduce a normalization algorithm **nbe**, for which we prove completeness (if $t = t'$ is derivable, then $\mathbf{nbe}(t)$ and $\mathbf{nbe}(t')$ are identical). The soundness of the algorithm (i.e., $t = \mathbf{nbe}(t)$ is derivable) is proven by logical relations in Section 6. In Section 7, we define a bi-directional algorithm for checking the type of normal forms and inferring the type of neutral terms. More related work is discussed in Section 8.1. The Haskell programs corresponding to the NbE, and type-checking algorithms are shown in the appendices A and B, respectively.

2. The Calculus as a Generalised Algebraic Theory

In this section, we introduce the type theory. In order to show the modularity of our approach, we present it as two calculi λ^{Sing} and λ^{Irr}: the first one has dependent function spaces, singleton types, and a universe closed under function spaces and singletons. In the second calculus we leave out singleton types and we add natural numbers, sigma types, and proof-irrelevant propositions. It is not clear if singleton types can be combined with proof-irrelevant propositions without turning the system inconsistent.

We present the calculi using the formalism proposed by Cartmell for generalised algebraic theories (GAT) [19]. A GAT consists of *sort symbols* and *operator symbols*, each with a dependent typing, and *equations* between *sort expressions* and *terms* ("operator expressions"). Following Dybjer [24], we are using "informal syntax" where redundant arguments to operators are left implicit.

2.1. **Calculus $\lambda^{\mathsf{Sing}}$with singleton types.**

We use capital Greek letters (Γ, Δ) for variables ranging over contexts; capital letters from the beginning of the Latin alphabet (A, B) for variables ranging over types; small Greek letters (δ, ρ, σ) are used for variables denoting substitutions; and minuscule Latin characters (r, s, t, u, a, b) for variables on terms. Words in sans face denote constants (e.g., $\mathsf{Type}, \mathsf{q}$).

2.1.1. *Sorts.* The set of sort symbols is $\{\mathsf{Ctx}, \rightarrow, \mathsf{Type}, \mathsf{Term}\}$ and their formation rules, in the sense of Cartmell's GATs, are:

$$\frac{}{\mathsf{Ctx}\text{ is a type}}\ (\textsc{ctx-sort}) \qquad \frac{\Gamma, \Delta \in \mathsf{Ctx}}{\Gamma \rightarrow \Delta\text{ is a type}}\ (\textsc{subs-sort})$$

$$\frac{\Gamma \in \mathsf{Ctx}}{\mathsf{Type}(\Gamma)\text{ is a type}}\ (\textsc{type-sort}) \qquad \frac{\Gamma \in \mathsf{Ctx} \qquad A \in \mathsf{Type}(\Gamma)}{\mathsf{Term}(\Gamma, A)\text{ is a type}}\ (\textsc{term-sort})$$

In the following, whenever a rule has a hypothesis $A \in \mathsf{Type}(\Gamma)$, then $\Gamma \in \mathsf{Ctx}$ shall be a further, implicit hypothesis. Similarly, $\sigma \in \Gamma \rightarrow \Delta$ presupposes $\Gamma \in \mathsf{Ctx}$ and $\Delta \in \mathsf{Ctx}$, and $t \in \mathsf{Term}(\Gamma, A)$ presupposes $A \in \mathsf{Type}(\Gamma)$, which in turn presupposes $\Gamma \in \mathsf{Ctx}$. Note that judgements of the form $\Gamma \in \mathsf{Ctx}$, $A \in \mathsf{Type}(\Gamma)$, $t \in \mathsf{Term}(\Gamma, A)$, and $\sigma \in \Gamma \rightarrow \Delta$ correspond to the more conventional forms $\Gamma \vdash$, $\Gamma \vdash A$, $\Gamma \vdash t : A$, and $\Gamma \vdash \sigma : \Delta$, resp. After we have defined the judgements, we will use the latter, more readable versions.

2.1.2. *Operators.* The set of operators is quite large and instead of giving it at once, we define it as the union of the disjoint sets of operators for contexts, substitutions, types, and terms.

Contexts. There are the usual two operators for constructing contexts: $S_C = \{\diamond, \_.\_\}$.

$$\frac{}{\diamond \in \mathsf{Ctx}}\ (\textsc{empty-ctx}) \qquad \frac{\Gamma \in \mathsf{Ctx} \qquad A \in \mathsf{Type}(\Gamma)}{\Gamma.A \in \mathsf{Ctx}}\ (\textsc{ext-ctx})$$

Substitutions. We have five operators for substitutions, which are the usual operators for explicit substitutions [1]: $S_S = \{\langle\rangle, (\_,\_), \mathsf{id}_\_, \_\,\_, \mathsf{p}\}$. Semantically, substitutions $\sigma \in \Gamma \to \Delta$ are sequences of values, one for every variable declaration in Δ. The sequences are constructed from the empty sequence $\langle\rangle$ by sequence extension (σ, t). Substitutions form a category with identity id_Γ and composition $\sigma\,\delta$. Finally, we have the first projection p on sequences.

$$\frac{\Gamma \in \mathsf{Ctx}}{\langle\rangle \in \Gamma \to \diamond}\ (\textsc{empty-subs}) \qquad \frac{\sigma \in \Gamma \to \Delta \qquad t \in \mathsf{Term}(\Gamma, A\,\sigma)}{(\sigma, t) \in \Gamma \to \Delta.A}\ (\textsc{ext-subs})$$

$$\frac{\Gamma \in \mathsf{Ctx}}{\mathsf{id}_\Gamma \in \Gamma \to \Gamma}\ (\textsc{id-subs}) \qquad \frac{\delta \in \Gamma \to \Theta \qquad \sigma \in \Theta \to \Delta}{\sigma\,\delta \in \Gamma \to \Delta}\ (\textsc{comp-subs})$$

$$\frac{A \in \mathsf{Type}(\Gamma)}{\mathsf{p} \in \Gamma.A \to \Gamma}\ (\textsc{fst-subs})$$

Types. The set of operators for types is $S_T = \{\mathsf{U}, \mathsf{Fun}\,\_\,\_, \_\,\_, \{\_\}_\_\}$. U is a universe of small types à la Russell, which means its elements are directly usable as types ($\textsc{u-el}$) without coercion. Besides dependent function types $\mathsf{Fun}\,A\,B$ we have the singleton type $\{t\}_A$—a subtype of A containing t as single inhabitant. Types A are closed under substitution $A\,\sigma$.

$$\frac{\Gamma \in \mathsf{Ctx}}{\mathsf{U} \in \mathsf{Type}(\Gamma)}\ (\textsc{u-f}) \qquad \frac{A \in \mathsf{Term}(\Gamma, \mathsf{U})}{A \in \mathsf{Type}(\Gamma)}\ (\textsc{u-el}) \qquad \frac{A \in \mathsf{Type}(\Gamma) \qquad B \in \mathsf{Type}(\Gamma.A)}{\mathsf{Fun}\,A\,B \in \mathsf{Type}(\Gamma)}\ (\textsc{fun-f})$$

$$\frac{A \in \mathsf{Type}(\Gamma) \qquad t \in \mathsf{Term}(\Gamma, A)}{\{t\}_A \in \mathsf{Type}(\Gamma)}\ (\textsc{sing-f}) \qquad \frac{A \in \mathsf{Type}(\Delta) \qquad \sigma \in \Gamma \to \Delta}{A\,\sigma \in \mathsf{Type}(\Gamma)}\ (\textsc{subs-type})$$

Terms. The set of operators for terms is $S_E = \{\mathsf{Fun}\,\_\,\_, \lambda\_, \mathsf{app}\ \_\ \_, \_\,\_, \mathsf{q}, \{\_\}_\_\}$. It includes function space $\mathsf{Fun}\,A\,B$ and singleton $\{t\}_A$ as small-type constructors in U. Lambda terms with explicit substitutions are obtained via the constructions λt, $\mathsf{app}\ t\ u$, q, and $t\,\sigma$. Since we have used juxtaposition for composition and application of substitutions, we have the explicit app for term application. Note that q stands for the top (0th) variable, the nth variable is expressed as $\mathsf{q}\,\mathsf{p}^n$.

$$\frac{A \in \mathsf{Term}(\Gamma, \mathsf{U}) \qquad B \in \mathsf{Term}(\Gamma.A, \mathsf{U})}{\mathsf{Fun}\,A\,B \in \mathsf{Term}(\Gamma, \mathsf{U})}\ (\textsc{fun-u-i}) \qquad \frac{t \in \mathsf{Term}(\Gamma.A, B)}{\lambda t \in \mathsf{Term}(\Gamma, \mathsf{Fun}\,A\,B)}\ (\textsc{fun-i})$$

$$\frac{B \in \mathsf{Type}(\Gamma.A) \qquad t \in \mathsf{Term}(\Gamma, \mathsf{Fun}\,A\,B) \qquad u \in \mathsf{Term}(\Gamma, A)}{\mathsf{app}\ t\ u \in \mathsf{Term}(\Gamma, B\,(\mathsf{id}_\Gamma, u))}\ (\textsc{fun-el})$$

$$\frac{A \in \mathsf{Type}(\Gamma)}{\mathsf{q} \in \mathsf{Term}(\Gamma.A, A\,\mathsf{p})}\ (\textsc{hyp}) \qquad \frac{\sigma \in \Gamma \to \Delta \qquad t \in \mathsf{Term}(\Delta, A)}{t\,\sigma \in \mathsf{Term}(\Gamma, A\,\sigma)}\ (\textsc{subs-term})$$

$$\frac{A \in \mathsf{Term}(\Gamma, \mathsf{U}) \qquad t \in \mathsf{Term}(\Gamma, A)}{\{t\}_A \in \mathsf{Term}(\Gamma, \mathsf{U})}\ (\textsc{sing-u-i}) \qquad \frac{t \in \mathsf{Term}(\Gamma, A)}{t \in \mathsf{Term}(\Gamma, \{t\}_A)}\ (\textsc{sing-i})$$

$$\frac{a \in \mathsf{Term}(\Gamma, A) \qquad t \in \mathsf{Term}(\Gamma, \{a\}_A)}{t \in \mathsf{Term}(\Gamma, A)}\ (\textsc{sing-el})$$

2.1.3. *Axioms for Equational Theory.* In the following, we present the axioms of the equational theory of λ^{Sing}. Equality is considered as the congruence closure of these axioms. Congruence rules, also called derived rules, are generated mechanically for each symbol from its typing. For instance, rule (SUBS-TYPE) induces the derived rule

$$\frac{A = B \in \mathsf{Type}(\Gamma) \qquad \gamma = \delta \in \Delta \to \Gamma}{A\,\gamma = B\,\delta \in \mathsf{Type}(\Delta)}\ .$$

Another instance of a derived rule is conversion, it holds because equality between sorts, such as $\mathsf{Term}(\Gamma, A) = \mathsf{Term}(\Gamma, A')$:

$$\frac{t \in \mathsf{Term}(\Gamma, A) \qquad A = A' \in \mathsf{Type}(\Gamma)}{t \in \mathsf{Term}(\Gamma, A')}$$

In the following, we present equality axioms without the premises concerning typing, except in the cases where they cannot be inferred.

Substitutions. The first two equations witness extensionality for the identity substitution, the next three the composition laws for the category of substitutions. Then there is a law for the first projection p, and the last two laws show how to propagate a substitution δ into a tuple.

$$\begin{aligned} \mathsf{id}_\diamond &= \langle\rangle & \mathsf{id}_{\Gamma.A} &= (\mathsf{p}, \mathsf{q}) \\ \mathsf{id}\,\sigma &= \sigma & \sigma\,\mathsf{id} &= \sigma \\ (\sigma\,\delta)\,\gamma &= \sigma\,(\delta\,\gamma) & \mathsf{p}\,(\sigma, t) &= \sigma \\ \langle\rangle\,\delta &= \langle\rangle & (\sigma, t)\,\delta &= (\sigma\,\delta, t\,\delta) \end{aligned}$$

Axioms for β and η, propagation and resolution of substitutions. An explicit substitution (id_Γ, r) is created by contracting a β-redex (first law). It is then propagated into the various term constructions until it can be resolved (last two laws).

$$\begin{aligned} \mathsf{app}\ (\lambda t)\ r &= t\,(\mathsf{id}_\Gamma, r) & \lambda(\mathsf{app}\ (t\,\mathsf{p})\ \mathsf{q}) &= t \\ \mathsf{U}\,\sigma &= \mathsf{U} & (\{t\}_A)\,\sigma &= \{t\,\sigma\}_{A\,\sigma} \\ (\mathsf{Fun}\,A\,B)\,\sigma &= \mathsf{Fun}\,(A\,\sigma)\,(B\,(\sigma\,\mathsf{p}, \mathsf{q})) & (\lambda t)\,\sigma &= \lambda(t\,(\sigma\,\mathsf{p}, \mathsf{q})) \\ (\mathsf{app}\ r\ s)\,\sigma &= \mathsf{app}\ (r\,\sigma)\ (s\,\sigma) & (t\,\delta)\,\sigma &= t\,(\delta\,\sigma) \\ \mathsf{q}\,(\sigma, t) &= t & t\,\mathsf{id} &= t \end{aligned}$$

Singleton types. All inhabitants of a singleton type are equal (SING-EQ-I). We mention the important derived rule (SING-EQ-EL) here explicitly.

$$\frac{t, t' \in \mathsf{Term}(\Gamma, \{a\}_A)}{t = t' \in \mathsf{Term}(\Gamma, \{a\}_A)}\ (\textsc{sing-eq-i}) \qquad \frac{t = t' \in \mathsf{Term}(\Gamma, \{a\}_A)}{t = t' \in \mathsf{Term}(\Gamma, A)}\ (\textsc{sing-eq-el})$$

There is a choice how to express the last two rules; they could be replaced with

$$\frac{t \in \mathsf{Term}(\Gamma, \{a\}_A)}{t = a \in \mathsf{Term}(\Gamma, \{a\}_A)}\ (\textsc{sing-eq-i'}) \quad \frac{t \in \mathsf{Term}(\Gamma, \{a\}_A)}{t = a \in \mathsf{Term}(\Gamma, A)}\ (\textsc{sing-eq-el'})$$

The rule (SING-EQ-EL) is essential; in fact, since we have eta-expansion for singletons, we would like to derive

$$\Gamma.\{\lambda t\}_{\mathsf{Fun}\,A\,B} \vdash \mathsf{app}\ \mathsf{q}\ a = t\,(\mathsf{id}, a) : B\,(\mathsf{id}, a)$$

from $\Gamma.\{\lambda t\}_{\mathsf{Fun}\,A\,B} \vdash \mathsf{q} = \lambda t : \{\lambda t\}_{\mathsf{Fun}\,A\,B}$, and $\Gamma \vdash a : A$. Which would be impossible if (SING-EQ-EL) were not a rule.

Conventions. We denote with $|\Gamma|$ the length of the context Γ; and $\Gamma!i$ is the projection of the i-th component of Γ, for $0 \leqslant i < |\Gamma|$; i.e. if $\Gamma = A_{n-1} \dots A_0$ and $0 \leqslant i < n$, then $\Gamma!i = A_i$. We say $\Delta \leqslant^i \Gamma$ if $\Delta \vdash \mathsf{p}^i : \Gamma$; where p^i is the i-fold composition of p with itself. [1]

We denote with *Terms* the set of words freely generated using symbols in $S_S \cup S_T \cup S_E$. We write $t \equiv_T t'$ for denoting syntactically equality of t and t' in $T \subseteq$ *Terms*. We call A the *tag* of $\{a\}_A$.

Remark 2.1. Note that if $\Delta \vdash \mathsf{p}^i : \Gamma$, and $\Gamma \vdash \mathsf{p}^j : \Theta$, then $\Delta \vdash \mathsf{p}^{i+j} : \Theta$.

Definition 2.2 (de Bruijn index)**.** The ith de Bruijn index v_i is defined as

$$\mathsf{v}_i = \begin{cases} \mathsf{q} & \text{if } i \leqslant 0 \\ \mathsf{q}\mathsf{p}^i & \text{if } i > 0. \end{cases}$$

For convenience, we identify negative indices with the 0th index.

The following grammar describes the set *Nf* of β-normal forms. As auxiliary notion, it uses the set *Ne* of neutral normal forms, *i.e.*, normal forms with a variable in head position, which blocks reduction. A bit sloppily, we refer to elements of *Ne* as "neutral terms"; in general, the attribute *neutral* shall mean *variable in head position* (this is stricter than Girard's concept of neutral [28]).

Definition 2.3 (Neutral terms, and normal forms)**.**

$$\begin{aligned} \mathit{Ne} \ni k &::= \mathsf{v}_i \mid \mathsf{app}\ k\ v \\ \mathit{Nf} \ni v, V, W &::= \mathsf{U} \mid \mathsf{Fun}\,V\,W \mid \{v\}_V \mid \lambda v \mid k \end{aligned}$$

An advantage of introducing the calculus as a GAT is that we can derive several syntactical results from the meta-theory of GATs; for instance, some of the following inversion results, which are needed in the proof of completeness of the type-checking algorithm.

Remark 2.4 (Weakening of judgements)**.** Let $\Delta \leqslant^i \Gamma$, $\Gamma \vdash A = A'$, and $\Gamma \vdash t = t' : A$; then $\Delta \vdash A\,\mathsf{p}^i = A'\,\mathsf{p}^i$, and $\Delta \vdash t\,\mathsf{p}^i = t'\,\mathsf{p}^i : A\,\mathsf{p}^i$.

[1]The direction $\Delta \leqslant^i \Gamma$ (as opposed to $\Delta \geqslant^i \Gamma$) has been chosen to be compatible with subtyping $A \leqslant B$. Weakening (Remark 2.4) is a special case of subsumption which states that $\Delta \leqslant \Gamma \vdash t : A \leqslant B$ implies $\Delta \vdash t : B$.

Remark 2.5 (Syntactic validity)**.**
(1) If $\Gamma \vdash t : A$, then $\Gamma \vdash A$.
(2) If $\Gamma \vdash t = t' : A$, then both $\Gamma \vdash t : A$, and $\Gamma \vdash t' : A$.
(3) If $\Gamma \vdash A = A'$, then both $\Gamma \vdash A$, and $\Gamma \vdash A'$.

Lemma 2.6 (Inversion of types)**.**
(1) *If* $\Gamma \vdash \mathsf{Fun}\, A\, B$*, then* $\Gamma \vdash A$*, and* $\Gamma.A \vdash B$*.*
(2) *If* $\Gamma \vdash \{a\}_A$*, then* $\Gamma \vdash A$*, and* $\Gamma \vdash a : A$*.*
(3) *If* $\Gamma \vdash k$*, then* $\Gamma \vdash k : \mathsf{U}$*.*

The following lemma can be proved directly by induction on derivations by checking the possibles rules used in the last step.

Lemma 2.7 (Inversion of typing)**.**
(1) *If* $\Gamma \vdash \mathsf{Fun}\, A'\, B' : A$*, then* $\Gamma \vdash A = \mathsf{U}$*,* $\Gamma \vdash A' : \mathsf{U}$*, and also* $\Gamma.A' \vdash B' : \mathsf{U}$*;*
(2) *if* $\Gamma \vdash \{b\}_B : A$*, then* $\Gamma \vdash A = \mathsf{U}$*,* $\Gamma \vdash B : \mathsf{U}$*, and also* $\Gamma \vdash b : B$*;*
(3) *if* $\Gamma \vdash \lambda t : A$*, then either*
  (a) $\Gamma \vdash A = \mathsf{Fun}\, A'\, B$ *with* $\Gamma.A' \vdash t : B$*; or*
  (b) $\Gamma \vdash A = \{a\}_{A'}$ *with* $\Gamma \vdash \lambda t = a : A'$*.*
(4) *if* $\Gamma \vdash \mathsf{app}\; t\; r : A$*, then* $\Gamma \vdash t : \mathsf{Fun}\, A'\, B'$*,* $\Gamma \vdash r : A'$*, and* $\Gamma \vdash A = B'\,(\mathsf{id}, r')$*.*

Proof. (1) The last rule used is one of (FUN-U-I), (CONV), (SING-I), or (SING-E). In the first case the premises of the rule are what is to be proved; in all other cases we have a premise with the form $\Gamma \vdash \mathsf{Fun}\, A'\, B' : B$, hence we can apply the i.h. (2-4) Analogously. □

Remark 2.8 (Inversion of substitution)**.** Any substitution $\Delta \vdash \sigma : \Gamma.A$ is equal to some substitution $\Delta \vdash (\sigma', t) : \Gamma.A$. It is enough to note $\mathsf{id}_{\Gamma.A} = (\mathsf{p}, \mathsf{q})$, hence we have the equalities $\sigma = \mathsf{id}\, \sigma = (\mathsf{p}, \mathsf{q})\, \sigma = (\mathsf{p}\, \sigma, \mathsf{q}\, \sigma)$.

2.2. λ^{Irr}: A type theory with proof-irrelevance.

In this section we keep the basic rules of the previous calculus (those that do not refer to singleton types), and introduce types for natural numbers, enumeration sets, sigma types, and proof-irrelevant types. The main difference with other presentations [45, 38], on the syntactic level, is that the eliminator operator (for each type) has as an argument the type of the result. The presence of the resulting type in the eliminator is needed in order to define the normalisation function; it is also necessary for the type-inference algorithm.

Sigma types. Both U and Type are closed under (strong) sigma-type formation; (a, b) introduces a dependent pair and $\mathsf{fst}\; t$ and $\mathsf{snd}\; t$ eliminate it.

$$\frac{A \in \mathsf{Term}(\Gamma, \mathsf{U}) \qquad B \in \mathsf{Term}(\Gamma.A, \mathsf{U})}{\Sigma\, A\, B \in \mathsf{Term}(\Gamma, \mathsf{U})}\ (\textsc{sum-u-i}) \qquad \frac{A \in \mathsf{Type}(\Gamma) \qquad B \in \mathsf{Type}(\Gamma.A)}{\Sigma\, A\, B \in \mathsf{Type}(\Gamma)}\ (\textsc{sum-f})$$

$$\frac{B \in \mathsf{Type}(\Gamma.A) \qquad a \in \mathsf{Term}(\Gamma, A) \qquad b \in \mathsf{Term}(\Gamma, B\,(\mathsf{id}, a))}{(a, b) \in \mathsf{Term}(\Gamma, \Sigma\, A\, B)}\ (\textsc{sum-in})$$

$$\frac{t \in \mathsf{Term}(\Gamma, \Sigma\, A\, B)}{\mathsf{fst}\; t \in \mathsf{Term}(\Gamma, A)}\ (\textsc{sum-el1}) \qquad \frac{t \in \mathsf{Term}(\Gamma, \Sigma\, A\, B)}{\mathsf{snd}\; t \in \mathsf{Term}(\Gamma, B\,(\mathsf{id}, \mathsf{fst}\; t))}\ (\textsc{sum-el2})$$

The β- and η-laws for pairs are given by the first three equations to follow. The remaining equations propagate substitutions into the new term constructors.

$$\begin{array}{ccc} \mathsf{fst}\ (a,b) = a & \mathsf{snd}\ (a,b) = b & (\mathsf{fst}\ t, \mathsf{snd}\ t) = t \\ (\mathsf{fst}\ t)\,\sigma = \mathsf{fst}\ (t\,\sigma) & (\mathsf{snd}\ t)\,\sigma = \mathsf{snd}\ (t\,\sigma) & (a,b)\,\sigma = (a\,\sigma, b\,\sigma) \end{array}$$
$$(\Sigma\,A\,B)\,\sigma = \Sigma\,(A\,\sigma)\,(B\,(\sigma\,\mathsf{p},\mathsf{q}))$$

Propagation laws can be obtained mechanically: to propagate σ into $\mathsf{c}\,\vec{t}$, just compose it with each t_i that is not a binder (e.g., A in $\Sigma\,A\,B$), and compose its lifted version $(\sigma\,\mathsf{p},\mathsf{q})$ with each t_j that is a binder (e.g., B in $\Sigma\,A\,B$). Binders are those formed in an extended context (here, $B \in \mathsf{Type}(\Gamma.A)$). In the following, we will skip the propagation laws.

Natural numbers. We add an inductive type Nat with constructors zero and suc and primitive recursion natrec.

$$\frac{\Gamma \in \mathsf{Ctx}}{\mathsf{Nat} \in \mathsf{Term}(\Gamma,\mathsf{U})}\ (\textsc{nat-u-i}) \quad \frac{\Gamma \in \mathsf{Ctx}}{\mathsf{zero} \in \mathsf{Term}(\Gamma,\mathsf{Nat})}\ (\textsc{nat-z-i}) \quad \frac{t \in \mathsf{Term}(\Gamma,\mathsf{Nat})}{\mathsf{suc}\ t \in \mathsf{Term}(\Gamma,\mathsf{Nat})}\ (\textsc{nat-s-i})$$

$$\frac{\begin{array}{c} B \in \mathsf{Type}(\Gamma.\mathsf{Nat}) \\ t \in \mathsf{Term}(\Gamma,\mathsf{Nat}) \qquad z \in \mathsf{Term}(\Gamma, B\,(\mathsf{id},\mathsf{zero})) \qquad s \in \mathsf{Term}(\Gamma,\mathsf{Rec}(B)) \end{array}}{\mathsf{natrec}\ B\ z\ s\ t \in \mathsf{Term}(\Gamma, B\,(\mathsf{id},t))}\ (\textsc{nat-el})$$

Here, we used $\mathsf{Rec}(B)$ as an abbreviation for $\mathsf{Fun}\,\mathsf{Nat}\,(\mathsf{Fun}\,B\ (B\,(\mathsf{p},\mathsf{suc}\ \mathsf{q})\,\mathsf{p}))$ which in conventional notation reads $\Pi x : \mathsf{Nat}.\,B \to B[\mathsf{suc}\ x/x]$. Since B is a big type, it can mention the universe U, thus, we can define small types by recursion via natrec. This so called *large elimination* excludes normalization proofs which use induction on type expressions [23, 22]. We add the usual computation laws for primitive recursion.

$$\begin{aligned} \mathsf{natrec}\ B\ z\ s\ \mathsf{zero} &= z \\ \mathsf{natrec}\ B\ z\ s\ (\mathsf{suc}\ t) &= \mathsf{app}\ (\mathsf{app}\ s\ t)\ (\mathsf{natrec}\ B\ z\ s\ t) \end{aligned}$$

Enumeration sets. The type N_n has the n canonical inhabitants c^n_0, …, c^n_{n-1}, which can be eliminated by the dependent case distinction $\mathsf{case}^n\ B\ t_0 \cdots t_{n-1}\ t$ with n branches.

$$\frac{\Gamma \in \mathsf{Ctx}}{\mathsf{N}_n \in \mathsf{Term}(\Gamma,\mathsf{U})}\ (\textsc{n}_n\textsc{-u-i}) \qquad \frac{\Gamma \in \mathsf{Ctx} \qquad i < n}{\mathsf{c}^n_i \in \mathsf{Term}(\Gamma,\mathsf{N}_n)}\ (\textsc{n}_n\textsc{-i})$$

$$\frac{\begin{array}{c} B \in \mathsf{Type}(\Gamma.\mathsf{N}_n) \qquad t \in \mathsf{Term}(\Gamma,\mathsf{N}_n) \\ t_0 \in \mathsf{Term}(\Gamma, B\,(\mathsf{id},\mathsf{c}^n_0)) \ \cdots\ t_{n-1} \in \mathsf{Term}(\Gamma, B\,(\mathsf{id},\mathsf{c}^n_{n-1})) \end{array}}{\mathsf{case}^n\ B\ t_0 \cdots t_{n-1}\ t \in \mathsf{Term}(\Gamma, B\,(\mathsf{id},t))}\ (\textsc{n}_n\textsc{-e})$$

We add the usual computational law for case distinction, and weak extensionality, which for booleans (N_2) reads "$\mathsf{if}\ t\ \mathsf{then}\ \mathsf{true}\ \mathsf{else}\ \mathsf{false} = t$" in sugared syntax.

$$\begin{aligned} \mathsf{case}^n\ B\ t_0 \cdots t_{n-1}\ \mathsf{c}^n_i &= t_i \\ \mathsf{case}^n\ \mathsf{N}_n\ \mathsf{c}^n_0 \cdots \mathsf{c}^n_{n-1}\ t &= t \end{aligned}$$

For N_0 and N_1 we can formulate strong η-laws: all their inhabitants are considered equal, since there is at most one. To realize this, we introduce a new term $\star$ in N_0 *if it already has an inhabitant* t; we consider $\star$ as normal form of t. Note that this seemingly paradoxical canonical form $\star \in \mathsf{N}_0$ does not threaten consistency, since it cannot exist in the empty context $\Gamma = \diamond$; otherwise there would have already been a term $t \in \mathsf{Term}(\diamond, \mathsf{N}_0)$.

$$\frac{t \in \mathsf{Term}(\Gamma, \mathsf{N}_0)}{\star \in \mathsf{Term}(\Gamma, \mathsf{N}_0)}\ (\textsc{n}_0\textsc{-tm}) \qquad \frac{t, t' \in \mathsf{Term}(\Gamma, \mathsf{N}_0)}{t = t' \in \mathsf{Term}(\Gamma, \mathsf{N}_0)}\ (\textsc{n}_0\textsc{-eq}) \qquad \frac{t, t' \in \mathsf{Term}(\Gamma, \mathsf{N}_1)}{t = t' \in \mathsf{Term}(\Gamma, \mathsf{N}_1)}\ (\textsc{n}_1\textsc{-eq})$$

On the one hand, rule ($\textsc{n}_0\textsc{-tm}$) destroys decidability of type checking: to check whether $\star \in \mathsf{Term}(\Gamma, \mathsf{N}_0)$ we would have to decide the consistency of Γ which is certainly impossible in a theory with natural numbers. On the other hand, it allows us to decide equality by computing canonical forms. We solve this dilemma by forbidding $\star$ in the user syntax which is input for the type-checker; $\star$ is only used internally in the NbE algorithm and in the canonical forms it produces. Formally this is reflected by having two calculi: one with the rule ($\textsc{n}_0\textsc{-tm}$) and one without it. For distinguishing the calculi, we decorate the turnstile ($\vdash^\star$) in judgements of the former and leave ($\vdash$) for the calculus without ($\textsc{n}_0\textsc{-tm}$). We also use the different turnstiles for referring to each calculus. In Sect. 2.3 we prove that ($\vdash^\star$) is a conservative extension of ($\vdash$).

Strong extensionality for booleans and larger enumeration sets is hard to implement [9, 12] and beyond the scope of this work.

In the sequel we use $\vec{t}$ for denoting the n terms $t_0 \cdots t_{n-1}$ in $\mathsf{case}^n\ B\ t_0 \cdots t_{n-1}\ r$. We will omit the superscript n in c_i, and in $\mathsf{case}\ B\ \vec{t}\ r$.

Proof irrelevance. Our treatment of proof-irrelevance is based on Awodey and Bauer [11] and Maillard [36]. The constructor Prf turns a type A into the *proposition* $\mathsf{Prf}\,A$ in the sense that only the fact matters *whether* A is inhabited, not by *what*. An inhabited proposition is regarded as *true*, an uninhabited as *false*. The proposition $\mathsf{Prf}\,A$ still has all inhabitants of A, but now they are considered equal. If A is not empty, we introduce a trivial proof $\star$ in $\mathsf{Prf}\,A$ which we regard as the normal form of any $t \in \mathsf{Term}(\Gamma, \mathsf{Prf}\,A)$.

$$\frac{A \in \mathsf{Term}(\Gamma, \mathsf{U})}{\mathsf{Prf}\,A \in \mathsf{Term}(\Gamma, \mathsf{U})}\ (\textsc{prf-f}) \qquad \frac{A \in \mathsf{Type}(\Gamma)}{\mathsf{Prf}\,A \in \mathsf{Type}(\Gamma)}\ (\textsc{prf-f})$$

$$\frac{a \in \mathsf{Term}(\Gamma, A)}{[a] \in \mathsf{Term}(\Gamma, \mathsf{Prf}\,A)}\ (\textsc{prf-i}) \qquad \frac{a \in \mathsf{Term}(\Gamma, A)}{\star \in \mathsf{Term}(\Gamma, \mathsf{Prf}\,A)}\ (\textsc{prf-tm})$$

$$\frac{A \in \mathsf{Type}(\Gamma) \qquad t, t' \in \mathsf{Term}(\Gamma, \mathsf{Prf}\,A)}{t = t' \in \mathsf{Term}(\Gamma, \mathsf{Prf}\,A)}\ (\textsc{prf-eq})$$

Note that ($\textsc{prf-tm}$) is analogous to ($\textsc{n}_0\textsc{-tm}$) and the same remarks apply; in particular, ($\textsc{prf-tm}$) is also a rule in ($\vdash^\star$) but not in ($\vdash$).

We use Awodey and Bauer's [11] elimination rule for proofs.

$$\frac{\Gamma \vdash t : \mathsf{Prf}\,A \qquad \Gamma \vdash B \qquad \Gamma, x\!:\!A \vdash b : B \qquad \Gamma, x\!:\!A, y\!:\!A \vdash b = b[y/x] : B}{\Gamma \vdash b\ \mathsf{where}\ [x] \leftarrow t : B}$$

The content $x : A$ of a proof $t : \mathsf{Prf}\,A$ can be used in b via the elimination $b\ \mathsf{where}\ [x] = t$ if b does not actually depend on it, which is expressed via the hypothesis that b should be equal to $b[y/x]$ for an arbitrary y. This elimination principle is stronger than "proofs can only be used inside of proofs" which is witnessed by the rule:

$$\frac{\Gamma \vdash t : \mathsf{Prf}\,A \qquad \Gamma \vdash B \qquad \Gamma, x\!:\!A \vdash b : \mathsf{Prf}\,B}{\Gamma \vdash b\ \mathsf{where}\ [x] \leftarrow t : \mathsf{Prf}\,B}$$

Note that this weaker elimination rule in the style of a *bind* operation for monads is an instance of the Awodey-Bauer rule, since the equation $\Gamma, x{:}A, y{:}A \vdash b = b[y/x] : \mathsf{Prf}\,B$ holds trivially due to proof irrelevance. An example which is typable with the Awoday-Bauer rule but not the monadic rule is the term magic given in the next section.

The Awodey-Bauer where fulfills β-, η-, and associativity laws analogous to the ones of a monad.

$$\begin{aligned} b \;\mathsf{where}\; [x] \leftarrow [a] &= b[a/x] \\ b[[x]/y] \;\mathsf{where}\; [x] \leftarrow t &= b[t/y] \\ a \;\mathsf{where}\; [x] \leftarrow (b \;\mathsf{where}\; [y] \leftarrow c) &= (a \;\mathsf{where}\; [x] \leftarrow b) \;\mathsf{where}\; [y] \leftarrow c \qquad \text{if } y \notin \mathsf{FV}(a) \end{aligned}$$

After this more readable presentation in named syntax, we add the eliminator and its equations to our GAT in de Bruijn style:

$$\frac{\begin{array}{c} t \in \mathsf{Term}(\Gamma, \mathsf{Prf}\,A) \\ B \in \mathsf{Type}(\Gamma) \qquad b \in \mathsf{Term}(\Gamma.A, B\,\mathsf{p}) \qquad b\,\mathsf{p} = b\,(\mathsf{p}\,\mathsf{p}, \mathsf{q}) \in \mathsf{Term}(\Gamma.A.A\,\mathsf{p}, B\,\mathsf{p}\,\mathsf{p}) \end{array}}{b\,\mathsf{where}^B\, t \in \mathsf{Term}(\Gamma, B)} \;(\text{PRF-EL})$$

$$\begin{aligned} b\,\mathsf{where}^B\,[a] &= b\,(\mathsf{id}, a) && (\text{PRF-}\beta) \\ b\,(\mathsf{p}, [\mathsf{q}])\,\mathsf{where}^B\, t &= b\,(\mathsf{id}, t) && (\text{PRF-}\eta) \\ a\,\mathsf{where}^A\,(b\,\mathsf{where}^B\, c) &= (a\,(\mathsf{p}\,\mathsf{p}, \mathsf{q})\,\mathsf{where}^{A\,\mathsf{p}}\, b)\,\mathsf{where}^B\, c && (\text{PRF-ASSOC}) \end{aligned}$$

After exposition of the formation, introduction, elimination, and equality rules for the types of λ^{Irr}, we continue with basic properties of derivations. From now, we use the more conventional notation for judgements.

Definition 2.9 (Neutral terms and normal forms)**.**

$$\begin{aligned} \mathit{Ne} \ni k &::= \ldots \mid \mathsf{fst}\ k \mid \mathsf{snd}\ k \mid \mathsf{natrec}\ V\ v\ v'\ k \mid \mathsf{case}^n\ V\ v_0 \cdots v_{n-1}\ k \mid v\,\mathsf{where}^V\ k \mid \star \\ \mathit{Nf} \ni v, V &::= \ldots \mid \Sigma\, V\, W \mid \mathsf{Nat} \mid \mathsf{N}_n \mid \mathsf{Prf}\ V \mid (v, v') \mid \mathsf{zero} \mid \mathsf{suc}\ v \mid \mathsf{c}^n_i \mid [v] \end{aligned}$$

Lemma 2.10 (Inversion of types)**.**

(1) *If* $\Gamma \vdash \Sigma\, A\, B$*, then* $\Gamma \vdash A$*, and* $\Gamma.A \vdash B$*.*
(2) *If* $\Gamma \vdash \mathsf{Prf}\,A$*, then* $\Gamma \vdash A$*.*

Lemma 2.11 (Inversion of typing)**.**

(1) *If* $\Gamma \vdash \Sigma\, A'\, B : A$*, then* $\Gamma \vdash A = \mathsf{U}$*, and* $\Gamma \vdash A' : \mathsf{U}$*, and* $\Gamma.A' \vdash B : \mathsf{U}$*.*
(2) *If* $\Gamma \vdash \mathsf{Nat} : A$*, then* $\Gamma \vdash A = \mathsf{U}$*.*
(3) *If* $\Gamma \vdash \mathsf{N}_n : A$*, then* $\Gamma \vdash A = \mathsf{U}$*.*
(4) *If* $\Gamma \vdash (t, b) : A$ *, then* $\Gamma \vdash A = \Sigma\, A'\, B$*, and* $\Gamma \vdash t : A'$*, and* $\Gamma \vdash b : B\,(\mathsf{id}, t)$*.*
(5) *If* $\Gamma \vdash \mathsf{fst}\, t : A$ *, then* $\Gamma \vdash A = A'$*, and* $\Gamma \vdash t : \Sigma\, A'\, B$*, for some* A'*, and* B*.*
(6) *If* $\Gamma \vdash \mathsf{snd}\, t : B$*, then* $\Gamma \vdash B = B'\,(\mathsf{id}, \mathsf{fst}\, t)$*, and* $\Gamma \vdash t : \Sigma\, A\, B'$*, for some* A*, and* B'*.*
(7) *If* $\Gamma \vdash \mathsf{zero} : A$*, then* $\Gamma \vdash A = \mathsf{Nat}$*.*
(8) *If* $\Gamma \vdash \mathsf{suc}\, t : A$*, then* $\Gamma \vdash t : \mathsf{Nat}$*, and* $\Gamma \vdash A = \mathsf{Nat}$*.*
(9) *If* $\Gamma \vdash \mathsf{natrec}\ B\ z\ s\ t : A$ *, then* $\Gamma.\mathsf{Nat} \vdash B$*,* $\Gamma \vdash z : B\,(\mathsf{id}, \mathsf{zero})$*,* $\Gamma \vdash s : \mathsf{Rec}(B)$*,* $\Gamma \vdash t : \mathsf{Nat}$*, and* $\Gamma \vdash A = B\,(\mathsf{id}, t)$*.*
(10) *if* $\Gamma \vdash \mathsf{c}^n_i : A$*, then* $\Gamma \vdash A = \mathsf{N}_n$*;*

(11) *If* $\Gamma \vdash \mathsf{case}\ B\ \vec{t}\ t' : A$, *then* $\Gamma.\mathsf{N}_n \vdash B$, $\Gamma \vdash t_i : B\,(\mathsf{id}, \mathsf{c}_i)$, $\Gamma \vdash t' : \mathsf{N}_n$, *and* $\Gamma \vdash A = B\,(\mathsf{id}, t)$.
(12) *If* $\Gamma \vdash [t] : A$, *then* $\Gamma \vdash A = \mathsf{Prf}\,A'$ *and* $\Gamma \vdash t' : A'$.
(13) *If* $\Gamma \vdash b\,\mathsf{where}^B\,t : A$, *then* $\Gamma \vdash A = B$, $\Gamma \vdash t : \mathsf{Prf}\,A'$ *for some* A', $\Gamma.A' \vdash b : B\,\mathsf{p}$, *and* $\Gamma.A'.A'\,\mathsf{p} \vdash b\,\mathsf{p} = b\,(\mathsf{pp}, \mathsf{q}) : B\,\mathsf{p}$.

2.3. **Conservativity of $\star$.**

In this section we prove that $(\vdash^\star)$ is a *conservative* extension of $(\vdash)$; i.e., any derivation in $(\vdash^\star)$ has a counterpart derivation in $(\vdash)$ and the components of the conclusions of those derivations are judgmentally equal in $(\vdash^\star)$.

Definition 2.12. A term is called *pure* if it does not contain any occurrence of $\star$. Let ν be a syntactical entity, if μ is obtained from ν by replacing all occurrences of $\star$ by pure terms, then μ is called a *lifting* of ν.

We will distinguish those liftings that are judgmentally equal to the lifted entity, these liftings are called *good liftings*.

Definition 2.13 (Good lifting)**.**
(1) A context $\Gamma' \vdash$ is a good lifting of $\Gamma \vdash^\star$ if Γ' is a lifting of Γ, such that $\vdash^\star \Gamma = \Gamma'$.
(2) A substitution $\Gamma' \vdash \sigma' : \Delta'$ is a good lifting of $\Gamma \vdash^\star \sigma : \Delta$ if $\Gamma' \vdash$ and $\Delta' \vdash$ are good liftings of $\Gamma \vdash^\star$ and $\Delta \vdash^\star$, resp., and σ' is a lifting of σ, such that $\Gamma \vdash^\star \sigma = \sigma' : \Delta$.
(3) A type $\Gamma \vdash A'$ is a good lifting of $\Gamma \vdash^\star A$ if $\Gamma' \vdash$ is a good lifting of $\Gamma \vdash^\star$ and A' is a lifting of A, such that $\Gamma \vdash^\star A = A'$.
(4) A term $\Gamma \vdash t' : A'$ is a good lifting of $\Gamma \vdash^\star t : A$ if $\Gamma' \vdash A'$ is a good lifting of $\Gamma \vdash^\star A$ and t' is a lifting of t, such that $\Gamma \vdash^\star t = t' : A$.

Now we can prove that there is a good lifting for each syntactic entity; for proving this, we need the stronger condition that any pair of good liftings for some entity are judgmentally equal.

Theorem 2.14.
(1) *Let* $\Gamma \vdash^\star$*; then there is a good lifting* $\Gamma' \vdash$ *of* $\Gamma \vdash^\star$*; moreover if* $\Gamma'' \vdash$ *is also a good lifting of* $\Gamma \vdash^\star$ *then* $\vdash \Gamma' = \Gamma''$.
(2) *Let* $\Gamma \vdash^\star \sigma : \Delta$*; then there is a good lifting* $\Gamma' \vdash \sigma' : \Delta'$ *of* $\Gamma \vdash^\star \sigma : \Delta$*; moreover if* $\Gamma'' \vdash \sigma'' : \Delta''$ *is also a good lifting of* $\Gamma \vdash^\star \sigma : \Delta$ *then* $\vdash \Gamma' = \Gamma''$, $\vdash \Delta' = \Delta''$, *and* $\Gamma' \vdash \sigma' = \sigma'' : \Delta'$.
(3) *Let* $\Gamma \vdash^\star A$*; then there is a good lifting* $\Gamma' \vdash A'$ *of* $\Gamma \vdash^\star A$ *; moreover if* $\Gamma'' \vdash A''$ *is also a good lifting of* $\Gamma \vdash^\star A$ *then* $\vdash \Gamma' = \Gamma''$ *and* $\Gamma' \vdash A' = A''$.
(4) *Let* $\Gamma \vdash^\star t : A$*; then there is a good lifting* $\Gamma' \vdash t' : A'$ *of* $\Gamma \vdash^\star t : A$ *; moreover if* $\Gamma'' \vdash t'' : A''$ *is also a good lifting of* $\Gamma \vdash^\star t : A$ *then* $\vdash \Gamma' = \Gamma''$, $\Gamma' \vdash A' = A''$, *and* $\Gamma' \vdash t' = t'' : A'$.

Proof. By induction on derivations, in each rule we use i.h., and build up the corresponding entity to the good lifting for each part of the judgement; then, given any other good lifting of the whole judgement, we do inversion on the definition of good lifting, and get the equalities for each part; we finish using congruence for showing that both good lifting are judgmental equal.

We show the case for (PRF-TM). First we prove the existence of a good lifting.

$\Gamma \vdash^{\star} \star : \mathsf{Prf}\, A$	hypothesis	(*)
$\Gamma \vdash^{\star} t : A$	by inversion on (*)	(†)
$\Gamma' \vdash t' : A'$	by ind. hyp. is a good lifting of (†)	
$\Gamma' \vdash [t'] : \mathsf{Prf}\, A'$	by (PRF-I), is a good lifting of(*)	

Now we prove the second half of the theorem.

$\Gamma'' \vdash s : B$	hypothesis, be other good lifting of (†)	(**)
$\Gamma'' \vdash B$	by inversion, good lifting of $\Gamma \vdash^{\star} \mathsf{Prf}\, A$	
$\Gamma' \vdash B'' = \mathsf{Prf}\, A'$	by ind. hyp.	
$\Gamma' \vdash [t'] = s : \mathsf{Prf}\, A'$	by (PRF-EQ) and (CONV).	

□

Corollary 2.15. *The calculus* $(\vdash^{\star})$ *is a conservative extension of* $(\vdash)$. □

Combining singleton types and proof-irrelevant propositions. For illustrating the difficulties one can find when extending λ^{Irr} with singleton types, consider a slightly different calculus where we drop the type annotation of the eliminator for proof-irrelevance terms; i.e. we would have $b\,\mathsf{where}\ t$ instead of $b\,\mathsf{where}^B\, t$. In the resulting system one can derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \mathsf{c}^2_0 : \{\mathsf{c}^2_0\}_{\mathsf{N}_2}}}{\vdash \star : \mathsf{Prf}\,\{\mathsf{c}^2_0\}_{\mathsf{N}_2}}}{\vdash x\ \mathsf{where}\ [x] \leftarrow \star : \{\mathsf{c}^2_0\}_{\mathsf{N}_2}}}{\vdash x\ \mathsf{where}\ [x] \leftarrow \star = \mathsf{c}^2_0 : \{\mathsf{c}^2_0\}_{\mathsf{N}_2}}}{\vdash x\ \mathsf{where}\ [x] \leftarrow \star = \mathsf{c}^2_0 : \mathsf{N}_2} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \mathsf{c}^2_1 : \{\mathsf{c}^2_1\}_{\mathsf{N}_2}}}{\vdash \star : \mathsf{Prf}\,\{\mathsf{c}^2_1\}_{\mathsf{N}_2}}}{\vdash x\ \mathsf{where}\ [x] \leftarrow \star : \{\mathsf{c}^2_1\}_{\mathsf{N}_2}}}{\vdash x\ \mathsf{where}\ [x] \leftarrow \star = \mathsf{c}^2_1 : \{\mathsf{c}^2_1\}_{\mathsf{N}_2}}}{\vdash x\ \mathsf{where}\ [x] \leftarrow \star = \mathsf{c}^2_1 : \mathsf{N}_2}}{\vdash \mathsf{c}^2_0 = \mathsf{c}^2_1 : \mathsf{N}_2}$$

This derivation shows that mixing the rule (SING-EQ-EL) with erasure of proof-terms leads to inconsistencies. It is yet unclear how to combine singleton types and erasure of proof-terms; we leave this topic for a future work. On the other hand, there are no problems in extending $(\vdash)$ with singletons types; in fact, we can construct (see Rem. 5.36) a model where $[\![\mathsf{c}^2_0]\!] \neq [\![\mathsf{c}^2_1]\!]$, which assures $\not\vdash \mathsf{c}^2_0 = \mathsf{c}^2_1 : \mathsf{N}_2$.

3. Examples

3.1. **Safe vector projection in λ^{Irr}.** We give a short demonstration how to use proof irrelevance in λ^{Irr}: we define vectors and a type safe projection function. While de Bruijn style is good for implementation and reasoning, it is virtually unreadable for humans, so we allow ourselves named λ-terms here which can be mechanically converted into actual terms of λ^{Irr}. For instance, we write $\mathsf{Fun}\, A\,(x.B)$ instead of the de Buijn style $\mathsf{Fun}\, A\, B$. For further convenience, let $(x : A) \to B = \mathsf{Fun}\, A\,(x.B)$ and $(x : A) \times B = \Sigma\, A\,(x.B)$. The non-dependent versions are written $A \to B = (\_ {:} A) \to B$ and $A \times B = (\_ {:} A) \times B$. The type $\mathsf{Vec}\ A\ n$ of vectors of length n over element type A can be defined recursively as follows.

$$\begin{array}{lcl} \mathsf{Vec} & : & (A{:}\mathsf{U}) \to (n{:}\mathsf{Nat}) \to \mathsf{U} \\ \mathsf{Vec} & = & \lambda A \lambda n.\ \mathsf{natrec}\ \mathsf{U}\ \mathsf{N}_1\ (\lambda n' \lambda V.\ A \times V)\ n \end{array}$$

A more suggestive notation for definitions by recursion is *pattern matching*; in this the definition of Vec reads as follows:

$$\begin{array}{lcl} \multicolumn{3}{l}{\mathsf{Vec} : (A{:}\mathsf{U}) \to (n{:}\mathsf{Nat}) \to \mathsf{U}} \\ \mathsf{Vec}\ A\ \mathsf{zero} & = & \mathsf{N}_1 \\ \mathsf{Vec}\ A\ (\mathsf{suc}\ n') & = & A \times \mathsf{Vec}\ A\ n' \end{array}$$

In the following, we use pattern matching as syntactic sugar for natrec. Our language already has booleans, so let us define comparison of natural numbers.

$$\begin{array}{lllcl@{\qquad}lcl} \multicolumn{5}{l}{\mathsf{leq} : (m{:}\mathsf{Nat}) \to (n{:}\mathsf{Nat}) \to \mathsf{Bool}} & \mathsf{Bool} & = & \mathsf{N}_2 \\ \mathsf{leq} & \mathsf{zero} & n & = & \mathsf{true} & \mathsf{true} & = & \mathsf{c}^2_0 \\ \mathsf{leq} & (\mathsf{suc}\ m) & \mathsf{zero} & = & \mathsf{false} & \mathsf{false} & = & \mathsf{c}^2_1 \\ \mathsf{leq} & (\mathsf{suc}\ m) & (\mathsf{suc}\ n) & = & \mathsf{leq}\ m\ n \end{array}$$

By reflecting booleans into U we can obtain witnesses of propositions.

$$\begin{array}{lcl} \multicolumn{3}{l}{\mathsf{True} : (b{:}\mathsf{Bool}) \to \mathsf{U}} \\ \mathsf{True\ true} & = & \mathsf{N}_1 \\ \mathsf{True\ false} & = & \mathsf{N}_0 \end{array}$$

$\mathsf{True}\ (\mathsf{leq}\ m\ n)$ is inhabited if $m \leqslant n$, because then $\mathsf{True}\ (\mathsf{leq}\ m\ n)$ simplifies to N_1 with trivial inhabitant c^1_0. If not $m \leqslant n$ then $\mathsf{True}\ (\mathsf{leq}\ m\ n)$ simplifies to the empty type N_0. A proposition Lt for "less than" is obtained as:

$$\begin{array}{lcl} \multicolumn{3}{l}{\mathsf{Lt} : (m{:}\mathsf{Nat}) \to (n{:}\mathsf{Nat}) \to \mathsf{U}} \\ \mathsf{Lt}\, m\, n & = & \mathsf{True}\ (\mathsf{leq}\ (\mathsf{suc}\ m)\ n) \end{array}$$

We are now ready to define a safe projection operation for vectors.

$$\begin{array}{llllcl} \multicolumn{6}{l}{\mathsf{lookup} : (A{:}\mathsf{U}) \to (n{:}\mathsf{Nat}) \to (m{:}\mathsf{Nat}) \to (p{:}\mathsf{Prf}\,(\mathsf{Lt}\, m\, n)) \to (v{:}\mathsf{Vec}\ A\ n) \to A} \\ \mathsf{lookup}\ A & \mathsf{zero} & m & p\ v & = & \mathsf{magic}\, A\, p \\ \mathsf{lookup}\ A & (\mathsf{suc}\ n) & \mathsf{zero} & p\ v & = & \mathsf{fst}\ v \\ \mathsf{lookup}\ A & (\mathsf{suc}\ n) & (\mathsf{suc}\ m) & p\ v & = & \mathsf{lookup}\ A\ n\ m\ p\ (\mathsf{snd}\, v) \end{array}$$

Since $\mathsf{Lt}\,(\mathsf{suc}\ m)\,(\mathsf{suc}\ n) = \mathsf{Lt}\, m\, n$ we can simply pass p to the recursive call in the last equation. In the first line we have to magically conjure an element of A from a proof $p : \mathsf{Prf}\ (\mathsf{Lt}\, m\, \mathsf{zero}) = \mathsf{Prf}\ (\mathsf{True}\ (\mathsf{leq}\ (\mathsf{suc}\ m)\ \mathsf{zero})) = \mathsf{Prf}\ (\mathsf{True\ false}) = \mathsf{Prf}\ \mathsf{N}_0$.

$$\begin{array}{l} \mathsf{magic} : (A{:}\mathsf{U}) \to (p{:}\mathsf{Prf}\,\mathsf{N}_0) \to A \\ \mathsf{magic}\, A\, p = \mathsf{case}^0\, A\, q\ \mathsf{where}\ [q] \leftarrow p \end{array}$$

This is well typed since all inhabitants of N_0 are equal, thus, the Awodey-Bauer rule (PRF-EL) is applicable.

The benefit of proof irrelevance is that now for any $p, q : \mathsf{Lt}\, m\, n$, $\mathsf{lookup}\ A\ n\ m\ p\ v = \mathsf{lookup}\ A\ n\ m\ q\ v : A$; for a more detailed discussion consult Werner [56].

3.2. **Isomorphisms in λ^{Irr}.** Any already irrelevant type is isomorphic to its Prf version, i.e., for $A \in \{\mathsf{N}_0, \mathsf{N}_1, \mathsf{Prf}\, B, \{t\}_B\}$ we have coercions

$$\begin{array}{lcll} \phi & = & \lambda x.\, [x] & : \ A \to \mathsf{Prf}\, A \\ \psi & = & \lambda x.\, y \ \mathsf{where}\ [y] \leftarrow x & : \ \mathsf{Prf}\, A \to A \end{array}$$

with $\psi \circ \phi = \lambda x.\, x$ by (PRF-β) and $\phi \circ \psi = \lambda x.\, x$ by proof irrelevance. How do these coercions extend to higher types? For arbitrary A, B we have.

$$\begin{array}{lcl} \phi^{\Sigma} & : & (x\!:\!\mathsf{Prf}\, A) \times \mathsf{Prf}\, B \to \mathsf{Prf}\, ((x\!:\!A) \times B) \\ \phi^{\Sigma} & = & \lambda p.\, [(a,b)] \ \mathsf{where}\ [a] \leftarrow \mathsf{fst}\, p \ \mathsf{where}\ [b] \leftarrow \mathsf{snd}\, p \\ \psi^{\Sigma} & : & \mathsf{Prf}\, ((x\!:\!A) \times B) \to (x\!:\!\mathsf{Prf}\, A) \times \mathsf{Prf}\, B \\ \psi^{\Sigma} & = & \lambda q.\, ([\mathsf{fst}\, p],\, [\mathsf{snd}\, p]) \ \mathsf{where}\ [p] \leftarrow q \\ \phi^{\Pi} & : & ((x\!:\!A) \to \mathsf{Prf}\, B) \to \mathsf{Prf}\, ((x\!:\!A) \to B) \\ \phi^{\Pi} & = & \lambda f.\, [\lambda x.\, ? \ \mathsf{where}\ [y] \leftarrow f\, x] \\ \psi^{\Pi} & : & \mathsf{Prf}\, ((x\!:\!A) \to B) \to (x\!:\!A) \to \mathsf{Prf}\, B \\ \psi^{\Pi} & = & \lambda f \lambda x.\, [g\, x] \ \mathsf{where}\ [g] \leftarrow f \end{array}$$

We would like to put y for the ? in ϕ^{Π}, but this is not well typed, since we do not have $y = z : B$ for arbitrary z. It seems that ϕ^{Π} is not definable in λ^{Irr}, as it is not definable in computational lambda-calculus [44] for an arbitrary monad Prf. Awodey and Bauer also have only $\phi^{\Pi} : ((x\!:\!A) \to \mathsf{Prf}\, B) \to \mathsf{Prf}\, ((x\!:\!A) \to \mathsf{Prf}\, B)$, which is trivial.

For arbitrary $p : (x\!:\!\mathsf{Prf}\, A) \times \mathsf{Prf}\, B$ we have

$$\begin{array}{rll} (\psi^{\Sigma} \circ \phi^{\Sigma})(p) & =_{\beta} & ([a],\, [b]) \ \mathsf{where}\ [a] \leftarrow \mathsf{fst}\, p \ \mathsf{where}\ [b] \leftarrow \mathsf{snd}\, p \\ & =_{\eta} & (\mathsf{fst}\, p,\, \mathsf{snd}\, p) =_{\eta} p. \end{array}$$

In the opposite direction, $\phi^{\Sigma} \circ \psi^{\Sigma} = \lambda q.\, q$ by proof irrelevance. Thus, ϕ and ψ establish an isomorphism, which means that Prf distributes over Σ.

3.3. **On subtyping in λ^{Sing}.** Subtyping can be defined in several ways, for instance, A is a subtype of A' in Γ, written $\Gamma \vdash A <: A'$, iff $\Gamma, x\!:\!A \vdash x\!:\!A'$. Most presentations of singleton types include subtyping [10, 22, 54], so it is natural to ask whether the usual rules hold in our calculus. Using the principle $u = t : A$ iff $u : \{t\}_A$, it is easy to see that we have Aspinall's two axioms [10]:

$$\begin{array}{lcl} x : \{t\}_A & \vdash & x : A \\ x : \{t\}_A & \vdash & x : \{t\}_{\{t\}_A} \quad \text{since } x = t : \{t\}_A \end{array}$$

Also, singleton formation is compatible with subtyping, if $\Gamma \vdash A <: B$ then $\Gamma \vdash \{t\}_A <: \{t\}_B$. Contravariant subtyping, however, only holds up to η-equality. If we relax the definition of subtyping $\Gamma \vdash A <: A'$ to $\Gamma, x : A \vdash \eta(x) : A'$ where $\eta(x)$ denotes any η-expansion of x, then we get contravariant subtyping

$$\frac{\Gamma, x\!:\!A' \vdash \eta_A(x) : A \qquad \Gamma, x\!:\!A', y\!:\!B[\eta_A(x)/x] \vdash \eta_B(y) : B'}{\Gamma, f\!:\!(x\!:\!A) \to B \vdash \lambda x.\, \eta_B(f(\eta_A(x))) : (x\!:\!A') \to B'}.$$

Furthermore, we have following two axioms, which hold definitionally in Stone and Harper's system [54]

$$\begin{array}{lcll} f : \{t\}_{(x:A)\to B} & \vdash & \lambda x.\, f\, x : (x\!:\!A) \to \{t\, x\}_B & \text{since } f = t : (x\!:\!A) \to B \\ f : (x\!:\!A) \to \{t\, x\}_B & \vdash & \lambda x.\, f\, x : \{t\}_{(x:A)\to B} & \text{since } f\, x = t\, x : B \end{array}$$

The first axiom is one of Courant's subtyping rules [22].

4. From Untyped to Typed Normalisation by Evaluation

In this section, we give a short introduction into normalisation-by-evaluation for typed lambda calculi, with a special emphasis on our novel method for generation of fresh identifiers during reification.

4.1. **Fresh Name Generation in NbE.**

The basic idea of NbE is to *evaluate* a term of type A into a suitable semantics $[\![A]\!]$ from which its *normal form* can be extracted by *reification*. In case of the simply-typed lambda-calculus, this is possible if we choose for base types $[\![o]\!]$ the set of terms of type o and for function types $[\![A \to B]\!]$ a suitable subset of the function space $[\![A]\!] \to [\![B]\!]$. During reification of a function $f \in [\![A \to B]\!]$ to a term $\lambda x.t$, the identifier x has to be chosen *fresh* to avoid capture of names in the body of the function f. However, since f is a semantic object, it is a non-trivial problem to compute a name which is fresh for f. Garillot and Werner [27] solve it by first letting x be a dummy identifier, computing the free variables in the reified function body t, and then reify f again with a name x which is fresh for t. This is, of course, horribly inefficient, and there are other solutions. In the original publication on NbE by Berger and Schwichtenberg [16], base types $[\![o]\!] = \widehat{\Lambda}$ are interpreted by *term families.* These are functions g from the natural numbers into a *de Bruijn level* representation of terms such that all instances $g(n)$ are α-equivalent but in $g(n)$ the bound variables are levels starting with n. In this setting, the reification of a function $f \in [\![A \to B]\!]$ is not a term but a term family, mapping f to the term family $n \mapsto \lambda x_n.\phi(f(\widehat{x_n}))(n+1)$, where ϕ denotes the reification function and $\widehat{x_n}$ is the variable x_n seen as an element in $[\![A]\!]$. Note that every λ in $\phi(f(\widehat{x_n}))(n+1)$, the body of the reified abstraction, will bind a variable from the set $\{x_{n+1}, x_{n+2}, \ldots\}$.

When considering NbE for the *untyped* lambda calculus, the type semantics collapses to a single domain $\mathsf{D} \cong \widehat{\Lambda} + [\mathsf{D} \to \mathsf{D}]$ which contains terms and functions,[2] as observed by Filinski and Rhode [26]. Aehlig and Joachimski [8] replace term families by functions h from natural numbers to a *de Bruijn index* representation of terms, where $h(n)$ shifts all *free* indices by n.

In this paper, instead of having term families $\widehat{\Lambda}$ in the semantics, we have a notion of *neutral ("term-like") value* built up from free variables x_i and application of the free variables to sequences of values $\vec{d}$. The free variables are *de Bruijn levels* in spirit, thus, no shifting is needed, just like in the *locally nameless* approach [49]. The second author has given a semantics with neutrals before [20], calling the free variables *generic values.* Also, this approach has been used by the first two authors together with Dybjer [5] for NbE without a reflection operation, and independently by Löh, McBride, and Swierstra [35]. In

[2] Let us notice here the *tagging* introduced by the disjoint sum operator $+$. Indeed, in the absence of a type structure, *tagless normalisation* seems impossible.

this article, we put the technique to a novel use by defining typed reification *and reflection* for this semantics.

4.2. **Untyped NbE.** Let $Var = \{x_0, x_1, \dots\}$ be a denumerable set of variables. We consider a set D and a notion of function space $[\mathsf{D} \to \mathsf{D}]$ with an embedding constructor $\mathsf{Lam} : [[\mathsf{D} \to \mathsf{D}] \to \mathsf{D}]$ and two further constructors $\mathsf{Var} : Var \to \mathsf{D}$ and $\mathsf{App} : [\mathsf{D} \times \mathsf{D} \to \mathsf{D}]$ for neutral values. An application function $\_ \cdot \_ : [\mathsf{D} \times \mathsf{D} \to \mathsf{D}]$ is given by

$$\begin{array}{lcll} (\mathsf{Lam}\, f) \cdot d & = & f(d) & \\ e \quad\quad\; \cdot d & = & \mathsf{App}(e, d) & \text{if } e \text{ not a } \mathsf{Lam}. \end{array}$$

Such a D can be realised by solving the recursive domain equation $\mathsf{D} \cong [\mathsf{D} \to \mathsf{D}] \oplus Var_\bot \oplus (\mathsf{D} \times \mathsf{D})$ or, for the practically minded, by defining a Haskell data type

```
data D where
  Lam : (D -> D) -> D
  Var : Var -> D
  App : D -> D -> D
```

and programming $\_ \cdot \_$ by pattern matching. Our definition of D is a bit "too big" since it does not restrict App to the construction of neutral values $\mathsf{App}(\dots \mathsf{App}(\mathsf{Var}\, x_i, d_1) \dots, d_n)$ but we have also $\mathsf{App}(\mathsf{Lam}\, f, d)$. However, we can ignore these unwanted elements since our NbE algorithm never produces any.

Remark 4.1. The relationship between the denotational model D and the Haskell data type `D` is not without subtleties. Domain theoretic functions such as application $\_ \cdot \_$ correspond to Haskell programs if our denotational semantics is computationally adequate for Haskell's operational semantics [48]. Filinski and Rhode [26] formally relate a NbE function on a reflexive domain D to a NbE program written in an ML-like, call-by-value language, by exploiting computational adequacy. We do not formally prove this connection for Haskell in this article, this is deferred to future work.

Untyped NbE is now given by a standard evaluator $[\![t]\!]\rho \in \mathsf{D}$ of terms t in environments ρ and a readback function $\mathsf{R}_j\, d$ from values d at de Bruijn level j to terms [31]. For the sake of readability, we use names instead of de Bruijn indices in the syntax of untyped terms.

$$\begin{array}{rcl@{\qquad\qquad}lcl} [\![x]\!]\rho & = & \rho(x) & \mathsf{R}_j\,(\mathsf{Var}\, x_i) & = & x_i \\ [\![r\, s]\!]\rho & = & [\![r]\!]\rho \cdot [\![s]\!]\rho & \mathsf{R}_j\,(\mathsf{App}(r, s)) & = & (\mathsf{R}_j\, r)\,(\mathsf{R}_j\, s) \\ [\![\lambda x.t]\!]\rho & = & \mathsf{Lam}\,(d \mapsto [\![t]\!]\rho[d/x]) & \mathsf{R}_j\,(\mathsf{Lam}\, f) & = & \lambda x_j.\, \mathsf{R}_{j+1}\,(f(\mathsf{Var}\, x_j)) \end{array}$$

To normalise a closed term t, compute $\mathsf{R}_0\, [\![t]\!]$. To normalise an open term t with free variables $y_0, \dots y_{n-1}$ compute $\mathsf{R}_n\, [\![t]\!]\rho$ with environment $\rho(y_i) = \mathsf{Var}\, x_i$.

To prepare for applying our method to λ^{Sing} and λ^{Irr}, let us switch to de Bruijn representation. Environments become tuples and variables de Bruijn indices v_i.

$$\begin{array}{lcl@{\qquad\qquad}lcl} [\![\mathsf{q}]\!](\rho, d) & = & d & \mathsf{R}_j\,(\mathsf{Var}\, x_i) & = & \mathsf{v}_{j-(i+1)} \\ [\![t\, \mathsf{p}]\!](\rho, d) & = & [\![t]\!]\rho & & & \\ [\![r\, s]\!]\rho & = & [\![r]\!]\rho \cdot [\![s]\!]\rho & \mathsf{R}_j\,(\mathsf{App}(r, s)) & = & (\mathsf{R}_j\, r)\,(\mathsf{R}_j\, s) \\ [\![\lambda t]\!]\rho & = & \mathsf{Lam}\,(d \mapsto [\![t]\!](\rho, d)) & \mathsf{R}_j\,(\mathsf{Lam}\, f) & = & \lambda\,(\mathsf{R}_{j+1}\,(f(\mathsf{Var}\, x_j))) \end{array}$$

To read back a de Bruijn level $\mathsf{Var}\, x_i$ as a de Bruijn index, we have to take the current length j of the variable context into account. While de Bruijn levels are absolute references, they are numbered $x_0, x_1, \dots, x_{j-1}$ in a context of length j, de Bruijn indices are relative to the length of the context, they are enumerated from right to left: $\mathsf{v}_{j-1}, \ldots, \mathsf{v}_1, \mathsf{v}_0$. The formula $j - (i+1)$ (assuming $i < j$) converts level i into the corresponding index.

4.3. **Typed NbE.** While untyped NbE returns a β-normal form (if it exists), typed normalisation by evaluation yields a $\beta\eta$-normal form, usually the η-long form. To obtain the η-long form, we have to modify our reification procedure. One method is to make read-back type directed [5], which corresponds to postponing η-expansion after β-normalisation. However, this strategy is not sufficient in the case of λ^{Sing}, because η-expansions at singleton types can trigger new β-reductions. The other method is to divide η-expansion into *reflection* and *"reification"*, the first expanding variables to enable new reductions, and the second expanding the result of β-normalisation to obtain an η-long form.[3]

The novel approach of this article is to do reflection $\uparrow$ and "reification" $\downarrow$, hence, η-expansion, completely at the level of the semantics D. Since our value domain D allows us to construct functions via Lam, the process of η-expansion is independent of any fresh name considerations.

$$\begin{array}{llll@{\qquad}llll}
\uparrow_o & e & = & e & \downarrow_o & d & = & d \\
\uparrow_{A\to B} & e & = & \mathsf{Lam}\,(d \mapsto \uparrow_B \mathsf{App}(e, \downarrow_A d)) & \downarrow_{A\to B} & d & = & \mathsf{Lam}\,(e \mapsto \downarrow_B (d \cdot (\uparrow_A e)))
\end{array}$$

To compute the long normal form of a closed term t of type A, run $\mathsf{R}_0\,(\downarrow_A [\![t]\!])$. For an open term $y_0 : A_0, \ldots, y_{n-1} : A_{n-1} \vdash t : A$, execute $\mathsf{R}_n\,(\downarrow_A [\![t]\!]\rho)$, where $\rho(y_i) = \uparrow_{A_i} (\mathsf{Var}\, x_i)$.

5. Semantics

In this section we define a domain D for denoting types, terms, and substitutions. Then we introduce a partial function R_j for reifying elements of the domain into the calculus; this function takes an extra argument $j \in \mathbb{N}$ indicating the next free variable. We continue by defining PERs over the domain; these PERs denote the axioms for types, terms, and substitutions. We need PERs for the evaluation function is defined over syntactical entities and not for typing judgements. We also introduce PERs $\mathcal{N}\!f$ and $\mathcal{N}\!e$ whose elements are invariably, in every context, reified as normal forms and neutral terms respectively. Using these PERs we define a family (indexed by denotations of types) of functions for "normalising" in the domain. We conclude this section proving completeness for this family of normalisation functions; here completeness means that two terms in the theory are read back as the same normal form. In this section we define a PER model of the calculus presented in the previous section. The model is used to define a normalisation function later.

[3]In tagless normalisers [16], reflection is necessary to inject variables $x : A$ of non-base types A into the semantics $[\![A]\!]$. However, for languages beyond pure type systems it is hard to obtain tagless normalisation. Classic is the problem of disjoint sum types [9]: to display a free variable of type $A + B$ as either a left or a right injection, we need control structures [12]. Alternatively, one can replace data types by their Church encodings. None of these approaches fit our purposes, thus, we are currently not aiming at tagless normalisation.

5.1. **PER semantics.** In this subsection we introduce the abstract notion of PER models for our theory. This subsection does not introduce any novelty (except for some notational issues). We refer the reader to [42] for a short report on the historical developments of PER models.

Definition 5.1 (Partial Equivalence Relations)**.** A partial equivalence relation (PER) over a set $\mathcal{A}$ is a binary relation over $\mathcal{A}$ which is symmetric and transitive.

If $\mathcal{R}$ is a PER over $\mathcal{A}$, and $(a, a') \in \mathcal{R}$ then it is clear that $(a, a) \in \mathcal{R}$. We define $\mathit{dom}(\mathcal{R}) = \{a \in \mathcal{A} \mid (a, a) \in \mathcal{R}\}$; clearly, $\mathcal{R}$ is an equivalence relation over $\mathit{dom}(\mathcal{R})$. If $(a, a') \in \mathcal{R}$, sometimes we will write $a = a' \in \mathcal{R}$, and $a \in \mathcal{R}$ if $a \in \mathit{dom}(\mathcal{R})$. We denote with $\mathrm{PER}(\mathcal{A})$ the set of all PERs over $\mathcal{A}$. Given two PERs $\mathcal{R}$ and $\mathcal{R}'$ over $\mathcal{A}$, we say $\mathcal{R}$ is included in $\mathcal{R}'$ if $(a, a') \in \mathcal{R}$ implies $(a, a') \in \mathcal{R}'$; we denote this inclusion with $\mathcal{R} \subseteq \mathcal{R}'$.

If $\mathcal{R} \in \mathrm{PER}(\mathcal{A})$ and $\mathcal{F} \colon \mathit{dom}(\mathcal{R}) \to \mathrm{PER}(\mathcal{A})$, we say that $\mathcal{F}$ is *a family of PERs indexed by* $\mathcal{R}$ iff for all $a = a' \in \mathcal{R}$, $\mathcal{F}\ a = \mathcal{F}\ a'$. If $\mathcal{F}$ is a family indexed by $\mathcal{R}$, we write $\mathcal{F} \colon \mathcal{R} \to \mathrm{PER}(\mathcal{A})$.

Definition 5.2 (Applicative structure)**.** An applicative structure is given by a pair $\mathcal{A} = (\mathcal{A}, \cdot)$, where $\mathcal{A}$ is a set and $\cdot$ is a binary operation on $\mathcal{A}$.

The following definitions are standard (*e.g.* [10, 21]) in definitions of PER models for dependent types. The first one is even standard for non-dependent types (*cf.* [43]) and "F-bounded polymorphism" ([17]); its definition clearly shows that equality is interpreted extensionally for dependent function spaces. The second one is the PER corresponding to the interpretation of singleton types; it has as its domain all the elements related to the distinguished element of the singleton, and it relates everything in its domain.

Definition 5.3. Let $\mathcal{A}$ be an applicative structure, $\mathcal{X} \in \mathrm{PER}(\mathcal{A})$, and $\mathcal{F} \in \mathcal{X} \to \mathrm{PER}(\mathcal{A})$.
(1) $\prod \mathcal{X}\, \mathcal{F} = \{(f, f') \mid f \cdot a = f' \cdot a' \in \mathcal{F}\ a,$ for all $a = a' \in \mathcal{X}\}$;
(2) $\{\!\{a\}\!\}_{\mathcal{X}} = \{(b, b') \mid a = b \in \mathcal{X}$ and $a = b' \in \mathcal{X}\}$.

Besides interpreting function spaces and singletons we need PERs for the denotation of the universe of small types, and for the set of large types; jointly with these PERs we need functions assigning a PER for each element in the domain of these universe PERs. Note that this forces the applicative structure to have some distinguished elements.

Definition 5.4 (Universe)**.** Given an applicative structure $\mathcal{A}$ with distinguished elements Fun and Sing, a universe $(\mathcal{U}, [\_])$ is a PER $\mathcal{U}$ over $\mathcal{A}$ and a family $[\_] \colon \mathcal{U} \to \mathsf{Per}(\mathcal{A})$ with the condition that $\mathcal{U}$ is closed under function and singleton types. This means:
(1) Whenever $X = X' \in \mathcal{U}$ and for all $a = a' \in [X]$, $F\ a = F'\ a' \in \mathcal{U}$, then $\mathsf{Fun}\, X\, F = \mathsf{Fun}\, X'\, F' \in \mathcal{U}$, with $[\mathsf{Fun}\, X\, F] = \prod [X]\, (a \mapsto [F\ a])$.
(2) Whenever $X = X' \in \mathcal{U}$ and $a = a' \in [X]$, then $\mathsf{Sing}\, a\, X = \mathsf{Sing}\, a'\, X' \in \mathcal{U}$ and $[\mathsf{Sing}\, a\, X] = \{\!\{a\}\!\}_{[X]}$.

An applicative structure paired with one universe for small types and one universe for large types is the minimal structure needed for having a model of our theory.

Definition 5.5 (PER model)**.** Let $\mathcal{A}$ be an applicative structure with distinguished elements U, Fun, and Sing; a *PER model* is a tuple $(\mathcal{A}, \mathcal{U}, \mathcal{T}, [\_])$ satisfying:
(1) $\mathcal{U} \subset \mathcal{T} \in \mathit{PER}(\mathcal{A})$, such that $(\mathcal{T}, [\_])$ and $(\mathcal{U}, [\_] \restriction_{\mathcal{U}})$ are both universes, and
(2) $\mathsf{U} \in \mathit{dom}(\mathcal{T})$, with $[\mathsf{U}] = \mathcal{U}$.

In the following definition we introduce an abstract concept for environments: since variables are represented as projection functions from lists (think of q as taking the head of a list, and p as taking the tail), it is enough having sequences together with projections.

Definition 5.6 (Sequences)**.** Given a set $\mathcal{A}$, a set $\mathcal{A}^*$ *has sequences* over $\mathcal{A}$ if there are distinguished operations $\top : \mathcal{A}^*$, $\mathsf{Pair} : \mathcal{A}^* \times \mathcal{A} \to \mathcal{A}^*$, $\mathsf{fst} : \mathcal{A}^* \to \mathcal{A}^*$, and $\mathsf{snd} : \mathcal{A}^* \to \mathcal{A}$ such that

$$\mathsf{fst}\,(\mathsf{Pair}\,a\,b) = a$$
$$\mathsf{snd}\,(\mathsf{Pair}\,a\,b) = b.$$

Now we need to extend the notion of PERs over $\mathcal{A}$ to PERs over $\mathcal{A}^*$ for interpreting substitutions.[4]

Definition 5.7. Let $\mathcal{A}$ be an applicative structure and let $\mathcal{A}^*$ have sequences over $\mathcal{A}$; moreover let $\mathcal{X} \in \mathrm{PER}(\mathcal{A}^*)$ and $\mathcal{F} \in \mathcal{X} \to \mathrm{PER}(\mathcal{A})$.
(1) $\mathbf{1} = \{(\top, \top)\}$;
(2) $\coprod \mathcal{X}\,\mathcal{F} = \{(a, a') \mid \mathsf{fst}\ a = \mathsf{fst}\ a' \in \mathcal{X} \text{ and } \mathsf{snd}\ a = \mathsf{snd}\ a' \in \mathcal{F}\ (\mathsf{fst}\ a)\}$;

Until here we have introduced semantic concepts. Now we are going to axiomatise the notion of evaluation, connecting the syntactic realm with the semantic one.

Definition 5.8 (Environment model)**.** Let $(\mathcal{A}, \mathcal{U}, \mathcal{T}, [\_])$ be a PER model and let $\mathcal{A}^*$ have sequences over $\mathcal{A}$. We call $\mathcal{M} = (\mathcal{A}, \mathcal{U}, \mathcal{T}, [\_], \mathcal{A}^*, [\![\,]\!], {}^{\mathsf{s}}[\![\,]\!])$ an *environment model* if the *evaluation functions* $[\![\_]\!]\_ \colon$ *Terms* $\times \mathcal{A}^* \to \mathcal{A}$ and ${}^{\mathsf{s}}[\![\_]\!]\_ \colon$ *Terms* $\times \mathcal{A}^* \to \mathcal{A}^*$ satisfy:

$$\begin{array}{rcl@{\qquad}rcl}
{}^{\mathsf{s}}[\![\mathsf{id}]\!]a &=& a & [\![\mathsf{U}]\!]a &=& \mathsf{U} \\
{}^{\mathsf{s}}[\![\langle\rangle]\!]a &=& \top & [\![\mathsf{Fun}\,A\,B]\!]a &=& \mathsf{Fun}\,([\![A]\!]a)\,F, \text{ where } F\ b = [\![B]\!](\mathsf{Pair}\ a\ b) \\
{}^{\mathsf{s}}[\![\sigma\,\delta]\!]a &=& [\![\sigma]\!]([\![\delta]\!]a) & [\![\{t\}_A]\!]a &=& \mathsf{Sing}\ ([\![t]\!]a)\ ([\![A]\!]a) \\
{}^{\mathsf{s}}[\![(\sigma, t)]\!]a &=& \mathsf{Pair}\ ([\![\sigma]\!]a)\ ([\![t]\!]a) & [\![t\,\sigma]\!]a &=& [\![t]\!]({}^{\mathsf{s}}[\![\sigma]\!]a) \\
{}^{\mathsf{s}}[\![\mathsf{p}]\!]a &=& \mathsf{fst}\ a & [\![\lambda t]\!]a &=& f, \text{ where } f \cdot b = [\![t]\!](\mathsf{Pair}\ a\ b) \\
 & & & [\![\mathsf{app}\ t\ u]\!]a &=& ([\![t]\!]a) \cdot ([\![u]\!]a) \\
 & & & [\![\mathsf{q}]\!]a &=& \mathsf{snd}\ a
\end{array}$$

Since no ambiguities arise, we shall henceforth write $[\![\sigma]\!]$ instead of ${}^{\mathsf{s}}[\![\sigma]\!]$.

Once we have an environment model $\mathcal{M}$ we can define the denotation for contexts. The second clause in the next definition is not well-defined a priori; its totality is a corollary of Thm. 5.11.

Definition 5.9. Given an environment model $\mathcal{M}$, we define recursively the semantic of contexts $(\![\_]\!) \colon \mathsf{Ctx} \to \mathrm{PER}(\mathcal{A}^*)$:
(1) $(\![\diamond]\!) = \mathbf{1}$,
(2) $(\![\Gamma.A]\!) = \coprod\, (\![\Gamma]\!)(a \mapsto [[\![A]\!]a])\,.$

We use PERs for validating equality judgements and the domain of each PER for validating typing judgements.

Definition 5.10 (Validity)**.** Let $\mathcal{M}$ be an environment model. We define inductively the predicate of satisfability of judgements by the model, denoted with $\Gamma \vDash^{\mathcal{M}} J$:

[4]The reader is invited to think of $\mathbf{1}$ as the terminal object of the category of PERs over $\mathcal{A}^*$ and PER preserving morphisms; looked this way our definition for $\mathbf{1}$ does not differ very much from others [10, 21].

(1) $\diamond \models$ iff true
(2) $\Gamma.A \models$ iff $\Gamma \models A$
(3) $\Gamma \models A$ iff $\Gamma \models A = A$
(4) $\Gamma \models A = A'$ iff $\Gamma \models$ and for all $d = d' \in (\![\Gamma]\!)$, $[\![A]\!]d = [\![A']\!]d' \in \mathcal{T}$
(5) $\Gamma \models t : A$ iff $\Gamma \models t = t : A$
(6) $\Gamma \models t = t' : A$ iff $\Gamma \models A$ and for all $d = d' \in (\![\Gamma]\!)$, $[\![t]\!]d = [\![t']\!]d' \in [[\![A]\!]d]$
(7) $\Gamma \models \sigma : \Delta$ iff $\Gamma \models \sigma = \sigma : \Delta$
(8) $\Gamma \models \sigma = \sigma' : \Delta$ iff $\Gamma \models$, $\Delta \models$, and for all $d = d' \in (\![\Gamma]\!)$, $[\![\sigma]\!]d = [\![\sigma']\!]d' \in (\![\Delta]\!)$.

Theorem 5.11 (Soundness of the Judgements). *Let $\mathcal{M}$ be a model. If $\Gamma \vdash J$, then $\Gamma \models^{\mathcal{M}} J$.*

Proof. By easy induction on $\Gamma \vdash J$. □

5.2. **A concrete PER model.** In this subsection we define a concrete PER model over a Scott domain. The definition of the evaluation function is post-poned to the next subsection after introducing the NbE machinery.

Definition 5.12. We define a domain

$$D = \mathbb{O} \oplus \mathit{Var}_\perp \oplus [D \to D] \oplus (D \times D) \oplus (D \times D) \oplus \mathbb{O} \oplus (D \times [D \to D]) \oplus (D \times D) \ ,$$

where Var is a denumerable set of variables (as usual we write x_i and assume $x_i \neq x_j$ if $i \neq j$, for $i, j \in \mathbb{N}$), $E_\perp = E \cup \{\perp\}$ is lifting, $\mathbb{O} = \{\top\}_\perp$ is the Sierpinski space, $[D \to D]$ is the set of continuous functions from D to D, $\oplus$ is the coalesced sum (this is the disjoint union where all the bottoms elements are identified), and $D \times D$ is the Cartesian product of D [6].

An element of D which is not $\perp$ can be of one of the forms:

$\top$	(d, d')	for $d, d' \in D$
$\mathsf{Var}\, x_i$	U	for $x_i \in \mathit{Var}$
$\mathsf{Lam}\, f$	$\mathsf{Fun}\, d\, f$	for $d \in D$, and $f \in [D \to D]$
$\mathsf{App}\, d\, d'$	$\mathsf{Sing}\, d\, d'$	for $d, d' \in D$.

Elements of the form $\mathsf{Var}\, x_i$ and $\mathsf{App}\, d\, d'$ are called *neutral*; in this section, we reuse the letter k to denote neutral elements of D.

In order to define an environment model over D, we endow it with an applicative structure. Note also that D has pairing, letting us to take the set of sequences over D simply as $D^* = D$ with $\mathsf{Pair}\, a\, b = (a, b)$. We define application $\_ \cdot \_ : [D \times D \to D]$ and the projections $\mathsf{fst}, \mathsf{snd} : [D \to D]$ by

$$\begin{array}{rcl} f \cdot d &=& \text{if } f = \mathsf{Lam}\, f' \text{ then } f'\, d \text{ else } \perp, \\ \mathsf{fst}\, d &=& \text{if } d = (d_1, d_2) \text{ then } d_1 \text{ else } \perp, \\ \mathsf{snd}\, d &=& \text{if } d = (d_1, d_2) \text{ then } d_2 \text{ else } \perp. \end{array}$$

We define a partial function $\mathsf{R}_{\_}\, \_ : \mathbb{N} \to D \to \mathit{Terms}$ which reifies elements from the model into terms; this function is similar to Grégoire and Leroy's read-back function [31].

Definition 5.13 (Read-back function).

$$\begin{array}{rclrcl} \mathsf{R}_j\, \mathsf{U} &=& \mathsf{U} & \mathsf{R}_j\, (\mathsf{App}\, d\, d') &=& \mathsf{app}\, (\mathsf{R}_j\, d)\, (\mathsf{R}_j\, d') \\ \mathsf{R}_j\, (\mathsf{Fun}\, X\, F) &=& \mathsf{Fun}\, (\mathsf{R}_j\, X)\, (\mathsf{R}_{j+1}\, (F(\mathsf{Var}\, x_j))) & \mathsf{R}_j\, (\mathsf{Lam}\, f) &=& \lambda(\mathsf{R}_{j+1}\, (f(\mathsf{Var}\, x_j))) \\ \mathsf{R}_j\, (\mathsf{Sing}\, d\, X) &=& \{\mathsf{R}_j\, d\}_{\mathsf{R}_j\, X} & \mathsf{R}_j\, (\mathsf{Var}\, x_i) &=& \mathsf{v}_{j-(i+1)} \end{array}$$

As explained in Sect. 4.2, the reification of variables turns de Bruijn levels into de Bruijn indices. Note that in case $j < i + 1$ we return the 0th de Bruijn index just not to be undefined; we will come back to this later.

The next PERs contain those elements of the domain D whose reification is defined for any context length. Moreover, their elements are reified as neutral terms and normal forms, respectively; allowing us to reason semantically about normal forms. Remember that $t \equiv_T t'$ denotes that t is syntactically equal to t' and $t \in T$.

Definition 5.14 ((Semantic) neutral terms and normal forms)**.**

$$\begin{array}{lll} d = d' \in \mathcal{N}e & :\Longleftrightarrow & \text{for all } i \in \mathbb{N},\ \mathsf{R}_i\, d \text{ and } \mathsf{R}_i\, d' \text{ are defined and } \mathsf{R}_i\, d \equiv_{Ne} \mathsf{R}_i\, d' \\ d = d' \in \mathcal{N}f & :\Longleftrightarrow & \text{for all } i \in \mathbb{N},\ \mathsf{R}_i\, d \text{ and } \mathsf{R}_i\, d' \text{ are defined and } \mathsf{R}_i\, d \equiv_{Nf} \mathsf{R}_i\, d' \end{array}$$

Notice that if the case $j < i + 1$ were undefined in the clause for variables in 5.13, then for any m the application $\mathsf{R}_0\, \mathsf{Var}\, x_m$ would be undefined; hence $\mathsf{Var}\, x_m \notin \mathcal{N}e$ and, consequently, $\mathcal{N}e$ would be empty. Since we depend on having a semantic representation of variables and neutrals we add the case $j < i + 1$. This case will not arise in our use of the readback function.

Remark 5.15. These are clearly PERs over D: symmetry is trivial and transitivity follows from transitivity of the syntactical equality.

Lemma 5.16 (Closure properties of $\mathcal{N}e$ and $\mathcal{N}f$)**.**
(1) $\mathsf{U} = \mathsf{U} \in \mathcal{N}f$.
(2) *Let* $X = X' \in \mathcal{N}e$. *If* $F \cdot k = F' \cdot k' \in \mathcal{N}f$ *for all* $k = k' \in \mathcal{N}e$, *then* $\mathsf{Fun}\, X\, F = \mathsf{Fun}\, X'\, F' \in \mathcal{N}f$.
(3) *If* $d = d' \in \mathcal{N}f$ *and* $X = X' \in \mathcal{N}f$, *then* $\mathsf{Sing}\, d\, X = \mathsf{Sing}\, d'\, X' \in \mathcal{N}f$.
(4) *If* $f \cdot k = f' \cdot k' \in \mathcal{N}f$ *for all* $k = k' \in \mathcal{N}e$, *then* $f = f' \in \mathcal{N}f$.
(5) $\mathsf{Var}\, x_i = \mathsf{Var}\, x_i \in \mathcal{N}e$ *for all* $i \in \mathbb{N}$.
(6) *If* $k = k' \in \mathcal{N}e$ *and* $d = d' \in \mathcal{N}f$, *then* $\mathsf{App}\, k\, d = \mathsf{App}\, k'\, d' \in \mathcal{N}e$.

We define $\mathcal{U}, \mathcal{T} \in \mathrm{PER}(D)$ and $[\_] : dom(\mathcal{T}) \to \mathrm{PER}(D)$ using Dybjer's schema of inductive-recursive definition [25]. We show then that $[\_]$ is a family of PERs over D.

Definition 5.17 (PER model)**.**
(1) Inductive definition of $\mathcal{U} \in \mathrm{PER}(D)$.
   (a) $\mathcal{N}e \subseteq \mathcal{U}$,
   (b) if $X = X' \in \mathcal{U}$ and $d = d' \in [X]$, then $\mathsf{Sing}\, d\, X = \mathsf{Sing}\, d'\, X' \in \mathcal{U}$,
   (c) if $X = X' \in \mathcal{U}$ and for all $d = d' \in [X]$, $F\ d = F'\ d' \in \mathcal{U}$ then $\mathsf{Fun}\, X\, F = \mathsf{Fun}\, X'\, F' \in \mathcal{U}$.
(2) Inductive definition of $\mathcal{T} \in \mathrm{PER}(D)$.
   (a) $\mathcal{U} \subseteq \mathcal{T}$,
   (b) $\mathsf{U} = \mathsf{U} \in \mathcal{T}$,
   (c) if $X = X' \in \mathcal{T}$, and $d = d' \in [X]$ then $\mathsf{Sing}\, d\, X = \mathsf{Sing}\, d'\, X' \in \mathcal{T}$,
   (d) if $X = X' \in \mathcal{T}$, and for all $d = d' \in [X]$, $F\ d = F'\ d' \in \mathcal{T}$, then $\mathsf{Fun}\, X\, F = \mathsf{Fun}\, X'\, F' \in \mathcal{T}$.
(3) Recursive definition of $[\_] \in dom(\mathcal{T}) \to \mathrm{PER}(D)$.
   (a) $[\mathsf{U}] = \mathcal{U}$,
   (b) $[\mathsf{Sing}\, d\, X] = \{\!\{d\}\!\}_{[X]}$,
   (c) $[\mathsf{Fun}\, X\, F] = \prod [X]\, (d \mapsto [F\ d])$,

(d) $[d] = \mathcal{N}e$, in all other cases.

Remark 5.18. The generation order $\sqsubset$ on $\mathcal{T}$ is well-founded. The minimal elements are U, and elements in $\mathcal{N}e$; $X \sqsubset \mathsf{Fun}\, X\, F$, and for all $d \in [X]$, $F\ d \sqsubset \mathsf{Fun}\, X\, F$; and, finally, $X \sqsubset \mathsf{Sing}\, d\, X$.

Lemma 5.19. *The function* $[\_] : dom(\mathcal{T}) \to PER(D)$ *is a family of* $PER(D)$ *over* $\mathcal{T}$*, i.e.,* $[\_] : \mathcal{T} \to PER(D)$*.*

Proof. By induction on $X = X' \in \mathcal{T}$. See Appendix C.1. □

The previous lemma leads us to the definition of a PER model over D. Note also that D has all the distinguished elements needed to call it a syntactical applicative structure.

Corollary 5.20. *The tuple* $(D, \mathcal{U}, \mathcal{T}, [\_])$ *is a PER model.* □

5.3. **Normalisation and η-Expansion in the Model.** In the following, we adopt the NbE algorithm outlined in Section 4 to the dependent type theory λ^{Sing}. Since read-back has already be defined, we only require reflection, reification and evaluation functions.

Definition 5.21 (Reflection and reification)**.** The partial functions $\uparrow_{\_}\,\_, \downarrow_{\_}\,\_ : [D \to [D \to D]]$ and $\Downarrow : [D \to D]$ are given as follows:

$$\begin{array}{rcl@{\qquad}rcl}
\uparrow_{\mathsf{Fun}\, X\, F} k &=& \mathsf{Lam}\,(d \mapsto \uparrow_{F\, d} (\mathsf{App}\, k \downarrow_X d)) & \downarrow_{\mathsf{Fun}\, X\, F} d &=& \mathsf{Lam}\,(e \mapsto \downarrow_{F\, \uparrow_X e} (d \cdot \uparrow_X e)) \\
\uparrow_{\mathsf{Sing}\, d\, X} k &=& d & \downarrow_{\mathsf{Sing}\, d\, X} e &=& \downarrow_X d \\
\uparrow_{\mathsf{U}} k &=& k & \downarrow_{\mathsf{U}} d &=& \Downarrow d \\
\uparrow_X k &=& k\text{, in all other cases.} & \downarrow_X e &=& e\text{, in all other cases.}
\end{array}$$

$$\begin{aligned}
&\Downarrow(\mathsf{Fun}\, X\, F) = \mathsf{Fun}\,(\Downarrow X)\,(d \mapsto \Downarrow(F \uparrow_X d)) \\
&\Downarrow(\mathsf{Sing}\, d\, X) = \mathsf{Sing}\,(\downarrow_X d)\,(\Downarrow X) \\
&\Downarrow \mathsf{U} = \mathsf{U} \\
&\Downarrow X = X\text{, in all other cases.}
\end{aligned}$$

In the following lemma we show that reflection $\uparrow$ corresponds to Berger and Schwichtenberg's "make self evaluating" and both reification functions $\downarrow$ and $\Downarrow$ correspond to"inverse of the evaluation function" [16]. Note that they are indexed by types values instead of syntactic types, since we are dealing with dependent instead of simple types.

Lemma 5.22 (Characterisation of $\uparrow$, $\downarrow$, and $\Downarrow$)**.** *Let* $X = X' \in \mathcal{T}$*, then*
(1) *if* $k = k' \in \mathcal{N}e$ *then* $\uparrow_X k = \uparrow_{X'} k' \in [X]$*;*
(2) *if* $d = d' \in [X]$*, then* $\downarrow_X d = \downarrow_{X'} d' \in \mathcal{N}f$*;*
(3) *and also* $\Downarrow X = \Downarrow X' \in \mathcal{N}f$*.*

Proof. By induction on $X = X' \in \mathcal{T}$. See C.2. □

Let us recapitulate what we have achieved: we have defined a PER model over the domain D; then we defined a family of functions $\downarrow_X$ indexed over denotation of types with the property that when applied to elements in the corresponding PER we get back elements which will be reified as normal forms. In fact, we have the stronger result that whenever we apply $\downarrow_X$ to two related elements $d = d' \in [X]$ we get elements to be reified as the same term.

Now we define evaluation which clearly satisfies the environment model conditions in Def. 5.8; hence, we have a model and, using Thm. 5.11, we conclude completeness for our normalisation algorithm.

Definition 5.23 (Semantics)**.** Evaluation of substitutions and terms into D is defined inductively by the following equations.

Substitutions.

$$\begin{aligned} [\![\diamond]\!] d &= \top & [\![\mathsf{id}]\!] d &= d \\ [\![(\gamma, t)]\!] d &= ([\![\gamma]\!] d, [\![t]\!] d) & [\![\mathsf{p}]\!] d &= \mathsf{fst}\ d \\ [\![\gamma\, \delta]\!] d &= [\![\gamma]\!]([\![\delta]\!] d) \end{aligned}$$

Terms (and types).

$$\begin{aligned} [\![\mathsf{U}]\!] d &= \mathsf{U} & [\![\mathsf{Fun}\, A\, B]\!] d &= \mathsf{Fun}\, ([\![A]\!] d)\, (e \mapsto [\![B]\!](d, e)) \\ [\![\{a\}_A]\!] d &= \mathsf{Sing}\, ([\![a]\!] d)\, ([\![A]\!] d) & [\![\mathsf{app}\ t\ u]\!] d &= [\![t]\!] d \cdot [\![u]\!] d \\ [\![\lambda t]\!] d &= \mathsf{Lam}\, (d' \mapsto [\![t]\!](d, d')) & [\![t\, \gamma]\!] d &= [\![t]\!]([\![\gamma]\!] d) \\ [\![\mathsf{q}]\!] d &= \mathsf{snd}\ d \end{aligned}$$

Theorem 5.24 (Completeness of NbE)**.** *Let $\Gamma \vdash t = t' : A$ and let also $d \in (\![\Gamma]\!)$, then $\downarrow_{[\![A]\!] d} [\![t]\!] d = \downarrow_{[\![A]\!] d} [\![t']\!] d \in \mathcal{N}f$.*

Proof. By Thm. 5.11 we have $[\![t]\!] d = [\![t']\!] d \in [[\![A]\!] d]$ and we conclude by Lem. 5.22. □

5.4. **Calculus $\lambda^{\mathsf{Irr}}$with proof irrelevance.** We extend all the definitions concerning the construction of the model.

Definition 5.25 (Extension of domain D)**.**

$$\begin{aligned} D = \ldots &\oplus D \times [D \to D] \oplus D \oplus D \\ &\oplus \mathbb{O} \oplus \mathbb{O} \oplus D \oplus [D \to D] \times D \times [D \to [D \to D]] \times D \\ &\oplus D \oplus \mathbb{O} \oplus \mathbb{N} \oplus \mathbb{N} \times \mathbb{N} \oplus \mathbb{N} \times [D \to D] \times D^\omega \times D\ . \end{aligned}$$

We use the following notations for the injections into D:

$\mathsf{Sum}\, d\, F$	$\mathsf{Fst}\ d,\ \mathsf{Snd}\ d$	for $d \in D, F \in [D \to D]$
zero	Nat	$\star$
$\mathsf{suc}\, d$	$\mathsf{Prf}\, d$	for $d \in D$
N_n	c_i^n	for $i, n \in \mathbb{N}$
$\mathsf{Natrec}(F, d, g, d')$		for $d, d' \in D, F \in [D \to D], g \in [D \to [D \to D]]$
$\mathsf{Case}^n(F, \vec{d}, d')$		for $d, d' \in D, F \in [D \to D], \vec{d} \in D^\omega, n \in \mathbb{N}$

In this extension, the injections Fst, Snd, Natrec, and Case construct neutral elements k. Soundness for the calculus ($\vdash^\star$) requires the canonical element for proof-irrelevant types ($\star$) to be in every PER; thus we need to redefine application $\_ \cdot \_$ to have $\star \in [\mathsf{Fun}\, X\, F]$:

$$\star \cdot d = \star\ .$$

We also redefine the projections fst and snd to account for neutrals and because they are used in the definition of $\coprod X F$, which will be used as the denotation of sigma types.

$$\mathsf{fst}\ d = \begin{cases} d_1 & \text{if } d = (d_1, d_2) \\ \star & \text{if } d = \star \\ \mathsf{Fst}\ d & \text{otherwise} \end{cases} \qquad \mathsf{snd}\ d = \begin{cases} d_2 & \text{if } d = (d_1, d_2) \\ \star & \text{if } d = \star \\ \mathsf{Snd}\ d & \text{otherwise} \end{cases}$$

Definition 5.26 (Read-back function)**.**

$$\begin{aligned}
\mathsf{R}_j\,(\mathsf{Sum}\, X\, F) &= \Sigma\,(\mathsf{R}_j\, X) & \mathsf{R}_j\,(d, d') &= (\mathsf{R}_j\, d, \mathsf{R}_j\, d') \\
&\quad\ (\mathsf{R}_{j+1}\,(F\ \mathsf{Var}\, x_j)) & \mathsf{R}_j\,(\mathsf{Fst}\ d) &= \mathsf{fst}\ (\mathsf{R}_j\, d) \\
\mathsf{R}_j\, \mathsf{Nat} &= \mathsf{Nat} & \mathsf{R}_j\,(\mathsf{Snd}\ d) &= \mathsf{snd}\ (\mathsf{R}_j\, d) \\
\mathsf{R}_j\, \mathsf{zero} &= \mathsf{zero} & \mathsf{R}_j\,(\mathsf{Natrec}(F, d, f, e)) &= \mathsf{natrec}\ (\mathsf{R}_{j+1}\,(F\ \mathsf{Var}\, x_j)) \\
\mathsf{R}_j\,(\mathsf{suc}\, d) &= \mathsf{suc}\ (\mathsf{R}_j\, d) & &\quad\ (\mathsf{R}_j\, d)\ (\mathsf{R}_j\, f)\ (\mathsf{R}_j\, e) \\
\mathsf{R}_j\,(\mathsf{Prf}\, d) &= \mathsf{Prf}\ (\mathsf{R}_j\, d) & \mathsf{R}_j\, \star &= \star \\
\mathsf{R}_j\,(\mathsf{N}_n) &= \mathsf{N}_n & \mathsf{R}_j\,(\mathsf{c}^n_i) &= \mathsf{c}^n_i
\end{aligned}$$

$$\mathsf{R}_j\,(\mathsf{Case}^n(F, \langle d_0, \ldots, d_{n-1}\rangle, e)) = \mathsf{case}^n\ (\mathsf{R}_{j+1}\,(F\ \mathsf{Var}\, x_j))\ (\mathsf{R}_j\, d_0) \cdots (\mathsf{R}_j\, d_{n-1})\ (\mathsf{R}_j\, e)$$

We define inductively new PERs for interpreting naturals and finite types. Note that $\mathcal{C}_0$ and $\mathcal{C}_1$ are irrelevant, in this way we can model η-expansion for N_0 and N_1; $|\mathcal{X}|$ is also irrelevant, even when $\mathcal{X}$ distinguishes its elements.

Definition 5.27 (More semantic types)**.**

(1) $\mathcal{N}$ is the smallest PER over D, such that
  (a) $\mathcal{N}e \subseteq \mathcal{N}$
  (b) $\mathsf{zero} = \mathsf{zero} \in \mathcal{N}$
  (c) $\mathsf{suc}\, d = \mathsf{suc}\, d' \in \mathcal{N}$, if $d = d' \in \mathcal{N}$
(2) If $\mathcal{X} \in \mathrm{PER}(D)$ then $|\mathcal{X}| := \{(d, d') \mid d, d' \in \mathit{dom}(\mathcal{X}) \cup \{\star\}\} \in \mathrm{PER}(D)$.
(3) $\mathcal{C}_0 = |\emptyset| = \{(\star, \star)\}$,
(4) $\mathcal{C}_1 = |\{\mathsf{c}^1_0\}| = \{(d, d') \mid d, d' \in \{\star, \mathsf{c}^1_0\}\}$,
(5) $\mathcal{C}_n = \{(\mathsf{c}^n_i, \mathsf{c}^n_i) \mid i < n\} \cup \mathcal{N}e$, for $n \geqslant 2$.

We add new clauses in the definitions of the partial equivalences for universe and types, these clauses do not affect the well-foundedness of the order $\sqsubset$ defined in 5.18, but now we have that N_n and Nat are also minimal elements for that order.

Definition 5.28 (Extension of $\mathcal{U}$ and $\mathcal{T}$)**.**

(1) Inductive definition of $\mathcal{U} \in \mathrm{PER}(D)$.
  (a) If $X = X' \in \mathcal{U}$, and for all $d = d' \in [X]$, $F\, d = F'\, d' \in \mathcal{U}$, then $\mathsf{Sum}\, X\, F = \mathsf{Sum}\, X'\, F' \in \mathcal{U}$.
  (b) $\mathsf{Nat} = \mathsf{Nat} \in \mathcal{U}$,
  (c) $\mathsf{N}_n = \mathsf{N}_n \in \mathcal{U}$,
  (d) if $X = X' \in \mathcal{U}$, then $\mathsf{Prf}\, X = \mathsf{Prf}\, X' \in \mathcal{U}$.
(2) Inductive definition of $\mathcal{T} \in \mathrm{PER}(D)$.

   (a) If $X = X' \in \mathcal{T}$, and for all $d = d' \in [X]$, $F\,d = F'\,d' \in \mathcal{T}$, then $\mathsf{Sum}\,X\,F = \mathsf{Sum}\,X'\,F' \in \mathcal{T}$.
   (b) if $X = X' \in \mathcal{T}$, then $\mathsf{Prf}\,X = \mathsf{Prf}\,X' \in \mathcal{T}$.

(3) Recursive definition of $[\_] \in \mathit{dom}(\mathcal{T}) \to \mathrm{PER}(D)$.
   (a) $[\mathsf{Sum}\,X\,F] = \coprod [X]\,(d \mapsto [F\ d])$,
   (b) $[\mathsf{N}_n] = \mathcal{C}_n$
   (c) $[\mathsf{Nat}] = \mathcal{N}$,
   (d) if $X \in \mathit{dom}(\mathcal{T})$, then $[\mathsf{Prf}\,X] = \{(\star,\star)\}$.

Note that in the PER model, all propositions $\mathsf{Prf}\,X$ are inhabited. In fact, all types are inhabited, for there is a reflection from variables into any type, be it empty or not. So, the PER model is unsuited for refuting propositions. However, the logical relation we define in the next section will only be inhabited for non-empty types.

Remark 5.29. It can be proved by induction on $X \in \mathcal{T}$ that $\star \in [X]$.

Definition 5.30 (Reflection and reification, *cf.* 5.21)**.**

$$\begin{array}{rcll}
\uparrow_{\mathsf{Sum}\,X\,F} k &=& (\uparrow_X \mathsf{Fst}\,k, \uparrow_{F\,(\uparrow_X \mathsf{Fst}\,k)} \mathsf{Snd}\,k) & \\
\uparrow_{\mathsf{Nat}} k &=& k & \\
\uparrow_{\mathsf{N}_0} k &=& \star & \\
\uparrow_{\mathsf{N}_1} k &=& \mathsf{c}^1_0 & \\
\uparrow_{\mathsf{N}_n} k &=& k & \text{for } n \geqslant 2 \\
\uparrow_{\mathsf{Prf}\,X} k &=& \star &
\end{array}
\qquad
\begin{array}{rcl}
\downarrow_{\mathsf{Sum}\,X\,F} d &=& (\downarrow_X \mathsf{fst}\,d, \downarrow_{F\,(\mathsf{fst}\,d)} \mathsf{snd}\,d) \\
\downarrow_{\mathsf{Nat}} d &=& d \\
\downarrow_{\mathsf{N}_0} d &=& \star \\
\downarrow_{\mathsf{N}_1} d &=& \mathsf{c}^1_0 \\
\downarrow_{\mathsf{N}_n} d &=& d \\
\downarrow_{\mathsf{Prf}\,X} d &=& \star
\end{array}$$

$$\Downarrow \mathsf{Sum}\,X\,F = \mathsf{Sum}\,(\Downarrow X)\,(d \mapsto \Downarrow(F \uparrow_X d)) \qquad \Downarrow \mathsf{Nat} = \mathsf{Nat}$$
$$\Downarrow \mathsf{N}_n = \mathsf{N}_n \qquad \Downarrow \mathsf{Prf}\,X = \mathsf{Prf}\,(\Downarrow X)$$

For giving semantics to eliminators for data types we need to define partial functions $\mathsf{natrec} : [D \to D] \times D \times D \times D \to D$, and $\mathsf{case} : [D \to D] \times D \times D \times D \to D$.

Definition 5.31 (Eliminations on D)**.**

(1) Elimination operator for naturals.

$$\begin{array}{lcl}
\mathsf{natrec}(F,d,f,\star) &=& \star \\
\mathsf{natrec}(F,d,f,\mathsf{zero}) &=& d \\
\mathsf{natrec}(F,d,f,\mathsf{suc}\,e) &=& (f \cdot e) \cdot \mathsf{natrec}(F,d,f,e) \\
\mathsf{natrec}(F,d,f,k) &=& \uparrow_{F\ k} (\mathsf{Natrec}(d' \mapsto \Downarrow F\ d', \\
&& \qquad\qquad\quad \downarrow_{F\ \mathsf{zero}} d, \\
&& \qquad\qquad\quad \mathsf{Lam}\,d' \mapsto (\mathsf{Lam}\,e' \mapsto \downarrow_{F\ (\mathsf{suc}\,d')} f \cdot d' \cdot e'), \\
&& \qquad\qquad\quad k))
\end{array}$$

(2) Elimination operator for finite types.

$$\begin{array}{r}
\mathsf{case}^n(F, \langle d_0,\ldots,d_{n-1}\rangle, \star) = \star \\
\mathsf{case}^n(F, \langle d_0,\ldots,d_{n-1}\rangle, \mathsf{c}^n_i) = d_i \\
\mathsf{case}^n(F, \langle \mathsf{c}^n_0,\ldots,\mathsf{c}^n_{n-1}\rangle, d) = d \\
\mathsf{case}^n(F, \langle d_0,\ldots,d_{n-1}\rangle, k) = \uparrow_{F\ k} \mathsf{Case}^n(e \mapsto \Downarrow F\ e, \langle \downarrow_{F\ \mathsf{c}^n_0} d_0, \ldots, \downarrow_{F\ \mathsf{c}^n_{n-1}} d_{n-1}\rangle, k)
\end{array}$$

Remark 5.32. If for all $d = d' \in \mathcal{N}$, $F\ d = F'd' \in \mathcal{T}$, and $z = z' \in [F\ \mathsf{zero}]$, and for all $d = d' \in \mathcal{N}$ and $e = e' \in [F\ d]$, $s \cdot d \cdot e = s' \cdot d' \cdot e' \in [F(\mathsf{suc}\, d)]$, and $d = d' \in \mathcal{N}$ then $\mathsf{natrec}(F, z, s, d) = \mathsf{natrec}(F, z, s, d') \in [F\ d]$.

With these new definitions we can now give the semantic equations for the new constructs.

Definition 5.33 (Extension of interpretation)**.**

$$\begin{array}{rclrcl}
[\![\Sigma\, A\, B]\!]d &=& \mathsf{Sum}\ ([\![A]\!]d)\ (d' \mapsto [\![B]\!](d, d')) & [\![\mathsf{N}_n]\!]d &=& \mathsf{N}_n \\
[\![\mathsf{Nat}]\!]d &=& \mathsf{Nat} & [\![\mathsf{Prf}\ A]\!]d &=& \mathsf{Prf}\ [\![A]\!]d \\
[\![\mathsf{fst}\ t]\!]d &=& \mathsf{fst}\ [\![t]\!]d & [\![\mathsf{snd}\ t]\!]d &=& \mathsf{snd}\ [\![t]\!]d \\
[\![(t, t')]\!]d &=& ([\![t]\!]d, [\![t']\!]d) & [\![\mathsf{zero}]\!]d &=& \mathsf{zero} \\
[\![\mathsf{natrec}\ B\ z\ s\ t]\!]d &=& \mathsf{natrec}(e \mapsto [\![B]\!](d, e), [\![z]\!]d, [\![s]\!]d, [\![t]\!]d) & [\![\mathsf{suc}\ t]\!]d &=& \mathsf{suc}\ [\![t]\!]d \\
[\![[a]]\!]d &=& \star & [\![\star]\!]d &=& \star \\
[\![b\, \mathsf{where}^B\ t]\!]d &=& [\![b]\!](d, \star) & [\![\mathsf{c}_i^n]\!]d &=& \mathsf{c}_i^n
\end{array}$$

$$[\![\mathsf{case}^n\ B\ t_0 \cdots t_{n-1}\ t]\!]d = \mathsf{case}^n(e \mapsto [\![B]\!](d, e), \langle [\![t_0]\!]d, \ldots, [\![t_{n-1}]\!]d\rangle, [\![t]\!]d)$$

Lemma 5.34 (Laws of proof elimination)**.** *β, η, and associativity for* where *are modeled by the extended applicative structure.*

Proof. See C.3. □

Remark 5.35. All of lemmata 5.19, 5.22, and theorems 5.11, and 5.24 are valid for the extended calculus.

Note that we have defined a *proof-irrelevant* semantics for $(\vdash^\star)$ that collapses all elements of $\mathsf{Prf}\, A$ to $\star$, which leads to a more efficient implementation of the normalisation function. However, this semantics is not sound if λ^{Irr} is extended with singleton types interpreted analogously to $\mathcal{C}_1$, i.e., $[\mathsf{Sing}\, d\, X] = |\{\!\{d\}\!\}_X|$, because it does not model (SING-EQ-EL). (We have $d = \star \in [\mathsf{Sing}\, d\, X]$ for all $d \in [X]$, but not necessarily $d = \star \in [X]$.) On the other hand, λ^{Irr} without $\star$ can be extended to singleton types as explained in the following remark.

Remark 5.36 (Extending λ^{Irr} by singleton types)**.** Singleton types can be added straightforwardly if we employ a *proof-relevant* semantics:

The domain D is not changed; in particular we have $\star \in D$, and it is readback as before, $\mathsf{R}_j \star = \star$; hence $\star \in \mathit{dom}(\mathcal{N}f)$.

All the enumerated types are modelled in a uniform way: $[\mathsf{N}_n] = \{(\mathsf{c}_i^n, \mathsf{c}_i^n) \mid i < n\} \cup \mathcal{N}e$; proof-irrelevance types $\mathsf{Prf}\, A$ are interpreted as the irrelevant PER with the same domain as the PER for A: $[\mathsf{Prf}\, X] = \{(d, d') \mid d, d' \in \mathit{dom}([X])\}$. Reflection and reification for $\mathsf{Prf}\, X$ are defined respectively as

$$\uparrow_{\mathsf{Prf}\, X} d = \uparrow_X d \qquad \text{and} \qquad \downarrow_{\mathsf{Prf}\, X} d = \star\ .$$

With these definitions it is clear that the corresponding result for Lem. 5.22 is still valid.

Since $\mathit{dom}([\mathsf{Prf}\, X]) = \mathit{dom}([X])$, introduction and elimination of proofs can be interpreted as follows

$$[\![[a]]\!]d = [\![a]\!]d \quad \text{and} \quad [\![b\, \mathsf{where}^B\ t]\!]d = [\![b]\!](d, [\![t]\!]d)\ ;$$

this model is sound with respect to the calculus $(\vdash)$ extended with singleton types; hence Thm. 5.24 is valid.

Remark 5.37. As was previously said we cannot use this PER model for proving that there is no closed term in N_0. Instead, one can build up a PER model, in the sense of 5.1, of closed values, where $[\mathsf{N}_0] = \emptyset$. By soundness (Thm. 5.11) it follows that there is no possible derivation of $\vdash t : \mathsf{N}_0$.

6. Correctness of NbE

In Thm. 5.24 we have proved that the NbE algorithm is complete with respect to the judgemntal equality of our calculi; a corollary of that fact is totality of NbE.

Remark 6.1. Let $\Gamma \vdash t : A$. Given some $d \in (\![\Gamma]\!)$, we can conclude $\mathsf{R}_i\,(\downarrow_{[\![A]\!]d} [\![t]\!]d)$ is a well-defined term in normal form.

In this section we prove correctness, with respect to the typing rules, for our NbE algorithm. This means that given a typing $\Gamma \vdash t : A$, when NbE is applied to t, the resulting normal form v, is provable equal to t; i.e. $\Gamma \vdash t = v : A$.

Let us anticipate the main results of this section. As a corollary of Thm. 6.11 we show that a term is related to its denotation with respect to some canonical environment (to be defined in Def. 6.12). Previously we prove in Lem. 6.6 that if a term is logically related with some semantic element, then its reification will be judgmentally equal to the term. Composing these facts we obtain correctness. As a consequence of having correctness and completeness for NbE, one gets decidability for judgmentally equality: normalise both terms and check they are syntactically the same. Another important corollary is injectivity for constructors.

6.1. Logical relations.

In this subsection we define logical relations and prove some technical lemmas about them. As is standard with logical relations one defines them by induction on types (here we define by induction on semantics of types, *i.e.* elements of $\mathcal{T}$) and for basic types they are defined by prescribing the property to be proved; while for higher order types they are defined using the relations of the domain and image types.

Definition 6.2 (Logical relations)**.** We define simultaneously two families of binary relations:

(a) If $\Gamma \vdash$ then $(\Gamma \vdash \_ \sim \_ \in \mathcal{T}) \subseteq \{A \mid \Gamma \vdash A\} \times \mathcal{T}$ shall be a Γ-indexed family of relations between well-formed syntactic types A and type values X.

(b) If $\Gamma \vdash A \sim X \in \mathcal{T}$ then $(\Gamma \vdash \_ : A \sim \_ \in [X]) \subseteq \{t \mid \Gamma \vdash t : A\} \times [X]$ shall be a (Γ, A, X)-indexed family of relations between terms t of type A and values d in PER $[X]$.

These relations are defined simultaneously by induction on $X \in \mathcal{T}$.

(1) Neutral types: $X \in \mathcal{N}e$.
    (a) $\Gamma \vdash A \sim X \in \mathcal{T}$ iff for all $\Delta \leqslant^i \Gamma$, $\Delta \vdash A\,\mathsf{p}^i = \mathsf{R}_{|\Delta|} \Downarrow X$.
    (b) $\Gamma \vdash t : A \sim d \in [X]$ iff $\Gamma \vdash A \sim X \in \mathcal{T}$, and for all $\Delta \leqslant^i \Gamma$, $\Delta \vdash t\,\mathsf{p}^i = \mathsf{R}_{|\Delta|} \downarrow_X d : A\,\mathsf{p}^i$.

(2) Universe.
    (a) $\Gamma \vdash A \sim \mathsf{U} \in \mathcal{T}$ iff $\Gamma \vdash A = \mathsf{U}$.

  (b) $\Gamma \vdash t : A \sim X \in [\mathsf{U}]$ iff $\Gamma \vdash A = \mathsf{U}$, and $\Gamma \vdash t \sim X \in \mathcal{T}$.

(3) Singletons.

  (a) $\Gamma \vdash A \sim \mathsf{Sing}\, d\, X \in \mathcal{T}$ iff $\Gamma \vdash A = \{a\}_{A'}$ and $\Gamma \vdash a : A' \sim d \in [X]$.

  (b) $\Gamma \vdash t : A \sim d' \in [\mathsf{Sing}\, d\, X]$ iff $\Gamma \vdash A = \{a\}_{A'}$ and $\Gamma \vdash t : A' \sim d \in [X]$, and $\Gamma \vdash A' \sim X \in \mathcal{T}$.

(4) Function spaces.

  (a) $\Gamma \vdash A \sim \mathsf{Fun}\, X\, F \in \mathcal{T}$ iff $\Gamma \vdash A = \mathsf{Fun}\, A'\, B$, and $\Gamma \vdash A' \sim X \in \mathcal{T}$, and $\Delta \vdash B\,(\mathsf{p}^i, s) \sim F\, d \in \mathcal{T}$ for all $\Delta \leqslant^i \Gamma$ and $\Delta \vdash s : A'\, \mathsf{p}^i \sim d \in [X]$.

  (b) $\Gamma \vdash t : A \sim f \in [\mathsf{Fun}\, X\, F]$ iff $\Gamma \vdash A = \mathsf{Fun}\, A'\, B$, $\Gamma \vdash A' \sim X$, and $\Delta \vdash \mathsf{app}\ (t\, \mathsf{p}^i)\ s : B\,(\mathsf{p}^i, s) \sim f \cdot d \in [F\, d]$ for all $\Delta \leqslant^i \Gamma$ and $\Delta \vdash s : A'\, \mathsf{p}^i \sim d \in [X]$.

The following technical lemmata show that the logical relations are preserved by judgmental equality, weakening of the judgement, and the equalities on the corresponding PERs. These lemmata are proved simultaneously for types and terms.

Lemma 6.3 (Closure under conversion)**.** *Let* $\Gamma \vdash A \sim X \in \mathcal{T}$ *and* $\Gamma \vdash A = A'$. *Then,*

(a) $\Gamma \vdash A' \sim X \in \mathcal{T}$, *and*

(b) *if* $\Gamma \vdash t : A \sim d \in [X]$ *and* $\Gamma \vdash t = t' : A$ *then* $\Gamma \vdash t' : A' \sim d \in [X]$.

Proof. By induction on $X \in \mathcal{T}$. See C.4. □

Lemma 6.4 (Monotonicity)**.** *Let* $\Delta \leqslant^i \Gamma$, *then*

(a) *if* $\Gamma \vdash A \sim X \in \mathcal{T}$, *then* $\Delta \vdash A\, \mathsf{p}^i \sim X \in \mathcal{T}$; *and*

(b) *if* $\Gamma \vdash t : A \sim d \in [X]$, *then* $\Delta \vdash t\, \mathsf{p}^i : A\, \mathsf{p}^i \sim d \in [X]$.

Proof. By induction on $X \in \mathcal{T}$. See C.5. □

Lemma 6.5 (Closure under PERs)**.** *Let* $\Gamma \vdash A \sim X \in \mathcal{T}$, *then*

(a) *if* $X = X' \in \mathcal{T}$, *then* $\Gamma \vdash A \sim X' \in \mathcal{T}$; *and*

(b) *if* $\Gamma \vdash t : A \sim d \in [X]$ *and* $d = d' \in [X]$, *then* $\Gamma \vdash t : A \sim d' \in [X]$.

Proof. By induction on $X = X' \in \mathcal{T}$. See C.6. □

The following lemma plays a key rôle in the proof of soundness. It proves that if a term is related to some element in (some PER), then it is convertible to the reification of the corresponding element in the PER of normal forms.

Lemma 6.6. *Let* $\Gamma \vdash A \sim X \in \mathcal{T}$. *Then,*

(a) $\Gamma \vdash A = \mathsf{R}_{|\Gamma|} \Downarrow X$,

(b) *if* $\Gamma \vdash t : A \sim d \in [X]$ *then* $\Gamma \vdash t = \mathsf{R}_{|\Gamma|} \downarrow_X d : A$; *and*

(c) *if* $k \in \mathcal{N}e$ *and for all* $\Delta \leqslant^i \Gamma$, $\Delta \vdash t\, \mathsf{p}^i = \mathsf{R}_{|\Delta|}\, k : A\, \mathsf{p}^i$, *then* $\Gamma \vdash t : A \sim \uparrow_X k \in [X]$.

Proof. By induction on $X \in \mathcal{T}$. See C.7. □

In order to finish the proof of soundness we have to prove that each well-typed term (and each well-formed type) is logically related to its denotation; with that aim we extend the definition of logical relations to substitutions and prove the fundamental theorem of logical relations.

Definition 6.7 (Logical relation for substitutions)**.**

(1) $\Gamma \vdash \sigma : \diamond \sim d \in \mathbf{1}$ always holds.

(2) $\Gamma \vdash (\sigma, t) : \Delta.A \sim (d, d') \in \coprod \mathcal{X}\,(d \mapsto [F\ d])$ iff $\Gamma \vdash \sigma : \Delta \sim d \in \mathcal{X}$, $\Gamma \vdash A\,\sigma \sim F\ d \in \mathcal{T}$, and $\Gamma \vdash t : A\,\sigma \sim d' \in [F\ d]$.

By the way this relation is defined, the counterparts of 6.3, 6.4, and 6.5 are easily proved by induction on the co-domain of the substitutions.

Remark 6.8. If $\Gamma \vdash \gamma = \delta : \Delta$, and $\Gamma \vdash \gamma : \Delta \sim d \in \mathcal{X}$, then $\Gamma \vdash \delta : \Delta \sim d \in \mathcal{X}$.

Remark 6.9. If $\Gamma \vdash \delta : \Delta \sim d \in \mathcal{X}$, then for any $\Theta \leqslant^i \Gamma$, $\Theta \vdash \delta\,\mathsf{p}^i : \Delta \sim d \in \mathcal{X}$.

Remark 6.10. If $\Gamma \vdash \gamma : \Delta \sim d \in \mathcal{X}$, and $d = d' \in \mathcal{X}$, then $\Gamma \vdash \gamma : \Delta \sim d' \in \mathcal{X}$.

Theorem 6.11 (Fundamental theorem of logical relations)**.** *Let* $\Delta \vdash \delta : \Gamma \sim d \in (\![\Gamma]\!)$.
(1) *If* $\Gamma \vdash A$, *then* $\Delta \vdash A\,\delta \sim [\![A]\!]d \in \mathcal{T}$;
(2) *if* $\Gamma \vdash t : A$, *then* $\Delta \vdash t\,\delta : A\,\delta \sim [\![t]\!]d \in [[\![A]\!]d]$; *and*
(3) *if* $\Gamma \vdash \gamma : \Theta$ *then* $\Delta \vdash \gamma\,\delta : \Theta \sim [\![\gamma]\!]d \in (\![\Theta]\!)$.

Proof. By mutual induction on the derivations. See C.9. □

We define for each context Γ an element ρ_Γ of D. This environment will be used to define the normalisation function.

Definition 6.12 (Canonical environment)**.** We define ρ_Γ by induction on Γ as follows:

$$\begin{aligned} \rho_\diamond &= \top \\ \rho_{\Gamma.A} &= (d', \uparrow_{[\![A]\!]d'} \mathsf{Var}\,x_n) \qquad\qquad \text{where } n = |\Gamma|, \text{ and } d' = \rho_\Gamma. \end{aligned}$$

By an immediate induction on contexts we can check the following.

Lemma 6.13. *If* $\Gamma \vdash$ *then* $\Gamma \vdash \mathsf{id}_\Gamma : \Gamma \sim \rho_\Gamma \in (\![\Gamma]\!)$.

Proof. By induction on $\Gamma \vdash$. See C.8 □

6.2. **Main results.** Now we can define concretely the normalisation function as the composition of reification with normalisation after evaluation under the canonical environment. The following corollaries just instantiate previous lemmata and theorems concluding correctness of NbE.

Definition 6.14 (Normalisation algorithm)**.** Let $\Gamma \vdash A$, and $\Gamma \vdash t : A$.

$$\begin{aligned} \mathbf{nbe}_\Gamma(A) &= \mathsf{R}_{|\Gamma|} \Downarrow [\![A]\!]\rho_\Gamma \\ \mathbf{nbe}^A_\Gamma(t) &= \mathsf{R}_{|\Gamma|} \downarrow_{[\![A]\!]\rho_\Gamma} [\![t]\!]\rho_\Gamma \end{aligned}$$

Notice that if we instantiate Thm. 6.11 with ρ_Γ, then a well-typed term t under Γ will be logically related to its denotation. Finally, using the key lemma 6.6 we conclude correctness for NbE.

Corollary 6.15. *Let* $\Gamma \vdash A$, *and* $\Gamma \vdash t : A$, *then by fundamental theorem of logical relations (and Lem. 6.3),*
(1) $\Gamma \vdash A \sim [\![A]\!]\rho_\Gamma \in \mathcal{T}$; *and*
(2) $\Gamma \vdash t : A \sim [\![t]\!]\rho_\Gamma \in [[\![A]\!]\rho_\Gamma]$,

Corollary 6.16 (Soundness of NbE)**.** *By way of Lem. 6.6, it follows immediately*

(1) $\Gamma \vdash A = \mathbf{nbe}(A)$, *and*
(2) $\Gamma \vdash t = \mathbf{nbe}(t) : A$.

We have now a decision procedure for judgmental equality; for deciding $\Gamma \vdash t = t' : A$, put both terms in normal formal and check if they are syntactically equal.

Corollary 6.17. *If $\Gamma \vdash A$, and $\Gamma \vdash A'$, then we can decide $\Gamma \vdash A = A'$. Also if $\Gamma \vdash t : A$, and $\Gamma \vdash t' : A$, we can decide $\Gamma \vdash t = t' : A$.*

As a byproduct we can conclude that type constructors are injective; this result is exploited in the next section where we introduce the type-checking algorithm. Injectivity of $\mathsf{Fun}\,\_\,\_$ plays a key rôle in all versions of dependent type theory with equality as judgement; *cf.* Adams' [7] proof of equivalence between PTS with equality as a judgement and equality taken as a relation between untyped terms, improved by Siles and Herbelin [52].

Remark 6.18. By expanding definitions, we easily check
(1) $\mathbf{nbe}_\Gamma(\mathsf{Fun}\,A\,B) = \mathsf{Fun}\,(\mathbf{nbe}_\Gamma(A))\,(\mathbf{nbe}_{\Gamma.A}(B))$, and
(2) $\mathbf{nbe}_\Gamma(\{a\}_A) = \{\mathbf{nbe}^A_\Gamma(a)\}_{\mathbf{nbe}_\Gamma(A)}$.

Corollary 6.19 (Injectivity of $\mathsf{Fun}\,\_\,\_$ and of $\{\_\}_\_$)**.** *If $\Gamma \vdash \mathsf{Fun}\,A\,B = \mathsf{Fun}\,A'\,B'$, then $\Gamma \vdash A = A'$, and $\Gamma.A \vdash B = B'$. Also $\Gamma \vdash \{t\}_A = \{t'\}_{A'}$, then $\Gamma \vdash A = A'$, and $\Gamma \vdash t = t' : A$.*

6.3. **Calculus $\lambda^{\mathsf{Irr}}$with proof irrelevance.** In this section we introduce the logical relations for the new types in λ^{Irr}. We skip the re-statement of the results given for λ^{Sing} in 6.1, instead we present in Appendix C the proof for some of the new cases arising in this calculus for each of lemmata 6.3, 6.4, 6.5, 6.6 and theorem 6.11.

Definition 6.20 (*cf.* 6.2)**.**
(1) Sigma types.
  (a) $\Gamma \vdash A \sim \mathsf{Sum}\,X\,F$ iff $\Gamma \vdash A = \Sigma\,A'\,B'$ and $\Gamma \vdash A' \sim X$ and for all $\Delta \leqslant^i \Gamma$ and $\Delta \vdash s : A'\,\mathsf{p}^i \sim d \in [X]$, $\Delta \vdash B'\,(\mathsf{p}^i, s) \sim F\,d$.
  (b) $\Gamma \vdash t : A \sim d \in [\mathsf{Sum}\,X\,F]$ iff $\Gamma \vdash A = \Sigma\,A'\,B'$ and $\Gamma \vdash \mathsf{fst}\ t : A' \sim \mathsf{fst}\,d \in [X]$ and $\Gamma \vdash \mathsf{snd}\ t : B'\,(\mathsf{id}_\Gamma, \mathsf{fst}\ t) \sim \mathsf{snd}\,d \in [F\,(\mathsf{fst}\,d)]$.
(2) Natural numbers.
  (a) $\Gamma \vdash A \sim \mathsf{Nat}$ iff $\Gamma \vdash A = \mathsf{Nat}$.
  (b) $\Gamma \vdash t : A \sim d \in [\mathsf{Nat}]$ iff $\Gamma \vdash A \sim \mathsf{Nat}$ and for all $\Delta \leqslant^i \Gamma$, $\Delta \vdash t\,\mathsf{p}^i = \mathsf{R}_{|\Delta|}\,d : \mathsf{Nat}$.
(3) Finite types.
  (a) $\Gamma \vdash A \sim \mathsf{N}_n$ iff $\Gamma \vdash A = \mathsf{N}_n$.
  (b) $\Gamma \vdash t : A \sim d \in [\mathsf{N}_n]$ iff $\Gamma \vdash A \sim \mathsf{N}_n$ and for all $\Delta \leqslant^i \Gamma$, $\Delta \vdash t\,\mathsf{p}^i = \mathsf{R}_{|\Delta|}\,d : \mathsf{N}_n$.
(4) Proof-irrelevance types.
  (a) $\Gamma \vdash A \sim \mathsf{Prf}\,X \in \mathcal{T}$ iff $\Gamma \vdash A = \mathsf{Prf}\,A'$ and $\Gamma \vdash A' \sim X \in \mathcal{T}$.
  (b) $\Gamma \vdash t : A \sim d \in [\mathsf{Prf}\,X]$ iff $\Gamma \vdash A \sim \mathsf{Prf}\,X$.

Remark 6.21.
(1) $\mathbf{nbe}_\Gamma(\Sigma\,A\,B) = \Sigma\,\mathbf{nbe}_\Gamma(A)\,\mathbf{nbe}_{\Gamma.A}(B)$;
(2) $\mathbf{nbe}^{\Sigma\,A\,B}_\Gamma((t,b)) = (\mathbf{nbe}^A_\Gamma(t), \mathbf{nbe}^{B\,(\mathsf{id},t)}_\Gamma(b))$;
(3) $\mathbf{nbe}(\mathsf{suc}\,t) = \mathsf{suc}\,\mathbf{nbe}(t)$.
(4) $\mathbf{nbe}(\mathsf{Prf}\,A) = \mathsf{Prf}\,\mathbf{nbe}(A)$.

Corollary 6.22. *If* $\Gamma \vdash \Sigma\, A\, B = \Sigma\, A'\, B'$*, then* $\Gamma \vdash A = A'$*, and* $\Gamma.A \vdash B = B'$*.*

7. Type-checking algorithm

In this section, we define a couple of judgements that represent a bidirectional type checking algorithm for terms in normal form; its implementation in Haskell can be found in the appendix. The algorithm is similar to previous ones [20, 4], in that it proceeds by analysing the possible types for each normal form, and succeeds only if the type's shape matches the one required by the introduction rule of the term. The only difference is introduced by the presence of singleton types; now we should take into account that a normal form can also have a singleton as its type.

This situation can be dealt in two possible ways; either one checks that the deepest tag of the normalised type (see Def. 7.2) has the form of the type of the introductory rule; or one adds a rule for checking any term against singleton types. The first approach requires to have more rules (this is due to the combination of singletons and a universe). We take the second approach, which requires to compute the eta-long normal form of the type before type-checking. We also note that the proof of completeness is more involved, because now the algorithm is not only driven by the term being checked, but also by the type.

Our algorithm depends on having a *good* normalisation function; note that this function does not need to be based on normalisation by evaluation. Also note that the second point asks for having correctness and completeness of the normalisation function.

Definition 7.1 (Good normalisation function)**.**
(1) $\mathbf{nbe}(\{a\}_A) = \{\mathbf{nbe}(a)\}_{\mathbf{nbe}(A)}$, and $\mathbf{nbe}(\mathsf{Fun}\, A\, B) = \mathsf{Fun}\, \mathbf{nbe}(A) \mathbf{nbe}(B)$;
(2) $\mathbf{nbe}_\Gamma(A) = \mathbf{nbe}_\Gamma(B)$ if and only if $\Gamma \vdash A = B$, and $\mathbf{nbe}^A_\Gamma(t) = \mathbf{nbe}^A_\Gamma(t')$, if and only if $\Gamma \vdash t = t' : A$.

From these properties we can prove the injectivity of Fun which is crucial for completeness of type checking λ-abstractions.

7.1. Type-checking λ^{Sing}.

In this section, let $V, V', W, v, v', w \in Nf$, and $k \in Ne$. For obtaining the deepest tag of a singleton type, we define an operation on types, which is essentially the same as the one defined by Aspinall [10].

Definition 7.2 (Singleton's tag)**.**

$$\overline{V} = \begin{cases} \overline{W} & \text{if } V \equiv \{w\}_W \\ V & \text{otherwise.} \end{cases}$$

The predicates for type-checking are defined mutually inductively, together with the function for inferring types.

Definition 7.3 (Type-checking and type-inference)**.** We define three mutually inductive algorithmic judgements

$\Gamma \vdash V \Leftarrow$	in context Γ, normal type V checks
$\Gamma \vdash v \Leftarrow V$	in context Γ, normal term v checks against type V
$\Gamma \vdash k \Rightarrow V$	in context Γ, the type of neutral term k is inferred as V.

All three judgements presuppose and maintain the invariant the input Γ is a well-formed context. The procedures $\Gamma \vdash V \Leftarrow$ and $\Gamma \vdash v \Leftarrow V$ expect their inputs V and v in β-normal form. Inference $\Gamma \vdash k \Rightarrow V$ expects a neutral term k and returns its principal type V in long normal form.

Well-formedness checking of types $\Gamma \vdash V \Leftarrow$.

$$\frac{}{\Gamma \vdash \mathsf{U} \Leftarrow} \qquad \frac{\Gamma \vdash V \Leftarrow \qquad \Gamma.V \vdash W \Leftarrow}{\Gamma \vdash \mathsf{Fun}\, V\, W \Leftarrow} \qquad \frac{\Gamma \vdash V \Leftarrow \qquad \Gamma \vdash v \Leftarrow \mathbf{nbe}(V)}{\Gamma \vdash \{v\}_V \Leftarrow} \qquad \frac{\Gamma \vdash k \Leftarrow \mathsf{U}}{\Gamma \vdash k \Leftarrow}$$

Type checking terms $\Gamma \vdash v \Leftarrow V$.

$$\frac{\Gamma \vdash V \Leftarrow \mathsf{U} \qquad \Gamma.V \vdash W \Leftarrow \mathsf{U}}{\Gamma \vdash \mathsf{Fun}\, V\, W \Leftarrow \mathsf{U}} \qquad \frac{\Gamma.V \vdash v \Leftarrow W}{\Gamma \vdash \lambda v \Leftarrow \mathsf{Fun}\, V\, W}$$

$$\frac{\Gamma \vdash V \Leftarrow \mathsf{U} \qquad \Gamma \vdash v \Leftarrow \mathbf{nbe}(V)}{\Gamma \vdash \{v\}_V \Leftarrow \mathsf{U}} \qquad \frac{\Gamma \vdash v \Leftarrow V' \qquad \Gamma \vdash v' = v : V'}{\Gamma \vdash v \Leftarrow \{v'\}_{V'}}$$

$$\frac{\Gamma \vdash k \Rightarrow V' \qquad \Gamma \vdash \overline{V'} = V}{\Gamma \vdash k \Leftarrow V}\ V \not\equiv \{w\}_W$$

Type inference $\Gamma \vdash k \Rightarrow V$.

$$\frac{}{\Gamma.A_i.\dots.A_0 \vdash \mathsf{v}_i \Rightarrow \mathbf{nbe}(A_i\, \mathsf{p}^{i+1})} \qquad \frac{\Gamma \vdash k \Rightarrow V \qquad \Gamma \vdash \overline{V} = \mathsf{Fun}\, V'\, W \qquad \Gamma \vdash v \Leftarrow V'}{\Gamma \vdash \mathsf{app}\ k\ v \Rightarrow \mathbf{nbe}(W\,(\mathsf{id}, v))}$$

Bidirectional type checking for dependent function types is well-understood [20, 35]; let us illustrate briefly how it works for singleton types, by considering the type checking problem $\{\mathsf{zero}\}_{\mathsf{Nat}} \vdash \mathsf{q} \Leftarrow \{\mathsf{zero}\}_{\mathsf{Nat}}$. Here is a skeletal derivation of this judgement, which is at the same time an execution trace of the type checker:

$$\frac{\dfrac{\dfrac{}{\{\mathsf{zero}\}_{\mathsf{Nat}} \vdash \mathsf{q} \Rightarrow \{\mathsf{zero}\}_{\mathsf{Nat}}} \qquad \{\mathsf{zero}\}_{\mathsf{Nat}} \vdash \overline{\{\mathsf{zero}\}_{\mathsf{Nat}}} = \mathsf{Nat}}{\{\mathsf{zero}\}_{\mathsf{Nat}} \vdash \mathsf{q} \Leftarrow \mathsf{Nat}} \qquad \{\mathsf{zero}\}_{\mathsf{Nat}} \vdash \mathsf{q} = \mathsf{zero} : \mathsf{Nat}}{\{\mathsf{zero}\}_{\mathsf{Nat}} \vdash \mathsf{q} \Leftarrow \{\mathsf{zero}\}_{\mathsf{Nat}}}$$

Since the type to check against is a singleton, the algorithm proceeds by checking $\{\mathsf{zero}\}_{\mathsf{Nat}} \vdash \mathsf{q} \Leftarrow \mathsf{Nat}$ and $\{\mathsf{zero}\}_{\mathsf{Nat}} \vdash \mathsf{q} = \mathsf{zero} : \mathsf{Nat}$. Now the type of the neutral q is inferred and its tag compared to the given type Nat; as the tag is also Nat, the check succeeds. The remaining equation $\{\mathsf{zero}\}_{\mathsf{Nat}} \vdash \mathsf{q} = \mathsf{zero} : \mathsf{Nat}$ is derivable by (SING-EQ-EL). Of course, the equations are checked by the $\mathbf{nbe}(\_)$ function; for example, by using our own function for normalisation we have $\mathbf{nbe}^{\mathsf{Nat}}_{\{\mathsf{zero}\}_{\mathsf{Nat}}}(\mathsf{q}) = \mathsf{zero} = \mathbf{nbe}^{\mathsf{Nat}}_{\{\mathsf{zero}\}_{\mathsf{Nat}}}(\mathsf{zero})$.

Theorem 7.4 (Correctness of type-checking)**.**
(1) *If* $\Gamma \vdash V \Leftarrow$*, then* $\Gamma \vdash V$*.*
(2) *If* $\Gamma \vdash v \Leftarrow V$*, then* $\Gamma \vdash v : V$*.*
(3) *If* $\Gamma \vdash k \Rightarrow V$*, then* $\Gamma \vdash k : V$*.*

Proof. By simultaneous induction on $\Gamma \vdash V \Leftarrow$, $\Gamma \vdash v \Leftarrow V$, and $\Gamma \vdash V \Rightarrow k$. See C.10. $\square$

In order to prove completeness we define a lexicographic order on pairs of terms and types, in this way we can make induction over the term, and the type.

Definition 7.5. Let $v, v' \in \mathit{Nf}$, and $A, A' \in \mathsf{Type}(\Gamma)$, then $(v, A) \prec (v', A')$ is the lexicographic order on $\mathit{Nf} \times \mathsf{Type}(\Gamma)$. The corresponding orders are $v \prec v'$ iff v is an immediate sub-term of v'; and $A \prec^{\Gamma} A'$, iff $\mathbf{nbe}(A') \equiv \{w\}_{\mathbf{nbe}(A)}$.

Theorem 7.6 (Completeness of type-checking)**.**
(1) *If* $\Gamma \vdash V$*, then* $\Gamma \vdash V \Leftarrow$.
(2) *If* $\Gamma \vdash v : A$*, then* $\Gamma \vdash v \Leftarrow \mathbf{nbe}(A)$.
(3) *If* $\Gamma \vdash k : A$*, and* $\Gamma \vdash k \Rightarrow V'$*, then* $\Gamma \vdash \overline{\mathbf{nbe}(A)} = \overline{V'}$.

Proof. We prove these three statements simultaneously by well-founded induction on the order $\prec$. The respective measures are (1) (V, U), (2) (v, A), and (3) (k, A). Details are in the Appendix C.11. □

7.2. **Calculus $\lambda^{\mathsf{Irr}}$with proof irrelevance.** We give additional rules for type-checking and type-inference algorithms for the constructs added in Sect. 2.2. Remember that we distinguished two calculi: the calculus ($\vdash^{\star}$) has rules (N$_0$-TM) and (PRF-TM); while ($\vdash$) lacks those rules.

Definition 7.7 (Type-checking and type-inference)**.** Σ-types.

$$\frac{\Gamma \vdash V \Leftarrow \qquad \Gamma.V \vdash W \Leftarrow}{\Gamma \vdash \Sigma\, V\, W \Leftarrow} \qquad \frac{\Gamma \vdash V \Leftarrow \mathsf{U} \qquad \Gamma.V \vdash W \Leftarrow \mathsf{U}}{\Gamma \vdash \Sigma\, V\, W \Leftarrow \mathsf{U}}$$

$$\frac{\Gamma \vdash v \Leftarrow V \qquad \Gamma \vdash v' \Leftarrow \mathbf{nbe}(W\,(\mathsf{id}_{\Gamma}, v))}{\Gamma \vdash (v, v') \Leftarrow \Sigma\, V\, W}$$

$$\frac{\Gamma \vdash k \Rightarrow \Sigma\, V\, W}{\Gamma \vdash \mathsf{fst}\ k \Rightarrow V} \qquad \frac{\Gamma \vdash k \Rightarrow \Sigma\, V\, W}{\Gamma \vdash \mathsf{snd}\ k \Rightarrow \mathbf{nbe}(W\,(\mathsf{id}_{\Gamma}, \mathsf{fst}\ k))}$$

Natural numbers.

$$\frac{}{\Gamma \vdash \mathsf{Nat} \Leftarrow} \qquad \frac{}{\Gamma \vdash \mathsf{Nat} \Leftarrow \mathsf{U}} \qquad \frac{}{\Gamma \vdash \mathsf{zero} \Leftarrow \mathsf{Nat}} \qquad \frac{\Gamma \vdash v \Leftarrow \mathsf{Nat}}{\Gamma \vdash \mathsf{suc}\ v \Leftarrow \mathsf{Nat}}$$

$$\frac{\begin{array}{c}\Gamma.\mathsf{Nat} \vdash V \Leftarrow \qquad \Gamma \vdash k \Rightarrow \mathsf{Nat} \\ \Gamma \vdash v \Leftarrow \mathbf{nbe}(V\,(\mathsf{id}_{\Gamma}, \mathsf{zero})) \qquad \Gamma \vdash v' \Leftarrow \mathsf{Fun}\,\mathsf{Nat}\,(\mathsf{Fun}\,V\,\mathbf{nbe}(V\,(\mathsf{p}\,\mathsf{p}, \mathsf{suc}\ (\mathsf{q}\,\mathsf{p}))))\end{array}}{\Gamma \vdash \mathsf{natrec}\ V\ v\ v'\ k \Rightarrow \mathbf{nbe}(V\,(\mathsf{id}, k))}$$

Finite types.

$$\frac{}{\Gamma \vdash \mathsf{N}_n \Leftarrow} \qquad \frac{}{\Gamma \vdash \mathsf{N}_n \Leftarrow \mathsf{U}} \qquad \frac{i < n}{\Gamma \vdash \mathsf{c}^n_i \Leftarrow \mathsf{N}_n}$$

$$\frac{\Gamma.\mathsf{N}_n \vdash V \Leftarrow \qquad \Gamma \vdash k \Rightarrow \mathsf{N}_n \qquad \Gamma \vdash v_i \Leftarrow \mathbf{nbe}(V\,(\mathsf{id}_{\Gamma}, \mathsf{c}^n_i))}{\Gamma \vdash \mathsf{case}^n\ V\ v_0 \cdots v_{n-1}\ k \Rightarrow \mathbf{nbe}(V\,(\mathsf{id}, k))}$$

Proof types.

$$\frac{\Gamma \vdash V \Leftarrow}{\Gamma \vdash \mathsf{Prf}\, V \Leftarrow} \qquad \frac{\Gamma \vdash V \Leftarrow \mathsf{U}}{\Gamma \vdash \mathsf{Prf}\, V \Leftarrow \mathsf{U}} \qquad \frac{\Gamma \vdash v \Leftarrow V}{\Gamma \vdash [v] \Leftarrow \mathsf{Prf}\, V}$$

$$\frac{\Gamma \vdash W \Leftarrow \qquad \Gamma \vdash k \Rightarrow \mathsf{Prf}\, V \qquad \Gamma.V \vdash v \Leftarrow \mathbf{nbe}(W\, \mathsf{p}) \qquad \Gamma.V.V\mathsf{p} \vdash v\mathsf{p} = v(\mathsf{pp}, \mathsf{q}) : W\mathsf{pp}}{\Gamma \vdash v\, \mathsf{where}^{W}\, k \Rightarrow W}$$

We do not show the proof for correctness, because nothing is to be gained from it; suffice it to say that we can prove correctness with respect to $(\vdash^{\star})$.

Theorem 7.8. *The type-checking algorithm is sound with respect to the calculus* $\vdash^{\star}$*.*

Proof. By simultaneous induction on the derivability of the type-checking judgements. □

It is clear that the given rules are not complete for checking $(\vdash^{\star})$, because there is no rule for checking $\Gamma \vdash \star \Leftarrow A$. Note that it is not possible to have a sound and complete type-checking algorithm with respect to $(\vdash^{\star})$, for it would imply the decidability of type-inhabitation. Since type checking happens always *before* normalisation, we can still use a good normalisation function with respect to the calculus $(\vdash^{\star})$ for normalising types or deciding equality. Indeed, if the term to type-check does not contain $\star$, the need of checking $\Gamma \vdash \star \Leftarrow V$ will never arise; this is clearly seen by verifying that only sub-terms are type-checked in the premises.

Theorem 7.9. *The type-checking algorithm is complete with respect to the calculus* $(\vdash)$*.*

Proof. By simultaneous induction on the normal form of types and terms, using inversion on the typing judgement and correctness of $\mathbf{nbe}(\_)$. □

Corollary 7.10. *The type-checking algorithm is correct (by Thm. 7.8 and Cor. 2.15) and complete with respect to the calculus* $(\vdash)$*.*

8. Conclusion

The main contributions of the paper are the definition of a correct and complete type-checking algorithm, and a simpler solution to the problem of generating fresh identifiers in the NbE algorithm for a calculus with singletons, one universe, and proof-irrelevant types. The type-checker is based on the NbE algorithm which is used to decide equality and to prove the injectivity of the type constructors. We emphasise that the type-checking algorithm is modular with respect to the normalisation algorithm. All the results can be extended to a calculus with annotated lambda abstractions, yielding a type-checking algorithm for terms not necessarily in normal forms. The NbE algorithm can be implemented fairly easily in Haskell (*cf.* Appendix A), but the correctness of the implementation depends on proving the computational adequacy of the domain semantics with respect to Haskell's operational semantics. We have not developed this proof in this article and leave it for to future work.

8.1. **Related and Further Work on Singleton Types.** Singleton types are used to model the SML module system and records with manifest fields [21].

Aspinall [10] presents a logical framework with singleton types and subtyping and shows its consistency via a PER model, yet not decidability. The second author, Pollack, and Takeyama [21] extend the Aspinall's framework by η-equality and records and a type checking algorithm which is correct wrt. the PER model. This work is unconventional since there is no complete syntactical specification of the LF in terms of syntax, typing and equality rules. Instead, in the style of Martin-Löf meaning explanations, they list a number of inference rules which are valid in the semantics and prove that type-checked expressions evaluate to values of the correct semantic type.

Courant [22] shows strong normalization for a variant of Aspinall's system with equality defined by reduction. He uses a typed Kripke model of strongly normalizing terms, a variant of Goguen's typed operational semantics [29].

Stone and Harper [54] extend Aspinall's framework by sigma types and eta-equality, which allows them to reduce singletons at higher types to singletons at base type. Their decision procedure is type-directed, its completeness is shown via a Kripke model. Crary [23] gives a simplified decision procedure via hereditary substitutions and proves its correctness in Twelf, without the need for a model construction. His purely syntactical approach does not scale to universes, since he cannot handle types defined by recursion. Goguen [30] follows a similar agenda, he shows decidability for singleton types in the presence of eta by an eta-expanding translation into a logical framework with beta-equality only. He works with fully annotated terms in the sense of Streicher [55]. He stresses that his approach does not scale to computation on the type level.

In the continuation of this work we want to investigate whether our type-checking algorithm can be simplified if we implement Stone and Harper's insight that singleton types at higher types can be defined in terms of singleton base types. Further, we would like to integrate subtyping in our calculus, which should not be too difficult, since the PER model already supports subtyping [10, 21].

8.2. **Related and Further Work on Proof Irrelevance.** Pfenning [47] presents a logical framework with proof irrelevance that supports irrelevant function arguments, with function introduction rule (writing $(x\!:\!\mathsf{Prf}\,A) \to B$ in our syntax):

$$\frac{\Gamma, x \div A \vdash B \qquad \Gamma, x \div A \vdash t : B}{\Gamma \vdash \lambda x t : (x\!:\!\mathsf{Prf}\,A) \to B}$$

He proves decidability using erasure, mentioning that his technique does not scale to universes. Elimination of irrelevance is implicitly handled by annotating variables to ensure proof variables $(x \div A)$ appear only in proofs, in contrast to our explicit use of $\_\ \mathsf{where}\ [\_] \leftarrow \_$ in the style of Awodey and Bauer [11]. However, we believe that Pfenning's proof irrelevance can be modeled via bracket types $\mathsf{Prf}\,A$, with the weaker "monadic" rule for where (see section 2).

Barras and Bernardo's [13] presentation of proof irrelevant functions

$$\frac{\Gamma, x : A \vdash B \qquad \Gamma, x : A \vdash t : B \qquad x \notin \mathsf{FV}(t^*)}{\Gamma \vdash \lambda x t : (x\!:\!\mathsf{Prf}\,A) \to B}$$

diverges from Pfenning's that they allow irrelevant variables x to be relevant in types B. (In t the variable x might only appear irrelevantly, expressed by the side condition that x

may not be free in the relevant parts t^* of t.) Barras and Bernardo justify their calculus by erasing into Miquel's Implicit Calculus of Constructions (ICC) [41]. The ICC style irrelevance seems more expressive than Awodey and Bauer's or Pfenning's, but the exact relationship is unclear to us.

Berger's Uniform Heyting Algebra [15] features uniform quantification $\{\forall x\}A$ (and $\{\exists x\}A$) to obtain optimized programs by extraction from proofs. A proof of a uniform universal

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \{\forall\}^+(\lambda x M) : \{\forall x\}A}$$

may not mention term variable x in a computational relevant position. Since the shape of formulas does not depend on terms, Berger's calculus can be seen as logical counterpart of either Pfenning's or Bruno and Bernardo's type system.

We see two interesting questions about the different approaches to proof irrelevance above:

(1) How can Barras and Bernardo's ICC* be understood in terms of judgmental equality à la Pfenning?
(2) How can ICC* and the calculus of Pfenning be extended to full bracket types à la Awodey and Bauer without explicit use of where.

Acknowledgments. The authors have led many discussions with Peter Dybjer on normalization by evaluation, generalized algebraic theories, and presentation of type theory as categories with families. The authors thank the referees of a previous version of this article for their helpful comments. The authors are also grateful for the suggestions made by the anonymous referees on this long version.

References

[1] M. Abadi, L. Cardelli, P. L. Curien, and J. J. Lévy. Explicit substitutions. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 31–46, New York, NY, USA, 1990. ACM.

[2] 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.

[3] 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.

[4] Andreas Abel, Thierry Coquand, and Peter Dybjer. On the algebraic foundation of proof assistants for intuitionistic type theory. In Jacques Garrigue and Manuel V. Hermenegildo, editors, *Proc. of the 9th Int. Symp. on Functional and Logic Programming, FLOPS 2008*, volume 4989 of *Lect. Notes in Comput. Sci.*, pages 3–13. Springer, 2008.

[5] Andreas Abel, Thierry Coquand, and Peter Dybjer. Verifying a semantic $\beta\eta$-conversion test for Martin-Löf type theory. In Philippe Audebaud and Christine Paulin-Mohring, editors, *Proc. of the 9th Int. Conf. on Mathematics of Program Construction, MPC 2008*, volume 5133 of *Lect. Notes in Comput. Sci.*, pages 29–56. Springer, 2008.

[6] Samson Abramsky and A Jung. *Handbook of Logic in Computer Science*, chapter Domain Theory, pages 1–168. Oxford University Press, 1994.

[7] Robin Adams. Pure type systems with judgemental equality. *Journal Functional Programming*, 16(2):219–246, 2006.

[8] Klaus Aehlig and Felix Joachimski. Operational aspects of untyped normalization by evaluation. *Math. Struct. in Comput. Sci.*, 14(4):587–611, August 2004.

[9] Thorsten Altenkirch, Peter Dybjer, Martin Hofmann, and Philip J. Scott. Normalization by evaluation for typed lambda calculus with coproducts. In *Proc. of the 16th IEEE Symp. on Logic in Computer Science (LICS 2001)*, pages 303–310. IEEE Computer Soc. Press, 2001.

[10] David Aspinall. Subtyping with singleton types. In Leszek Pacholski and Jerzy Tiuryn, editors, *Computer Science Logic, 8th Int. Wksh., CSL '94*, volume 933 of *Lect. Notes in Comput. Sci.*, pages 1–15. Springer, 1995.

[11] Steven Awodey and Andrej Bauer. Propositions as [Types]. *J. Log. Comput.*, 14(4):447–471, 2004.

[12] Vincent Balat, Roberto Di Cosmo, and Marcelo P. Fiore. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. In Neil D. Jones and Xavier Leroy, editors, *Proc. of the 31st ACM Symp. on Principles of Programming Languages, POPL 2004*, pages 64–76. ACM Press, 2004.

[13] Bruno Barras and Bruno Bernardo. The implicit calculus of constructions as a programming language with dependent types. In *FoSSaCS*, pages 365–379, 2008.

[14] Stefano Berardi. About the sets-as-propositions embedding of HOL in CC. February 2004.

[15] Ulrich Berger. Uniform Heyting arithmetic. *Ann. Pure Appl. Logic*, 133(1–3):125–148, 2005.

[16] Ulrich Berger and Helmut Schwichtenberg. An inverse to the evaluation functional for typed λ-calculus. In *Proc. of the 6th IEEE Symp. on Logic in Computer Science (LICS'91)*, pages 203–211. IEEE Computer Soc. Press, 1991.

[17] Kim Bruce and John C. Mitchell. Per models of subtyping, recursive types and higher-order polymorphism. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 316–327, New York, NY, USA, 1992. ACM.

[18] N. G. de Bruijn. Some extensions of Automath : the AUT-4 family. 1994.

[19] J. Cartmell. Generalised algebraic theories and contextual categories. *Annals of Pure and Applied Logic*, pages 32–209, 1986.

[20] Thierry Coquand. An algorithm for type-checking dependent types. In *Proc. of the 3rd Int. Conf. on Mathematics of Program Construction, MPC '95*, volume 26 of *Sci. Comput. Program.*, pages 167–177. Elsevier, May 1996.

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

[22] Judicaël Courant. Strong normalization with singleton types. In *Intersection Types and Related Systems (ITRS 2002)*, volume 70 of *Electr. Notes in Theor. Comp. Sci.* Elsevier, 2002.

[23] Karl Crary. A syntactic account of singleton types via hereditary substitution. In James Cheney and Amy Felty, editors, *4th Int. Wksh. on Logical Frameworks and Meta-languages: Theory and Practice (LFMTP 2009)*, pages 21–29. ACM Press, 2009.

[24] Peter Dybjer. Internal type theory. In Stefano Berardi and Mario Coppo, editors, *Types for Proofs and Programs, Int. Wksh., TYPES'95*, volume 1158 of *Lect. Notes in Comput. Sci.*, pages 120–134. Springer, 1996.

[25] Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *J. Symb. Logic*, 65(2):525–549, 2000.

[26] Andrzej Filinski and Henning Korsholm Rohde. A denotational account of untyped normalization by evaluation. In Igor Walukiewicz, editor, *Proc. of the 7th Int. Conf. on Foundations of Software Science and Computational Structures, FoSSaCS 2004*, volume 2987 of *Lect. Notes in Comput. Sci.*, pages 167–181. Springer, 2004.

[27] François Garillot and Benjamin Werner. Simple types in type theory: Deep and shallow encodings. In Klaus Schneider and Jens Brandt, editors, *Theorem Proving in Higher Order Logics, TPHOLs 2007*, volume 4732 of *Lect. Notes in Comput. Sci.*, pages 368–382. Springer, 2007.

[28] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoret. Comput. Sci.* Cambridge University Press, 1989.

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

[30] Healfdene Goguen. A syntactic approach to eta equality in type theory. In Jens Palsberg and Martín Abadi, editors, *Proc. of the 32nd ACM Symp. on Principles of Programming Languages, POPL 2005*, pages 75–84. ACM Press, January 2005.

[31] 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.

[32] Robert Harper, Furio Honsell, and Gordon Plotkin. A Framework for Defining Logics. *Journal of the Association of Computing Machinery*, 40(1):143–184, January 1993.

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

[34] Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of Standard ML. In Martin Hofmann and Matthias Felleisen, editors, *Proc. of the 34th ACM Symp. on Principles of Programming Languages, POPL 2007*, pages 173–184. ACM Press, 2007.

[35] Andreas Löh, Conor McBride, and Wouter Swierstra. A tutorial implementation of a dependently typed lambda calculus. *Fundam. Inform.*, 102(2):177–207, 2010.

[36] Odalric-Ambrym Maillard. Proof-irrelevance, strong-normalisation in Type-Theory and PER. Technical report, Chalmers Institute of Technology, 2006.

[37] Per Martin-Löf. An Intuitionistic Theory of Types. Technical report, 1972.

[38] Per Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, 1984.

[39] Per Martin-Löf. Normalization by evaluation and by the method of computability. Talk at JAIST, Japan Advanced Institute of Science and Technology, Kanazawa, June 2004.

[40] Conor McBride. Epigram: Practical programming with dependent types. In Varmo Vene and Tarmo Uustalu, editors, *5th Int. School on Advanced Functional Programming, AFP 2004, Revised Lectures*, volume 3622 of *Lect. Notes in Comput. Sci.*, pages 130–170. Springer, 2005.

[41] Alexandre Miquel. The implicit calculus of constructions. In Samson Abramsky, editor, *Proc. of the 5th Int. Conf. on Typed Lambda Calculi and Applications, TLCA 2001*, volume 2044 of *Lect. Notes in Comput. Sci.*, pages 344–359. Springer, 2001.

[42] John C. Mitchell. A type-inference approach to reduction properties and semantics of polymorphic expressions (summary). In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 308–319, New York, NY, USA, 1986. ACM.

[43] John C. Mitchell. Type systems for programming languages. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 365–458. 1990.

[44] E. Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in computer science*, pages 14–23, Piscataway, NJ, USA, 1989. IEEE Press.

[45] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin Löf's Type Theory: An Introduction.* Clarendon Press, Oxford, 1990.

[46] Ulf Norell. *Towards a Practical Programming Language Based on Dependent Type Theory.* PhD thesis, Dept of Comput. Sci. and Engrg., Chalmers, Göteborg, Sweden, September 2007.

[47] Frank Pfenning. Intensionality, extensionality, and proof irrelevance in modal type theory. In *LICS 2001: IEEE Symposium on Logic in Computer Science*, June 2001.

[48] Gordon Plotkin. LCF considered as a programming language. *Theor. Comput. Sci.*, 5:223–255, 1977.

[49] Robert Pollack. Closure under alpha-conversion. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs (TYPES'93), Nijmegen, The Netherlands*, volume 806 of *Lect. Notes in Comput. Sci.*, pages 313–332. Springer, 1994.

[50] Giovanni Sambin and Silvio Valentini. Building up a toolbox for Martin-Löf's type theory: subset theory. In *Twenty-five years of constructive type theory (Venice, 1995)*, volume 36, chapter Oxford Logic Guides, pages 221–244. Oxford University Press, New York, 1998.

[51] Natarajan Shankar and Sam Owre. Principles and Pragmatics of Subtyping in PVS. In *WADT '99: Selected papers from the 14th International Workshop on Recent Trends in Algebraic Development Techniques*, pages 37–52, London, UK, 2000. Springer-Verlag.

[52] Vincent Siles and Hugo Herbelin. Equality is typable in semi-full pure type systems. In *LICS*, pages 21–30, 2010.

[53] Matthieu Sozeau. Subset coercions in Coq. In Thorsten Altenkirch and Conor McBride, editors, *Types for Proofs and Programs, Int. Wksh., TYPES 2006*, volume 4502 of *Lect. Notes in Comput. Sci.*, pages 237–252. Springer, 2007.

[54] Christopher A. Stone and Robert Harper. Extensional equivalence and singleton types. *ACM Trans. Comput. Logic*, 7(4):676–722, 2006.

[55] Thomas Streicher. *Semantics of Type Theory.* Progress in Theoretical Computer Science. Birkhäuser Verlag, Basel, 1991.

[56] Benjamin Werner. On the strength of proof-irrelevant type theories. *Logical Meth. in Comput. Sci.*, 4, 2008.

Appendix A. Normalisation by evaluation

```
type Type  = Term
data Term = U                                  -- universe
          | Fun Type Type                      -- dependent function space
          | Singl Term Type                    -- singleton type ({a}_A)
          | App Term Term                      -- application
          | Lam Term                           -- abstraction
          | Q                                  -- variable
          | Sub Term Subst                     -- substitution
          | Sigma Type Type                    -- dependent pair type
          | Fst Term                           -- first projection
          | Snd Term                           -- second projection
          | Pair Term Term                     -- dependent pair
          | Nat                                -- naturals
          | Zero                               -- 0
          | Suc Term                           -- +1
          | Natrec Type Term Term Term         -- elimination for Nat
          | Prf Type                           -- proof (with proof irrelevance)
          | Box Term                           -- a term in Prf A
          | Star                               -- canonical element of Prf A
          | Where Type Term Term               -- Box elimination
          | Enum Int                           -- Enum n has n elements
          | Const Int Int                      -- Const n i is the ith element
          | Case Int Type [Term] Term          -- elimination for Enum n
            deriving (Eq, Show)
data Subst = E                                 -- empty substitution
           | Is                                -- identity substitution
           | Ext Subst Term                    -- extension
           | P                                 -- weakening
           | Comp Subst Subst                  -- composition
             deriving (Eq, Show)
type DT    = D                                 -- semantic types
data D     = T                                 -- terminal object (empty context)
           | Ld (D → D)                        -- function
           | FunD DT (D → DT)                  -- dependent function type
           | UD                                -- universe
           | SingD D DT                        -- singleton type
           | Vd Int                            -- free variable
           | AppD D D                          -- neutral application
```

```
              | SumD DT (D → DT)             -- dependent pair type
              | PairD D D                    -- context comprehension
              | FstD D                       -- first projection of neutral
              | SndD D                       -- second projection of neutral
              | NatD                         -- natural number type
              | ZeroD                        -- 0
              | SucD D                       -- +1
              | NatrecD (D → DT) D D D       -- recursion on neutrals
              | PrfD DT                      -- proof type
              | StarD                        -- don't care
              | EnumD Int                    -- enumeration type
              | ConstD Int Int               -- constants in EnumD
              | CaseD Int (D → DT) [D] D     -- elimination on neutrals
type Ctx   = [Type]

pi1, pi2 ::   D → D
pi1 (PairD d d')   = d
pi1 StarD          = StarD
pi1 k              = FstD k
pi2 (PairD d d')   = d'
pi2 StarD          = StarD
pi2 k              = SndD k

ap :: D → D → D
ap (Ld f) d = f d
ap StarD _ = StarD

neutralD :: D → Bool
neutralD (Vd _)             = True
neutralD (AppD _ _)         = True
neutralD (FstD _)           = True
neutralD (SndD _)           = True
neutralD (NatrecD _ _ _ _)  = True
neutralD (CaseD _ _ _ _)    = True
neutralD StarD              = True
neutralD _                  = False

natrec :: (D → DT) → D → D → D → D
natrec b z s StarD              = StarD
natrec b z s ZeroD              = z
natrec b z s (SucD e)           = (s `ap` e) `ap` (natrec b z s e)
natrec b z s d | neutralD d     = up (b d)
                                     (NatrecD (λe → downT (b e))
                                              (down (b ZeroD) z)
                                              downSuc
                                              d)
```

```
        where downSuc = down (FunD NatD
                                       (λn → FunD (b n)
                                       (λe → b (SucD n))))
                             s
downs :: Int → (D → DT) → [D] → Int → [D]
downs _ _ []       _ = []
downs n f (d : ds) i = down (f (ConstD n i)) d : downs n f ds (i + 1)
constD :: Int → Int → D → Bool
constD n i (ConstD m j) = m ≡ n ∧ i ≡ j
constD _ _ _            = False
caseD :: Int → (D → DT) → [D] → D → D
caseD n b ds StarD                                      = StarD
caseD n b ds (ConstD m i) | n ≡ m ∧ i < n               = ds !! i
caseD n b ds d | neutralD d ∧
                 and [constD n i (ds !! i) | i ← [0 . . n − 1]] = up (b d) d
caseD n b ds d | neutralD d                             = up (b d)
                             (CaseD n (λe → downT (b e))
                                      (downs n b ds 0)
                                      d)
up :: DT → D → D
up (SingD a x)   k = a
up (FunD a f)    k = Ld (λd → up (f d) (AppD k (down a d)))
up (SumD a f)    k = PairD (up a (FstD k))
                           (up (f (up a (FstD k))) (SndD k))
up (PrfD a)      k = StarD
up (EnumD 0)     k = StarD
up (EnumD 1)     k = ConstD 1 0
up d             k = k
down :: DT → D → D
down UD          d = downT d
down (SingD a x) d = down x a
down (FunD a f)  d = Ld (λe → down (f (up a e)) (d ‘ap‘ (up a e)))
down (SumD a b)  d = PairD (down a (pi1 a)) (down (b (pi1 d)) (pi2 d))
down (PrfD a)    d = StarD
down (EnumD 1)   d = ConstD 1 0
down d           e = e
downT :: DT → DT
downT (SingD a x) = SingD (down x a) (downT x)
downT (FunD a f)  = FunD  (downT a) (λd → downT (f (up a d)))
downT (SumD a b)  = SumD  (downT a) (λd → downT (b (up a d)))
downT (PrfD a)    = PrfD  (downT a)
downT d           = d
```

```
readback :: Int → D → Term
readback i UD                    = U
readback i (FunD a f)            = Fun     (readback i a) (readback (i + 1) (f (Vd i)))
readback i (SingD a x)           = Singl   (readback i a) (readback i x)
readback i (Ld f)                = Lam     (readback (i + 1) (f (Vd i)))
readback i (Vd n)                = mkvar   (i − n − 1)
readback i (AppD k d)            = App     (readback i k) (readback i d)
readback i (FstD d)              = Fst     (readback i d)
readback i (SndD d)              = Snd     (readback i d)
readback i (PairD d e)           = Pair    (readback i d) (readback i e)
readback i (SumD a b)            = Sigma   (readback i a) (readback (i + 1) (b (Vd i)))
readback i NatD                  = Nat
readback i ZeroD                 = Zero
readback i (SucD e)              = Suc     (readback i e)
readback i (NatrecD b z s e)     = Natrec  (Fun Nat (readback (i + 1) (b (Vd i))))
                                           (readback i z)
                                           (readback i s)
                                           (readback i e)
readback i (PrfD d)             = Prf      (readback i d)
readback i StarD                = Star
readback i (EnumD n)            = Enum n
readback i (ConstD n j)         = Const n  j
readback i (CaseD n b ds d)     = Case n   (readback (i + 1) (b (Vd i)))
                                           (map (readback i) ds)
                                           (readback i d)
  -- Evaluation
type Env = D
eval :: Term → Env → D
eval U              d = UD
eval (Fun t f)      d = FunD (eval t d) (λd′ → eval f (PairD d d′))
eval (Singl t a)    d = SingD (eval t d) (eval a d)
eval (Lam t)        d = Ld (λd′ → eval t (PairD d d′))
eval (App t r)      d = (eval t d) ‘ap‘ (eval r d)
eval Q              d = pi2 d
eval (Sub t s)      d = eval t (evalS s d)
eval (Sigma t r)    d = SumD (eval t d) (λe → eval r (PairD d e))
eval (Fst t)        d = pi1 (eval t d)
eval (Snd t)        d = pi2 (eval t d)
eval (Pair t r)     d = PairD (eval t d) (eval r d)
eval Nat            d = NatD
eval Zero           d = ZeroD
eval (Suc t)        d = SucD (eval t d)
eval (Natrec b z s t) d = natrec (λe → eval b (PairD d e))
                                 (eval z d)
                                 (eval s d)
                                 (eval t d)
eval (Prf t)          d = PrfD (eval t d)
eval (Box t)          d = StarD
eval Star             d = StarD
eval (Where t b p)    d = eval b (PairD d StarD)
eval (Enum n)         d = EnumD n
eval (Const n i)      d = ConstD n i
eval (Case n b ts t)  d = caseD n   (λe → eval b (PairD d e))
                                    (map ((flip eval) d) ts)
                                    (eval t d)
evalS :: Subst → Env → Env
evalS E            d = T
evalS Is           d = d
evalS (Ext s t)    d = PairD (evalS s d) (eval t d)
evalS P            d = pi1 d
evalS (Comp s s′)  d = (evalS s ∘ evalS s′) d
nbe :: Type → Term → Term
nbe ty t = readback 0 (down (eval ty T) (eval t T))
nbeTy :: Type → Type
nbeTy ty = readback 0 (downT (eval ty T))
nbeOpen :: Ctx → Type → Term → Term
nbeOpen ctx ty t  = readback n (down (eval ty env) (eval t env))
  where n         = length ctx
        env       = mkenv n ctx
nbeOpenTy :: Ctx → Type → Type
nbeOpenTy ctx ty  = readback n (downT (eval ty env))
  where n         = length ctx
        env       = mkenv n ctx
mkenv :: Int → Ctx → Env
mkenv 0 []        = T
mkenv n (t : ts)  = PairD d′ (up td (Vd (n − 1)))
  where d′        = mkenv (n − 1) ts
        td        = eval t d′
mkvar :: Int → Term
mkvar n | n ≡ 0       = Q
        | otherwise   = Sub Q (subs (n − 1))
subs n  | n ≡ 0       = P
subs n  | otherwise   = Comp P (subs (n − 1))
```

Appendix B. Type-checking algorithm

Type checking algorithm for normal forms, and type inference algorithm for neutral terms.

Checking well-formedness of types.

```
chkType :: Ctx → Type → Bool
chkType ts U                       = True
chkType ts (Fun t r)               = chkType ts t ∧ chkType (t : ts) r
chkType ts (Singl a t)             = chkType ts t ∧ chkTerm ts t a
chkType ts (Sigma t r)             = chkType ts t ∧ chkType (t : ts) r
chkType ts Nat                     = True
chkType ts (Prf t)                 = chkType ts t
chkType ts (Enum n)                = True
chkType ts Q                       = chkNeTerm ts U Q
chkType ts w@(Sub Q s)             = chkNeTerm ts U w
chkType ts w@(App k v)             = chkNeTerm ts U w
chkType ts w@(Fst k)               = chkNeTerm ts U w
chkType ts w@(Snd k)               = chkNeTerm ts U w
chkType ts w@(Natrec t' v v' k)    = chkNeTerm ts U w
chkType _ _                        = False
```

Checking the types of terms.

```
sgSub :: Term → Term → Term
sgSub t t' = Sub t (Ext Is t')

chkTerm :: Ctx → Type → Term → Bool
chkTerm ts U           (Fun t t')   = chkTerm ts U t ∧
                                      chkTerm (t : ts) U t'
chkTerm ts U           (Singl e t)  = chkTerm ts U t ∧
                                      chkTerm ts t e
chkTerm ts U           (Sigma t t') = chkTerm ts U t ∧
                                      chkTerm (t : ts) U t'
chkTerm ts U             Nat        = True
chkTerm ts (Fun t t')  (Lam e)      = chkTerm (t : ts) t' e
chkTerm ts (Singl e t) e'           = chkTerm ts (nbeOpenTy ts t) e' ∧
                                      (nbeOpen ts e t) ≡ (nbeOpen ts e' t)
chkTerm ts (Sigma t r) (Pair e e')  = chkTerm ts t e ∧
                                      chkTerm ts (nbeOpenTy ts (sgSub r e)) e'
chkTerm ts Nat         Zero         = True
chkTerm ts Nat         (Suc t)      = chkTerm ts Nat t
chkTerm ts (Prf t)     (Box e)      = chkTerm ts t e
chkTerm ts (Enum n)    (Const m i)  = m ≡ n ∧ i < n
chkTerm ts t           e | neutral e = chkNeTerm ts t e
chkTerm _ _                _                 = False

neutral :: Term → Bool
neutral Q                   = True
neutral (Sub Q s)           = True
neutral (App k v)           = True
neutral (Fst k)             = True
neutral (Snd k)             = True
neutral (Natrec t' v v' k)  = True
neutral (Case n b ts t)     = True
neutral (Where t b p)       = True
neutral _                   = False

erase :: Type → Type
erase (Singl e t) = erase t
erase t           = t

maybeEr :: Maybe Type → Maybe Type
maybeEr = maybe Nothing (Just ∘ erase)

chkNeTerm :: Ctx → Type → Term → Bool
chkNeTerm ts t e = case maybeEr (infType ts e) of
                     Just t'   → t ≡ t'
                     Nothing → False
```

Inferring the types of neutral terms.

```
nbeType :: Ctx → Type → Maybe Type
nbeType ctx t = Just (nbeOpenTy ctx t)

infType :: Ctx → Term → Maybe Type
infType (t : ts) Q          = nbeType (t : ts) (Sub t P)
infType ts   (Sub Q s)      = case infType (infCtx ts s) Q of
                                Just t → nbeType ts (Sub t s)
                                _ → Nothing
infType ts (App e e')       = case maybeEr (infType ts e) of
                                  Just (Fun t t') →
                                    if chkTerm ts t e'
                                    then nbeType ts (sgSub t' e')
                                    else Nothing
                                  _ → Nothing
infType ts (Fst e)          = case maybeEr (infType ts e) of
                                  Just (Sigma t t') → Just t
                                  _ → Nothing
infType ts (Snd e)          = case maybeEr (infType ts e) of
                                  Just (Sigma t t') → nbeType ts (sgSub t' (Fst e))
                                  _ → Nothing
```

```
infType ts (Natrec t v w k) = case maybeEr (infType ts k) of
                                Just Nat → if
                                  chkType (Nat : ts) t ∧
                                  chkTerm ts (nbeOpenTy ts (sgSub t Zero)) v ∧
                                  chkTerm (Nat : ts)
                                      (Fun (sgSub t Q)
                                         (sgSub t (Suc (Sub Q P)))) w
                                  then nbeType ts (sgSub t k)
                                  else Nothing
                                _ → Nothing
infType ts (Where t b k)    = case maybeEr (infType ts k) of
                                Just (Prf t′) → if chkType ts t ∧
                                  chkTerm (t′ : ts) t b ∧
                                    nbeOpen ts′ w (Sub b (Ext (subs 1) Q)) ≡
                                    nbeOpen ts′ w (Sub b P)
                                  then Just t
                                  else Nothing
                                  where ts′ = Sub t′ P : t′ : ts
                                    w = Sub t (subs 1)
                                _ → Nothing
infType ts (Case n b cs k)  = case maybeEr (infType ts k) of
                                Just (Enum m) → if m ≡ n ∧
                                  chkType (Enum n : ts) b ∧
                                  chkList ts n b 0 cs
                                  then nbeType ts (sgSub b k)
                                  else Nothing
                                _ → Nothing
infType _ _                 = Nothing
chkList :: Ctx → Int → Type → Int → [Term] → Bool
chkList ts _ _ _ []         = True
chkList ts n b i (e : es)   = chkTerm ts (nbeOpenTy ts (sgSub b (Const n i))) e ∧
                                chkList ts n b (i + 1) es
infCtx :: Ctx → Subst → Ctx
infCtx (t : ts) P          = ts
infCtx (t : ts) (Comp P s) = infCtx ts s
```

Appendix C. Proofs

C.1. Proof of Lemma 5.19. By induction on $X = X' \in \mathcal{T}$. We do not show the base cases, for they are trivial.

(1) Let $\mathsf{Sing}\, d\, X = \mathsf{Sing}\, d'\, X' \in \mathcal{T}$.

$$\begin{aligned}
&[X] = [X'] && \text{by ind. hyp.}\\
&d = d' \in [X] && \text{by ind. hyp.}\\
&e = d \in [X] \text{ and } e' = d \in [X] && \text{hypothesis}\\
&e = d' \in [X] \text{ and } e' = d' \in [X] && \text{by transitivity}\\
&\{\!\{d\}\!\}_X = \{\!\{d'\}\!\}_{X'} && \text{by definition.}
\end{aligned}$$

(2) Let $\mathsf{Fun}\, X\, F = \mathsf{Fun}\, X'\, F' \in \mathcal{T}$.

$$\begin{aligned}
&[X] = [X'] && \text{by ind. hyp.}\\
&\text{for all } d \in dom([X]), F\ d = F'\ d \in \mathcal{T} && \text{by definition} && (*)\\
&\text{for all } d = d' \in [X], f \cdot d = f' \cdot d' \in [F\ d] && \text{hypothesis}\\
&f \cdot d = f' \cdot d' \in [F'\ d] && \text{by ind. hypothesis in } (*). && \square
\end{aligned}$$

C.2. Proof of Lemma 5.22. By induction on $X = X' \in \mathcal{T}$.

(a) Case $\mathsf{Sing}\, d\, X = \mathsf{Sing}\, d'\, X' \in \mathcal{T}$.

(1) The partial function $\uparrow$ maps neutrals to related elements in the corresponding PER.

$$\begin{aligned}
&k = k \in \mathcal{N}e && \text{hypothesis}\\
&d = d' \in [X] \text{ and } X = X' \in \mathcal{T} && \text{by inversion}\\
&\uparrow_{\mathsf{Sing}\, d\, X} k = d \text{ and } \uparrow_{\mathsf{Sing}\, d'\, X'} k' = d' && \text{by def.}\\
&d = d' \in \{\!\{d\}\!\}_X && \text{by def. of this PER.}
\end{aligned}$$

(2) The partial function $\downarrow$ maps related elements to related normal forms.

$$\begin{aligned}
&d_1 = d_2 \in \{\!\{d\}\!\}_X && \text{hypothesis}\\
&d_1 = d_2 = d = d' \in [X] \text{ and } X = X' \in \mathcal{T} && \text{by inversion}\\
&\downarrow_X d = \downarrow_{X'} d' \in \mathcal{N}f && \text{by ind. hyp.}\\
&\downarrow_{\mathsf{Sing}\, d\, X} d_1 = \downarrow_{\mathsf{Sing}\, d'\, X'} d_2 \in \mathcal{N}f && \text{by def.}
\end{aligned}$$

(3) The function $\Downarrow$ maps related elements in $\mathcal{T}$ to normal forms.

$$\begin{aligned}
&\Downarrow \mathsf{Sing}\, d\, X = \mathsf{Sing}\, (\downarrow_X d)\, (\Downarrow X) && \text{by def.}\\
&\Downarrow \mathsf{Sing}\, d'\, X' = \mathsf{Sing}\, (\downarrow_{X'} d')\, (\Downarrow X') && \text{by def.}\\
&\downarrow_X d = \downarrow_{X'} d' \in \mathcal{N}f && \text{by ind. hyp.}\\
&\Downarrow X = \Downarrow X' \in \mathcal{N}f && \text{by ind. hyp.}\\
&\mathsf{Sing}\, (\downarrow_X d)\, (\Downarrow X) = \mathsf{Sing}\, (\downarrow_X d)\, (\Downarrow X) \in \mathcal{N}f && \text{by Lem. 5.16.}
\end{aligned}$$

(b) Case $\mathsf{Fun}\, X\, F = \mathsf{Fun}\, X'\, F' \in \mathcal{T}$.

(1) The partial function $\uparrow$ maps neutrals to related elements in the corresponding PER.

$k = k' \in \mathcal{N}e$	hypothesis	
$d = d' \in [X]$	hypothesis	(*)
$X = X' \in \mathcal{T}$	by inversion	(†)
$F\ d = F'\ d' \in \mathcal{T}$	by inversion	(**)
$\downarrow_X d = \downarrow_{X'} d' \in \mathcal{N}f$	by ind. hyp. on (*) and (†)	
$\mathsf{App}\, k\, (\downarrow_X d) = \mathsf{App}\, k'\, (\downarrow_{X'} d') \in \mathcal{N}e$	by Lem. 5.16	(‡)
$\uparrow_{F\,d} (\mathsf{App}\, k\, (\downarrow_X d)) = \uparrow_{F'\,d'} (\mathsf{App}\, k'\, (\downarrow_{X'} d')) \in [F\ d]$	by ind. hyp. on (**) and (‡)	
$\uparrow_{\mathsf{Fun}\, X\, F}\, k = \uparrow_{\mathsf{Fun}\, X'\, F'}\, k' \in [\mathsf{Fun}\, X\, F]$	by def.	

(2) The partial function $\downarrow$ maps related elements to related normal forms.

$X = X' \in \mathcal{T}$	by inversion	(*)
$f = f' \in [\mathsf{Fun}\, X\, F]$	hypothesis	
$k = k' \in \mathcal{N}e$	hypothesis	
$\uparrow_X k = \uparrow_{X'} k' \in [X]$	by ind. hyp. on (*)	(†)
$d := \uparrow_X k$	abbreviation	
$d' := \uparrow_{X'} k'$	abbreviation	
$F\ d = F'\ d' \in \mathcal{T}$	by inversion and (†)	(**)
$f \cdot d = f' \cdot d' \in [F\ d]$	definition of $[\mathsf{Fun}\, X\, F]$	(‡)
$\downarrow_{F\ d} (f \cdot d) = \downarrow_{F'\ d'} (f' \cdot d') \in \mathcal{N}f$	by ind. hyp. on (‡)	
$(\downarrow_{\mathsf{Fun}\, X\, F} f) \cdot k = (\downarrow_{\mathsf{Fun}\, X'\, F'} f') \cdot k' \in \mathcal{N}f$	by def.	
$\downarrow_{\mathsf{Fun}\, X\, F} f = \downarrow_{\mathsf{Fun}\, X'\, F'} f' \in \mathcal{N}f$	by Lem. 5.16	

(3) The function $\Downarrow$ maps related elements in $\mathcal{T}$ to normal forms.

$X = X' \in \mathcal{T}$	by inversion	(*)
$\Downarrow X = \Downarrow X' \in \mathcal{N}f$	by ind. hyp. on (*)	(**)
$k = k' \in \mathcal{N}e$	hypothesis.	
$\uparrow_X k = \uparrow_{X'} k' \in [X]$	by ind. hyp. on (*)	(†)
$d := \uparrow_X k$	abbr.	
$d' := \uparrow_{X'} k'$	abbr.	
$F\, d = F'\, d' \in \mathcal{T}$	by inversion and (†)	(‡)
$\Downarrow(F\, d) = \Downarrow(F'\, d') \in \mathcal{N}f$	by ind. hyp. on (‡)	
$\Downarrow(\mathsf{Fun}\, X\, F) = \Downarrow(\mathsf{Fun}\, X'\, F') \in \mathcal{N}f$	by Lem. 5.16	□

C.3. Proof of Lemma 5.34. The proofs of soundness for (PRF-β) and (PRF-η) have the same structure, so we show only the first one.

(PRF-β) $b\,\mathsf{where}^B\,[a] = b\,(\mathsf{id}, a)$

$[\![b\,\mathsf{where}^B\,[a]]\!] d$

$= [\![b]\!](d, \star)$	def. of semantics for $b\,\mathsf{where}^B\,[a]$
$= [\![b]\!](d, [\![a]\!] d)$	ind. hypothesis on $\Gamma.A.A\,\mathsf{p} \vdash b\,\mathsf{p} = b\,(\mathsf{p}\,\mathsf{p}, \mathsf{q}) : B\,\mathsf{p}\,\mathsf{p}$
$= [\![b]\!]([\![(\mathsf{id}, a)]\!] d)$	def. of semantics for substitutions
$= [\![b\,(\mathsf{id}, a)]\!] d$	

(PRF-ASSOC) $a\,\mathsf{where}^A\,(b\,\mathsf{where}^B\,c) = (a\,(\mathsf{p}\,\mathsf{p}, \mathsf{q})\,\mathsf{where}^{A\,\mathsf{p}}\,b)\,\mathsf{where}^B\,c$

$$\begin{aligned}
[\![a\,\mathsf{where}^A\,(b\,\mathsf{where}^B\,c)]\!] d &= [\![a]\!](d, \star)\\
&= [\![a]\!](d, [\![b]\!](d, \star))\\
&= [\![a\,(\mathsf{p}\,\mathsf{p}, \mathsf{q})]\!]((d, \star), [\![b]\!](d, \star))\\
&= [\![a\,(\mathsf{p}\,\mathsf{p}, \mathsf{q})\,\mathsf{where}^{A\,\mathsf{p}}\,b]\!](d, \star)\\
&= [\![(a\,(\mathsf{p}\,\mathsf{p}, \mathsf{q})\,\mathsf{where}^{A\,\mathsf{p}}\,b)\,\mathsf{where}^B\,c]\!] d
\end{aligned}$$
□

C.4. Proof of Lemma 6.3. By induction on $X \in \mathcal{T}$.

(a) Types; in all cases we use symmetry and transitivity to show the conditions. We only show the case for $\mathsf{Fun}\,X\,F$.

(1) $X = \mathsf{Fun}\,X'\,F$:

$\Gamma \vdash A = \mathsf{Fun}\,B\,C$	by definition	(*)
$\Gamma \vdash B \sim X'$	by definition	
$\Delta \vdash C\,(\mathsf{p}^i, s) \sim F\ d \in \mathcal{T}$	by definition	
for all $\Delta \leqslant^i \Gamma$ and $\Delta \vdash s : B\,\mathsf{p}^i \sim d \in [X']$		
$\Gamma \vdash A' = \mathsf{Fun}\,B\,C$	by sym. and trans. on (*)	

(2) $\mathsf{N}_n \in \mathcal{T}$.

$\Gamma \vdash t : A \sim d \in [\mathsf{N}_1]$	hypothesis	(*)
$\Gamma \vdash t = t' : A$	hypothesis	(†)
$\Delta \vdash t\,\mathsf{p}^i = \mathsf{R}_i\,d : A\,\mathsf{p}^i$	by inversion on (*)	(**)
$\Delta \vdash t\,\mathsf{p}^i = t'\,\mathsf{p}^i : A\,\mathsf{p}^i$	by congruence on (†)	(‡)
$\Delta \vdash t'\,\mathsf{p}^i = \mathsf{R}_i\,d : A\,\mathsf{p}^i$	by sym. and trans. on (**) and (‡)	

(b) Terms. As in the case for types, we use symmetry and transitivity. We show only the case for singletons and functions.

(1) $X = \mathsf{Sing}\,d\,X'$:

$\Gamma \vdash A = \{b\}_B$	by hypothesis	(*)
$\Gamma \vdash B \sim X' \in \mathcal{T}$	by hypothesis	
$\Gamma \vdash t : B \sim d \in [X']$	by hypothesis	(†)
$\Gamma \vdash A' = \{b\}_B$	by sym. and trans. on (*)	
$\Gamma \vdash t' : B \sim d \in [X']$	By i.h. on (†)	

(2) $X = \mathsf{Fun}\, X'\, F$:

$\Gamma \vdash A = \mathsf{Fun}\, B\, C$	by hypothesis	(*)
$\Gamma \vdash B \sim X'$	by hypothesis	
$\Delta \vdash \mathsf{app}\; t\, \mathsf{p}^i\; s : C\,(\mathsf{p}^i, s) \sim f \cdot d \in [F\;\, d]$	by hypothesis	
for all $\Delta \leqslant^i \Gamma$ and $\Delta \vdash s : B\,\mathsf{p}^i \sim d \in [X']$		
$\Gamma \vdash A' = \mathsf{Fun}\, B\, C$	by sym. and trans. on (*)	(†)
$\Delta \vdash \mathsf{app}\; t\, \mathsf{p}^i\; s = \mathsf{app}\; t'\, \mathsf{p}^i\; s : C\,(\mathsf{p}^i, s)$	by congruence on (†)	(‡)
$\Delta \vdash \mathsf{app}\; t'\, \mathsf{p}^i\; s : C\,(\mathsf{p}^i, s) \sim f \cdot d \in [F\;\, d]$	by i.h. on (‡)	□

C.5. Proof of Lemma 6.4. By induction on $X \in \mathcal{T}$. This property is trivial for the base cases; for singletons is obtained by applying the i.h. We show two cases.

(1) Let $X = \mathsf{Fun}\, X'\, F$.

$\Gamma \vdash A = \mathsf{Fun}\, B\, C$	by hypothesis	(*)
$\Gamma \vdash B \sim X'$		(†)
$\Theta \vdash C\,(\mathsf{p}^i, s) \sim F\;\, d \in \mathcal{T}$	by hypothesis	
for all $\Theta \leqslant^i \Gamma$ and $\Theta \vdash s : B\,\mathsf{p}^i \sim d \in [X']$		
$\Delta \vdash A\,\mathsf{p}^i = \mathsf{Fun}\,(B\,\mathsf{p}^i)\,(C\,(\mathsf{p}^i\,\mathsf{p}, \mathsf{q}))$	by congruence on (*)	
$\Delta \vdash B\,\mathsf{p}^i \sim X$	by i.h. on (†)	
$\Theta' \vdash s : (B\,\mathsf{p}^i)\,\mathsf{p}^j \sim d \in [X]$, with $\Theta' \leqslant^j \Delta$	hypothesis	
$\Theta' \vdash s : B\,\mathsf{p}^{i+j} \sim d \in [X]$	by rem. 2.1 and 6.3	(‡)
$\Theta' \vdash C\,(\mathsf{p}^{i+j}\,\mathsf{q}, s) \sim F\;\, d$	by hyp. using (‡)	
$\Theta' \vdash C\,(\mathsf{p}^i\,\mathsf{p}, \mathsf{q})\,(\mathsf{p}^j, s) \sim F\;\, d$	By congruence and 6.3	

(2) $\mathsf{Prf}\, X \in \mathcal{T}$. As mentioned earlier if $\Gamma \vdash A \sim \mathsf{Prf}\, X \in \mathcal{T}$ then $\Gamma \vdash \_ : A \sim \_ \in [\mathsf{Prf}\, X]$ is non-empty if and only if $\Gamma \vdash \_ : A$ is not empty.

$\Gamma \vdash t : A \sim d \in [\mathsf{Prf}\, X]$	hypothesis	(*)
$\Gamma \vdash t : A$	by inversion on (*)	(†)
$\Gamma \vdash A \sim \mathsf{Prf}\, X \in \mathcal{T}$	by inversion on (*)	(**)
$\Delta \vdash t\,\mathsf{p}^i : A\,\mathsf{p}^i$	by weakening on (†)	
$\Delta \vdash A\,\mathsf{p}^i \sim \mathsf{Prf}\, X \in \mathcal{T}$	by monotonicity for types on (**)	
$\Delta \vdash t\,\mathsf{p}^i : A\,\mathsf{p}^i \sim d \in [\mathsf{Prf}\, X]$	by definition of log. rel.	

We do not show proofs for the second part, since the most involved case is dealt analogously to the case for $\mathsf{Fun}\, X'\, F$. □

C.6. Proof of Lemma 6.5. By induction on $X = X' \in \mathcal{T}$. Note that the first part for the base cases is trivial; the second point is also trivial for $X \in \mathcal{N}e$. Thus we do not show those parts of the proof.

(a) Types.

(1) $\mathsf{Sing}\, d\, X = \mathsf{Sing}\, d'\, X'$.

$\Gamma \vdash A = \{b\}_B$	by hypothesis	(*)
$\Gamma \vdash B \sim X \in \mathcal{T}$	by hypothesis	
$\Gamma \vdash t : B \sim d \in [X]$	by hypothesis	(†)
$\Gamma \vdash t : B \sim d' \in [X']$	By i.h. on (*) and (†)	

(2) $\mathsf{Fun}\, X\, F = \mathsf{Fun}\, X'\, F'$.

$\Gamma \vdash A = \mathsf{Fun}\, B\, C$	by hypothesis	
$\Gamma \vdash B \sim X'$	by hypothesis	(*)
$\Theta \vdash C\,(\mathsf{p}^i, s) \sim F\ d \in \mathcal{T}$	by hypothesis	(†)

$$\text{for all } \Theta \leqslant^i \Gamma \text{ and } \Theta \vdash s : B\,\mathsf{p}^i \sim d \in [X']$$

$\Gamma \vdash B \sim X' \in \mathcal{T}$	By i.h. on (*)	
$\Theta \vdash B\,(\mathsf{p}^i, s) \sim F'\ d \in \mathcal{T}$	by i.h. on (†)	

(b) Terms.

(1) $e = e' \in [\mathsf{Sing}\, d\, X]$.

$\Gamma \vdash A = \{b\}_B$	by hypothesis	
$\Gamma \vdash B \sim X \in \mathcal{T}$	by hypothesis	(*)
$\Gamma \vdash t : B \sim d \in [X]$	by hypothesis	(†)
$e' = d \in [X]$	by def. of $e = e' \in [\mathsf{Sing}\, d\, X]$	(**)
$\Gamma \vdash t : B \sim e' \in [X]$	by i.h. on (*), (†), and (**),	

(2) $f = f' \in [\mathsf{Fun}\, X\, F]$.

$\Gamma \vdash A = \mathsf{Fun}\, B\, C$

$\Gamma \vdash B \sim X$ (*)

$\Delta \vdash \mathsf{app}\; t\, \mathsf{p}^i\ s : C\,(\mathsf{p}^i, s) \sim f \cdot d \in [F\ d]$ (†)

$$\text{for all } \Delta \leqslant^i \Gamma \text{ and } \Delta \vdash s : B\,\mathsf{p}^i \sim d \in [X]\ . \qquad (**)$$

By i.h. on (*) and (**) and monotonicity 6.4

$$\Delta \vdash s : A'\,\mathsf{p}^i \sim d' \in [X']\ .$$

By i.h. on (†)

$$\Delta \vdash \mathsf{app}\; (t\,\mathsf{p}^i)\ s : B\,(\mathsf{p}^i, s) \sim f' \cdot d' \in [F\ d']\ .$$

(3) $d = d' \in [\mathsf{Sum}\, X\, F]$.

$d = d' \in [\mathsf{Sum}\, X\, F]$	hypothesis	(*)
$\Gamma \vdash t : A \sim d \in [\mathsf{Sum}\, X\, F]$	hypothesis	(**)
$\Gamma \vdash A = \Sigma\, A'\, B$	by inversion on (**)	
$\Gamma \vdash \Sigma\, A'\, B \sim \mathsf{Sum}\, X\, F \in \mathcal{T}$	by inversion on (**)	
$\Gamma \vdash \mathsf{fst}\, t : A' \sim \mathsf{fst}\, d \in [X]$	by inversion on (*)	(†)
$\Gamma.A' \vdash \mathsf{snd}\, t : B\,(\mathsf{id}, \mathsf{fst}\ t) \sim \mathsf{snd}\, d \in [F\ \mathsf{fst}\, d]$	by inversion on (**)	(‡)
$\mathsf{fst}\, d = \mathsf{fst}\, d' \in [X]$	by definition of (*)	(ǂ)
$\mathsf{snd}\, d = \mathsf{snd}\, d' \in [F\ \mathsf{fst}\, d]$	by definition of (*)	(⧧)
$\Gamma \vdash \mathsf{fst}\, t : A' \sim \mathsf{fst}\, d' \in [X]$	by ind. hyp. on (†) and (ǂ)	
$\Gamma.A' \vdash \mathsf{snd}\, t : B\,(\mathsf{id}, \mathsf{fst}\ t) \sim \mathsf{snd}\, d' \in [F\ \mathsf{fst}\, d']$	by ind. hyp. on (‡) and (⧧).	□

C.7. Proof of Lemma 6.6. By induction on $X \in \mathcal{T}$. By induction on $X \in \mathcal{T}$. For a better organisation of the proof we show the proofs for each point separately.

(a) $\Gamma \vdash A = \mathsf{R}_{|\Gamma|} \Downarrow X$. We skip the part for the minimal elements in $\mathcal{T}$.

(1) $\mathsf{Sing}\, d\, X$:

$\Gamma \vdash A' = \mathsf{R}_{|\Gamma|} \Downarrow X$	by ind. hyp.
$\Gamma \vdash t = \mathsf{R}_{|\Gamma|} \downarrow_X d : A'$	by ind. hyp.
$\Gamma \vdash \{a\}_{A'} = \{\mathsf{R}_{|\Gamma|} \downarrow_X d\}_{\mathsf{R}_{|\Gamma|} \Downarrow X}$	by congruence and transitivity

(2) $\mathsf{Fun}\, X\, F$:

$\Gamma \vdash A' = \mathsf{R}_{|\Gamma|} \Downarrow X$	by ind. hyp.	
$\Delta \vdash B\,(\mathsf{p}^i, s) = \mathsf{R}_{|\Delta|} \Downarrow F\ d$		(*)
for any $\Delta \leqslant^i \Gamma$ and $\Delta \vdash s : A'\, \mathsf{p}^i \sim d \in [X]$		
$\Gamma.A' \vdash \mathsf{q} : A'\, \mathsf{p} \sim \uparrow_X \mathsf{Var}\, x_{|\Gamma|}$	by ind. hyp.	(†)
$\Gamma.A' \vdash B\,(\mathsf{p}, \mathsf{q}) = \mathsf{R}_{|\Gamma.A'|} \Downarrow F\ \uparrow_X \mathsf{Var}\, x_{|\Gamma|}$	by instantiating (*) with (†)	
$\Gamma.A' \vdash B = \mathsf{R}_{|\Gamma.A'|} \Downarrow F\ \uparrow_X \mathsf{Var}\, x_{|\Gamma|}$	by 6.3.	

(b) $\Gamma \vdash t = \mathsf{R}_{|\Gamma|} \downarrow_X d : A$. We skip the part for the minimal elements in $\mathcal{T}$.

(1) $d' \in [\mathsf{Sing}\, d\, X]$:

$\Gamma \vdash A = \{b\}_B$		(*)
$\Gamma \vdash B \sim X \in \mathcal{T}$		
$\Gamma \vdash t : B \sim d \in [X]$		(†)
$\Gamma \vdash t = \mathsf{R}_{|\Gamma|} \downarrow_X d : B$	by ind. hyp. in (†)	
$\Gamma \vdash t = \mathsf{R}_{|\Gamma|} \downarrow_X d : \{t\}_B$	by conversion and (SING-EQ-I)	
$\Gamma \vdash t = \mathsf{R}_{|\Gamma|} \downarrow_X d : A$	by conversion	

(2) $f \in [\mathsf{Fun}\, X\, F]$:

$\Gamma.A' \vdash \mathsf{q} : A'\,\mathsf{p} \sim \uparrow_X \mathsf{Var}\, x_{\vert\Gamma\vert} \in [X]$	by ind. hyp. on the third part
$d := \uparrow_X \mathsf{Var}\, x_{\vert\Gamma\vert}$	abbreviation
$\Gamma.A' \vdash \mathsf{app}\ (t\,\mathsf{p})\ \mathsf{q} : B\,(\mathsf{p},\mathsf{q}) \sim f \cdot d \in [F\ d]$	by definition of the logical relation
$\Gamma.A' \vdash \mathsf{app}\ (t\,\mathsf{p})\ \mathsf{q} = \mathsf{R}_{\vert\Gamma.A\vert} \downarrow_{F\ d} f \cdot d : B\,(\mathsf{p},\mathsf{q})$	by ind. hyp.
$\Gamma.A' \vdash \mathsf{app}\ (t\,\mathsf{p})\ \mathsf{q} = \mathsf{R}_{\vert\Gamma.A\vert} \downarrow_{F\ d} f \cdot d : B$	by (conv)
$\Gamma \vdash \lambda(\mathsf{app}\ (t\,\mathsf{p})\ \mathsf{q}) = \lambda(\mathsf{R}_{\vert\Gamma.A\vert} \downarrow_{F\ d} f \cdot d) : \mathsf{Fun}\, A'\, B$	by congruence
$\Gamma \vdash t = \lambda(\mathsf{app}\ (t\,\mathsf{p})\ \mathsf{q}) : \mathsf{Fun}\, A'\, B$	by (eta)
$\Gamma \vdash t = \mathsf{R}_{\vert\Gamma\vert} \downarrow_{\mathsf{Fun}\, X\, F} f : \mathsf{Fun}\, A'\, B$	by trans.

(c)

(1) $\mathsf{Sing}\, d\, X$:

$\Gamma \vdash A = \{a\}_B$	by hypothesis
$\Gamma \vdash B \sim X \in \mathcal{T}$	by hypothesis
$\Gamma \vdash t : B \sim d \in [X]$	by hypothesis
$\Gamma \vdash B \sim X \in \mathcal{T}$	by monotonicity 6.4
$\Delta \vdash t\,\mathsf{p}^i : B\,\mathsf{p}^i \sim d \in [X]$	by monotonicity 6.4
$\Delta \vdash A\,\mathsf{p}^i = \{a\,\mathsf{p}^i\}_{B\,\mathsf{p}^i}$	by congruence

(2) $\mathsf{Fun}\, X\, F$:

$\Delta \vdash s : A'\,\mathsf{p}^i \sim d' \in [X]$	hypothesis	(*)
$\Delta \vdash s = \mathsf{R}_{\vert\Delta\vert} \downarrow_X d' : A'\,\mathsf{p}^i$	by ind. hyp. on (*)	
$\Delta \vdash \mathsf{app}\ (t\,\mathsf{p}^i)\ s = \mathsf{app}\ ((\mathsf{R}_{\vert\Gamma\vert}\, d)\,\mathsf{p}^i)\ (\mathsf{R}_{\vert\Delta\vert}\,(\downarrow_X d')) : B\,(\mathsf{p}^i, s)$	by congruence	
$\mathsf{R}_{\vert\Delta\vert}\, \mathsf{App}\, d \downarrow_X d' = \mathsf{app}\ ((\mathsf{R}_{\vert\Gamma\vert}\, d)\,\mathsf{p}^i)\ (\mathsf{R}_{\vert\Delta\vert}\,(\downarrow_X d'))$	by definition	
$\Delta \vdash \mathsf{app}\ (t\,\mathsf{p}^i)\ s : B\,(\mathsf{p}^i, s) \sim \uparrow_{F\ d'} \mathsf{App}\, d \downarrow_X d' \in [F\ d']$	by ind. hyp.	□

C.8. Proof of Lemma 6.13. By induction on $\Gamma\ \vdash$; we show only the inductive case. Let $\Gamma.A \vdash$.

$d := \rho_\Gamma$	definition	
$\Gamma \vdash \mathsf{id} : \Gamma \sim d \in (\!\vert\Gamma\vert\!)$	by inversion and i.h.	(*)
$\Gamma.A \vdash \mathsf{id}\,\mathsf{p} : \Gamma \sim d \in (\!\vert\Gamma\vert\!)$	from (*) by Rem. 6.9	(†)
$\Gamma.A \vdash \mathsf{p} : \Gamma \sim d \in (\!\vert\Gamma\vert\!)$	from (†) by Rem. 6.8	(**)
$\Gamma.A \vdash A\,\mathsf{p} \sim [\![A]\!]d \in \mathcal{T}$	by inversion and Thm. 6.11	
$\Gamma.A \vdash \mathsf{q} : A\,\mathsf{p} \sim \uparrow_{[\![A]\!]d} \mathsf{Var}\, x_n \in [[\![A]\!]d]$	by Thm. 6.11	
$\Gamma.A \vdash (\mathsf{p},\mathsf{q}) : \Gamma.A \sim (d, \uparrow_{[\![A]\!]d} \mathsf{Var}\, x_n) \in \coprod (\!\vert\Gamma\vert\!)\,(e \mapsto [[\![A]\!]e])$	by Def. 6.7	(‡)
$\Gamma.A \vdash \mathsf{id} : \Gamma.A \sim \rho_{\Gamma.A} \in (\!\vert\Gamma.A\vert\!)$	from (‡) by Rem. 6.8	□

C.9. Proof of Theorem 6.11. We note that for terms we show only the cases when the last rule used was the introductory rule, or the rule for introducing elements in singletons; for the case of the conversion rule, we can conclude by i.h., and lemma 6.3.

(a) Types. We show only the case for (FUN-F).

$\Delta \vdash s : A' \,\mathsf{p}^i \sim e \in [X]$	hypothesis	(*)
$\Theta \vdash \delta\, \mathsf{p}^i : \Gamma \sim d \in (\![\Gamma]\!)$	By monotonicity for substitutions 6.9	(†)
$\Theta \vdash (\delta\, \mathsf{p}^i, s) : \Gamma.A \sim (d, e) \in (\![\Gamma.A]\!)$	From (*) and (†)	
$\Theta \vdash B\,(\delta\, p^i, s) \sim [\![B]\!](d,e) \in \mathcal{T}$	by ind. hyp. on $\Gamma.A \vdash B$ and using 6.3 and 6.5	

(b) Terms. We show the case for application (FUN-EL) and for (N_n-E). The case for abstraction (FUN-I) is analogous to (FUN-F).

(1) (FUN-EL)

$\Gamma \vdash \mathsf{app}\; t\; r : B\,(\mathsf{id}, r)$	hypothesis	
$\Delta \vdash r\,\delta : A\,\delta \sim [\![r]\!]d \in [[\![A]\!]d]$	by ind. hyp.	(*)
$\Delta \vdash t\,\delta : \mathsf{Fun}\, A\, B\, \delta \sim [\![t]\!]d \in [[\![\mathsf{Fun}\, A\, B]\!]d]$	by ind. hyp.	(†)
$\Delta \vdash \mathsf{app}\;(t\,\delta)\;(r\,\delta) : B\,(\mathsf{id}, r\,\delta) \sim$ $[\![t]\!]d \cdot [\![r]\!]d \in [[\![B]\!](d, [\![r]\!]d)]$	by def. of log. rel. for (†) with (*)	
$\Delta \vdash (\mathsf{app}\; t\; r)\,\delta : B\,(\mathsf{id}, r\,\delta) \sim$ $[\![\mathsf{app}\; t\; r]\!]d \in [[\![B]\!](d, [\![r]\!]d)]$	by 6.3 and 6.5	

(2) (N_n-E)

$\Gamma \vdash \mathsf{case}^B\; t_0 \cdots t_{n-1}\; t \;: B\,(\mathsf{id}, t)$	hypothesis
$\Delta \vdash t\,\delta : \mathsf{N}_n \sim [\![t]\!]d \in [\mathsf{N}_n]$	by inversion and by ind. hyp.
$\Delta \vdash t\,\delta = \mathsf{R}_{\lvert\Delta\rvert}\, [\![t]\!]d : \mathsf{N}_n$	by 6.6
$\Delta \vdash t_i\,\delta : B\,(\delta, \mathsf{c}_i) \sim [\![t_i]\!]d \in [[\![B]\!](d, [\![t]\!]d)]$	by inversion and by ind. hyp.
if $\mathsf{R}_{\lvert\Delta\rvert}\, [\![t]\!]d \equiv \mathsf{c}_i$:	
$\Delta \vdash (\mathsf{case}\; B\; t_0 \cdots t_{n-1}\; \mathsf{c}_i)\,\delta = t_i\,\delta : B\,(\mathsf{id}, t)$	by subst.
$\Delta \vdash (\mathsf{case}\; B\; t_0 \cdots t_{n-1}\; \mathsf{c}_i)\,\delta : B\,(\mathsf{id}, t) \sim$ $[\![\mathsf{case}^B\; t_0 \cdots t_{n-1}\; t\;]\!]d \in [[\![B\,(\mathsf{id}, t)]\!]d]$	by 6.3 and 6.5
if $\mathsf{R}_{\lvert\Delta\rvert}\, [\![t]\!]d \in Ne$:	
$\Delta.\mathsf{N}_n \vdash B\,(\delta\,\mathsf{p}, \mathsf{q}) = \mathsf{R}_{\lvert\Delta\rvert+1} \Downarrow [\![B]\!](d, \mathsf{Var}\, x_{\lvert\Delta\rvert})$	
$\Delta \vdash t_i\,\delta = \mathsf{R}_{\lvert\Delta\rvert}\, [\![t_i]\!]d : B\,(\delta, \mathsf{c}_i)$	by 6.6
$t'_i := \mathsf{R}_{\lvert\Delta\rvert}\, [\![t_i]\!]d$	abbreviation
$t' := \mathsf{R}_{\lvert\Delta\rvert} \downarrow_{[\![B]\!](d,\mathsf{c}_i)} [\![t]\!]d$	abbreviation
$B' := \mathsf{R}_{\lvert\Delta\rvert+1}\, [\![B]\!](d, \mathsf{Var}\, x_{\lvert\Delta\rvert})$	abbreviation
$\Delta \vdash (\mathsf{case}\; B\; t_0 \cdots t_{n-1}\; t)\,\delta =$ $\mathsf{case}\; B'\; t'_0 \cdots t'_{n-1}\; t' : B\,(\delta, t)$	congruence

$$\Delta \vdash (\mathsf{case}\ B\ t_0 \cdots t_{n-1}\ t)\,\delta : B\,(\delta, t) \sim$$
$$\uparrow_{[\![B]\!](d,[\![t]\!]d)} (\mathsf{case}\ B'\ t'_0 \cdots t'_{n-1}\ t' \in [\![B]\!](d, [\![t]\!]d)) \qquad \text{by 6.6 and 6.4}$$
$$\Delta \vdash (\mathsf{case}\ B\ t_0 \cdots t_{n-1}\ \mathsf{c}_i)\,\delta : B\,(\mathsf{id}, t) \sim$$
$$[\![\mathsf{case}^B\ t_0 \cdots t_{n-1}\ t\ ]\!]d \in [[\![B\,(\mathsf{id}, t)]\!]d] \qquad \text{by 6.3 and 6.5}$$

(c) Substitutions. Only the proof for (EXT-SUBS) is shown.

$\Gamma \vdash \Theta.A : (\gamma, t)$	hypothesis	
$\Delta \vdash \gamma\,\delta : \Theta \sim [\![\gamma]\!]d \in (\![\Theta]\!)$	by ind. hyp.	(*)
$\Delta \vdash t\,\delta : (A\,\gamma)\,\delta \sim [\![t]\!]d \in [[\![A\,\gamma]\!]d]$		(†)
$([\![\gamma]\!]d, [\![t]\!]d) \in \coprod (\![\Theta]\!)\,(e \mapsto [[\![A\,\gamma]\!]e])$	from (*) and (†)	
$\Delta \vdash (\gamma, t)\,\delta : \Theta.A \sim [\![(\gamma, t)]\!]d \in (\![\Theta.A]\!)$	by 6.8 and 6.10	□

C.10. Proof of Theorem 7.4. By simultaneous induction on $\Gamma \vdash V \Leftarrow$, $\Gamma \vdash v \Leftarrow V$, and $\Gamma \vdash V \Rightarrow k$.

(1) Types:
- the case for $\mathsf{Fun}\,V\,W$ is also obtained directly from the derivations we get using the i.h. on $\Gamma \vdash V \Leftarrow$, and $\Gamma.V \vdash W \Leftarrow$; and use them for deriving $\Gamma \vdash \mathsf{Fun}\,V\,W$
- for $\{v\}_V$, we can apply the same reasoning as before: by i.h. on $\Gamma \vdash V \Leftarrow$, and $\Gamma \vdash v \Leftarrow \mathbf{nbe}(V)$ we know that there are, respectively, derivations with conclusions $\Gamma \vdash V$, and $\Gamma \vdash v : V$; from which we can conclude $\Gamma \vdash \{v\}_V$
- here we'll consider the three cases when V is a neutral term, because the reasoning is the same. By i.h. on $\Gamma \vdash V \Leftarrow \mathsf{U}$, we have a derivation with conclusion $\Gamma \vdash V : \mathsf{U}$; hence we use (U-EL).

(2) Terms:
- let $V = \mathsf{U}$, and $v = \mathsf{Fun}\,V'\,W$. By i.h. $\Gamma \vdash V' : \mathsf{U}$, and $\Gamma.V' \vdash W : \mathsf{U}$, and using both derivations we can derive $\Gamma \vdash \mathsf{Fun}\,V\,W : \mathsf{U}$.
- consider $V = \mathsf{U}$, and $v = \{v'\}_{V'}$. by i.h. on $\Gamma \vdash V' \Leftarrow \mathsf{U}$, and $\Gamma \vdash v' \Leftarrow \mathbf{nbe}(V)$, we have $\Gamma \vdash V : \mathsf{U}$, and $\Gamma \vdash v' : \mathbf{nbe}(V)$, and using conversion we derive $\Gamma \vdash v' : V$; and these are the premises we need to show $\Gamma \vdash \{v'\}_V : \mathsf{U}$.
- $V = \mathsf{Fun}\,V'\,W$, and $v = \lambda v'$: we have $\Gamma.V' \vdash v' \Leftarrow W$. From this we can conclude by i.h. $\Gamma.V' \vdash v' : W$; and this is the key premise for concluding $\Gamma \vdash \lambda v' : \mathsf{Fun}\,V'\,W$.
- $V = \{w\}_W$: by hypothesis we know $\Gamma \vdash w : W$, and $\Gamma \vdash v \Leftarrow W$, and $\Gamma \vdash w = v : W$; by the i.h. on the second one we get $\Gamma \vdash v : W$; then we can conclude using (SING-I).
- $v = k \in Ne$, and $V \not\equiv \{w\}_W$: let $\Gamma \vdash k \Rightarrow V'$, then we distinguish the cases when V' is a singleton, and when V' is not a singleton. In the latter case, the derivation is obtained directly from the correctness of type-inference. In the first case we use the rule (SING-EL), with the derivation obtained by i.h. and then we conclude with conversion.

(3) Inference:
- for $\mathsf{q}\,\mathsf{p}^i$, if $i = 0$, then we use (HYP), and conversion; if $i > 0$, then we have a derivation with conclusion $\Gamma \vdash \mathsf{q} : A_i\,\mathsf{p}$, and clearly $\Gamma \vdash \mathsf{p}^i : \Gamma.A_i \ldots .A_0$, hence by (SUBS-TERM), we have $\Gamma.A_i \ldots .A_0 \vdash \mathsf{q}\,\mathsf{p}^i : A_i\,\mathsf{p}^{i+1}$, we conclude by correctness of $\mathbf{nbe}(\_)$ and by conversion.

- by i.h. we have derivations with conclusions $\Gamma \vdash k : V'$, with $\overline{V'} = \mathsf{Fun}\, V\, W$, hence we have a derivation $\Gamma \vdash k : \mathsf{Fun}\, V\, W$ (using (SING-EL) if necessary) and $\Gamma \vdash v : V$, hence by the rule (FUN-EL), we have $\Gamma \vdash \mathsf{app}\ k\ v : W\,(\mathsf{id}, v)$. We conclude by conversion and correctness of $\mathbf{nbe}(\_)$. □

C.11. Proof of Theorem 7.6. We prove simultaneously all the points. The first point is by induction on the structure of the type. In the last two points we use well-founded induction on the order $\prec$.

(1) Types:
- $\Gamma \vdash \mathsf{Fun}\, V'\, W$; by inversion we know $\Gamma \vdash V'$, and $\Gamma.V' \vdash W$; hence by i.h. we have respectively $\Gamma \vdash V' \Leftarrow$, and $\Gamma.V' \vdash W \Leftarrow$.
- $V = \{v\}_{V'}$: by inversion we have $\Gamma \vdash V'$, and $\Gamma \vdash v : \mathbf{nbe}(V')$, hence by i.h. we have both $\Gamma \vdash V' \Leftarrow$, and $\Gamma \vdash v \Leftarrow V'$.
- $\Gamma \vdash k$, we have to show $\Gamma \vdash k \Leftarrow$. By lemma 2.7, we know $\Gamma \vdash k : \mathsf{U}$; hence by i.h. we have $\Gamma \vdash k \Rightarrow A$, and $\Gamma \vdash A = \mathsf{U}$, hence $\Gamma \vdash k \Leftarrow \mathsf{U}$.

(2) Terms: We omit the trivial cases, e.g. (U, A); we have re-arranged the order of the cases for the sake of clarity.
- $v = \mathsf{Fun}\, V'\, W$:
  (a) either $\Gamma \vdash A = \mathsf{U}$, $\Gamma \vdash V' : \mathsf{U}$, and $\Gamma.V' \vdash W : \mathsf{U}$; hence, by i.h. we know both $\Gamma \vdash V' \Leftarrow \mathsf{U}$, and $\Gamma.V' \vdash W \Leftarrow \mathsf{U}$; hence we can conclude $\Gamma \vdash \mathsf{Fun}\, V\, W \Leftarrow \mathsf{U}$.
  (b) Or $\Gamma \vdash A = \{a\}_{A'}$, $\Gamma \vdash v : A'$, and $\Gamma \vdash v = a : A'$, hence by i.h. we know $\Gamma \vdash v \Leftarrow \mathbf{nbe}(B)$, by conversion we also have and transitivity of the equality $\Gamma \vdash \mathbf{nbe}(a) = v : \mathbf{nbe}(B)$, hence $\Gamma \vdash v \Leftarrow \{\mathbf{nbe}(a)\}_{\mathbf{nbe}(B)}$.
- $v = \{v'\}_V$:
  (a) $\Gamma \vdash V : \mathsf{U}$, and $\Gamma \vdash v' : V$. From those derivations we have by i.h. $\Gamma \vdash V \Leftarrow \mathsf{U}$, and $\Gamma \vdash v' \Leftarrow \mathbf{nbe}(V)$, respectively; from which we conclude $\Gamma \vdash \{v'\}_V \Leftarrow \mathsf{U}$
  (b) $\Gamma \vdash A = \{a\}_{A'}$, with $\Gamma \vdash v : A'$, and $\Gamma \vdash v = a : A'$, hence by i.h. we know $\Gamma \vdash v \Leftarrow \mathbf{nbe}(B)$. We can also derive $\Gamma \vdash \mathbf{nbe}(a) = v : \mathbf{nbe}(B)$, hence $\Gamma \vdash v \Leftarrow \{\mathbf{nbe}(a)\}_{\mathbf{nbe}(B)}$.
- $v = \lambda v'$
  (a) $\Gamma \vdash V = \mathsf{Fun}\, A'\, B$, and $\Gamma.A' \vdash v' : B$; from this we can conclude $\Gamma.\mathbf{nbe}(A') \vdash v' : B$ by ind. hyp. we get $\Gamma.\mathbf{nbe}(A') \vdash v' \Leftarrow \mathbf{nbe}(B)$; therefore $\Gamma \vdash \lambda v' \Leftarrow \mathsf{Fun}\, \mathbf{nbe}(A')\, \mathbf{nbe}(B')$.
  (b) Or $\Gamma \vdash A = \{a\}_{A'}$, $\Gamma \vdash v : A'$, and $\Gamma \vdash v = a : A'$, hence by i.h. we know $\Gamma \vdash v \Leftarrow \mathbf{nbe}(B)$, by conversion we also have and transitivity of the equality $\Gamma \vdash \mathbf{nbe}(a) = v : \mathbf{nbe}(B)$, hence $\Gamma \vdash v \Leftarrow \{\mathbf{nbe}(a)\}_{\mathbf{nbe}(B)}$.
- $v \in \mathit{Ne}$: then we do case analysis on $\mathbf{nbe}(A)$.
  (a) If $\mathbf{nbe}(A) = \{w\}_W$, then by soundness of $\mathbf{nbe}(\_)$, and conversion we have $\Gamma \vdash k : \{w\}_W$; and by inversion of singletons we have $\Gamma \vdash k : W$, and also $\Gamma \vdash k = w : W (*)$. Clearly $(k, W) \prec (k, A)$, hence we can apply the inductive hypothesis and conclude $\Gamma \vdash k \Leftarrow W$; from that and $(*)$, we conclude $\Gamma \vdash k \Leftarrow \{w\}_W$, i.e., $\Gamma \vdash k \Leftarrow \mathbf{nbe}(A)$.
  (b) If $V \not\equiv \{w\}_W$, then $\overline{V} \equiv V$. We use the last clause for concluding $\Gamma \vdash k \Leftarrow \mathbf{nbe}(A)$; but we need to show that if $\Gamma \vdash k \Rightarrow V'$, then $\Gamma \vdash \overline{V} = \overline{V'}$; we show this in the next point.

(3) Inference: let $\Gamma \vdash k : A$, $\Gamma \vdash k \Rightarrow V'$, and $V = \mathbf{nbe}(A)$. Show $\Gamma \vdash \overline{V} = \overline{V'}$.

- let us consider first the case when $V = \{w\}_W$; by inversion we have derivations $\Gamma \vdash k : W$, and $\Gamma \vdash k = w : W$. Hence by i.h. we know that $\Gamma \vdash \overline{V'} = \overline{W}$, and $\overline{W} = \overline{\{w\}_W}$.
- Now we consider the case when V is not a singleton, and $k = \mathsf{q}\,\mathsf{p}^i$; this case is trivial because by inversion we know that $\Gamma \vdash V = \mathbf{nbe}((\Gamma!i)\,\mathsf{p}^{i+1})$.
- the last case to consider is $k = \mathsf{app}\ k'\ v$ and V not a singleton. By inversion we know $\Gamma \vdash \mathsf{app}\ k\ v : B\,(\mathsf{id}, v)$, and $\Gamma \vdash k : \mathsf{Fun}\,A\,B$, hence $\Gamma \vdash k : \mathsf{Fun}\,\mathbf{nbe}(A)\,\mathbf{nbe}(B)$, and $\Gamma \vdash v : A$, hence $\Gamma \vdash v : \mathbf{nbe}(A)$. By i.h. we know that if $\Gamma \vdash k \Rightarrow V'$, then $\overline{V'} = \mathsf{Fun}\,\mathbf{nbe}(A)\,\mathbf{nbe}(B)$, and also $\Gamma \vdash v \Leftarrow \mathbf{nbe}(A)$. Hence we can conclude $\Gamma \vdash \mathsf{app}\ k\ v \Rightarrow \mathbf{nbe}(\mathbf{nbe}(B)\,(\mathsf{id}, v))$. And $\Gamma \vdash \mathbf{nbe}(\mathbf{nbe}(B)\,(\mathsf{id}, v)) = \mathbf{nbe}(B\,(\mathsf{id}, v))$ (by correctness of the $\mathbf{nbe}(\_)$ algorithm). $\square$