Normalization by evaluation for sized dependent types
Andreas Abel, Andrea Vezzosi, Théo Winterhalter

To cite this version:
Andreas Abel, Andrea Vezzosi, Théo Winterhalter. Normalization by evaluation for sized dependent types. Proceedings of the ACM on Programming Languages, ACM, 2017, 1, pp.33. 10.1145/3110277 hal-01596179

HAL Id: hal-01596179
https://hal.archives-ouvertes.fr/hal-01596179
Submitted on 27 Sep 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers. L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Normalization by Evaluation for Sized Dependent Types

ANDREAS ABEL, Gothenburg University, Sweden
ANDREA VEZZOSI, Chalmers University of Technology, Sweden
THEO WINTERHALTER, École normale supérieure Paris-Saclay, France

Sized types have been developed to make termination checking more perspicuous, more powerful, and more modular by integrating termination into type checking. In dependently-typed proof assistants where proofs by induction are just recursive functional programs, the termination checker is an integral component of the trusted core, as validity of proofs depend on termination. However, a rigorous integration of full-fledged sized types into dependent type theory is lacking so far. Such an integration is non-trivial, as explicit sizes in proof terms might get in the way of equality checking, making terms appear distinct that should have the same semantics.

In this article, we integrate dependent types and sized types with higher-rank size polymorphism, which is essential for generic programming and abstraction. We introduce a size quantifier $\forall$ which lets us ignore sizes in terms for equality checking, alongside with a second quantifier $\Pi$ for abstracting over sizes that do affect the semantics of types and terms. Judgmental equality is decided by an adaptation of normalization-by-evaluation for our new type theory, which features type shape-directed reflection and reification. It follows that subtyping and type checking of normal forms are decidable as well, the latter by a bidirectional algorithm.

CCS Concepts: • Theory of computation → Type theory; Type structures; Program verification; Operational semantics;

Additional Key Words and Phrases: dependent types, eta-equality, normalization-by-evaluation, proof irrelevance, sized types, subtyping, universes.

ACM Reference Format:

1 INTRODUCTION

Dependently-typed programming languages and proof assistants, such as Agda [2017] and Coq [INRIA 2016], require programs to be total, for two reasons. First, for consistency: since propositions are just types and proofs of a proposition just programs which inhabit the corresponding type, some types need to be empty; otherwise, each proposition would be true. However, in a partial language with general recursion, each type is inhabited by the looping program $f = f$. Secondly, totality is needed for decidability of type checking. Since types can be the result of a computation, we need computation to terminate during type checking, even for open terms, i.e., terms with free variables.

Consequently, the aforementioned languages based on Type Theory come with a termination checker, which needs to reject all non-terminating programs, and should accept sufficiently many
terminating programs to allow the user to express her algorithms. In current termination checkers, programs are required to terminate by structural descent [Giménez 1995]; the structural order may be extended to a lexicographic [Abel and Altenkirch 2002] or size-change termination criterion [Lee et al. 2001; Wahlstedt 2007]. This is not a fundamental limitation, since Type Theory allows many functions to be expressed in a structurally recursive manner, if needed by the help of a well-founded relation [Nordström 1988], inductive domain predicates [Bove and Capretta 2005], or inductive descriptions of the function graph [Bove 2009]. However, the syntactic termination check is very sensitive to reformulations of the program and hostile to abstraction [Abel 2012].

Sized types [Hughes et al. 1996] delegate the checking for structural descent to the type system by annotating data types with a size parameter. The type checker can then ensure that in recursive calls the size goes down, certifying termination. In the simplest setting [Abel 2008; Barthe et al. 2004], the size is just an upper bound on the tree height of the data structure; however, more sophisticated size annotations have also been considered [Blanqui 2004; Xi 2002]. Most sized type systems are non-dependent [Abel and Pientka 2016; Amadio and Coupet-Grimal 1998; Barthe et al. 2008a,b; Blanqui and Riba 2006; Lago and Grellois 2017], yet the combination of sized and dependent types has been studied as well [Barthe et al. 2006; Blanqui 2005; Gréggoire and Sacchini 2010; Sacchini 2013, 2014]. However, to the best of our knowledge, no study combines higher-rank size polymorphism with full-fledged dependent types.1

Higher-rank size quantification takes termination checking to the next level; it is necessary for abstraction and generic programming. For instance, it allows us to write a generic tree traversal which applies a user-given preprocessor on subtrees before recursively descending into these trees, and a postprocessor after surfaced from the descent. The condition is that preprocessing does not increase the size of the subtree; otherwise, termination could not be guaranteed. Concretely, assume a type T i of trees of size < i with a constructor node : ∀i. List (T i) → T (i + 1) which takes a finite list of subtrees to form a new tree. In the following definition of trav, the preprocessing pre : ∀i. T i → T i can be safely applied to input tree t because the type of pre bounds the size of pre t by the size of t. In case pre t = node ts, the trees in the list ts are still guaranteed to be of strictly smaller size than t, thus, the recursive call to trav, communicated via the map function for lists, is safe.

\[
\text{trav} : (\text{pre} : \forall i. \text{T} i \to \text{T} i) (\text{post} : \forall j. \text{T} \to \text{T} j) \to \forall i. \text{T} i \to \text{T} \infty \\
\text{trav pre post t = post (case pre t of \{ node ts -> node (map (trav pre post) ts) \})}
\]

The display shows the Curry-style program as provided by the user, but state-of-the-art type checkers elaborate the program from surface syntax into an internal Church-style syntax with explicit type abstractions and type applications.2 With implicit type and size applications elaborated, trav would look as follows:

\[
\text{trav pre post i t = post (case pre i t of \{ node j ts -> node \infty (map (T j) (T \infty) (trav pre post j) ts) \})}
\]

Church-style syntax is the basis for all program analyses and transformations to follow and should be considered as the true syntax. However, from a dependent-type perspective, explicit size applications in terms can be problematic when the type checker compares terms for equality—which is necessary as types can depend on values. Inferred sizes may not be unique, as we have subtyping T i ≤ T j for i ≤ j: we can always weaken an upper bound. For instance, given ts : List (T i), any of the terms node i ts, node (i + 1) ts, . . . , node \infty ts has type T \infty. Yet semantically, all these trees are equal, thus, the syntactic equality check should ignore the size argument to node. Similarly, in the

1 Xi [2002] has first-class size polymorphism, but only indexed types, no universes or large eliminations.

2 Agda, Coq, Idris [Brady 2013], and Haskell [Sulzmann et al. 2007] all have Church-style internal languages.
application \( pre\ i\ t \) the size argument \( i \) should be ignored by the equality check. Yet \( pre\ i : T\ i \to T\ i \) and \( pre\ j : T\ j \to T\ j \) have different types for \( i \neq j \), and moreover these function types are not in the subtyping relation due to the mixed-variant occurrence of the size parameter. It seems that during equality checking we have to consider terms of different types, at least for a while. Once we apply \( pre\ i \) and \( pre\ j \) to the same tree \( t : T\ k \), which determines \( i = j = k \), we are back to safety. However, allowing types to differ during an equality check needs special consideration, especially when the equality-check is type directed.

Consider the analogous situation for the polymorphic lambda calculus System F, be it the predicative variant or not, extended by a unit type 1. For Church-style, we can give a type-directed \( \beta\eta \)-equality test which equates all terms at the unit type. The most interesting rules are the \( \eta \)-rules for unit and function type and the congruence rule for type application:

- \[ \frac{\Gamma \vdash t = t' : 1}{\Gamma, x : A \vdash t\ x = t'\ x : B} \]
- \[ \frac{\Gamma \vdash t = t' : A \to B}{\Gamma \vdash t\ A = t'\ A : B[A/X]} \]

The Curry-style version replaces the last conclusion by \( \Gamma \vdash t = t' : [B/A][X] \) where the type \( A \) to instantiate \( X \) has to be guessed. However, in Curry-style more terms are equated than in Church-style, as for instance the Church-style terms \( t\ A(\lambda x : A\ x) \) and \( t\ B(\lambda x : B\ x) \) map to the same Curry-style term \( t(\lambda x : x) \). How would we adapt the algorithm for Church-style such that it equates all terms that are equal in Curry-style? The conclusion of the last rule could be changed to \( \Gamma \vdash t\ A = t'\ A' : [B/A][X] \), but then the second term \( t'\ A' \) does not have the ascribed type \( B[A/X] \), and \( \eta \)-laws applied to this term would be unsound. For instance, the algorithm would yield \( t\ 1\ x = t\ (A \to A)\ y \) even for \( x \neq y \). We could also consider a heterogeneous check \((\Gamma \vdash t : A) = (\Gamma' \vdash t' : A')\) where each term is paired with its own type and context, but this leaves us with the dilemma of explaining the meaning of this judgement when \( A \) and \( A' \) are incompatible.

Does the literature offer a solution to this problem? In fact, a Church-style calculus with Curry-style equality has been studied before, it is ICC\( \ast \) [Barras and Bernardo 2008; Mishra-Linger and Sheard 2008] based on Miquel’s Implicit Calculus of Constructions [2001]. In ICC\( \ast \), equality is checked by erasing all type abstractions and applications, and comparing the remaining untyped terms for \( \beta\eta \)-equality. While this works for \( \eta \)-laws that can be formulated on untyped terms, such as \( \eta \)-contraction of functions \( \lambda x : t\ x \to t \) (when \( x \) not free in \( t \)), it does not extend to type-directed \( \eta \)-laws such as extensionality for the unit type. Further, ICC\( \ast \) is not a type theory formulated with a typed equality judgement, which makes it hard to define its models [Miquel 2000]—we wish not to go there, but stay within the framework of Martin-Löf Type Theory [1975].

Now, if the types of compared Curry-style terms are not equal, can they be sufficiently related to give a proper meaning to the algorithmic equality judgement? It has already been observed that for a type-directed equality check the precise type is not necessary, a shape or skeleton is sufficient. The skeleton informs the algorithm whether the terms under comparison are functions, inhabitants of the unit type, or something else, to possibly apply the appropriate \( \eta \)-law. For the Logical Framework (LF), the simplest dependent lambda-calculus, the skeletons are simple types that can be obtained from the original dependent types by erasing the dependencies: dependent function types map to non-dependent ones and indexed data types to simple data types. Harper and Pfenning [2005] present such an equality check for LF which is directed by simple types, and their technique should scale to other type theories that admit dependency erasure.

By large eliminations [Werner 1992] we refer to types computed by case distinction over values; they occur in type theories that feature both universes and data types. In the presence of large

---

3For instance, the types of the Calculus of Constructions erase to F\( \omega \)-types [Geuvers 1994], and the latter could be used to direct the equality check. Lovas and Pfenning [2010] consider also refinement types for logical frameworks which can be erased to simple types.
eliminations, dependency erasure fails, and it is not clear what the skeleton of a dependent type should be. For instance consider the type \((n:\mathbb{N}) \to A \to \cdots \to A \to A\) of \(n\)-ary functions; its shape is dependent on the value of \(n\), thus cannot be determined statically. Thus, the “skeleton” idea is also not directly applicable.

Going beyond the standard syntax-directed equality check, there is a technique that can deal with dynamic \(\eta\)-expansion. It is a type-directed normalization function inspired by normalization-by-evaluation (NbE) that computes \(\eta\)-long normal forms [Berger and Schwichtenberg 1991; Danvy 1999]. We can check the computed normal forms for identity and, thus, decide definitional equality. NbE has proven to be a robust method to decide equality in powerful type theories with non-trivial \(\eta\)-laws. It scales to universes and large eliminations [Abel et al. 2007], topped with singleton types or proof irrelevance [Abel et al. 2011], and even impredicativity [Abel 2010]. At its heart there are reflection \(\uparrow^T\) and reification \(\downarrow^T\) functions directed by type \(T\) and orchestrating just-in-time \(\eta\)-expansion. Reflection \(\uparrow^T x\) maps variables \(x\) into the realm of values of type \(T\) and lets us compute with open terms. Reification \(\downarrow^T a\) takes a value \(a\) of type \(T\) and computes its long normal form. For instance, the normal form of a closed function \(f: U \to T\) would be \(\lambda x. \downarrow^T(f(\uparrow^U x))\), and for its dependently-typed variant \(f: (x: U) \to T[x]\) it would be \(\lambda x. \downarrow^T(f(\uparrow^U x))\).

The central technical observation is that reflection and reification do not need the precise type \(T\), they work the same for any \textit{shape} \(S\) of \(T\). We managed, while not introducing a new syntax for shapes, to define a relation \(T \in S\) on type values stating that type \(S\) qualifies as shape for type \(T\). Shapes unfold dynamically during reflection and reification. For example, when reflecting a variable \(x\) into the polymorphic function type \(\forall i. F i\) where \(F i = \text{Nat} \to \text{Nat}\), we obtain \((\uparrow^{\forall i. F i} x)\) \(i = \uparrow^{F i}(x\ i)\) for size \(i\) and \((\uparrow^{\forall i. F i} x)\) \(j = \uparrow^{F j}(x\ j)\) for size \(j\). The new types \(F i\) and \(F j\) reflect at are no longer equal (and they are not subtypes of each other), but they still have the same shape, \(\text{Nat} \to \text{Nat}\). This means they will still move in lock-step in respect to \(\eta\)-expansion, which is sufficient to prove NbE correct for judgmental equality. We call the enabling property of \(F\) \textit{shape irrelevance}, meaning that for any pair \(i, j\) of legal arguments, \(F i\) and \(F j\) have the same shape. Whenever we form a irrelevant function type \(\forall x: U. T[x]\), we require \(T[x]\) to be shape-irrelevant in \(x\). This is the middle ground between ICC\(^*\), where no restriction is placed on \(T\) but \(\eta\) for unit types is out of reach (at least for the moment), and Pfennig’s [2001] irrelevance modality, adapted to full dependent types by Abel and Scherer [2012], which requires \(T\) to be irrelevant in \(x\) and, thus, has type equality \(T[i] = T[j]\).

For the time being, we do not (and cannot) develop a general theory of shape irrelevance. We confine ourselves to size-irrelevant function types \(\forall i. T[i]\). This relieves us from defining a special shape-irrelevance modality, since all size-indexed types \(T[i]\) \textit{are} shape irrelevant in \(i\), simply because there is no case distinction on size, and sizes appear relevantly only under a sized type constructor such as \(\text{Nat}\). Our technique would not extend to the polymorphic types \(\forall X. B[X]\) of System F. Even though there is no case distinction on types, shape irrelevance of \(B[X]\) fails in general, as \(X\) could appear as a type on the top-level, e.g. in \(B[X] = X \to X\), and then \(B[1]\) and \(B[A \to A]\) would have distinct shapes.

To summarize, this article makes the following novel contributions:

(1) We present the first integration of a dependent type theory with higher-rank size polymorphism. Concretely, we consider a type theory à la Martin-Löf with dependent function types, cumulative universes, subtyping, a judgmental equality with \(\eta\)-laws, a sized type of natural numbers and two size quantifiers: an irrelevant one \((\forall)\) for binding of sizes in irrelevant positions, and a relevant one \((\Pi)\) for binding of sizes in shape-irrelevant positions (Section 3). Judgmental equality features a “Curry-style” rule for irrelevant size application which ignores
the size arguments, and consequently, the corresponding typing rule will also ignore the size argument. (In the following rules, \( a, a', \) and \( b \) stand for arbitrary size expressions.)

\[
\begin{align*}
\Gamma \vdash t &= t' : \forall i. T \\
\Gamma \vdash t &= t' : \Pi i. T \\
\Gamma \vdash t a &= t' a' : T[b/i] \\
\Gamma \vdash t a &= t' a' : T[a/i] \\
\Gamma \vdash t a &= t' a : T[a/i]
\end{align*}
\]

Our substitution theorem distinguishes term- from type-side substitutions.

(2) We adapt normalization-by-evaluation (NbE) to sized types and size quantification and show that it decides judgmental equality (sections 4 and 5). The novel technical tool is a relation \( T ⊏ \sim S \) which relates a type \( T \) to its possible shapes \( S \). This approximation relation allows reflection and reification at size-polymorphic types \( ∀ i. T \). As usual for the meta-theory of Type Theory with large eliminations, the machinery is involved, but we just require the usual two logical relations: First, a PER model to define the semantics of types and prove the completeness of NbE (Section 4). Secondly, a relation between syntax and semantics to prove soundness of NbE (Section 5).

(3) We present an bidirectional type checking algorithm [Coquand 1996] which takes the irrelevant size argument as reliable hint for the type checker (sections 6 and 7). It is complete for normal forms which can be typed with the restricted rule for \( ∀ \)-elimination:

\[
\Gamma \vdash t : ∀ i. T \\
\Gamma \vdash t a : T[a/i]
\]

The algorithm employs the usual lazy reduction for types, i.e., just-in-time weak-head evaluation, in type and sub-type checker [Huet 1989]. In this, it improves on Fridlender and Pagano [2013] which instruments full normalization (NbE) at every step.

This article is accompanied by a prototypical type checker \( \text{S1t} \) which implements the type system and type checking algorithm as described in the remainder of the paper. But before going into the technical details, we will motivate our type system from a practical perspective: reasoning about programs involving sized types in Agda.

2 SIZE IRRELEVANCE IN PRACTICE

In this section, we show how the lack of size irrelevance prevents us from reasoning naturally about programs involving sized types in Type Theory. We focus on Agda, at the time of writing the only mature implementation of Type Theory with an experimental integration of sized types.

The problem of the current implementation of sized types in Agda can be demonstrated by a short example. Consider the type of sized natural numbers.

\[
data \text{Nat} : \text{Size} \rightarrow \text{Set} \text{ where}
\begin{align*}
\text{zero} & : \forall i \rightarrow \text{Nat} (i + 1) \\
\text{suc} & : \forall i \rightarrow \text{Nat} \ i \rightarrow \text{Nat} \ (i + 1)
\end{align*}
\]

The predecessor function is \textit{size preserving}, i.e., the output can be assigned the same upper bound \( i \) as the input. In the code to follow, the dot on the left hand side, preceding \((i + 1)\), marks an \textit{inaccessible} pattern. Its value is determined by the subsequent match on the natural number argument, no actual matching has to be carried out on this argument.

\[
\begin{align*}
\text{pred} & : \forall i \rightarrow \text{Nat} \ i \rightarrow \text{Nat} \ i \\
\text{pred} \ (i + 1) \ (\text{zero} \ i) &= \text{zero} \ i \\
\text{pred} \ (i + 1) \ (\text{suc} \ i \ x) &= x
\end{align*}
\]

Note that in the second clause, we have applied subtyping to cast \( x \) from \( \text{Nat} \ i \) to \( \text{Nat} \ (i + 1) \).
We now define subtraction \( x - y \) on natural numbers, sometimes called the \textit{monus} function, which computes \( \max(0, x - y) \). It is defined by induction on the size \( j \) of the second argument \( y \), while the output is bounded by size \( i \) of the first argument \( x \). The input-output relation of \textit{monus} is needed for a natural implementation of Euclidean division.

There are several ways to implement \textit{monus}, we have chosen a tail-recursive variant which treats the first argument as accumulator. It computes the result by applying the predecessor function \( y \) times to \( x \).

\[
\text{monus} : \forall i \to \text{Nat} \to \forall j \to \text{Nat} \to \text{Nat} \\
\text{monus} \ x \ . (j + 1) \ (\text{zero} \ j) = x \\
\text{monus} \ i \ x \ . (j + 1) \ (\text{suc} \ j \ y) = \text{monus} \ i \ (\text{pred} \ i \ x) \ j \ y
\]

To document subgoals in proof terms, we introduce a mixfix version of the identity function with a visible type argument:

\[
\text{prove}_\text{by}_\text{} : \ (A : \text{Set}) \to A \to A \\
\text{prove} \ A \text{ by } x = x
\]

We now wish to prove that subtracting \( x \) from itself yields 0, by induction on \( x \). The case \( x = 0 \) should be trivial, as \( x - 0 = x \) by definition, hence, \( 0 - 0 = 0 \). As simple proof by reflexivity should suffice. In case \( x + 1 \), the goal \( 0 \equiv (x + 1) - (x + 1) \) should reduce to \( 0 \equiv x - x \), thus, an application of the induction hypothesis should suffice. The following display shows that partial proofs, leaving holes {...!} already filled with the desired proof terms.

\[
\text{monus-diag} : \forall i \to (x : \text{Nat} \ i) \to \text{zero} \ i \equiv \text{monus} \ i \ x \ i \ x \\
\text{monus-diag} \ . (i + 1) \ (\text{zero} \ i) = \text{prove} \ \text{zero} \ i \equiv \text{zero} \ i \ \text{by} \ {! \ \text{refl} \ !} \\
\text{monus-diag} \ . (i + 1) \ (\text{suc} \ i \ x) = \text{prove} \ \text{zero} \ i \equiv \text{monus} \ (i + 1) \ x \ i \ x \ \text{by} \ {! \ \text{monus-diag} \ i \ x \ !}
\]

Unfortunately, in Agda our proof is not accepted, as sizes get in the way. In the first goal, there is a mismatch between size \( \infty \) and size \( i \), the latter coming from the computation of \text{monus} \ (i + 1) \ (\text{zero} \ i) \ (i + 1) \ (\text{zero} \ i). \) In the second goal, there is a mismatch between size \( i + 1 \) in term \text{monus} \ (i + 1) \ x \ i \ x \ of the reduced goal and size \( i \) of the respective term \text{monus} \ i \ x \ i \ x \ from the induction hypothesis we wish to apply.

The proof would go through if Agda ignored sizes where they act as type argument, i.e., in constructors and term-level function applications, but not in types where they act as regular argument, e.g., in \text{Nat} \ i.

The solution we present in this article already works in current Agda,\(^4\) but the implementation is not perfect. Thus, it is hidden under a scarcely documented flag:

\[
\begin{align*}
\text{{-# OPTIONS --experimental-irrelevance #-}}
\end{align*}
\]

We mark the size argument of \text{Nat} as \textit{shape irrelevant} by preceding the binder with two dots. In a future implementation, we could treat all data type parameters as shape irrelevant by default. In the types of the constructors, we mark argument \( i \) as \textit{irrelevant} by prefixing the binder with a single dot. This is sound because \( i \) occurs in subsequent parts of the type only in shape-irrelevant positions.

\[
\text{data Nat} : .. (i : \text{Size}) \to \text{Set} \ \text{where} \\
\text{zero} : \forall . i \to \text{Nat} \ (i + 1) \\
\text{suc} : \forall . i \to \text{Nat} \ i \to \text{Nat} \ (i + 1)
\]

\(^4\)https://github.com/agda/agda, development version of 2017-02-27.
Similarly, “type” argument \( i \) to \texttt{pred} is irrelevant. Agda checks that it only occurs shape-irrelevantly in the type and irrelevantly in the term. The latter is the case since \( i \) is also an irrelevant argument to the constructors \texttt{zero} and \texttt{suc}; otherwise, we would get a type error.

\[
\begin{align*}
\texttt{pred} & : \forall . i \to \texttt{Nat} \; i \to \texttt{Nat} \; i \\
\texttt{pred} \; (i + 1) \; (\texttt{zero} \; i) & = \texttt{zero} \; i \\
\texttt{pred} \; (i + 1) \; (\texttt{suc} \; i \; x) & = x
\end{align*}
\]

The two size arguments \( i \) and \( j \) to \texttt{monus} are also irrelevant. In this case, type checking succeeds since the size argument to \texttt{pred} has been declared irrelevant.

\[
\begin{align*}
\texttt{monus} & : \forall . i \to \texttt{Nat} \; i \to \forall . j \to \texttt{Nat} \; j \to \texttt{Nat} \; i \\
\texttt{monus} \; i \; x \; (j + 1) \; (\texttt{zero} \; j) & = x \\
\texttt{monus} \; i \; x \; (j + 1) \; (\texttt{suc} \; j \; y) & = \texttt{monus} \; i \; (\texttt{pred} \; i \; x) \; j \; y
\end{align*}
\]

Now, with sizes being ignored in the involved terms, we can complete the proof of our lemma:

\[
\begin{align*}
\texttt{monus-diag} : \forall . i \to (x : \texttt{Nat} \; i) \to \texttt{zero} \; \infty & \equiv \texttt{monus} \; i \; x \; i \; x \\
\texttt{monus-diag} \; (i + 1) \; (\texttt{zero} \; i) & = \texttt{prove} \; \texttt{zero} \; \infty \equiv \texttt{zero} \; i & & \text{by refl} \\
\texttt{monus-diag} \; (i + 1) \; (\texttt{suc} \; i \; x) & = \texttt{prove} \; \texttt{zero} \; \infty \equiv \texttt{monus} \; (i + 1) \; x \; i \; x & & \text{by \texttt{monus-diag} \; i \; x}
\end{align*}
\]

### 3 A TYPE SYSTEM WITH IRRELEVANT SIZE APPLICATION

In this section, we give the syntax and the declarative typing, equality, and subtyping judgements. The typing relation \( \Gamma \vdash t : T \) will not be decidable; instead, we present algorithmic typing \( \Gamma \vdash t \equiv T \) in Section 7. However, equality and subtyping will be decidable for well-formed input, see sections 4–6.

We present our type theory as (domain-free) pure type system [Barendregt 1991] with extra structure. The sorts \( s \) are drawn from an infinite predicative hierarchy of universes \( \texttt{Set}_\ell \) for \( \ell \in \mathbb{N} \). Universes provide us with polymorphism and the capability to define types by recursion on values. Whether we have just two universes \( \texttt{Set}_0 \) and \( \texttt{Set}_1 \) or infinitely many, does not matter for the technical difficulty of the meta theory. The present setup have the advantage that every sort has again a sort since \( \texttt{Set}_\ell : \texttt{Set}_{\ell+1} \), thus, we do not have to introduce a separate judgement \( \Gamma \vdash T \) for well-formedness of types, we can define it as \( \exists s. \; \Gamma \vdash T : s \).

\[
\begin{array}{lcl}
\text{Sort} & \ni & s \\
\text{Ann} & \ni & \star \\
\text{Exp} & \ni & t, u, T, U \\
\text{Whnf} & \ni & w, W \\
\text{Data} & \ni & c \\
\text{NeExp} & \ni & n \\
\text{Elim} & \ni & e \\
\text{SizeExp} & \ni & a, b \\
\text{Cxt} & \ni & \Gamma, \Delta \\
\text{Subst} & \ni & \eta, \rho, \sigma, \tau, \xi
\end{array}
\]

\[
\begin{array}{ll}
\text{ ::= } & \text{Set}_\ell (\ell \in \mathbb{N}) \\
\text{ ::= } & \div | : \\
\text{ ::= } & w | t \; e \\
\text{ ::= } & n \; | \; s \; | \; \text{Size} \; | \; \Pi^* U \; T \; | \; \lambda t \; | \; \text{Nat} \; a \; | \; c \\
\text{ ::= } & \text{zero}(a) \; | \; \text{suc}(a) \; t \\
\text{ ::= } & \nu_i \; | \; n \; e \\
\text{ ::= } & t \; | \; a \; | \; \langle a \rangle \; | \; \text{case}_\ell T \; t_z \; t_s \; | \; \text{fix}_\ell T \; t \\
\text{ ::= } & \infty \; | \; o \; | \; \nu_i \; + \; o \; (o \in \mathbb{N}) \\
\text{ ::= } & () \; | \; \Gamma. T \; | \; \Gamma. \; \text{Size} \\
\text{ ::= } & () \; | \; (\sigma, t)
\end{array}
\]

| \text{Fig. 1. Syntax.} |

For the expression syntax (see Fig. 1), we use de Bruijn [1972] indices \( \nu_i \) to represent variables. The index \( i \in \mathbb{N} \) points to the \( i \)th enclosing binder of variable \( \nu_i \). Binders are lambda abstraction \( \lambda t \) and
dependent function types $\Pi^* U T$, which bind the 0th index in $t$ and $T$, resp. For instance, the term $\lambda x. x (\lambda y. y)$ with named variables $x, y$ has de Bruijn representation $\lambda. v_0 (\lambda. v_0) (\lambda. v_1 v_0)$.

The notation $\Pi^* U T$ is an umbrella for three kinds of function types, where $\star \in \{\vdash, \cdot\}$ is a relevance annotation borrowed from Pfenning [2001]. $\Pi U T$ is the ordinary dependent function type, $\Pi \text{Size } T$ is relevant size quantification, and $\Pi^* \text{Size } T$ is irrelevant size quantification. We omit the $\vdash, \cdot$-markers from $\Pi$ by default (and also in contexts $\Gamma$) and write $\forall T$ for $\Pi^* \text{Size } T$. Examples for relevant size quantification $\Pi \text{Size } T$ are $\Pi \text{Size } Set_0$ and $\Pi \text{Size } \Pi (\text{Nat } v_0) Set_0$.

In a syntax with named variables and non-dependent function type they could be written as $\Pi \rightarrow Set_0$ and $(z: \text{Size}) \rightarrow \Pi \text{Nat } z \rightarrow Set_0$, resp. An instance of irrelevant quantification $\forall T$ would be $\forall. \Pi (\text{Nat } v_0) (\text{Nat } v_1)$ which is $\forall z. \Pi \text{Nat } z \rightarrow \text{Nat } z$ in a named syntax. Herein, $\Pi \text{Nat } z$ denotes the type of natural numbers below $z$. The expression Size is a possible instance of $U$ in $\Pi^* U T$, or a possible type of a variable in a typing context $\Gamma$, but not a first-class type, i.e., we cannot construct our own functions on sizes.

Canonical natural numbers $c$ are constructed by zero$(a)$ and suc$(a)t$. A size expression $a$ is either a constant $o \in \mathbb{N}$, a variable $v_i + o$ possibly with increment $o$, or the limit ordinal $\omega$ which stands for $\omega$. The size argument $a$ in the constructors zero and suc is a suggestion for the type checker but bears no semantic significance. For example, in the declarative typing presented here, we can have $\vdash \text{zero}(5) : \text{Nat } 1$. In the algorithmic typing however, $\vdash \text{zero}(5) \equiv \text{Nat } 1$ will be an error. Note, however, that $\vdash \text{zero}(a) : \text{Nat } 0$ is impossible for any $a$, as zero is not strictly below 0 (when both term and size are interpreted as natural numbers).

Regular application $tu$, relevant size application $ta$, and irrelevant size application $t\langle a\rangle$ eliminate functions $t$ and are subsumed under the form $te$ with $e ::= u \mid a \mid \langle a \rangle$. We have two further eliminations, which make sense when $t$ stands for a natural number. These are case distinction $e = \text{case}_\ell T t_1 t_2$ and recursive function application $e' = \text{fix}_\ell T' t'$. Application of case distinction zero$(a)e$ will reduce to the zero-branch $t_2$, and application (suc$(a)t$) to the instantiation $t_1 t$ of the successor branch. The type annotation $T$ in case allows us to infer the type of the whole case statement $t e$ as $T t$. The function call $c e'$ for a canonical number $c$ and elimination $e' = \text{fix}_\ell T' t' \rightarrow \text{Nat } z = \text{Nat } 1 \rightarrow \Pi \text{Nat } 1$ of zero$(a)$.

As already done in some examples, we may use a named dependent function type notation as syntactic sugar for the corresponding de Bruijn representation. For instance, $(z: \text{Size}) \rightarrow \Pi \text{Nat } z \rightarrow \text{Set}_\ell$ is sugar for $\Pi \text{Size } \Pi (\text{Nat } v_0) \text{Set}_\ell$. We abbreviate this type by $\text{FixK } \ell$ and let $\text{FixT } T$ stand for $\forall z. ((x : \text{Nat } z) \rightarrow T z x) \rightarrow (x : \text{Nat } (z + 1)) \rightarrow T (z + 1) x$. Similarly to for $\Pi$, we use named
Inductively defined judgements (mutual).
\[ \Gamma \vdash \text{context } \Gamma \text{ is well-formed} \]
\[ \Gamma(i) = *T \quad \text{in context } \Gamma, \text{ de Bruijn index } i \text{ has type } T \text{ and annotation } \star \]
\[ \Gamma \vdash a : \text{Size} \quad \text{in context } \Gamma, \text{ size expression } a \text{ is well-formed} \]
\[ \Gamma \vdash t : T \quad \text{in context } \Gamma, \text{ term } t \text{ has type } T \]
\[ \Gamma \vdash t = t' : T \quad \text{in context } \Gamma, \text{ terms } t \text{ and } t' \text{ are equal of type } T \]
\[ \Gamma \vdash T \leq T' \quad \text{in context } \Gamma, \text{ type } T \text{ is a subtype of } T' \]
\[ \Gamma \vdash \sigma : \Delta \quad \sigma \text{ is a valid substitution for } \Delta \]
\[ \Gamma \vdash \sigma = \sigma' : \tau : \Delta \quad \sigma/\sigma'/\tau \text{ are a equal term/term/type-level substitutions for } \Delta \]

Derived judgements.
\[ \Gamma \vdash T \quad \iff \quad \Gamma \vdash T : s \quad \text{for some } s \]
\[ \Gamma \vdash T = T' \quad \iff \quad \Gamma \vdash T = T' : s \quad \text{for some } s \]
\[ \Gamma \vdash a : \text{Size} \quad \iff \quad \Gamma \vdash a : \text{Size} \text{ and } \Gamma \vdash b : \text{Size} \text{ and } a = b \]
\[ \Gamma \vdash T : \text{Adm } \ell \quad \iff \quad \Gamma \vdash T : \text{FixK } \ell \text{ and } \Gamma : \text{Size } x : \text{Nat } z \vdash T z x \leq T \circ x \]
\[ \xi : \Gamma \leq \Delta \quad \iff \quad \Gamma \vdash \xi : \Delta \text{ and } \xi = \uparrow^k_m \text{ with } m = |\Delta| \text{ and } k = |\Gamma| - m \]

Fig. 2. Judgements.

In typing contexts \( \Gamma \), we distinguish relevant (\( \cdot \)) and irrelevant (\( \vdash \)) bindings. When type checking a variable, it needs to be bound in the context relevantly. However, when entering an irrelevant position, for instance when checking size \( a \) in term \( \text{succ}(a)t \) we declare previously irrelevant variables as relevant. This operation on the context has been coined resurrection by Pfenning [2001]; formally \( \Gamma^\bowtie \) removes the “\( \vdash \)”-markers from all bindings in \( \Gamma \), i.e., replaces them by “\( \cdot \)”-markers. Note that, trivially, resurrection is idempotent: \( \Gamma^{\bowtie\bowtie} = \Gamma^{\bowtie} \).

Size increment \( \overline{a + o'} \) for \( o' \in \mathbb{N} \) extends addition by \( \infty + o' = \infty \) and \( (v_i + o) + o' = v_i + (o + o') \). Sizes are partially ordered; size comparison \( \overline{a \leq b} \) holds as expected if either \( b = \infty \) or \( a \leq o' \) where either \( a = 0 \) and \( b = o' \) or \( a \in \{o, v_i + o\} \) and \( b = v_i + o' \). Different size variables are incomparable.

Fig. 2 lists the inductive and derived judgements of our type theory and figures 3 and 4 the inference rules. We have \( \boxed{\text{boxed}} \) the rules dealing with irrelevant size application. Fig. 5 adds the typing and equality rules for case distinction and recursion on natural numbers. Judgement \( \Gamma \vdash T : \text{Adm } \ell \) characterizes the valid type annotations \( T \) in recursion \( \text{fix}_\ell T t \). The type constructor \( T \) has to be monotone in the size argument; this is a technical condition for type-based termination [Barthe et al. 2004]. We will make use of it in Section 4.7. We write \( \otimes : J \) to express that \( \otimes \) is a derivation of judgement \( J \).

In the typing judgement \( \Gamma \vdash t : T \), the term \( t \) is in scope of \( \Gamma \), i.e., may not mention irrelevant variables in relevant positions. However, the type \( T \) is in scope of the resurrected context \( \Gamma^\bowtie \), hence, can mention all variables declared in \( \Gamma \). The other judgements are organized similarly. To understand this distinction, consider judgement \( z \vdash \text{Size} + \text{Nat } z \). This would mean that \( z \) is irrelevant in \( \text{Nat } z \).
Fig. 3. Typing, subtyping, and substitution judgements.
Computation rules.

\[
\begin{align*}
\Gamma, U \vdash t : T & \quad \Gamma \vdash u : U \\
\Gamma \vdash (\lambda t) u = t[u] : T[u] & \quad \Gamma, \text{Size} \vdash t : T \\
\Gamma \vdash t \vdash a : \text{Size} & \quad \Gamma, \text{Size} \vdash t : T \\
\Gamma \vdash (\lambda t) a = t[a] : T[a] & \quad \Gamma, \text{Size} \vdash t : T \\
\Gamma \vdash (\lambda t) (a) = t[a] : T[b] & \\
\end{align*}
\]

Extensionality rules.

\[
\begin{align*}
\Gamma \vdash t : \Pi U T & \quad \Gamma \vdash t = \lambda x : \Pi U T \\
\Gamma \vdash t = \lambda x : \Pi U T & \\
\Gamma \vdash \forall T & \quad \Gamma, \text{Size} \vdash a : \text{Size} \\
\Gamma, \text{Size} \vdash T = T : s & \\
\end{align*}
\]

Congruence rules.

\[
\begin{align*}
\vdash \Gamma & \quad \vdash \Gamma \vdash a : \text{Size} \\
\vdash \Gamma \vdash U = U' : s & \quad \vdash \Gamma \vdash U \vdash T = T' : s \\
\vdash \Gamma \vdash \Pi U T = \Pi U' T' : s & \quad \vdash \Gamma \vdash \Pi \text{Size} T = \Pi \text{Size} T' : s \\
\vdash \Gamma \vdash \Gamma(i) = 'T & \quad \vdash \Gamma, \text{Size} \vdash t = t' : T \\
\vdash \Gamma \vdash v_i = v_i : T & \quad \vdash \Gamma \vdash \lambda t = \lambda t' : \Pi \text{Size} T \\
\vdash \Gamma \vdash t = t' : \Pi \text{Size} T & \quad \vdash \Gamma \vdash t u = t' u' : T[u] \\
\vdash \Gamma \vdash t = t' : \Pi \text{Size} T & \quad \vdash \Gamma \vdash t a = t' a : T[a] \\
\vdash \Gamma \vdash a : \text{Size} & \quad \vdash \Gamma \vdash (t') a = t'(a') : T[b] \\
\Gamma \vdash a, a', b : \text{Size} & \quad \Gamma \vdash a, a', b : \text{Size} \\
\Gamma \vdash \text{zero}(a) = \text{zero}(a') : \text{Nat}(b + 1) & \quad \Gamma \vdash \text{suc}(a) t = \text{suc}(a') t' : \text{Nat}(b + 1) \\
\Gamma \vdash t = t' : T & \quad \Gamma \vdash t' : T \leq T' \\
\Gamma \vdash t = t' : T \\
\end{align*}
\]

Equivalence rules.

\[
\begin{align*}
\Gamma \vdash t : T & \quad \Gamma \vdash t : T \\
\Gamma \vdash t = t' \vdash t : T & \quad \Gamma \vdash t = t' \vdash t : T \\
\Gamma \vdash t_1 = t_2 : T & \quad \Gamma \vdash t_1 = t_3 : T \\
\Gamma \vdash t_1 = t_3 : T \\
\end{align*}
\]

Fig. 4. Definitional equality (implies \(\Gamma, a \vdash a = a' \text{ and } \Gamma, t, t' \vdash t \text{ and } \Gamma, \text{Size} \vdash t = t' \).)

and thus, \(\vdash \text{Nat} \ a = \text{Nat} \ a' \) for all sizes \(\Gamma, a \vdash a = a'\). But this is exactly wrong! However, judgement \(z \vdash \text{Size} \vdash \text{zero}(z) : \text{Nat}(z + 1) \) is fine, it implies \(\vdash \text{zero}(a) = \text{zero}(a') : \text{Nat}(b + 1) \) for all \(\Gamma, a, a', b : \text{Size} \).

Our substitution theorem needs to reflect the distinct scope of things left of the colon vs. things right of the colon. In the last example we have applied the substitution triple \(\vdash [a] = [a'] = [b] : (z \vdash \text{Size}) \) to judgement \(z \vdash \text{Size} \vdash \text{zero}(z) : \text{Nat}(z + 1) \). The first two substitutions apply to the term side while the third substitution applies to the type side. The fact that we replace an irrelevant variable \(z\) allows \(a, a', b\) to refer to irrelevant variables from \(\Gamma\), thus, they are in scope of \(\Gamma, a\).

Typing requires from annotations \(\langle a \rangle\) in a term only that they are well-scoped size expressions, i.e., just mention relevant size variables. Let \(t^\text{erasured}\) denote the erasure of term \(t\), meaning that we replace all annotations \(\langle a \rangle\) in \(t\) by \(\langle \infty \rangle\). Let \(t \approx u\) relate terms that only differ in their annotations, i.e., \(t \approx u : \iff \ t^\text{erasured} = u^\text{erasured}\). Erasure does not change the term modulo judgmental equality:
We should remark here that we have neither type unicity nor principal types property of substitutions: only replacement for irrelevant sizes may refer to irrelevant size variables.

As such, it is comparable to an extended context or a context where types have been replaced of the following lemma. A resurrected context is more permissive, as it brings more variable into scope. In the following, we list syntactic properties of our judgements. To this end, let $J$ match a part of a judgement.

**Lemma 3.1 (Erasure and similarity).**

1. If $\Gamma \vdash t : T$ then $\Gamma \vdash t = t^{\omega} : T$.
2. If $\Gamma \vdash t, u : T$ and $t \approx u$ then $\Gamma \vdash t = u : T$.

We should remark here that we have *neither type unicity nor principal types* due to the irrelevant size application rule. In the following, we list syntactic properties of our judgements. To this end, let $J$ match a part of a judgement.

**Lemma 3.2 (Context well-formedness).**

1. If $\vdash \Gamma, \Delta$ then $\vdash \Gamma$
2. If $\vdash J$ then $\vdash \Gamma$.

All types in a context are considered in the resurrected context, which justifies the first statement of the following lemma. A resurrected context is more permissive, as it brings more variable into scope. As such, it is comparable to an extended context or a context where types have been replaced by subtypes. This intuition accounts for the remaining statements but (4). The latter is a defining property of substitutions: only replacement for irrelevant sizes may refer to irrelevant size variables.

**Lemma 3.3 (Resurrection).**
Normalization by Evaluation for Sized Dependent Types

(1) \( \vdash \Gamma \iff \vdash \Gamma^\circ \). Then \( \Gamma^\circ \vdash \text{id} : \Gamma \), which can be written \( \text{id} : \Gamma^\circ \leq \Gamma \).

(2) If \( \Gamma \vdash J \) then \( \Gamma^\circ \vdash J \).

(3) If \( \Gamma \vdash \sigma : \Delta^\circ \) then \( \Gamma \vdash \sigma : \Delta \).

(4) If \( \Gamma \vdash \sigma : \Delta \) then \( \Gamma^\circ \vdash \sigma : \Delta^\circ \).

**Lemma 3.4 (Substitution).**

(1) If \( \Gamma \vdash \sigma : \Delta \) and \( \Delta \vdash J \) then \( \Gamma \vdash J \sigma \).

(2) If \( \Gamma \vdash \sigma = \sigma' \equiv \tau : \Delta \) and \( \Delta \vdash t : T \) then \( \Gamma \vdash t \sigma : T\tau \) and \( \Gamma \vdash t \sigma' : T\tau \).

**Lemma 3.5 (Specific substitutions).**

(1) If \( \Gamma, \Delta \vdash \tau \equiv \nu : \Delta \) then \( \Gamma \vdash \nu : \Delta \).

(2) If \( \Gamma \vdash \nu : \Delta \) then \( \Gamma \vdash \id : \Delta \).

(3) If \( \Gamma \vdash u : U \) then \( \Gamma \vdash [u] : \Gamma U \).

The relation \( \vdash \sigma = \sigma' \equiv \tau : \Delta \) is a partial equivalence relation (PER) on term-side substitutions \( \sigma, \sigma' \). Note that usually we cannot resurrect this judgement to \( \Gamma^\circ \vdash \sigma = \sigma' \equiv \tau : \Delta^\circ \). For instance, \( z_1 Size \equiv z_2 Size \vdash [z_1] = [z_2] \equiv [\infty] : z Size \) holds but \( z_1 Size \equiv z_2 Size \vdash [z_1] = [z_2] \equiv [\infty] : z Size \) clearly not.

**Lemma 3.6 (Substitution equality).**

(1) Conversion: If \( \Gamma \vdash \sigma = \sigma' \equiv \tau_1 : \Delta \) and \( \Gamma^\circ \vdash \tau_1 = \tau_2 \equiv \tau : \Delta^\circ \) then \( \Gamma \vdash \sigma = \sigma' \equiv \tau_2 : \Delta \).

(2) Reflexivity: If \( \Gamma \vdash \sigma \equiv \sigma \) then \( \Gamma \vdash \sigma = \sigma \).

(3) Symmetry: If \( \Gamma \vdash \sigma \equiv \tau : \Delta \) then \( \Gamma \vdash \sigma = \tau : \Delta \).

(4) Transitivity: If \( \Gamma \vdash \sigma_1 \equiv \sigma_2 \equiv \tau : \Delta \) and \( \Gamma \vdash \sigma_2 = \tau : \Delta \) then \( \Gamma \vdash \sigma_1 = \sigma_2 = \tau : \Delta \).

(5) Functionality: Let \( \Gamma \vdash \sigma = \sigma' \equiv \tau : \Delta \).

(a) If \( \Delta \vdash t : T \) then \( \Gamma \vdash t \sigma = t \sigma' : T\tau \).

(b) If \( \Delta \vdash t = t' : T \) then \( \Gamma \vdash t \sigma = t' \sigma' : T\tau \).

(c) Corollary: If \( \Delta \vdash T \leq T' \) then \( \Gamma \vdash T\sigma \leq T\sigma' \).

**Lemma 3.7 (Inversion of typing).**

(1) If \( \Gamma \vdash v_1 : T' \) then \( \Gamma (i) = T \) and \( \Gamma^\circ \vdash T \leq T' \) for some \( T \).

(2) If \( \Gamma \vdash \lambda t : T' \) then \( \Gamma \vdash \Pi U T \leq T' \) for some \( U, T \) or \( \Gamma \vdash \Pi^* \) Size \vdash t : T and \( \Gamma^\circ \vdash \Pi^* \) Size \( T \leq T' \) for some \( T \).

(3) If \( \Gamma \vdash t u : T' \) then \( \Gamma \vdash t : \Pi U T \) and \( \Gamma \vdash u : U \) and \( \Gamma^\circ \vdash T[u] \leq T' \) for some \( U, T \).

(4) If \( \Gamma \vdash t a : T' \) then \( \Gamma \vdash t : \Pi \) Size \( T \) and \( \Gamma \vdash a : \) Size \( \) and \( \Gamma^\circ \vdash T[a] \leq T' \) for some \( T \).

(5) If \( \Gamma \vdash t \langle a \rangle : T' \) then \( \Gamma \vdash t \vdash \forall T \) and \( \Gamma^\circ \vdash a, b : \) Size \( \) and \( \Gamma^\circ \vdash T[b] \leq T' \) for some \( T, b \).

(6) Analogous properties for the remaining term and type constructors.

**Proof.** Each by induction on the typing derivation, gathering applications of the conversion rule via transitivity of subtyping.

\[ \square \]

**4 SEMANTICS AND COMPLETEENESS OF NORMALIZATION BY EVALUATION**

In this section we present an operational semantics of our language, define the NbE algorithm, construct a PER model, and demonstrate that NbE is complete for definitional equality, i.e., if \( \Gamma \vdash t = t' : T \), then \( t \) and \( t' \) have the same normal form up to annotations.

\[
\begin{align*}
\text{Ne} \ni m &::= n \mid m v \mid m a \mid m(a) \mid m \text{case} \ell V v \mid m \text{fix} \ell V v & \text{neutral n.f.} \\
\text{NF} \ni v &::= m \mid \lambda v \mid \text{zero}(a) \mid \text{succ}(a)v \mid \text{Set} \ell \mid \text{Nat} a \mid \Pi V u V i \mid \Pi^* \text{Size} V & \text{normal form}
\end{align*}
\]

For the operational semantics, instead of defining a separate language of values, we extend the syntax of expressions by de Bruijn levels \( x_k \) to be used as generic values (unknowns), and type
notations $\uparrow^A n$ and $\downarrow^A t$ for lazy realizations of the reflection and reification operations of NbE.

Terms are expressions that do not contain these new expression forms. Values $f, g, A, B, F \in D$ are expressions with no free de Bruijn indices, where each neutral $n$ is under a reflection marker $\uparrow^A n$. The types $A$ that direct reflection $\uparrow^A n$ and reification $\downarrow^A f$ also live in the value world.

De Bruijn levels are the mirror images of de Bruijn indices. While de Bruijn indices index the context from the right, i.e., $v_0$ refers the last type that entered the context, de Bruijn levels index it from the left, i.e., $x_0$ refers to the first type in the context. This way, de Bruijn levels are stable under context extensions, and suitable to represent unknowns.

Size values $\alpha, \beta \in \text{Size}$ are size expressions that use de Bruijn levels instead of de Bruijn indices. Comparison of size values $\alpha \leq \beta$ is analogous to comparison of size terms $a \leq b$. In the following, we will reuse letter $a$ for a value if it cannot be confused for a size term.

Finally, we identify two expression classes for NbE. Neutrals $n \in \text{DNe}$ are the ones that will appear in values under the reflection marker $\uparrow^A$. Reified values $d \in \text{DNf}$ are values under a reification marker $\downarrow^A$.

Figure 6, adapted from Abel [2013] summarized the syntactic categories and main operations involved in NbE in what is called locally nameless style. The red path $\text{Exp} \rightarrow D \rightarrow D_{\text{nf}} \rightarrow \text{Nf}$ decomposes $\beta\eta$-normalization into three steps.

(1) First, we close the term $t$ with an environment $\eta$ that maps the free de Bruijn indices of $t$ to reflected de Bruijn levels. Reflection of de Bruijn levels follows the blue path $\text{Level} \rightarrow \text{DNe} \rightarrow D$: Levels embed via constructor $x$ into semantic neutrals $\text{DNe}$ which are labeled with their type $A \in D$ to become an element $\uparrow^A x_j \in D$.

(2) Then, we label value $t\eta \in D$ with its type $A$ to obtain $\downarrow^A t\eta \in \text{DNf}$. 

(3) Finally, read back \( R_k \downarrow^\lambda \eta \) produces a long normal form \( \nu \in \text{Nf} \), converting de Bruijn levels back to indices. Herein, \( k \) should be the length of the context the original term \( t \) lived in. If this is the case, each de Bruijn level encountered during read back is below \( k \) and can be safely converted to a de Bruijn index.

4.1 Weak Head Reduction

We define the operational semantics of our language by the weak head evaluation relation \( t \downarrow_w \) which is defined on expressions, thus works on values as well as on terms. It is defined mutually with auxiliary relation \( \rightarrow_{\text{w}} \) stating that weak head normal form \( w \) is eliminated by \( e \) into weak head normal form \( w' \).

\[
\begin{align*}
\frac{t[u] \downarrow w}{(\lambda t) @ u \downarrow w} & \quad \frac{t[\alpha] \downarrow w}{(\lambda t) @ \alpha \downarrow w} \\
\frac{t[\alpha] \downarrow w}{t \downarrow w \quad w \downarrow w} & \quad \frac{t \downarrow w \quad w @ e \downarrow w'}{t \downarrow w \quad w' \downarrow w'} \\
\frac{t_z \downarrow w}{t_z \downarrow w} & \quad \frac{t \downarrow w \quad (\text{zero}(\alpha)) \rightarrow_{\text{w}} \text{case}_t \, t_z \downarrow w}{t \downarrow w \quad (\text{zero}(\alpha)) \rightarrow_{\text{w}} \text{case}_t \, t_z \downarrow w}
\end{align*}
\]

For NbE, we add evaluation rules that deal with elimination of delayed reflection:

\[
\begin{align*}
\frac{A' \downarrow \Pi AB}{(\uparrow^\lambda \alpha) @ u \downarrow \uparrow^B \alpha | (\downarrow^\lambda u)} & \quad \frac{A \downarrow \Pi \text{Size } B}{(\uparrow^\lambda \alpha) @ \alpha \downarrow \uparrow^B \alpha | (\downarrow^\lambda \alpha)} & \quad \frac{A \downarrow \forall B}{(\uparrow^\lambda \alpha) @ \alpha \downarrow \uparrow^B \alpha | (\downarrow^\lambda \alpha)}
\end{align*}
\]

\[
\begin{align*}
(\uparrow^\lambda n) \rightarrow_{\text{w}} \text{case}_t \, \beta f_x f_s & \quad (\uparrow^\lambda n) \rightarrow_{\text{w}} \text{case}_t \, \beta f_x f_s \\
(\uparrow^\lambda n) \rightarrow_{\text{w}} \text{fix}_t \, \beta f_x f_s & \quad (\uparrow^\lambda n) \rightarrow_{\text{w}} \text{fix}_t \, \beta f_x f_s
\end{align*}
\]

4.2 Read Back

The read back phase of NbE [Grégoire and Leroy 2002] transforms a reified value \( d \) into a normal form \( \nu \). It is specified via an inductively defined relation \( R_k \downarrow^v \) and several auxiliary relations. The number \( k \), will be instantiated by the length of the context \( \Gamma \) later. It allows us to transform a de Bruijn level \( l \) into a de Bruijn index \( i \), via the law \( i + l + 1 = k \). At this point, we do not ensure that the \( k \) is large enough to accommodate the de Bruijn levels in \( d \). Levels \( l \geq k \) which are to big will simply be mapped to de Bruijn index 0. The correct \( k \) is later ensured by our logical relation (Section 5). Even though read back operates on values in practice, formally it is defined on expressions.

\[
R_k \downarrow^v \nu \quad \text{Read back reified value } d.
\]

\[
\begin{align*}
U \downarrow s & \quad R_k \downarrow^U T \downarrow V & U \downarrow \text{Nat } \alpha & \quad R_k \downarrow^U u \downarrow v & U \downarrow N & \quad R_k \downarrow^U u \downarrow m \\
R_k \downarrow^U T \downarrow V & \quad R_k \downarrow^U u \downarrow v & U \downarrow N & \quad R_k \downarrow^U u \downarrow m \\
U \downarrow \Pi AB & \quad R_{k+1} \downarrow^B \uparrow^\lambda \alpha | (\downarrow^\lambda \alpha) & U \downarrow \forall B & \quad R_{k+1} \downarrow^B \uparrow^\lambda \alpha | (\downarrow^\lambda \alpha)
\end{align*}
\]

\[
\begin{align*}
R_{k+1} \downarrow^B \uparrow^\lambda \alpha | (\downarrow^\lambda \alpha) & \quad R_{k+1} \downarrow^B \uparrow^\lambda \alpha | (\downarrow^\lambda \alpha)
\end{align*}
\]

Read back neutrals under annotation. (The annotation is ignored.)

\[ R_{k}^{\text{ne}} n \not\rightarrow m \quad R_{k}^{\text{up}} t \not\rightarrow m \]

Read back natural number value.

\[ R_{k}^{\text{up}} t \not\rightarrow m \quad t \not\rightarrow \text{zero}(\alpha) \quad R_{k}^{\text{size}} \alpha \not\rightarrow a \quad t \not\rightarrow \text{suc}(\alpha) u \quad R_{k}^{\text{size}} \alpha \not\rightarrow a \quad R_{k}^{\text{nat}} u \not\rightarrow v \]

Read back size value.

\[ R_{k}^{\text{size}} \alpha \not\rightarrow a \]

Read back size value \( \alpha \).

\[ R_{k}^{\text{nat}} t \not\rightarrow v \quad R_{k}^{\text{up}} t \not\rightarrow m \quad R_{k}^{\text{nat}} t \not\rightarrow m \]

Read back unreflected neutral.

\[ R_{k}^{\text{lim}} e_{i} \not\rightarrow e_{i}^{v} \quad \text{for all } i \quad R_{k}^{\text{lim}} d \not\rightarrow v \quad R_{k}^{\text{lim}} \alpha \not\rightarrow b \quad R_{k}^{\text{lim}} \alpha \not\rightarrow b \]

Read back type value.

\[ R_{k}^{\text{lim}} (\text{case}_{\ell} D d z d_{s}) \not\rightarrow \text{case}_{\ell} V v_{z} v_{s} \quad R_{k}^{\text{lim}} (\text{fix}_{\ell} D d) \not\rightarrow \text{fix}_{\ell} V v \]

\[ R_{k}^{\text{lim}} T \not\rightarrow V \]

4.3 Partial Equivalence Relations

A type \( T \) will be interpreted as a partial equivalence relation (PER) \( \mathcal{A} \) on terms, i.e., a relation which is symmetric and transitive. The domain \( \text{dom}(\mathcal{A}) \) of the relation can be thought of as the set of terms which denotes the extension of the type; on \( \text{dom}(\mathcal{A}) = \{ a \mid \exists a'. \ (a, a') \in \mathcal{A} \} \) the relation \( \mathcal{A} \) is in fact an equivalence relation. We write \( a = a' \in \mathcal{A} \) for relatedness in \( \mathcal{A} \) and \( a \in \mathcal{A} \) if \( a \in \text{dom}(\mathcal{A}) \).

The PERs \( \mathcal{N}e \) and \( \mathcal{N}f \) characterize (neutral) normalizing values. For instance, two values \( n \) and \( n' \) are related in \( \mathcal{N}e \) if at any \( k \in \mathbb{N} \) they can be read back to neutral normal forms \( m \) and \( m' \) which are identical up to annotations.

\[
\begin{align*}
\frac{n = n' \in \mathcal{N}e}{R_{k}^{\text{ne}} n \not\rightarrow m \quad R_{k}^{\text{ne}} n' \not\rightarrow m' \quad \text{and} \quad m \approx m' \quad \text{for all } k} \\
\frac{d = d' \in \mathcal{N}f}{R_{k}^{\text{lim}} d \not\rightarrow v \quad R_{k}^{\text{lim}} d' \not\rightarrow v' \quad \text{and} \quad v \approx v' \quad \text{for all } k} \\
\frac{e = e' \in \mathcal{E}lim}{R_{k}^{\text{lim}} e \not\rightarrow e_{v} \quad R_{k}^{\text{lim}} e' \not\rightarrow e'_{v} \quad \text{and} \quad e_{v} \approx e'_{v} \quad \text{for all } k} \\
\frac{A = A' \in \mathcal{T}y}{R_{k}^{\text{lim}} A \not\rightarrow V \quad R_{k}^{\text{lim}} A' \not\rightarrow V' \quad \text{and} \quad V \approx V' \quad \text{for all } k}
\end{align*}
\]

Once we have established useful closure properties of these PERs, they abstract most of the reasoning about the read-back relation from our proofs. This idea is due to Coquand [Abel et al. 2009].

LEMMA 4.1 (Closure properties of \( N_e \)).
(1) \( x_k = x_k \in N_e \).
(2) If \( n = n' \in N_e \) and \( e = e' \in \mathcal{E}\text{lim} \) then \( n e = n' e' \in N_e \).

LEMMA 4.2 (Closure properties of \( \mathcal{E}\text{lim} \)).
(1) If \( d = d' \in N_f \) then \( d = d' \in \mathcal{E}\text{lim} \).
(2) If \( \alpha \in \text{Size} \) then \( \alpha = \alpha \in \mathcal{E}\text{lim} \).
(3) If \( \alpha , \alpha' \in \text{Size} \) then \( \langle \alpha \rangle = \langle \alpha' \rangle \in \mathcal{E}\text{lim} \).
(4) If \( A = A' \in \mathcal{F}_y \) and \( d_z = d_z' \in N_f \) and \( d_s = d_s' \in N_f \) then \( \text{case}_\ell A \ d_z \ d_s = \text{case}_\ell A' \ d_z' \ d_s' \in \mathcal{E}\text{lim} \).
(5) If \( D = D' \in N_f \) and \( d = d' \in N_f \) then \( \text{fix}_\ell D \ d = \text{fix}_\ell D' \ d' \in \mathcal{E}\text{lim} \).

Now we define some PERs and PER constructors on values. All these PERs \( \mathcal{A} \) are closed under weak head equality, meaning if \( a = b \in \mathcal{A} \) and \( a' = b' \in \mathcal{A} \) has the same weak head normal form as \( a \), then \( a' = b' \in \mathcal{A} \). (By symmetry, \( \mathcal{A} \) is also closed under weak head equality on the second argument.)

PER \( N \mathcal{E} \) interprets all neutral types.

\[
\begin{align*}
\text{Nat}(\alpha) :& \implies t \downarrow^T n \text{ and } t' \downarrow^T n' \text{ and } n = n' \in N_e.
\end{align*}
\]

\( \text{Nat}(\alpha) \) interprets \( \text{Nat} \alpha \) and is defined inductively by the following rules.

\[
\begin{align*}
\text{Size} & \text{ interprets Size and is a discrete PER of size values:} \\
\infty = \infty & \in \text{Size} \quad o = o & \in \text{Size} \quad x_k + o = x_k + o & \in \text{Size}
\end{align*}
\]

Let \( \mathcal{A} \) be a PER (including \( \mathcal{A} = \text{Size} \)) and \( \mathcal{F} \) a family of PERs over \( \mathcal{A} \) such that \( \mathcal{F}(\alpha) = \mathcal{F}(\alpha') \) whenever \( u = u' \in \mathcal{A} \). We define

\[
\begin{align*}
\mathcal{F} & \mathcal{A} \implies \{(t, t') \mid t u = t' u' \in \mathcal{F}(u) \text{ for all } u = u' \in \mathcal{A}\}.
\end{align*}
\]

For a family \( \mathcal{F} \) over \( \text{Size} \) we also have the irrelevant function space

\[
\begin{align*}
\forall \mathcal{F} & \implies \{(t, t') \mid t(\alpha) = t'(\alpha') \in \mathcal{F}(\beta) \text{ for all } \alpha, \alpha', \beta \in \text{Size}\}.
\end{align*}
\]

4.4 PER Model

Semantic types and their interpretation as PERs are now defined via a family of inductive-recursive definitions [Dybjor 2000], one for each universe level \( \ell \). The construction follows Abel et al. [2007].

By induction on \( \ell \in \mathbb{N} \) we define the PER family \( \_ = \_ \in \text{Set}_\ell \) of types together with the extension \( \mathcal{E}\ell \mathcal{T} \) (for \( \mathcal{T} = T' \in \text{Set}_\ell \)) which is a PER of values of type \( T \). The rules for \( \mathcal{T} = T' \in \text{Set}_\ell \) are listed in Fig. 7. All relations involved here are closed under weak head equality.

LEMMA 4.3 (Well-definedness). Let \( \mathcal{D}_1 : T_1 = T_2 \in \text{Set}_\ell \).
(1) Symmetry: \( T_2 = T_1 \in \text{Set}_\ell \).
(2) Transitivity: \( T_2 = T_3 \in \text{Set}_\ell \) then \( T_1 = T_3 \in \text{Set}_\ell \).
(3) Extension: \( \mathcal{E}\ell(T_1) = \mathcal{E}\ell(T_2) \) and “both” are PERs.

LEMMA 4.4 (Derivation independence of extension). If \( \mathcal{D}_1 : T = T_1 \in \text{Set}_\ell \) and \( \mathcal{D}_2 : T_2 = T \in \text{Set}_\ell \) then \( \mathcal{E}\ell(T_1) = \mathcal{E}\ell(T_2) \).

Since \( \mathcal{E}\ell(T) \) does not depend on \( \ell \) nor the derivation that introduced \( T = T' \in \text{Set}_\ell \), we may simply write \( t = t' \in \mathcal{E}(T) \) or even \( t = t' \in T \).
4.5 Subtyping

The semantic types (PERs) admit subsumption:

**Lemmas 4.5 (Subsumption).**

1. If $\alpha \leq \beta$ then $\text{Nat}(\alpha) \subseteq \text{Nat}(\beta)$.
2. If $\mathcal{F}(\alpha) \subseteq \mathcal{F}(\alpha')$ for all $\alpha \in \text{Size}$, then $\forall \mathcal{F} \subseteq \forall \mathcal{F}'$.
3. If $\mathcal{F}' \subseteq \mathcal{F}$ and $\mathcal{F}(u) \subseteq \mathcal{F}'(u)$ for all $u \in \mathcal{A}'$, then $\prod \mathcal{A} \mathcal{F} \subseteq \prod \mathcal{A}' \mathcal{F}'$.
4. If $\ell \leq \ell'$ then $\text{Set}_\ell \subseteq \text{Set}_{\ell'}$.

We define subtyping of type values $T \leq T'$ by induction on $T \in \text{Set}_\ell$ and $T' \in \text{Set}_{\ell'}$.

Simultaneously, we need to prove correctness, namely that $T \leq T' \in \text{Type}$ implies $\mathcal{E}(T) \subseteq \mathcal{E}(T')$. The correctness follows from Lemma 4.5 and we do not spell it out here.

**Lemma 4.6 (Subtyping is a preorder).**

1. If $T = T' \in \text{Set}_\ell$ then $T \leq T'$.
2. If $T_1 \leq T_2 \in \text{Type}$ and $T_2 \leq T_3 \in \text{Type}$ then $T_1 \leq T_3$.

---

Fig. 7. Semantic types and their interpretation.
Reflection and reification perform $\eta$-expansion so that we arrive at an $\eta$-long $\beta$-normal form. To perform the $\eta$-expansion, the precise type is not needed, just the approximate shape, in particular, whether it is a function type (do expand) or a base type (do not expand). For the logical framework, the shape of a dependent type is just its underlying simple type [Harper and Pfenning 2005]. However, in the presence of universes and large eliminations, there is no underlying simple type. Of course, we can take a type as its own shape, but we want at least that $\text{Nat} \alpha$ and $\text{Nat} \beta$ have the same shape even for different $\alpha, \beta$. Also all neutral types can be summarized under a single shape.

We make our intuition precise by defining a relation $T \subseteq S$ between type values, to express that $S$ is a possible shape of type $T$. The asymmetry of this relation stems from the case for function types. If function types $\Pi A B \subseteq \Pi R S$, we take $S$ to be a family over domain $A$, not $R$! We cannot take $R$ since we have to compare families $B$ and $S$ at a common domain, and $A$ and $R$ are not equal.

Fig. 8 defines $T \subseteq S$ for $T \in \text{Set}_\ell$. We call $T$ the template and $S$ one of its possible shapes. Note that $T \in \text{Set}_\ell$ and $T \subseteq S$ do not imply $S \in \text{Set}_\ell$. Type shapes are not well-defined types in general. For instance, assume a term $F : \text{Nat} 0 \rightarrow \text{Set}_0$ which diverges if applied to a successor term. Then $T := (x : \text{Nat} 0) \rightarrow F \ x$ is a well-defined type; we have $T \in \text{Set}_0$. Now consider $S := (x : \text{Nat} \infty) \rightarrow F \ x$. We have $T \subseteq S$, but $S$ is not well-defined; $S \notin \text{Set}_0$.

**Lemma 4.7 (Types are their own shapes).** If $T = T' \in \text{Set}_\ell$ then $T \subseteq T'$.

**Lemma 4.8 (Templates are up to equality).** If $T = T' \in \text{Set}_\ell$ and $T' \subseteq S$ then $T \subseteq S$.

However, templates are not closed under subtyping in either direction because subtyping is contravariant for function type domains but the shape relation is covariant.

Further, it is not true that equal types make equally good shapes. We do not have that $T \subseteq S$ and $S = S' \in \text{Set}_\ell$ imply $T \subseteq S'$. This property fails for function types. Given $\Pi U T \subseteq \Pi R S$ and $\Pi R S = \Pi R' S' \in \text{Set}_\ell$ we would need to show that $T[u] \subseteq S'[u']$ for all $u = u' \in U$, but we only have $S[u] = S'[u'] \in \text{Set}_\ell$ for all $u = u' \in R$, thus the induction does not go through. The fact that $U \subseteq R$ does not give us a handle on their inhabitants, we would need a stronger relation such as $U \leq R \in \text{Type}_\pi$. It is possible to construct an actual counterexample, using $\Pi R' S' = (x : \text{Nat} 0) \rightarrow F \ x$ from above and $\Pi R S = (x : \text{Nat} 0) \rightarrow G \ x$ such that $G$ is defined on all of Nat $\infty$ but agrees with $F$ only on $x \in \text{Nat} 0$. Then $\Pi U T = (x : \text{Nat} \infty) \rightarrow G \ x$ gives the desired counterexample.

Shapes are used to direct $\eta$-expansion when we reflect neutrals into semantic types and reify semantic values to long normal forms. The following theorem is the heart of our technical development.

**Theorem 4.9 (Reflection and reification).** Let $T \in \text{Set}_\ell$ and $T \subseteq S_1$ and $T \subseteq S_2$.
1. If $n_1 = n_2 \in \textsf{Ne}$ then $\uparrow^{n_1} n_1 = \uparrow^{n_2} n_2 \in T$.
2. If $t_1 = t_2 \in T$ then $\downarrow^{n_1} t_1 = \downarrow^{n_2} t_2 \in \textsf{Nf}$.

**Proof.** By induction on $T \in \textsf{Set}_\ell$ and cases on $T \in S_1$ and $T \in S_2$.

**Case** $T \not\in \forall B$ with $B[\alpha] \in \textsf{Set}_\ell$ for all $\alpha \in \textsf{Size}$

$$
\begin{array}{c}
S_1 \not\in \forall B_1 \quad B[\alpha] \in B_1[\alpha'] \text{ for all } \alpha, \alpha' \in \textsf{Size} \\
S_2 \not\in \forall B_2 \quad B[\alpha] \in B_2[\alpha'] \text{ for all } \alpha, \alpha' \in \textsf{Size}
\end{array}
$$

(1) To show $\uparrow^{n_1} n_1 = \uparrow^{n_2} n_2 \in T$ assume arbitrary $\alpha_1, \alpha_2 \in \textsf{Size}$. Since $n_1(\alpha_1) = n_2(\alpha_2) \in \textsf{Ne}$ by Lemma 4.1, we obtain $\uparrow^{B_1[\alpha_1]}(n_1(\alpha_1)) = \uparrow^{B_2[\alpha_1]}(n_2(\alpha_2)) \in B[\alpha_1]$ by induction hypothesis. Thus, $(\uparrow^{n_1} n_1)(\alpha_1) = (\uparrow^{n_2} n_2)(\alpha_2) \in B[\alpha_1]$ by weak head expansion, which entails the goal by definition of $\mathcal{E}(T)$.

(2) Assume $k \in \mathbb{N}$ and note that $x_k = x_k \in \textsf{Size}$, hence, $t_1(x_k) = t_2(x_k) \in B[x_k]$. Thus, by induction hypothesis, $R_{k+1} \downarrow^{B_1[x_k]}(t_1(x_k)) \not\in \forall v_i$ with $v_1 \approx v_2$, and finally $R_k \downarrow^{n_1} t_1 \not\in \lambda v_i$ by definition of read back.

**Corollary 4.10.** Let $T \in \textsf{Set}_\ell$.

(1) If $n = n' \in \textsf{Ne}$ then $\uparrow^T n = \uparrow^T n' \in T$.
(2) If $t = t' \in T$ then $\downarrow^T t = \downarrow^T t' \in \textsf{Nf}$.

### 4.7 Computation with Natural Numbers

In this section we show that the eliminations for natural numbers are accurately modeled.

**Lemma 4.11 (Case).** If $a = a' \in \textsf{Nat}(\alpha + 1)$ and $B = B' \in \textsf{Nat}(\alpha + 1) \rightarrow \textsf{Set}_\ell$ and $f_z = f'_z \in B(\text{zero}(\beta))$ and $f_z = f'_z \in (x : \textsf{Nat} \alpha) \rightarrow B(\text{suc}(\gamma) x)$ then a case $\ell B f_z f'_z = a' \text{ case } B' f'_z f'_z = B a$.

**Proof.** By induction on $a = a' \in \textsf{Nat}(\alpha + 1)$.

**Lemma 4.12 (Nat is Cocontinuous).** $\textsf{Nat}(\infty) = \bigcup_{\alpha < \infty} \textsf{Nat}(\alpha)$.

**Proof.** By induction on $a = a' \in \textsf{Nat}(\infty)$, we can easily show $a = a' \in \textsf{Nat}(\alpha)$ for some $\alpha < \infty$. For instance, $\alpha$ could be the number of uses of the successor rule plus one.

As the semantic counterpart of judgement $\Gamma \vdash T : \text{Adm } \ell$, let us write $[B = B' \in \text{Adm } \ell]$ iff $B = B' \in \text{FixK }\ell$ and for all $\beta \in \textsf{Size}$ and $a \in \textsf{ Nat} \beta$ we have $B \beta a \leq B \infty a \in \textsf{ Type }$ and $B' \beta a \leq B' \infty a \in \textsf{ Type }$.

**Lemma 4.13 (Fix).** Let $g = a \text{ fix } \ell B f$ and $g' = a' \text{ fix } \ell B' f'$. If $a = a' \in \textsf{Nat }\alpha$ and $B = B' \in \text{Adm } \ell$ and $f = f' \in \text{FixT } B$ then $g = g' \in B \alpha a$.

**Proof.** By well-founded induction on $a$.

### 4.8 Fundamental Theorem

In this section we show that the declarative judgements are sound, in particular, well-formed syntactic types map to semantic types, and definitionally equal terms map to related values in the PER model. The proof runs the usual course. First, we define inductively a PER of substitutions.
We write definitionally equal terms have the same normal form. We may write simply:

\[ \Gamma \vdash s \quad \iff \quad \Gamma \]
\[ \Gamma \vdash T = T' \quad \iff \quad \Gamma \vdash T = T' : s \text{ for some } s \]
\[ \Gamma \vdash T : \text{Adm } \ell \quad \iff \quad \Gamma \vdash T : \text{FixK } \ell \text{ and } T\eta = T'\eta' \in \text{Adm } \ell \text{ for all } \eta = \eta' \vdash \rho \in \Gamma \]
\[ \Gamma \vdash T \leq T' \quad \iff \quad \Gamma \vdash T \text{ and } \Gamma \vdash T' \text{ and } T\eta = T'\eta' \in \text{Type} \text{ for all } \eta = \eta' \vdash \rho \in \Gamma \]
\[ \Gamma \vdash t : T \quad \iff \quad \Gamma \vdash t = t : T \]
\[ \Gamma \vdash t = t' : T \quad \iff \quad \Gamma \vdash T \text{ and } T\eta = T'\eta' \in T\rho \text{ for all } \eta = \eta' \vdash \rho \in \Gamma \]
\[ \Gamma \vdash \sigma : \Delta \quad \iff \quad \Gamma \vdash \sigma = \sigma \vdash \sigma : \Delta \]
\[ \Gamma \vdash \sigma = \sigma' = \tau : \Delta \quad \iff \quad \Gamma \text{ and } \Delta \text{ and } \sigma\eta = \sigma'\eta' \vdash \tau\rho \in \Delta \text{ for all } \eta = \eta' \vdash \rho \in \Gamma \]

We write \( \rho \in \Gamma \) for \( \rho = \rho \vdash \rho \in \Gamma \).

**Lemma 4.14 (Resurrection).** If \( \eta = \eta' \vdash \rho \in \Gamma \) then \( \rho \in \Gamma^\circ \).

Then, in Fig. 9, we define semantic counterparts of our declarative judgements by recursion on the length of the context.

**Theorem 4.15 (Fundamental Theorem).**

1. If \( \vdash \Gamma \) then \( \vdash \Gamma \).
2. If \( \vdash \Gamma \vdash J \) then \( \Gamma \vdash J \).

**Proof.** Simultaneously, by induction on the derivation. \( \square \)

### 4.9 Completeness of NbE

From the fundamental theorem, we harvest completeness of NbE in this section, i.e., we show that definitionally equal terms have the same normal form. We may write simply \( \Gamma \) for its length \( |\Gamma| \) when there is no danger of confusion, for instance in de Bruijn level \( x_\Gamma \) or in read back \( R_\Gamma \). We define the identity environment \( \rho_{\Gamma} \) by induction on \( \Gamma \), setting \( \rho_{(\)} = (\) and \( \rho_{\Gamma \cdot \text{Size}} = (\rho_{\Gamma}, x_{\Gamma}) \) and \( \rho_{\Gamma,T} = (\rho_{\Gamma}, T_{\rho_{\Gamma}}) \).

**Lemma 4.16 (Identity Environment).** If \( \vdash \Gamma \) then \( \rho_{\Gamma} \vdash \Gamma \).

We now define the normalization relation for base types, but need to be strengthened for function types.

**Theorem 4.17 (Completeness of NbE).** If \( \vdash \ell + t = t' : T\) then there are normal forms \( v \approx v'\) such that \( \text{nbe}_T^\ell t \under\neg \ \v s\), and \( \text{nbe}_T^\ell t' \under\neg \ \v s\).

**Proof.** By the fundamental theorem, \( T\rho T \in \text{Set}_T\) for some \( \ell \) and \( t\rho T = t'\rho T \in T\rho T\). By reification (Cor. 4.10) we have \( \downarrow T\rho T(t\rho T) = \downarrow T\rho T(t'\rho T) \in \mathcal{Nf}\) which implies the theorem by read back with \( k = [\Gamma]\).

\[\]
Case $A' \land \Pi A B$.

\[ \Gamma \vdash T' \land A' \in s \quad \equiv \quad \Gamma \vdash T' \land \Pi U T : s \text{ for some } U, T \text{ and } \Gamma \vdash U \land A \in s \]
\[ \text{and } \forall \xi : \Gamma' \leq \Gamma. \quad \Gamma' + u : U \xi \land a \in A \implies \Gamma' + T(\xi, u) \land B[a] \in s. \]

\[ \Gamma \vdash t : T' \land f \in A' \quad \equiv \quad \Gamma' \vdash T' \land \Pi U T : s \text{ for some } U, T \text{ and } \Gamma' \vdash U \land A \in s \]
\[ \text{and } \forall \xi : \Gamma' \leq \Gamma. \quad \Gamma' + u : U \xi \land a \in A \implies \Gamma' + T(\xi, u) \land f \in B[a]. \]

Case $A' \land \Pi A B$.

\[ \Gamma \vdash T' \land A' \in s \quad \equiv \quad \Gamma \vdash T' \land \Pi T : s \text{ for some } T 
\[ \text{and } \forall \xi : \Gamma' \leq \Gamma. \quad \Gamma' + b \equiv R_{\text{size}} \beta \implies \Gamma' + T(\xi, b) \land B[\beta] \in s. \]

\[ \Gamma \vdash t : T' \land f \in A' \quad \equiv \quad \Gamma' \vdash T' \land \forall T : s \text{ for some } T 
\[ \text{and } \forall \xi : \Gamma' \leq \Gamma. \quad \Gamma' + a \equiv R_{\text{size}} \alpha \implies \Gamma' + f(\xi a : T(\xi, b) \land f \in B[\beta]). \]

We may prove theorems "by induction on $\Gamma \vdash T \land A \in s$", even if in reality this will be proofs by induction on $A \in s$ and cases on $\Gamma \vdash T \land A \in s$. We write $\Gamma \vdash T \land A$ if $\Gamma \vdash T \land A \in s$ for some sort $s$. The logical relations are closed under weakening.

**Theorem 5.2 (Into and Out of the Logical Relation).** Let $\Gamma \vdash T \land A \in s$ and $A \in S$. Then:

1. If $\Gamma \vdash t \equiv R_{\text{ne}} n : T$, then $\Gamma \vdash t : T \equiv R_{\text{ne}} n \in A$.
2. If $\Gamma \vdash t : T \land f \in A$ then $\Gamma \vdash t \equiv R_{\text{for}} f : T$.
3. If $\Gamma \vdash T \equiv R_{\text{for}} A : s$.

**Proof.** Simultaneously by induction on $\Gamma \vdash T \land A \in s$. \hfill \Box

**Lemma 5.3 (Semantic implies judgmental subtyping)** [Fridlender and Pagano 2013].

1. If $\Gamma \vdash a \equiv R_{\text{size}} \alpha$ and $\Gamma \vdash b \equiv R_{\text{size}} \beta$ and $\alpha \leq \beta$ then $a \leq b$.
2. If $\Gamma \vdash T \land A$ and $\Gamma \vdash T' \land A' \land A \leq A' \in \text{Type}$ then $\Gamma \vdash T \leq T'$.

**Lemma 5.4 (Subsumption for the logical relation)** [Fridlender and Pagano 2013]. If $\Gamma \vdash T \land A$ and $\Gamma \vdash T' \land A' \land A \leq A' \in \text{Type}$ then $\Gamma \vdash t : T \land f \in A$ implies $\Gamma \vdash t : T' \land f \in A'$.

\[ \begin{array}{ll}
+ \, \Gamma & \Gamma \vdash \sigma \equiv \tau : \Delta \land \eta \equiv \rho \\
\Gamma \vdash () \equiv () : () \land () \equiv () & \Gamma \vdash (\sigma, a) \equiv (\tau, a) : \Delta \land \text{Size}(\eta, \alpha) \equiv (\rho, \alpha) \\
\hline
\Gamma \vdash \sigma \equiv \tau : \Delta \land \eta \equiv \rho & \Gamma \vdash (\sigma, a) \equiv (\tau, a) : \Delta \land \text{Size}(\eta, \alpha) \equiv (\rho, \alpha)
\end{array} \]

\[ \begin{array}{ll}
\Gamma \vdash a \land \Delta \land \eta \equiv \rho & \Gamma \vdash (\sigma, a) \equiv (\tau, a) : \Delta \land \text{Size}(\eta, \alpha) \equiv (\rho, \beta) \\
\Delta \land T & \Gamma \vdash u : T \land T \tau & \Gamma \vdash u : T \land T \tau \land f \in T \land f = g \in T \rho \\
\Gamma \vdash (\sigma, u) \equiv (\tau, t) : \Delta \land T \land (\eta, f) \equiv (\rho, q)
\end{array} \]

Fig. 10. Logical relation for substitutions. $\Gamma \vdash \sigma \equiv \tau : \Delta \land \eta \equiv \rho$. 

Fig. 10 defines a logical relation for substitutions \( \Gamma \vdash \sigma \equiv \tau : \Delta \odot \eta \equiv \rho \). We write \( \Gamma \vdash \tau : \Delta \odot \rho \) for \( \Gamma \vdash \tau \not\equiv \Delta \odot \rho \).

The following judgements are used to state the fundamental theorem of typing.

\[
\begin{align*}
\Gamma \vdash t : T & \iff \Gamma' \vdash t \sigma : T\sigma \odot t\eta \in T\rho \text{ for all } \Gamma' \vdash \sigma \equiv \tau : \Gamma' \odot \eta \equiv \rho \\
\Gamma \vdash \sigma_0 : \Delta & \iff \Gamma' \vdash \sigma_0\sigma \equiv \sigma_0\tau : \Delta \odot \sigma_0\eta \equiv \sigma_0\rho \text{ for all } \Gamma' \vdash \sigma \equiv \tau : \Gamma' \odot \eta \equiv \rho
\end{align*}
\]

**Theorem 5.5 (Fundamental Theorem of Typing).**

1. If \( \Gamma \vdash t : T \) then \( \Gamma \vdash t : T \).
2. If \( \Gamma \vdash \sigma : \Delta \) then \( \Gamma \vdash \sigma : \Delta \).

**Proof.** Each by induction on the derivation. \( \square \)

**Lemma 6.1 (Soundness of Algorithmic Subtyping).** \( \Gamma \vdash T \leq T' \) if \( \Gamma \vdash T \leq T' \).

**Corollary 5.7 (Soundness of NbE).**

1. If \( \Gamma \vdash t : T \) then \( \Gamma \vdash t \approx \text{nbe}^T \).
2. If \( \Gamma \vdash t, t' : T \) and \( \text{nbe}^T t \approx \text{nbe}^T t' \) then \( \Gamma \vdash t = t' : T \).

**Proof.** (1) For the identity environment \( \Gamma \vdash \text{id} : \Gamma \odot \rho \) (Lemma 5.6) the Fundamental Theorem for Typing gives \( \Gamma \vdash t : T \odot t\rho \in T\rho \). This implies \( R_{\rho} \downarrow (T\rho)^1(t\rho) \) \( \leq \nu \) for some normal form \( \nu \) and \( \Gamma \vdash t = \nu : T \) by Thm. 5.2. Then (2): From (1), using Lemma 3.1: \( \Gamma \vdash t \approx \text{nbe}^T t = \text{nbe}^T t' = t' : T \). \( \square \)

**Corollary 5.8 (Decidability of Judgemental Equality).** \( \Gamma \vdash t, t' : T \) then the test whether \( \text{nbe}^T t \approx \text{nbe}^T t' \) terminates and decides \( \Gamma \vdash t = t' : T \).

From correctness of NbE and the logical relations we can further prove injectivity of type constructors, inversion of subtyping, and subject reduction. The proofs follow roughly Fridlender and Pagano [2013], for details, see the long version of this article.

### 6 Algorithmic Subtyping

Fig 11 defines an incremental subtyping algorithm \( \Gamma \vdash T \leq T' \). Neutral types are subtypes iff they are equal, which is checked using NbE.

| \( T \downarrow n \) | \( T' \downarrow n' \) | \( \text{nbe}^T n \approx \text{nbe}^T n' \) | \( T \downarrow \text{Set}_\ell \) | \( T' \downarrow \text{Set}_\ell' \) | \( \ell \leq \ell' \) | \( T \downarrow \text{Nat} a \) | \( T' \downarrow \text{Nat} a' \) | \( a \leq a' \) | \( T_1 \downarrow \Pi^* \text{Size} T_1 \) | \( T_2 \downarrow \Pi^* \text{Size} T_2 \) | \( \Gamma \vdash \text{Size} + T_1 \leq T_2 \) | \( \Gamma \vdash T_1' \leq T_2' \) | \( \Gamma \vdash U_1 \vdash T_1 \) | \( T_2 \downarrow \Pi U_2 T_2 \) | \( \Gamma \vdash U_2 \vdash T_1 \leq T_2 \) | \( \Gamma \vdash T_1' \leq T_2' \) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

**Fig. 11.** Algorithmic subtyping \( \Gamma \vdash T \leq T' \)

**Lemma 6.1 (Soundness of Algorithmic Subtyping).** \( \Gamma \vdash T \leq T' \) then \( \Gamma \vdash T \leq T' \).

**Proof.** By induction on \( \Gamma \vdash T \leq T' \), soundness of NbE, and subject reduction. \( \square \)
LEMMA 6.2 (SEMANTIC SUBTYPING IMPLIES ALGORITHMIC SUBTYPING).
If $\Gamma \vdash T \otimes A$ and $\Gamma \vdash T' \otimes A'$ and $A \leq A' \in \text{Type}$ then $\Gamma \vdash T \ll T'$.

Proof. By induction on $\Gamma \vdash T \otimes A$ and $\Gamma \vdash T' \otimes A'$ and cases on $A \leq A' \in \text{Type}$. □

COROLARY 6.3 (COMPLETENESS OF ALGORITHMIC SUBTYPING). If $\Gamma \vdash T \leq T'$ then $\Gamma \vdash T \ll T'$.

Proof. By the fundamental theorems $\Gamma \vdash T \otimes T \rho_T$ and $\Gamma \vdash T' \otimes T' \rho_{T'}$ and $T \rho_T \leq T' \rho_{T'} \in \text{Type}$.

By Lemma 6.2, $\Gamma \vdash T \ll T'$.

LEMMA 6.4 (TERMINATION OF ALGORITHMIC SUBTYPING). If $\Gamma \vdash T \otimes A$ and $\Gamma \vdash T' \otimes A'$ then the query $\Gamma \vdash T \ll T'$ terminates.

Proof. By induction on $A \in s$ and $A' \in s'$ and cases on $\Gamma \vdash T \otimes A$ and $\Gamma \vdash T' \otimes A'$. □

THEOREM 6.5 (DECIDABILITY OF SUBTYPING). If $\Gamma \vdash T, T'$, then $\Gamma \vdash T \leq T'$ is decided by the query $\Gamma \vdash T \ll T'$.

Proof. By the fundamental theorem of typing, $\Gamma \vdash T \otimes A$ and $\Gamma \vdash T' \otimes A'$, thus, the query $\Gamma \vdash T \ll T'$ terminates by Lemma 6.4. If successfully, then $\Gamma \vdash T \leq T'$ by soundness of algorithmic equality. Otherwise $\Gamma \vdash T \leq T'$ is impossible by completeness of algorithmic equality. □

7 TYPE CHECKING

In this section, we show that type checking for normal forms is decidable, and succeeds for those which can be typed via the restricted rule for size polymorphism elimination:

$$\frac{\Gamma, \vdash t : \forall T}{\Gamma \vdash t \langle a \rangle : T[a]}$$

We refer to the restricted typing judgement as $\Gamma \vdash s t : T$ and obviously, if $\Gamma \vdash s t : T$ then $\Gamma \vdash t : T$.

Figure 12 displays the rules for bidirectional typing of normal forms. Note that we could go beyond normal forms, by adding inference rules for the Nat-constructors:

$$\frac{\Gamma \vdash a : \text{Size}}{\Gamma \vdash \text{zero}(a) \Rightarrow \text{Nat}(a + 1)} \quad \frac{\Gamma \vdash a : \text{Size}}{\Gamma \vdash \text{succ}(a)t \Rightarrow \text{Nat}(a + 1)}$$

THEOREM 7.1 (SOUNESS OF TYPE CHECKING). Let $\vdash \Gamma$.

1. If $\Gamma \vdash T$ and $\emptyset : \Gamma \vdash t \Rightarrow T$ then $\Gamma \vdash s t : T$.
2. If $\emptyset : \Gamma \vdash t \Rightarrow T$ then $\Gamma \vdash T$ and $\Gamma \vdash s t : T$.

LEMMA 7.2 (WEAK HEAD REDUCTION OF SUBTYPES). Let $\emptyset : \Gamma \vdash T \ll T'$.

1. If $T' \ll \text{Nat } a'$ then $T \ll \text{Nat } a$ and $\Gamma \vdash a \ll a' : \text{Size}$.
2. If $T' \ll \text{Set}_\ell$ then $T \ll \text{Set}_\ell$ and $\ell \ll \ell'$.
3. If $T' \ll \Pi A B'$ then $T \ll \Pi A B$ and $\Gamma \vdash A' \ll A$ and $\Gamma.A' \ll B \ll B'$.
4. If $T' \ll \Pi^{*} B'$ and $T \ll \Pi^{*} B$ and $\Gamma.\text{Size} \vdash B < : B'$.

Proof. By cases on $\emptyset$, since weak head evaluation is deterministic. □

This lemma also holds in the other direction of subtyping, i.e., when $T \ll T'$ and $T$ weak head evaluates, then $T'$ weak head evaluates to a type of the same form.

LEMMA 7.3 (SUBSUMPTION FOR TYPE CHECKING). Let $\text{id} : \Gamma' \leq \Gamma$.

1. If $\emptyset : \Gamma \vdash t \ll T$ and $\Gamma^{\otimes} \vdash T \ll T'$ then $\Gamma' \vdash t \ll T'$.
2. If $\emptyset : \Gamma \vdash t \Rightarrow T$ then $\Gamma' \vdash t \Rightarrow T'$ and $\Gamma^{\otimes} \vdash T \ll T'$.

Checking \( \Gamma \vdash t \equiv T \). Input: \( \Gamma, t, T \). Output: yes/no.

\[
\begin{align*}
T' \not\owns s & \quad \Gamma \vdash a : \text{Size} & \quad T' \not\owns \text{Set}_\ell & \quad \ell < \ell' \\
\Gamma \vdash \text{Nat } a & \equiv T' & \Gamma \vdash \text{Set}_\ell & \equiv T'
\end{align*}
\]

\[
\begin{align*}
T' \not\owns s & \quad \Gamma \vdash U \equiv s & \quad \Gamma. U + T & \equiv s \\
\Gamma \vdash \Pi U T & \equiv T' & T' \not\owns s & \quad \Gamma. \text{Size} \vdash T \equiv s \\
\Gamma \vdash \text{Nat } a & \equiv T' & \Gamma \vdash \text{Set}_\ell & \equiv T'
\end{align*}
\]

Inference \( \Gamma \vdash t \Rightarrow T \). Input: \( \Gamma, t \). Output: \( T \) or no.

\[
\begin{align*}
\Gamma(i) = 't' & \quad \Gamma \vdash T' & \quad T' \not\owns \Pi U T & \quad \Gamma \vdash u \equiv U \\
\Gamma \vdash v_i & \equiv T & \Gamma \vdash t u & \Rightarrow T[u]
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash t \Rightarrow T' & \quad T' \not\owns \Pi \text{Size } T & \quad \Gamma \vdash a : \text{Size} \\
\Gamma \vdash t a & \Rightarrow T[a]
\end{align*}
\]

\[
\begin{align*}
\Gamma^\circ \vdash T & \equiv \text{Nat } (a + 1) \rightarrow \text{Set}_\ell \\
\Gamma \vdash t_z & \equiv T(\text{zero}(a)) & \Gamma \vdash t_s & \equiv (x : \text{Nat } a) \rightarrow T(\text{suc}(a) x) \\
\Gamma \vdash u \equiv \text{Nat } a & \quad \Gamma^\circ \vdash T & \equiv \text{FixK } \ell & \quad \Gamma \vdash t & \equiv \text{FixT } T \\
\Gamma \vdash u \text{ case}_\ell T t_z t_s & \equiv T u \\
\Gamma \vdash u \text{ fix}_\ell T t & \equiv T a u
\end{align*}
\]

Fig. 12. Bidirectional type-checking of normal forms.

**Proof.** Simultaneously by induction on \( \mathcal{D} \), using Lemma 7.2 and soundness and completeness of algorithmic subtyping. \( \square \)

**Theorem 7.4 (Completeness of Type Checking for Normal Terms).**

1. If \( \mathcal{D} :: \Gamma \vdash_s v : T \) then \( \Gamma \vdash v \equiv T \).
2. If \( \mathcal{D} :: \Gamma \vdash_s m : T \) then \( \Gamma \vdash m \Rightarrow U \) and \( \Gamma^\circ \vdash U \leq T \).

**Proof.** Simultaneously by induction on \( \mathcal{D} \), using (strong) inversion and Lemma 7.3. \( \square \)

**Lemma 7.5 (Termination of Type Checking).** Let \( \vdash \Gamma \).

1. The query \( \Gamma \vdash t \Rightarrow ? \) terminates.
2. If \( \Gamma^\circ \vdash T \) then the query \( \Gamma \vdash t \Rightarrow T \) terminates.

**Proof.** By induction on \( t \), using type weak head normalization and soundness of type checking, to maintain well-formedness of types. And, of course, decidability of subtyping. \( \square \)

**Theorem 7.6 (Decidability of Type Checking for Normal Terms).** Let \( \vdash \Gamma \) and \( \Gamma^\circ \vdash T \). Then \( \Gamma \vdash_s v : T \) is decided by \( \Gamma \vdash v \equiv T \).

8 DISCUSSION AND CONCLUSIONS

In this article, we have described the first successful integration of higher-rank size polymorphism into a core type theory with dependent function types, a sized type of natural numbers, a predicative hierarchy of universes, subtyping, and $\eta$-equality. This is an important stepping stone for the smooth integration of sized types into dependently-typed proof assistants. In these final paragraphs, we discuss some questions and insights that follow from our work and go beyond it.

It is now straightforward to add a unit type $1$ with extensional equality $t = \ast : 1$ for all $t : 1$. We simply extend reification such that $\downarrow 1a = \ast$. Further, $1$ is a new type shape with rule $1 \triangleright 1$.

In the long run, we wish for a type-directed equality check that does not do normalization in one go, but interleaves weak head normalization with structural comparison. Such an equality test is at the heart of Agda’s type checker and it generates constraints for meta variables involved in type reconstruction [Norell 2007]. However, the usual bidirectional construction [Abel and Scherer 2012] does not seem to go through as we lack uniqueness of types (and even principal types).

For now, we have only exploited shape-irrelevance of sized types, but this directly extends to universe levels. If we consider all universes as a single shape $\text{Set}_{\ell_1} \triangleright \text{Set}_{\ell_2}$, we can quantify over levels irrelevantly, as $\text{Set}$ is a shape-irrelevant type constructor. This is a stepping stone for integrating universe cumulativity with Agda’s explicit universe-polymorphism. If levels are no longer unique (because of subsumption), they will get in the way of proofs, analogously to sizes. With an irrelevant quantifier we can ignore levels where they do not matter. We will still respect them where they matter, thus, we keep consistency.

Our reflections on level irrelevance lead us to the question: can a type theory $T$ with a stratified universe hierarchy be understood as a sort of refinement of the inconsistent System U (Type:Type)? Intuitively, when checking two terms of $T$ for equality, could we ignore the stratification in the type $A$ which directs the equality check (thus, consider $A$ coming from $U$)? Such a perspective would put stratification in one pot with size assignment: Size annotations and levels are both just annotations for the termination checker, but do not bear semantic relevance. We could switch the universe checker temporarily off as we do with the termination checker—cf. the work of Stump et al. [2010] on termination casts.

Finally, we would like a general theory of shape-irrelevance that extends beyond size-indexed types. For instance, any data type constructor could be considered shape-irrelevant in all its indices, with the consequence that index arguments in the data constructors could be declared irrelevant. However, our notion of judgmental equality does not support irrelevant arguments of dependent type. It works for the non-dependent type $\text{Size}$, but we also relied on having a closed inhabitant $\infty$ in $\text{Size}$. More research is needed to tell a more general story of shape-irrelevance.

ACKNOWLEDGMENTS

This material is based upon work supported by the Swedish Research Council (Vetenskapsrådet) under Grant No. 621-2014-4864 Termination Certificates for Dependently-Typed Programs and Proofs via Refinement Types. The first author is grateful for recent discussions with Thierry Coquand, Nils Anders Danielsson, and Sandro Stucki which helped clarifying the thoughts leading to this work. He also acknowledges past discussions with Christoph-Simon Senjak. The incentive to write this article came during the EU Cost Action CA15123 EUTYPES meeting in Ljubljana in January 2017; thanks to Andrej Bauer for organizing it.

REFERENCES
