Logical properties of a modality for erasure

Nils Anders Danielsson

University of Gothenburg

Abstract

This text develops some theory for a type constructor *Erased*, which turns out to be a modality. *Erased* is similar to an identity function for types, but values of type *Erased A* should be erased by the compiler. Special typing rules are used to avoid getting into a situation in which a program has to make a decision based on an erased piece of data, and this arguably makes the theory of *Erased* more interesting than that of the identity function.

The theory has been formalised in Agda. It is developed based on some assumptions that are satisfied in Cubical Agda, and also in more traditional Agda with the K rule.

The text focuses on the logical properties of *Erased*, provable inside Agda, rather than external properties related to, say, time or space complexity. One question is when an erased value can be resurrected. In an attempt to shed some light on this question the text develops some theory of two notions of *stability*, one of which corresponds to the concept of a modal type.

The text includes an example suggesting how the theory can be put to use in practice: a type that is equivalent to the unary natural numbers and computes roughly like the unary natural numbers at compile-time (for some operations), but computes roughly like an arbitrary, possibly efficient implementation of natural numbers at run-time.

1 Introduction

Dependently typed programs, especially those written using the "intrinsic" approach in which invariants and other proofs are mixed with programs, can exhibit poor performance if the proof parts are not erased by the compiler. As a simple example (Brady et al. 2004), consider length-indexed vectors:

```
data Vec (A : Set) : ℕ → Set where
  []  : Vec A 0
  _::_ : A → Vec A n → Vec A (1 + n)
```

It may seem as if the cons constructor (_::_) takes two arguments—the head and the tail—but it also takes a third one: the length of the tail. If the length of every tail is stored, without any sharing, and a unary representation of natural numbers is used, then the space required for a vector of length n is $\Omega(n^2)$.

Data or code that is not required to actually run a program can be erased by the compiler. One can erase types (Coquand and Huet 1988; Augustsson

1998) or other things that lack computational content (Hayashi and Nakano 1988), or let the system figure out things that can be erased automatically (Brady et al. 2004; Brady 2005; Mishra-Linger 2008; Fredriksson and Gustafsson 2011; Tejiščák and Brady 2015). One can also let the programmer mark parts that should be erased, and let the type-checker check that erased parts cannot influence the results of run-time computations (Paulin-Mohring 1989; Paulin-Mohring and Werner 1993; Raamsdonk and Severi 2002; Letouzey 2003; Fernandez et al. 2003; Barras and Bernardo 2008; Mishra-Linger and Sheard 2008; Mishra-Linger 2008; Gundry and McBride 2013; Bernardy and Moulin 2013; Gundry 2013; Weirich et al. 2017). One variant of this approach—based on ideas due to McBride (2016) and Atkey (2018)—is available in Agda (this feature was implemented by Andreas Abel). Function arguments and definitions can be marked as erased using @0 ("used zero times"):

```
data Vec (@0 A : Set) : @0 ℕ → Set where
  []  : Vec A 0
  _::_ : {@0 n : ℕ} → A → Vec A n → Vec A (1 + n)
```

With this definition of vectors the natural number indices can safely be erased by the compiler (assuming that the type-checker does its job).

Note that @0 is not a type former. However, this is easily rectified (Mishra-Linger 2008):

```
record Erased (@0 A : Set a) : Set a where
  constructor [_]
  field
    @0 erased : A
```

Erased A is a record type with a single, erased field (called erased) of type A. This text is dedicated to the study of this type former.

It may seem as if *Erased*, which is similar to an identity function for types (*Set a* is a universe of types), is not very interesting. However, it is possible to develop quite a bit of theory for it:

- *Erased* is a monad that commutes with several type constructors, including "is an n-type" (see Section 3).
- An equivalence between erased equality and equality of erased values turns out to be a key property for the development of the theory (see Section 3.4). This property can be proved in cubical type theory, or in plain type theory with the K rule (given some assumptions). It can also be proved in plain type theory without the K rule if one assumes that equality of functions is extensional (see Section 4.6).
- *Erased* and the constructor [_] form a left exact modality in the sense of Rijke et al. (2019), see Section 3.6.
- Section 4 discusses a number of conditions under which a type A is *stable* (when *Erased A* implies A), or *very stable* (when A is modal, i.e. when [_] is an equivalence at type $A \to$ *Erased A*).
- While it may not be possible to prove that, say, the natural numbers are very stable, this can be proved for equality of natural numbers—and

many other types—and this turns out to be sufficient to develop some applications (see Section 5). The property of having very stable equality can be characterised in the following way: equality is very stable for a type A if and only if $[\_]$ is an embedding for A (see Section 4.3).

The main example in Section 5 is an implementation of natural numbers that computes roughly like the unary natural numbers at compile-time (for some operations), but roughly like an arbitrary, possibly efficient implementation of natural numbers at run-time.

The text is accompanied by machine-checked proofs: all of the main results in the text—including everything with an equation number—have been checked using Agda (but no claim is made that Agda is free of bugs, and there are minor differences between the source code and the text). The source code is, at the time of writing, available from `http://www.cse.chalmers.se/~nad/`. Some parts of the code are developed in plain Agda with the K rule turned off, others with the K rule turned on, and the rest in Cubical Agda; the text describes which parts of the development depend on the K rule or Cubical Agda. Except when Cubical Agda is used it is also documented which results depend on extensionality for functions (which is provable in Cubical Agda).

2 The annotation @0

The annotation @0 is based on Atkey's quantitative type theory (2018), instantiated with the boolean semiring containing 0 (used for things that are unused at run-time) and ω (used for things that are allowed to be used an arbitrary number of times).

Let me use some examples to illustrate how the annotation works. The following code is OK, because the result of the function can be computed without making use of the (implicit) type argument A:

$$\begin{aligned}&ok : \{@0\ A : Set\} \to A \to A\\&ok\ x = x\end{aligned}$$

However, the following code is not OK (in a non-erased context), because the argument of type A is marked as erased:

$$\begin{aligned}&not\text{-}ok : \{@0\ A : Set\} \to @0\ A \to A\\&not\text{-}ok\ x = x\end{aligned}$$

Erased arguments and definitions can always be used in erased contexts, which include type signatures, erased function arguments (for instance the subexpression e of the expression $f\ e$, if the first explicit argument of f is erased), and definitions that have been marked as erased. Thus the following variant of *not-ok* is OK:

$$\begin{aligned}&@0\ ok\text{-}in\text{-}an\text{-}erased\text{-}context : \{@0\ A : Set\} \to @0\ A \to A\\&ok\text{-}in\text{-}an\text{-}erased\text{-}context\ x = x\end{aligned}$$

Pattern matching on erased arguments is not allowed in non-erased contexts, for data types with two or more constructors:

$$
\begin{aligned}
&\mathit{not\text{-}ok} : @0\ \mathit{Bool} \to \mathit{Bool}\\
&\mathit{not\text{-}ok}\ \mathsf{true}\ = \mathsf{false}\\
&\mathit{not\text{-}ok}\ \mathsf{false} = \mathsf{true}
\end{aligned}
$$

However, pattern matching is allowed for the empty type, a data type with zero constructors:

$$
\begin{aligned}
&\textbf{data}\ \bot : \mathit{Set}\ \textbf{where}\\
&\\
&\mathit{OK} : @0\ \bot \to \mathit{Set}\\
&\mathit{OK}\ ()
\end{aligned}
$$

Should pattern matching be allowed for single-constructor data types? Atkey only treats a non-recursive tensor product, not recursive types like W-types or inductive families like the equality (or identity) type. My guess is that this kind of pattern matching is fine for non-indexed types (if constructor arguments are treated as erased): code of the form $f\ (\mathsf{c}\ x) = C\ [x]$, where x is only used in erased contexts, can almost be rewritten to $f\ y = C\ [\mathit{proj}\ y]$, where $\mathit{proj}\ (\mathsf{c}\ x) = x$ is an erased definition. The termination checker might not accept the resulting definition of f, but if the code is only used for execution of closed terms this might not matter. (If strict evaluation is used, then complete removal of the erased argument might lead to an infinite loop. An alternative is to replace the argument with a dummy one (Letouzey 2003).)

The situation is perhaps more interesting for inductive families. Consider the equality type, defined as an inductive family (here $\mathit{Set}\ a$ is a universe with universe level a):

$$
\begin{aligned}
&\textbf{data}\ \_{\equiv}\_\ \{A : \mathit{Set}\ a\}\ (x : A) : A \to \mathit{Set}\ a\ \textbf{where}\\
&\quad \mathsf{refl} : x \equiv x
\end{aligned}
$$

In a setting with the K rule—and thus uniqueness of identity proofs—it might be OK to allow pattern-matching for erased values from this type family: in the terminology of Brady et al. (2004) it is concretely collapsible. (I have not checked to what extent the theory of Brady et al. is compatible with that of Atkey.) However, in a setting with computing univalence this seems to make less sense. Consider the following code, where $\mathit{swap} : \mathit{Bool} \equiv \mathit{Bool}$ is constructed by applying univalence to the equivalence corresponding to the *not* function:

$$
\begin{aligned}
&\mathit{transport} : (P : A \to \mathit{Set}\ p) \to @0\ x \equiv y \to P\ x \to P\ y\\
&\mathit{transport}\ P\ \mathsf{refl}\ p = p\\
&\\
&t\ f : \mathit{Bool}\\
&t\ =\ \mathit{transport}\ \mathit{id}\ \mathsf{refl}\quad \mathsf{true}\\
&f\ =\ \mathit{transport}\ \mathit{id}\ \mathit{swap}\ \mathsf{true}
\end{aligned}
$$

The result of running t should be true, and the result of running f should be false, but if the equality proof is erased these computations presumably return the same value. Currently there is no implementation of type theory that properly supports computing univalence and the equality type given above, but perhaps such an implementation emerges in the future.

Given that the theory of pattern-matching for erased single-constructor data types (in non-erased contexts) has, to my knowledge, not been worked out properly, I will point out which results below depend on this feature.

Another point worth mentioning is that there is currently no compiler for Cubical Agda, and it is not known if erasure can be handled in a sensible way in this setting. Here I do not count the approach of simply ignoring the @0 annotations in the compiler as sensible, but this approach at least suggests that the presence of @0 should not introduce any inconsistencies into the theory (but as far as I know this has not been proved). Below I point out which results depend on Cubical Agda.

3 Erased

Let us now discuss some properties of *Erased*:

```
record Erased (@0 A : Set a) : Set a where
  constructor [_]
  field
    @0 erased : A
```
(1)

This record type has a single field, erased, which is erased, and a constructor [_]—read this as *box*. It is only the field that is marked as erased, not the entire record type. However, Agda's compiler (at least the GHC backend) erases record types where all fields are erased.

Below the following variant of [_] with an explicit type argument will also be used:

```
[_|_] : (@0 A : Set a) → @0 A → Erased A
[ _ | x ] = [ x ]
```

3.1 *Erased* is a monad

Erased is a monad (Mishra-Linger 2008), with [_] as the return function, and bind defined in the following way:

```
_>>=_ : {@0 A : Set a} {@0 B : Set b} →
        Erased A → (A → Erased B) → Erased B
x >>= f = [ erased (f (erased x)) ]
```
(2)

It is OK to use the erased projection above, because the argument of [_] is erased. The monad laws hold by definition (in the case of the left and right identity laws this follows from a definitional η-law for *Erased*, $x = [\ \mathsf{erased}\ x\ ]$).

Note that I have given type signatures for some of the implicit arguments of bind (A and B), but not others (the universe levels a and b). In this text I usually do not omit implicit argument declarations for erased arguments, and in cases where such a declaration is omitted it is mentioned in the text. Thus it should be clear when an argument is assumed to be erased and when it is not.

We also have the following property (Mishra-Linger 2008):

$$\{@0\ A : \mathit{Set}\ a\} \to \mathit{Erased}\ (\mathit{Erased}\ A) \simeq \mathit{Erased}\ A \tag{3}$$

Here $A \simeq B$ means that A and B are *equivalent*, in the sense of The Univalent Foundations Program (2013). Two types A and B are equivalent if and only if there are two functions, one from A to B and one from B to A, that are inverses of each other.

3.2 The relationship between @0 and *Erased*

How is @0 related to *Erased*? Mishra-Linger (2008) notes (basically) that the following equivalence can be proved:

$$(\mathit{Erased}\ A \to B) \simeq (@0\ A \to B) \tag{4}$$

We can prove a similar property for dependent functions:

$$\begin{array}{l}\{P : \mathit{Erased}\ A \to \mathit{Set}\ p\} \to \\ ((x : \mathit{Erased}\ A) \to P\ x) \simeq ((@0\ x : A) \to P\ [\ x\ ])\end{array} \tag{5}$$

The proof is very easy (using definitional η-equality for *Erased* and Π) and omitted. However, note that it can be tricky to use this result in non-erased contexts: the argument P is not erased, so it cannot use erased in a non-erased context. Can the result be proved when P is erased?

Agda supports several definitions of equality (including the data type given in Section 2, Cubical Agda paths, and the Cubical Agda identity type family (Cohen et al. 2018; Swan 2016)). Some of the code accompanying this paper, including the property above, has not been developed using a concrete definition of equality, but instead under the assumption that there is an equality type family _≡_, a canonical proof of reflexivity *refl*, a J rule, and a *propositional* computation rule for J:

$$\begin{array}{ll}\_\equiv\_ & : \{A : \mathit{Set}\ a\} \to A \to A \to \mathit{Set}\ a \\ \mathit{refl} & : (x : A) \to x \equiv x \\ J & : (P : \{x\ y : A\} \to x \equiv y \to \mathit{Set}\ p) \to \\ & \quad (\forall\ x \to P\ (\mathit{refl}\ x)) \to \\ & \quad (\mathit{eq} : x \equiv y) \to P\ \mathit{eq} \\ \mathit{J\text{-}refl} & : (P : \{x\ y : A\} \to x \equiv y \to \mathit{Set}\ p) \\ & \quad (r : \forall\ x \to P\ (\mathit{refl}\ x)) \to \\ & \quad J\ P\ r\ (\mathit{refl}\ x) \equiv r\ x\end{array} \tag{6}$$

Using this formulation of equality I have not been able to prove the following variant of Lemma 5:

$$\begin{array}{l}\{@0\ A : \mathit{Set}\ a\}\ \{@0\ P : \mathit{Erased}\ A \to \mathit{Set}\ p\} \to \\ ((x : \mathit{Erased}\ A) \to P\ x) \simeq ((@0\ x : A) \to P\ [\ x\ ])\end{array} \tag{7}$$

One reason is that the arguments of *refl* are not erased. However, this property can be proved in the following settings:

- In Cubical Agda, using paths for equality.
- If the equality type is defined as a data type as in Section 2, the K rule is active, and pattern matching is allowed for erased equality proofs. Below I call this setting *traditional Agda*.

3.3 *Erased* commutes

Erased commutes with the unit and empty types, Π and Σ in the following ways (note that Π satisfies two properties):

$$\mathit{Erased}\ \top \simeq \top \tag{8}$$

$$\mathit{Erased}\ \bot \simeq \bot \tag{9}$$

$$\{@0\ P : A \to \mathit{Set}\ p\} \to \\ \quad \mathit{Erased}\ ((x : A) \to P\ x) \simeq ((x : A) \to \mathit{Erased}\ (P\ x)) \tag{10}$$

$$\{@0\ A : \mathit{Set}\ a\}\ \{@0\ P : A \to \mathit{Set}\ p\} \to \\ \quad \mathit{Erased}\ ((x : A) \to P\ x) \simeq \\ \quad ((x : \mathit{Erased}\ A) \to \mathit{Erased}\ (P\ (\mathsf{erased}\ x))) \tag{11}$$

$$\{@0\ A : \mathit{Set}\ a\}\ \{@0\ P : A \to \mathit{Set}\ p\} \to \\ \quad \mathit{Erased}\ (\Sigma\ A\ P) \simeq \Sigma\ (\mathit{Erased}\ A)\ (\lambda\ x \to \mathit{Erased}\ (P\ (\mathsf{erased}\ x))) \tag{12}$$

The proofs are very simple.

Does *Erased* commute with W-types, defined in the following way?

$$\begin{array}{l}\textbf{data}\ W\ (A : \mathit{Set}\ a)\ (P : A \to \mathit{Set}\ p) : \mathit{Set}\ (a \sqcup p)\ \textbf{where} \\ \quad \mathsf{sup} : (x : A) \to (P\ x \to W\ A\ P) \to W\ A\ P\end{array} \tag{13}$$

(Here $(a \sqcup p)$ is the maximum of the universe levels a and p.) We can start by proving the following logical equivalence:

$$\{@0\ A : \mathit{Set}\ a\}\ \{@0\ P : A \to \mathit{Set}\ p\} \to \\ \mathit{Erased}\ (W\ A\ P) \Leftrightarrow W\ (\mathit{Erased}\ A)\ (\lambda\ x \to \mathit{Erased}\ (P\ (\mathsf{erased}\ x))) \tag{14}$$

The right-to-left direction is easy to implement:

$$\mathit{from}\ (\mathsf{sup}\ [\ x\ ]\ f) = [\ \mathsf{sup}\ x\ (\lambda\ y \to \mathsf{erased}\ (\mathit{from}\ (f\ [\ y\ ])))\ ]$$

The other direction can be defined if matching is allowed for erased arguments of W-types:

$$\mathit{to}\ [\ \mathsf{sup}\ x\ f\ ] = \mathsf{sup}\ [\ x\ ]\ (\lambda\ ([\ y\ ]) \to \mathit{to}\ [\ f\ y\ ])$$

It is also easy to prove that these two functions are inverses of each other, given the assumptions that equality is extensional for functions, and that we can prove $[]$-*cong*.

3.4 The $[]$-*cong* property

$[]$-*cong* is a lemma with the following type:

$$[]\text{-}\mathit{cong} : \{@0\ A : \mathit{Set}\ a\}\ \{@0\ x\ y : A\} \to \\ \quad \mathit{Erased}\ (x \equiv y) \to [\ x\ ] \equiv [\ y\ ]$$

Given an erased proof of equality between two (erased) values of type A this lemma returns a proof of equality between two values of type *Erased* A.

A large part of the development below relies on this lemma. I have not managed to prove the lemma using just the J rule (6). The problem is that none of the arguments of the J rule are allowed to be erased (in a non-erased context). However, it is easy to prove the lemma in traditional Agda:

$$[]\text{-}\mathit{cong}\ [\ \mathsf{refl}\ ] = \mathsf{refl} \tag{15}$$

It is also easy to prove the lemma in Cubical Agda, if _≡_ stands for the path type family. A path of type _≡_ $\{A = A\}\ x\ y$ (where $\{A = B\}$ means that the implicit argument A is B) is a kind of function f from the "interval" to A,

subject to the restriction that $f\ 0$ has to be definitionally equal to x, and $f\ 1$ to y, where 0 and 1 are the two endpoints of the interval. Thus the lemma can be proved in the following way:

$$[]\text{-}cong\ [\ eq\ ] = \lambda\ i \to [\ eq\ i\ ] \tag{16}$$

Note that the erased proof *eq* is only used in an erased context, and that there is no pattern matching on erased arguments.

Every family of equality types satisfying the axioms (6) given in Section 3.2 is pointwise equivalent to every other, with the equivalences mapping reflexivity to reflexivity. This fact can be used to define []-*cong* for any such family of equality types, in terms of []-*cong* for paths (see the accompanying code for details). Thus we get that []-*cong* can be defined also for Cubical Agda's identity type family.

All definitions of []-*cong* mentioned above satisfy further properties that will be used below:

$$\begin{aligned}&\{@0\ A : Set\ a\}\ \{@0\ x\ y : A\} \to \\ &\quad Is\text{-}equivalence\ (\lambda\ (eq : Erased\ (x \equiv y)) \to []\text{-}cong\ eq)\end{aligned} \tag{17}$$

$$\{@0\ A : Set\ a\}\ \{@0\ x : A\} \to []\text{-}cong\ [\ refl\ x\ ] \equiv refl\ [\ x\ ] \tag{18}$$

[]-*cong* is an equivalence that interacts with the canonical proof of reflexivity in a specific way. The latter property will be referred to as the computation rule for []-*cong*.

As we will see in Section 4.6 it is also possible to define []-*cong*, in such a way that the two properties above hold, given the assumption that equality of functions is extensional.

3.5 H-levels

In homotopy type theory/univalent foundations there is the concept of an n-type (The Univalent Foundations Program 2013). I use the following definition, numbering the levels from 0 rather than -2:

$$\begin{aligned}&H\text{-}level : \mathbb{N} \to Set\ a \to Set\ a \\ &H\text{-}level\ n\ A = For\text{-}iterated\text{-}equality\ n\ Contractible\ A\end{aligned} \tag{19}$$

A type A has h-level n if every iterated equality type with n levels above A is contractible. *For-iterated-equality* and *Contractible* are defined in the following way:

$$\begin{aligned}&For\text{-}iterated\text{-}equality : \mathbb{N} \to (Set\ a \to Set\ a) \to (Set\ a \to Set\ a) \\ &For\text{-}iterated\text{-}equality\ \mathsf{zero}\quad P\ A = P\ A \\ &For\text{-}iterated\text{-}equality\ (\mathsf{suc}\ n)\ P\ A = \\ &\quad (x\ y : A) \to For\text{-}iterated\text{-}equality\ n\ P\ (x \equiv y)\end{aligned} \tag{20}$$

$$\begin{aligned}&Contractible : Set\ a \to Set\ a \\ &Contractible\ A = \Sigma\ A\ (\lambda\ x \to \forall\ y \to x \equiv y)\end{aligned} \tag{21}$$

How does *Erased* interact with *H-level*? We can prove that *Erased* commutes with *H-level n*:

$$\begin{aligned}&\{@0\ A : Set\ a\} \to \\ &\forall\ n \to Erased\ (H\text{-}level\ n\ A) \simeq H\text{-}level\ n\ (Erased\ A)\end{aligned} \tag{22}$$

The proof proceeds by induction on the natural number. In the base case we can calculate in the following way:

$$\begin{array}{ll}
\mathit{Erased}\ (\mathit{H\text{-}level}\ \mathsf{zero}\ A) & \simeq \\
\mathit{Erased}\ (\Sigma\ A\ (\lambda\ x \to (y : A) \to x \equiv y)) & \simeq \\
\Sigma\ (\mathit{Erased}\ A)\ (\lambda\ x \to \mathit{Erased}\ ((y : A) \to \mathsf{erased}\ x \equiv y)) & \simeq \\
\Sigma\ (\mathit{Erased}\ A)\ (\lambda\ x \to & \\
\quad (y : \mathit{Erased}\ A) \to \mathit{Erased}\ (\mathsf{erased}\ x \equiv \mathsf{erased}\ y)) & \simeq \\
\Sigma\ (\mathit{Erased}\ A)\ (\lambda\ x \to (y : \mathit{Erased}\ A) \to x \equiv y) & \simeq \\
\mathit{H\text{-}level}\ \mathsf{zero}\ (\mathit{Erased}\ A) &
\end{array}$$

The first and last steps hold by definition, the second and third steps use commutation properties from Section 3.3, and the fourth step uses the assumption that []-*cong* is an equivalence. In the step case we can calculate in the following way:

$$\begin{array}{ll}
\mathit{Erased}\ (\mathit{H\text{-}level}\ (\mathsf{suc}\ n)\ A) & \simeq \\
\mathit{Erased}\ ((x\ y : A) \to \mathit{H\text{-}level}\ n\ (x \equiv y)) & \simeq \\
((x\ y : \mathit{Erased}\ A) \to \mathit{Erased}\ (\mathit{H\text{-}level}\ n\ (\mathsf{erased}\ x \equiv \mathsf{erased}\ y))) & \simeq \\
((x\ y : \mathit{Erased}\ A) \to \mathit{H\text{-}level}\ n\ (\mathit{Erased}\ (\mathsf{erased}\ x \equiv \mathsf{erased}\ y))) & \simeq \\
((x\ y : \mathit{Erased}\ A) \to \mathit{H\text{-}level}\ n\ (x \equiv y)) & \simeq \\
\mathit{H\text{-}level}\ (\mathsf{suc}\ n)\ (\mathit{Erased}\ A) &
\end{array}$$

Again the first and last steps hold by definition. The second step uses a commutation property for Π (11) twice, the third step uses the induction hypothesis, and the fourth step uses the assumption that []-*cong* is an equivalence.

The proof above makes use of extensionality for functions, which is used to prove that *H-level n* preserves equivalences, and similarly for Π. However, one can prove that *Erased* commutes with *H-level n* up to *logical* equivalence without making use of extensionality:

$$\begin{array}{l}
\{@0\ A : \mathit{Set}\ a\} \to \\
\forall\ n \to \mathit{Erased}\ (\mathit{H\text{-}level}\ n\ A) \Leftrightarrow \mathit{H\text{-}level}\ n\ (\mathit{Erased}\ A)
\end{array} \tag{23}$$

3.6 *Erased* is a modality

Now we can show that *Erased* is a left exact (or lex) modality in the sense of Rijke et al. (2019), with *Erased* as the modal operator and [_] as the modal unit. First one can show that *Erased* is a (uniquely eliminating) modality:

$$\mathit{Is\text{-}equivalence}\ (\lambda\ (f : (x : \mathit{Erased}\ A) \to \mathit{Erased}\ (P\ x)) \to f \circ [\_]) \tag{24}$$

The function $\lambda\ (f : (x : \mathit{Erased}\ A) \to \mathit{Erased}\ (P\ x)) \to f \circ [\_]$ has the following type:

$$((x : \mathit{Erased}\ A) \to \mathit{Erased}\ (P\ x)) \to ((x : A) \to \mathit{Erased}\ (P\ [\ x\ ]))$$

It is easy to prove that the domain and the codomain are equivalent using two commutation properties for Π (Lemmas 10 and 11):

$$\begin{array}{ll}
((x : \mathit{Erased}\ A) \to \mathit{Erased}\ (P\ x)) & \simeq \\
\mathit{Erased}\ ((x : A) \to (P\ [\ x\ ])) & \simeq \\
((x : A) \to \mathit{Erased}\ (P\ [\ x\ ])) &
\end{array}$$

If these two equivalences are defined as in the accompanying code, then it turns out that the forward direction of the obtained equivalence is definitionally equal to the given function.

The modality is also left exact:

$$\begin{aligned}&\{x\ y : A\} \to \\ &\mathit{Contractible}\ (\mathit{Erased}\ A) \to \mathit{Contractible}\ (\mathit{Erased}\ (x \equiv y))\end{aligned} \tag{25}$$

Given that *Erased* A is contractible we get an erased proof of *Contractible* A (using the fact that *Erased* commutes with *H-level* n up to logical equivalence (23)). We can use the following *map* function to turn this into an erased proof of *Contractible* $(x \equiv y)$:

$$\begin{aligned}&\mathit{map} : \\ &\quad \{@0\ A : \mathit{Set}\ a\}\ \{@0\ P : A \to \mathit{Set}\ p\} \to \\ &\quad @0\ ((x : A) \to P\ x) \to (x : \mathit{Erased}\ A) \to \mathit{Erased}\ (P\ (\mathsf{erased}\ x)) \\ &\mathit{map}\ f\ [\ x\ ] = [\ f\ x\ ]\end{aligned} \tag{26}$$

Finally we can commute *Erased* and *Contractible* again, thus showing that *Erased* $(x \equiv y)$ is contractible.

Rijke et al. (2019) have developed a fair amount of theory for modalities, and some of the theory in this text is an instance of their theory. Some connections are pointed out below, and some results above that were not used to prove that *Erased* is a (lex) modality could perhaps have been obtained via their general theory. For instance, Rijke et al. prove that *map* [_] is an equivalence (Lemma 1.22), and that lex modalities preserve n-types for $n \geq -1$ (see the proof of Corollary 3.9).

However, Rijke et al. state that they "will freely use function extensionality and the univalence axiom, often without comment". In this text I try to avoid using these axioms/constructions, and document when they are used, because I want the results to be applicable in multiple settings (for instance in traditional Agda, which is incompatible with univalence, and in which it should not be possible to prove function extensionality).

Rijke et al. are more careful in the Coq code that accompanies their paper. For instance, while they prove that *map* [_] is an equivalence for any reflective subuniverse without using any extra assumptions (they use a definition of reflective subuniverse that differs from the one in their paper, partly to avoid uses of extensionality for functions), the proof showing that lex modalities preserve n-types for $n \geq -1$ does depend on extensionality for functions. I have not tried to find every relevant lemma from their paper in their Coq code to find out exactly what assumptions are used for each result (and some results from the paper are not formalised in the Coq development). However, I have proved that, if []-*cong* can be proved, then *Erased* is a Σ-closed reflective subuniverse, using a definition of reflective subuniverses that is based on the Coq code of Rijke et al. (see the accompanying code for details).

Rijke et al. define a function $f : A \to B$ to be *Erased*-connected when *Erased* applied to $\Sigma\ A\ (\lambda\ x \to f\ x \equiv y)$ is contractible for every y. In the present setting this property can be expressed in a different way:

$$\begin{aligned}&((y : B) \to \mathit{Contractible}\ (\mathit{Erased}\ (\Sigma\ A\ (\lambda\ x \to f\ x \equiv y)))) \simeq \\ &\mathit{Erased}\ (\mathit{Is\text{-}equivalence}\ f)\end{aligned} \tag{27}$$

This equivalence can be proved assuming that equality is extensional for functions. In the absence of this assumption one can still prove the corresponding logical equivalence. The equivalence can be established through the following calculation:

$$\begin{aligned}
&((y : B) \to \mathit{Contractible}\ (\mathit{Erased}\ (\Sigma\ A\ (\lambda\ x \to f\ x \equiv y)))) &&\simeq \\
&((y : B) \to \mathit{Erased}\ (\mathit{Contractible}\ (\Sigma\ A\ (\lambda\ x \to f\ x \equiv y)))) &&\simeq \\
&\mathit{Erased}\ ((y : B) \to \mathit{Contractible}\ (\Sigma\ A\ (\lambda\ x \to f\ x \equiv y))) &&\simeq \\
&\mathit{Erased}\ (\mathit{Is\text{-}equivalence}\ f)
\end{aligned}$$

The first step uses Lemma 22, the second step uses Lemma 10, and the final step holds by definition. (*Is-equivalence* can be defined in several ways. If another, equivalent definition were used, then the final step could also make use of the fact that *Erased* preserves equivalences, which is proved below (34).)

3.7 Is [_] an embedding?

In general one should perhaps not expect it to be possible to prove that [_] is an equivalence, because this would mean that one could always resurrect identical copies of erased data. However, can one at least show that [_] is injective, or an embedding?

A function f is an embedding if the functions of type $x \equiv y \to f\ x \equiv f\ y$ obtained from $\mathit{cong} : (f : A \to B) \to x \equiv y \to f\ x \equiv f\ y$ (which is unique up to pointwise equality if it is required to map the canonical proof of reflexivity to the canonical proof of reflexivity) are always equivalences:

$$\begin{aligned}
&\mathit{Is\text{-}embedding} : \{A : \mathit{Set}\ a\}\ \{B : \mathit{Set}\ b\} \to (A \to B) \to \mathit{Set}\ (a \sqcup b) \\
&\mathit{Is\text{-}embedding}\ f = \\
&\quad \forall\ x\ y \to \mathit{Is\text{-}equivalence}\ (\lambda\ (\mathit{eq} : x \equiv y) \to \mathit{cong}\ f\ \mathit{eq})
\end{aligned} \tag{28}$$

Note that embeddings are injective.

To start with we can note that [_] is an embedding when the underlying type is a (mere) proposition (i.e. if it has h-level 1):

$$\mathit{H\text{-}level}\ 1\ A \to \mathit{Is\text{-}embedding}\ [\ A\ |\_] \tag{29}$$

A function between sets (types of h-level 2) is an embedding if and only if it is injective, and a function from a proposition is always injective. (Note that the type of $[\ A\ |\_]$ is $@0\ A \to \mathit{Erased}\ A$. Agda supports subtyping for @0, which is why the expression can be used at type $A \to \mathit{Erased}\ A$ above.)

If we restrict ourselves to traditional Agda, then [_] is always injective and an embedding (the proofs are easy and omitted):

$$\{@0\ A : \mathit{Set}\ a\} \to \mathit{Injective}\ \ [\ A\ |\_] \tag{30}$$

$$\{@0\ A : \mathit{Set}\ a\} \to \mathit{Is\text{-}embedding}\ [\ A\ |\_] \tag{31}$$

In this setting it is also easy to prove that A is a proposition whenever $\mathit{Erased}\ A$ is:

$$\{@0\ A : \mathit{Set}\ a\} \to \mathit{H\text{-}level}\ 1\ (\mathit{Erased}\ A) \to \mathit{H\text{-}level}\ 1\ A \tag{32}$$

This result strengthens Lemma 23. (There is no point in considering h-levels greater than 1, because in this setting every type is a set.)

In Section 4 we will see that [_] can be an embedding also in cases not covered above.

3.8 More commutation properties

In Section 3.3 we saw that *Erased* commutes with Π (11). Here we will see that *Erased* commutes with several other kinds of "function formers"—those for logical equivalences, equivalences, split surjections (or retractions), functions with quasi-inverses, injections and embeddings (using definitions based on those of The Univalent Foundations Program (2013)):

$$\begin{aligned}
&\mathit{Erased}\ (A \mathbin{⇔} B) && \simeq (\mathit{Erased}\ A \mathbin{⇔} \mathit{Erased}\ B) && (33)\\
&\mathit{Erased}\ (A \simeq B) && \simeq (\mathit{Erased}\ A \simeq \mathit{Erased}\ B) && (34)\\
&\mathit{Erased}\ (A \twoheadrightarrow B) && \simeq (\mathit{Erased}\ A \twoheadrightarrow \mathit{Erased}\ B) && (35)\\
&\mathit{Erased}\ (A \leftrightarrow B) && \simeq (\mathit{Erased}\ A \leftrightarrow \mathit{Erased}\ B) && (36)\\
&\mathit{Erased}\ (A \rightarrowtail B) && \simeq (\mathit{Erased}\ A \rightarrowtail \mathit{Erased}\ B) && (37)\\
&\mathit{Erased}\ (\mathit{Embedding}\ A\ B) && \simeq \mathit{Embedding}\ (\mathit{Erased}\ A)\ (\mathit{Erased}\ B) && (38)
\end{aligned}$$

(In all these cases $A : \mathit{Set}\ a$ and $B : \mathit{Set}\ b$ are erased.) All these results, except for the one for logical equivalences, have been proved assuming that equality of functions is extensional. However, if the results are stated as logical equivalences instead of equivalences, then they can be proved without making use of extensionality.

Let us start with the proof for logical equivalences. The proof makes use of the following preservation lemma for equivalences, which is easy to prove (using $[]$-*cong*):

$$\begin{aligned}
&\{@0\ A : \mathit{Set}\ a\}\ \{@0\ B : \mathit{Set}\ b\} \rightarrow\\
&@0\ A \simeq B \rightarrow \mathit{Erased}\ A \simeq \mathit{Erased}\ B
\end{aligned} \tag{39}$$

We can calculate in the following way:

$$\begin{aligned}
&\mathit{Erased}\ (A \mathbin{⇔} B) && \simeq\\
&\mathit{Erased}\ ((A \rightarrow B) \times (B \rightarrow A)) && \simeq\\
&\mathit{Erased}\ (A \rightarrow B) \times \mathit{Erased}\ (B \rightarrow A) && \simeq\\
&(\mathit{Erased}\ A \rightarrow \mathit{Erased}\ B) \times (\mathit{Erased}\ B \rightarrow \mathit{Erased}\ A) && \simeq\\
&(\mathit{Erased}\ A \mathbin{⇔} \mathit{Erased}\ B)
\end{aligned}$$

The first and last steps use the fact that a logical equivalence is equivalent to two functions, and the first step also uses Lemma 39. The second and third steps use commutation properties from Section 3.3.

Let us now prove the following equivalence, which refers to the *map* function (26):

$$\begin{aligned}
&\{@0\ A : \mathit{Set}\ a\}\ \{@0\ B : \mathit{Set}\ b\}\ \{@0\ f : A \rightarrow B\} \rightarrow\\
&\mathit{Erased}\ (\mathit{Is\text{-}equivalence}\ f) \simeq \mathit{Is\text{-}equivalence}\ (\mathit{map}\ f)
\end{aligned} \tag{40}$$

Given Lemma 27 this equivalence is related to some results due to Rijke et al. (2019, Lemma 1.35 and Theorem 3.1 (xii)). The proof is a simple calculation (with $\exists\ P = \Sigma\ \_\ P$):

$$\begin{aligned}
&\mathit{Erased}\ (\mathit{Is\text{-}equivalence}\ f) && \simeq\\
&\mathit{Erased}\ (\forall\ y \rightarrow \mathit{Contractible}\ (\exists\ (\lambda\ x \rightarrow f\ x \equiv y))) && \simeq\\
&(\forall\ y \rightarrow \mathit{Erased}\ (\mathit{Contractible}\ (\exists\ (\lambda\ x \rightarrow f\ x \equiv \mathsf{erased}\ y)))) && \simeq\\
&(\forall\ y \rightarrow \mathit{Contractible}\ (\mathit{Erased}\ (\exists\ (\lambda\ x \rightarrow f\ x \equiv \mathsf{erased}\ y)))) && \simeq
\end{aligned}$$

$$
\begin{array}{ll}
(\forall\ y \to \mathit{Contractible}\ (\exists\ (\lambda\ x \to \mathit{Erased}\ (f\ (\mathsf{erased}\ x) \equiv \mathsf{erased}\ y)))) & \simeq \\
(\forall\ y \to \mathit{Contractible}\ (\exists\ (\lambda\ x \to \mathit{map}\ f\ x \equiv y))) & \simeq \\
\mathit{Is\text{-}equivalence}\ (\mathit{map}\ f) &
\end{array}
$$

The first and last steps are unfoldings of the definition of *Is-equivalence*, the second and fourth steps use commutation properties from Section 3.3, the third step uses the commutation property for *H-level* (22), and the fifth step uses the assumption that []-*cong* is an equivalence. (The proof makes use of extensionality for functions; again this is not needed if the result is stated as a logical equivalence instead of as an equivalence.)

Using this property we can now prove that *Erased* commutes with _≃_:

$$
\begin{array}{ll}
\mathit{Erased}\ (A \simeq B) & \simeq \\
\mathit{Erased}\ (\Sigma\ (A \to B)\ (\lambda\ f \to \mathit{Is\text{-}equivalence}\ f)) & \simeq \\
\Sigma\ (\mathit{Erased}\ (A \to B))\ (\lambda\ f \to \mathit{Erased}\ (\mathit{Is\text{-}equivalence}\ (\mathsf{erased}\ f))) & \simeq \\
\Sigma\ (\mathit{Erased}\ A \to \mathit{Erased}\ B)\ (\lambda\ f \to \mathit{Is\text{-}equivalence}\ f) & \simeq \\
\mathit{Erased}\ A \simeq \mathit{Erased}\ B &
\end{array}
$$

The third step uses a commutation property (11), the forward direction of which is $\lambda\ ([\ f\ ]) \to \mathit{map}\ f$, the previously proved equivalence for *Is-equivalence*, and the following preservation result (where _≃_.to gives the forward direction of an equivalence):

$$(f : A \simeq B) \to (\forall\ x \to P\ x \simeq Q\ (\_\simeq\_.\mathsf{to}\ f\ x)) \to \Sigma\ A\ P \simeq \Sigma\ B\ Q \quad (41)$$

The proofs for split surjections, functions with quasi-inverses, injections, and embeddings are similar, relying on the following lemmas, where A, B and $f : A \to B$ are erased:

$$
\begin{array}{lll}
\mathit{Erased}\ (\mathit{Split\text{-}surjective}\ f) & \simeq \mathit{Split\text{-}surjective}\ (\mathit{map}\ f) & (42) \\
\mathit{Erased}\ (\mathit{Has\text{-}quasi\text{-}inverse}\ f) & \simeq \mathit{Has\text{-}quasi\text{-}inverse}\ (\mathit{map}\ f) & (43) \\
\mathit{Erased}\ (\mathit{Injective}\ f) & \simeq \mathit{Injective}\ (\mathit{map}\ f) & (44) \\
\mathit{Erased}\ (\mathit{Is\text{-}embedding}\ f) & \simeq \mathit{Is\text{-}embedding}\ (\mathit{map}\ f) & (45)
\end{array}
$$

These lemmas can be proved by making use of the assumption that []-*cong* is an equivalence (as well as extensionality for functions; yet again extensionality is not required if the results are stated as logical equivalences instead of equivalences). Let us study the proof of the last lemma. Again it is a calculation:

$$
\begin{array}{ll}
\mathit{Erased}\ (\mathit{Is\text{-}embedding}\ f) & \simeq \\
\mathit{Erased}\ (\forall\ x\ y \to \mathit{Is\text{-}equivalence}\ (\lambda\ (eq : x \equiv y) \to \mathit{cong}\ f\ eq)) & \simeq \\
(\forall\ x\ y \to \mathit{Erased}\ (\mathit{Is\text{-}equivalence} & \\
\qquad (\lambda\ (eq : \mathsf{erased}\ x \equiv \mathsf{erased}\ y) \to \mathit{cong}\ f\ eq))) & \simeq \\
(\forall\ x\ y \to \mathit{Is\text{-}equivalence} & \\
\qquad (\mathit{map}\ (\lambda\ (eq : \mathsf{erased}\ x \equiv \mathsf{erased}\ y) \to \mathit{cong}\ f\ eq))) & \simeq \\
(\forall\ x\ y \to \mathit{Is\text{-}equivalence} & \\
\qquad (\lambda\ (eq : \mathit{Erased}\ (\mathsf{erased}\ x \equiv \mathsf{erased}\ y)) \to & \\
\qquad\quad []\text{-}\mathit{cong}^{-1}\ (\mathit{cong}\ (\mathit{map}\ f)\ ([]\text{-}\mathit{cong}\ eq)))) & \simeq \\
(\forall\ x\ y \to \mathit{Is\text{-}equivalence}\ (\lambda\ (eq : x \equiv y) \to \mathit{cong}\ (\mathit{map}\ f)\ eq)) & \simeq \\
\mathit{Is\text{-}embedding}\ (\mathit{map}\ f) &
\end{array}
$$

The first and last steps hold by definition, the second step uses a commutation property for Π twice, and the third step uses the equivalence for *Is-equivalence*

proved above. The fifth step uses the 2-out-of-3 property (The Univalent Foundations Program 2013) and the assumption that []-*cong* is an equivalence. The fourth step uses the fact that *Is-equivalence f* is equivalent to *Is-equivalence g* when the functions f and g have the same type and are pointwise equal (assuming that equality of functions is extensional). It also uses the following family of equalities, where []-$cong^{-1}$ is the inverse of []-*cong* obtained from the assumption that []-*cong* is an equivalence:

$$\begin{aligned}&\{@0\ A : Set\ a\}\ \{@0\ B : Set\ b\}\ \{@0\ x\ y : A\}\\&\{@0\ f : A \to B\}\ \{eq : Erased\ (x \equiv y)\} \to\\&map\ (cong\ f)\ eq \equiv []\text{-}cong^{-1}\ (cong\ (map\ f)\ ([]\text{-}cong\ eq))\end{aligned} \tag{46}$$

This lemma can be proved by using the J rule and the computation rule for []-*cong*.

The commutation properties established above can be turned into preservation lemmas (where A and B are erased):

$$@0\ A \leftrightarrows B \to Erased\ A \leftrightarrows Erased\ B \tag{47}$$
$$@0\ A \simeq B \to Erased\ A \simeq Erased\ B \tag{48}$$
$$@0\ A \twoheadrightarrow B \to Erased\ A \twoheadrightarrow Erased\ B \tag{49}$$
$$@0\ A \leftrightarrow B \to Erased\ A \leftrightarrow Erased\ B \tag{50}$$
$$@0\ A \rightarrowtail B \to Erased\ A \rightarrowtail Erased\ B \tag{51}$$
$$@0\ Embedding\ A\ B \to Embedding\ (Erased\ A)\ (Erased\ B) \tag{52}$$

(Lemma 48 has the same type as Lemma 39.) Note that these lemmas can be proved without making use of extensionality for functions.

The *map* function is functorial, in the sense that it maps identity to identity and commutes with composition (Rijke et al. (2019, Lemma 1.21) prove that there is a non-dependent variant of *map* that preserves identity and composition up to homotopy). This is true also for these preservation lemmas, at least when they are proved exactly as in the accompanying code. The proof is easy for logical equivalences. The proofs are also easy for equivalences and embeddings, because these kinds of functions are defined as functions that satisfy *propositional* properties (assuming that equality of functions is extensional). The proofs are a little trickier for injections, split surjections and functions with quasi-inverses, see the accompanying code for details (in all three cases the proofs make use of extensionality, and in the latter two cases they also make use of the computation rule for []-*cong*, as well as some results from Section 4).

4 Stability

When can erased values be resurrected? This section discusses two notions of *stability*.

4.1 Stable types

A type A is stable if *Erased A* implies A:

$$\begin{aligned}&Stable : Set\ a \to Set\ a\\&Stable\ A = Erased\ A \to A\end{aligned} \tag{53}$$

The term stability is also used for stability under double-negation. The two concepts are related. Stability under double-negation implies stability under *Erased*:

$$\{@0\ A : \mathit{Set}\ a\} \to (\neg\ \neg\ A \to A) \to \mathit{Stable}\ A \tag{54}$$

This follows from the following fact, which is easy to prove:

$$\{@0\ A : \mathit{Set}\ a\} \to \mathit{Erased}\ A \to \neg\ \neg\ A \tag{55}$$

Types for which it is known whether or not they are inhabited are also stable (*Dec A* is the binary sum $A \uplus \neg\ A$):

$$\{@0\ A : \mathit{Set}\ a\} \to \mathit{Dec}\ A \to \mathit{Stable}\ A \tag{56}$$

If the type A is inhabited, then it is stable, and if it is not inhabited, then the assumption that *Erased A* is inhabited leads to a contradiction.

4.2 Very stable types

A type is *very stable* if [_] is an equivalence:

$$\begin{aligned} &\mathit{Very\text{-}stable} : \mathit{Set}\ a \to \mathit{Set}\ a \\ &\mathit{Very\text{-}stable}\ A = \mathit{Is\text{-}equivalence}\ [\ A\ |\_] \end{aligned} \tag{57}$$

The property of being very stable is perhaps more interesting than plain stability, because for very stable types one can resurrect the erased element, not just some element of the same type.

The definition of *Very-stable A* says that A is *modal* with respect to the *Erased* modality (Rijke et al. 2019). I was not aware of the concept of a modal type when I came up with the definition above. Before I had tried something like the following definition:

$$\begin{aligned} &\mathit{Very\text{-}stable'} : \mathit{Set}\ a \to \mathit{Set}\ a \\ &\mathit{Very\text{-}stable'}\ A = \mathit{Erased}\ A \simeq A \end{aligned} \tag{58}$$

However, I found it easier to work with the former definition (see Lemma 81), and I have not found a way to prove that *Very-stable′ A* implies *Very-stable A* in a non-erased context (the other direction is immediate).

Note that very stable types are stable. Because equivalences are embeddings we also get that [_] is an embedding for very stable types.

Stable types are very stable if [_] is a right inverse of the proof of stability (Rijke et al. 2019, Lemma 1.20):

$$(s : \mathit{Stable}\ A) \to (\forall\ x \to s\ [\ x\ ] \equiv x) \to \mathit{Very\text{-}stable}\ A \tag{59}$$

In this case one can prove that [_] is also a left inverse of s: it suffices to prove that $[\ s\ [\ x\ ]\ ] \equiv [\ x\ ]$ for an arbitrary erased x, which follows from []-*cong* and the assumption that s is a right inverse of [_].

Erased A is a proposition if A is (23), so when A is a stable proposition [_] has an inverse. Thus stable propositions are very stable:

$$\mathit{Stable}\ A \to \mathit{H\text{-}level}\ 1\ A \to \mathit{Very\text{-}stable}\ A \tag{60}$$

However, it is not the case that every very stable type is a proposition. *Erased A* is always very stable (because the function $\lambda\ ([\ x\ ]) \to [\ \mathsf{erased}\ x\ ]$ is an inverse of [_]; see also Rijke et al. (2019, Lemma 1.11)):

$$\{@0\ A : \mathit{Set}\ a\} \to \mathit{Very\text{-}stable}\ (\mathit{Erased}\ A) \tag{61}$$

Thus, if every very stable type were a proposition, then *Erased Bool* would be a proposition, but it is not (23).

Rijke et al. (2019, Theorem 3.11) prove that an *accessible* modality is left exact if and only if a certain universe of modal types is modal in a certain sense. I do not know if *Erased* is accessible, but I can prove that universes of very stable types are very stable (assuming extensionality for functions and univalence):

$$\mathit{Very\text{-}stable}\ (\Sigma\ (\mathit{Set}\ a)\ \ \mathit{Very\text{-}stable}) \tag{62}$$

By Lemma 59 it suffices to prove that there is a stability proof which is a left inverse of [_]. It is easy to prove that the type is stable (this result does not rely on []-*cong*, function extensionality or univalence):

$$\begin{aligned} &\mathit{stable} : \mathit{Stable}\ (\Sigma\ (\mathit{Set}\ a)\ \ \mathit{Very\text{-}stable}) \\ &\mathit{stable}\ [\ A\ ] = (\mathit{Erased}\ (\mathsf{proj}_1\ A)\ ,\ \mathit{Very\text{-}stable\text{-}Erased}) \end{aligned} \tag{63}$$

Here proj_1 returns the first projection of a pair, and *Very-stable-Erased* is a proof of Lemma 61. To prove that $\mathit{stable}\ [\ (A\ ,\ s)\ ]$ is equal to $(A\ ,\ s)$ we can note that it suffices to prove that the first projections are equal, because *Very-stable* is propositional (assuming extensionality for functions). The first projections are *Erased A* and *A*. By the assumption $s : \mathit{Very\text{-}stable}\ A$ these types are equivalent, so by univalence it follows that they are equal.

4.3 Stability for equality types

Let us now investigate stability for equality types. Equality is (very) stable for *A* if $x \equiv y$ is (very) stable for all $x,\ y\ :\ A$:

$$\begin{aligned} &\mathit{Stable\text{-}}{\equiv} : \mathit{Set}\ a \to \mathit{Set}\ a \\ &\mathit{Stable\text{-}}{\equiv} = \mathit{For\text{-}iterated\text{-}equality}\ 1\ \mathit{Stable} \end{aligned} \tag{64}$$

$$\begin{aligned} &\mathit{Very\text{-}stable\text{-}}{\equiv} : \mathit{Set}\ a \to \mathit{Set}\ a \\ &\mathit{Very\text{-}stable\text{-}}{\equiv} = \mathit{For\text{-}iterated\text{-}equality}\ 1\ \mathit{Very\text{-}stable} \end{aligned} \tag{65}$$

Above it was proved that stable propositions are very stable. This result can be generalised:

$$\begin{aligned} &\forall\ n \to \mathit{For\text{-}iterated\text{-}equality}\ n\ \mathit{Stable}\ A \to \mathit{H\text{-}level}\ (1 + n)\ A \to \\ &\mathit{For\text{-}iterated\text{-}equality}\ n\ \mathit{Very\text{-}stable}\ A \end{aligned} \tag{66}$$

For *n* equal to 1 we get that equality is very stable for *A* if it is stable and *A* is a set. If equality is decidable for a type, then equality is stable (56). Furthermore the type is a set (Hedberg 1998). Thus we get the following lemma:

$$((x\ y : A) \to \mathit{Dec}\ (x \equiv y)) \to \mathit{Very\text{-}stable\text{-}}{\equiv}\ A \tag{67}$$

Contractible types are very stable because they are stable propositions. This result can be generalised (using a preservation lemma for *For-iterated-equality*):

$$\forall\ n \to \mathit{H\text{-}level}\ n\ A \to \mathit{For\text{-}iterated\text{-}equality}\ n\ \mathit{Very\text{-}stable}\ A \tag{68}$$

For n equal to 1 we get that equality is very stable for propositions.

Equality is stable for a type if and only if $[\_]$ is injective:

$$\mathit{Stable\text{-}}{\equiv}\ A \Leftrightarrow \mathit{Injective}\ [\ A\ |\_] \tag{69}$$

These two types are very similar: one uses $\mathit{Erased}\ (x \equiv y)$ and one $[\ x\ ] \equiv [\ y\ ]$, so the result follows from the assumption that $[]$-*cong* is an equivalence. If equality is extensional for functions, then the logical equivalence can be strengthened to an equivalence.

In a similar vein equality is *very* stable for a type if and only if $[\_]$ is an *embedding*:

$$\mathit{Very\text{-}stable\text{-}}{\equiv}\ A \Leftrightarrow \mathit{Is\text{-}embedding}\ [\ A\ |\_] \tag{70}$$

We can prove this result using the following calculation:

$$\begin{array}{ll}
\mathit{Very\text{-}stable\text{-}}{\equiv}\ A & \Leftrightarrow \\
((x\ y : A) \to \mathit{Is\text{-}equivalence}\ (\lambda\ (eq : x \equiv y) \to [\ eq\ ])) & \Leftrightarrow \\
((x\ y : A) \to \mathit{Is\text{-}equivalence}\ (\lambda\ (eq : x \equiv y) \to []\text{-}\mathit{cong}\ [\ eq\ ])) & \Leftrightarrow \\
((x\ y : A) \to \mathit{Is\text{-}equivalence}\ (\lambda\ (eq : x \equiv y) \to \mathit{cong}\ [_]\ eq)) & \Leftrightarrow \\
\mathit{Is\text{-}embedding}\ [\ A\ |_] &
\end{array}$$

The first and last steps hold by definition, and the second step uses the 2-out-of-3 property and the assumption that $[]$-*cong* is an equivalence. The third step uses the fact that $\mathit{Is\text{-}equivalence}\ f$ is logically equivalent to $\mathit{Is\text{-}equivalence}\ g$ when the functions f and g have the same type and are pointwise equal, as well as the J rule and the computation rule for $[]$-*cong*. Again, if equality is extensional for functions, then the logical equivalence can be strengthened to an equivalence (in this case both sides are propositions).

As mentioned in Section 3.7 $[\_]$ is an embedding in traditional Agda (31). Thus equality is always very stable in this setting.

4.4 Map-like functions

Let us now take a look at some map-like functions for stability. If A and B are logically equivalent and A is stable, then B is stable:

$$A \Leftrightarrow B \to \mathit{Stable}\ A \to \mathit{Stable}\ B \tag{71}$$

One can go from $\mathit{Erased}\ B$ to $\mathit{Erased}\ A$ using the logical equivalence and the *map* function (26), then to A using the stability proof, and finally to B using the other direction of the logical equivalence. This proof does not rely on $[]$-*cong*.

A similar result, stated using equivalence instead of logical equivalence, can be proved for *Very-stable* (using $[]$-*cong*):

$$A \simeq B \to \mathit{Very\text{-}stable}\ A \to \mathit{Very\text{-}stable}\ B \tag{72}$$

If equality is extensional for functions, then the last function space can be replaced by an equivalence (because in that case *Very-stable* is propositional):

$$A \simeq B \to \mathit{Very\text{-}stable}\ A \simeq \mathit{Very\text{-}stable}\ B \tag{73}$$

Lemma 72 can be used to prove the following lemma, where _⇝_ stands for regular functions or any of the "function formers" discussed in Section 3.8:

$$\begin{aligned}&\{@0\ A : \mathit{Set}\ a\}\ \{@0\ B : \mathit{Set}\ b\} \to \\ &\mathit{Very\text{-}stable}\ (\mathit{Erased}\ A \rightsquigarrow \mathit{Erased}\ B)\end{aligned} \tag{74}$$

Erased $(A \rightsquigarrow B)$ is very stable (61), and the commutation properties from Sections 3.3 and 3.8 tell us that *Erased* $(A \rightsquigarrow B)$ is equivalent to *Erased* $A \rightsquigarrow$ *Erased* B. (This result makes use of extensionality in all cases except for regular functions and logical equivalences. One can prove that the types are stable, rather than very stable, without using extensionality.)

4.5 Closure properties

Above we have seen some examples of types that are stable or very stable (in some cases assuming that equality of functions is extensional, and in one case assuming univalence):

- Types that are inhabited or uninhabited are stable.
- Types that are stable under double-negation are stable.
- Equality is stable if and only if [_] is injective.
- Very stable types are stable.
- *Erased* A is very stable.
- *Erased* $A \rightsquigarrow$ *Erased* B is very stable.
- Contractible types are very stable.
- Equality is very stable for propositions.
- Stable types for which [_] is a right inverse of the proof of stability are very stable.
- Stable propositions are very stable.
- Equality is very stable for stable sets.
- Equality is very stable if it is decidable.
- Equality is very stable if and only if [_] is an embedding.
- Universes of very stable types are very stable.

This section contains a number of closure properties (or similar results) that can be used to prove that a type is stable or very stable.

First note that equality is very stable for very stable types (Rijke et al. 2019, Lemma 1.25):

$$\mathit{Very\text{-}stable}\ A \to \mathit{Very\text{-}stable}\text{-}\equiv\ A \tag{75}$$

If A is very stable, then [_] is an equivalence for A, and thus also an embedding, which implies that equality is very stable for A (70). This property can be generalised:

$$\begin{aligned}&\forall\ n \to\\ &\mathit{For\text{-}iterated\text{-}equality}\ n\quad \mathit{Very\text{-}stable}\ A \to\\ &\mathit{For\text{-}iterated\text{-}equality}\ (1 + n)\ \mathit{Very\text{-}stable}\ A\end{aligned} \tag{76}$$

Many properties can be generalised in this way, but such generalised properties are typically not included below.

Some of the closure properties do not rely on []-*cong*. The commutation properties for the unit (8) and empty (9) types can be used to prove that these types are very stable (and thus also stable; for the second property, see also Rijke et al. (2019, Lemma 1.27)):

$$\mathit{Very\text{-}stable}\ \top \tag{77}$$
$$\mathit{Very\text{-}stable}\ \bot \tag{78}$$

The right-to-left directions of these commutation properties are defined to be $[\_]$.

One of the commutation properties for Π (10) can in turn be used to prove the following closure properties (the second proof uses the assumption that equality is extensional for functions; see also Rijke et al. (2019, Lemma 1.26)):

$$(\forall\ x \to \mathit{Stable}\ (P\ x)) \to \mathit{Stable}\ ((x : A) \to P\ x) \tag{79}$$
$$(\forall\ x \to \mathit{Very\text{-}stable}\ (P\ x)) \to \mathit{Very\text{-}stable}\ ((x : A) \to P\ x) \tag{80}$$

$\mathit{Erased}\ ((x : A) \to P\ x)$ is equivalent to $(x : A) \to \mathit{Erased}\ (P\ x)$, which due to stability implies $(x : A) \to P\ x$. For the closure property involving *Very-stable* the second step is an equivalence (assuming that equality is extensional for functions), and one can check that the right-to-left direction of the equivalence from $\mathit{Erased}\ ((x : A) \to P\ x)$ to $(x : A) \to P\ x$ is $[\_]$ (given that the commutation property is defined in a suitable way, and using definitional η-equality for Π).

In a similar way the commutation property for Σ can be used to prove two closure properties (for the second one, see also Rijke et al. (2019, Theorem 1.32)):

$$\mathit{Very\text{-}stable}\ A \to (\forall\ x \to \mathit{Stable}\ (P\ x)) \to \mathit{Stable}\ (\Sigma\ A\ P) \tag{81}$$
$$\begin{aligned}&\mathit{Very\text{-}stable}\ A \to (\forall\ x \to \mathit{Very\text{-}stable}\ (P\ x)) \to\\ &\quad \mathit{Very\text{-}stable}\ (\Sigma\ A\ P)\end{aligned} \tag{82}$$

Note here that the closure property for stability has *Very-stable* A as an assumption, rather than *Stable* A. In fact, I have not been able to prove this property even with *Very-stable′* A as the assumption. The commutation property for Σ gives that $\mathit{Erased}\ (\Sigma\ A\ P)$ is equivalent to $\Sigma\ (\mathit{Erased}\ A)\ (\lambda\ x \to \mathit{Erased}\ (P\ (\mathsf{erased}\ x)))$. How can one prove that the latter type implies $\Sigma\ A\ P$? Given a function s from $\mathit{Erased}\ A$ to A it suffices to find a function of type $(x : \mathit{Erased}\ A) \to \mathit{Erased}\ (P\ (\mathsf{erased}\ x)) \to P\ (s\ x)$. A function of this type is easy to construct from a proof showing that P is pointwise stable if s is a right inverse of $[\_]$.

If the Σ-types are replaced by non-dependent cartesian products, then it suffices to have *Stable* A as the assumption:

$$\mathit{Stable}\ A \to \mathit{Stable}\ B \to \mathit{Stable}\ (A \times B) \tag{83}$$
$$\mathit{Very\text{-}stable}\ A \to \mathit{Very\text{-}stable}\ B \to \mathit{Very\text{-}stable}\ (A \times B) \tag{84}$$

If []-*cong* is available, pattern matching is allowed for erased arguments of single-constructor data types, and equality is extensional for functions, then the following closure property can be proved for W-types:

$$\mathit{Very\text{-}stable}\ A \to \mathit{Very\text{-}stable}\ (W\ A\ P) \tag{85}$$

$W\ A\ P$ is equivalent to $\Sigma\ A\ (\lambda\ x \to P\ x \to W\ A\ P)$, which might explain why I failed to find a similar closure property for plain stability.

If equality is stable for A, then the property of being a proposition is stable for A:

$$\mathit{Stable\text{-}}\equiv\ A \to \mathit{Stable}\ (\mathit{H\text{-}level}\ 1\ A) \tag{86}$$

The fact that A is a proposition if and only if all values of type A are equal means that this result follows from $\mathit{Stable\text{-}}\equiv\ A \to \mathit{Stable}\ ((x\ y : A) \to x \equiv y)$ (using a map-like function (71)). This lemma can in turn be proved using a closure property for Π (10). The result can be generalised:

$$\begin{aligned}&\forall\ n \to \mathit{For\text{-}iterated\text{-}equality}\ (1 + n)\ \mathit{Stable}\ A \to\\ &\mathit{Stable}\ (\mathit{H\text{-}level}\ (1 + n)\ A)\end{aligned} \tag{87}$$

A similar result can also be proved for the property of being very stable (if []-*cong* is available and equality is extensional for functions):

$$\begin{aligned}&\forall\ n \to \mathit{For\text{-}iterated\text{-}equality}\ n\ \ \mathit{Very\text{-}stable}\ A \to\\ &\mathit{Very\text{-}stable}\ (\mathit{H\text{-}level}\ n\ A)\end{aligned} \tag{88}$$

Note that this result starts "one level lower":

$$\mathit{Very\text{-}stable}\ A \to \mathit{Very\text{-}stable}\ (\mathit{Contractible}\ A) \tag{89}$$

This latter result can be proved using the closure properties for Σ (82) and Π (80), and the fact that equality is very stable for very stable types (75) (again assuming that []-*cong* is available and that equality is extensional for functions). See also Rijke et al. (2019, Remark 1.29).

Let us now consider data types with several constructors, like the booleans. The booleans are stable, because they are inhabited. However, resurrecting an identical copy of an erased boolean, without any knowledge of which boolean it was, seems hard, suggesting that it might not be possible to prove that the booleans are very stable in a non-erased context. (The booleans are provably very stable in an erased context, just like every other type, so it should not be possible to prove that they are *not* very stable in a non-erased context.) If we move one level up we can note that equality is decidable for the booleans, so equality is very stable for them.

What if we take general binary sums for which equality might not be decidable? In this case we can prove the following closure properties (the proof of the second one makes use of []-*cong*):

$$\mathit{Stable\text{-}}\equiv\ A \to \mathit{Stable\text{-}}\equiv\ B \to \mathit{Stable\text{-}}\equiv\ (A \uplus B) \tag{90}$$

$$\mathit{Very\text{-}stable\text{-}}\equiv\ A \to \mathit{Very\text{-}stable\text{-}}\equiv\ B \to \mathit{Very\text{-}stable\text{-}}\equiv\ (A \uplus B) \tag{91}$$

The first result can be proved by case analysis: if the two values of type $A \uplus B$ are both left injections or both right injections, then the assumptions can be

used, and otherwise a contradiction is obtained, because a left injection is not equal to a right injection. The second result can also be proved by case analysis.

This kind of result is not limited to non-recursive data types. It can also be proved for lists (again the proof of the second one makes use of []-*cong*):

$$\begin{aligned} &\mathit{Stable}\text{-}\equiv && A \to \mathit{Stable}\text{-}\equiv && (\mathit{List}\ A) && (92)\\ &\mathit{Very\text{-}stable}\text{-}\equiv && A \to \mathit{Very\text{-}stable}\text{-}\equiv && (\mathit{List}\ A) && (93) \end{aligned}$$

The proofs use recursion on the structure of lists. As another example this kind of result can be proved for set quotients, defined using a higher inductive type (in Cubical Agda, roughly following The Univalent Foundations Program (2013)):

$$\begin{aligned} &\mathit{Is\text{-}equivalence\text{-}relation}\ R \to (\forall\ x\ y \to \mathit{H\text{-}level}\ 1\ (R\ x\ y)) \to\\ &(\forall\ x\ y \to \mathit{Stable}\ (R\ x\ y)) \to \mathit{Very\text{-}stable}\text{-}\equiv\ (A\ /\ R) \end{aligned} \tag{94}$$

The proof uses the fact that if R is a propositional equivalence relation, then $R\ x\ y$ is equivalent to $[\ x\ ] \equiv [\ y\ ]$, where $[\_]$ is the (overloaded) canonical surjection for the quotient $A\ /\ R$.

4.6 []-*cong* can be proved using extensionality

If extensionality holds for functions, then one can prove []-*cong*, show that it is an equivalence, and prove its computation rule. All the main results in this section make use of function extensionality for a fixed universe level a.

First note that one can prove Lemma 59 without making use of []-*cong* (this proof is based on the proof of Lemma 1.20 due to Rijke et al. (2019)):

$$\{A : \mathit{Set}\ a\}\ (s : \mathit{Stable}\ A) \to (\forall\ x \to s\ [\ x\ ] \equiv x) \to \mathit{Very\text{-}stable}\ A \tag{95}$$

The following calculation shows that if $s : \mathit{Stable}\ A$ is a left inverse of $[\_]$, then it is also a right inverse, which implies that A is very stable:

$$\begin{aligned} &(\forall\ x \to s\ [\ x\ ] \equiv x) && \to\\ &(\forall\ x \to [\ s\ [\ x\ ]\ ] \equiv [\ x\ ]) && \to\\ &[\_] \circ s \circ [\_] \equiv [\_] && \to\\ &[\_] \circ s \equiv \mathit{id} && \to\\ &(\forall\ x \to [\ s\ x\ ] \equiv x) \end{aligned}$$

The first step uses the fact that any non-dependent function, including $[\_]$, preserves equality. The second step uses extensionality, and the final step uses the inverse of extensionality. The third step uses the fact that *Erased* is a uniquely eliminating modality (24), along with the following fact:

$$(A{\simeq}B : A \simeq B) \to (\_{\simeq}\_.\mathsf{to}\ A{\simeq}B\ x \equiv \_{\simeq}\_.\mathsf{to}\ A{\simeq}B\ y) \simeq (x \equiv y) \tag{96}$$

One can also prove Lemma 75 without using []-*cong* (this proof is based on the proof of Lemma 1.25 due to Rijke et al. (2019)):

$$\{A : \mathit{Set}\ a\} \to \mathit{Very\text{-}stable}\ A \to \mathit{Very\text{-}stable}\text{-}\equiv\ A \tag{97}$$

If A is very stable, then $[\_]$ has an inverse $s : \mathit{Stable}\ A$. Using this inverse we can prove that equality is stable for A. Given $x : A$, $y : A$ and $eq : \mathit{Erased}\ (x \equiv y)$ we can calculate in the following way:

$$
\begin{array}{ll}
x & \equiv \\
s\ [\ x\ ] & \equiv \\
s\ [\ y\ ] & \equiv \\
y &
\end{array}
$$

The first and last steps use the assumption that s is an inverse of $[\_]$. For the second step we can start by noticing that extensionality implies that the function $\lambda\ (\_ : x \equiv y) \to [\ x\ ]$ is equal to $\lambda\ (\_ : x \equiv y) \to [\ y\ ]$. The fact that *Erased* is a uniquely eliminating modality (24) implies that

$$\lambda\ (f : \mathit{Erased}\ (x \equiv y) \to \mathit{Erased}\ A) \to f \circ [\_]$$

is injective, so we get that the function $\lambda\ (\_ : \mathit{Erased}\ (x \equiv y)) \to [\ x\ ]$ is equal to $\lambda\ (\_ : \mathit{Erased}\ (x \equiv y)) \to [\ y\ ]$. If we apply the function

$$\lambda\ (f : \mathit{Erased}\ (x \equiv y) \to \mathit{Erased}\ A) \to s\ (f\ \mathit{eq})$$

to these functions we get $s\ [\ x\ ]$ and $s\ [\ y\ ]$. If we can prove that this stability proof is a left inverse of $[\_]$ (for every x and y), then Lemma 95 implies that equality is very stable for A. For details of such a proof, see the accompanying code.

Let us now prove []-*cong*:

$$\{@0\ A : \mathit{Set}\ a\}\ \{@0\ x\ y : A\} \to \mathit{Erased}\ (x \equiv y) \to [\ x\ ] \equiv [\ y\ ] \tag{98}$$

This will be done in two steps. First a preliminary variant of []-*cong* will be given, and then this variant will be used to define a variant that is an equivalence. For the first variant we can calculate in the following way:

$$
\begin{array}{ll}
\mathit{Erased}\ (x \equiv y) & \to \\
\mathit{Erased}\ ([\ x\ ] \equiv [\ y\ ]) & \to \\
[\ x\ ] \equiv [\ y\ ] &
\end{array}
$$

The first step uses the *map* function (26), and the second step uses the fact that erased types are very stable (61) and the previous lemma (97). Using this preliminary definition of []-*cong* we can easily prove the following preservation lemma for equivalences:

$$\{@0\ A\ B : \mathit{Set}\ a\} \to @0\ A \simeq B \to \mathit{Erased}\ A \simeq \mathit{Erased}\ B$$

(Note that, unlike Lemma 39, this preservation lemma is defined for a fixed universe level.) This preservation lemma can then be used to define the second variant of []-*cong*, which is an equivalence, using the following calculation:

$$
\begin{array}{ll}
\mathit{Erased}\ (x \equiv y) & \simeq \\
\mathit{Erased}\ ([\ x\ ] \equiv [\ y\ ]) & \simeq \\
[\ x\ ] \equiv [\ y\ ] &
\end{array}
$$

One can also prove that this definition satisfies the computation rule for []-*cong*. The proof is similar to the final, omitted step of the proof of Lemma 97, see the accompanying code for details.

Rijke et al. prove that for a lex modality a certain function with the same type as []-*cong* (except for the @0 annotations) is an equivalence (2019, Theorem 3.1 (ix)).

5 Examples

This section contains some examples showing how *Erased* and the theory developed above can be used.

5.1 Singleton types with erased equality proofs

First let us discuss singleton types. Singleton types of the form $\Sigma\ A\ (\lambda\ y \to y \equiv x)$ are contractible. What if the equality proof is erased? In that case the following result can be proved:

$$\mathit{Very\text{-}stable\text{-}}{\equiv}\ A \to \mathit{Contractible}\ (\Sigma\ A\ (\lambda\ y \to \mathit{Erased}\ (y \equiv x))) \tag{99}$$

If equality is very stable for A, then $\mathit{Erased}\ (y \equiv x)$ is equivalent to $y \equiv x$, so the type is equivalent to a regular singleton type. If x, which witnesses the inhabitance of A, is erased, then one can still prove that the type is a proposition:

$$\begin{aligned}&\{@0\ x : A\} \to\\ &\mathit{Very\text{-}stable\text{-}}{\equiv}\ A \to \mathit{H\text{-}level}\ 1\ (\Sigma\ A\ (\lambda\ y \to \mathit{Erased}\ (y \equiv x)))\end{aligned} \tag{100}$$

In an erased context $\Sigma\ A\ (\lambda\ y \to \mathit{Erased}\ (y \equiv x))$ is contractible, and thus also a proposition. We can conclude if $\mathit{H\text{-}level}\ 1\ (\Sigma\ A\ (\lambda\ y \to \mathit{Erased}\ (y \equiv x)))$ is stable. This type is stable if equality is stable for $\Sigma\ A\ (\lambda\ y \to \mathit{Erased}\ (y \equiv x))$ (86). Given the assumption that equality is very stable for A we in fact get that equality is very stable for this type: this follows from the fact that *Very-stable-*$\equiv$ is closed under Σ (this is an instance of a generalisation of Lemma 82), the fact that *Erased* B is always very stable (61), and the fact that if C is very stable, then equality is very stable for C (75).

We can also prove the following lemma:

$$\Sigma\ (\mathit{Erased}\ A)\ (\lambda\ x \to \Sigma\ A\ (\lambda\ y \to \mathit{Erased}\ (y \equiv \mathsf{erased}\ x))) \simeq A \tag{101}$$

The proof is a simple calculation:

$$\begin{aligned}&\Sigma\ (\mathit{Erased}\ A)\ (\lambda\ x \to \Sigma\ A\ (\lambda\ y \to \mathit{Erased}\ (y \equiv \mathsf{erased}\ x))) && \simeq\\ &\Sigma\ A\ (\lambda\ y \to \Sigma\ (\mathit{Erased}\ A)\ (\lambda\ x \to \mathit{Erased}\ (y \equiv \mathsf{erased}\ x))) && \simeq\\ &\Sigma\ A\ (\lambda\ y \to \mathit{Erased}\ (\Sigma\ A\ (\lambda\ x \to y \equiv x))) && \simeq\\ &A \times \mathit{Erased}\ \top && \simeq\\ &A\end{aligned}$$

The second and fourth steps use commutation properties from Section 3.3, and the third step uses a preservation lemma (48) as well as the fact that singleton types are equivalent to the unit type.

5.2 Efficient natural numbers

Natural numbers can be represented in several ways. One common representation is the inductive data type with two constructors, $\mathsf{zero} : \mathbb{N}$ and $\mathsf{suc} : \mathbb{N} \to \mathbb{N}$. This representation is perhaps more suited for proofs than for computation. If one cares about performance—or at least asymptotic complexity—then some other representation (perhaps a binary one) might be more suitable.

In this section I will outline how one can have a natural number type that—for some operations—computes roughly like the unary representation at compile-time (when code is type-checked), but gets compiled to something possibly more efficient.

The following development works for any type of (possibly) efficient natural numbers $Nat' : Set$ which is equivalent to the unary natural numbers, $Nat' \simeq \mathbb{N}$. Let us denote the forward direction of this equivalence by *to*-ℕ, and the other direction by *from*-ℕ. The efficient natural number n is assumed to stand for the unary natural number *to*-ℕ n. As an example the accompanying code contains an implementation of natural numbers as lists of bits with the least significant bit first and an erased invariant that ensures that there are no trailing zeros. (No claim is made that this representation is efficient compared to something based on machine integers.)

Let us start by defining a type of efficient natural numbers representing a specific natural number n:

$$\begin{aligned} &Nat\text{-}[\_] : @0\ \mathbb{N} \to Set \\ &Nat\text{-}[\ n\ ] = \Sigma\ Nat'\ (\lambda\ m \to Erased\ (to\text{-}\mathbb{N}\ m \equiv n)) \end{aligned} \tag{102}$$

The equality proof is erased, to make sure that it is not present at run-time. Note that it might be better to switch to a variant of the Σ-type where the second field is erased: a compiler might generate actual pairs for the type above, rather than just truncated efficient natural numbers. However, such a change would presumably not affect the asymptotic complexity of the code.

The type *Nat*-[n] is a proposition:

$$\{@0\ n : \mathbb{N}\} \to H\text{-}level\ 1\ (Nat\text{-}[\ n\ ]) \tag{103}$$

Equality is very stable for the unary natural numbers (this follows from Lemma 67, because equality is decidable for this type). Thus equality is very stable also for Nat' (this follows from Lemma 72), so by Lemma 100 the type

$$\Sigma\ Nat'\ (\lambda\ m \to Erased\ (m \equiv from\text{-}\mathbb{N}\ n))$$

is a proposition. Furthermore this type is equivalent to *Nat*-[n], and *H-level* 1 respects equivalences.

Given the definition above one can now define another type of (possibly efficient) natural numbers:

$$\begin{aligned} &Nat : Set \\ &Nat = \Sigma\ (Erased\ \mathbb{N})\ (\lambda\ n \to Nat\text{-}[\ \mathsf{erased}\ n\ ]) \end{aligned} \tag{104}$$

A value of type *Nat* is an erased unary natural number index n, and an underlying natural number that represents n.

Let the (erased) function $\lfloor\_\rfloor$ return the erased index:

$$\begin{aligned} &@0\ \lfloor\_\rfloor : Nat \to \mathbb{N} \\ &\lfloor\ ([\ n\ ]\ ,\ \_)\ \rfloor = n \end{aligned} \tag{105}$$

Assume that we have two values m, $n : Nat$. Because the second projections are propositional (103) there is an equivalence between $m \equiv n$ and equality of the first projections, $\mathsf{proj}_1\ m \equiv \mathsf{proj}_1\ n$. By combining this observation with the equivalence []-*cong* we get the following lemma:

$$\{m\ n : \mathit{Nat}\} \to \mathit{Erased}\ (\lfloor\ m\ \rfloor \equiv \lfloor\ n\ \rfloor) \simeq (m \equiv n) \tag{106}$$

One can prove that two values of type *Nat* are equal by constructing an erased proof showing that the erased indices are equal.

5.2.1 Arithmetic

Let us now see how one can implement arithmetic operations for *Nat*-[_] and *Nat*. I will focus on unary operations; other arities can be treated in a similar way. Given a function f' on *Nat*′ that corresponds in a certain sense to a function f on $\mathbb{N}$ one can construct a function from *Nat*-[n] to *Nat*-[$f\ n$]:

$$\begin{aligned}
&\mathit{unary}\text{-}[] :\\
&\quad \{@0\ n : \mathbb{N}\}\ \{@0\ f : \mathbb{N} \to \mathbb{N}\}\ (f' : \mathit{Nat}' \to \mathit{Nat}') \to\\
&\quad @0\ (\forall\ n \to \mathit{to}\text{-}\mathbb{N}\ (f'\ n) \equiv f\ (\mathit{to}\text{-}\mathbb{N}\ n)) \to\\
&\quad \mathit{Nat}\text{-}[\ n\ ] \to \mathit{Nat}\text{-}[\ f\ n\ ]
\end{aligned} \tag{107}$$

Note that n, f and the proof are erased. The input of type *Nat*-[n] gives us an efficient natural number n' and an erased proof of type $\mathit{to}\text{-}\mathbb{N}\ n' \equiv n$. We can return a pair containing $f'\ n'$ and an erased proof of the equality $\mathit{to}\text{-}\mathbb{N}\ (f'\ n') \equiv f\ n$.

We can also construct a function on *Nat*:

$$\begin{aligned}
&\mathit{unary} :\\
&\quad (@0\ f : \mathbb{N} \to \mathbb{N})\ (f' : \mathit{Nat}' \to \mathit{Nat}') \to\\
&\quad @0\ (\forall\ n \to \mathit{to}\text{-}\mathbb{N}\ (f'\ n) \equiv f\ (\mathit{to}\text{-}\mathbb{N}\ n)) \to\\
&\quad \mathit{Nat} \to \mathit{Nat}\\
&\mathit{unary}\ f\ f'\ \mathit{eq}\ ([\ n\ ]\ ,\ n') = ([\ f\ n\ ]\ ,\ \mathit{unary}\text{-}[]\ f'\ \mathit{eq}\ n')
\end{aligned} \tag{108}$$

Note that the index computes like f. Because equality for *Nat* is equivalent to (erased) equality of the erased indices this means that one can ignore the second components at compile-time. However, at run-time the first components are erased, and the computational behaviour should be, roughly speaking, that of the underlying, possibly efficient natural numbers. (There is one caveat to the previous statement: as mentioned in Section 2 there is currently no compiler for Cubical Agda, and it is not known if there is a reasonable way to handle erasure in this setting. However, there is at least one compiler for traditional Agda that respects erasure annotations.)

5.2.2 Converting *Nat* to $\mathbb{N}$

At this point one may wonder whether natural numbers of type *Nat* can be used for other things than computing new values of the same type. For instance, can these natural numbers be converted to (non-erased) unary natural numbers? It turns out that there is a run-time equivalence between *Nat* and $\mathbb{N}$.

First note that, because *Nat*′ is equivalent to $\mathbb{N}$, we get an equivalence between *Nat*-[n] and a type that does not refer to *Nat*′:

$$\{@0\ n : \mathbb{N}\} \to \mathit{Nat}\text{-}[\ n\ ] \simeq \Sigma\ \mathbb{N}\ (\lambda\ m \to \mathit{Erased}\ (m \equiv n)) \tag{109}$$

We can now construct an equivalence between *Nat* and $\mathbb{N}$:

$$
\begin{array}{ll}
\mathit{Nat} & \simeq \\
\Sigma\ (\mathit{Erased}\ \mathbb{N})\ (\lambda\ n \to \mathit{Nat}\text{-}[\!|\ \mathsf{erased}\ n\ |\!]) & \simeq \\
\Sigma\ (\mathit{Erased}\ \mathbb{N})\ (\lambda\ n \to \Sigma\ \mathbb{N}\ (\lambda\ m \to \mathit{Erased}\ (m \equiv \mathsf{erased}\ n))) & \simeq \\
\mathbb{N} &
\end{array}
\tag{110}
$$

The first step holds by definition, the second step uses the previous equivalence (note that it is important that this equivalence is proved for an erased argument n), and the last step is an instance of Lemma 101.

Let us denote the forward direction of the equivalence by $\mathit{Nat}{\to}\mathbb{N}$. This function does not simply return the index (which is erased), it computes a run-time natural number based on non-erased information. However, in an erased context we can prove that this function returns something that is equal to the index:

$$
@0\ \mathit{Nat}{\to}\mathbb{N}\text{-}\mathit{returns\text{-}index} : \forall\ n \to \mathit{Nat}{\to}\mathbb{N}\ n \equiv \lfloor\ n\ \rfloor \tag{111}
$$

Note that n must have the form $([\!|\ m\ |\!]\ ,\ m'\ ,\ \mathit{eq})$ for some index m, number $m' : \mathit{Nat}'$ and proof $\mathit{eq} : \mathit{Erased}\ (\mathit{to}\text{-}\mathbb{N}\ m' \equiv m)$. The application of $\mathit{Nat}{\to}\mathbb{N}$ to such a tuple is definitionally equal to $\mathit{to}\text{-}\mathbb{N}\ m'$. (This text does not include every detail of every function. The definitional equality holds if the functions are implemented as in the accompanying code.) Furthermore $\lfloor\ n\ \rfloor$ is definitionally equal to m, so the proof can be completed by using $\mathsf{erased}\ \mathit{eq}$.

We can now show that *unary* (108) is well-behaved in the following sense:

$$
\begin{array}{l}
\{@0\ \mathit{eq} : \forall\ n \to \mathit{to}\text{-}\mathbb{N}\ (f'\ n) \equiv f\ (\mathit{to}\text{-}\mathbb{N}\ n)\} \to \\
\forall\ n \to \mathit{Nat}{\to}\mathbb{N}\ (\mathit{unary}\ f\ f'\ \mathit{eq}\ n) \equiv f\ (\mathit{Nat}{\to}\mathbb{N}\ n)
\end{array}
\tag{112}
$$

We can calculate in the following way:

$$
\begin{array}{ll}
\mathit{Nat}{\to}\mathbb{N}\ (\mathit{unary}\ f\ f'\ \mathit{eq}\ n) & \equiv \\
\lfloor\ \mathit{unary}\ f\ f'\ \mathit{eq}\ n\ \rfloor & \equiv \\
f\ \lfloor\ n\ \rfloor & \equiv \\
f\ (\mathit{Nat}{\to}\mathbb{N}\ n) &
\end{array}
$$

The second step holds by definition, and the first and third steps follow from Lemma 111. However, note that $\lfloor\_\rfloor$ and Lemma 111 can only be used in erased contexts. The proof can be completed by making use of the fact that equality is stable for the natural numbers.

5.2.3 Decidable equality

As mentioned in Section 5.2.1 the function $\mathit{unary}\ f\ f'\ \mathit{eq}$ computes roughly like f at compile-time: when the function is applied to a pair $([\!|\ n\ |\!]\ ,\ n')$ the first projection of the result is $[\!|\ f\ n\ |\!]$. However, there are functions that do not have *Nat* as their codomain. The forward direction of the equivalence between *Nat* and $\mathbb{N}$ (110) provides one example, and in this section an equality test of the following type is constructed:

$$
(m\ n : \mathit{Nat}) \to \mathit{Dec}\ (\mathit{Erased}\ (\lfloor\ m\ \rfloor \equiv \lfloor\ n\ \rfloor)) \tag{113}
$$

Recall that $\mathit{Dec}\ A$ is the binary sum $A \uplus \neg\ A$: the function either returns an erased proof showing that the two numbers' indices are equal, or a refutation

of such proofs. If the erasure mechanism works properly, then the choice of whether to return the left injection or the right injection must be taken based on information that is present at run-time, which means that the function cannot return any useful information based only on the erased indices. Note that this argument applies to any function with a type of the form $(x : A) \to P\ x \uplus Q\ x$ where the result can be both a left injection and a right injection.

So, how can the equality test be defined? Let us assume that there is a (possibly efficient) equality test for *Nat′*:

$$\forall\ m\ n \to \mathit{Dec}\ (\mathit{Erased}\ (\mathit{to}\text{-}\mathbb{N}\ m \equiv \mathit{to}\text{-}\mathbb{N}\ n))$$

Using this equality test we can implement an equality test for *Nat*-[_]:

$$\{@0\ m\ n : \mathbb{N}\} \to \mathit{Nat}\text{-}[\ m\ ] \to \mathit{Nat}\text{-}[\ n\ ] \to \mathit{Dec}\ (\mathit{Erased}\ (m \equiv n)) \quad (114)$$

We can take apart the two *Nat*-[_] arguments and apply the equality test for *Nat′* to the underlying efficient natural numbers. The erased equalities from the *Nat*-[_] arguments can then be used to construct an answer of the right type. Finally it is very easy to construct the equality test for *Nat* (113): it suffices to apply the equality test for *Nat*-[_] to the second projections of the two arguments of type *Nat*.

5.3 Queues

The technique discussed above is not restricted to natural numbers. The accompanying code contains a similar construction for queues, using lists as the erased indices. Here are the type signatures of the enqueue and dequeue operations:

$$\mathit{enqueue} : A \to \mathit{Queue}\ A \to \mathit{Queue}\ A \quad (115)$$
$$\mathit{dequeue} : \mathit{Very\text{-}stable}\text{-}\equiv\ A \to \mathit{Queue}\ A \to \mathit{Maybe}\ (A \times \mathit{Queue}\ A) \quad (116)$$

(*Maybe A* is the binary sum $\top \uplus A$.) Note that the dequeue operation requires that equality is very stable for the carrier type: this assumption is used to prove that certain types are propositions, using Lemma 100. The enqueue operation computes similarly to the function *unary* (108), but the dequeue operation computes more like the equality test for *Nat* (113): note that the dequeue operation's return type is a binary sum, just like the equality test's.

6 Discussion and related work

I have presented some theory of *Erased*, along with some examples of how this modality can be used.

Erased is defined in Agda, using the @0 annotation. I would like to know whether something similar can be defined, along with the theory, in related systems. For instance, it should be possible to define something like *Erased* in Coq, using the *Prop* universe (which is used to control erasure (Letouzey 2003)), but I do not know if the theory can be developed in that setting.

Note that *Erased* is not the same thing as the squash operator (Constable et al. 1986; Mendler 1990), or the bracket operator (Awodey and Bauer 2004), or the propositional truncation operator (The Univalent Foundations Program 2013), because *Erased A* is not necessarily a proposition (23). Thus these type

formers are not replacements for *Erased*, and *Erased* is not a replacement for them.

Perhaps it is instructive to see what happens if *Erased* $\mathbb{N}$ is replaced by the propositional truncation $\| \mathbb{N} \|$ in the type *Nat* (104). First note that the type Σ *Set Contractible* is a proposition (assuming univalence). Furthermore *Nat*-[n] is contractible (because it is an inhabited proposition), so it can be the first projection of a pair of type Σ *Set Contractible*. Thus we can implement a function from truncated natural numbers that constructs this kind of pair using the recursion principle for the propositional truncation (which gives access to a natural number n), and then returns the first projection of the pair. If we call this function *Nat*-[_]′, then we get the following alternative implementation of *Nat*: $\Sigma \| \mathbb{N} \|$ *Nat*-[_]′. However, this implementation has a problem: it is equivalent to the unit type, and thus not equivalent to $\mathbb{N}$ (see the accompanying code for details).

I am aware of two previous works discussing something like *Erased* in some detail: the general theory of modalities developed by Rijke et al. (2019), which has been mentioned frequently above, and the work of Mishra-Linger (2008). Mishra-Linger defines a type function that he calls squash or $\bigcirc$. He discusses this function in two settings, EPTS and EPTS$^\bullet$. EPTS has support for erasure, roughly like the @0 annotation of Agda. EPTS$^\bullet$ additionally incorporates erasure into definitional equality, so that terms are compared after erasure. In the setting of EPTS$^\bullet$ Mishra-Linger's squash type always produces propositions. However, in the setting of EPTS it seems to be closer to *Erased*. Mishra-Linger notes that in this setting he can prove that $\bigcirc$ is an applicative functor (McBride and Paterson 2008), and he notes that he can also prove that it is a monad and that $\bigcirc \bigcirc A$ is isomorphic to $\bigcirc A$ if "token type target erasure"—basically support for pattern-matching for erased arguments of single-constructor data types—is allowed for $\bigcirc$ (but he argues against allowing this). He also proves that $\bigcirc A \to B$ is isomorphic to $@0\ A \to B$ (his notation is different), and states that $(x : \bigcirc A) \to B$ is not equivalent to $(@0\ x : A) \to B$, where B can mention x. It is unclear to me what this means, because x does not have the same type in the two expressions. Mishra-Linger views this difference between $\bigcirc$ and @0 "as a principal advantage of [his] approach over squash types". As noted above (7) something like this can be proved for *Erased* in Agda:

$$\{@0\ A : \mathit{Set}\ a\}\ \{@0\ P : \mathit{Erased}\ A \to \mathit{Set}\ p\} \to \\ ((x : \mathit{Erased}\ A) \to P\ x) \simeq ((@0\ x : A) \to P\ [\ x\ ])$$

I suspect that the projection erased is not supported in Mishra-Linger's setting, and this equivalence seems less interesting then.

Acknowledgements

I would like to thank Andreas Abel for adding support for @0 to Agda, and Andreas Abel, Jesper Cockx, Thierry Coquand, Ulf Norell, Mike Shulman and Andrea Vezzosi for useful discussions.

Financial support

This research received no specific grant from any funding agency, commercial or not-for-profit sectors.

Conflicts of interest

No known conflicts of interest.

References

Atkey, R. (2018). "Syntax and Semantics of Quantitative Type Theory". In *LICS '18 Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, edited by A. Dawar and E. Grädel. ACM New York, NY, USA. Pages 56–65. doi:10.1145/3209108.3209189.

Augustsson, L. (1998). "Cayenne — a language with dependent types". In *ICFP '98, Proceedings of the third ACM SIGPLAN international conference on Functional programming*, edited by M. Felleisen, P. Hudak, and C. Queinnec. ACM New York, NY, USA. Pages 239–250. doi:10.1145/289423.289451.

Awodey, S. and Bauer, A. (2004). "Propositions as [Types]". In *J. Logic Comput.* 14(4). Pages 447–471. doi:10.1093/logcom/14.4.447.

Barras, B. and Bernardo, B. (2008). "The Implicit Calculus of Constructions as a Programming Language with Dependent Types". In *Foundations of Software Science and Computational Structures, 11th International Conference, FOSSACS 2008*, edited by R. Amadio. LNCS 4962. Springer-Verlag Berlin Heidelberg. Pages 365–379. doi:10.1007/978-3-540-78499-9_26.

Bernardy, J.-P. and Moulin, G. (2013). "Type-Theory In Color". In *ICFP'13, Proceedings of the 2013 ACM SIGPLAN International Conference on Functional Programming*, edited by T. Uustalu. The Association for Computing Machinery, 2 Penn Plaza, Suite 701, New York, New York 10121-0701. Pages 61–71. doi:10.1145/2500365.2500577.

Brady, E. C. (2005). "Practical Implementation of a Dependently Typed Functional Programming Language". PhD thesis. University of Durham. URL: `http://eb.host.cs.st-andrews.ac.uk/writings/thesis.pdf` (visited on 2019-11-07).

Brady, E., McBride, C., and McKinna, J. (2004). "Inductive Families Need Not Store Their Indices". In *TYPES 2003: Types for Proofs and Programs*, edited by S. Berardi, M. Coppo, and F. Damiani. LNCS 3085. Springer-Verlag Berlin Heidelberg New York. Pages 115–129. doi:10.1007/978-3-540-24849-1_8.

Cohen, C., Coquand, T., Huber, S., and Mörtberg, A. (2018). "Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom". In *21st International Conference on Types for Proofs and Programs, TYPES 2015*, edited by T. Uustalu. LIPIcs 69. Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. 5:1–5:34. doi:10.4230/LIPIcs.TYPES.2015.5.

Constable, R. L., Allen, S. F., Bromley, H. M., Cleaveland, W. R., Cremer, J. F., Harper, R. W., Howe, D. J., Knoblock, T. B., Mendler, N. P., Panangaden, P., Sasaki, J. T., and Smith, S. F. (1986). *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Inc., A Division of Simon & Schuster, Englewood Cliffs, New Jersey 07632. URL: `http://nuprl.org/book/` (visited on 2019-11-07).

Coquand, T. and Huet, G. (1988). "The Calculus of Constructions". In *Inform. Comput.* 76 (2–3). Pages 95–120. doi:10.1016/0890-5401(88)90005-3.

Fernandez, M., Mackie, I., Severi, P., and Szasz, N. (2003). "Reduction Strategies for Program Extraction". In *CLEI Electronic Journal* 6(1). doi:10.19153/cleiej.6.1.2.

Fredriksson, O. and Gustafsson, D. (2011). "A totally Epic backend for Agda". Master's thesis. Chalmers University of Technology and University of Gothenburg. HDL: `20.500.12380/146807`.

Gundry, A. M. (2013). "Type Inference, Haskell and Dependent Types". PhD thesis. University of Strathclyde. URL: `http://adam.gundry.co.uk/pub/thesis/` (visited on 2019-11-07).

Gundry, A. and McBride, C. (2013). "Phase Your Erasure". URL: `https://personal.cis.strath.ac.uk/conor.mcbride/pub/phtt.pdf` (visited on 2019-11-07).

Hayashi, S. and Nakano, H. (1988). *PX: A Computational Logic*. The MIT Press, Cambridge, Massachusetts, London, England.

Hedberg, M. (1998). "A coherence theorem for Martin-Löf's type theory". In *J. Funct. Program.* 8(4). Pages 413–436. doi:10.1017/S0956796898003153.

Letouzey, P. (2003). "A New Extraction for Coq". In *Types for Proofs and Programs, International Workshop, TYPES 2002*, edited by H. Geuvers and F. Wiedijk. LNCS 2646. Springer-Verlag Berlin Heidelberg New York. Pages 200–219. doi:10.1007/3-540-39185-1_12.

McBride, C. (2016). "I Got Plenty o' Nuttin'". In *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, edited by S. Lindley, C. McBride, P. Trinder, and D. Sannella. LNCS 9600. Springer International Publishing Switzerland. Pages 207–233. doi:10.1007/978-3-319-30936-1_12.

McBride, C. and Paterson, R. (2008). "Applicative programming with effects". In *J. Funct. Program.* 18(1). Pages 1–13. doi:10.1017/S0956796807006326.

Mendler, N. P. (1990). "Quotient types via coequalizers in Martin-Löf type theory". In *Proceedings of the First Workshop on Logical Frameworks*, edited by G. Huet and G. Plotkin. URL: `http://www.cse.chalmers.se/research/group/logic/Types/proc90.ps` (visited on 2019-11-07).

Mishra-Linger, N. and Sheard, T. (2008). "Erasure and Polymorphism in Pure Type Systems". In *Foundations of Software Science and Computational Structures, 11th International Conference, FOSSACS 2008*, edited by R. Amadio. LNCS 4962. Springer-Verlag Berlin Heidelberg. Pages 350–364. doi:10.1007/978-3-540-78499-9_25.

Mishra-Linger, R. N. (2008). "Irrelevance, Polymorphism, and Erasure in Type Theory". PhD thesis. Portland State University. doi:10.15760/etd.2669.

Paulin-Mohring, C. (1989). "Extracting F_w's Programs from Proofs in the Calculus of Constructions". In *POPL '89 Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM New York, NY, USA. Pages 89–104. doi:10.1145/75277.75285.

Paulin-Mohring, C. and Werner, B. (1993). "Synthesis of ML programs in the system Coq". In *J. Symb. Comput.* 15 (5–6). Pages 607–640. doi:10.1016/S0747-7171(06)80007-6.

Raamsdonk, F. van and Severi, P. (2002). "Eliminating Proofs from Programs". In *Electronic Notes in Theoretical Computer Science* 70(2). Pages 42–59. doi:10.1016/S1571-0661(04)80505-X.

Rijke, E., Shulman, M., and Spitters, B. (2019). *Modalities in homotopy type theory*. arXiv: `1706.07526v5 [math.CT]`.

Swan, A. (2016). "An Algebraic Weak Factorisation System on 01-Substitution Sets: A Constructive Proof". In *Journal of Logic & Analysis* 8(1). Pages 1–35. doi:10.4115/jla.2016.8.1.

Tejiščák, M. and Brady, E. (2015). "Practical Erasure in Dependently Typed Languages". Draft. URL: `http://eb.host.cs.st-andrews.ac.uk/drafts/dtp-erasure-draft.pdf` (visited on 2019-11-07).

The Univalent Foundations Program (2013). *Homotopy Type Theory: Univalent Foundations of Mathematics*. First edition. URL: `https://homotopytypetheory.org/book/` (visited on 2019-11-07).

Weirich, S., Voizard, A., Amorim, P. H. A. de, and Eisenberg, R. (2017). "A Specification for Dependent Types in Haskell". In *Proceedings of the ACM on Programming Languages* 1 (ICFP). 31:1–31:30. doi:10.1145/3110275.