Nameless, Painless

Nicolas Pouillard
INRIA
Nicolas.Pouillard@inria.fr

Abstract

De Bruijn indices are a well known technique for programming with names and binders. They provide a representation that is both simple and canonical.

However, programming errors tend to be really easy to make. We propose a safer programming interface implemented as a library. Whereas indexing the types of names and terms by a numerical bound is a famous technique, we index them by worlds, a different notion of index that is both finer and more abstract. While being more finely typed, our approach incurs no loss of expressiveness or efficiency.

Via parametricity we obtain properties about functions polymorphic on worlds. For instance, well-typed world-polymorphic functions over open λ-terms commute with any renaming of the free variables.

Our whole development is conducted within Agda, from the code of the library, to its soundness proof and the properties of external functions. The soundness of our library is demonstrated via the construction of a logical relations argument.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Data types and structures; Polymorphism

General Terms Design, Languages, Theory

Keywords names, binders, meta-programming, name abstraction, de Bruijn indices

1. Introduction

It is quite common in the programming realm to deal with the mundane business of data structures with names and binders. Compilers, code generators, static analysers, theorem provers, and type-checkers have this in common. They manipulate programs, formulae, proofs, and types. When seen as pieces of data, there is a common difficulty: the representation of variables (names and binders).

One traditional approach is to represent all the occurrences of a bound variable identically by using character strings or integers. While being the most obvious representation it is known to cause a lot of trouble when dealing with operations like substitution. In particular we name it the capture-avoiding substitution because it has to rename some variables to avoid accidental changes called captures.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICFP'11, September 19–21, 2011, Tokyo, Japan.

Copyright © 2011 ACM 978-1-4503-0865-6/11/09…$10.00

In this article we focus on a different kind of representation, namely de Bruijn indices [6]. This representation is said to be nameless because variables are no longer identified by a name but a notion of "distance" to the binding point. De Bruijn indices are the most famous nameless representation and in the following we will use the term "nameless" as a synonym for "de Bruijn indices".

The nameless approach solves part of the problem by providing a canonical representation. More precisely binding occurrences are no longer named (we now use `λ.` instead of `λx.`). Then bound variables are represented by the "distance" to the binding λ. This distance is the number of λs to cross in order to reach the binding λ.

A major issue with this nameless representation is its arithmetic flavor. Indeed properties about names and binders are turned into arithmetic formulae. The result can be harder to understand, and worse, easier to get wrong. To remedy this, several techniques have been proposed to give finer grained types to these representations [1, 2, 5, 7]. We continue further in this direction by providing a safer system to program with a nameless representation.

1.1 Contributions

In a previous paper [10] we developed an abstract interface to program with names and binders. The interface can be both implemented in a named or a nameless style. Here by focusing on a nameless representation we greatly simplify the programming interface needed to achieve our safety goals. This is at the expense of some loss of abstraction. However while the goals are less ambitious, the results are significantly more applicable since simpler and closer to a concrete representation.

Our previous nameless implementation was relying on the `Fin` approach (described in section 1.4.3). The notion of worlds presented here (in section 2.3) is significantly more precise than `Fin`. However the worlds were already abstract to the programmer and they still are. Indeed abstraction is necessary to retain the parametricity properties we want from world-polymorphic functions.

While a fair amount of proofs were already mechanized in [10], the logical relation argument was still largely on paper. Here we not only finished these mechanized proofs but have every part well connected in one single setup. The code and proofs are accessible online at [9] and we encourage the reader to have a look for reference on technical details and applications.

1.2 Outline of the paper

This paper is organized as follows. In the remainder of this section we present several nameless techniques to represent data structures with bindings. In section 2 we detail our approach to the problem. In section 3 we give some examples and advanced operations. In section 4 we develop a logical relation construction which ensures the soundness properties we want, and we show how to exploit these properties on concrete examples. All the construction is built in the AGDA language, our system is an AGDA library and its soundness proof is done in AGDA as well.

1.3 A brief introduction to AGDA notation

Throughout the paper, our definitions are presented in the syntax of AGDA. In AGDA, `Set` (or $\mathtt{Set}_0$) is the type of small types like `Bool`, `Maybe (List Bool)`, or ℕ. $\mathtt{Set}_1$ is the type of `Set`, `Set → Bool`, or `ℕ → Set`. The function space is written `A → B`, while the dependent function space is written `(x : A) → B` or `∀ (x : A) → B`. An implicit parameter, introduced via `∀{x : A} → B`, can be omitted at a call site if its value can be inferred from the context. There are shortcuts for introducing multiple arguments at once or for omitting a type annotation, as in `∀{A} {i j : A} x → e`. There is no specific sort for propositions in AGDA, everything is in `Set ℓ` for some ℓ. The unit type is a record type with no fields named ⊤, it also represents the `True` proposition. The empty type is an (inductive) data type with no constructors named ⊥, it also represents the `False` proposition. The negation `¬ A` is defined as `A → ⊥`.

The same name can be used for different data constructors. AGDA makes use of type annotations to resolve ambiguities.

As in Haskell, a definition consists of a type signature and a sequence of defining equations, which may involve pattern-matching. The `with` construct extends a pattern-matching-based definition with new columns. An ellipsis `...` is used to elide a redundant equation prefix.

AGDA is strict about whitespace: `x≤y` is an identifier, whereas `x ≤ y` is an application. This allows naming a variable after its type (deprived of any whitespace). We use mixfix declarations, such as `_⊕_`. We use some definitions from AGDA's standard library: natural numbers, booleans, lists, and applicative functors (`pure`, `_⊛_`).

For the sake of conciseness, the code fragments presented in the paper are sometimes not perfectly self-contained. However, a complete AGDA development is available online [9].

1.4 Related work: nameless representations

Various techniques have been discovered to build a nameless representation. We have chosen a few of them which gradually setup the stage.

1.4.1 `bare`: The original approach

We call this one `bare`, because it solely relies on natural numbers. To make things more concrete here is an example of its use when defining the untyped λ-calculus with local bindings (`let`). Our naming convention is as follows: we call `Tm` the type of λ-terms, `V` is the data constructor for variables, `_·_` is for the application, `ƛ` is for the λ-abstraction, and finally `Let` is for local bindings.

```
data Tm : Set where
  V    : (x   : ℕ)  → Tm
  _·_  : (t u : Tm) → Tm
  ƛ    : (t   : Tm) → Tm
  Let  : (t u : Tm) → Tm
```

From the point of view of the binding structure, it is striking that no difference appears between the constructors of the data type. It is completely up to the programmer to manage the scoping difference introduced by `ƛ` and `Let`. This is even more worrying in the `Let` case since we have no clue of difference between the arguments.

Here is an example using this representation to build the λ-term for `λf.λx.f x`.

```
appTm : Tm
appTm = ƛ (ƛ (V 1 · V 0))
```

The main advantages of this approach are its simplicity and its expressiveness. The expressiveness is almost maximal since no restriction is put on the usage of variables. Actually expressiveness is really maximal only in partial languages where all types are inhabited by a looping term. Indeed the absence of restrictions on data can be constraining when receiving data as argument. In a total language all cases have to be covered, even those we consider as impossible. For instance a function accepting only closed terms will have to provide a return value for the variable case anyway. Sometimes the return type is so constrained that the type is actually empty in this case, and so no dummy value can be returned. However, even in a partial language the expressiveness is maximal if we are willing to live in an error monad.

1.4.2 `Maybe`: The nested data type approach

The nested data type approach [2, 3, 5] is a first step towards better properties about the binding structure of terms. Let us start with the definition of `Tm` with this approach:

```
data Tm (A : Set) : Set where
  V    : (x : A)                          → Tm A
  _·_  : (t u : Tm A)                     → Tm A
  ƛ    : (t : Tm (Maybe A))               → Tm A
  Let  : (t : Tm A) (u : Tm (Maybe A))    → Tm A
```

There are three points to look at. The `Tm` type is parameterized by another type called `A`, so we can look at it as a kind of container. Note also that the variable case `V` does not hold a ℕ but an `A`. Last but not least the `ƛ` case holds a term whose parameter is not simply an `A` but a `Maybe A`.

This last point makes the `Tm` type a nested data type, also called a non-regular type. This has the consequence of requiring polymorphic recursion to write recursive functions on such a type.

To understand why this is an adequate representation of λ-terms one has to look a bit more at the meaning of `Maybe`. If types are seen as sets, then `Maybe` takes a set and returns a set with one extra element. So each time we cross a `ƛ`, there is one extra element in the set of allowed variables, exactly capturing the fact we are introducing a variable.

To see the difference with the previous approach, here is the `appTm` λ-term again:

```
appTm : Tm ⊥
appTm = ƛ (ƛ (V (just nothing) · V nothing))
```

Note the use of the empty type ⊥ to state the closedness of the `appTm` term. Stating this kind of properties was impossible to do with the previous approach, without resorting to logical properties on the side.

1.4.3 The `Fin` approach

Another approach already described and used in [1, 7] is to index everything (terms for example) by a bound. This bound is the maximum number of distinct free variables allowed in the value. This rule is enforced in two parts: variables have to be strictly lower than their bound, and the bound is incremented by one when crossing a name abstraction (a λ-abstraction for instance, called `ƛ` here).

The `Fin n` type is used for variables and represents natural numbers strictly lower than `n`. The name `Fin n` comes from the fact that it defines finite sets of size `n`. We call this approach `Fin` for its use of this type. The definition found in AGDA standard library is the following:

```
data Fin : ℕ → Set where
  zero : {n : ℕ} → Fin (suc n)
  suc  : {n : ℕ} (i : Fin n) → Fin (suc n)
```

Given this `Fin` type, one can define the `Tm` data type using this approach:

```
data Tm n : Set where
  V    : (x : Fin n)                  → Tm n
  _·_  : (t u : Tm n)                 → Tm n
  ƛ    : (t : Tm (suc n))             → Tm n
  Let  : (t : Tm n) (u : Tm (suc n))  → Tm n
```

As the previous approach, this representation helps enforcing some wellformedness properties, for instance `Tm 0` is the type of closed λ-terms.

Here is the `appTm` λ-term in this approach:

```
appTm : Tm 0
appTm = ƛ (ƛ (V (suc zero) · V zero))
```

We can easily draw a link with the `Maybe` approach. Indeed, the `Fin (suc n)` type has exactly one more element than the `Fin n` type. Although they are not equivalent for at least two reasons. The `Maybe` approach can accept any type to represent variables. This makes the structure more like a container and this can be particularly helpful to define the substitution as the composition of `mapTm : ∀{A B} → (A → B) → Tm A → Tm B` and `joinTm : ∀{A} → Tm (Tm A) → Tm A` as in [2, 5]. The `Fin` approach has advantages as well, the representation is concrete and simpler since closer to the `bare` approach. However this apparent simplicity comes at a cost, we will see that its concrete representation is one root of the problem.

2. NaPa, a safer nameless representation

2.1 Motivation

While these nameless approaches easily guarantee wellformedness of terms, nothing is said about functions manipulating terms. How are the inputs of a function related to its output? What should a function be able to do given unknown free variables? We claim that free variables should be kept abstract. More precisely a function f operating on terms should commute with a renaming Φ of free variables, namely $f \circ \Phi = \Phi \circ f$. By renaming we mean a permutation of the free variables (more precisely an injective function). In fact this is a bit more flexible, depending on the type of the function f this property will be enforced or not, as we will see in section 4.5.

The goal is to catch some programming errors by having finer types for the manipulated values and for the functions manipulating values. We will show later that this is not a restriction by itself since the types do not force the programmer to be precise everywhere.

One common kind of programming error with these nameless representations is to forget to shift the free variables here and there. While these errors are already caught by the `Maybe` and the `Fin` approaches, in these approaches the programmer is provided with means to workaround the type error. In complex situations making the distinction between a valid coercion and a wrong one can be challenging. With finer types, the mistakes will result in more informative type errors.

2.2 A data definition kit

The three previous approaches are closely related. Just a few details change each time. We can capture all of them with a single abstraction: a triple (`World`, `Name`, `_↑1`) where `World` is the index of types like `Tm`, `Name` is the type for names (indexed by worlds) as used in the `V` constructor, and finally `_↑1` is an operation to shift a world by one as used in the `ƛ` constructor.

```
bareKit    : DataKit
bareKit    = ⟨ ⊤ , λ _ → ℕ , id ⟩

maybeKit   : DataKit
maybeKit   = ⟨ Set , id , Maybe ⟩

finKit     : DataKit
finKit     = ⟨ ℕ , Fin , suc ⟩
```

Figure 1. `DataKit`s for the three previous approaches

In Agda, this triple can be defined with a record type named `DataKit`. This type is made universe polymorphic to accept the nested data types approach.

```
record DataKit {ℓ} : Set (suc ℓ) where
  constructor ⟨_,_,_⟩
  field
    World : Set ℓ
    Name  : World → Set
    _↑1   : World → World
```

The three previous approaches are summed up in figure 1 by defining their `DataKit`. Note that for the `bare` approach the index type is the unit type. In the `Maybe` approach the index type is `Set` and the type for names is directly the index so we use the identity function.

We can now give a single definition of `Tm` generalizing the three previous ones. To this end, we simply take a `DataKit` as argument and use it to define the type `Tm`.

```
module DataTm {ℓ} (kit : DataKit {ℓ}) where
  open DataKit kit
  data Tm α : Set ℓ where
    V    : (x : Name α)               → Tm α
    _·_  : (t u : Tm α)               → Tm α
    ƛ    : (t : Tm (α ↑1))            → Tm α
    Let  : (t : Tm α) (u : Tm (α ↑1)) → Tm α
```

We claim that nothing more is required from the kit. As complex binding structures can be defined in the original nameless representation (`bare`), they can also be defined using this kit. In particular having an empty world or making an induction over a world is not necessary.

Translation between data types using different kits can be done modularly as we do in the Agda development. In any direction (any pair of kits $\mathtt{k}_1$ $\mathtt{k}_2$) one can define a function of type $\forall\ \{\alpha_1\ \alpha_2\} \to$ (`Name` $\mathtt{k}_1\ \alpha_1$ → `Name` $\mathtt{k}_2\ \alpha_2$) → `Tm` $\mathtt{k}_1\ \alpha_1$ → `Tm` $\mathtt{k}_2\ \alpha_2$. This function takes a mapping for free variables and lifts this mapping to terms.

Whereas using these translations is fine in many cases, when going from `bare` to a more precise representation, an issue arises. Since Agda is a total language, the initial mapping must be a total function. The `bare` approach ensuring nothing about free variables, one cannot supply the initial mapping to go to a closed term. One has to keep at least one free variable and then dynamically check for occurrences of the free variable in a second transformation.

We will now build on this `DataKit` abstraction to introduce our own solution.

```
World    : Set
∅        : World
_+1      : World → World
_↑1      : World → World
Name     : World → Set
_⊆_      : World → World → Set

napaKit : DataKit
napaKit = ⟨ World , Name , _↑1 ⟩
```

Figure 2. Core types of NAPA

```
zeroᴺ      : ∀ {α}    → Name (α ↑1)
addᴺ       : ∀ {α} k  → Name α
                      → Name (α +ᵂ k)
subtractᴺ  : ∀ {α} k  → Name (α +ᵂ k)
                      → Name α
cmpᴺ       : ∀ {α} ℓ  → Name (α ↑  ℓ)
                      → Name (∅ ↑  ℓ)
                      ⊎ Name (α +ᵂ ℓ)
_==ᴺ_      : ∀ {α} (x y : Name α) → Bool
coerceᴺ    : ∀ {α β}  → α ⊆ β
                      → Name α  → Name β
¬Name∅     : ¬(Name ∅)

syntax addᴺ      k x = x +ᴺ k
syntax subtractᴺ k x = x ∸ᴺ k
syntax cmpᴺ      ℓ x = x <ᴺ ℓ
```

Figure 3. Core operations on names

2.3 NAPA types

We call our approach NAPA as in **Na**meless, **Pa**inless. At the level of types, NAPA exposes the interface shown in figure 2. This is a superset of the DataKit, featuring an empty world, a _+1 operation on worlds described later on, and a notion of world inclusion.

If we look closely at the notion of worlds in the Fin approach, we see that it gives us too much information. Indeed a function could look at the given world (by pattern-matching or induction) and behave differently for some values of the world.

On the other hand, Fin worlds are too coarse: a single number ℓ is used to represent a set, namely the complete interval $[0..\ell\text{-}1]$.

In NAPA, a world denotes a subset of $\mathbb{N}$ but is kept abstract (no pattern-matching nor induction). Only the empty world $\emptyset$ and two operations are given, namely _+1 and _↑1. Semantically $\alpha + 1$ is defined by $\{x + 1 \mid x \in \alpha\}$ and $\alpha \uparrow 1$ by $\{0\} \cup (\alpha + 1)$.

Internally we represent worlds by lists of Boolean values. Having true at the nth position of the list means that n is in the world. Here are the AGDA definitions:

```
World : Set
World = List Bool

∅ : World
∅ = []

_+1 : World → World
α +1 = false :: α

_↑1 : World → World
α ↑1 = true :: α
```

These one step definitions _+1 and _↑1 are extended to any number to produce _+ᵂ_ and _↑_ of type World → ℕ → World.

To the best of our knowledge, making the distinction between two forms of shifting operations for worlds has never been investigated in the context of representing names and binders.

In NAPA the type for names is a refinement of $\mathbb{N}$, kept abstract by not exporting its definition. From a set point of view, inhabitants of Name α are members of α. However we want worlds to form static information only and to be able to erase them before running the programs. To do so we will use a pair of a natural number and a proof it belongs to its world. Here is the membership predicate and the record definition for names:

```
_∈_ : ℕ → World → Set
_      ∈ []              = ⊥
zero   ∈ (false :: _)    = ⊥
zero   ∈ (true  :: _)    = ⊤
suc n  ∈ (_     :: xs)   = n ∈ xs

record Name α : Set where
  constructor _,_
  field
    name    : ℕ
    name∈α  : name ∈ α
```

Thanks to our definition of world membership which is canonical, equality of the name field implies equality on whole Name values. This means we get provable proof-irrelevance for the Name type without requiring an additional axiom.

2.4 Operations on names

The core operations on names are given in figure 3. The simplest name is zeroᴺ, it represents 0 and lives in any world shifted by one. One can add any constant to a name in any world with addᴺ, the resulting world clearly shows that this function (if parametric in α) does exactly its job. One can do the opposite operation with subtractᴺ. Thanks to its precise type this function is total and the inverse of addᴺ. Figure 4 depict the core operations we have on names. Starting from the bottom, worlds of the form ∅ ↑ k are names which are definitely bound. There value is completely known as stated by arrows with the Fin k type. Above we have names that may be bound or free (Name (α ↑ k)), a dynamic test (cmpᴺ) can tell whether it is bound or not. Above we have names that are known to be greater than k (Name (α +ᵂ k)), apart from that they are free names. Above we have free names (Name α). On the top we have impossible names since they are said to belong to an empty world. From them we can derive everything.

Given any world α, a name in the world $\alpha \uparrow \ell$ is either strictly lower than ℓ (and so also lives in $\emptyset \uparrow \ell$), or greater or equal to ℓ (thus also lives in $\alpha +^{W} \ell$). This is exactly what the cmpᴺ function is about. Given a name, it returns a disjoint sum of names which can be read in two parts. It first gives which side of the disjoint sum it stands, and second it gives a refined version of the input name.

The coerceᴺ function will be described in length in the next sub-section. The only way left to extract information from a name in an arbitrary world is to compare it with another name for equality

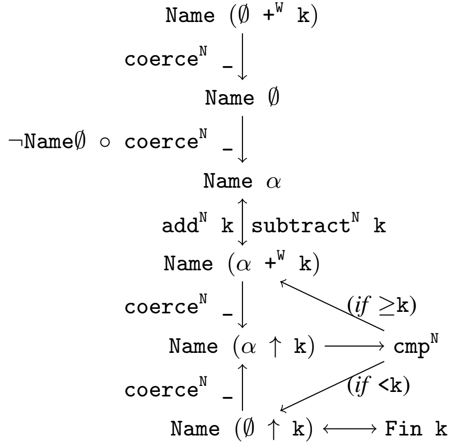

Figure 4. Operations on names

with $_{==}^{N}_$. Finally ¬Name∅ asserts that no name lives in the empty world. In a total language this is of great use to tackle impossible cases.

These primitives are enough to show an isomorphism between `Fin n` and `Name (∅ ↑ n)`. From this, every program involving `Fin` can be translated into our system. This means that our system does not restrict the programmer more than the `Fin` approach. However as soon as one uses finer types than `Name (∅ ↑ n)`, then fewer “wrong programs” type-check and more properties hold as we will see later.

2.5 A type for world inclusion witnesses

Having finer types also implies being able to give more types to the same value. Then cheaply moving from different types is a must. We call a function which has an output type different from the input type and behaves as the identity a coercion. To capture a great deal of coercions from a world α to a world β we focus on the case where α is included in β. In NAPA a type `_⊆_` is introduced to witness the inclusion between two worlds. Then the $\texttt{coerce}^{N}$ function turns a $\alpha \subseteq \beta$ into an identity function of type $\texttt{Name}\ \alpha \to \texttt{Name}\ \beta$. Here is the $\texttt{coerce}^{N}$ function and its alias `_⟨-because_-⟩` which is useful to keep the code separated from the typing/proof, in particular what is between the angle brackets can be safely skipped.

```
coerce^N : ∀ {α β} → α ⊆ β → Name α → Name β
coerce^N α⊆β (x , x∈α) = (x , ⊆-sound α⊆β x x∈α)

infix 0 _⟨-because_-⟩
_⟨-because_-⟩ : ∀{α β}→ Name α → α ⊆ β → Name β
_⟨-because_-⟩ n pf = coerce^N pf n
```

The inclusion relation also expresses the emptiness of a world α by using $\alpha \subseteq \emptyset$. We use this definition of emptiness as opposed to an equality with the empty world for two reasons. First $\emptyset$ is not the only empty world ($\emptyset$ `+1` is empty as well). Second having world equalities would require new definitions and would be heavy in comparison to the use of inclusions. Combining $\texttt{coerce}^{N}$ and ¬Name∅ turns any contradiction on names into an inclusion problem, reusing any automation done on this side.

```
¬Name : ∀ {α} → α ⊆ ∅ → ¬(Name α)
¬Name α⊆∅ = ¬Name∅ ∘ coerce^N α⊆∅
```

```
⊆-refl    : ∀{α}→ α ⊆ α

⊆-trans   : ∀{α β γ}→ α ⊆ β → β ⊆ γ → α ⊆ γ

⊆-∅       : ∀{α}→ ∅ ⊆ α

⊆-∅+1     : ∅ +1 ⊆ ∅

⊆-↑1-↑1   : ∀{α β}→ α ⊆ β ↔ α ↑1 ⊆ β ↑1

⊆-+1-+1   : ∀{α β}→ α ⊆ β ↔ α +1 ⊆ β +1

⊆-+1-↑1   : ∀{α}→ α +1 ⊆ α ↑1
```

Figure 5. Rules for world inclusion

We expose the inclusion relation with a set of rules given in figure 5. This states that the inclusion relation is reflexive, transitive, and has the empty world as lowest element. The ⊆-∅+1 rule states that ∅ +1 is empty. The inclusion is preserved both ways by the contexts `_↑1` and `_+1`. Finally `_+1` can be weakened in `_↑1`. This accounts for the fact that α ↑1 means $\{0\} \cup (\alpha\ +1)$ and so is a superset of α +1. This set of rules has been shown sound according to a semantic definition of inclusion, namely $\forall\ x \to x \in \alpha \to x \in \beta$. On top of these base rules, we derived others that we omit to define here but some are used in the code.

Sometimes one has to build complex inclusion witnesses. While inference would be of great effect here, we propose a modest syntactic tool to build them, namely the `⊆-Reasoning` module. It gives access to the transitivity `⊆-trans` in a style which focuses more on the intermediate states of the reasoning rather than the steps. The syntax is a list of worlds interleaved by inclusion witnesses, with two ■ around the last world. There is one example of its use in the following section. The code for `⊆-Reasoning` is given below for reference and can be safely skipped.

```
module ⊆-Reasoning where
  infix  2 finally
  infixr 2 _⊆⟨_⟩_

  _⊆⟨_⟩_ : ∀ x {y z} → x ⊆ y → y ⊆ z → x ⊆ z
  _ ⊆⟨ x⊆y ⟩ y⊆z = ⊆-trans x⊆y y⊆z

  finally : ∀ x y → x ⊆ y → x ⊆ y
  finally _ _ x⊆y = x⊆y

  syntax finally x y x⊆y = x ⊆⟨ x⊆y ⟩■ y ■
```

2.6 Singleton worlds!

We said that our worlds denote finite subsets of $\mathbb{N}$ and are more precise than in the `Fin` approach. Actually they can be as precise as we wish, since any subset of $\mathbb{N}$ can be described by our operations on worlds (∅, `_+1`, and `_↑1`). In particular they can be singleton worlds. From singleton worlds we build singleton types for names:

```
Worlds : ℕ → World
Worlds n = ∅ ↑1 +W n

Names : ℕ → Set
Names = Name ∘ Worlds

_s : ∀ n → Names n
_s n = zeroN +N n
```

Singleton worlds not only exist, they are also preserved by our two updating operations, namely $\texttt{add}^{N}$ and $\texttt{subtract}^{N}$.

```
adds : ∀ {n} k → Names n → Names (k +ℕ n)
adds {n} k x = addN k x
  ⟨-because ⊆-assoc-+ ⊆-refl n k −⟩

subtracts : ∀ {n} k → Names (k +ℕ n) → Names n
subtracts {n} k x = subtractN k x
  ⟨-because ⊆-assoc-+′ ⊆-refl n k -⟩
```

3. Using NAPA: examples and advanced operations

3.1 Some convenience functions

Here are a few functions built on top of the interface (without using the concrete representation of names). $\texttt{suc}^{N}$ is $\texttt{add}^{N}$ 1 and $\texttt{suc}^{N}\uparrow$ is a variant that includes a coercion from α +1 to α ↑1. The function $_^{N}$ turns a number n into a name that inhabits any world with at least n+1 consecutive binders.

```
sucN : ∀ {α} → Name α → Name (α +1)
sucN = addN 1

sucN↑ : ∀ {α} → Name α → Name (α ↑1)
sucN↑ = coerceN ⊆-+1↑1 ∘ sucN

_N : ∀ {α} n → Name (α ↑ suc n)
_N {α} n = zeroN +N n
  ⟨-because α ↑1 +W n ⊆⟨ ⊆-+-↑ n ⟩
            α ↑1 ↑  n ⊆⟨ ⊆-exch-↑-↑ 1 n ⟩■
            α ↑ suc n ■ -⟩
 where open ⊆-Reasoning
```

We call ℓ-bound, a name bound somewhere in the scope of ℓ binders. We call ℓ-free, a name free for all ℓ binders. In other words, a de Bruijn index is ℓ-bound if it is strictly less than ℓ; it is ℓ-free otherwise.

The `exportName` function tells whether a given name is ℓ-bound or ℓ-free. In case the name is free, an exported version of it is returned. This function forms the base case of exporting functions like `exportTm` explained later on.

```
-- Partial functions from A to B
A →? B = A → Maybe B

exportName : ∀ {α} ℓ →  Name (α ↑ ℓ)
                     →? Name α
exportName ℓ x
  with x <N ℓ
...  | inj1 _  = nothing
...  | inj2 x′ = just (x′ -N ℓ)
```

The function `shiftName` ℓ `k pf` shifts its argument by `k` if this name is ℓ-free, otherwise it leaves the ℓ-bound name untouched. This function makes use of $\texttt{cmp}^{N}$ and coerces the outputs to the required type. It also perform a coercion on the fly, giving extra flexibility for free.

```
shiftName : ∀ {α} ℓ k → (α +W k) ⊆ β
                      → Name (α ↑ ℓ)
                      → Name (β ↑ ℓ)
shiftName ℓ k pf x
 with x <N ℓ
... | inj1 x′ = x′
                ⟨-because pf1 -⟩
... | inj2 x′ = x′ +N k
                ⟨-because pf2 -⟩
  where
  pf1 = ⊆-cong-↑ ⊆-∅ ℓ
  pf2 = ⊆-trans (⊆-exch-+-+ ⊆-refl ℓ k)
                (⊆-ctx-+↑ pf ℓ)
```

The `protect↑` function shifts a name transformer. Let `f` be a function from names to names. The function `protect↑` ℓ `f` is a version of `f` that is applicable under ℓ binders. Let `x` be a name under ℓ binders. When `x` is ℓ-bound, it is left untouched by `protect↑`. When `x` is ℓ-free, we can subtract ℓ to `x`, give it to `f`, and then add ℓ to get the result. By combining `protect↑` and $\texttt{add}^{N}$ one obtain an alternative implementation of `shiftName` called `shiftName′`. However `shiftName` is more efficient since it avoids to subtracting ℓ to add it back after adding `k`.

```
protect↑ : ∀ {α β} ℓ
           → (Name α → Name β)
           → (Name (α ↑ ℓ) → Name (β ↑ ℓ))
protect↑ ℓ f x
 with x <N ℓ
... | inj1 x′ = x′
               ⟨-because ⊆-cong-↑ ⊆-∅ ℓ -⟩
... | inj2 x′ = f (x′ -N ℓ) +N ℓ
               ⟨-because ⊆-+-↑ ℓ -⟩

shiftName′ : ∀ {α β} ℓ k → (α +W k) ⊆ β
           → Name (α ↑ ℓ) → Name (β ↑ ℓ)
shiftName′ ℓ k pf = protect↑ (coerceN pf ∘ addN k) ℓ
```

3.2 Building terms

Building terms in NAPA is as easy as building them in the other nameless approaches we have seen. The structure is exactly the same, and the variables are made of numbers (of type ℕ) using $_^{N}$. Below we define the representation of the identity function as `idTm`, the application operator as `appTm` and finally the composition function as `compTm`:

```
idTm : ∀ {α} → Tm α
idTm = ƛ(V (0 N))

appTm : ∀ {α} → Tm α
appTm = ƛ(ƛ(V (1 N) · V (0 N)))

compTm : ∀ {α} → Tm α
compTm = ƛ(ƛ(ƛ(V (2 N)) · (V (1 N) · V (0 N)))))
```

3.3 Computing free variables

Our first example of functions over terms simply computes the list of free variables in the input term. The `fv` function below, while straightforward, has the subtle cases of binding constructs (ƛ and `Let`). In these cases we have to remove the bound variable from the list of free variables given by the recursive call. In this nameless representation this amounts to removing occurrences of 0 and subtract 1 to other name occurrences. This is done by the function `rm₀| which calls {agda|exportName 1` on each element of the list and merges the results. Note that forgetting to remove the bound variable will result in a type error. In the same vein the typing of `fv` ensures that all returned variables do appear free in the given term.

```
rm₀ : ∀ {α} → List (Name (α ↑1))
           → List (Name α)
rm₀ [] = []
rm₀ (x :: xs) with exportName 1 x
...              |  just x'  = x' :: rm₀ xs
...              |  nothing  = rm₀ xs

fv : ∀ {α} → Tm α → List (Name α)
fv (V x)         = [ x ]
fv (fct · arg)  = fv fct ++ fv arg
fv (ƛ t)         = rm₀ (fv t)
fv (Let t u)     = fv t ++ rm₀ (fv u)
```

3.4 Generic traversal

In order to build multiple traversal functions at once, we first define a generic function based on ideas from [8]. It is essentially a map/subst function where the free variables are transformed by a user-supplied function. Moreover a class of effects (an applicative functor) is allowed during the traversal. An applicative functor is halfway between a functor and a monad. Like a monad, an applicative functor has a unit called `pure`. It allows to embed any pure value as a potentially effectful one. The second operation called `_⊗_` is an effectful application, taking both an effectful function and argument and resulting in an effectful result.

```
module TraverseTm
          {E}   (E-app  : Applicative E)
          {α β} (trName : ∀ ℓ → Name (α ↑ ℓ)
                                → E (Tm (β ↑ ℓ)))
  where
  open Applicative E-app

  tr : ∀ ℓ → Tm (α ↑ ℓ) → E (Tm (β ↑ ℓ))
  tr ℓ (V x)     = trName ℓ x
  tr ℓ (t · u)   = pure _·_ ⊗ tr ℓ t ⊗ tr ℓ u
  tr ℓ (ƛ t)     = pure ƛ   ⊗ tr (suc ℓ) t
  tr ℓ (Let t u) = pure Let ⊗ tr ℓ t
                             ⊗ tr (suc ℓ) u

  trTm : Tm α → E (Tm β)
  trTm = tr 0
```

To put this traversal at work we successively instantiate some of its arguments. For instance you may have noticed the special case we made for variables. Mapping names to terms will bring us capture avoiding substitution almost for free. However, in the mean time we build `trTm′` which maps names to names. It does so by applying `pure V` to the name to name function:

```
open TraverseTm

trTm′ :
  ∀ {E} (E-app : Applicative E) {α β}
    (trName : ∀ ℓ → Name (α ↑ ℓ)
                  → E (Name (β ↑ ℓ)))
  → Tm α → E (Tm β)
trTm′ E-app trName
  = trTm E-app (λ ℓ x → pure V ⊗ trName ℓ x)
    where open Applicative E-app
```

Renaming functions In many functions over terms the handling of variables shares a common part. Given a variable under ℓ binders, we test if the variable is ℓ-bound. If so we leave it untouched, otherwise we subtract ℓ and go on a specific processing after which we add ℓ again to the free variables of the result. The traversal function is augmented by this processing of bound variables to build `renameTmA`. Then by picking either the identity functor (`id-app`), or the `Maybe` one we build two functions (a total and a partial one) to perform renamings, namely `renameTm` and `renameTm?`:

```
renameTmA : ∀ {E} (E-app : Applicative E)
              {α β} (θ : Name α → E (Name β))
            → Tm α → E (Tm β)
renameTmA E-app θ
  = trTm′ E-app (protect↑A E-app θ)

renameTm : ∀ {α β} → (Name α → Name β)
                   → Tm α → Tm β
renameTm θ = trTm′ id-app (protect↑ θ)
-- or
-- renameTm = renameTmA id-app

renameTm? : ∀ {α β} → (Name α →? Name β)
                    → Tm α →? Tm β
renameTm? = renameTmA Maybe.applicative
```

Lifting name functions Any operation on names can now be lifted to terms:

```
addTm : ∀ {α} k → Tm α → Tm (α +ᵂ k)
addTm = renameTm ∘ addᴺ

subtractTm : ∀ {α} k → Tm (α +ᵂ k) → Tm α
subtractTm = renameTm ∘ subtractᴺ

exportTm? : ∀ {α} ℓ → Tm (α ↑ ℓ) →? Tm α
exportTm? = renameTm? ∘ exportName
```

While `coerceᴺ` can be lifted to terms in the same way, using `trTm′` directly enables to by-pass the `protect↑` dynamic tests and directly "protect" the inclusion witness with an appropriate inclusion rule.

```
coerceTm : ∀ {α β} → α ⊆ β → Tm α → Tm β
coerceTm pf = trTm′ id-app (coerceᴺ ∘ ⊆-cong-↑ pf)
-- or less efficiently:
-- coerceTm = renameTm ∘ coerceᴺ
```

Lifting $\texttt{add}^{\mathbb{N}}$ to terms can be done more efficiently (than using `protect↑`) as well. Here the dynamic test performed by `protect↑` is necessary. However when the name is ℓ-free we subtract ℓ to add it back after adding `k`. The function `shiftName` avoids this extra computation, hence the following `shiftTm`:

```
shiftTm : ∀ {α β} k → (α +ᵂ k)⊆ β → Tm α → Tm β
shiftTm k p = trTm′ id-app (λ ℓ → shiftName ℓ k p)
-- or less efficiently:
-- shiftTm k pf = renameTm (coerceᴺ pf ∘ addᴺ k)
```

A special case of `renameTm?` is the so-called `closeTm?`. This function takes a term in any world and checks if the term is closed. If so, the same term is returned in the empty world. Otherwise the function fails by returning `nothing`:

```
closeTm? : ∀ {α} → Tm α →? Tm ∅
closeTm? = renameTm? (const nothing)
```

Capture avoiding substitution To implement capture avoiding substitution for the `Tm` type we solely need a specific `trName` function for `trTm`. Here substitutions are represented as functions from names to terms. The function `substVarTm` handles the case for variables. This function is very close to `protect↑` but extended to functions returning terms.

```
substVarTm : ∀{α β} → (Name α → Tm β) →
               ∀ ℓ → Name (α ↑ ℓ) → Tm (β ↑ ℓ)
substVarTm f ℓ x
 with x <ᴺ ℓ
... | inj₁ x′ = V (x′
                ⟨-because ⊆-cong-↑ ⊆-∅ ℓ -⟩)
... | inj₂ x′ = shiftTm ℓ (⊆-+-↑ ℓ) (f (x′ -ᴺ ℓ))
```

The main function `substTm` instantiates `trTm` with the identity applicative functor and `substVarTm`:

```
substTm : ∀ {α β} → (Name α → Tm β)
                  → Tm α → Tm β
substTm = trTm id-app ∘ substVarTm
```

As an illustration, the function `β-red` takes the body of a λ-abstraction and calls `substTm` with the substitution that replaces 0 with a received replacement term:

```
β-red : ∀ {α} → Tm (α ↑1) → Tm α → Tm α
β-red {α} f a = substTm (ϕ ∘ exportName 1) f
  where ϕ : Maybe (Name α) → Tm α
        ϕ (just x) = V x
        ϕ nothing  = a
```

3.5 Deciding term α-equivalence

Our next example focuses on comparison of terms. We first define `αEq`, where `αEq F` is the type of functions comparing `F` structures.

```
αEq : (F : World → Set) (α β : World) → Set
αEq F α β = F α → F β → Bool
```

We show that all the subtle work is done at the level of names in a separate and reusable function called `αeqName`. This function takes a function that compares two free names and builds one that compares two names under ℓ bindings. It does so by comparing both of them to ℓ. If they are both bound they can be safely compared using `_==ᴺ_` since they now are of the same type. If they are both free, they can be compared using the function received as argument. Otherwise they are different.

```
αeqName : ∀ {α β} ℓ → αEq Name α β
                    → αEq Name (α ↑ ℓ) (β ↑ ℓ)
αeqName ℓ Γ x y with x <ᴺ ℓ | y <ᴺ ℓ
... | inj₁ x′ | inj₁ y′  = x′ ==ᴺ y′
... | inj₂ x′ | inj₂ y′  = Γ (x′ -ᴺ ℓ) (y′ -ᴺ ℓ)
... | _       | _        = false
```

The `αeqTm` function structurally compares two terms in a simple way, only keeping track of the number of traversed binders and calling `αeqName` at variables.

```
αeqTm : ∀ {α β} → αEq Name α β → αEq Tm α β
αeqTm {α} {β} Γ = go 0 where
  go : ∀ ℓ → αEq Tm (α ↑ ℓ) (β ↑ ℓ)
  go ℓ (V x)      (V y)      = αeqName ℓ Γ x y
  go ℓ (t · u)    (v · w)    = go ℓ t v ∧ go ℓ u w
  go ℓ (ƛ t)      (ƛ u)      = go (suc ℓ) t u
  go ℓ (Let t u)  (Let v w)  = go ℓ t v
                             ∧ go (suc ℓ) u w
  go _ _ _                   = false
```

4. Soundness using logical relations

Since our library is written in a type-safe language, one may wonder what soundness properties are to be proved. Moreover our names are indexed by worlds and hold membership proofs. The functions over names are given precise types and have been shown to be type-safe. However we still wish to show that our library respects a model of good behavior with respect to names and binders. Our model is based on logical relations indexed/directed by types. This technique [4] is independent of this work and enables to define a notion of program equivalence. We use this technique to capture good behavior of functions involving names and binders. Using this technique, the set of specific definitions is kept to a minimum of one per introduced type (`World`, `Name`, and `_⊆_`). One proof per value introduced has to be done, which keeps the development modular and forward-compatible to the addition of more features.

This section is organized as follows. First we recall the basics of logical relations. Then we give a toy example to practice a bit. Then relations for worlds, names, and world inclusions are given. Finally we make use of the construction to obtain free theorems [12] about world-polymorphic functions over the `Tm` type.

4.1 Recap of the framework

A relation is said to be type-directed when it is recursively defined on the structure of types. Let $\mathcal{R}$ be such a type-directed relation, and let τ be a type. Then, $\mathcal{R}\tau$ is a relation on values of type τ, namely $\mathcal{R}\tau$ `:` $\tau \to \tau \to$ `Set`. Recall that `Set` is also the type of propositions in AGDA.

A type-directed relation is called "logical" when the case for functions is defined extensionally. Here extensionally means that two functions are related when they produce related results out of related arguments. Let A_r be a relation for the arguments and B_r a relation for results. Two functions f_1 and f_2 are related if and only if for every pair of arguments (x_1, x_2) related by A_r, the results $f_1\ x_1$ and $f_2\ x_2$ are related by B_r. This definition can be given in AGDA as well:

```
RelatedFunctions Aᵣ Bᵣ f₁ f₂ =
          ∀ {x₁ x₂} → Aᵣ x₁ x₂
                    → Bᵣ (f₁ x₁) (f₂ x₂)
```

We say that a program or a value fits a logical relation when it is related to itself by the relation indexed by its type. We say that a logical relation is universal if every well-typed program fits the logical relation. This notion of universality was originally coined by John Reynolds as the "Abstraction Theorem" [11]. We call the "AGDA logical relation" the one defined by Bernardy et al. [4] for a $\mathcal{PTS}$ (pure type system) and naturally extended as they suggest to other features of AGDA. While no complete mechanized proof has been done for this we will consider the AGDA logical relation as universal.

To simplify matters, the definitions shown here are not universe polymorphic. You can find universe-polymorphic definitions in our complete AGDA development [9].

To formally define a logical relation indexed by types, a common technique is to first inductively define the structure of types. This is known as a "universe of codes" `U`. Then one defines a function called `El` from codes to types. Finally one defines by induction a function called ⟦_⟧ from codes to relations on elements of type described by the given code. In AGDA the ⟦_⟧ function has the following type: `(τ : U) → El τ → El τ → Set`. Because, when types contain variables, a good deal of complexity is added to this scheme, we opt for a lighter scheme. We do not define `U`, `El`, and ⟦_⟧.

Instead, for each type constructor κ, we define a relation ⟦κ⟧ For the function type constructor the `RelatedFunctions` definition above is a good start. Actually this is a fine definition for non-dependent functions. The dependent version of `RelatedFunctions`, called ⟦Π⟧ here, passes the relation argument A_r x_1 x_2 called x_r to the relation for results B_r. In short the relation for results now depends on the relation for arguments. Here is the definition in AGDA:

```
⟦Π⟧ Aᵣ Bᵣ f₁ f₂ = ∀ {x₁ x₂} (xᵣ : Aᵣ x₁ x₂)
                        → Bᵣ xᵣ (f₁ x₁) (f₂ x₂)
```

Note that this definition generalizes the case of non-dependent functions and universal quantifications as well. For non-dependent functions we simply provide a relation for results which ignores its first argument (equivalent to `RelatedFunctions` and noted _⟦→⟧_ from now on). For universal quantifications, since the arguments are types, all we need is a relation for types (members of $\mathtt{Set}_0$) themselves. Following our convention we call this relation ⟦$\mathtt{Set}_0$⟧, its definition is the set of relations between its arguments:

```
⟦Set₀⟧ : Set₀ → Set₀ → Set₁
⟦Set₀⟧ A₁ A₂ = A₁ → A₂ → Set₀
```

For reference, full definitions for core type theory are given in figure 6. We now have all the building blocks of the meta-function ⟦_⟧ and we will not materialize it more here. We will apply the ⟦_⟧ function manually. To this end it suffices to replace each constructor κ by ⟦κ⟧, each non dependent arrow by ⟦→⟧, each dependent arrow `(x : A) → B` by ⟨ x_r : ⟦ A ⟧ ⟩⟦→⟧ ⟦ B ⟧. By convention we subscript the variables by `r`. Applications are translated to applications. Because of dependent types, this translation has to be extended to all terms but we will not do it here. Finally here are a few examples of the manual use of the ⟦_⟧ function:

```
⟦ ℕ → ℕ → Bool ⟧ =
⟦ℕ⟧ ⟦→⟧ ⟦ℕ⟧ ⟦→⟧ ⟦Bool⟧ =
λ f₁ f₂ →
  ∀ {x₁ x₂} (xᵣ : ⟦ℕ⟧ x₁ x₂)
    {y₁ y₂} (yᵣ : ⟦ℕ⟧ y₁ y₂)
  → ⟦Bool⟧ (f₁ x₁ y₁) (f₂ x₂ y₂)
```

```
data ⟦⊥⟧ : ⟦Set₀⟧ ⊥ ⊥ -- no constructors

data ⟦Bool⟧ : ⟦Set₀⟧ Bool Bool where
  ⟦true⟧  : ⟦Bool⟧ true  true
  ⟦false⟧ : ⟦Bool⟧ false false

data ⟦ℕ⟧ : ⟦Set₀⟧ ℕ ℕ where
  ⟦zero⟧ : ⟦ℕ⟧                 zero zero
  ⟦suc⟧  : (⟦ℕ⟧ ⟦→⟧ ⟦ℕ⟧)  suc  suc

data _⟦⊎⟧_ {A₁ A₂ B₁ B₂} (Aᵣ : ⟦Set₀⟧ A₁ A₂)
                          (Bᵣ : ⟦Set₀⟧ B₁ B₂) :
             A₁ ⊎ B₁ → A₂ ⊎ B₂ → Set₀ where
  ⟦inj₁⟧ : (Aᵣ → Aᵣ ⟦⊎⟧ Bᵣ) inj₁ inj₁
  ⟦inj₂⟧ : (Bᵣ → Aᵣ ⟦⊎⟧ Bᵣ) inj₂ inj₂
```

Figure 7. Logical relations for data types

```
⟦ (A : Set₀) → A → A ⟧ =
⟦Π⟧ ⟦Set₀⟧ (λ Aᵣ → Aᵣ ⟦→⟧ Aᵣ) =
λ f₁ f₂ →
  ∀ {A₁ A₂} (Aᵣ : A₁ → A₂ → Set₀)
    {x₁ x₂} (xᵣ : Aᵣ x₁ x₂)
  → Aᵣ (f₁ A₁ x₁) (f₂ A₂ x₂)


-- Using the notation instead of ⟦Π⟧:
⟦ (A : Set₀) → List A ⟧ =
⟨ Aᵣ : ⟦Set₀⟧ ⟩⟦→⟧ ⟦List⟧ Aᵣ =
λ l₁ l₂ →
  ∀ {A₁ A₂} (Aᵣ : A₁ → A₂ → Set₀)
  → ⟦List⟧ Aᵣ (l₁ A₁) (l₂ A₂)
```

We now have the definition of the AGDA logical relation for the core type theory part. It extends nicely to inductive data types and records. The process is as follows: for each constructor κ of type τ, declare a new constructor ⟦κ⟧ whose type is ⟦ τ ⟧ κ κ. This process applies to type constructors and data constructors of data types, and type constructors and fields of record types. For reference, the logical relations for data types we use in this development are in figure 7.

4.2 An example: booleans represented by numbers

We wish to explain how logical relations can help build a safe interface to an abstract type. To do so we introduce a tiny example about booleans represented using natural numbers. We want 0 to represent `false` and any other number to represent `true`. Therefore the boolean disjunction can be implemented using addition. We show that logical relations help build a model, ensure that a given implementation respects this model, and finally show that a client that uses only the interface will also respect the model.

Note however that this is a toy example in several ways. There are no polymorphic functions in the interface, so no interesting free theorems are to be expected. While we could prove the safety by defining a representation predicate in unary style, the logical relations approach is different. It relies on comparing concrete data as opposed to mapping to abstract data. The unary construction would allow for a simpler construction, however this oversimplifies the problem here and would be no longer useful for proving NAPA.

Our tiny implementation of booleans using natural numbers is given below. It contains a type `B` that we want to keep abstract. It contains obvious definitions for `true`, `false`, and the disjunction

```
⟦Set₀⟧ : ∀ (A₁ A₂ : Set₀) → Set₁
⟦Set₀⟧ A₁ A₂ = A₁ → A₂ → Set₀

⟦Set₁⟧ : ∀ (A₁ A₂ : Set₁) → Set₂
⟦Set₁⟧ A₁ A₂ = A₁ → A₂ → Set₁

_⟦→⟧_ : ∀ {A₁ A₂ B₁ B₂} → ⟦Set₀⟧ A₁ A₂ → ⟦Set₀⟧ B₁ B₂ → ⟦Set₀⟧ (A₁ → B₁) (A₂ → B₂)
Aᵣ ⟦→⟧ Bᵣ = λ f₁ f₂ → ∀ {x₁ x₂} → Aᵣ x₁ x₂ → Bᵣ (f₁ x₁) (f₂ x₂)

infixr 0 _⟦→⟧_

⟦Π⟧ : ∀ {A₁ A₂} (Aᵣ : ⟦Set₀⟧ A₁ A₂)
        {B₁ B₂} (Bᵣ : (Aᵣ ⟦→⟧ ⟦Set₀⟧) B₁ B₂)
      → ((x : A₁) → B₁ x) → ((x : A₂) → B₂ x) → Set₁
⟦Π⟧ Aᵣ Bᵣ = λ f₁ f₂ → ∀ {x₁ x₂} (xᵣ : Aᵣ x₁ x₂) → Bᵣ xᵣ (f₁ x₁) (f₂ x₂)

syntax ⟦Π⟧ Aᵣ (λ xᵣ → f) = ⟨ xᵣ : Aᵣ ⟩⟦→⟧ f

⟦∀⟧ : ∀ {A₁ A₂} (Aᵣ : ⟦Set₀⟧ A₁ A₂)
        {B₁ B₂} (Bᵣ : (⟦Set₀⟧ ⟦→⟧ ⟦Set₀⟧) B₁ B₂)
      → ⟦Set₁⟧ ({x : A₁} → B₁ x) ({x : A₂} → B₂ x)
⟦∀⟧ Aᵣ Bᵣ = λ f₁ f₂ → ∀ {x₁ x₂} (xᵣ : Aᵣ x₁ x₂) → Bᵣ xᵣ (f₁ {x₁}) (f₂ {x₂})

syntax ⟦∀⟧ Aᵣ (λ xᵣ → f) = ∀⟨ xᵣ : Aᵣ ⟩⟦→⟧ f
```

Figure 6. Logical relations for core types

_∨_. It intentionally has a dubious function `is42?` which breaks our still informal expectations from such a module.

```
B : Set
B = ℕ

false : B
false = 0

true : B
true = 1


_∨_ : B → B → B
m ∨ n = m + n


is42? : B → B
is42? 42 = true
is42? _  = false
```

The next step is to define our expectations. To do so, we give a binary relation which tells when two B values have the same meaning. We do so with an (inductive) data type named ⟦B⟧ which states that 0 is related only with itself, and that any two non zero numbers are related:

```
data ⟦B⟧ : B → B → Set where
  ⟦false⟧ : ⟦B⟧ 0 0
  ⟦true⟧  : ∀ {m n} → ⟦B⟧ (suc m) (suc n)
```

When plugged into the machinery of logical relations this single definition suffices to define a complete model of well-typed programs. However, the plumbing requires some care. While the AGDA logical relation is universal, we have no such guarantee about the AGDA logical relation where the relation for B is no longer ⟦ℕ⟧ but ⟦B⟧. Fortunately changing the relation at a given type (B here) can be done safely. All we have to do is to consider programs abstracted away from B and its operations: `true`, `false` and _∨_. This can be done either through a mechanism for abstract types, or by requiring the client to be a function taking the implementation for B and its operations as argument.

However to use ⟦B⟧ as the relation for B, we have to show that the definitions which make use of the representation of B actually fit the relation. Since ⟦`true`⟧ and ⟦`false`⟧ are obvious witnesses for `true` and `false`, only _∨_ and `is42?` are left to be proved. Each time the goal to prove is mechanical: wrap the type with ⟦·⟧ on each constructor and put the name of the function twice to state we want it to be related to itself. Here is the definition for _⟦∨⟧_:

```
_⟦∨⟧_ : (⟦B⟧ ⟦→⟧ ⟦B⟧ ⟦→⟧ ⟦B⟧) _∨_ _∨_
```

The type of _⟦∨⟧_ means that given inputs related in the model, the results are related in the model as well. Once unfolded the type looks like:

```
_⟦∨⟧_ : ∀ {x₁ x₂} (xᵣ : ⟦B⟧ x₁ x₂)
          {y₁ y₂} (yᵣ : ⟦B⟧ y₁ y₂)
        → ⟦B⟧ (x₁ ∨ y₁) (x₂ ∨ y₂)
```

The fact that input arguments come as implicit arguments will greatly shorten definitions. Now, thanks to the inductive definition of _+_, simply pattern-matching on the first relation suffices to reduce the goal, and allows this nice looking definition where we see the usual lazy definition of the left biased disjunction:

```
⟦false⟧ ⟦∨⟧ x = x
⟦true⟧  ⟦∨⟧ _ = ⟦true⟧
```

Let us now consider a proof for the `is42?` function. Fortunately there is no such proof since this function gladly breaks the intended abstraction. Instead we simply prove its negation by exhibiting that

given two related inputs (42 and 27) we get non related outputs (is42? 42 = 1 and is42? 27 = 0).

```
¬⟦is42?⟧ : ¬((⟦B⟧ ⟦→⟧ ⟦B⟧) is42? is42?)
¬⟦is42?⟧ ⟦is42?⟧ with ⟦is42?⟧ {42} {27} ⟦true⟧
...                   | () -- absurd
```

Note that is42? is rejected by our model with no considerations about the other exported functions. Indeed with another implementation of _∨_ there would be no way to produce 42 and so no way to expose the wrong behavior of is42? using the interface. Using a model provides a better forward compatibility and enables proofs to be done in a modular way.

4.3 Relations for NAPA

For NAPA, we apply the same process as with booleans. We define our expectations, by defining relations for introduced types (worlds, names, and inclusions). Finally we prove that each value/function exported fits the relation.

4.3.1 Relations for NAPA types

Valid names are those which belong to their worlds, names with the same meaning are those related by the relation between their worlds. What matters is not just the fact that two worlds are related, but "how" they are related. Indeed this will dictate when two names are related. We need to define a set of valid relations between worlds. The more relations are accepted, the more power it gives to the free theorems of world-polymorphic functions. However we want the equality test on names to be accepted. Thus we at least need the relation to preserve name equalities across the relation in both directions. We require these relations to be functional and injective:

```
FunctionalAndInjective R =
   ∀ x1 y1 x2 y2 → R x1 x2 → R y1 y2
                 → x1 ≡ y1 ↔ x2 ≡ y2
```

The relation ⟦World⟧ between two worlds α_1 and α_2 is the set of functional and injective relations between Name α_1 and Name α_2. Then, two names x (in α_1) and y (in α_2) are related by ⟦Name⟧ α_r if and only if they are related by the α_r relation.

For the _⟦⊆⟧_ relation, we exploit the fact there is only one way to use an inclusion witness, namely coerce^{N}. Thus, for the purpose of building the model, we identify inclusions with their use in coerce^{N}. Put differently, whatever the representation for inclusions is, the model takes them as functions from names to names. Yet another way to look at it is from the perspective of relation inclusions. A relation $\mathcal{R}_1$ is included in a relation $\mathcal{R}_2$ if and only if all pairs related by $\mathcal{R}_1$ are related by $\mathcal{R}_2$ as well. The coerce^{N} function behaving like the identity function all these definitions coincide. For instance if we expand the definitions for ⟦Name⟧ and ⟦→⟧, we get the definition for relation inclusion modulo the coercions:

```
_⟦⊆⟧_ αr βr p1 p2
  = ∀ {x1 x2} → (x1 , x2) ∈ αr
              → (coerceᴺ p1 x1 , coerceᴺ p2 x2) ∈ βr
```

We now have to define operations on worlds that fit the logical relation. The case for ⟦∅⟧ is trivial. Then α_r ⟦+1⟧ is defined by $\{ (x+1, y+1) \mid (x, y) \in \alpha_r \}$ and α_r ⟦↑1⟧ is defined by $\{(0, 0)\} \cup \alpha_r$ ⟦+1⟧. We shown that both operations preserve functionality and injectiveness. Here are the signatures that these operations fit the relation:

```
record ⟦World⟧ α1 α2 : Set where
  constructor _,_
  field
    R : Name α1 → Name α2 → Set
    R-fun-inj : FunctionalAndInjective R

⟦Name⟧ : (⟦World⟧ ⟦→⟧ ⟦Set0⟧) Name Name
⟦Name⟧ (R , _) x1 x2 = R x1 x2

_⟦⊆⟧_ : (⟦World⟧ ⟦→⟧ ⟦World⟧ ⟦→⟧ ⟦Set0⟧)
        _⊆_ _⊆_
_⟦⊆⟧_ αr βr α1⊆β1 α2⊆β2
  = (⟦Name⟧ αr ⟦→⟧ ⟦Name⟧ βr) (coerceᴺ α1⊆β1)
                               (coerceᴺ α2⊆β2)
```

Figure 8. Relations for NAPA types

```
⟦∅⟧    : ⟦World⟧ ∅ ∅
_⟦+1⟧  : (⟦World⟧ ⟦→⟧ ⟦World⟧) _+1 _+1
_⟦↑1⟧  : (⟦World⟧ ⟦→⟧ ⟦World⟧) _↑1 _↑1
```

4.3.2 NAPA values fit the relation

We now give a short overview of the proofs needed to show that our functions fit the relation. Thanks to the definition of _⟦↑1⟧, zero^{N} fits the relation by definition. Thanks to the definition of _⟦+1⟧, add^{N} 1 and subtract^{N} 1 fit the relation as well. These two are later extended to add^{N} k and subtract^{N} k by an induction on k. Since ⟦¬Name∅⟧ receives names in the empty world, it trivially holds. Here is for instance the type signature for ⟦add^{N}⟧:

```
⟦addᴺ⟧ : (∀⟨ αr : ⟦World⟧    ⟩⟦→⟧
          ⟨ kr : ⟦ℕ⟧        ⟩⟦→⟧
          ⟦Name⟧ αr ⟦→⟧
          ⟦Name⟧ (αr ⟦+ᵂ⟧ kr)) addᴺ addᴺ
```

For _⟦==^{N}⟧_, once unfolded, the statement tells that the equality test commutes with a renaming. This means that the result of the equality test does not change when its inputs are consistently renamed. The proof for _⟦==^{N}⟧_ is done in two parts. First, we have to relate the Boolean-valued function _==^{N}_ to the fact it decides equality on names. Second, we make use of the two properties (functionality and injectiveness) of the relation between the two worlds.

The definition of cmp^{N} is not a simple induction on its first argument. It calls _<=_ (which does an induction) and returns a Boolean value. Based on this Boolean value, the function cmp^{N} returns either inj$_1$ or inj$_2$ (the constructors of _⊎_) with the same name (with a different proof). To prove _⟦cmp^{N}⟧_ we show the equivalence with a simpler inductive function and show that this simpler function is in the relation. Thanks to the extensionality of the logical relation, no additional axiom is required to show that cmp^{N} fits the relation.

Thanks to the definition of _⟦⊆⟧_, the proof that coerce^{N} fits the relation is a simple application of the hypotheses. Then the real job is to show that all the inclusion rules fit the relation. This means that they all behave as the identity function. All eleven rules stated in figure 5 have been shown to fit the relation.

4.4 On the strength of free theorems

Every well-typed function comes with a free theorem [12]. However depending on the type of the function the strength of the theorem varies a lot. For instance at type `Int → Bool` it says no more than the function is deterministic. At type `∀{A : Set} → A → A` it says that the function behaves like the identity function, which is much stronger. We now give a few elements of what can affect the strength of free theorems in our context. We will continue to use our `Tm` type to represent some data structures with names and binders but it could be any other.

Various term types The weakest type we can give to a value with names and binders is `Tm (∅ ↑ ℓ)` for a given ℓ. Such a term can have a statically unknown number of distinct free variables, but we know that these variables are comprised in the interval [0 .. ℓ-1]. The free theorem of `∀{ℓ} → Tm (∅ ↑ ℓ) → Tm (∅ ↑ ℓ)` says no more than "the function is deterministic". We also know because of typing that the resulting term cannot have ℓ-free occurrences.

A stronger type is to use any world instead of the empty world, but still have a known amount of bindings. A function of type `∀{α ℓ} → Tm (α ↑ ℓ) → Tm (α ↑ ℓ)` has a stronger free theorem. It says that the function commutes with a renaming of the variables in the world α (section 4.5). This is a common type to deal with open terms under a partial environment: `∀{α ℓ} → Vec Info ℓ → Tm (α ↑ ℓ) → Tm (α ↑ ℓ)`.

The strongest type for terms is to make no assumptions on their free variables. This can be done by quantifying by an arbitrary world. The free theorem associated with the type `∀ {α} → Tm α → Tm α` says that the function commutes with renamings of the free variables. This particular type is studied in detail in the section 4.5.

Extra arguments Note that adding extra arguments to a function can drastically affect the strength of its free theorem. An extreme example is the type `∀{α} → (α ≡ ∅) → Tm α → Tm α`, which is ruined by its `α ≡ ∅` argument. While the above example is extreme, this is an important point to watch out for when adding extra arguments to a function. Another example is the type `∀{α β} → Tm α → Tm β → Tm β` versus `∀{α β} → (Name α → Name β → Bool) → Tm α → Tm β → Tm β`, the first one cannot compare the free variables of the two given terms while the second can apply a user-supplied function to do so.

Shifting versus adding Last but not least using $+^{W}$ instead of ↑ significantly improves the strength of the associated free theorem. Consider the function:

```
protectedAdd : ∀ {α} ℓ k → Name (α ↑ ℓ)
                         → Name (α +ᵂ k ↑ ℓ)
protectedAdd ℓ k = protect↑ ℓ (addᴺ k)
```

Consider now a weaker type, namely:

```
protectedAdd↑ : ∀ {α} ℓ k → Name (α ↑ ℓ)
                          → Name (α ↑ k ↑ ℓ)
protectedAdd↑ ℓ k = protect↑ ℓ (addᴺ↑ k)
```

We simply replaced the occurrence of $+^{W}$ by ↑. The consequences of this change are disastrous: this type allows more behaviors for its functions. Indeed the $\texttt{add}^{N}$ `k` function can be given the latter type (but not the former) when using the appropriate inclusion witness to exploit the commutativity of ↑:

```
unprotectedAdd : ∀ {α} ℓ k → Name (α ↑ ℓ)
                           → Name (α ↑ k ↑ ℓ)
unprotectedAdd ℓ k
  = coerceᴺ (⊆-exch-↑-↑′ ℓ k) ∘ addᴺ↑ k
```

In our previous work [10] we had only the ↑ operation. Our function `shiftName` (`protectedAdd` in this example) was defined with the `unprotectedAdd` behavior. It took us a long time to discover this mistake, since we thought that our logical relation argument was enough. In summary we want to emphasis that all the logical relations proofs have to be taken with great care. A weaker function type than expected can ruin the intended informal properties.

4.5 Using logical relations and parametricity

To formally show that a world-polymorphic function `f` of type `∀ {α} → Tm α → Tm α` commutes with a renaming of the free variables, we proceed as follows. First we recall the natural definition of logical relation on the type `Tm`. Second we present the type `Ren` of renamings as injective functions. Third renaming is shown equivalent to being related at the type `Tm`. Finally we prove our commutation lemma by using the free-theorem associated to the function `f`.

The logical relation for the type `Tm` is mechanical. It states that two terms are related if they have the same structure and related free variables:

```
data ⟦Tm⟧ {α₁ α₂} (αᵣ : ⟦World⟧ α₁ α₂) :
          Tm α₁ → Tm α₂ → Set where
  ⟦V⟧     : ∀ {x₁ x₂} (xᵣ : ⟦Name⟧ αᵣ x₁ x₂)
           → ⟦Tm⟧ αᵣ (V x₁) (V x₂)
  _⟦·⟧_   : ∀ {t₁ t₂ u₁ u₂}
              (tᵣ : ⟦Tm⟧ αᵣ t₁ t₂)
              (uᵣ : ⟦Tm⟧ αᵣ u₁ u₂)
           → ⟦Tm⟧ αᵣ (t₁ · u₁) (t₂ · u₂)
  ⟦ƛ⟧     : ∀ {t₁ t₂} (tᵣ : ⟦Tm⟧ (αᵣ ⟦↑1⟧) t₁ t₂)
           → ⟦Tm⟧ αᵣ (ƛ t₁) (ƛ t₂)
  ⟦Let⟧   : ∀ {t₁ t₂ u₁ u₂}
              (tᵣ : ⟦Tm⟧ αᵣ t₁ t₂)
              (uᵣ : ⟦Tm⟧ (αᵣ ⟦↑1⟧) u₁ u₂)
           → ⟦Tm⟧ αᵣ (Let t₁ u₁) (Let t₂ u₂)
```

Then we need a notion of renaming. We choose to use injective functions over names. The type for a renaming is `Ren`, and the functions ⟨_⟩ and ⟪_⟫ respectively convert a renaming to a function over names, and to a relation over worlds:

```
Ren : (α β : World) → Set

⟨_⟩ : ∀ {α β} → Ren α β → Tm α → Tm β

⟪_⟫ : ∀ {α β} → Ren α β → ⟦World⟧ α β
```

We now observe that given a renaming Φ, it is equivalent for two terms $\texttt{t}_1$ and $\texttt{t}_2$ to be related by `⟦Tm⟧ ⟪ Φ ⟫` and for $\texttt{t}_2$ to be equal to $\texttt{t}_1$ renamed with Φ.

```
⟦Tm⟧⇔rename :
   ∀ {α β} (Φ : Ren α β) {t₁ t₂}
   → ⟦Tm⟧ ⟪ Φ ⟫ t₁ t₂ ⇔ (⟨ Φ ⟩ t₁) ≡ t₂
```

Finally given a function `f` and a proof $\mathtt{f}_r$ that `f` is in the logical relation, we can show that any renaming Φ commutes with the function `f`. To prove so we apply our `⟦Tm⟧⇔rename` lemma in both directions and use $\mathtt{f}_r$ with the renaming Φ lifted as a `⟦World⟧`.

```
ren-comm :
  (f : ∀ {α} → Tm α → Tm α)
  (fᵣ:(∀⟨ αᵣ : ⟦World⟧ ⟩⟦→⟧ ⟦Tm⟧ αᵣ ⟦→⟧ ⟦Tm⟧ αᵣ)f f)
  → ∀ {α β} (Φ : Ren α β)
  → ⟨ Φ ⟩ ∘ f ≗ f ∘ ⟨ Φ ⟩
ren-comm f fᵣ Φ t
    = ⟦Tm⟧⇒rename Φ
          (fᵣ ⟪ Φ ⟫ (rename⇒⟦Tm⟧ Φ ≡.refl))
```

5. Conclusion and future work

We have shown a new approach for a safer nameless programming interface. Our work relies on a different notion of worlds both finer and more abstract. The type of names while being represented as a natural number is kept abstract as well. Only a few functions are required from the interface to get complete expressiveness. Indeed while our worlds are more precise, nothing forces the programmer to be precise with them. Thus there is no loss of expressiveness compared to the `Fin` approach. Through concrete examples we have shown how we can program in this system using classical examples like capture avoiding substitution. However the most challenging result comes from the solid mechanized development we have made in AGDA. This development not only demonstrates the soundness of our approach but allows to derive properties of functions using the system. This way we have shown that world-polymorphic functions over terms commute with renamings of free variables. These soundness results are shown in a modular way and reuse the solid foundations of logical relations.

As future work we would like to explore more properties of this system. We have also seen that a major convenience problem in our system was to build world inclusion witnesses. We would like to address this problem through a witness inference system built within AGDA, using the recent reflection mechanism. Another time-consuming task was to apply the AGDA logical relations to types and operations, so we would like to explore the integration of inspection primitives and meta-programming facilities for AGDA, namely TEMPLATE AGDA. Finally we would like to investigate how this system could scale to representations of well-typed terms.

Acknowledgements Thanks to François Pottier, Jean-Philippe Bernardy, Alexandre Pilkiewicz, and the anonymous reviewers for providing us with very valuable feedback.

References

[1] Thorsten Altenkirch. A formalization of the strong normalization proof for System F in LEGO. In J.F. Groote M. Bezem, editor, *Typed Lambda Calculi and Applications*, LNCS 664, pages 13 – 28, 1993.

[2] Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda terms using generalized inductive types. In *Computer Science Logic*, volume 1683 of *Lecture Notes in Computer Science*, pages 453–468. Springer, 1999.

[3] Françoise Bellegarde and James Hook. Substitution: A formal methods case study using monads and transformations. *Sci. Comput. Program.*, 23(2-3):287–311, 1994.

[4] Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. Parametricity and dependent types. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 345–356, New York, NY, USA, 2010. ACM.

[5] Richard Bird and Ross Paterson. de Bruijn notation as a nested datatype. *Journal of Functional Programming*, 9(1):77–91, January 1999.

[6] Nicolaas G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

[7] Conor McBride and James McKinna. The view from the left. *J. Funct. Program.*, 14:69–111, January 2004.

[8] Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008.

[9] Nicolas Pouillard. Nameless, painless (Agda code), 2011. http://tiny.nicolaspouillard.fr/NaPa.agda.

[10] Nicolas Pouillard and François Pottier. A fresh look at programming with names and binders. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 217–228, New York, NY, USA, September 2010. ACM.

[11] John C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing 83*, pages 513–523. Elsevier Science, 1983.

[12] Philip Wadler. Theorems for free! In *Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pages 347–359, September 1989.