A unified treatment of syntax with binders

Nicolas Pouillard, François Pottier

JFP **TODO** (): 1–90, 2012.
doi:TODO Printed in the United Kingdom

A unified treatment of syntax with binders

Nicolas POUILLARD and François POTTIER
INRIA
(*e-mail:* np@nicolaspouillard.fr and francois.pottier@inria.fr)

Abstract

Atoms and de Bruijn indices are two well-known representation techniques for data structures that involve names and binders. However, using either technique, it is all too easy to make a programming error that causes one name to be used where another was intended.

We propose an abstract interface to names and binders that rules out many of these errors. This interface is implemented as a library in Agda. It allows defining and manipulating term representations in nominal style and in de Bruijn style. The programmer is not forced to choose between these styles: on the contrary, the library allows using both styles in the same program, if desired.

Whereas indexing the types of names and terms with a natural number is a well-known technique to better control the use of de Bruijn indices, we index types with worlds. Worlds are at the same time more precise and more abstract than natural numbers. Via logical relations and parametricity, we are able to demonstrate in what sense our library is safe, and to obtain theorems for free about world-polymorphic functions. For instance, we prove that a world-polymorphic term transformation function must commute with any renaming of the free variables. The proof is entirely carried out in Agda.

1 Introduction

Many programmers have to deal with the mundane business of building and transforming data structures that contain names and binders. Compilers, code generators, static analysers, theorem provers, and type-checkers have this in common. They manipulate programs, formulae, proofs, and types. One central difficulty is the representation of variables, that is, the representation of names and binders.

One traditional approach is to represent all occurrences of a variable identically. For this purpose, one typically uses character strings or integers. One can in fact use any data type that has an infinite number of elements and admits an equality test. The nature of its elements does not matter, which is why they are often known as *atoms*. While this seems to be the most obvious representation, it causes trouble when dealing with operations like substitution. Substitution must be capture-avoiding: variables must sometimes be renamed in order to avoid an accidental change of meaning of a name. The mathematical foundations of this technique, which we refer to as the *nominal* approach, are described by Pitts (2006).

Another traditional approach is to use de Bruijn indices (de Bruijn, 1972). This representation is often referred to as "nameless" because variables are no longer identified by a name but by a notion of "distance" to the binding point. This approach solves part of the

problem by providing a canonical representation. However, due to its arithmetic flavor, the manipulation of de Bruijn indices remains an error-prone activity.

In the end, regardless of which representation is used, programs that work with names can be hard to understand and are easy to get wrong. To remedy this, many proposals have been made, often in the form of finer-grained type systems for these representations (Altenkirch, 1993; McBride & McKinna, 2004; Bellegarde & Hook, 1994; Bird & Paterson, 1999; Altenkirch & Reus, 1999; Shinwell *et al.*, 2003; Licata & Harper, 2009; Pottier, 2007). We continue in this direction by presenting a library whose abstract types allow safe and fine-grained programming with both named and nameless representations.

This paper combines the conference papers by Pouillard and Pottier (2010) and by Pouillard (2011a) and extends them significantly. In the first conference paper, we present an abstract interface to names and binders. This interface comes with *two* implementations, one of which is in nominal style, the other in de Bruijn style. Thus, client code can be written and type-checked independently of which representation technique is ultimately chosen. In the second conference paper, Pouillard drops the nominal implementation and focuses on the de Bruijn side. Furthermore, he exploits dependent types, whereas the first conference paper avoids them. These decisions allow him to greatly simplify the interface of the library, which now has just one implementation. In the present paper, we stick with a single interface and *a single implementation*, but we extend them in such a way that *the nominal representation and de Bruijn's representation* (as well as several others) *are simultaneously available* to the client.

The connection with our previous papers is described in greater depth in section 12. Here, let us just stress again our main contribution: whereas the first conference paper "unifies" the nominal representation and de Bruijn's representation by supporting one *or* the other, the present paper "unifies" them by supporting them *simultaneously*.

This paper is organized as follows. In section 2, we briefly recall a few facts about AGDA, the language in which we write both our programs and proofs about these programs. Then, we begin the presentation of our library. We decide to initially focus on the fragment of the library that supports programming in nominal style (sections 3 to 7). We informally recall several existing approaches to representing data structures with names and binders in a nominal style (section 3). Then, we explain our new approach, by presenting the interface of (the nominal fragment of) our library (section 4), its usage (section 5), and its implementation (section 6). In section 7, we use logical relations and parametricity in order to explain and prove in what sense our library is safe. At this point, we are ready to extend the library with support for de Bruijn indices. We follow an analogous path as in the nominal case. First, we informally recall several existing approaches to working with de Bruijn indices in a "safe" manner. Then, we extend the interface and implementation of our library (section 9). We explain how these extensions can be exploited by the library's users (section 10) and why they are sound (section 11). Let us stress again that this is not an alternative implementation of the library, but an extension of it. Finally, we review some of the related work (section 12) and discuss several directions for future work (section 13).

2 A brief introduction to AGDA

Throughout the paper, our definitions are presented in the syntax of AGDA. Naturally, we cannot possibly provide a self-contained introduction to AGDA in this paper. In the following, we present a few of AGDA's notations and conventions. We hope that this is enough for someone who is familiar with functional programming to understand our code.

To better grasp the features of AGDA, we recommend the following further reading. Norell's Ph.D. thesis (2007) describes the theoretical and practical aspects of the development of AGDA. Another introduction to AGDA, of intermediate size, can be found in the introduction of Pouillard's Ph.D. thesis (2012).

Types AGDA is a dependent type theory. This means, in particular, that it has very few built-in types. In fact, only two forms of types are built-in, namely function types and the type of all types. Everything else is a user-defined type. The language offers facilities for defining record types, inductive types, and co-inductive types.

An ordinary (non-dependent) function type is written `A → B`. A dependent function type is written `(x : A) → B` or `∀ (x : A) → B`. A function parameter can be implicit: in this case, the function type is written `∀{x : A} → B`. This allows an actual parameter to be omitted at a call site if its value can be inferred from the context. The distinction between `∀ (x : A) → B` and `∀{x : A} → B` is quite superficial: in principle, after reconstructing the value of every omitted actual parameter, one can put every program in a form where every function parameter is explicit. This means, in particular, that the logical relations (and the "free theorems") associated with the types `∀ (x : A) → B` and `∀{x : A} → B` are the same.

The syntax of function types offers shortcuts for introducing multiple arguments at once and for omitting a type annotation, as in `∀{A} {i j : A} x → C`.

The type `Set`, also known as $\texttt{Set}_0$, is the type of all "small" types, such as `List String`, $\mathbb{N}$, and `Maybe (Bool × ℕ)`. The type $\texttt{Set}_1$ is the type of `Set` and "others like it", such as `Set → Bool`, `ℕ → Set`, and `Set → Set`. There is in fact an infinite hierarchy of types of the form `Set ℓ`, where ℓ is a universe level, roughly, a natural integer.

Propositions There is no specific sort of propositions: instead, propositions inhabit `Set ℓ` for some ℓ. The unit type $\top$, a record type with no fields, represents the true proposition. The empty type $\bot$, an inductive type with no constructors, represents the false proposition. Negation `¬ A` is defined as `A → ⊥`.

Lexical conventions Whitespace is significant: `x≤y` is an identifier, whereas `x ≤ y` is an application. This makes it possible to name a variable after its type, deprived of any whitespace. For example, if one follows this convention, then a variable of type `x ≤ y` (that is, a proof of `x ≤ y`) is named `x≤y`.

AGDA offers infix and "mixfix" notation. For instance, the expression `x ≤ y` is syntactic sugar for an application of the function `_≤_` to the arguments `x` and `y`. Note how, in the name of the function, the underscore character "_" is used to indicate where the arguments should appear.

Note that the same name can be used for different data constructors of distinct data types. AGDA makes use of type annotations to resolve ambiguities.

AGDA programs can be constructed incrementally. To do so, one makes use of "holes", which represent the unfinished parts of a program. Whatever appears inside a hole is not type-checked. The hole itself can receive any type. A hole is delimited by curly braces and exclamation marks, `{!like this!}`. In the following, we use holes when we want to omit a definition or a proof.

A comment begins with `--` and extends to the end of the line.

Notions of equality In type theory, there are several notions of equality. *Definitional equality* is the most basic notion. It is deeply rooted in the computation rules of the programming language, here, AGDA. Two terms are *definitionally equal* if and only if they reduce to a common term. In AGDA, reduction combines β-reduction, η-conversion, and expansion of the equations that appear as part of definitions.

For instance, `(λ x → x) suc zero` is definitionally equal to `suc zero`, and `suc` is definitionally equal to `λ x → suc x`. The term `zero + n` is definitionally equal to `n`, because the definition of addition is by case analysis upon its first argument. For the same reason, `n + zero` is not definitionally equal to `n`.

Definitional equality is decidable but weak. When one wishes for a notion of equality that relies not just on computation, but also on reasoning, on turns to *propositional equality*. In AGDA and in this document, this equality is written $\equiv$. Its negation is written $\not\equiv$. Propositional equality is defined as an inductive type, in the following style:

```
data _≡₀_ {A : Set₀} (x : A) : A → Set₀ where
  refl : x ≡₀ x
```

For simplicity, the above definition defines $\_\equiv_0\_$, a version of propositional equality that is restricted to "small" types of type $\mathtt{Set}_0$. We cannot fully explain the subtleties of this definition. Let us only point out that this inductive type has a single constructor, which requires both sides of the equality to be the same, that is, to be definitionally equal.

Propositional equality subsumes definitional equality. For example, the proposition `zero + n ≡₀ n` is true, because the terms `zero + n` and `n` are definitionally equal and (as a result) the term `refl` is an inhabitant of the type `zero + n ≡₀ n`. Propositional equality is strictly more powerful than definitional equality. For instance, the proposition `∀ n → n + zero ≡₀ n` can be proved by a simple induction on `n`.

Although propositional equality is stronger than definitional equality, it is still an *intensional* notion of equality. It is sometimes the case that it cannot relate two functions even though they are *pointwise equal*, that is, they map every input to the same output. Thus, it makes sense to introduce an *extensional* notion of equality, namely pointwise equality, written $\stackrel{\circ}{=}$. The definition presented here is specialized to small types, non-dependent functions, and the equality remains *intensional* on the codomain of the functions:

```
_≗₀_ : {A B : Set₀} (f g : A → B) → Set₀
f ≗₀ g = ∀ x → f x ≡₀ g x
```

It is possible to show, for instance, that the functions `λ x → not (not x)` and `λ x → x` are pointwise equal. This amounts to proving `∀ x → not (not x) ≡ x`; the proof is by cases on `x`. It is not possible to show that these functions are propositionally equal.

The logical relation which we introduce in section 7 can be viewed as a generalized version of pointwise equality. It is extensional: it relates all functions that map related inputs to related outputs.

Everything is online We use some definitions from AGDA's standard library, including natural numbers, booleans, lists, and applicative functors (`pure` and `_⊛_`).

For the sake of conciseness, the code fragments presented in this document are sometimes not perfectly self-contained. For instance, we sometimes use the operation $\circ'$, a variant of the standard composition operation $\circ$, which is equipped with a simpler (non-dependent) type and facilitates type reconstruction. We choose to gloss over these details. The complete code is available online (Pouillard, 2011b).

3 Introduction to the nominal approach

In order to explain what problem we are trying to solve, let us informally review how people traditionally encode abstract syntax as an algebraic data type and why these encodings are usually not satisfactory.

3.1 Warm-up: the bare nominal approach

The bare approach to syntax in nominal style requires very little infrastructure to begin with. Names and binders are represented by so-called atoms. The most important operation that atoms offer is an equality test. The set of atoms is countably infinite, and there must be a way of obtaining "fresh" atoms when desired. (In the following informal discussion, we elude this aspect and simply assume that we have a number of constants of type `Atom` at hand.) Atoms are usually represented as natural numbers.

```
-- A set of atoms (could be ℕ)
Atom : Set

-- Atom is countably infinite; here are some atoms:
-- x,y,z... could be represented by 0,1,2...
x y z f g ... : Atom

-- The equality test on atoms
_==ᴬ_ : (x y : Atom) → Bool
```

Using the type `Atom` , we can readily define algebraic data types for abstract syntax with names and binders. Our running example is the untyped λ-calculus; its definition appears below. The type `Tmᴬ` (`Tm` for "term" and `ᴬ` for "atom") has four data constructors. The constructor `V` is for variables and carries just an atom. The constructor `_·_` represents function application and carries two subterms. The constructor `ƛ` carries an atom and

a subterm in which this atom is informally considered bound. Thus, the atom carried by ƛ is informally considered a binder, whereas the atom carried by V is a (free or bound) occurrence of a name: it refers to some binder. The constructor Let carries an atom and two subterms. The atom is informally considered bound in the second subterm only.

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

It is striking that there is no formal distinction between the atoms that represent binders and those that represent occurrences. Neither is there any indication of the scope of the binders. This calls for improvement.

Here are two examples of terms that represent object-level syntax. They represent the identity function and the application function of the λ-calculus.

```
-- λx. x
idTm^A : Tm^A
idTm^A = ƛ x (V x)

-- λf. λx. f x
apTm^A : Tm^A
apTm^A = ƛ f (ƛ x (V f · V x))
```

The strength of the "bare nominal" approach resides in its simplicity. The representation of names and binders is the same as in the concrete syntax. One important issue, though, is α-equivalence: there are multiple equivalent representations of the "same" term. Another issue is capture: it is easy to inadvertently change the meaning of a name by placing it in a context that happens to bind this name.

The need for a notion of α-equivalence arises out of the fact that choices of atoms are arbitrary. One would like to consider that two terms represent the "same" piece of syntax when they differ only by a consistent renaming of bound names. Two such terms are said to be α-equivalent. For example, here are two α-equivalent terms:

```
t^x : Tm^A
t^x = ƛ x (V f · V x)
t^y : Tm^A
t^y = ƛ y (V f · V y)
```

The term t^x is α-equivalent to t^y because consistently replacing the bound name x with the name y in the term t^x yields the term t^y. Conversely, consistently replacing y with x

in $\mathtt{t}^{\mathtt{y}}$ yields $\mathtt{t}^{\mathtt{x}}$. However, α-equivalence can be a subtle notion. The term $\mathtt{t}^{\mathtt{f}}$ below is *not* α-equivalent to $\mathtt{t}^{\mathtt{x}}$ and $\mathtt{t}^{\mathtt{y}}$:

```
t^f : Tm^A
t^f = ƛ f (V f · V f)
```

Although replacing the bound name `x` with the name `f` in the term $\mathtt{t}^{\mathtt{x}}$ does yield $\mathtt{t}^{\mathtt{f}}$, this would be an *inconsistent* renaming. The name `f` that occurs in the renaming would be captured by the binder `ƛ f` that occurs in the term $\mathtt{t}^{\mathtt{x}}$. Conversely, if one attempts to consistently replace `f` with `x` in $\mathtt{t}^{\mathtt{f}}$, one finds that such a replacement is permitted, but does not yield $\mathtt{t}^{\mathtt{x}}$.

Without delving any further into α-equivalence, let us emphasize the point of view that we adopt in this paper. Choices of bound names are purely a representation issue, so they should not be able to influence computation in an observable way. The flip-side of this slogan is, "good" computations should not observably depend on choices of bound names. This can be stated as follows:

> *A function is well-behaved if, when applied to α-equivalent arguments, it produces α-equivalent results.*

Let us give a few examples of well-behaved functions. The function $\mathtt{rm}^{\mathtt{A}}$ removes all occurrences of an atom from a list of atoms. It operates only on free names and thus does not depend on choices of bound names, hence is well-behaved. The function `fv`, which constructs a list of the free variables/atoms of a term is well-behaved.

```
rm^A : Atom → List Atom → List Atom
rm^A _ []       = []
rm^A x (y :: ys) =
  if x ==^A y then rm^A x ys
              else y :: rm^A x ys

-- rm^A behaves well; for instance, this holds:
test-rm^A : rm^A x [ x ] ≡ rm^A y [ y ]
test-rm^A = refl -- both sides reduce to []

fv : Tm^A → List Atom
fv (V x)       = [ x ]
fv (t · u)     = fv t ++ fv u
fv (ƛ x t)     = rm^A x (fv t)
fv (Let x t u) = fv t ++ rm^A x (fv u)

-- fv behaves well; for instance, this holds:
test-fv : fv t^x ≡ fv t^y
test-fv = refl -- both sides reduce to [ f ]
```

We now illustrate misbehaving functions with two examples. The function `ba` computes the list of "bound atoms" of a term, that is, the list of atoms that appear as binders in this term. The function `cmp-ba` accepts two terms and, if they are `ϰ`-abstractions, compares the atoms bound by `ϰ`. The codomains of these functions are respectively `List Atom` and `Bool`. At these types, which do not contain any binders, α-equivalence is just equality.

```
ba : Tm^A → List Atom
ba (ϰ x t)      = x :: ba t
ba (Let x t u) = x :: ba t :: ba u
ba (t · u)      = ba t ++ ba u
ba (V _)        = []

[x]≠[y] : [ x ] ≠ [ y ]
[x]≠[y] = {! omitted !}

-- ba does not behave well:
test-ba : ba t^x ≠ ba t^y
test-ba = [x]≠[y]

cmp-ba : Tm^A → Tm^A → Bool
cmp-ba (ϰ x _) (ϰ y _) = x ==^A y
cmp-ba _       _       = false

-- cmp-ba does not behave well:
test-cmp-ba : cmp-ba t^x t^x ≠ cmp-ba t^x t^y
test-cmp-ba () -- true ≠ false
```

In traditional informal proofs, α-equivalence is identified with equality. This means that every function must map α-equivalent inputs to α-equivalent outputs, or it does not deserve to be called a "function". Thus, whenever one defines a function, one must prove that it is well-behaved.

In this paper, we wish to exploit the type system in order to meet this proof obligation. Using logical relations and parametricity, we will be able to prove, once and for all, that well-typed functions are well-behaved (with respect to a notion of α-equivalence which is itself type-directed). For an appropriate definition of the type of λ-terms, we will find that the function `cmp-ba` cannot be typed at all, and that the function `ba` can be typed only at a type that makes it harmless.

3.2 Using well-formedness judgements

One way to define the scoping rules of the object language is to define a well-formedness judgement. This can be done using an inductive predicate over the structure of terms. It is a well-known technique used in the definition of type systems. The standard presentation makes use of a set of inference rules where judgements are made of an environment, a term

and a type. The environment tracks the type of each introduced variable. To specify just the scoping rules, one follows the same presentation, without types. We directly focus on a formal presentation in AGDA since inference rules (and grammars) have a direct translation into inductive types. First, one defines environments, which are just lists of atoms:

```
data Env : Set where ε   : Env
                     _,_ : (Γ : Env) (x : Atom) → Env
```

Then comes the definition of the membership predicate. It states when an atom is a member of some environment. The definition consists of two rules. One rule applies when the atom is present at the first position in the environment. The other rule applies when it is present in the tail of the environment. We use AGDA comments to give each inference rule a traditional visual appearance:

```
data _∈_ x : (Γ : Env) → Set where
  here  :              ------------
                         x ∈ (Γ , x)

  there : ∀ {y}      →  x ∈ Γ
                     → ------------
                         x ∈ (Γ , y)
```

Finally, one can give the scoping rules of the λ-calculus. The definition boils down to using an environment to track bound atoms, using the membership predicate at variable occurrences and pushing the binders onto the environment:

```
data _⊢_ Γ : Tm^A → Set where
  V : ∀ {x}          →  x ∈ Γ
                     → ---------
                         Γ ⊢ V x

  _·_ : ∀ {t u}      →  Γ ⊢ t
                     →  Γ ⊢ u
                     → -----------
                         Γ ⊢ t · u

  ƛ : ∀ {t b}        →  Γ , b ⊢ t
                     → -----------
                         Γ ⊢ ƛ b t

  Let : ∀ {t u b}    →  Γ ⊢ t
                     →  Γ , b ⊢ u
                     → ---------------
                         Γ ⊢ Let b t u
```

We can now state that our example terms are well-scoped in the empty environment. Thanks to implicit arguments and to the definition of the membership predicate, the proof that a term is well-scoped is the term itself, expressed in de Bruijn notation (`here` and `there` respectively play the role of zero and successor).

```
⊢id : ε ⊢ idTmᴬ
⊢id = ƛ (V here)

⊢ap : ε ⊢ apTmᴬ
⊢ap = ƛ (ƛ (V (there here) · V here))
```

However, this technique is not exactly what we are looking for. Indeed, here, the scoping properties have to be explicitly stated on the side of each definition. We are looking for something more integrated into the types. This would enable well-formedness to be enforced in a more pervasive and automatic manner. Fortunately, this technique can be adapted so as to merge the scoping information and the term. We study this idea next.

3.3 Well-scoped terms

We now merge the inductive definition of terms and the inductive definition of the scoping predicate. Thus, we end up with an inductive type of terms that is indexed by an environment. This environment is extended at binders and queried at variable occurrences:

```
data Tmᴶ Γ : Set where
  V   : ∀ {x} → x ∈ Γ → Tmᴶ Γ
  _·_ : Tmᴶ Γ → Tmᴶ Γ → Tmᴶ Γ
  ƛ   : ∀ b → Tmᴶ (Γ , b) → Tmᴶ Γ
  Let : ∀ b → Tmᴶ Γ → Tmᴶ (Γ , b) → Tmᴶ Γ
```

The idea of encoding well-scopedness (and well-typedness as well) as part of the inductive definition of terms goes back at least as far as Pfenning and Lee (1989). It also appears in the original LF paper (Harper *et al.*, 1993) and in the nested data type approach (Bellegarde & Hook, 1994; Bird & Paterson, 1999; Altenkirch & Reus, 1999). Hence, it is by now widely known. An instance of it in the recent literature is Chlipala's type-preserving compiler (2007).

One can define our two example terms in this style as well:

```
idTmᴶ : Tmᴶ ε
idTmᴶ = ƛ x (V here)

apTmᴶ : Tmᴶ ε
apTmᴶ = ƛ f (ƛ x (V (there here) · V here))
```

The function `fv` can easily be adapted to this style, while the function $\texttt{rm}^{\texttt{A}}$ can be reused:

```
fv : ∀ {Γ} → Tmᴶ Γ → List Atom
fv (V {x} _)   = [ x ]
fv (t · u)     = fv t ++ fv u
fv (ƛ x t)     = rmᴬ x (fv t)
fv (Let x t u) = fv t ++ rmᴬ x (fv u)
```

Alas, the functions `ba` and `cmp-ba` are not rejected by this style. Their types becomes:

```
ba : ∀ {Γ} → Tmᴶ Γ → List Atom
cmp-ba : ∀ {Γ₁ Γ₂} → Tmᴶ Γ₁ → Tmᴶ Γ₂ → Bool
```

Thus, although we have been able to formally describe the scoping rules, and to ensure that all terms are well-scoped by construction, we have not made much progress towards our goal. Indeed, two different but α-equivalent terms can still be distinguished.

Worse, by introducing environments and membership proofs as explicit objects, we have introduced new issues. A function that receives a term as an argument now also receives (or otherwise has access to) the environment in which this term is well-scoped. This extra information can be used by the function. For instance, here are two functions that examine the environment by pattern-matching:

```
fast-fv : ∀ {Γ} → Tmᴶ Γ → List Atom
fast-fv {ε} _ = []  -- for sure the term is closed
fast-fv     t = fv t

env-length : Env → ℕ
env-length ε       = 0
env-length (Γ , _) = 1 + env-length Γ

env-length-Tmᴶ : ∀ {Γ} → Tmᴶ Γ → ℕ
env-length-Tmᴶ {Γ} _ = env-length Γ
```

The issue is that Γ is a concrete input to the functions `fv`, `ba`, `cmp-ba`, `fast-fv`, and $\texttt{env-length-Tm}^{\texttt{J}}$. It is not erased: it is an ordinary argument, which can be examined by the function. The fact that it is an *implicit* argument offers a notational convenience, nothing more.

This is not satisfactory. We would like to think of Γ as an "index", that is, a piece of information that the type-checker keeps track of in order to reject ill-behaved code and that can be *erased* prior to running the program. Here, this is impossible: for instance, the value of $\texttt{env-length-Tm}^{\texttt{J}}$ `t` depends not just on the λ-term that `t` represents but also on the environment Γ in which `t` is considered.

In order to address these issues, we make use of abstract types. We replace environments, which have the concrete type `Env` above, with *worlds*, where the type `World` of worlds

remains abstract. Thus, worlds cannot be examined by client code. We set everything up so that worlds need not exist at runtime: that is, they can be erased prior to execution. However, we do not formally prove that worlds can be erased: we leave this issue for future work (see section 13). We replace the type `Atom` with two distinct types. An atom that is used in a binding position receives the abstract type `Binder`. We do not equip this type with an equality test, so the function `cmp-ba` is ruled out. An atom that serves as a (free or bound) occurrence is glued together with a proof of its membership in some world α, and this pair receives the abstract type `Name` α. Because these types are abstract, we must equip them with a set of operations, otherwise they would be completely unusable. We select these operations so as to be able to prove, eventually, that well-typed functions are well-behaved. For instance, it is sound to equip the type `Name` α with an equality test. The framework of logical relations, which we develop in section 7, serves as a guide in the choice of these operations. For each operation *independently*, it allows us to automatically construct the statement of a lemma that must be proved to justify that this operation is safe.

4 The NOMPA interface (nominal fragment)

Our library is called NOMPA. To the client, it offers an interface whose types are abstract. Its implementation is hidden from the client. This allows us to exploit parametricity and to prove that well-typed client code is well-behaved.

As announced earlier, in sections 4 to 7, we focus on a fragment of the library, which supports programming in nominal style. The rest of the library, which supports programming in de Bruijn style as well as in combinations of the two styles, is presented in section 8. Thus, the listing that appears in Figure 1 represents just the nominal fragment of the library.

We now present each ingredient in turn. First, here are the building blocks needed to define algebraic data types with names and binders in a nominal style.

4.1 Everything we need to define nominal syntax

In order to replace environments in the definition of well-scoped terms, we introduce an abstract notion of worlds. Hence, the interface begins with an abstract type of worlds:

```
World : Set
```

We find it necessary to distinguish atoms that are used in a binding position and atoms that are used as (free or bound) occurrences. We refer to the former as *binders* and to the latter as *names*. To that end, we provide abstract types of binders and names.

A binder is an atom that is meant to be used in a binding position. Internally, it is just an atom, but this fact is not exposed. Although, by traversing a term, one can gain access to all of the binders that appear in it, this does not imply that one can distinguish two α-equivalent terms. Indeed, the interface does not provide any means of distinguishing two binders: there is no equality test at type `Binder`.

```
Binder : Set
```

```
record NomPa : Set₁ where
  constructor mk

  infixr 5 _◁_
  infix 3 _⊆_
  infix 2 _#_

  field
    -- Abstract types for worlds, names, and binders
    World  : Set
    Name   : World → Set
    Binder : Set

  _→ᴺ_ : (α β : World) → Set
  α →ᴺ β = Name α → Name β

  field
    -- Constructing worlds
    ∅   : World
    _◁_ : Binder → World → World

    -- An infinite set of binders
    zeroᴮ : Binder
    sucᴮ  : Binder → Binder

    -- Converting back and forth between names and binders
    nameᴮ   : ∀ {α} b → Name (b ◁ α)

    -- There is no name in the empty world
    ¬Name∅  : ¬ (Name ∅)

    -- Two names can be compared; a binder and a name can be compared
    _==ᴺ_   : ∀ {α} (x y : Name α) → Bool
    exportᴺ : ∀ {α b} → Name (b ◁ α) → Name (b ◁ ∅) ⊎ Name α

    -- The fresh-for relation
    _#_  : Binder → World → Set
    _#∅  : ∀ b → b # ∅
    suc# : ∀ {α b} → b # α → (sucᴮ b) # (b ◁ α)

    -- World inclusion
    _⊆_     : World → World → Set
    coerceᴺ : ∀ {α β} → (α ⊆ β) → (α →ᴺ β)
    ⊆-refl  : Reflexive _⊆_
    ⊆-trans : Transitive _⊆_
    ⊆-∅     : ∀ {α} → ∅ ⊆ α
    ⊆-◁     : ∀ {α β} b → α ⊆ β → (b ◁ α) ⊆ (b ◁ β)
    ⊆-#     : ∀ {α b} → b # α → α ⊆ (b ◁ α)
```

Figure 1. The NOMPA interface (nominal fragment)

The type of names, `Name`, is indexed by a world. A world α can be thought of informally as a set of atoms. A name of type `Name α` can be thought of as a pair of an atom and a proof that this atom is a member of the set α. In other words, a name has type `Name α` if it "makes sense" in the world α.

```
Name : World → Set
```

Throughout the paper, we often use "name transformers", that is, functions from type `Name α` to type `Name β`. We write $\alpha \to^{\mathrm{N}} \beta$ as a shorthand for this type.

The intuitive view of worlds as sets of atoms can be referred to as the *unary view* of worlds. In section 7.3, where we study logical relations, we introduce a *binary view* of worlds, whereby worlds are viewed as certain relations between sets of atoms. Although the unary view represents a good intuition (and we stick with it for the moment), the binary view is required in order to understand why (and prove that) well-typed functions are well-behaved.

We now introduce a way of extending a world with a binder. This is analogous to the constructor `_,_` of section 3.2 for extending an environment with an atom.

```
_◁_ : Binder → World → World
```

If a world is thought of as a set of atoms, then the set `b ◁ α` is just the union of the singleton set {b} and of the set α. There is no requirement that these sets be disjoint: the world `b ◁ α` is defined even if `b` is already a member of α. This reflects the fact that, in the nominal representation, it is permitted for two binders to bind the same name, even if one of them is nested in the other. In that case, one traditionally considers that the most recent binder "hides", or "shadows", the previous binder. For this reason, we pronounce `b ◁ α` as "`b` hides α".

At this point, we have everything that is needed to build data types with names and binders. The new definition of `Tm` is very close to the previous one. All we have to do is use `_◁_` instead of `_,_`, rename Γ to α, and use `Name` instead of a pair of an atom and a membership proof:

```
data Tm α : Set where
  V   : Name α → Tm α
  _·_ : Tm α → Tm α → Tm α
  ƛ   : ∀ b → Tm (b ◁ α) → Tm α
  Let : ∀ b → Tm α → Tm (b ◁ α) → Tm α

_→Tm_ : (α β : World) → Set
α →Tm β = Tm α → Tm β
```

The description of the binding constructs `ƛ` and `Let` relies on dependent types: the binder `b` is used in the type of the subterm.

Here is a trivial example of a function that traverses a term and measures its size. It is remarkable for its simplicity: name abstractions are traversed without fuss. The code

would be exactly the same in the bare nominal approach of section 3.1. It is efficient: no renaming is involved. In comparison, in FreshML (Shinwell *et al.*, 2003) or in the locally nameless approach (Aydemir *et al.*, 2008; Charguéraud, 2011), crossing a binder can have a cost that is linear in the size of the subterm. Polymorphic recursion is exploited: one the line that deals with `ϰ`, the recursive call to `size t` is at some inner world.

```
size : ∀ {α} → Tm α → ℕ
size (V _)       = 1
size (t · u)     = 1 + size t + size u
size (ϰ _ t)     = 1 + size t
size (Let _ t u) = 1 + size t + size u
```

4.2 Building binders and names

In order to build terms, we need binders and names. Binders are introduced via two primitive operations called `zero`$^{\mathtt{B}}$ and `suc`$^{\mathtt{B}}$. We then lift any natural number to a binder with a cheap convenience function:

```
zero^B : Binder
suc^B  : Binder → Binder

_^B : ℕ → Binder
zero   ^B = zero^B
(suc n) ^B = suc^B (n ^B)
```

In effect, binders are natural numbers with a restricted interface. Here, we give only zero and successor, but all of the arithmetic operations on natural numbers could be exposed as well. One key limitation, however, is that no information must be allowed to "leak" out of a binder. Hence, no equality test over binders is provided. The reason why this must be so will appear more clearly when we build the logical relation in section 7.3.

In order to build names, we provide a function, called `name`$^{\mathtt{B}}$, which turns a binder into a name.

```
name^B : ∀ {α} b → Name (b ◁ α)
```

While `name`$^{\mathtt{B}}$ can turn an arbitrary binder `b` into a name, it imposes that the resulting name inhabit a world of the form `b ◁ α`. Even though α can be instantiated at will, this is an important limitation. We will soon introduce a notion of world inclusion that allows overcoming this limitation (section 4.3). For the moment, we have enough building blocks to define a representation of the identity function!

```
idTm : ∀ {α} → Tm α
idTm = ϰ x (V (name^B x))   where x = 0 ^B
```

As another example, here is a representation of the λ-term that represents "false" in Church's encoding, that is, $\lambda x.\lambda x.x$. We use the binder `x` twice on purpose to show that shadowing is permitted. To type-check the variable occurrence of `x`, the world $x \triangleleft \alpha$ is implicitly passed to $\mathtt{name}^{\mathtt{B}}$, which returns a name in the world $x \triangleleft x \triangleleft \alpha$. Then, each of the two "ƛ x"'s takes away one "$x \triangleleft$", so the final term inhabits the world α.

```
falseTm : ∀ {α} → Tm α
falseTm = ƛ x (ƛ x (V (name^B x)))
   where x = 0 ^B
```

However, one faces an issue when one attempts to build a representation of the Church encoding of "true", that is, $\lambda x.\lambda y.x$. The naïve approach does not type-check:

```
-- this does not type-check
trueTm : ∀ {α} → Tm α
trueTm = ƛ x (ƛ y (V (name^B x)))
   where x = 0 ^B
         y = 1 ^B
```

While $\mathtt{name}^{\mathtt{B}}\ x$ inhabits any world of the form $x \triangleleft \beta$, the context expects a name in the world $y \triangleleft x \triangleleft \alpha$. We introduce means of moving a name from one world to another in section 4.3.

Closed terms The above well-typed terms (`idTm` and `falseTm`) have no free variables. They are said to be closed. They both have type $\forall\ \{\alpha\} \to \mathtt{Tm}\ \alpha$. This world-polymorphic type reflects the closedness property. Indeed, if any world will do, then the empty world, written $\emptyset$, will do as well. Thus, `idTm` also has type $\mathtt{Tm}\ \emptyset$. Conversely, because the empty world is included in every world, we will be able to move any term from the empty world to an arbitrary world. In short, $\mathtt{Tm}\ \emptyset$ and $\forall\ \{\alpha\} \to \mathtt{Tm}\ \alpha$ are *isomorphic* types.

To allow exploiting the fact that a term that inhabits the empty world must be closed, we introduce $\neg\mathtt{Name}\emptyset$, which witnesses that there is no name in the empty world:

```
∅ : World
¬Name∅ : ¬(Name ∅)
```

In various situations, this operation enables arguing that "this case is impossible". For instance, when writing a function that looks up a name in an environment represented as a list, one typically uses $\neg\mathtt{Name}\emptyset$ in the case where the environment is the empty list. This amounts to arguing that "because the name we are looking for is properly bound, the search cannot fall off the end of the environment".

4.3 World inclusion

It is often necessary to widen the world which a name inhabits. Our earlier attempt to define `trueTm` illustrates this: for this definition to type-check, we need to widen the world

of the name `x`. Widening a world causes a loss of static information. For instance, we have seen that a term in the empty world must be closed. If one widens its world, then this term is no longer statically known to be closed. We call this operation "weakening" because it causes the loss of some static information. The name "weakening" is also vastly used for the same purpose when referring to typing environments.

To account for the multiple ways in which we could widen a world, we introduce a world inclusion predicate. In AGDA terminology, we introduce a type `_⊆_` for witnesses of world inclusion. If a world α is included in a world β then it is permitted to transport a name (and a term as well, as we shall see) from α to β. The primitive operation `coerceᴺ` serves this purpose. It takes an inclusion witness, a name, and returns the same name at a wider world.

```
_⊆_     : World → World → Set
coerceᴺ : ∀ {α β} → α ⊆ β → (α →ᴺ β)
```

We also introduce an alias for `coerceᴺ`, called `_⟨-because_-⟩`. It helps keep the code separate from the type-checking argument: the proof of world inclusion, which appears between the angle brackets, can safely be skipped by the reader.

```
infix 0 _⟨-because_-⟩
_⟨-because_-⟩ : ∀ {α β} → Name α → α ⊆ β → Name β
_⟨-because_-⟩ n pf = coerceᴺ pf n
 -- We can now write:          x ⟨-because some proof -⟩
 -- Which is less noisy than: coerceᴺ (some proof) x
```

World inclusion rules A set of world inclusion rules is given in figure 1. World inclusion is reflexive and transitive (`⊆-refl` and `⊆-trans`). The empty world is a least element for world inclusion (`⊆-∅`). For every binder `b`, the operation `b ◃ _` is covariant, which means that it preserves world inclusion (`⊆-◃`).

The last world inclusion rule, `⊆-#`, states that a world α is included into `b ◃ α` under the condition that `b` is not a member of α. This condition is required for soundness. At this point, we expect that the reader might be surprised, because an interpretation of worlds as sets suggests that this condition is unnecessary: indeed, if α is a set, then it is unconditionally a subset of $\{\ \mathtt{b}\ \} \cup \alpha$. This shows the limitations of this way of thinking. When we introduce logical relations in section 7.3, we show that interpreting worlds as *relations* provides a richer viewpoint and explains why this condition is necessary. Without waiting until then, the following code snippet demonstrates that in the absence of this condition one could write ill-behaved programs:

```
wrong : Binder → Binder → Bool
wrong x y = nameᴮ x ==ᴺ nameᴮ y ⟨-because wrongProof -⟩
    where postulate wrongProof : y ◃ ∅ ⊆ x ◃ y ◃ ∅
```

As this code snippet shows, in the absence of this condition, it would be permitted to compare names in different worlds, hence to compare binders, and, finally, to distinguish two α-equivalent terms.

The "fresh-for" relation The last inclusion rule, `⊆-#`, uses a "fresh-for" relation, written `_#_`. As suggested above, the proposition `b # α` must guarantee at least that the name `b` is not a member of the world α, interpreted as a set. In fact, we choose to equip the "fresh-for" relation with a strictly stronger meaning. Taking advantage of the fact that atoms are integers internally, we take `b # α` to mean that `b` dominates α, that is, `b` is strictly greater than every name that inhabits α.

We introduce two rules to produce witnesses of the relation `_#_`. These rules appear in figure 1. The first rule, `_#∅`, states that every binder is fresh for the empty world. The second rule, `suc#`, is the reason why we equip `_#_` with a stronger meaning. It states that, if `b` dominates α, then the successor of `b` dominates `b ◁ α`. This gives us a simple and efficient way of building "fresh-for" witnesses. We note that, although this axiomatization of the "fresh-for" relation is not complete, it has proved sufficient for our purposes.

Emptiness of worlds The inclusion relation can express emptiness. A world α is empty if and only if α is included in $\emptyset$. We favor this definition of emptiness, as opposed to an intensional equality with the empty world, because it is more flexible. In section 9, we introduce a new operation on worlds, called `_+1`. We will see that the world `∅ +1` is empty, in the sense that it is included in the empty world, yet it does not reduce to $\emptyset$.

By combining `coerceᴺ` and `¬Name∅`, we obtain a proof that, if the world α is empty, then there is no name in the world α. This formulation reduces the goal of obtaining a contradiction to a world inclusion goal. This is beneficial if there is automation for constructing inclusion proofs.

```
¬Name : ∀ {α} → α ⊆ ∅ → ¬(Name α)
¬Name α⊆∅ = ¬Name∅ ∘ coerceᴺ α⊆∅
```

One can also introduce a version of this operation whose codomain is an arbitrary type `A` instead of the empty type:

```
Name-elim : {A : Set} {α} → α ⊆ ∅ → Name α → A
Name-elim pf x = ⊥-elim (¬Name pf x)
```

Here is a prototypical example of a function where we can avoid the case for variables:

```
ex-Name-elim : Tm ∅ → {! ... !}
ex-Name-elim (t · u)     = {! ... !}
ex-Name-elim (ƛ b t)     = {! ... !}
ex-Name-elim (Let b t u) = {! ... !}
 -- This last case is discarded by Name-elim
ex-Name-elim (V x)       = Name-elim ⊆-refl x
```

Relational reasoning Sometimes, one must build complex inclusion witnesses. While inference would be of great effect here, we leave it for future work. For the time being, we propose a modest syntactic tool to help build inclusion witnesses. The `⊆-Reasoning` module gives access to the transitivity rule `⊆-trans` in a style which focuses on the intermediate states of the reasoning, as opposed to the reasoning steps. The syntax is a list of worlds interspersed with inclusion witnesses. After the last world, a box ■ ends the proof. We present an example of the use of this notation when we define `trueTm` in the next paragraph. The code for `⊆-Reasoning` is a trick commonly used in the AGDA standard library (Danielsson, 2011) and is given below for reference, but can safely be skipped.

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

  _⊆⟨_⟩_ : ∀ α {β γ} → α ⊆ β → β ⊆ γ → α ⊆ γ
  _⊆⟨_⟩_ _ = ⊆-trans

  _■ : ∀ α → α ⊆ α
  _■ _ = ⊆-refl
```

Building any term We can now finally build all nominal terms (up to α-equivalence). In section 5.5, we prove that we can build all λ-terms by defining a function that converts a "bare nominal" λ-term in the style of section 3.1 to a term of type `Tm`. Pouillard (2012) shows that our system can encode not just the type of λ-terms, but more generally an arbitrary nominal signature in the sense of Pitts (2006). For the time being, we focus on the term `trueTm`, and show that we can now type-check it.

```
trueTm : ∀ {α} → Tm α
trueTm {α} = ƛ x (ƛ y (V xᴺ))
  where x   = 0 ᴮ
        y   = 1 ᴮ
        xᴺ  = nameᴮ x ⟨-because x ◃ ∅       ⊆⟨ ⊆-# (suc# (x #∅)) ⟩
                              y ◃ x ◃ ∅   ⊆⟨ ⊆-◃ y (⊆-◃ x ⊆-∅) ⟩
                              y ◃ x ◃ α   ■ -⟩
```

The proof that is required in order to "move `x` into the correct world" is involved in comparison with the simplicity of the example. Here, by fixing the empty world instead of allowing world polymorphism, we could cut the proof in half. To build larger examples, we define smart constructors which require only that one specify the distance to the binding site (Pouillard, 2011b).

4.4 Comparing and refining names

While two binders cannot be compared, our interface allows comparing two names that inhabit a common world. This may seem contradictory, since one can turn binders into names. In fact, binder comparison cannot be implemented in terms of name comparison because two arbitrary binders can be turned only into names in distinct worlds.

```
_==ᴺ_ : ∀ {α} (x y : Name α) → Bool
```

While world inclusion gives a means of weakening the type of a name, we also need a means of strengthening the type of a name, that is, of refining its world. Naturally, this requires a dynamic check. We choose to offer an operation that compares a binder `b` and a name `x` and refines the type of `x` according to the outcome of the comparison. This operation is known as `exportᴺ`.

Assume that `x` is in the scope of `b`, that is, the name `x` inhabits the world `b ◁ α`. The function `exportᴺ` tests whether `x` is equal to `b`. If so, `x` is refined to the world `b ◁ ∅`, which only `b` inhabits. Otherwise, it is refined to `α`. In short, given a name of type `Name (b ◁ α)`, the function `exportᴺ` returns the same name, with a refined type that tells whether this name stands on the `b` side or on the `α` side.

```
exportᴺ : ∀ {b α} → Name (b ◁ α) → Name (b ◁ ∅) ⊎ Name α
exportᴺ = maybe inj₂ (inj₁ (nameᴮ _)) ∘′ exportᴺ?
```

Actually, the primitive operation offered by the library is the function `exportᴺ?`, which returns a result of type `Maybe (Name α)`. The function `exportᴺ` is then built on top of `exportᴺ?`.

```
-- A →? B is the type of partial functions from A to B
A →? B = A → Maybe B

exportᴺ? : ∀ {b α} → Name (b ◁ α) →? Name α
```

The idea that a dynamic check yields refined static type information is quite old, and it is difficult to determine to whom it should be attributed. It is present, for instance, in Floyd and Hoare's rules for reasoning about programs (Floyd, 1967; Hoare, 1969). Nevertheless, it is worth noting that `exportᴺ?` is very much analogous to McBride's type-refining name comparison operation `thick` (McBride, 2003). The function `exportTm?`, which we define later on top of `exportᴺ?` (section 5.4.7), is analogous to McBride's type-refining occur-check, but is able to deal with terms that contain binders.

On top of `exportᴺ?`, we also build a convenient eliminator for names. It is simply the elimination of the result of `exportᴺ?`.

```
exportWith : ∀ {b α A} → A → (Name α → A) → Name (b ◁ α) → A
exportWith v f = maybe f v ∘′ exportᴺ?
```

5 Programming on top of NOMPA (nominal fragment)

5.1 Example: computing free variables

We now have enough tools to present a more interesting example, namely the function `fv`, which constructs a list of the free variables of a term. At variables and applications, the code is straightforward. At a name abstraction, one easily collects the free variables of the body via a recursive call. However, this yields a list of names that inhabit the inner world of the abstraction, that is, a value of type `List (Name (b ◁ α))`. This list cannot be returned, because the codomain of `fv` is declared to be `List (Name α)`. This is fortunate, since returning this list would let the bound name leak out of its scope! As before, we rely on an auxiliary function, `rm`, which removes all occurrences of a binder `b` in a list of names. A new feature of `rm` is that it now performs type refinement in the style of (and by using) $\texttt{export}^{\texttt{N}}$.

```
rm : ∀ {α} b → List (Name (b ◁ α)) → List (Name α)
rm b [] = []
rm b (x :: xs) with exportᴺ x -- b is implicit
... {- bound: x≡b -} | inj₁ _    = rm b xs
... {- free:  x≢b -} | inj₂ x′   = x′ :: rm b xs

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

The function `rm` applies $\texttt{export}^{\texttt{N}}$ `{b}` to every name `x` in the list and builds a list of only those `x`'s that successfully export to the world α. It exhibits a typical way of using $\texttt{export}^{\texttt{N}}$ to perform a comparison of a name against a binder together with a type refinement. This idiom is recurrent in the programs that we have written.

5.2 Example: working with environments

Here is another example, where we introduce the use of an environment.

```
occurs : ∀ {α} → Name α → Tm α → Bool
occurs x₀ = occ (λ y → x₀ ==ᴺ y)
  where
   OccEnv  : World → Set
   OccEnv  α = Name α → Bool
   extend  : ∀ {α b} → OccEnv α → OccEnv (b ◁ α)
   extend  = exportWith false

   occ : ∀ {α} → OccEnv α → Tm α → Bool
   occ Γ (V x)       = Γ x
   occ Γ (t · u)     = occ Γ t ∨ occ Γ u
   occ Γ (ƛ _ t)     = occ (extend Γ) t
   occ Γ (Let _ t u) = occ Γ t ∨ occ (extend Γ) u
```

The function `occurs` tests whether the name x_0 occurs free in a term. An environment Γ is carried down, augmented when a binder is crossed, and looked up at variable occurrences. This environment is represented as a function of type `Name α → Bool`.

The definition of `extend` states how to look up `x` in the environment `extend Γ`. (Recall that the function `extend` takes two implicit parameters, so `extend Γ` is synonymous with `extend {α} {b} Γ`.) To this end, one must first compare `x` and `b`. If `x` and `b` are equal, then this occurrence of `x` is not free, so `occ Γ (V x)` must return `false`. If they differ, one must look up `x` in Γ. This case analysis is concisely implemented by using the function `exportWith`, which was built on top of `export`$^{\mathtt{N}}$.

We believe that this code is written in a relatively natural and uncluttered style. There is no hidden cost: no renaming is required when a name abstraction is crossed.

The type system forces us to use names in a sound way. For instance, in the definition of `occ`, forgetting to extend the environment when crossing a binder (that is, writing Γ instead of `extend Γ`) would cause a type error. In the definition of `extend`, attempting to check whether `x` occurs in Γ without first comparing `x` and `b` would cause a type error. Recall that the definition of the type `Tm` allows a newer binding to shadow an earlier one. Our type discipline guarantees that the code works properly in the presence of shadowing.

Although representing an environment as a function is a simple and elegant representation, others exist. For instance, in the case of `occurs`, we could represent the environment as a list of binders: the code for this variant is online (Pouillard, 2011b).

Let us now consider an example where an environment is represented as an explicit data structure, namely an association list, where keys are binders. Here are the definitions of this data structure and of the environment lookup function:

```
data DataEnv (A : Set) : (α β : World) → Set where
  ε : ∀ {β} → DataEnv A β β
  _,_↦_ : ∀ {α β} (Γ : DataEnv A α β) b (x : A)
          → DataEnv A (b ◃ α) β

lookup : ∀ {A α β} → DataEnv A α β → Name α → Name β ⊎ A
lookup ε             = inj₁
lookup (Γ , _ ↦ v) = exportWith (inj₂ v) (lookup Γ)
```

The type `DataEnv A α β` is the type of an environment, or environment fragment, where every name in the environment is associated with a datum of type `A`. We refer to the parameter α as the "inner world", and to the parameter β as the "outer world". The outer world can be thought of as the world that exists before the binders in the environment are introduced. The inner world is the world obtained after these binders are introduced. The expression `lookup Γ x` looks up the name `x` in the environment Γ. The name `x` must make sense in the scope of Γ, that is, `x` must inhabit the inner world α. If `x` is found among the bindings, then the information associated with `x` is returned. This information has type `A`. If `x` is not found among the bindings, then `x` is returned, with a more precise type: indeed, since `x` is not among the names introduced by Γ, it must make sense outside Γ, that is, in the outer world β.

We illustrate the use of `DataEnv` with an alternative definition of the function `fv`. Here, the payload type parameter `A` is instantiated with the unit type $\top$. This variant avoids the need to take the bound atoms off the list by not inserting them in the first place. At variable occurrences, we use `lookup` to test whether the name is free or bound. If it is free, we wrap it in a singleton list (using the function `[_]`) and return it. If it is bound, we ignore it and return an empty list (using the function `const []`). The function `[_,_]′` allows eliminating the sum produced by `lookup`. At every other node, we simply carry out a recursive traversal. Whenever a name abstraction is entered, the current environment Γ is extended with the bound name `b`.

```
fv' : ∀ {α β} → DataEnv ⊤ α β → Tm α → List (Name β)
fv' Γ (V x)       = [ [_] , const [] ]′ (lookup Γ x)
fv' Γ (t · u)     = fv' Γ t ++ fv' Γ u
fv' Γ (ƛ b t)     = fv' (Γ , b ↦ _) t
fv' Γ (Let b t u) = fv' Γ t ++ fv' (Γ , b ↦ _) u

fv : ∀ {α} → Tm α → List (Name α)
fv = fv' ε
```

Admittedly, neither functions nor lists are the most efficient representation of environments. It would be nice to be able to implement environments using, say, balanced binary

search trees. At the moment, this cannot be done by the user, outside our library. The reason is, the library does not expose a total ordering on names. We cannot expose such an ordering: the logical relation which we build in section 7 forbids it. The library could, however, offer an efficient implementation of association maps whose keys are names: this would be permitted by the logical relation. We leave a deeper study of this issue for future work.

5.3 Example: comparing terms

We now show how terms can be tested for α-equivalence, or, more generally, for α-equivalence up to a certain relation over their free names.

We first define the type `|Cmp| F` of functions that compare `F`-structures, where `F` is an indexed type. In the following, the type `Ix` of indices will be instantiated with `World`, and the type `F` will be instantiated with `Name` or `Tm`.

```
|Cmp| : ∀ {Ix} (F : Ix → Set) (i j : Ix) → Set
|Cmp| F i j = F i → F j → Bool
```

In order to compare two terms, we carry down an environment, which tells us how to compare two names. At name occurrences, we consult this environment. At application nodes, we carry it down. At name abstractions, we must extend it. To this end, we define the function `extendNameCmp`. This function receives a name comparator `f` for worlds α_1 and α_2, a name x_1 in the world $b_1 \triangleleft \alpha_1$, and a name x_2 in the world $b_2 \triangleleft \alpha_2$. Then, the function `extendNameCmp` attempts to export x_1 through b_1 and x_2 through b_2. If both attempts succeed, then we can use `f` to compare x_1' and x_2'. If both attempts fail, then x_1 is b_1 and x_2 is b_2; hence, each of these two names is equal to the nearest enclosing binder, and we return `true`. If one attempt succeeds and the other fails, then we return `false`.

```
extendNameCmp : ∀ {α₁ α₂ b₁ b₂} → |Cmp| Name α₁ α₂
                                 → |Cmp| Name (b₁ ◁ α₁) (b₂ ◁ α₂)
extendNameCmp f x₁ x₂
  with exportᴺ? x₁   | exportᴺ? x₂
...  | just x₁′      | just x₂′    = f x₁′ x₂′
...  | nothing       | nothing     = true
...  | _             | _           = false

cmpTm : ∀ {α₁ α₂} (Γ : |Cmp| Name α₁ α₂) → |Cmp| Tm α₁ α₂
cmpTm Γ (V x₁)       (V x₂)       = Γ x₁ x₂
cmpTm Γ (t₁ · u₁)    (t₂ · u₂)    = cmpTm Γ t₁ t₂ ∧ cmpTm Γ u₁ u₂
cmpTm Γ (ƛ _ t₁)     (ƛ _ t₂)     = cmpTm (extendNameCmp Γ) t₁ t₂
cmpTm Γ (Let b₁ t₁ u₁) (Let b₂ t₂ u₂)
  = cmpTm Γ t₁ t₂ ∧ cmpTm (extendNameCmp Γ) u₁ u₂
cmpTm _ _            _            = false
```

In the above code, there is no need to compare names or binders found in the first term with names or binders found in the second term. Instead, we consider that two bound names are equal if and only if they were bound at the same time. In short, bound names are compared positionally. This explains why the lack of a function that compares binders is not a problem.

The function `cmpTm` must be able to accept two terms in different worlds for the recursion to go through successfully. In the end, though, this generality is often unnecessary. By supplying the name comparison function `_==ᴺ_` as the initial name comparator, we obtain a specialized version of `cmpTm`, baptised `_==ᵀᵐ_`. This homogeneous comparison function tests whether its arguments are α-equivalent.

```
_==ᵀᵐ_ : ∀ {α} → Tm α → Tm α → Bool
_==ᵀᵐ_ = cmpTm _==ᴺ_
```

5.4 Kits and traversals

We have seen that working with worlds requires explicitly moving names from world to world using operations like `coerceᴺ` and `exportᴺ?`. It quickly appears that it is necessary to lift these operations to user-defined algebraic data types, such as the type `Tm`, so that user-defined data structures can be moved from world to world. More generally, a number of operations on names can be lifted to user-defined algebraic data types.

Our experience with the library leads us to emphasize two points. First, in most of the operations on terms that we are about present, only the parts that deal with the binding structure vary. The code that carries out the traversal is fixed and can be written only once. Second, the parts that are specific to each operation can be made reusable, so as to work not only with the generic traversal but with custom traversals as well.

In order to share code and separate concerns, we introduce some infrastructure, which we later instantiate for the type `Tm`.

5.4.1 Traversal kits

We begin by introducing the notion of a *traversal kit*. A traversal kit is a record whose components indicate how a traversal should deal with names and binders.

The first component of a traversal kit is the parameterized type `Env` of the environments that are carried down during the traversal. We are interested in traversals that perform some kind of translation. Thus, in an environment of type `Env α β`, the parameter α represents the world which the original term inhabits, while the parameter β represents the world which the transformed term inhabits. Such an environment maps names of type `Name α` to data of type `Res β`, where the type `Res` is itself a component of the traversal kit, so that it can vary from application to application.

The last three components of a traversal kit are functions. The function `trName` looks up a name in the environment, and is typically used when the traversal reaches an occurrence of a variable. The function `trBinder` indicates how to translate a binder. For instance, this function could be the identity. Or, if there is a need to avoid capture (as is the case

when defining capture-avoiding substitution), it could be a function that returns a fresh binder. The function `trBinder` has access to the environment, which, as we will see, can encapsulate a supply of fresh binders. Finally, the function `extEnv` indicates how to extend the environment when descending under a binder. It is worth noting that, while the source world α is extended with the binder `b`, the destination world β is extended with its image through the translation, that is, `trBinder Δ b`.

```
record TrKit (Env : (α β : World) → Set)
             (Res : World → Set) : Set where
  constructor mk
  field
    trName   : ∀ {α β} → Env α β → Name α → Res β
    trBinder : ∀ {α β} → Env α β → Binder → Binder
    extEnv   : ∀ {α β} b (Δ : Env α β)
                       → Env (b ◃ α) (trBinder Δ b ◃ β)
```

In the following, we build a number of traversal kits. The coercing kit allows applying a world inclusion witness. The renaming kit allows applying a potentially effectful function of names to names. The substitution kit allows applying a function of names to terms. We also present a few ways of building new kits out of existing kits.

5.4.2 *The coercing kit*

Our first kit, called `coerceKit`, is simple and to the point. Its environment type is `_⊆_`. Its result type is `Name`. The action on names is $\texttt{coerce}^{\texttt{N}}$. The action on binders is the identity, which means that this kit does not perform any kind of renaming or freshening. Finally, the environment extension operation is just the world inclusion rule `⊆-◃`.

```
coerceKit : TrKit _⊆_ Name
coerceKit = mk coerceᴺ (const id) ⊆-◃
```

To illustrate the use of `coerceKit`, we lift $\texttt{coerce}^{\texttt{N}}$ from names to terms. Here is an inductive definition of the function `coerceTm`:

```
module CoerceTmWithCoerceKit where
  open TrKit coerceKit

  coerceTm : ∀ {α β} → α ⊆ β → α →ᵀᵐ β
  coerceTm Δ (V x)       = V (trName Δ x)
  coerceTm Δ (t · u)     = coerceTm Δ t · coerceTm Δ u
  coerceTm Δ (ƛ b t)     = ƛ (trBinder Δ b)
                             (coerceTm (extEnv b Δ) t)
  coerceTm Δ (Let b t u) = Let (trBinder Δ b) (coerceTm Δ t)
                               (coerceTm (extEnv b Δ) u)
```

The function `coerceTm` takes an inclusion witness and an input term. The inclusion witness is carried down during the traversal, used at names, and extended at abstractions. In this code, because of the declaration `open TrKit coerceKit`, the variable `trName` refers to the `trName` component of `coerceKit`, which by definition is $\texttt{coerce}^{\texttt{N}}$. Similarly, `trBinder` is the identity, and `extEnv` is $\subseteq$`-`.

We have formulated this code in such a way that the traversal is actually independent of which traversal kit is used. Permitting such a formulation is the reason why we introduced traversal kits in the first place. We will soon see that it is possible to define a generic traversal function and to redefine `coerceTm` as an instance of the generic traversal with the coercing kit (sections 5.4.6 and 5.4.7).

5.4.3 The renaming kits

We now wish to define a "renaming kit" that allows applying an arbitrary function of names to names to (the free names of) a term. In order to avoid capture, we must perform "freshening", that is, replace the binders found in the original term with fresh binders. For this purpose, we introduce the concept of a name supply. A name supply for the world α is just a pair of a binder, called $\texttt{seed}^{\texttt{B}}$, and a proof that $\texttt{seed}^{\texttt{B}}$ is fresh for α:

```
record Supply α : Set where
  constructor _,_
  field
    seed^B : Binder
    seed# : seed^B # α
```

It may seem surprising that a single fresh binder can be thought of as a name supply. The reason is that, thanks to the operation `suc#` (which was presented in section 4), a single fresh binder gives rise to an infinite stream of fresh binders. The function $\texttt{suc}^{\texttt{s}}$, defined below, helps do this: it increments both the seed and the "fresh-for" proof. The constant $\texttt{zero}^{\texttt{s}}$ is an initial name supply.

```
zero^s : Supply ∅
zero^s = 0 ^B , 0 ^B #∅

suc^s : ∀ {α} → (s : Supply α) → Supply (Supply.seed^B s ◁ α)
suc^s (seed^B , seed#) = suc^B seed^B , suc# seed#
```

In our system, a world-polymorphic function that does not need to generate fresh names is parameterized over just a world α, whereas a world-polymorphic function that needs to generate fresh names is typically parameterized over a world α and a name supply of type `Supply` α.

We are now in a position to define a renaming kit. We first define its environment type, `SubstEnv` α β. It is a record type. Its first two components, `Res` and `trName`, specify an action on names. This action is chosen by the end user: hence, the renaming kit is parametric in it. The last component of the environment, `supply`, is a name supply for

the destination world β. This reflects the fact that we need to create fresh binders in the transformed term.

```
record SubstEnv (Res : World → Set) α β : Set where
  constructor _,_
  field trName : Name α → Res β
        supply : Supply β
  open Supply supply public
```

The renaming kit, `renameKit`, is defined as follows. First, we let `Res` be `Name`.

```
RenameEnv : (α β : World) → Set
RenameEnv = SubstEnv Name
```

Then, we provide definitions for the functions `trName`, `trBinder`, and `extEnv`. The definition of `trName` is trivial: it is the `trName` component of the environment. The function `trBinder` uses the `supply` component of the environment to obtain a fresh binder. The definition of `extEnv` is the most involved part of the kit. Because an environment is a pair of an action `trName` and a supply, the job of `extEnv` is to lift these two components through a binder. The manner in which `trName` is lifted, so as to obtain a new function `trName′`, is depicted in figure 2. The function `trName′` takes a name and uses `exportWith` to test whether this name is bound or free. If this name is bound (that is, equal to `b`), then the fresh binder that was chosen to stand for `b`, namely $\texttt{seed}^{\texttt{B}}$, is returned. If this name is free, then `exportWith` refines its type to `Name α`, which allows us to apply `trName` to it. This produces a name in the world β, which is then imported back using $\texttt{coerce}^{\texttt{N}}$. This call to $\texttt{coerce}^{\texttt{N}}$ is valid only because $\texttt{seed}^{\texttt{B}}$ is known to be fresh for the destination world.

```
renameKit : TrKit RenameEnv Name
renameKit = mk SubstEnv.trName trBinder extEnv
  where
    -- Each binder is translated to a fresh binder
    trBinder : ∀ {α β} → RenameEnv α β → Binder → Binder
    trBinder (_ , (seed^B , _)) _ = seed^B

    extEnv : ∀ {α β} b (Δ : RenameEnv α β)
                          → RenameEnv (b ◁ α) (_ ◁ β)
    extEnv _ (trName   , (seed^B      , seed#β))
           = (trName′   , (suc^s (seed^B , seed#β)))
      where
       trName′ = exportWith
                   (name^B seed^B) -- bound
                   (coerce^N (⊆-# seed#β) ∘ trName) -- free
```

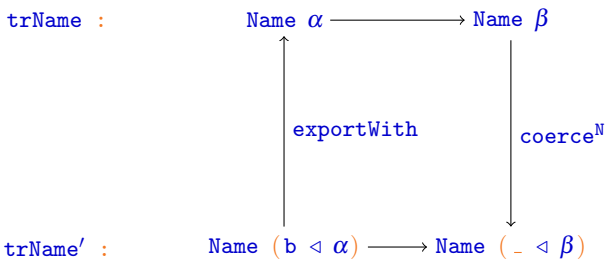

Figure 2. Lifiting `trName`

Because we have defined `Res` to be `Name`, the above renaming kit works with *total* functions of names to names. In order to lift the function `export`$^{\mathbb{N}}$`?` from names to terms, we need to deal with *partial* functions as well. This leads us to define another renaming kit, which is parameterized over a notion of effectful computation, that is, over an applicative functor. An applicative functor (McBride & Paterson, 2008) is halfway between a functor and a monad. Like a monad, an applicative functor has a unit, called `pure`. The function `pure` allows viewing a pure value as a potentially effectful one. Furthermore, an applicative functor comes with an effectful application, written `_⊛_`. This operation takes an effectful function, an effectful argument, and produces an effectful result. As an illustration, here is how one uses an applicative functor to map an effectful function over a list:

```
module MapA {E} (E-app : Applicative E) where
  open Applicative E-app

  mapA : {A B : Set} → (A → E B) → List A → E (List B)
  mapA _ []       = pure []
  mapA f (x :: xs) = pure _::_ ⊛ f x ⊛ mapA f xs
```

In order to define our second and more general renaming kit, we reuse the type `SubstEnv`, but define the result type `Res` to be `E ∘ Name`, as opposed to just `Name`. The construction is parameterized with the applicative functor `E`. This allows us to support several kinds of effects. The code for `renameAKit` is similar to that of `renameKit`, so we omit it and show only its type:

```
RenameAEnv : (E : Set → Set) (α β : World) → Set
RenameAEnv E = SubstEnv (E ∘ Name)

renameAKit : ∀ {E} → Applicative E →
               TrKit (RenameAEnv E) (E ∘ Name)
renameAKit = {! code similar to renameKit omitted !}
```

5.4.4 The substitution kit

We now generalize the renaming kit along a different direction. Instead of actions that map names to names, we now wish to work with actions that map names to "terms". The type family for terms does not have to be `Tm`: we parameterize the substitution kit over a type family `F`. We require that `F` be equipped with two operations. First, we require an operation that turns a name into a term. We call this operation `V`, by analogy with the data constructor `V` of the type `Tm`. Second, we require a way of coercing a term from one world to another. The substitution kit defines the type of environments to be `SubstEnv F`. The use of `SubstEnv` reflects the fact that we need to generate fresh binders in order to avoid capture, and the use of the parameter `F` reflects the fact that names are mapped to terms..

```
-- Index-respecting functions
F -̇> G = ∀ {i} → F i → G i

-- The type for 'coerce' on an F-term
Coerce F = ∀ {α β} → α ⊆ β → F α → F β

substKit : ∀ {F}
             (V       : Name -̇> F)
             (coerceF : Coerce F)
           → TrKit (SubstEnv F) F
substKit = {! code similar to renameKit omitted !}
```

5.4.5 Other kits and combinators

We build a few other kits and combinators (Pouillard, 2011b). For instance, `∘-Kit` composes two kits by pairing the two environments and composing their operations. Another combinator, `starKit`, takes a kit whose environments have type `Env` and builds a kit whose environments have type `Star Env`, where `Star` is AGDA's reflexive and transitive closure operator. Finally, the combinator `mapKit` allows pre-composing a function (of names to names) and post-composing a function (of results to results) with a kit to obtain a new kit. Here is the definition of `mapKit`:

```
mapKit : ∀ {Env F G} (f : Name -̇> Name) (g : F -̇> G)
         → TrKit Env F → TrKit Env G
mapKit f g kit = mk (λ Δ → g ∘ trName Δ ∘ f) trBinder extEnv
  where open TrKit kit
```

5.4.6 A reusable traversal

We now write a term-to-term transformation function that works with an arbitrary effect and with an arbitrary "name-to-term" kit. It is essentially a "map" function over terms: it maps terms to terms, and transforms names and binders as specified by the kit. More

precisely, the function `trTm` traverses the term, carrying an environment. Name occurrences are transformed into terms via `trName`. (The data constructor `V` is not necessarily preserved). Binders are transformed using `trBinder`. (In the code that follows, this information is implicit and is reconstructed by AGDA.) The structure of the term is otherwise preserved. The operations of the applicative functor are used when constructing the new term. The function `extEnv` allows carrying the environment under a binding.

```
module TraverseTm {E}   (E-app : Applicative E)
                  {Env} (trKit : TrKit Env (E ∘ Tm)) where
  open Applicative E-app
  open TrKit trKit

  trTm : ∀ {α β} → Env α β → (Tm α → E (Tm β))
  trTm Δ (V x)       = trName Δ x
  trTm Δ (t · u)     = pure _·_ ⊛ trTm Δ t ⊛ trTm Δ u
  trTm Δ (ƛ b t)     = pure (ƛ _) ⊛ trTm (extEnv b Δ) t
  trTm Δ (Let b t u) = pure (Let _) ⊛ trTm Δ t
                                    ⊛ trTm (extEnv b Δ) u
```

It is convenient to also define a specialized version of `trTm`, which accepts a "name-to-name" kit and preserves the data constructor `V`.

```
open TraverseTm
trTm′  : ∀ {E}   (E-app : Applicative E)
           {Env} (trKit : TrKit Env (E ∘ Name))
           {α β} → Env α β → (Tm α → E (Tm β))
trTm′ E-app trKit
  = trTm E-app (mapKit id (Applicative._<$>_ E-app V) trKit)
```

5.4.7 Reusing the traversal

We can now collect the fruit of our work, by combining the reusable traversal with various kits. For the sake of simplicity, we demonstrate this at type `Tm`. In the actual implementation (Pouillard, 2011b), we further abstract over `Tm` and `trTm` by defining a sequence of parameterized modules.

We now revisit the definition of `coerceTm`. Our earlier definition (section 5.4.2) can be replaced with a more concise one: all we have to do is instantiate the generic traversal `trTm′` with the identity applicative functor (which denotes the absence of side effects) and with the coercing kit.

```
-- The identity applicative functor
id-app : Applicative id
id-app = {! definition omitted !}
```

```
coerceTm : ∀ {α β} → α ⊆ β → α →ᵀᵐ β
coerceTm = trTm′ id-app coerceKit
```

We would like to think of `coerceTm` as a coercion, that is, an identity function. One can informally check that if worlds and proofs of membership in a world were erased, then `coerceTm` would indeed boil down to the identity function, which means that applications of `coerceTm` could be optimized away. However, formally studying world erasure, as well as persuading AGDA to perform this erasure, are left for future work.

To obtain a function that renames a term, we instantiate the generic traversal `trTm′` with the identity applicative functor and with the total renaming kit.

```
renameTm : ∀ {α β} → Supply β → (α →ᴺ β) → (α →ᵀᵐ β)
renameTm s f = trTm′ id-app renameKit (f , s)
```

To obtain a function that renames a term while allowing for failure, we instantiate it with the applicative functor `Maybe` and with the partial renaming kit.

```
renameTmA : ∀ {E} → Applicative E →
            ∀ {α β} → Supply β → (Name α → E (Name β))
                               → (Tm α   → E (Tm β))
renameTmA E-app s f = trTm′ E-app (renameAKit E-app) (f , s)

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

Obtaining a function that exports a term is now just a matter of instantiating `renameTm?` with the partial function `exportᴺ?`.

```
exportTm? : ∀ {b α} → Supply α → Tm (b ◁ α) →? Tm α
exportTm? s = renameTm? s exportᴺ?
```

Another useful special case of `renameTm?` is `closeTm?`. This function takes a term in any world and checks if the term is closed. If so, the same term is returned, and its type is refined to the empty world. Otherwise, the function fails by returning `nothing`:

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

Finally, in order to define capture-avoiding substitution, we use the substitution kit with arguments `V` (which means that free variables are mapped to themselves) and `coerceTm`.

```
substTm : ∀ {α β} → Supply β → (Name α → Tm β) → (α →ᵀᵐ β)
substTm (s , s#) f = trTm id-app (substKit V coerceTm) (f , s, s#)
```

To illustrate the use of `substTm`, here is a simple function, baptised `β-red`, which performs a β-reduction when a β-redex appears at the root of its argument. The function `exportWith a V` maps `a` to `b` and maps `x` to `V x` when `x` $\not\equiv$ `b`.

```
β-red : ∀ {α} → Supply α → Tm α → Tm α
β-red s (ƛ b f · a) = substTm s (exportWith a V) f
β-red _ t           = t
```

5.5 Building any λ-term

One way to argue that every λ-term can be represented using our type `Tm` is to define a conversion function from another type for λ-terms to the type `Tm`. We do so by choosing the "bare nominal" type $\mathtt{Tm}^{\mathtt{A}}$ of section 3.1 as the source language.

The process is very close to the combination of a specific renaming kit and a specific traversal function. The kit is specific because the source names are of type `Atom` and not `Name`. The traversal function is specific because the source and target types are not the same and because we fix the identity functor for simplicity.

First, we introduce the type of environments. An environment holds a mapping from free atoms to free names and a name supply:

```
module Conv-Tm^A→Tm where
  record Env α : Set where
    constructor _,_
    field
      trAtom : Atom → Name α
      supply : Supply α
    open Supply supply public
  open Env
```

Then, we define how an environment is extended. This is similar to what we did for the renaming kit:

```
  extEnv : ∀ {α} → Atom → (Δ : Env α) → Env (seed^B Δ ◃ α)
  extEnv b^A Δ = tr^N , suc^s (supply Δ)
    where tr^N = λ x^A → if b^A ==^A x^A
                         then name^B (seed^B Δ)
                         else coerce^N (⊆-# (seed# Δ)) (trAtom Δ x^A)
```

It is then straightforward to define the conversion function, `conv`. We use `trAtom` at variable occurrences and `extEnv` when crossing a binding.

```
conv : ∀ {α} → Env α → Tm^A → Tm α
conv Δ (V x)       = V (trAtom Δ x)
conv Δ (ƛ b t)     = ƛ _ (conv (extEnv b Δ) t)
conv Δ (t · u)     = conv Δ t · conv Δ u
conv Δ (Let b t u) = Let _ (conv Δ t) (conv (extEnv b Δ) u)
```

In order to use the function `conv`, one must provide an environment, that is, a mapping of (all) atoms to names in the world α together with a name supply for α. Of course, this is possible only if α is a non-empty world. For instance, the following environment, whose action maps all atoms to the name `0 ᴺ` and whose name supply begins at `1 ᴺ`, is an environment for the singleton world `0 ᴮ ◁ ∅`.

```
emptyEnv : Env (0 ᴮ ◁ ∅)
emptyEnv = const (0 ᴺ) , sucˢ zeroˢ
```

By post-composing the conversion function `conv emptyEnv` with the test for closedness `closeTm?`, we obtain a function that converts a closed "bare nominal" term of type `Tm^A` to a term of type `Tm ∅`. This function fails if its argument is not a closed term.

```
conv∅? : Tm^A →? Tm ∅
conv∅? = closeTm? ∘ conv emptyEnv
```

5.6 Towards elaborate uses of worlds

The type `Tm` is just one basic example of an algebraic data type that involves names and binders. Let us briefly present a few algebraic data type definitions that make more advanced use of worlds.

Contexts Consider a type `C` of one-hole contexts associated with `Tm`. The type `C` is indexed with two worlds α and β, which respectively play the role of an "outer world" and an "inner

world". The idea is, plugging a term of type `Tm β` into the hole of a context of type `C α β` produces a term of type `Tm α`. The definition of the type `C` is as follows:

```
data C α : World → Set where
  Hole   : C α α
  _·1_   : ∀ {β}    → C α β → Tm α → C α β
  _·2_   : ∀ {β}    → Tm α  → C α β → C α β
  ƛ      : ∀ {β} b → C (b ◃ α) β → C α β
  Let₁   : ∀ {β} b → C α β
                   → Tm (b ◃ α) → C α β
  Let₂   : ∀ {β} b → Tm α
                   → C (b ◃ α) β → C α β
```

Contexts bind names: the hole can appear under one or several binders. This is why, in general, a context has distinct outer and inner worlds. A context contains a list of binders that "connects" the outer and inner worlds: these binders are carried by the constructors `ƛ` and `Let₂`.

A context and a term can be paired to produce a term-in-context. This can be viewed as a user-defined binding construct: the names introduced by the context are in scope in the term. In fact, a one-hole context for a data structure that involves binders is exactly what de Bruijn calls a "telescope" (1991). A telescope is a first-class object that has binding power, that is, it binds zero or more names.

```
CTm : World → Set
CTm α = ∃[ β ](C α β × Tm β)
```

It is straightforward to define a function `plug` from `CTm α` to `Tm α`, which accepts a pair of a context and a term and plugs the latter into the former. Conversely, one can define a family of focusing functions of type `∀{α} → Tm α → CTm α` that split a term into a pair of a context and a term. There are several such functions, according to where one wishes to focus.

The contexts presented here are "ordinary" contexts: the root of the context is the root of the term, and as one goes down into the context, one goes down into the term. Of course, since a context is just a list, it is possible to hold it from the other end. A context that is "inside-out" is known as a "zipper" (Huet, 1997; McBride, 2001). Our system can express zippers, as well as the list reversal functions that allow transforming a telescope into a zipper and vice-versa. Unfortunately, describing this in detail would take us a little too far, so we leave this for another occasion.

Multiple sorts of names Some object languages have multiple sorts of names. For instance, in Girard and Reynolds' System F, there are term variables, which occur in terms, and type variables, which occur in types and in terms. Thus, it is natural to index object-level types with one world (which tells which type variables are in scope) and to index

object-level terms with two worlds (one of which concerns type variables, the other of which concerns term variables).

```
module SysF where
  infixr 5 _⇒_
  data Ty α : Set where
    V    : (x : Name α)          → Ty α
    _⇒_  : (σ τ : Ty α)          → Ty α
    '∀'  : ∀ b (τ : Ty (b ◃ α))  → Ty α

  data Tm α γ : Set where
    V     : ∀  (x : Name α)             → Tm α γ
    _·_   : ∀  (t u : Tm α γ)           → Tm α γ
    ƛ     : ∀ b (τ : Ty γ)
                (t : Tm (b ◃ α) γ)      → Tm α γ
    _·τ_  : ∀  (t : Tm α γ) (τ : Ty γ)  → Tm α γ
    Λ     : ∀ b (t : Tm α (b ◃ γ))      → Tm α γ
```

5.7 Advanced example: normalization by evaluation

As an advanced example, we show how to express a normalization by evaluation algorithm in our system. This algorithm has been previously used as a benchmark by several researchers (Shinwell *et al.*, 2003; Pitts, 2006; Licata & Harper, 2009; Cave & Pientka, 2012). The challenge lies in the way in which the algorithm mixes computational functions, name abstractions, and fresh name generation.

The object language of interest is again the pure λ-calculus. The algorithm exploits two different representations of object-level terms, which are respectively known as *syntactic* and *semantic* representations. Because these representations differ only in their treatment of name abstractions, they can be given a common definition, which is parameterized over the representation of name abstractions:

```
module M (Abs : (World → Set) → World → Set) where
  data T α : Set where
    V    : Name α     → T α
    ƛ    : Abs T α    → T α
    _·_  : T α → T α → T α
```

The parameter `Abs` has kind `(World → Set) → (World → Set)`: it is an indexed-type transformer.

In order to obtain the syntactic representation, we instantiate `Abs` with the nominal abstractions that we have used everywhere so far: an abstraction is a package of a binder and of a term that inhabits an extended world. This yields the type `Term` of syntactic terms.

```
SynAbsᴺ : (World → Set) → World → Set
SynAbsᴺ F α = ∃[ b ](F (b ◁ α))

open M SynAbsᴺ renaming (T to Term)
```

In order to obtain the semantic representation, we instantiate `Abs` with a different notion of abstraction, in the style of higher-order abstract syntax: an abstraction is a computational function, which substitutes a term for the bound name of the abstraction. This yields the type `Sem` of semantic terms.

```
SemAbs : (World → Set) → World → Set
SemAbs F α = ∀ {β} → α ⊆ β → F β → F β

open M SemAbs renaming (T to Sem)
```

`Sem` is not an inductive data type. Fortunately, with the `--no-positivity-check` flag, AGDA accepts this type definition, at the cost of breaking strong normalization. (To minimize risk, we isolate this code in a module where this flag is activated.) Naturally, because untyped λ-calculus is not terminating, one cannot expect to be able to implement a terminating normalization procedure.

Our semantic name abstractions involve bounded polymorphism in a world: we define `SemAbs F α` as `∀{β} → α ⊆ β → F β → F β`, as opposed to the more naïve `F α → F α`. This provides a more accurate and more flexible description of the behavior of substitution. Indeed, when instantiating an abstraction `t` with some term `u`, it makes perfect sense for `u` to inhabit a larger world than `t`, that is, for `u` to refer to certain names that are fresh for `t`. The result of the substitution then inhabits the same world as `u`: that is, it potentially refers to these fresh names, in addition to all of the names that occurred free in the abstraction `t`.

The types `SemAbs` and `Sem` are covariant with respect to the parameter α. This would not be the case had we adopted the naïve definition of `SemAbs`. In other words, it is possible to define a "coerce" operation for semantic terms:

```
coerceSem : ∀ {α β} → α ⊆ β → (Sem α → Sem β)
coerceSem pf (V a)     = V (coerceᴺ pf a)
coerceSem pf (ƛ f)     = ƛ (λ pf′ v → f (⊆-trans pf pf′) v)
coerceSem pf (t · u)   = coerceSem pf t · coerceSem pf u
```

At a semantic abstraction, no recursive call is performed, because the body of the abstraction is opaque: it is a computational function. Instead, we exploit the transitivity of world inclusion and build a new semantic abstraction that inhabits the desired world. Like `coerceTm` (section 5.4.7), `coerceSem` is a "coercion", in the sense that, if worlds and

proofs of membership in a world were erased, `coerceSem` would boil down to the identity function.

The normalization by evaluation algorithm makes use of environments. Here, environments are functions from names to semantic terms. The function `_,_↦_` extends such an environment:

```
EvalEnv : (α β : World) → Set
EvalEnv α β = Name α → Sem β
-- α is the inner world
-- β is the outer world

_,_↦_ : ∀ {α β} (Γ : EvalEnv α β) b → Sem β → EvalEnv (b ◁ α) β
_,_↦_ Γ b v = exportWith v Γ
-- meaning: b ↦ v
--          x ↦ Γ x
```

An environment, of type `EvalEnv α β` maps a name of type `Name α` to a semantic term that lies outside the scope of the environment, that is, a semantic term of type `Sem β`. The type `EvalEnv α β` is covariant in its destination world, as witnessed by the following coercion function:

```
coerceEnv : ∀{α β₁ β₂}→ β₁ ⊆ β₂ → EvalEnv α β₁ → EvalEnv α β₂
coerceEnv pf Γ = coerceSem pf ∘ Γ
```

The first part of the normalization by evaluation algorithm is a function `eval` that evaluates a syntactic term within an environment to produce a semantic term. When evaluating a λ-abstraction, we build a semantic abstraction, which encapsulates a recursive call to `eval`. The bounded polymorphism required by the definition of semantic abstractions forces us to coerce the environment Γ via `coerceEnv`.

```
eval : ∀ {α β} → EvalEnv α β → Term α → Sem β
eval Γ (ƛ (a , t))
  = ƛ (λ pf v → eval (coerceEnv pf Γ , a ↦ v) t)
eval Γ (V x)       = Γ x
eval Γ (t · u)     = app (eval Γ t) (eval Γ u) where
  app : ∀ {α} → Sem α → Sem α → Sem α
  app (ƛ f)  v = f ⊆-refl v
  app n      v = n · v
```

The second part of the algorithm reifies a semantic term back into a syntactic term. When reifying a semantic abstraction, we build a syntactic abstraction. This requires generating a fresh name, and leads us to parameterizing `reify` with a supply of fresh names.

```
reify : ∀ {α} → Supply α → Sem α → Term α
reify s         (V a)   = V a
reify s         (n · v) = reify s n · reify s v
reify (sᴮ , s#) (ƛ f)   =
    ƛ (sᴮ , reify (sucˢ (sᴮ , s#)) (f (⊆-# s#) (V (nameᴮ sᴮ))))
```

The constructor `V` has type `Name α → Sem α`. Hence, it is a valid initial environment of type `EvalEnv α α`. Evaluation under this initial environment, followed with reification, yields a normalization algorithm. This algorithm works with open terms: its argument, as well as its result, are terms in an arbitrary world α, provided we have a name supply for the world α.

```
nf : ∀ {α} → Supply α → Term α → Term α
nf supply = reify supply ∘ eval V
```

In particular, `zero`^{s} is a name supply for the empty world, so we can normalize closed terms. Here is an example of the normalization of a closed term:

```
idᵀ : Term ∅
idᵀ = ƛ (0 ᴮ , V (0 ᴺ))

test-nf : nf zeroˢ ((idᵀ · (idᵀ · idᵀ)) · idᵀ) ≡ idᵀ
tset-nf = refl
```

6 The NOMPA implementation (nominal fragment)

The implementation of our library (of which only the nominal fragment has been presented so far) is not surprising. Most of the code consists of types and proofs. Although we now present some of the internal details of our library, we emphasize that none of these definitions are meant to be known to or used by the client.

Worlds are defined first. A world is represented by a list of Booleans. An integer atom n is deemed a member of the world if and only if the n^{th} element of the list is `true`. More formally, the meaning of a world is defined by the following membership predicate.

```
World : Set
World = List Bool

∅ : World
∅ = []
```

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

The choice of a list of Booleans to represent a world was guided by two facts. First, operations over worlds are defined by structural induction. This makes type-level computation easier, and becomes especially important when we introduce support for de Bruijn indices (section 9). Second, because elements are ordered, modulo trailing occurrences of `false`, two equivalent sets are represented in the same way.

Worlds are meant to be computationally irrelevant. This means that, prior to running a program, it should be possible in principle to erase worlds as well as proofs of membership in a world, proofs of world inclusion, and proofs of freshness. A program in which worlds have been erased should behave in the same manner as the original program in which worlds are present, but opaque. There are two reasons why we wish to have such an erasure property: first, it means that there is a clear "phase distinction" between the code that we wish to run and the world annotations that explain why this code makes sense; second, this guarantees that world annotations incur no performance penalty.

Although we do not formally demonstrate that it is possible to erase worlds, the library is designed with this goal in mind. In particular, we are careful *not* to include in the library any operation that constructs a non-erasable result out of an erasable argument. An example of this would be an operation that accepts a world α and produces a binder `b` that is fresh for α. To compensate for the lack of such an operation, the client of the library must work with explicit name supplies where required. It might be possible to add this operation to the library (somewhat amazingly, it seems that the soundness proof in section 7 would support it) and to implement it, after erasure, as an effectful "global gensym" operation. We leave this idea for future work.

A notion of irrelevance has recently been introduced in AGDA. The irrelevant function space is noted `.(x : A) → B`. The value of an irrelevant argument not only cannot influence the result of a computation, but also cannot influence the type-checking process: any two irrelevant values of the same type are considered equal. Hence, irrelevance can be used only in situations where the only thing that matters is the existence of a value of a

certain type. In our setting, worlds cannot be considered irrelevant. The following example shows that considering an arbitrary world α as equal to the empty world leads to nonsense:

```
module Worlds-should-be-erased-but-are-relevant
  (World : Set)            -- A type for worlds
  (∅ : World)              -- An empty world
  (Name : .World → Set)    -- Names are made world irrelevant
  (¬Name∅ : ¬(Name ∅))     -- No name inhabits ∅
 .(α : World)              -- An irrelevant world
  (x : Name α)             -- A name
where
  bot : ⊥                  -- ...and that's the end of the world
  bot = ¬Name∅ x
```

One might however be able to apply AGDA irrelevance to proof terms, such as world membership witnesses, fresh-for witnesses, and maybe inclusion witnesses as well. Our experiments were successful as far as the implementation is concerned, but led to trouble in the proofs. We leave this aspect to future work.

Binders are represented by natural numbers. The operation `_◁_` defines how to extend a world with a binder. Given a binder `n` and a world α, it updates the world α with the value `true` at index `n`.

```
Binder : Set
Binder = ℕ

zeroᴮ : Binder
zeroᴮ = zero

sucᴮ : Binder → Binder
sucᴮ = suc

_◁_ : Binder → World → World
zero  ◁ []       = true  :: []
suc n ◁ []       = false :: n ◁ []
zero  ◁ (_ :: α) = true  :: α
suc n ◁ (b :: α) = b     :: n ◁ α

infixr 5 _◁_
```

The proof that `_◁_` has the intended set-theoretic semantics is offered by the following lemma:

```
◁-sem : ∀ α x y → (x ∈ y ◁ α) ≡ (if x ==ℕ y then ⊤ else x ∈ α)
```

A name of type `Name α` is a pair of a binder (that is, a number) and a proof that this binder is a member of the world `α`. In AGDA, we use a record:

```
record Name α : Set where
  constructor _,_
  field
    binderᴺ   : Binder
    b∈α       : binderᴺ ∈ α
infixr 4 _,_
open Name public
```

In order to produce a name out of a binder `b`, the operation `nameᴮ` simply packs `b` together with a proof that `b` is a member of the world `b ◁ α`. This proof is easily manually constructed.

```
nameᴮ : ∀ {α} b → Name (b ◁ α)
nameᴮ b = b , {! proof omitted !}
```

The equality test `_==ᴺ_` and the function `exportᴺ?` compare the integer values that underlie names and binders.

```
_==ᴺ_ : ∀ {α} (x y : Name α) → Bool
_==ᴺ_ (x , _) (y , _) = x ==ℕ y

exportᴺ? : ∀ {b α} → Name (b ◁ α) →? Name α
exportᴺ? {b} {α} (x , pf)
 if x ==ℕ b then nothing
            else just (x , {! proof omitted !})
```

World inclusion is defined as set-theoretic inclusion, that is, as the preservation of membership. This is exploited in the definition of `coerceᴺ`, where we need to build a proof that `b` is a member of `β`. Note that, after erasure, `coerceᴺ` boils down to the function that maps `b` to `b`, that is, the identity function.

```
infix 2 _⊆_
_⊆_ : (α β : World) → Set
α ⊆ β = ∀ x → x ∈ α → x ∈ β

coerceᴺ : ∀ {α β} → α ⊆ β → (α →ᴺ β)
coerceᴺ α⊆β (b , b∈α) = b , α⊆β b b∈α
```

The proofs of the world inclusion rules (figure 1) are computationally irrelevant. We omit them here.

The remaining part is the fresh-for relation (`_#_`). To cope with the proof of `suc#`, we give two characterizations of this relation. One is a set of syntactic rules (omitted here)

and the other is semantic. These presentations are equivalent (Pouillard, 2011b). The semantic version was given earlier (section 4.3): it states that `x # α` holds if and only if `x` dominates `α`, that is, `x` is strictly greater than every name `y` that inhabits `α`.

```
_#_ : Binder → World → Set
x # α = ∀ y → y ∈ α → x > y
```

This strong definition of "fresh-for" allows us to implement the operation `suc#` without inspecting the world: after erasure, `suc#` is just the successor operation on natural numbers. Thus, we are able to generate fresh names in a manner that is compatible with erasure and is efficient.

7 Soundness of NOMPA (nominal fragment)

Our library is written in AGDA, a type-safe language. Thus, the property that "well-typed programs do not go wrong" comes for free. However, this does not quite satisfy us. Indeed, we have explained that certain operations, such as an equality test for binders, must not be provided to programmers, or it would be possible to write "ill-behaved" code. Yet, these operations are perfectly type-safe. So, type safety is not a sufficient criterion in order to determine which operations can and cannot be provided.

Earlier, we stated informally that "a function is well-behaved if, when applied to α-equivalent arguments, it produces α-equivalent results". This is, roughly speaking, the criterion that we are looking for: we would like to guarantee that every function that can be written by a client of our library is well-behaved. Of course, we must define this criterion in a more formal and more general manner. This involves defining what we mean by "α-equivalence": we must define this relation not just at our example type `Tm`, where we have a pretty clear idea of what "α-equivalence" means, but at every type. Similarly, we must define "well-behavedness" not just at function types, but at every type.

Fortunately, these two problems are the same. In the following, we build a logical relation, that is, a type-indexed equivalence relation. This relation gives rise to a notion of "α-equivalence": we consider that two AGDA expressions of type τ are "α-equivalent" if and only if they are related at type τ. It also gives rise to a notion of "well-behavedness": we consider that an AGDA expression of type τ is "well-behaved" if and only if it is related to itself at type τ.

The construction of a logical relation for AGDA is a standard technique (Bernardy *et al.*, 2010). It is independent of our work. By relying on this technique, all we have to do is define the relation at each of the abstract types that we introduce (namely `World`, `Name`, `_⊆_`, etc.) and prove that each of the values that we introduce is related to itself. Each of these little proofs is independent of the others. This makes the soundness proof modular. This also facilitates the addition of new features: when considering the addition of a new operation, it is easy to construct the proof obligation that comes with it and to find out whether it is safe to add this operation.

This section is organized as follows. First, we recall the basics of logical relations and parametricity (section 7.1). We give a toy example, so as to practice a bit (section 7.2). Then, we define the logical relation at each of our abstract types, and briefly describe the

proof obligations that arise about the operations of the library (section 7.3). Finally, we discuss the meaning of the "free theorems" that arise out of this construction (section 7.4).

7.1 Recap of the framework

A relation is said to be type-indexed, or type-directed, when it is inductively defined over the structure of types. Let $\mathscr{R}$ be such a type-directed relation, and let τ be a type. Then, $\mathscr{R}_\tau$ is a relation on values of type τ, that is, we have $\mathscr{R}_\tau$ `:` $\tau \to \tau \to$ `Set`. Recall that `Set` serves as the type of propositions in AGDA.

A type-directed relation is called a "logical" relation when it relates functions in an extensional manner, that is, when two functions are related if and only if they produce related results out of related arguments. Let $\mathtt{A}_r$ be a relation for the arguments and $\mathtt{B}_r$ a relation for results. Two functions $\mathtt{f}_1$ and $\mathtt{f}_2$ are "logically" related if and only if for every pair of arguments $(\mathtt{x}_1, \mathtt{x}_2)$ related by $\mathtt{A}_r$, the results $\mathtt{f}_1\ \mathtt{x}_1$ and $\mathtt{f}_2\ \mathtt{x}_2$ are related by $\mathtt{B}_r$. This definition can be given in AGDA as well:

```
RelatedFunctions A_r B_r f_1 f_2 =
                 ∀ {x_1 x_2} → A_r x_1 x_2
                             → B_r (f_1 x_1) (f_2 x_2)
```

An expression of type τ "fits" a logical relation if and only if it is related to itself at type τ. A logical relation is *universal* if every well-typed program fits this logical relation. John Reynolds defined a logical relation for the polymorphic λ-calculus and proved that it is universal: this is the "Abstraction Theorem" (Reynolds, 1983). Bernardy et al. (2010) define a logical relation for every pure type system (PTS) and informally suggest how to extend it to AGDA. While no complete mechanized definition and proof exist, we refer to this extension as the "AGDA logical relation" and assume that it is universal. In the following, we briefly explain how the AGDA logical relation is defined and state our assumption in a precise way.

The AGDA logical relation In order to simplify things, the definitions that follow are not universe-polymorphic. The reader can find universe-polymorphic definitions in the full implementation (Pouillard, 2011b).

When attempting to formally define a logical relation within AGDA, one immediately faces a difficulty: AGDA does not allow structural induction over types, that is, over values of type `Set`. In order to work around this difficulty, a natural and common technique is to introduce an algebraic data type `U` that represents the syntax of AGDA's types. The type `U` is known as a "universe", and its elements are known as "codes". Then, in order to construct an explicit connection between codes and the types that they are supposed to represent, one defines a function that assigns meanings to codes. This function, called `El`, has type `U → Set`. Thus, if τ is a code, then `El` τ is a type, and can be thought of as "the elements of τ". Finally, the logical relation is defined by induction over codes. It is a function $[\![\_]\!]$ that maps a code τ to a relation over the elements of τ. In other words, the function $[\![\_]\!]$ has type $(\tau$ `:` `U`$)$ `→ El` τ `→ El` τ `→ Set`.

Unfortunately, because AGDA's types involve quantification and dependent types, the algebraic data type of codes must involve some representation of names and binders. This adds a good deal of complexity to the universe technique. Perhaps ironically, we do not wish to deal with this complexity, as it would obscure our idea. Thus, we do *not* adopt the universe technique.

Instead, we follow a simpler and more limited approach. First, we give a formal (non-inductive) definition of the logical relation at every type constant. In AGDA, the type constants are $\_\to\_$, Π, $\mathtt{Set}_0$, and the user-defined inductive data types, such as $\mathbb{N}$. Thus, to each such constant κ, we associate a relation, which we write $[\![\kappa]\!]$. (Note that there are no spaces in this name: $[\![\kappa]\!]$ is just the name of a new constant. We are *not* formally defining a function called $[\![\_]\!]$.) Thus, we define $\_[\![\to]\!]\_$, $[\![\Pi]\!]$, $[\![\mathtt{Set}_0]\!]$, and one constant per user-defined inductive data type, such as $[\![\mathbb{N}]\!]$. Then, instead of giving a formal inductive definition of the logical relation at every type, we view the application of the logical relation to a type as an informal "macro-expansion" process. For instance, imagine we wish to compute the definition of the logical relation at type $\mathbb{N} \to \mathbb{N} \to \mathtt{Bool}$. We cannot write $[\![\ \mathbb{N} \to \mathbb{N} \to \mathtt{Bool}\ ]\!]$, with spaces near the brackets, because we have not formally defined a function called $[\![\_]\!]$. Instead, we manually distribute the semantic brackets over the arrows and write $[\![\mathbb{N}]\!]\ [\![\to]\!]\ [\![\mathbb{N}]\!]\ [\![\to]\!]\ [\![\mathtt{Bool}]\!]$, without spaces near the brackets. Because we have formally defined the constants $[\![\mathbb{N}]\!]$, $[\![\mathtt{Bool}]\!]$, and $\_[\![\to]\!]\_$, this is a valid AGDA expression, whose meaning can be automatically computed by AGDA.

The definition of $\_[\![\to]\!]\_$ is just `RelatedFunctions`. The definition of $[\![\Pi]\!]$ is a dependent version of `RelatedFunctions`, where the relation that is required of the results is allowed to depend on the manner in which the arguments are related:

```
⟦Π⟧ A_r B_r f₁ f₂ = ∀ {x₁ x₂} (x_r : A_r x₁ x₂)
                  → B_r x_r (f₁ x₁) (f₂ x₂)
```

Following Bernardy et al. (2010), the definition of the constant $[\![\mathtt{Set}_0]\!]$ is as follows:

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

The type $\mathtt{A}_1 \to \mathtt{A}_2 \to \mathtt{Set}_0$ is the type of all relations between the types $\mathtt{A}_1$ and $\mathtt{A}_2$. Although this definition may seem somewhat cryptic at first glance, it allows recovering Reynolds' definition of the logical relation at polymorphic types. In AGDA, a polymorphic type is encoded as a dependent type of the form $(\mathtt{A} : \mathtt{Set}_0) \to \tau$. By combining the above definitions of $[\![\Pi]\!]$ and $[\![\mathtt{Set}_0]\!]$, one finds that the logical relation at such a polymorphic type involves a universal quantification over three things, namely two types $\mathtt{A}_1$ and $\mathtt{A}_2$ and a relation $\mathtt{A}_r$ between these types. This is Reynolds' definition.

We recall all of the above definitions in figure 3, and introduce some syntactic sugar. These definitions cover core type theory. Inductive data types are covered in a simple and systematic manner. The process is as follows: for each constructor κ of type τ, declare a new constructor $[\![\kappa]\!]$ whose type is $[\![\tau]\!]\ \kappa\ \kappa$. Record types are treated in an analogous manner. As an illustration, the logical relations for the inductive data types that are used in this paper are given in figure 4.

As announced earlier, we do not formally define a function $[\![\_]\!]$. Instead, if τ is a closed type, we view $[\![\ \tau\ ]\!]$ as a "macro", which can be manually expanded by replacing within τ every constant κ with $[\![\kappa]\!]$, every non-dependent arrow $\mathtt{A} \to \mathtt{B}$ with $[\![\ \mathtt{A}\ ]\!]\ [\![\to]\!]\ [\![\ \mathtt{B}\ ]\!]$, every dependent arrow $(\mathtt{x} : \mathtt{A}) \to \mathtt{B}$ with $\langle\ \mathtt{x}_r : [\![\ \mathtt{A}\ ]\!]\ \rangle[\![\to]\!]\ [\![\ \mathtt{B}\ ]\!]$, etc. By convention, we use the subscript $\mathtt{r}$ in the expansion of dependent arrows. Here are a few examples of the manual expansion of the notation $[\![\_]\!]$:

```
-- What we would like to write but cannot:
[[ ℕ → ℕ → Bool ]] =
-- What we write instead:
[[ℕ]] [[→]] [[ℕ]] [[→]] [[Bool]] =
-- What this means:
λ f₁ f₂ →
  ∀ {x₁ x₂} (xᵣ : [[ℕ]] x₁ x₂)
    {y₁ y₂} (yᵣ : [[ℕ]] y₁ y₂)
  → [[Bool]] (f₁ x₁ y₁) (f₂ x₂ y₂)

-- The logical relation at a polymorphic type:
[[ (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)
```

The parametricity hypothesis As announced earlier, we assume that the AGDA logical relation is universal. We can now state this assumption in a precise way:

(Parametricity hypothesis for AGDA*) We assume that, for every well-typed term* $\mathtt{M}$ *of closed type* τ*, the theorem* $[\![\ \tau\ ]\!]\ \mathtt{M}\ \mathtt{M}$ *is provable.*

Because $[\![\_]\!]$ is an informal notation, as opposed to an AGDA function, the above hypothesis must be stated in an informal manner. Nevertheless, for a specific type τ, it is possible to give a formal statement of this hypothesis. For instance, in section 11.2, where we prove that "world-polymorphic functions commute with renamings", we explicitly and formally use a hypothesis of the form $[\![\ \tau\ ]\!]\ \mathtt{f}\ \mathtt{f}$, for a specific type τ and for a specific term $\mathtt{f}$.

```
⟦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)
A_r ⟦→⟧ B_r = λ f₁ f₂ → ∀ {x₁ x₂} → A_r x₁ x₂ → B_r (f₁ x₁) (f₂ x₂)

infixr 0 _⟦→⟧_

⟦Π⟧ : ∀ {A₁ A₂} (A_r : ⟦Set₀⟧ A₁ A₂)
        {B₁ B₂} (B_r : (A_r ⟦→⟧ ⟦Set₀⟧) B₁ B₂)
      → ((x : A₁) → B₁ x) → ((x : A₂) → B₂ x) → Set₁
⟦Π⟧ A_r B_r = λ f₁ f₂ → ∀ {x₁ x₂} (x_r : A_r x₁ x₂) → B_r x_r (f₁ x₁) (f₂ x₂)

syntax ⟦Π⟧ A_r (λ x_r → f) = ⟨ x_r : A_r ⟩⟦→⟧ f

⟦∀⟧ : ∀ {A₁ A₂} (A_r : ⟦Set₀⟧ A₁ A₂)
        {B₁ B₂} (B_r : (⟦Set₀⟧ ⟦→⟧ ⟦Set₀⟧) B₁ B₂)
      → ⟦Set₁⟧ ({x : A₁} → B₁ x) ({x : A₂} → B₂ x)
⟦∀⟧ A_r B_r = λ f₁ f₂ → ∀ {x₁ x₂} (x_r : A_r x₁ x₂) → B_r x_r (f₁ {x₁}) (f₂ {x₂})

syntax ⟦∀⟧ A_r (λ x_r → f) = ∀⟨ x_r : A_r ⟩⟦→⟧ f
```

Figure 3. Logical relations for core type theory

We warmly encourage the reader to study Bernardy et al. (2010) in order to understand the subject in greater depth.

The theorems obtained by instantiating the parametricity hypothesis with a specific type τ are known as "free theorems" (Wadler, 1989) because they allows us to establish a property of a term `M` of type τ without requiring us to reason about the definition of `M`. Usually, this property is non-trivial only if the type τ involves polymorphism. Indeed, in this case, the statement of the "free theorem" begins with universal quantifiers that can be instantiated in useful ways. In our setting, polymorphism arises out of two distinct sources. First, because the types defined by our library are abstract, the client must be polymorphic with respect to these types. Hence, the free theorem about the client begins with a series of universal quantifiers which we can instantiate in a suitable manner. Second, when the client defines a world-polymorphic function, this particular function comes with a powerful "free theorem". Later on (section 11.3), we give a more detailed account to various function types and the strength of their associated "free theorems".

7.2 An example: Boolean values represented by numbers

Logical relations help understand in what sense the interface offered by an abstract type is safe, or in other words, in what sense the abstraction offered by the interface is independent of the underlying representation (Reynolds, 1983; Mitchell, 1986). In order to explain this,

```
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 4. Logical relations for inductive data types

we introduce a tiny example, where Boolean values are represented using natural numbers. We want 0 to represent `false` and any other number to represent `true`. Therefore, Boolean disjunction can be implemented using addition. We show that logical relations help build a "model" and ensure that an implementation respects this model. Then, parametricity can be used to show that a client that uses only the interface must also respect the model.

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 disjunction `_∨_`. Furthermore, it intentionally offers a dubious operation, `is42?`.

```
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 function `is42?` is intuitively not "well-behaved" because, even though the natural numbers 41 and 42 both encode the Boolean value `true`, this function maps them to distinct results. Thus, we wish to define a criterion that allows us to easily and formally

tell which operations are safe and which are not. To begin, we define a binary relation ⟦B⟧ over the type B. The idea is, two natural numbers are related if and only if they have the same meaning, that is, if and only if they encode the same truth value. This relation is an "invariant" which every operation must preserve. Technically, we define ⟦B⟧ as an inductive data type. Its definition states that 0 is related 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)
```

In this approach, one explicitly defines when two values mean "the same thing", but one does not explicitly define what that "thing" is. In this toy example, one could easily adopt a different (and simpler) approach, where one explicitly defines that 0 represents `false` and that any other number represents `true`. There would naturally follow that two numbers are equivalent if and only if they represent the same truth value. This works well because the inhabitants of our intended model, namely the Boolean values, have a canonical representation. In our real-world application, where the problem is to define α-equivalence at every type, it is easier to define when two terms are "equivalent" than it is to map every term to a canonical representative. This is why logical relations seem particularly natural and useful in our setting.

Defining ⟦B⟧ suffices to define the logical relation at every type. This defines what we view as "good behavior". If a piece of client code has type τ, we expect it to satisfy the free theorem ⟦ τ ⟧. The reader might wonder, however, why we are allowed to choose the definition of ⟦B⟧. After all, since B is internally defined as ℕ, mustn't we define ⟦B⟧ as ⟦ℕ⟧? If we define ⟦B⟧ in some other way, how do we know that the logical relation is still universal? To see why this makes sense, consider a client of the library. This client can be thought of as a function that expects an implementation of the library as an argument. Thus, this function is parameterized over the type B and over the operations `true`, `false`, `_∨_`, and `is42?`. In other words, this function is polymorphic in B. Thus, the "theorem for free" that comes with this function is universally quantified with respect to B *and with respect to a relation* ⟦B⟧. This explains why we may define the relation ⟦B⟧ however we please.

Naturally, we must still satisfy a few proof obligations. The "theorem for free" that comes with the client is further parameterized with the operations `true`, `false`, `_∨_`, `is42?` *and with proofs* ⟦`true`⟧, ⟦`false`⟧, `_`⟦∨⟧`_`, ⟦`is42?`⟧ *that each of these operations is well-behaved*. That is, we must prove that each of the operations offered by the library is related to itself.

The data constructors ⟦`true`⟧ and ⟦`false`⟧ are obvious witnesses to the fact that the operations `true` and `false` are well-behaved. The remains to check whether `_∨_` and `is42?` are well-behaved. Every time, the statement that must be proved is constructed in a systematic manner: if an operation has type τ, then one must check that its implementation

is related to itself by the relation ⟦ τ ⟧. For instance, here is the statement that must be proven about _∨_ :

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

Once unfolded, this statement looks like this:

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

This proposition states that the disjunction operation maps related arguments to related results. Now, thanks to the inductive definition of _+_, pattern-matching on the first relation argument suffices to cause the goal to reduce. Thus, we are able to offer the following nice-looking definition of _⟦∨⟧_, which one can recognize as the usual lazy definition of left-biased disjunction:

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

Let us now consider the question of the well-behavedness of the function `is42?`. Of course, there is no proof that this function is well-behaved. In fact, it is easy to prove that it is ill-behaved. It suffices to exhibit two related inputs, say 42 and 27, that are mapped to non-related outputs (we have `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 consideration of which other operations are exported. Once the relation ⟦B⟧ is defined, it suffices to "turn the crank" to find out that `is42?` is ill-behaved. This modularity is precious and has helped us easily determine which operations could or could not be offered as part of the NOMPA library.

7.3 *Relations for* NOMPA

For NOMPA, we apply the same process as in the toy example. We define our expectations by defining a relation for each of the abstract types that the library advertises. Then, we prove that each operation offered by the library is well-behaved with respect to the logical relation that arises out of these definitions.

7.3.1 *Relations for* NOMPA *types*

For reference, the definitions are given in figure 5. We now describe them in turn.

The first thing to do is to define the constant ⟦World⟧. What does it mean for two worlds to be related? It is useful to think of the manner in which ⟦Set$_0$⟧ is defined. Recall that it is defined by the equation ⟦Set$_0$⟧ A_1 A_2 = A_1 → A_2 → Set$_0$. This means that the "theorem for free" that describes a polymorphic object is universally quantified over two types A_1 and A_2 and over a relation A_r between these types. Now, we would like to define ⟦World⟧ in such a way that, similarly, the "theorem for free" that describes a world-polymorphic object is universally quantified over two worlds α_1 and α_2 and over a relation between these worlds.

Thus, it seems that we could perhaps let ⟦World⟧ α_1 α_2 be Name α_1 → Name α_2 → Set$_0$, that is, the set of all relations between (the sets of names denoted by) α_1 and α_2. However, if we adopted such a definition, we would later be unable to prove that the name comparison operation _==^{N}_ is well-behaved. Because this operation is world-polymorphic, we will have to prove that, for all worlds α_1 and α_2 and for every relation α_r of type ⟦World⟧ α_1 α_2, this operation maps α_r-related arguments to *equal* results. (Indeed, the relation ⟦Bool⟧ is just equality.) If α_r was allowed to range over arbitrary relations, it would follow that _==^{N}_ must be a constant function. Thus, we must restrict the set of allowable relations. It is clear that the equality test _==^{N}_ is well-behaved if and only if every relation in ⟦World⟧ α_1 α_2 preserves equality in both directions, i.e., is functional and injective:

```
Preserve-≡ ℛ =
  ∀ x₁ y₁ x₂ y₂ → ℛ x₁ x₂ → ℛ y₁ y₂
                → x₁ ≡ y₁ ↔ x₂ ≡ y₂
```

We allow ⟦World⟧ α_1 α_2 to contain all such relations. Thus, the "theorem for free" that describes a world-polymorphic object will be universally quantified over two worlds and over a functional and injective relation between them. In other words, we choose the definition of ⟦World⟧ that leads to the strongest possible "free theorems", under the requirement that _==^{N}_ be well-behaved. It turns out that, with this definition, we will be able to prove that every other operation is well-behaved too.

Because the type Name is parameterized with a world, the relation ⟦Name⟧ is parameterized with a relation between worlds. It is defined as ⟦Name⟧ the identity: two names are related by ⟦Name α_r⟧ if and only if they are related by α_r.

The definition of the relation ⟦Binder⟧ is surprisingly simple: it is the full relation. Thus, every two binders are related by ⟦Binder⟧. Thus, a function that allows distinguishing between two binders is considered ill-behaved. This is consistent with our goal of disallowing functions that distinguish between two α-equivalent representations of a λ-term.

A consequence of this definition is that the library cannot provide an equality test over binders. Indeed, as demonstrated by the following theorem, if `f` is a function of type `Binder → Binder → Bool`, then it must be a constant function.

```
-- ⟦f⟧ is the parametricity theorem for f
⟦f⟧ : (⟦Binder⟧ ⟦→⟧ ⟦Binder⟧ ⟦→⟧ ⟦Bool⟧) f f

-- f-const is a corollary of ⟦f⟧.
-- f-const shows that f is a constant function.
f-const : ∀ x₁ x₂ y₁ y₂ → f x₁ y₁ ≡ f x₂ y₂
f-const x₁ x₂ y₁ y₂ with ⟦f⟧ {x₁} {x₂} _ {y₁} {y₂} _
... | ⟦true⟧  = refl
... | ⟦false⟧ = refl
```

Next, we define the relation $\_[\![\subseteq]\!]\_$. Again, we wish to adopt the most liberal definition with respect to which we can prove that the operations offered by the library are well-behaved. For this purpose, we exploit the fact there is ultimately only one way to use an inclusion witness, which is to pass it as an argument to the operation $\mathtt{coerce}^{\mathtt{N}}$. Thus, we posit that two world inclusion witnesses $\alpha_1 \subseteq \beta_1$ and $\alpha_2 \subseteq \beta_2$ are related if and only if $\mathtt{coerce}^{\mathtt{N}}\ \alpha_1 \subseteq \beta_1$ and $\mathtt{coerce}^{\mathtt{N}}\ \alpha_2 \subseteq \beta_2$ are related (see figure 5). Although this definition is arguably quite elegant, it may seem somewhat "magic" and opaque. Fortunately, one can formulate an equivalent definition in terms of inclusion of relations. Let us say that a relation $\mathscr{R}_1$ is a subset of a relation $\mathscr{R}_2$ if and only if every pair that is related by $\mathscr{R}_1$ is related by $\mathscr{R}_2$ as well. Then, two relations α_r and β_r are related by $\_[\![\subseteq]\!]\_$ if and only if α_r is a subset of β_r. Indeed, if one considers the definition of $\_[\![\subseteq]\!]\_$ in figure 5 and expands the definitions of $[\![\mathtt{Name}]\!]$ and $\_[\![\rightarrow]\!]\_$, one finds:

$$
\begin{array}{l}
\_[\![\subseteq]\!]\_\ \alpha_r\ \beta_r\ \alpha_1 \subseteq \beta_1\ \alpha_2 \subseteq \beta_2 \\
\quad = \forall\ \{\mathtt{x}_1\ \mathtt{x}_2\} \rightarrow (\mathtt{x}_1\ ,\ \mathtt{x}_2) \in \alpha_r \\
\qquad\qquad \rightarrow (\mathtt{coerce}^{\mathtt{N}}\ \alpha_1 \subseteq \beta_1\ \mathtt{x}_1\ ,\ \mathtt{coerce}^{\mathtt{N}}\ \alpha_2 \subseteq \beta_2\ \mathtt{x}_2) \in \beta_r
\end{array}
$$

Because the function $\mathtt{coerce}^{\mathtt{N}}$ behaves (after erasure) as the identity function, the right-hand side of the above equation informally means that the relation α_r is a subset of the relation β_r.

We now define $[\![\emptyset]\!]$ and $\_[\![\triangleleft]\!]\_$. Whereas $\emptyset$ is a world, $[\![\emptyset]\!]$ is a relation between the empty world and itself. We have no choice: there is only one such relation, namely the empty relation. We show its type and omit its definition:

```
⟦∅⟧ : ⟦World⟧ ∅ ∅
```

Whereas $\_\triangleleft\_$ maps a binder and a world to a new world, $\_[\![\triangleleft]\!]\_$ maps a pair of binders and a relation between worlds to a new relation between worlds. In other words, it extends an existing relation between worlds, say α_r, with a new pair of binders, say $\mathtt{b}_r$. How should we define $\_[\![\triangleleft]\!]\_$? An idea that naturally comes to mind is to construct the set-theoretic union of the relation α_r and of the singleton set $\{\mathtt{b}_r\}$. However, this would not make

```
Preserve-≡ : {A B : Set₀} (ℛ : A → B → Set₀) → Set₀
Preserve-≡ ℛ =
  ∀ x₁ y₁ x₂ y₂ → ℛ x₁ x₂ → ℛ y₁ y₂
                → x₁ ≡ y₁ ↔ x₂ ≡ y₂

--     ⟦World⟧ : ⟦Set₁⟧ World World
record ⟦World⟧ (α₁ α₂ : World) : Set₁ where
  constructor _,_
  field
    ℛ         : Name α₁ → Name α₂ → Set
    ℛ-pres-≡ : Preserve-≡ ℛ

⟦Name⟧ : (⟦World⟧ ⟦→⟧ ⟦Set₀⟧) Name Name
    -- : ∀ {α₁ α₂} → ⟦World⟧ α₁ α₂ → Name α₁ → Name α₂ → Set
⟦Name⟧ (ℛ , _) x₁ x₂ = ℛ x₁ x₂

⟦Binder⟧ : ⟦Set₀⟧ Binder Binder
      -- : Binder → Binder → Set
⟦Binder⟧ _ _ = ⊤

⟦∅⟧ : ⟦World⟧ ∅ ∅
⟦∅⟧ = (λ _ _ → ⊥) , (λ())

_⟦◁⟧_ : (⟦Binder⟧ ⟦→⟧ ⟦World⟧ ⟦→⟧ ⟦World⟧) _◁_ _◁_
        -- not proper Agda
b_r ⟦◁⟧ α_r ≝ { (b₁, b₂) } ∪ { (x, y) | (x, y) ∈ α_r ∧ x ≢ b₁ ∧ y ≢ b₂ }

_⟦#⟧_ : (⟦Binder⟧ ⟦→⟧ ⟦World⟧ ⟦→⟧ ⟦Set₀⟧) _#_ _#_
  -- : ∀ {b₁ b₂} → ⟦Binder⟧ b₁ b₂ → ∀ {α₁ α₂} → ⟦World⟧ α₁ α₂
  --   → b₁ # α₁ → b₂ # α₂ → Set
_⟦#⟧_ _ _ _ _ = ⊤

_⟦⊆⟧_ : (⟦World⟧ ⟦→⟧ ⟦World⟧ ⟦→⟧ ⟦Set₀⟧)
        _⊆_ _⊆_
  -- : ∀ {α₁ α₂} → ⟦World⟧ α₁ α₂ →
  --   ∀ {β₁ β₂} → ⟦World⟧ β₁ β₂ →
  --     α₁ ⊆ β₁ → α₂ ⊆ β₂ → Set
_⟦⊆⟧_ α_r β_r α₁⊆β₁ α₂⊆β₂
  = (⟦Name⟧ α_r ⟦→⟧ ⟦Name⟧ β_r) (coerce^N α₁⊆β₁) (coerce^N α₂⊆β₂)
```

Figure 5. Relations for NOMPA types

sense: we must be careful to ensure that the resulting relation is functional and injective. If the first component of the pair b_r is already a member of the domain of the relation α_r, or (symmetrically) if the second component of b_r is already a member of the codomain of α_r, then the set-theoretic union of α_r and $\{b_r\}$ might not be functional and injective. This corresponds to a situation where a new binder "shadows" a previous one. In that case, we would like the new pair b_r to take precedence over any earlier bindings. Thus, the definition that we ultimately adopt can be described as the set-theoretic union of the

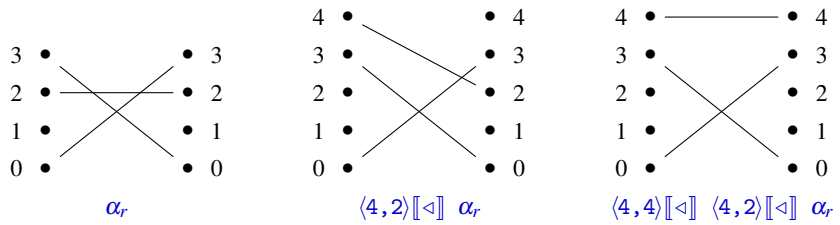

Figure 6. The effect of $\_[\![\triangleleft]\!]\_$ on relations

relation α_r, *deprived of any bindings that conflict with* $\mathtt{b}_r$, and of the singleton set $\{\mathtt{b}_r\}$. It can be informally defined as follows:

```
_[[◁]]_ : ([[Binder]] [[→]] [[World]] [[→]] [[World]]) _◁_ _◁_
          -- not proper Agda
b_r [[◁]] α_r ≝ { (b1, b2) } ∪ { (x, y) | (x, y) ∈ α_r ∧ x ≢ b1 ∧ y ≢ b2 }
```

The effect of $\_[\![\triangleleft]\!]\_$ is illustrated in figure 6.

7.3.2 NOMPA *values fit the relation*

We now give a short overview of the proofs needed to show that the operations offered by the library fit the relation. Formally, for each operation `p` of type τ that appears in the interface of the library, we have to exhibit a proof $[\![\mathtt{p}]\!]$ of the statement $[\![\ \tau\ ]\!]\ \mathtt{p}\ \mathtt{p}$. All proofs can be found online (Pouillard, 2011b).

Many of these proofs are immediate. For instance, because the relation $[\![\mathtt{Binder}]\!]$ is the full relation, any operation whose return type is `Binder` is well-behaved. For some operations, the proof is just a matter of expanding the definitions. For instance, in the case of the operation $[\![\mathtt{name}^{\mathtt{B}}]\!]$, which converts a binder to a name, the statement that must be proved is the following:

```
[[name^B]] : (∀⟨ α_r : [[World]] ⟩[[→]]
                ⟨ b_r : [[Binder]] ⟩[[→]]
                [[Name]] (b_r [[◁]] α_r)
              ) name^B name^B
--         : ∀ {α1 α2} (α_r : [[World]] α1 α2)
--             {b1 b2} (b_r : [[Binder]] b1 b2)
--           → [[Name]] (b_r [[◁]] α_r) (name^B {α1} b1) (name^B {α2} b2)
```

That is, roughly speaking, we must prove that the names $\mathtt{b}_1$ and $\mathtt{b}_2$ are related by the relation $\mathtt{b}_r\ [\![\triangleleft]\!]\ \alpha_r$. This follows immediately from the definition of $\_[\![\triangleleft]\!]\_$, since the effect of this operation is precisely to extend the relation α_r with the pair $(\mathtt{b}_1\ ,\ \mathtt{b}_2)$.

In the case of the equality test $\_[\![==^{\mathbb{N}}]\!]\_$, once unfolded, the statement requires that the equality test commute with a renaming. In other words, the outcome of an equality test must not change when its inputs are consistently renamed.

```
_[[==ᴺ]]_ : (∀⟨ αᵣ : [[World]] ⟩[[→]]
              [[Name]] αᵣ [[→]]
              [[Name]] αᵣ [[→]]
              [[Bool]]
            ) _==ᴺ_ _==ᴺ_
--        : ∀ {α₁ α₂} (αᵣ : [[World]] α₁ α₂)
--            {x₁ x₂} (xᵣ : [[Name]] αᵣ x₁ x₂)
--            {y₁ y₂} (yᵣ : [[Name]] αᵣ y₁ y₂)
--          → [[Bool]] (x₁ ==ᴺ y₁) (x₂ ==ᴺ y₂)
```

The proof is in two parts. First, we prove that the Boolean-valued function $\_==^{\mathbb{N}}\_$ decides propositional equality on names. Second, we exploit the fact that the relation α_r preserves equality, that is, α_r is functional and injective.

The proof that `export`$^{\mathbb{N}}$`?` is in the relation relies on two points. First, the success of `(` `export`$^{\mathbb{N}}$`?` `{b}` `x` `)` (that is, whether it returns `just` or `nothing`) depends only on the equality between `(` `name`$^{\mathbb{B}}$ `b` `)` and `x`. Second, every pair in the relation `(` $\mathtt{b}_r$ $[\![\triangleleft]\!]$ α_r `)` is either in the relation `(` $\mathtt{b}_r$ $[\![\triangleleft]\!]$ $[\![\emptyset]\!]$ `)` or in the relation α_r. This follows from our definition of $\_[\![\triangleleft]\!]\_$.

Thanks to the definition of $\_[\![\subseteq]\!]\_$, the proof that `coerce`$^{\mathbb{N}}$ is well-behaved is immediate. There remains to show that each of the world inclusion rules (figure 1) is well-behaved. This can be done informally by a simple inspection of these rules, while keeping in mind that a world α must now interpreted as a relation between worlds and world inclusion $\_\subseteq\_$ must now be interpreted as inclusion of relations. In the last rule, `⊆-#`, we can now see why the freshness hypothesis `b # α` is required. Indeed, the goal is to prove that the relation α_r is a subset of the relation `(` $\mathtt{b}_r$ $[\![\triangleleft]\!]$ α_r `)`. In the absence of any hypothesis about $\mathtt{b}_r$ and α_r, this is false, because the operation $\_[\![\triangleleft]\!]\_$ can deprives α_r of certain pairs if there is shadowing. In the presence of the above freshness hypothesis, we can further assume that $\mathtt{b}_1$ is not in the domain of α_r and that $\mathtt{b}_2$ is not in the codomain of α_r. In this case, α_r is indeed a subset of `(` $\mathtt{b}_r$ $[\![\triangleleft]\!]$ α_r `)`.

7.3.3 An example of an ill-behaved operation

Consider the following function, which exposes a total ordering on names:

```
_<=ᴺ_ : ∀ {α} → Name α → Name α → Bool
(m , _) <=ᴺ (n , _) = ℕ._<=_ m n
```

This operation is well-typed. Yet, it is ill-behaved, because its outcome *can* change when its inputs are consistently renamed.

```
¬⟦<=ᴺ⟧ : ¬((∀⟨ αᵣ : ⟦World⟧ ⟩⟦→⟧
                    ⟦Name⟧ αᵣ ⟦→⟧
                    ⟦Name⟧ αᵣ ⟦→⟧ ⟦Bool⟧) _<=ᴺ_ _<=ᴺ_)
--      : ¬(∀ {α₁ α₂} (αᵣ : ⟦World⟧ α₁ α₂)
--            {x₁ x₂} (xᵣ : ⟦Name⟧ αᵣ x₁ x₂)
--            {y₁ y₂} (yᵣ : ⟦Name⟧ αᵣ y₁ y₂)
--          → ⟦Bool⟧ (x₁ <=ᴺ y₁) (x₂ <=ᴺ y₂))
¬⟦<=ᴺ⟧ ⟦<=⟧ = ¬⟦Bool⟧-true-false (⟦<=⟧ ? {0 , _} {1 , _} ?
                                       {1 , _} {0 , _} ?)
  -- parts (‘?’) of the proof are omitted for conciseness
```

This explains why, even though this operation would be pragmatically useful in certain circumstances (see section 5.2), it cannot be made part of the library.

7.4 What does this mean?

We have formally proved that every operation offered by the library is well-behaved with respect to the logical relation (Pouillard, 2011b). By combining this fact with the parametricity hypothesis for AGDA, there follows that every well-typed client of our library is also well-behaved with respect to our logical relation.

What is the impact of the result? In order to find out, one must examine the theorem that comes “for free” (Wadler, 1989) with clients of various type.

Consider, for instance, a client of type `Tm ∅ → Bool`, that is, a function `f` that maps a closed λ-term to a Boolean result. The “free theorem” guarantees that this function maps $(\llbracket \mathtt{Tm} \rrbracket\ \llbracket \emptyset \rrbracket)$-related terms to $\llbracket \mathtt{Bool} \rrbracket$-related results. In the first conference paper (Pouillard & Pottier, 2010), we informally prove that the relation $\llbracket \mathtt{Tm} \rrbracket\ \llbracket \emptyset \rrbracket$ coincides with our intuitive notion of α-equivalence of λ-terms. (We do not produce a machine-checked version of this proof, although it would admittedly be desirable to do so.) Furthermore, the relation $\llbracket \mathtt{Bool} \rrbracket$ is equality. Thus, the “free theorem” guarantees that the function `f` maps two α-equivalent terms to the same result. In short, a well-typed client cannot distinguish two α-equivalent (closed) terms.

There are other types that come with interesting “free theorems”. For instance, consider a world-polymorphic and homogeneous term transformer, that is, a client function `f` of type $\forall\ \{\alpha\} \to \mathtt{Tm}\ \alpha \to \mathtt{Tm}\ \alpha$. The “free theorem” associated with this type guarantees that, for every relation α_r, this function maps $\llbracket \mathtt{Tm} \rrbracket\ \alpha_r$-related arguments to $\llbracket \mathtt{Tm} \rrbracket\ \alpha_r$-related results. Furthermore, one can prove that two terms are related by $\llbracket \mathtt{Tm} \rrbracket\ \alpha_r$ if and only if their free names are related by α_r and these terms are otherwise α-equivalent. Thus, we find that `f` must commute with an arbitrary renaming of the free names. In other words, `f` is equivariant (Pitts, 2006).

We do not formally prove the above claims. After we extend the library with support for de Bruijn indices, we come back to this issue and formally prove (with respect to a de

Bruijn representation of terms) that world-polymorphic homogeneous term transformers commute with renamings (section 11.2).

8 Introduction to de Bruijn indices

So far, we have focused on the representation of object-level terms in nominal style, and we have presented only a fragment of the library, which is sufficient to support this style. This way, we hope to avoid some of the confusion that could have resulted if we had decided to deal at once with multiple representation styles. We now focus on the representation of syntax with names and binders via de Bruijn indices (de Bruijn, 1972). This representation is sometimes referred to as "nameless" because binders disappear completely, while names are no longer represented by atoms but by their "distance" to the point where they were bound. However, in spite of the important differences that exist between the nominal representation and de Bruijn's representation, the two representations ultimately also exhibit several common points. For instance, in both representations, names are just natural numbers. Also, in both settings, we wish to exclude certain ill-behaved pieces of code. In the de Bruijn setting, for instance, we would like to reject a piece of code where the programmer omits to "shift" a de Bruijn index. Quite surprisingly perhaps, it turns out that, in order to support de Bruijn indices, we do not need to throw away any of the infrastructure that we have constructed up to this point. All we need to do is *extend* the library's interface, implementation, and soundness proof with a small number of new types and operations.

In the remainder of this section, we informally introduce de Bruijn indices and go through several variations on "well-typed de Bruijn indices" that exist in the literature. In the sections that follow, we extend the library's interface and implementation (section 9). We explain how to use the library so as to define and work with de Bruijn representations (section 10). Finally, we extend the logical relation so as demonstrate the soundness of the extended library, and we study the meaning of some of the "free theorems" that can be obtained about client code (section 11).

8.1 The bare approach

The "bare" approach relies solely on natural numbers. To make things concrete, here is an example of its use. This is our running example, namely a representation of the terms of the untyped λ-calculus:

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

From the point of view of the binding structure, it is quite striking that there is no visible difference between the constructors of this data type. It is completely up to the programmer to manage the fact that `ƛ` and `Let` introduce a new variable. This is particularly worrying

in the case of `Let`, where we have no clue that there should be a difference of treatment between the arguments.

Here is how one might build the λ-term for function application, namely $\lambda f.\lambda x.f\ x$.

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

The main advantages of the "bare" approach are its simplicity and its expressiveness. Expressiveness is in a sense maximal, since no restriction is put on the usage of variables.

8.2 The `Maybe` approach

The `Maybe` approach, also known as the nested data type approach (Bellegarde & Hook, 1994; Bird & Paterson, 1999; Altenkirch & Reus, 1999), is a first step towards better describing the binding structure of terms, and enforcing stronger guarantees about code that manipulates terms. Let us start with the definition of the type of λ-terms with this approach:

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

Three points must be noticed. First, the type `Tm^M` is parameterized with a type `A`, so one can look at it as a kind of container. Second, the data constructor `V` does not carry a raw de Bruijn index, which would have type $\mathbb{N}$, but a value of type `A`. Last, but not least, the data constructor `ƛ` carries a term whose index is not `A` but `Maybe A`.

The last point makes the type `Tm^M` a nested data type, also known as a non-regular data type. This has the consequence that polymorphic recursion is required in order to define recursive functions over such a type.

To understand why this is an adequate representation of λ-terms, let us examine the meaning of `Maybe`. If types are viewed as sets of values, then the type transformer `Maybe` can be viewed as a function that takes a set and produces a set with one extra element. Thus, each time we cross a `ƛ`, we add one element to the set of variables that can be referred to. This captures the fact that we are introducing a variable.

To better see the difference with the previous approach, let us look again the λ-term for function application:

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

The use of the empty type $\bot$ reflects the fact that the term `appTm^M` is closed. Stating such a property as part of the type was impossible in the bare approach; it would have to be stated as a separate assertion.

8.3 The `Fin` approach

Another approach, which has been described and used by several authors (Altenkirch, 1993; McBride & McKinna, 2004), is to index the type of terms with a natural number. This number represents a bound on the free variables that are allowed to occur in a term.

This approach relies on the type `Fin n`, whose definition, found in AGDA's 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)
```

The type `Fin n` is isomorphic to the set of the natural numbers that are less than `n`. More generally, it is isomorphic to every set of exactly `n` elements. This explains the name `Fin`, which stands for "finite".

In this approach, the inductive type of λ-terms is defined as follows:

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

The data constructor `V` carries a de Bruijn index, and requires that this index be less than `n`. The data constructor `ƛ`, which binds a new variable, increments the index `n`, thus making one more index available.

Like the `Maybe` approach, this representation helps enforce certain well-formedness properties. For instance, `Tmᶠ 0` is the type of closed λ-terms.

Here is the λ-term for application in this approach:

```
appTmᶠ : Tmᶠ 0
appTmᶠ = ƛ (ƛ (V (suc zero) · V zero))
```

One can easily recognize a similarity between the `Maybe` and `Fin` approaches. Indeed, the type `Fin (suc n)` has exactly one more element than the type `Fin n`. However, these approaches are not equivalent, for at least two reasons. The `Maybe` approach can use any type `A` to represent the free variables of a term. This allows terms to be viewed as containers, and allows defining substitution as the composition of `mapTm : ∀{A B} → (A → B) → Tmᴹ A → Tmᴹ B` and `joinTm : ∀{A} → Tmᴹ (Tmᴹ A) → Tmᴹ A` (Bellegarde & Hook, 1994; Bird & Paterson, 1999; Altenkirch & Reus, 1999). The `Fin` approach has the advantage of being more concrete, hence simpler. However, this apparent simplicity comes at a cost. In the `Maybe` approach, a function that examines a term of type `Tmᴹ A` and that is polymorphic in `A` comes with a strong "free theorem", which guarantees that this function views the free variables as abstract, so that (for instance) a free variable and a bound variable cannot be mistakenly compared. In the `Fin` approach, in contrast, all

```
-- Constructing worlds
_+1 : World → World

_↑1 : World → World
α ↑1 = 0 B◁ (α +1)

-- Name arithmetic
add^N      : ∀ {α} k  → Name α
                      → Name (α +^W k)
subtract^N : ∀ {α} k  → Name (α +^W k)
                      → Name α
cmp^N      : ∀ {α} ℓ  → Name (α ↑ ℓ)
                      → Name (∅ ↑ ℓ) ⊎ Name (α +^W ℓ)

syntax add^N      k x = x +^N k
syntax subtract^N k x = x ∸^N k
syntax cmp^N      ℓ x = x <^N ℓ

-- World inclusion
⊆-∅+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 7. The NOMPA interface (de Bruijn fragment)

variables are represented as integer indices, regardless of whether they are free or bound, so that confusion is possible.

9 The NOMPA interface and implementation (de Bruijn fragment)

We wish to offer an approach to "well-typed de Bruijn indices" that exploits parametricity to offer strong guarantees of well-behavedness, like the `Maybe` approach, and at the same time offers the ability to work with natural integer indices and to perform low-level arithmetic operations, like the `Fin` approach.

It turns out that we can do this by *extending* the library that we have presented so far with support for arithmetic operations over worlds and over names. We need not throw anything away. This is in contrast with our first conference paper (Pouillard & Pottier, 2010), where we presented two distinct implementations of a common interface. Here, there is no need to make a monolithic choice between the nominal style and de Bruijn's style. The library simultaneously supports both styles.

Thus, everything that we have built up to this point remains valid and useful. The types and operations of figure 4 remain available to the client, and their implementation is unchanged. The new types and operations in figure 7 come in addition to those of figure 4.

9.1 New operations on worlds

What must we add to the library in order to enable working with de Bruijn indices? First and foremost, we must be able to describe how the current world is affected by the introduction of a new name. In the nominal setting, we introduced the operation `_◁_` for this purpose. If one thinks of worlds as sets of names, then `b ◁ α` is the set obtained by adding the binder `b` to the set `α`. In the de Bruijn setting, things are different. At a binding construct, such as λ, two things occur: the name 0 is introduced and considered a "new" name, while all previous names are incremented by one. Thus, the key feature of this approach is that descending under a new binder changes the manner in which previous binders are referred to.

In order to account for this phenomenon, we introduce a new operation on worlds, written `_+1` (figure 7). It is a translation operation: if `α` is viewed as a set of names, then `α +1` is the set obtained by adding one to each element of `α`.

This single extension of the library allows us to build name abstractions in de Bruijn style. Here is our running example, a representation of the syntax of the untyped λ-calculus, this time in de Bruijn style:

```
data Tm^D α : Set where
  V   : Name α → Tm^D α
  _·_ : Tm^D α → Tm^D α → Tm^D α
  ƛ   : Tm^D (0 ^B ◁ (α +1)) → Tm^D α
  Let : Tm^D α → Tm^D (0 ^B ◁ (α +1)) → Tm^D α
```

Just as in the nominal setting, the type of terms is indexed with a world. The only difference between `Tm` and `Tm^D` is in the representation of the binding constructs. In the nominal version, the data constructor `ƛ` carries a binder `b` and a subterm where `b` is considered bound. Here, in contrast, `ƛ` carries just a subterm, where the binder `0 ^B` is considered bound, and where all previous binders are translated up by one.

Because the operation of incrementing by one and introducing `0 ^B` is idiomatic, it deserves a concise notation, which we now define (see also figure 7):

```
_↑1 : World → World
α ↑1 = 0 ^B ◁ (α +1)
```

We refer to the operation `_↑1` as "shifting" a world. The inductive definition of the type `Tm^D` can now be reformulated in a more concise and transparent manner:

```
data Tm^D α : Set where
  V   : Name α → Tm^D α
  _·_ : Tm^D α → Tm^D α → Tm^D α
  ƛ   : Tm^D (α ↑1) → Tm^D α
  Let : Tm^D α → Tm^D (α ↑1) → Tm^D α
```

If one replaces `World` by `Set`, `Name` by the identity, and `_↑1` by `Maybe`, then one recovers the nested data type approach described in section 8.2. Similarly, if one replaces `World` by $\mathbb{N}$, `Name` by `Fin`, and `_↑1` by `suc`, then one recovers the `Fin` approach of section 8.3. In our approach, `World`, `Name`, and `_↑1` are abstract, which allows us to obtain stronger "free theorems". Nevertheless, as we will see very soon (section 9.2), we are still able to offer low-level arithmetic operations on names.

Our internal representation of worlds as lists of Boolean values (section 6) makes it easy to implement `_+1`.

```
_+1 : World → World
α +1 = false :: α
```

The operation `_↑1` was defined above in terms of `_+1` and `_◁_`. If we expand this definition, we find that α `↑1` is just `true :: α`.

The definitions of the "one-step" operations `_+1` and `_↑1` are extended to "any number of steps" to produce the operations `_+ᵂ_` and `_↑_`, which have type `World → ℕ → World`. The world α `+ᵂ k` is α `+1...+1`, where `_+1` is iterated `k` times. The world α `↑ k` is α `↑1...↑1`, where `_↑1` is iterated `k` times.

To the best of our knowledge, the distinction between translation (`_+1`) and shifting (`_↑1`) at the level of worlds has never been investigated. We argue in section 11.3 that this distinction can be important. By exploiting (`_+1`) instead of (`_↑1`) where appropriate, one can express more precise types, with which stronger "free theorems" are associated. Conversely, if one uses (`_↑1`) everywhere, one must declare coarser types, and one runs a greater risk that intuitively incorrect code is considered well-typed.

9.2 New operations on names

We extend the library with a small number of new operations on names. These operations allow performing low-level arithmetic on de Bruijn indices. They receive abstract types that describe their effect in terms of worlds.

We need to be able to construct the name 0. The nominal fragment of the library offers the constant `zeroᴮ`, which has type `Binder`. When working in de Bruijn style, we never use the type `Binder`, so we prefer to define a constant `zeroᴺ` whose type is an instance of `Name`. This constant inhabits every world that has been shifted by one. It can be defined in terms of the existing interface, so it does not technically represent an extension of the library:

```
zeroᴺ : ∀ {α} → Name (α ↑1)
zeroᴺ = nameᴮ (0 ᴮ)
```

We introduce three new operations on names, whose signatures are given in figure 7. These operations are addition, subtraction, and comparison.

One can add a constant to a name using `addᴺ` and perform the opposite operation using `subtractᴺ`. Thanks to the world translation operation `_+ᵂ_` , the types assigned to these operations are as precise as possible. Because the domain of the subtraction operation is

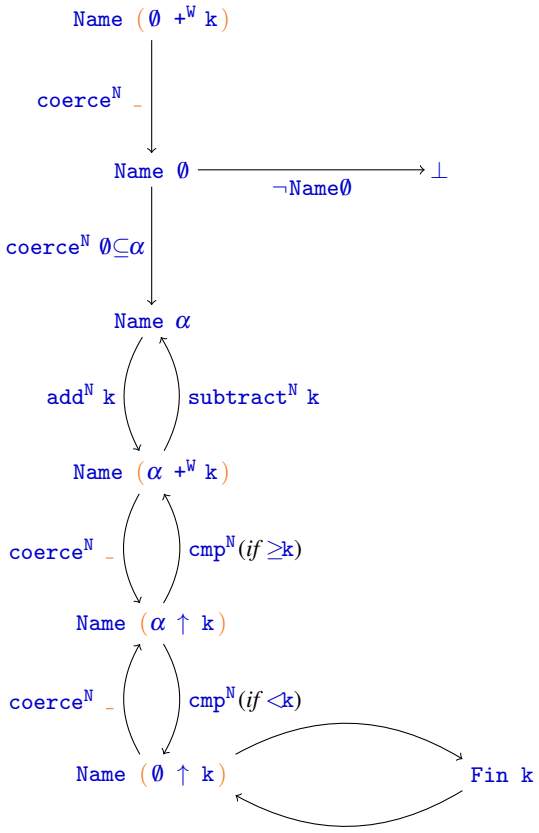

Figure 8. The typical types of names

Name (α +W k), this operation is a total function, and is the inverse of addN. Thus, the types Name α and Name (α +W k) are effectively isomorphic.

When one writes a recursive function that descends into a term in de Bruijn style, one usually carries a natural number ℓ that represents the number of binders that have been entered. In such a situation, a de Bruijn index that is strictly less than ℓ refers to one of the binders that was entered on the way down: we say that such a name is "ℓ-bound". A de Bruijn index that is greater than or equal to ℓ refers to a free name of the term that is being examined: we say that such a name is "ℓ-free". It is necessary to be able to distinguish between these cases. For this purpose, we introduce the comparison operation cmpN. This operation compares a natural number ℓ and a name x. The name x is assumed to inhabit the world α ↑ ℓ. This world is the disjoint union of the world ∅ ↑ ℓ, which represents the entire interval [0, ℓ), and of the world α +W ℓ, which represents a certain set of names all of which are greater than or equal to ℓ. The codomain of cmpN reflects this disjoint union. The comparison cmpN ℓ x produces a disjoint sum whose tag ($\mathtt{inj}_1$ or $\mathtt{inj}_2$) encodes the Boolean outcome of the comparison and whose payload is the name x at a refined type. Like exportN, this operation combines a dynamic test and a static type refinement operation.

Figure 8 depicts several types of names and how our operations relate them. Let us begin at the bottom. A name of type `Name (∅ ↑ k)` is definitely `k`-bound: it is a member of the interval [0, k). The type `Fin k` also represents this interval: `Name (∅ ↑ k)` and `Fin k` are isomorphic types. Higher up, we find a type `Name (α ↑ k)` of names that may be `k`-bound or `k`-free. A dynamic test (permitted by the operation $\texttt{cmp}^{\texttt{N}}$) allows telling whether such a name is `k`-bound or `k`-free, and (via a type refinement) sends this name into either `Name (∅ ↑ k)`, one level down, or `Name (α +W k)`, one level up. A name of type `Name (α +W k)` is definitely `k`-free. As noted earlier, this type is isomorphic to `Name α`. Finally, in the top part of the diagram, we find empty types. The type `Name ∅` is empty, so it is a subset of `Name α` and more generally of the empty type $\bot$. The type `Name (∅ +W k)` is also empty, because the world inclusion rule `⊆-∅+1` (to be introduced shortly) states that the world `∅ +1` is empty.

The isomorphism between `Fin n` and `Name (∅ ↑ n)` means that every program that uses the `Fin` approach can be translated into our system. This means that our approach is at least as expressive as the `Fin` approach. It also means that our approach is not inherently safer than the `Fin` approach. In our approach, stronger guarantees than in the `Fin` approach can be obtained if one decides to use more precise types than `Name (∅ ↑ n)`. We come back to this point in section 11.3.

9.3 New world inclusion rules

We extend the world inclusion relation with a set of new rules, which appear in figure 7. These rules state that the world `∅ +1` is empty, that the operations `_↑1` and `_+1` preserve world inclusion (both ways), and that `_+1` can be weakened to `_↑1`. The last rule accounts for the fact that `α ↑1` is the disjoint union of the singleton set $\{0\}$ and of the world `(α +1)` and is thus a proper superset of `α +1`.

It is worth noting that the candidate rule `α ⊆ α ↑1` is not valid. This follows from the fact that α denotes an arbitrary set of names, as opposed to, say, an interval of the form [0, n). As a result, in a situation where α is an abstract world, the only way of transporting a name from `Name α` to `Name (α ↑1)` is to add one to it. In contrast, in the `Fin` approach, the type `Fin n` is a subset of the type `Fin (n + 1)`, so there are two ways of transporting a name from `Fin n` to `Fin (n + 1)`: one is the identity, the other is the addition of one. It is in principle possible to mistakenly use the former in place of the latter: this amounts to "omitting to shift" a de Bruijn index. In our approach, provided the programmer was wise enough to assign a world-polymorphic type to the function where the mistake lies, this results in an ill-typed program.

10 Programming on top of NOMPA (de Bruijn fragment)

We now illustrate how the library allows working with terms in de Bruijn style. We show how to build terms, how to write recursive functions over terms, and how to build a generic term traversal function whose instances include shifting and capture-avoiding substitution.

10.1 Some convenience functions

A number of useful functions can be built on top of the interface, that is, without access to the library's internal details. In practice, these functions can be shipped with the library, as part of an "external library layer".

The function $\mathtt{suc}^{\mathtt{N}}$ is just $\mathtt{add}^{\mathtt{N}}\ 1$. The function $\mathtt{suc}^{\mathtt{N}}\!\uparrow$ is a variant of $\mathtt{suc}^{\mathtt{N}}$ that includes a coercion from $\alpha\ +1$ to $\alpha\ \uparrow 1$. The function $\mathtt{add}^{\mathtt{N}}\!\uparrow$ is a variant of $\mathtt{add}^{\mathtt{N}}$ that includes a coercion from $\alpha\ +^{\mathtt{W}}\ \mathtt{k}$ to $\alpha\ \uparrow\ \mathtt{k}$.

```
suc^N : ∀ {α} → Name α → Name (α +1)
suc^N = add^N 1

suc^N↑ : ∀ {α} → Name α → Name (α ↑1)
suc^N↑ = coerce^N ⊆-+1↑1 ∘ suc^N

add^N↑ : ∀ {α} ℓ → Name α → Name (α ↑ ℓ)
add^N↑ ℓ = coerce^N (⊆-+-↑ ℓ) ∘ add^N ℓ
```

The function $\_^{\mathtt{N}}$ turns a natural number, say $\mathtt{n}$, into a name. This name inhabits any world that has been shifted at least $\mathtt{n}\ +\ 1$ times.

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

Like `cmp`, the function $\mathtt{subtract}^{\mathtt{N}}?$ tests whether a name $\mathtt{x}$ is ℓ-bound or ℓ-free. If $\mathtt{x}$ is ℓ-bound, the operation fails; otherwise, it returns the name $\mathtt{x}\ -\ \ell$, which can be considered as "a version of the name $\mathtt{x}$ that has been exported through ℓ binders". This function forms the base case of $\mathtt{subtractTm}^{\mathtt{D}}?$, the user-defined function that "exports" a term, which is presented later on (section 10.5).

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

The function $\mathtt{pred}^{\mathtt{N}}?$ is a simple specialization of $\mathtt{subtract}^{\mathtt{N}}?$.

```
pred^N? : ∀ {α} → Name (α ↑1) →? Name α
pred^N? = subtract^N? 1
```

On top of `subtract^N?`, we build a convenient eliminator for names. It is simply the elimination of the result of `subtract^N?`.

```
subtractWith^N : ∀ {α A} ℓ → A → (Name α → A) → Name (α ↑ ℓ) → A
subtractWith^N v f = maybe f v ∘′ subtract^N?
```

The function `shiftName ℓ k pf` tests whether its argument, a name `x`, is `ℓ`-bound or `ℓ`-free. If the former, then `x` is returned unmodified. If the latter, then `x + k` is returned. In other words, this function behaves as the identity at `ℓ`-bound names and as a translation at `ℓ`-free names. If one takes `β` to be `α +^W k`, one finds that this function admits the type `(α ↑ ℓ) →^N ((α +^W k) ↑ ℓ)`, which reflects the behavior that was just described. More generally, for greater flexibility, we allow `β` to be a superset of `α +^W k`, and we build a coercion of `α +^W k` to `β` into the definition of `shiftName`.

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

The function `protect↑` shifts a name transformer. If `f` is a function of names to names, then `protect↑ ℓ f` is also a function of names to names, which behaves as the identity at `ℓ`-bound names and behaves like `f` at `ℓ`-free names. More precisely, its behavior at an `ℓ`-free name `x` is to subtract `ℓ` from `x`, apply `f`, then add `ℓ` back to the result.

```
protect↑ : ∀ {α β} ℓ → (α →^N β) → ((α ↑ ℓ) →^N (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 -⟩
```

The reader may notice that, since `shiftName` behaves as the identity at `ℓ`-bound names, one could define it as an instance of `protect↑`. That is, one could give the following alternative definition:

```
shiftName′ : ∀ {α β} ℓ k → (α +^W k) ⊆ β → ((α ↑ ℓ) →^N (β ↑ ℓ))
shiftName′ ℓ k pf = protect↑ (coerce^N pf ∘ add^N k) ℓ
```

This version is fine, but slightly less efficient than `shiftName`, which, in the case where `x` is `ℓ`-free, avoids subtracting `ℓ` and adding `ℓ` back.

10.2 Building terms

Building terms in de Bruijn style is just as easy with our library as it is in the bare de Bruijn index approach. The structure is exactly the same. Natural numbers are converted to variables using $\_^{N}$. Below, we show how to construct representations of the identity function ($\mathtt{idTm}^{D}$), of the application operator ($\mathtt{appTm}^{D}$), of the composition function ($\mathtt{compTm}^{D}$), and of the Church encodings. of true and false ($\mathtt{trueTm}^{D}$ and $\mathtt{falseTm}^{D}$).

```
idTmᴰ    : ∀ {α} → Tmᴰ α
appTmᴰ   : ∀ {α} → Tmᴰ α
compTmᴰ  : ∀ {α} → Tmᴰ α
trueTmᴰ  : ∀ {α} → Tmᴰ α
falseTmᴰ : ∀ {α} → Tmᴰ α

idTmᴰ    = ƛ(V (0 ᴺ))
appTmᴰ   = ƛ(ƛ(V (1 ᴺ) · V (0 ᴺ)))
compTmᴰ  = ƛ(ƛ(ƛ(V (2 ᴺ)) · (V (1 ᴺ) · V (0 ᴺ)))))

trueTmᴰ  = ƛ(ƛ(V (1 ᴺ)))
falseTmᴰ = ƛ(ƛ(V (0 ᴺ)))
```

10.3 Example: computing free variables

As an example of a function that is defined by induction over a term, we again show how to compute a list of the free variables of a term. The overall structure of the code is the same as in the nominal setting (section 5.1). As before, in the case of a binding construct, such as ƛ, we must remove the bound name from the list of variables produced by the recursive call. In this nameless representation, this means that we must remove from the list every occurrence of 0 and we must subtract 1 to every other name. This is done by the auxiliary function $\mathtt{rm}_0$, which applies $\mathtt{pred}^{N}?$ to each element of the list and merges the results.

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

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

10.4 Example: comparing terms for equality

As an example of a function that simultaneously traverses two terms, we show how to compare two terms for equality. We begin with a very simple homogeneous equality test, which has type $\forall\ \{\alpha\} \to |\mathtt{Cmp}|\ \mathtt{Tm}^{\mathtt{D}}\ \alpha\ \alpha$, that is, $\forall\ \{\alpha\} \to \mathtt{Tm}^{\mathtt{D}}\ \alpha \to \mathtt{Tm}^{\mathtt{D}}\ \alpha \to \mathtt{Bool}$. Then, we present a slightly more involved function, which compares two terms for equality up to a user-supplied notion of equality of their free names. This function is able to compare two terms in different worlds α and β, provided the user supplies a name comparator of type $|\mathtt{Cmp}|\ \mathtt{Name}\ \alpha\ \beta$.

De Bruijn's representation is canonical: two terms are α-equivalent if and only if their representations are identical. Thus, a homogeneous equality test is a plain and simple structural equality test. Polymorphic recursion is exploited: after crossing a binder, $\mathtt{eqTm}^{\mathtt{D}}$ is recursively invoked at $\alpha\ \uparrow 1$.

```
eqTm^D : ∀ {α} → |Cmp| Tm^D α α
eqTm^D (V x)     (V y)     = x ==^N y
eqTm^D (t · u)   (v · w)   = eqTm^D t v ∧ eqTm^D u w
eqTm^D (ƛ t)     (ƛ u)     = eqTm^D t u
eqTm^D (Let t u) (Let v w) = eqTm^D t v ∧ eqTm^D u w
eqTm^D _         _         = false
```

Let us now consider the slightly more difficult problem of comparing two terms up to a user-supplied comparator of their free names. Because all of the subtle work takes place at the level of names, we first define a separate and reusable function, called $\mathtt{cmpName}\uparrow$, which shifts a name comparator so that it can be used under ℓ binders. It is analogous in spirit to $\mathtt{protect}\uparrow$ (section 10.1), which shifts a name transformer. The name comparator $\mathtt{cmpName}\uparrow\ \ell\ \Gamma$ compares two names $\mathtt{x}$ and $\mathtt{y}$ in the following manner. If both are ℓ-bound, then they can be safely compared using $\_{=}{=}^{\mathtt{N}}\_$. If both are ℓ-free, then they cannot be compared directly. They are compared by subtracting ℓ and applying the original name comparator Γ. If one of them is ℓ-bound and the other is ℓ-free, then they are considered different.

```
cmpName↑ : ∀ {α β} ℓ → |Cmp| Name α β
                     → |Cmp| Name (α ↑ ℓ) (β ↑ ℓ)
cmpName↑ 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
```

It is now straightforward to define a function that compares two terms for equality, up to a user-supplied comparator of their free names. One compares the two terms in a simple structural fashion. One keeps track of the number ℓ of binders that have been traversed,

and one calls `cmpName↑ ℓ` at variables. Note that keeping incorrect track of ℓ would lead to a type error.

```
cmpTmᴰ : ∀ {α β} → |Cmp| Name α β → |Cmp| Tmᴰ α β
cmpTmᴰ {α} {β} Γ = go 0 where
  go : ∀ ℓ → |Cmp| Tmᴰ (α ↑ ℓ) (β ↑ ℓ)
  go ℓ (V x)      (V y)      = cmpName↑ 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
```

By instantiating the name comparator Γ with the homogeneous name comparison function `_==ᴺ_`, we recover a homogeneous equality test, that is, an equality test for terms that inhabit a common world. This variant is however less efficient than `eqTmᴰ`.

```
_==Tmᴰ_ : ∀ {α} → |Cmp| Tmᴰ α α
_==Tmᴰ_ = cmpTmᴰ _==ᴺ_
```

10.5 Traversals

10.5.1 A reusable traversal

As in the nominal setting (section 5.4), we build a generic term traversal function, so as to avoid code duplication among various operations on terms. Fortunately, in this nameless style, things are simpler. The information that must be carried down during the traversal can be summarized by a single natural number, namely the parameter ℓ that we have already used. This parameter records how many binders have been entered.

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

  tr : ∀ ℓ → Tmᴰ (α ↑ ℓ) → E (Tmᴰ (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

  trTmᴰ : Tmᴰ α → E (Tmᴰ β)
  trTmᴰ = tr 0
```

As in section 5.4, we require that `trName` map names to terms, and as a consequence, we do not wrap the call `trName ℓ x` with the data constructor `V`. This allows us to obtain capture-avoiding substitution as an instance of $\texttt{trTm}^{\texttt{D}}$. When one wishes to map names to names, one can use the following specialized version of $\texttt{trTm}^{\texttt{D}}$, where the name-to-name transformation is composed with `V`.

```
open TraverseTmᴰ

trTmᴰ' :
  ∀ {E} (E-app : Applicative E) {α β}
    (trName : ∀ ℓ → Name (α ↑ ℓ)
                  → E (Name (β ↑ ℓ)))
  → Tmᴰ α → E (Tmᴰ β)
trTmᴰ' E-app trName
  = trTmᴰ E-app (λ ℓ x → pure V ⊛ trName ℓ x)
    where open Applicative E-app
```

10.5.2 Renaming terms

Composing the above function with (a generalized version of) `protect↑` yields a generic term renaming function, $\texttt{renameTm}^{\texttt{D}}\texttt{A}$. This function is parameterized with a user-supplied (and potentially effectful) renaming θ of the free variables.

```
-- Like protect↑, but generalized to applicative functors
protect↑A = {! omitted !}

renameTmᴰA : ∀ {E} (E-app : Applicative E)
               {α β} (θ : Name α → E (Name β))
             → (Tmᴰ α → E (Tmᴰ β))
renameTmᴰA E-app θ
  = trTmᴰ' E-app (protect↑A E-app θ)
```

By instantiating `E` with the identity functor, we obtain a term renaming function that is parameterized with a total renaming of the free variables. By instantiating it with the functor `Maybe`, we obtain a term renaming function that is parameterized with a partial renaming of the free variables.

```
renameTmᴰ : ∀ {α β} → (α →ᴺ β) → (Tmᴰ α → Tmᴰ β)
renameTmᴰ = renameTmᴰA id-app

renameTmᴰ? : ∀ {α β} → (Name α →? Name β)
                     → (Tmᴰ α →? Tmᴰ β)
renameTmᴰ? = renameTmᴰA Maybe.applicative
```

10.5.3 Shifting and coercing terms

Thanks to the above term renaming functions, any operation that maps names to names can be lifted to the level of terms. For instance, the operation of adding `k` to a name can be lifted to the level of terms. This yields a function that adds `k` to the free variables of a term:

```
addTmᴰ : ∀ {α} k → Tmᴰ α → Tmᴰ (α +ᵂ k)
addTmᴰ = renameTmᴰ ∘ addᴺ
```

Although the above definition is concise and elegant, it is somewhat unsatisfactory. Indeed, because `renameTmᴰ` uses `protect↑`, we effectively end up using a combination of `protect↑` and `addᴺ` which we have noted is slightly inefficient (see the discussion of `shiftName′` in section 10.1). The function `shiftName` offers a more efficient alternative. This leads us to the following definition of `shiftTmᴰ`, a function that adds `k` to the free variables of a term:

```
shiftTmᴰ : ∀ {α β} k → (α +ᵂ k) ⊆ β → (Tmᴰ α → Tmᴰ β)
shiftTmᴰ k p = trTmᴰ′ id-app (λ ℓ → shiftName ℓ k p)
```

The operations `subtractᴺ` and `subtractᴺ?` can also be lifted to the level of terms. We obtain functions that subtract (or attempt to subtract) a natural number `k` from the free variables of a term:

```
subtractTmᴰ : ∀ {α} k → Tmᴰ (α +ᵂ k) → Tmᴰ α
subtractTmᴰ = renameTmᴰ ∘ subtractᴺ

subtractTmᴰ? : ∀ {α} ℓ → Tmᴰ (α ↑ ℓ) →? Tmᴰ α
subtractTmᴰ? = renameTmᴰ? ∘ subtractᴺ?
```

The operation `coerceᴺ` can also be lifted to terms by using `renameTmᴰ`. However, it is preferable to use `trTmᴰ′` in a direct manner, as follows:

```
coerceTmᴰ : ∀ {α β} → α ⊆ β → (Tmᴰ α → Tmᴰ β)
coerceTmᴰ pf = trTmᴰ′ id-app (coerceᴺ ∘ ⊆-cong-↑ pf)
-- or less efficiently:
-- coerceTmᴰ = renameTmᴰ ∘ coerceᴺ
```

The existence of two possible definitions of `coerceTmᴰ` reflects the fact that there are two ways of turning a proof `pf` of $\alpha \subseteq \beta$ into a "shifted" name transformer of type $(\alpha \uparrow \ell) \to^{N} (\beta \uparrow \ell)$. One way is to first turn `pf` into a name transformer of type $\alpha \to^{N} \beta$ and then apply `protect↑`. This is written `protect↑ ∘ coerceᴺ pf`. This works, but `protect↑` involves a dynamic test, because, in general, it treats ℓ-bound and ℓ-free names differently. The other way is to first turn `pf` into a proof of $\alpha \uparrow \ell \subseteq \beta \uparrow \ell$ and then turn this proof into a name transformer. This is written `coerceᴺ ∘ ⊆-cong-↑ pf`. This term has no computational content: up to erasure, it is the identity.

Finally, a function that tests whether a term is closed can be defined in the same manner as in the nominal setting (section 5.4):

```
closeTmᴰ? : ∀ {α} → Tmᴰ α →? Tmᴰ ∅
closeTmᴰ? = renameTmᴰ? (const nothing) -- a free var causes a failure
```

10.5.4 Capture-avoiding substitution

To implement capture-avoiding substitution for the type $\mathtt{Tm}^{\mathsf{D}}$, all we need is a specific `trName` that we can pass to $\mathtt{trTm}^{\mathsf{D}}$. If a user-supplied substitution is represented as a function of names to terms, of type `Name` $\alpha \to \mathtt{Tm}^{\mathsf{D}}\ \beta$, then all we need is to "shift" this function so that it can be used under ℓ binders. This is done by the following function, which again is analogous in spirit to `protect`$\uparrow$.

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

Capture-avoiding substitution is then obtained by instantiating $\mathtt{trTm}^{\mathsf{D}}$ with the identity applicative functor and composing it with $\mathtt{substVarTm}^{\mathsf{D}}$:

```
substTmᴰ : ∀ {α β} → (Name α → Tmᴰ β) → (Tmᴰ α → Tmᴰ β)
substTmᴰ = trTmᴰ id-app ∘ substVarTmᴰ
```

As an illustration, here is again a simple function that performs a β-reduction when a β-redex appears at the root of its argument:

```
β-redᴰ : ∀ {α} → Tmᴰ α → Tmᴰ α
β-redᴰ (ƛ fct · arg) = substTmᴰ (subtractWithᴺ 1 arg V) fct
β-redᴰ t             = t
```

11 Soundness of NOMPA (de Bruijn fragment)

We now extend our soundness proof so as to cover the new types and operations of figure 7. The required definitions and proofs (section 11.1) are fairly immediate. What is perhaps less obvious is exactly what guarantees are offered by the resulting "free theorems". We informally show that more precise types lead to stronger "free theorems" (section 11.3). Furthermore, as a case study, we formally study the "free theorem" associated with world-polymorphic term transformation functions, and we prove that every such function commutes with a renaming of the free variables of its argument (section 11.2).

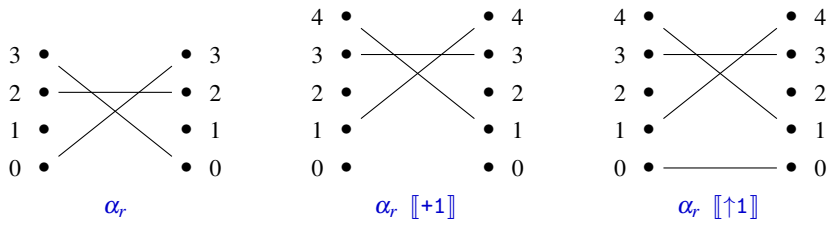

Figure 9. Translating and shifting relations

11.1 Extending the logical relation and proofs

Because we have introduced a new abstract operation `_+1`, which maps a world to a world, we must now define the corresponding operation `_⟦+1⟧`, which maps a relation α_r between two worlds α_1 and α_2 to a relation α_r `⟦+1⟧` between the worlds α_1 `+1` and α_2 `+1`. Only one definition makes sense: the relation α_r `⟦+1⟧` must be the image of the relation α_r under the translation "up by one". This is depicted in figure 9. The definition (in pseudo-code) is as follows:

```
_⟦+1⟧ : (⟦World⟧ ⟦→⟧ ⟦World⟧) _+1 _+1
α_r ⟦+1⟧ =def { (x+1, y+1) | (x, y) ∈ α_r } -- not proper Agda
```

The definition of the operation `_↑1` in terms of `_+1` gives rise to an analogous definition of `_⟦↑1⟧` in terms of `_⟦+1⟧`. The effect of this operation on a relation α_r is to translate it up by one and then to add the pair (0, 0). This is also depicted in figure 9.

```
_⟦↑1⟧ : (⟦World⟧ ⟦→⟧ ⟦World⟧) _↑1 _↑1
_⟦↑1⟧ α_r = ⟦zero^B⟧ ⟦◁⟧ (α_r ⟦+1⟧)
```

There remains to check that the operations and the world inclusion rules of figure 7 are well-behaved, that is, they fit the logical relation. In almost all cases, the proof is immediate. In the case of `cmp^N`, it turns out that our implementation of `cmp^N` is not very well suited to a direct proof. We provide an alternative implementation, which we prove is extensionally equivalent to `cmp^N` and is well-behaved. Because the logical relation is compatible with extensional equality, this allows us to conclude that `cmp^N` itself is well-behaved.

11.2 A free theorem: world-polymorphic functions commute with renamings

In the nominal setting, we exploited the logical relation and parametricity in order to prove that two α-equivalent terms could not be distinguished, or in other words, that two representations of a single object-level term could not be distinguished. This was a non-trivial property because a single object-level term could have multiple representations. In the de Bruijn setting, the situation is somewhat different. De Bruijn's representation is canonical: an object-level term has exactly one representation. Thus, the fact that "α-equivalent terms cannot be distinguished" becomes a tautology.

Nevertheless, in this setting, parametricity can still be exploited in order to establish non-trivial properties of well-typed programs. One such property is the fact that "world-polymorphic homogeneous term transformers commute with renamings", that is, every function `f` of type $\forall\ \{\alpha\} \to \mathtt{Tm^D}\ \alpha \to \mathtt{Tm^D}\ \alpha$ commutes with a renaming of the free variables. In other words, every such function is equivariant (Pitts, 2006). Intuitively, this means that the function `f` must treat the free variables of its argument as "black boxes". It can test whether two free variables are equal, but it cannot depend on the manner in which the free variables happen to be "named" (numbered).

In the following, we choose a "renaming" to be an injective function Φ of names to names. The type of a renaming that maps names in the world α to names in the world β is `Ren` $\alpha\ \beta$. Note that there is a slight difference between a renaming from α to β (`Ren` $\alpha\ \beta$) and the worlds α and β being "related" ($[\![\mathtt{World}]\!]\ \alpha\ \beta$. On renamings, we define three functions $\langle \_ \rangle^{\mathtt{N}}$, $\langle \_ \rangle^{\mathtt{W}}$ and $\langle \_ \rangle^{\mathtt{Tm}}$ which respectively turn a renaming into a function of names to names, into a relation between worlds, and into a function of terms to terms.

```
Ren : (α₁ α₂ : World) → Set
Ren α₁ α₂ = Injection (Nm α₁) (Nm α₂)
  -- * Injection From To is the set of the injective functions
  --   from the setoid From to the setoid To.
  -- * Nm α is the setoid on Name α

⟨_⟩ᴺ : ∀ {α β} → Ren α β → (α →ᴺ β)
⟨ Φ ⟩ᴺ = {! omitted !} -- Projects out the function over names

⟨_⟩ᵂ : ∀ {α β} → Ren α β → [[World]] α β
⟨_⟩ᵂ {α} {β} Φ = ℛ , ℛ-pres-≡
  where ℛ : Name α → Name β → Set
        ℛ x y = ⟨ Φ ⟩ᴺ x ≡ y
        ℛ-pres-≡ : Preserve-≡ ℛ
        ℛ-pres-≡ = {! omitted !}

⟨_⟩ᵀᵐ : ∀ {α β} → Ren α β → (Tmᴰ α → Tmᴰ β)
⟨ Φ ⟩ = renameTmᴰ ⟨ Φ ⟩ᴺ
```

In order to establish that "world-polymorphic homogeneous term transformers commute with renamings", the key is to examine how the logical relation behaves at the type $\mathtt{Tm^D}$.

As described in section 7.1, the definition of the relation ⟦Tm^{D}⟧ is mechanically generated from the definition of the type Tm^{D}:

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

Let us informally study the meaning of this definition. The parameter α_r is "shifted" when a binding construct is entered, and is used at variables. Thus, when two terms are related by ⟦Tm^{D}⟧ α_r, two matching variable occurrences x_1 and x_2 that lie under ℓ binders must be related by α_r ⟦↑⟧ ℓ. This means that either x_1 and x_2 are both ℓ-bound and x_1 is equal to x_2, or x_1 and x_2 are both ℓ-free and α_r relates x_1 - ℓ and x_2 - ℓ. One may summarize this by stating that two terms are related by ⟦Tm^{D}⟧ α_r if and only if they are identical, except for their free variables, which are related by α_r.

We formally prove that for any renaming Φ of type Ren α_1 α_2, two terms t_1 and t_2 are related by ⟦Tm^{D}⟧ ⟨ Φ ⟩^{W} if and only if t_2 is the image of t_1 under the renaming Φ:

```
Rename : ∀ {α β} → Ren α β → Tm^D α → Tm^D β → Set
Rename Φ t₁ t₂ = ⟨ Φ ⟩^Tm t₁ ≡ t₂

⟦Tm^D⟧⇔Rename : ∀ {α β} (Φ : Ren α β) → ⟦Tm^D⟧ ⟨ Φ ⟩^W ⇔ Rename Φ
```

Now, let f be a world-polymorphic term transformer, and let f_r be a proof that f is well-behaved. (That is, f_r is the "free theorem", instantiated with f.) Then, it is easy to prove that every renaming Φ commutes with f. Indeed, by instantiating f_r with the relation ⟨ Φ ⟩^{W}, we find that f must map ⟦Tm^{D}⟧ ⟨ Φ ⟩^{W}-related arguments to ⟦Tm^{D}⟧ ⟨ Φ ⟩^{W}-related results. Then, by applying the lemma ⟦Tm^{D}⟧⇔Rename in both directions, we find that renaming a term before applying f to it has the same effect as applying f first and renaming the result. In other words, the functions ⟨ Φ ⟩^{Tm} ∘ f and f ∘ ⟨ Φ ⟩^{Tm} are extensionally equivalent.

```
-- Pointwise equality
f ≐ g = ∀ x → f x ≡ g x
```

```
ren-comm : (f   : ∀ {α} → Tm^D α → Tm^D α)
           (f_r : (∀⟨ α_r : ⟦World⟧ ⟩⟦→⟧ ⟦Tm^D⟧ α_r ⟦→⟧ ⟦Tm^D⟧ α_r) f f)
           {α β} (Φ : Ren α β)
            → ⟨ Φ ⟩^Tm ∘ f ≐ f ∘ ⟨ Φ ⟩^Tm
ren-comm f f_r Φ t
  = ⟦Tm^D⟧⇒Rename Φ (f_r ⟨ Φ ⟩^W (Rename⇒⟦Tm^D⟧ Φ ≡.refl))

-- The proof, step by step:

⟨ Φ ⟩^Tm t ≡ ⟨ Φ ⟩^Tm t                {- definition of Rename -}
Rename Φ t (⟨ Φ ⟩^Tm t)                 {- Rename Φ ⇒ ⟦Tm^D⟧ ⟨ Φ ⟩^W -}
⟦Tm^D⟧ ⟨ Φ ⟩^W t (⟨ Φ ⟩^Tm t)          {- parametricity of f (called f_r) -}
⟦Tm^D⟧ ⟨ Φ ⟩^W (f t) (f (⟨ Φ ⟩^Tm t))  {- ⟦Tm^D⟧ ⟨ Φ ⟩^W ⇒ Rename Φ -}
Rename Φ (f t) (f (⟨ Φ ⟩^Tm t))        {- definition of Rename -}
⟨ Φ ⟩^Tm (f t) ≡ ⟨ Φ ⟩^Tm (f t)        {- definition of _≐_ -}
⟨ Φ ⟩^Tm ∘ f ≐ ⟨ Φ ⟩^Tm ∘ f
```

A reviewer pointed out that the idea that "any world-polymorphic function commutes with renamings" comes up in the category-theoretic accounts of variable binding, such as Stark's Ph.D thesis (Stark, 1994) and also in Fiore, Plotkin, and Turi's (Fiore *et al.*, 1999).

The type of terms forms a presheaf over a category whose objects are worlds and morphisms are renamings. The condition `ren-comm` states exactly that world-polymorphic functions between terms can be seen as morphisms in the same presheaf category. We do acknowledge the existence of this connection, but cannot give here a detailed account of it: more space and more work would be required.

11.3 On the strength of free theorems

Every type comes with a "free theorem" (section 7.1). We have just exploited the free theorem associated with the type $\forall\ \{\alpha\} \to \mathtt{Tm^D}\ \alpha \to \mathtt{Tm^D}\ \alpha$ to establish that every function of this type commutes with renamings. In general, the meaning and strength of the free theorem varies greatly with the type that is considered. For instance, at type $\mathbb{N}$, the free theorem says: "every term of type $\mathbb{N}$ is equal to itself", a tautology. At type $\mathbb{N} \to \mathtt{Bool}$, the free theorem says: "every function of type $\mathbb{N} \to \mathtt{Bool}$ is deterministic", that is, every such function maps equal arguments to equal results. This is again a tautology, since AGDA is a pure language. At type $\forall\{\mathtt{A} : \mathtt{Set}\} \to \mathtt{A} \to \mathtt{A}$, the free theorem can be used to prove that "every function of this type behaves as the identity function", a much stronger statement. The universal quantification over `A` is key to the strength of this theorem.

11.3.1 Free theorems for homogeneous term transformers

We now discuss what can affect the strength of the free theorems in our context. For simplicity, we continue to focus on homogeneous term transformers, that is, on functions that map a term to a term in the same world.

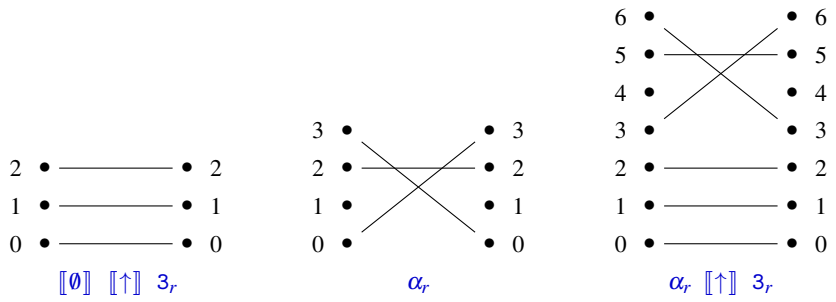

Figure 10. Various typical forms of worlds and their relations

Naturally, polymorphism again plays an important role in the strength of free theorems. In this case, however, there is no type polymorphism, since these functions map terms to terms. What matters instead is world polymorphism. There are several ways in which a homogeneous term transformer can be world-polymorphic. Obviously, it can be parameterized with a world α and work with terms of type $\mathtt{Tm^D}\ \alpha$. It can also be parameterized with a natural number ℓ and work with terms of type $\mathtt{Tm^D}\ (\emptyset \uparrow \ell)$. Or, somewhere in between these two approaches, it can be parameterized with both α and ℓ and work with terms of type $\mathtt{Tm^D}\ (\alpha \uparrow \ell)$. The relations that correspond to these various forms of worlds have different shapes, suggested in figure 10. The more constrained the shape of the relation, the weaker the free theorem, because the free theorem is universally quantified over all relations of that shape.

Among these three forms of worlds, the most constrained one is $\emptyset \uparrow \ell$. This world corresponds to the interval $[0, \ell)$, or in other words, to the type $\mathtt{Fin}\ \ell$. Thus, type-checking alone guarantees that a term transformer of type $\forall\ \{\ell\} \to \mathtt{Tm^D}\ (\emptyset \uparrow \ell) \to \mathtt{Tm^D}\ (\emptyset \uparrow \ell)$ maps terms whose indices are in the range $[0, \ell)$ to terms whose indices are also in this range. This is the same guarantee that is offered by the `Fin` approach. It turns out that the free theorem associated with this type does not offer anything beyond this guarantee. Indeed, as suggested by figure 10, the relation $[\![\emptyset]\!]\ [\![\uparrow]\!]\ \ell$ is the identity relation on the interval $[0, \ell)$. As a result, the free theorem states that "a term transformer of type $\forall\ \{\ell\} \to \mathtt{Tm^D}\ (\emptyset \uparrow \ell) \to \mathtt{Tm^D}\ (\emptyset \uparrow \ell)$ maps equal arguments to equal results", which is trivial. In summary and as noted earlier (section 9.2), every program that is well-typed in the `Fin` approach can be translated in our system by replacing $\mathtt{Fin}\ \ell$ with $\mathtt{Name}\ (\emptyset \uparrow \ell)$, but no new guarantees are magically obtained by doing so.

At the opposite end, among these three forms of worlds, the least constrained one is α. That is, the strongest possible free theorem is obtained by quantifying over an arbitrary world. As proved earlier (section 11.2), this theorem implies that "a term transformer of type $\forall\ \{\alpha\} \to \mathtt{Tm^D}\ \alpha \to \mathtt{Tm^D}\ \alpha$ commutes with an arbitrary renaming". Thus, there are strictly fewer functions of type $\forall\ \{\alpha\} \to \mathtt{Tm^D}\ \alpha \to \mathtt{Tm^D}\ \alpha$ than there are functions of type $\forall\ \{\ell\} \to \mathtt{Tm^D}\ (\emptyset \uparrow \ell) \to \mathtt{Tm^D}\ (\emptyset \uparrow \ell)$. Pragmatically, this means that, by declaring the former type instead of the latter, one makes it possible for the type-checker to detect more programming errors. This includes the family of errors where one "forgets to shift" a free variable and inadvertently turns it into a bound variable. In contrast, the `Fin` approach does not guarantee that these errors are detected.

In terms of abstraction, the world $\alpha \uparrow \ell$ is in between $\emptyset \uparrow \ell$ and α. As suggested by figure 10, a relation of the form α_r $[\![\uparrow]\!]$ ℓ must be the identity on the interval $[0, \ell)$, but can exhibit arbitrary structure above this interval. When working in de Bruijn style, a term transformer that operates in a world of the form $\alpha \uparrow \ell$ is often used as an auxiliary function in order to eventually obtain a term transformer that operates in a world of the form α. We have seen examples of this in sections 10.4 and 10.5. For instance, here is a typical type for an auxiliary function that has access to some kind of information about ℓ-bound variables but must treat ℓ-free variables in a parametric manner:

```
some-op : ∀ {α ℓ} → (Fin ℓ → Info) → Tmᴰ (α ↑ ℓ) → Tmᴰ (α ↑ ℓ)
some-op Γ t = {! omitted !}
```

The free theorem associated with the type of `some-op` guarantees that this function commutes with a renaming of the ℓ-free variables.

When reasoning about free theorems, one must keep in mind a potential pitfall: adding extra arguments to a function type can drastically decrease the strength of the associated free theorem. An extreme example of this phenomenon is the type $\forall\ \{\alpha\} \to (\alpha \equiv \emptyset) \to \mathtt{Tm}^{\mathtt{D}}\ \alpha \to \mathtt{Tm}^{\mathtt{D}}\ \alpha$. The free theorem is "ruined" by the argument $\alpha \equiv \emptyset$. Even though the theorem is universally quantified over a relation α_r, the presence of this argument requires that α_r be the empty relation, and the free theorem becomes trivial. In this case, the problem is obvious: it is clear that the type $\forall\ \{\alpha\} \to (\alpha \equiv \emptyset) \to \mathtt{Tm}^{\mathtt{D}}\ \alpha \to \mathtt{Tm}^{\mathtt{D}}\ \alpha$ is isomorphic to $\mathtt{Tm}^{\mathtt{D}}\ \emptyset \to \mathtt{Tm}^{\mathtt{D}}\ \emptyset$, whose free theorem is trivial. Sometimes, however, it is not immediately obvious how the presence of certain arguments affects the strength of the free theorem.

An interesting example where the presence of an extra argument does *not* weaken the free theorem is offered by the type $\forall\ \{\alpha\} \to \mathtt{Supply}\ \alpha \to \mathtt{Term}\ \alpha \to \mathtt{Term}\ \alpha$. This is the type of a homogeneous term transformer that is equipped with a supply of fresh names. It is the type of the normalization-by-evaluation algorithm (section 5.7). By definition of the type `Supply` (section 5.4.3), a supply is just a pair of a binder and a "fresh-for" assertion. By definition (figure 5), the relations that govern binders and "fresh-for" assertions are full. Thus, the relation $[\![\mathtt{Supply}]\!]\ \alpha_r$ is full too: requiring that it be inhabited by two arbitrary supplies does not constrain α_r in any way. As a result, the presence of this extra argument does not weaken the theorem. The free theorem associated with the type $\forall\ \{\alpha\} \to \mathtt{Supply}\ \alpha \to \mathtt{Term}\ \alpha \to \mathtt{Term}\ \alpha$ still guarantees that every function of this type commutes with a renaming of the free variables. In fact, it also guarantees that "choices of fresh names do not matter", that is, the result produced by such a function is independent of which supply is used.

11.4 The influence of translating versus shifting

Our first conference paper (Pouillard & Pottier, 2010) contains a mistake that we would like to briefly explain because it emphasizes why it can be interesting to have both $\_+^{\mathtt{W}}\_$ and $\_\uparrow\_$ instead of just the latter.

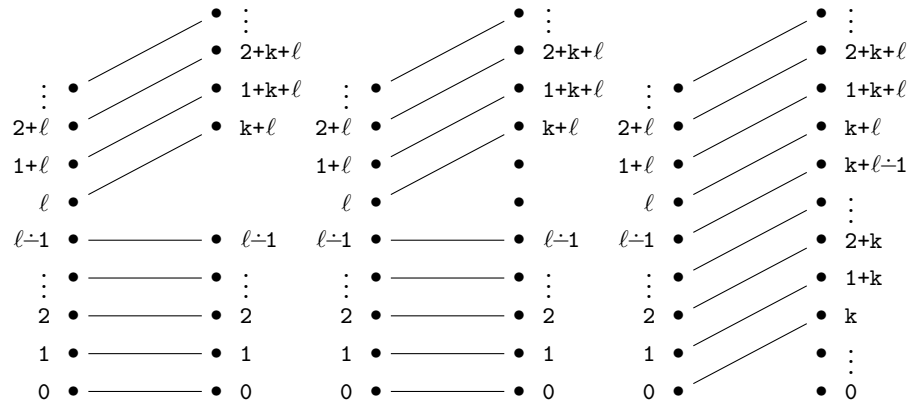

Figure 11. The graphs of `protectedAdd`, `protectedAdd↑`, and `unprotectedAdd`

Consider the function `protectedAdd`, a variant of which we have already encountered under the name `shiftName′` (section 10.1):

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

It is used as a building block in the definition of the function that "shifts" a term. The leftmost diagram in figure 11 depicts its graph, that is, which arguments are mapped to which results.

Here, we have ascribed `protectedAdd` a very precise type. In the first conference paper, however, the operation `_+ᵂ_` was not available, so we used `_↑_` instead. Thus, we wanted to define the following variant of `protectedAdd`, whose type is less precise:

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

The graph of this function appears in the middle of figure 11. The edges are the same as in the case of `protectedAdd`, but we have added new nodes so as to suggest that the codomain is now larger. The trouble with this less precise type is, it has more inhabitants. In particular, the function `addᴺ k`, which adds `k` to every name, regardless of whether this name is ℓ-bound or ℓ-free, admits this type, whereas it does not admit the earlier, more precise type:

```
unprotectedAdd : ∀ {α} k ℓ → (α ↑ ℓ) →ᴺ (α ↑ k ↑ ℓ)
unprotectedAdd k ℓ = coerceᴺ (⊆-exch-↑-↑′ ℓ k) ∘ addᴺ↑ k
  -- this is ok since (α ↑  k) ↑ ℓ ≡ (α ↑ ℓ) ↑  k
  -- whereas          (α +ᵂ k) ↑ ℓ ≢ (α ↑ ℓ) +ᵂ k
```

The graph of the function `unprotectedAdd` appears as the rightmost diagram in figure 11. In the conference paper (Pouillard & Pottier, 2010), we fell into this trap and defined a function (called `shiftName` there) with the behavior of `unprotectedAdd`, whereas the intended behavior was that of `protectedAdd`↑. We did not notice the bug, and indeed were quite confident that there was no bug, because the soundness proof based on logical relations went through. In fact, the proof was correct, but the resulting "free theorem" did not guarantee that this particular kind of mistake would be ruled out. Indeed, because the "generalized import" function (which "shifts" a term) was polymorphic in the source world α, the "free theorem" did guarantee that this function would deal with *free names* in a correct manner, that is, in an equivariant manner. The "free theorem" did not guarantee, however, that *bound names* would be correctly dealt with, and indeed our mistake caused a bound name to be incorrectly replaced with another bound name.

One moral of this story is that, even when the logical relations argument "goes through", one must be careful to check that the "free theorems" do mean what one hopes they mean. Another possible moral is that more precise types can help detect more mistakes; but this requires that the programmer be wise enough to make use of these precise types in the first place.

12 Related work

The difficulty of programming with, or reasoning about, names, binders, and α-equivalence has been known for a long time. It has recently received a lot of attention, due in part to the PoplMark challenge (Aydemir *et al.*, 2005). Despite this attention, the problem is still largely unsolved: according to Guillemette and Monnier, for instance, "none of the existing representations of bindings is suitable" (Guillemette & Monnier, 2008).

12.1 Systems

In the following, we review several systems that are intended to facilitate the manipulation of names and binders. This review is not exhaustive: we focus on relatively recent related work. Pouillard's Ph.D. thesis (2012) contains a much more detailed review of the connections between NomPa and several approaches to representing syntax with binders, including nominal signatures (Pitts, 2006), Cαml (Pottier, 2006), FreshLib (Cheney, 2005), Binders Unbound (Weirich *et al.*, 2011), and the Locally Nameless (Aydemir *et al.*, 2008; Charguéraud, 2011) and Locally Named (Sato & Pollack, 2010; Pollack *et al.*, 2011) representations.

Our previous papers This paper combines and extends two conference papers (Pouillard & Pottier, 2010; Pouillard, 2011a). Here is a summary of the differences between these papers and the present work and of the path that we have followed.

In the first conference paper, we present an abstract interface for programming with names and binders. By design, this interface does not use any dependent types, so it can be expressed, if desired, in a programming language such as Haskell. We point out that this interface can be implemented either in nominal style or using de Bruijn indices. These two implementations have different cost models. In particular, some operations are no-ops in

one implementation, but not in the other. World inclusion, for instance, is a no-op in the nominal model, but is implemented as addition in de Bruijn model. For each of the two implementations, separately, we construct a family of logical relations, and use them to obtain certain theorems for free. For the nominal implementation, in particular, we focus on the type `Tm` of untyped lambda-terms; we prove that the logical relation at this type coincides with a standard notion of α-equivalence; and we conclude that every function of type $\forall\ \{\alpha\} \to \texttt{Tm}\ \alpha \to \texttt{Tm}\ \alpha$ respects α-equivalence (and, in fact, commutes with an arbitrary permutation of the free names).

In the second conference paper, the first author reconsiders some of the design decisions of the first paper. In particular, the decision to not use any dependent types, as well as the decision to abstract away the differences between the nominal and de Bruijn implementations, are quite costly: they make the system significantly more complex than it could be. Thus, Pouillard decides to reverse these decisions, by taking full advantage of dependent types, and by focusing on the de Bruijn implementation scheme, which in several ways is simpler than the nominal scheme. This allows several simplifications: for instance, the distinction between "weak" and "strong" links disappears, and the notion of "link" itself disappears, because, in the de Bruijn implementation, a link from α to β is just an equality between β and $\alpha\ \uparrow 1$. World inclusion becomes a no-op. The resulting system is closely related to earlier treatments of well-scoped de Bruijn indices, but is more precise, thanks to the distinction between the worlds $\alpha\ +^{\texttt{w}}\ \texttt{k}$ and $\alpha \uparrow \texttt{k}$, and yields stronger guarantees, that is, stronger "theorems for free". The logical relations argument is machine-checked. A flaw in the first conference paper is fixed (see section 11.4).

In the present paper, we stick with dependent types. Furthermore, we point out that there is no need to choose between the nominal approach and de Bruijn's approach. We propose a system where these techniques co-exist, so it is possible for some data structures to use a nominal representation, while others use de Bruijn indices. Furthermore, as demonstrated in Pouillard's Ph.D. thesis (2012), the system supports other representation techniques, including de Bruijn levels and the Locally Nameless representation. In short, the present system is very expressive and is simpler than the one found in our first conference paper. Its soundness proof involves just one logical relation, and is machine-checked.

In the present approach, it is still possible, if desired, to abstract away the differences between the nominal style and de Bruijn's style. On top of NOMPA, one defines the interface of a little library, which advertises an abstract type of "links" between worlds, together with a small number of operations, such as exporting or importing a name through a link. One provides two implementations of this interface, one in nominal style, the other in de Bruijn style. Client code can then be written against the abstract interface, independently of which implementation is eventually chosen. More details appear in Pouillard's Ph.D. thesis (2012).

FRESHML and Pure FRESHML FreshML (Shinwell *et al.*, 2003) extends ML with primitive types for names (known as atoms) and name abstractions. The semantics of FRESHML dictates that pattern matching against a name abstraction silently replaces the bound atom with a fresh atom. FRESHML guarantees that "name abstractions cannot be broken", that is, two α-equivalent terms cannot be distinguished. Nevertheless, FRESHML is unsafe: it is possible for a name to escape its scope. Put another way, FRESHML is

impure: name generation is an observable side effect. There are functions which, when applied twice to the same argument, produce observably different results.

Pure FRESHML (Pottier, 2007) imposes additional static proof obligations, which ensure that freshly created atoms do not escape their scope, and correspond to Pitts' *freshness condition for binders* (2006). Because these proof obligations are expressed in a specialized logic, they can be discharged automatically. Because it is safe, Pure FRESHML can be implemented either using atoms (such as the original FRESHML) or using de Bruijn indices. This is an implementation choice, which the programmer need not know about.

In contrast with Pure FRESHML, the approach proposed in the present work does require more runtime checks. For instance, the operation `exportTm?` (section 5.4.7) fails if its argument contains a free name that cannot be exported, whereas Pure FRESHML requires a static proof that no such name occurs in the term that is exported. Of course, in Pure FRESHML, inserting an explicit dynamic check into the program is sometimes the only way of meeting this static proof obligation. In that case, the two approaches are ultimately equivalent.

In Pure FRESHML, name abstraction is a primitive notion, and the fact that deconstructing an abstraction automatically freshens the bound atom is used to guarantee that all terms effectively live in a single world. In our work, in contrast, name abstraction is explained in terms of more basic notions, and it is possible to deconstruct a name abstraction without substituting a fresh name for the bound name. This leads to a finer-grained understanding of binding, to greater expressiveness, and, in some cases, to greater runtime efficiency.

Nominal System T Pitts' Nominal System *T* (2010) follows the tradition of FRESHML and guarantees that name abstractions cannot be violated. In order to ensure that names do not escape their scope, Pitts uses a dynamic technique: the ν construct, which can be applied to any term, turns every name occurrence that is about to escape its scope into a harmless "anonymous name", known as `new`. One potential advantage of this approach is that the application of ν to a term can be lazily evaluated, whereas our `exportTm?` operation requires eagerly traversing the entire term in order to determine whether it succeeds or fails.

We can emulate Pitts' approach on top of our system by introducing a special value that represents `new`. This does not require any change to NOMPA. We define the type `Name? α` as `Maybe (Name α)`, and let `new` be `nothing`. (Another approach is to define `Name? α` as `Name (α ↑1)` and `new` as `zero`$^{\mathtt{N}}$.) Most of our operations on names can then be easily lifted to the type `Name?`. In particular, the lifted version of `export`$^{\mathtt{N}}$`?` receives the type `Name? (b ◁ α) → Name? α` and can thus be viewed as a total function over possibly undefined names. The lifted version of `exportTm?` receives the type `Tm (b ◁ α) → Tm α` and is thus a total function. Of course, one must keep in mind that terms can now contain occurrences of the undefined name `new`. In this approach, there is no direct equivalent of the operation `¬Name∅`, since the type `Name? ∅` is inhabited by `new`.

Elphin, Delphin, Beluga, Beluga$^{\mu}$ Elphin (Schürmann *et al.*, 2005), Delphin (Poswolsky & Schürmann, 2008; Poswolsky & Schürmann, 2009), and Beluga (Pientka, 2008) are closely related to one another in several ways. These programming languages have separate data and computation layers. The types in the data layer are built out of LF types and terms.

Elphin only has a limited way to include types from the data layer in the computation layer. Both Delphin and Beluga have contexts, but handle them differently. Delphin deals with an implicit "current context", while Beluga has explicit contexts. Delphin allows incremental changes of the current context using ν, whereas Beluga combines contexts and LF objects at the boundary between data and computation.

At the data level, Elphin, Delphin, Beluga provide substitution and higher-order matching as primitive operations. This ambitious approach can eliminate some boilerplate code, at the cost of a complex meta-theory. By contrast, the meta-theory of our proposal is very simple, as it only extends AGDA's existing logical relation with a few new abstract types and operations. In principle, we should be able to eliminate some of the boilerplate via generic programming, following Cheney's FRESHLIB, Licata and Harper, and BINDERS UNBOUND.

Beluga$^{\mu}$ (Cave & Pientka, 2012) extends Beluga with richer computational types. It allows the definition of recursive data types not only in the data layer, but in the computation layer as well. These data types can be indexed by LF objects of the data layer. While substitution comes for free in the data layer, it has to be defined by the user in the computation layer. Incidentally, Cave and Pientka use a version of our system in order to establish the meta-theoretic properties of Beluga$^{\mu}$ in the COQ proof assistant. They use a simplified version of the "links" found in our first conference paper (Pouillard & Pottier, 2010) which corresponds to the "little link library" that we have mentioned above.

Well-scoped de Bruijn indices We have reviewed in section 8 how type-theoretic machinery (such as nested algebraic data types, generalized algebraic data types, or dependent types) can be used to ensure that every de Bruijn index remains within range (Bellegarde & Hook, 1994; Bird & Paterson, 1999; Altenkirch & Reus, 1999). In fact, dependent types can be used to encode not only the lexical scoping discipline, but also the type discipline of an object language: see, for instance, Chen and Xi (2003a; 2003b) and Chlipala (2007). This is an aspect that we did not investigate and leave for future work. De Bruijn indices are, by nature, very low-level: it is desirable to build more abstract representations of top of them. For instance, Donnelly and Xi (2005) define an algebraic data type of terms that is based on well-scoped de Bruijn indices, but is indexed with a higher-order abstract syntax representation of terms. Licata and Harper's system (2009) is also implemented on top of well-scoped de Bruijn indices. Our system exposes de Bruijn indices, but offers stronger guarantees than conventional systems based on well-scoped de Bruijn indices. If world polymorphism is appropriately exploited, the system can guarantee not only that every index is within range, but also that free variables are treated abstractly (section 11.2), which implies that certain programming errors, such as "forgetting to shift" a free variable, can be detected by the type system.

Licata and Harper's system Licata and Harper (2009) aim to provide substitution for free when possible, whereas we do not explicitly explore this aspect. They impose the use of well-scoped de Bruijn indices to the programmer, whereas our "names" are more flexible and can be used as atoms or as de Bruijn indices.

This said, there are numerous similarities between the two systems. Both keep track of the context, or world, within which each name makes sense. Both offer flexible ways

of parameterizing or quantifying types over worlds. Both offer ways of moving data from one world to another: Licata and Harper's weakening and strengthening respectively correspond to our import (coerce) and export operations. Both systems support first-class computational functions. Not all functions can be imported or exported, but some can: for instance, in both systems, the example of normalization by evaluation (section 5.7), which requires importing a function into a larger world, is made type-correct by planning ahead and making this function polymorphic with respect to an arbitrary world extension.

FRESHLIB and BINDERS UNBOUND Like us, Cheney (2005) (FRESHLIB), Weirich et al. (2011) (BINDERS UNBOUND) strive to develop libraries that facilitate programming with names and binders inside a mainstream programming language (in these cases, Haskell). The facilities that they offer in order to describe data structures with names and binders are quite powerful. For instance, BINDERS UNBOUND is able to express "telescopes", which are analogous to the one-hole contexts of section 5.6. BINDERS UNBOUND is more recent than FRESHLIB, and, as far as we understand, subsumes it. BINDERS UNBOUND is less expressive than our system (fewer data structures can be defined) and can be less efficient (some operations, such as traversing a term, are more costly). On the other hand, using a more restricted data description language makes generic programing easier: code for traversal, substitution, etc. can be automatically generated, whereas, in our system, generic programming remains a topic for future study. Pouillard (2012) gives a more detailed account of the connection between BINDERS UNBOUND and our system, including an encoding of the former into the latter.

Moving across representations It is arguably desirable to be able to offer several choices of representation within a single system, and to be able to migrate from one representation to another. Our system supports multiple representation styles, including the nominal representation (section 4), de Bruijn's representation (section 9), and more (Pouillard, 2012). Furthermore, our implementation of normalization by evaluation (section 5.7) illustrates how to move back and forth between "syntactic" name abstractions and "semantic" name abstractions in the style of higher-order abstract syntax.

Atkey and co-authors (Atkey, 2009; Atkey *et al.*, 2009) investigate how to move back and forth between higher-order abstract syntax and de Bruijn indices. The translation out of higher-order abstract syntax produces well-scoped de Bruijn indices, but the proof of this fact is meta-theoretic. Atkey uses Kripke logical relations to argue that the current world at the time of application of a certain function must be larger than the world at the time of construction of this function. This seems somewhat related with our use of bounded polymorphism in the definition of semantic name abstractions (section 5.7). An exact connection remains to be investigated.

12.2 Questions

One traditionally distinguishes several broad approaches to the problem of names and binders, which employ seemingly different tools, namely: atoms and atom abstractions; well-scoped de Bruijn indices; higher-order abstract syntax. We believe that this distinction

can be superficial. In fact, our work presents strong connections with all three schools of thought. Perhaps more important are the following questions:

Can α-equivalent terms be distinguished? Except for the bare nominal and bare de Bruijn approaches (sections 3.1 and 8.1), all of the systems that we have discussed guarantee that two α-equivalent terms cannot be distinguished. These systems offer adequate encodings of nominal terms and guarantee that "the identity of a bound name cannot be observed".

What hygiene properties are enforced by the system? The most basic hygiene property is the guarantee that α-equivalent terms cannot be distinguished. This is a fairly weak property: some system offer further guarantees. In FRESHML (Shinwell *et al.*, 2003) name generation is an observable effect. The same is true of several other approaches, including Cαml (Pottier, 2006), BINDERS UNBOUND (Weirich *et al.*, 2011), and the Locally Nameless (Aydemir *et al.*, 2008; Charguéraud, 2011) and Locally Named (Sato & Pollack, 2010; Pollack *et al.*, 2011) approaches. In other words, in these approaches, applying a function twice to the same argument can produce distinct results. Nominal System *T* (Pitts, 2010) repairs this problem by dynamically detecting that an atom is about to escape and by turning it into a harmless "anonymous atom". Systems based on well-scoped de Bruijn indices enforce the invariant that every index is within range, that is, every name refers to some binding site. However, this alone does not imply that indices are correctly adjusted where needed, or that comparisons between names are allowed only when they make sense. In systems based on higher-order abstract syntax, and in Pure FRESHML (Pottier, 2007), name manipulation is hygienic by design: this is built in the syntax and semantics of the programming language.

In the present paper, hygiene is built in at a very low level. We provide a small number of abstract types, such as `Binder` and `Name`, together with a small number of operations. These operations are restricted on purpose: for instance, comparing two binders for equality is disallowed; comparing two names for equality is permitted only if they inhabit a common world. Through logical relations and parametricity, we are able to explore the consequences of these restrictions and to find out which hygiene properties can be expected of a well-typed program. In particular, we guarantee that two α-equivalent terms cannot be distinguished and that functions turn α-equivalent arguments into α-equivalent results. These results hold because we consider that, by definition, "α-equivalence" at type τ is the logical relation $[\![\tau]\!]$, and because parametricity guarantees that every well-typed program inhabits the logical relation. We do check that our definition of α-equivalence corresponds to the "standard" notion of α-equivalence at type `Tm` (section 7.4) and more generally at every type that encodes a nominal signature in the sense of Pitts (2006).

Is there a type of names, or do names inhabit every type? In the "first-order" systems, there is a dedicated type of names. This includes the `Fin` approach to de Bruijn indices, the Locally Nameless and Locally Named approaches, FRESHML, Nominal System *T*, Cαml, BINDERS UNBOUND, and the present work. Names inhabit this type and none other. In "higher-order" systems, such as Delphin and Beulga, there is no fixed type of names. Instead, a name can inhabit any type of the data layer.

Does the system distinguish data and computation layers? Elphin, Delphin, Beluga, Beluga$^{\mu}$ clearly distinguish two layers. The data layer is restricted and does not have computational functions. In many of the systems that use code generation or generic programming, including Cαml, BINDERS UNBOUND, and several implementations of the Locally Nameless and Locally Named approaches, algebraic data types with binding structure are not allowed to contain computational functions, so there are effectively two layers as well. Pure FRESHML does not support first-class functions at all. FRESHML and its implementation FRESH OCAML do allow computational functions to appear within data. The only generic operation that is provided is name swapping. Similarly, Nominal System T supports first-class functions and name swapping, but is not implemented. In Licata and Harper's work (2009), data and computation can also be mixed. However, they are both part of a "universe" which is then interpreted in terms of the computation types of the host language.

At which types can substitution be defined? Does it come "for free"? In Elphin, Delphin, and Beluga, substitution comes "for free" at every type of the data layer. This is the reason why there is a data layer in the first place. In Beluga$^{\mu}$, substitution comes for free in the data layer but must be defined by the user (when this makes sense) in the computation layer.

In FRESHML, name swapping comes for free, but it is up to the user to define substitution. Similarly, in Cαml, a code generator produces the definition of several operations, including renaming, but not substitution. In BINDERS UNBOUND, substitution is obtained via generic programming for a family of types that does not include functions. In Licata and Harper's work, a generic definition of substitution is available at many (albeit not all) types.

In our system, renaming and substitution are defined by the user when this makes sense. As shown in Pouillard's thesis (2012), a wide range of other systems, including Cαml and BINDERS UNBOUND, can be embedded in our system. It should be possible to define generic operations for the subset of our types that lie in the image of these embeddings.

How does the system keep track of the context or world in which a name makes sense? In Pure FRESHML and in Nominal System T, there is effectively just one world, within which every name makes sense; static or dynamic checks guarantee that no confusion can arise. In Elphin, Delphin, or in Licata and Harper's system, the meaning of types is relative to a "current context", and a number of modalities are provided to discard the current context, extend it with one new name, etc. In Beluga and Beluga$^{\mu}$, contexts are explicit: a data-layer type, once annotated with a context, becomes a computation-layer type. In the present work, worlds are explicit, and are built into algebraic data type definitions by the programmer. There is no notion of a "current world": multiple worlds can co-exist, and mechanisms are offered for transporting names (or whole data structures) from one world to another.

Which high-level operations are involved in the semantics? The semantics of FRESHML, Pure FRESHML, and Nominal System T involves renaming. The Locally Nameless and Locally Named approaches, as well as BINDERS UNBOUND, involve "open" and "close"

operations which can be viewed as renamings of a particular kind. The semantics of Elphin, Delphin, Beluga, and Beluga$^{\mu}$ involve higher-order matching. In the present work, as well as in Licata and Harper's work, no costly operations are built into the semantics; high-level operations, such as `exportTm?` and `shiftTm`$^{\mathtt{D}}$, are programmed explicitly or obtained via generic programming.

13 Conclusion and future work

Our contributions are practical and fundamental. On the practical side, we have presented a library, written in AGDA, for programming with names and binders. This library is economical: it defines a relatively small number of abstract types and operations. It is expressive: it offers multiple ways of representing conventional syntax and supports the definition of complex data structures that involve binding. It is sound: this has been mechanically checked.

On the fundamental side, we re-use the logical relation defined for AGDA by Bernardy et al. (2010) and extend it so as to mechanically obtain a definition of "α-equivalence" at every type. This includes complex algebraic data types as well as types that involve computational functions. We show how the "free theorems" that concern certain world-polymorphic functions provide guarantees about well-typed client code.

We believe that this work establishes numerous connections between three schools of thought that are sometimes considered separate, namely the "nominal", "de Bruijn", and "higher-order abstract syntax" schools. Although the first part of this paper (sections 3 to 7) relies on a "nominal" intuition, the second part (sections 8 to 11) shows that it does not take much to add support for de Bruijn indices. Furthermore, our normalization by evaluation algorithm (section 5.7) shows how to work simultaneously with "syntactic" name abstractions in the style of the "nominal" school and with "semantic" name abstractions in the style of higher-order abstract syntax.

There are many avenues for future work.

We would like to make our library easier to use. In particular, we would like to be able to automatically construct most of the world inclusion witnesses that are needed in order to move a name or a term from a world to another. We hope that AGDA's experimental reflection mechanism can be used for this purpose.

We have illustrated how hand-written generic traversal functions help avoid code duplication. We would like to go one step further by giving generic definitions of these functions for a fixed universe of types.

We would like to study how to support the presence of side effects. One could attempt to do this either by remaining within AGDA and modeling effects as monads or by stepping out of AGDA and extending an imperative programming language with operations on names and binders inspired by our library.

We would like to explore how one can reason about (that is, prove properties of) the programs that use our library. The library currently advertises a number of abstract types and operations. It seems that, in order to allow a client to reason, the library should also publish a number of laws, or properties, that describe the behavior of these operations.

Although we have claimed informally that "worlds can be erased", we have not carried out a formal proof of this claim. One would be tempted to define an alternative implemen-

tation of our library where `World` is the unit type and to prove that this implementation behaves in the same way as the implementation presented here. This is probably not as easy as one might think, though. If `World` is the unit type, then there is no way to implement `¬Name∅` in a total language. One must somehow implement a version of `¬Name∅` that always fails at runtime, and argue that a well-typed program will never encounter a failure.

In order to achieve reasonable runtime efficiency, it would also be desirable to study how names can be implemented by machine integers and how "coercion" functions such as `coerceTm`, whose semantics is the identity, can be optimized away. One might also wish to study whether our "name supplies" can be erased and replaced with a global, effectful "gensym" operation.

Finally, although we have focused exclusively on "well-scoped syntax", it would be desirable to study how to work with syntax that is both "well-scoped" and "well-typed" by construction.

References

Altenkirch, Thorsten. (1993). A formalization of the strong normalization proof for System F in LEGO. *Pages 13 – 28 of:* M. Bezem, J.F. Groote (ed), *Typed Lambda Calculi and Applications*. LNCS 664.

Altenkirch, Thorsten, & Reus, Bernhard. (1999). Monadic presentations of lambda terms using generalized inductive types. *Pages 453–468 of: Computer Science Logic*. Lecture Notes in Computer Science, vol. 1683. Springer.

Atkey, Robert. (2009). Syntax for free: representing syntax with binding using parametricity. *Pages 35–49 of: International Conference on Typed Lambda Calculi and Applications (TLCA)*. Lecture Notes in Computer Science, vol. 5608. Springer.

Atkey, Robert, Lindley, Sam, & Yallop, Jeremy. 2009 (Sept.). Unembedding domain-specific languages. *Pages 37–48 of: Haskell symposium*.

Aydemir, Brian, Charguéraud, Arthur, Pierce, Benjamin C., Pollack, Randy, & Weirich, Stephanie. 2008 (Jan.). Engineering formal metatheory. *Pages 3–15 of: ACM Symposium on Principles of Programming Languages (POPL)*.

Aydemir, Brian E., Bohannon, Aaron, Fairbairn, Matthew, Foster, J. Nathan, Pierce, Benjamin C., Sewell, Peter, Vytiniotis, Dimitrios, Washburn, Geoffrey, Weirich, Stephanie, & Zdancewic, Steve. (2005). Mechanized metatheory for the masses: The POPLMARK challenge. *International Conference on Theorem Proving in Higher-Order Logics (TPHOLs)*. Lecture Notes in Computer Science, vol. 3603. Springer.

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

Bernardy, Jean-Philippe, Jansson, Patrik, & Paterson, Ross. (2010). Parametricity and dependent types. *Pages 345–356 of: Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*. ICFP '10. New York, NY, USA: ACM.

Bird, Richard, & Paterson, Ross. (1999). de Bruijn notation as a nested datatype. *Journal of Functional Programming*, **9**(1), 77–91.

Cave, Andrew, & Pientka, Brigitte. 2012 (Jan.). Programming with binders and indexed data-types. *Pages 413–424 of: ACM Symposium on Principles of Programming Languages (POPL)*.

Charguéraud, Arthur. (2011). The locally nameless representation. *Journal of Automated Reasoning*, 1–46. 10.1007/s10817-011-9225-2.

Chen, Chiyan, & Xi, Hongwei. 2003a (June). Implementing typeful program transformations. *Pages 20–28 of: ACM Workshop on Evaluation and Semantics-Based Program Manipulation (PEPM)*.

Chen, Chiyan, & Xi, Hongwei. 2003b (Aug.). Meta-programming through typeful code representation. *Pages 275–286 of: ACM International Conference on Functional Programming (ICFP)*.

Cheney, James. 2005 (Sept.). Scrap your nameplate. *ACM International Conference on Functional Programming (ICFP)*.

Chlipala, Adam. 2007 (June). A certified type-preserving compiler from lambda calculus to assembly language. *Pages 54–65 of: ACM Conference on Programming Language Design and Implementation (PLDI)*.

Danielsson, Nils Anders. (2011). *Agda standard library*. `http://wiki.portal.chalmers.se/agda/agda.php?n=Libraries.StandardLibrary`.

de Bruijn, Nicolaas G. (1972). 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.

de Bruijn, Nicolaas G. (1991). Telescopic mappings in typed lambda calculus. *Information and Computation*, **91**(2), 189–204.

Donnelly, Kevin, & Xi, Hongwei. (2005). Combining higher-order abstract syntax with first-order abstract syntax in ATS. *Pages 58–63 of: ACM Workshop on Mechanized Reasoning about Languages with Variable Binding*.

Fiore, Marcelo, Plotkin, Gordon, & Turi, Daniele. (1999). Abstract syntax and variable binding (extended abstract). *Pages 193–202 of: IEEE Symposium on Logic in Computer Science (LICS)*. Computer Society Press.

Floyd, R. W. (1967). Assigning meanings to programs. *Pages 19–32 of: Mathematical Aspects of Computer Science*. Proceedings of Symposia in Applied Mathematics, vol. 19.

Guillemette, Louis-Julien, & Monnier, Stefan. (2008). A type-preserving compiler in Haskell. *ACM International Conference on Functional Programming (ICFP)*.

Harper, Robert, Honsell, Furio, & Plotkin, Gordon. (1993). A framework for defining logics. *Journal of the ACM*, 194–204.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM*, **12**(10), 576–580.

Huet, Gérard. (1997). The zipper. *Journal of Functional Programming*, **7**(5), 549–554.

Licata, Daniel R., & Harper, Robert. 2009 (Sept.). A universe of binding and computation. *Pages 123–134 of: ACM International Conference on Functional Programming (ICFP)*.

McBride, Conor. (2001). *The derivative of a regular type is its type of one-hole contexts*. Unpublished.

McBride, Conor. (2003). First-order unification by structural recursion. *Journal of Functional Programming*, **13**(6), 1061–1075.

McBride, Conor, & McKinna, James. (2004). The view from the left. *J. Funct. Program.*, **14**(January), 69–111.

McBride, Conor, & Paterson, Ross. (2008). Applicative programming with effects. *Journal of Functional Programming*, **18**(1), 1–13.

Mitchell, John C. (1986). Representation independence and data abstraction. *Pages 263–276 of: ACM Symposium on Principles of Programming Languages (POPL)*.

Norell, Ulf. 2007 (September). *Towards a practical programming language based on dependent type theory*. Ph.D. thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden.

Pfenning, Frank, & Lee, Peter. (1989). LEAP: A language with eval and polymorphism. *Pages 345–359 of: International Joint Conference on Theory and Practice of Software Development (TAPSOFT)*. Lecture Notes in Computer Science, vol. 352. Springer.

Pientka, Brigitte. 2008 (Jan.). A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. *Pages 371–382 of: ACM Symposium on Principles of Programming Languages (POPL)*.

Pitts, Andrew M. (2006). Alpha-structural recursion and induction. *Journal of the ACM*, **53**, 459–506.

Pitts, Andrew M. 2010 (Jan.). Nominal System *T*. *Pages 159–170 of: ACM Symposium on Principles of Programming Languages (POPL)*.

Pollack, Randy, Sato, Masahiko, & Ricciotti, Wilmer. (2011). A canonical locally named representation of binding. May, 1–23.

Poswolsky, Adam, & Schürmann, Carsten. (2008). Practical programming with higher-order encodings and dependent types. *Pages 93–107 of: European Symposium on Programming (ESOP)*. Lecture Notes in Computer Science, vol. 4960. Springer.

Poswolsky, Adam, & Schürmann, Carsten. (2009). System description: Delphin – A functional programming language for deductive systems. *Electronic Notes in Theoretical Computer Science*, **228**, 113–120.

Pottier, François. 2006 (Mar.). An overview of Cαml. *Pages 27–52 of: ACM Workshop on ML*. Electronic Notes in Theoretical Computer Science, vol. 148, no. 2.

Pottier, François. 2007 (July). Static name control for FreshML. *Pages 356–365 of: IEEE Symposium on Logic in Computer Science (LICS)*.

Pouillard, Nicolas. (2011a). Nameless, painless. *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*. ICFP '11. ACM.

Pouillard, Nicolas. (2011b). *Nompa (Agda code)*. `http://tiny.nicolaspouillard.fr/NomPa.agda`.

Pouillard, Nicolas. 2012 (Jan.). *A unifying approach to safe programming with first-order syntax with binders*. Ph.D. thesis, Université Paris 7.

Pouillard, Nicolas, & Pottier, François. (2010). A fresh look at programming with names and binders. *Pages 217–228 of: Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*. ICFP '10. New York, NY, USA: ACM.

Reynolds, John C. (1983). Types, abstraction and parametric polymorphism. *Pages 513–523 of: Information Processing 83*. Elsevier Science.

Sato, Masahiko, & Pollack, Randy. (2010). External and internal syntax of the lambda-calculus. *J. Symb. Comput.*, **45**(5), 598–616.

Schürmann, Carsten, Poswolsky, Adam, & Sarnat, Jeffrey. (2005). The ∇-calculus: Functional programming with higher-order encodings. *Pages 339–353 of: International Conference on Typed Lambda Calculi and Applications (TLCA)*. Lecture Notes in Computer Science, vol. 3461. Springer.

Shinwell, Mark R., Pitts, Andrew M., & Gabbay, Murdoch J. 2003 (Aug.). FreshML: Programming with binders made simple. *Pages 263–274 of: ACM International Conference on Functional Programming (ICFP)*.

Stark, Ian. 1994 (Dec.). *Names and higher-order functions*. Ph.D. thesis, University of Cambridge. Also available as Technical Report 363, University of Cambridge Computer Laboratory.

Wadler, Philip. 1989 (Sept.). Theorems for free! *Pages 347–359 of: Conference on Functional Programming Languages and Computer Architecture (FPCA)*.

Weirich, Stephanie, Yorgey, Brent, & Sheard, Tim. (2011). Binders unbound. *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*. ICFP '11. ACM.