A Fresh Look at Programming with Names and Binders

Nicolas Pouillard François Pottier
INRIA
{nicolas.pouillard,francois.pottier}@inria.fr

Abstract

A wide range of computer programs, including compilers and theorem provers, manipulate data structures that involve names and binding. However, the design of programming idioms which allow performing these manipulations in a safe and natural style has, to a large extent, remained elusive.

In this paper, we present a novel approach to the problem. Our proposal can be viewed either as a programming language design or as a library: in fact, it is currently implemented within Agda. It provides a safe and expressive means of programming with names and binders. It is abstract enough to support multiple concrete implementations: we present one in nominal style and one in de Bruijn style. We use logical relations to prove that "well-typed programs do not mix names with different scope". We exhibit an adequate encoding of Pitts-style nominal terms into our system.

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

General Terms Design, Languages, Theory

Keywords names, binders, meta-programming, name abstraction, higher-order abstract syntax

1. Introduction

A wide range of computer programs, including compilers and theorem provers, manipulate and transform data structures that involve names and bindings. Significant effort has been invested in the design of programming idioms or languages that support these tasks in a safe and natural style. Nevertheless, a definitive solution is yet to be found. One challenge is to abstract away the details of any one particular implementation technique, such as atoms and permutations, or de Bruijn indices and shifting. A greater still challenge is to design a lightweight yet expressive static discipline to ensure that names are handled in a sound way.

One must first ask: what does it mean to handle names in a sound way? The question is trickier than it seems. There are several informal slogans that attempt to describe what this means:

1. *"name abstractions cannot be violated"*; or: *"the representations of two α-equivalent terms cannot be distinguished"*;
2. *"names do not escape their scope"*;
3. *"names with different scopes cannot be mixed"*.

These slogans are not equivalent; we have listed them in increasing order of strength. A traditional representation of names as strings or de Bruijn indices satisfies none of these slogans. A system such as FreshML [22] satisfies only the first slogan. A strongly-typed representation of names as well-scoped de Bruijn indices satisfies the first two slogans, but, we argue (§4.5), not the third one. Finally, several systems in the literature [10, 12, 16–18, 21], as well as the one presented in this paper, satisfy all three.

Our approach This paper describes a new way of addressing these challenges. We present an interface composed of a number of types and operations for declaring and manipulating data structures that involve names and binding. This interface can be viewed either as a library or as a programming language design.

In support of the "library" point of view, we provide an implementation as a library within Agda. It could also be implemented within another variant of type theory, such as Coq. Our implementation exploits dependent types to express internal invariants and to guarantee that certain operations cannot fail.

In support of the "language" point of view, our proposal could also be viewed as an extension of a standard calculus, such as System F^{ω}, with new primitive types and operations. The types of our primitive operations do not involve dependency.

As far as the programmer is concerned, our types and operations remain abstract. In particular, the nature of names is not revealed. As a result, multiple implementations of our interface are possible. We currently have two: one is based on atoms, in the style of FreshML, while the other is based on de Bruijn indices.

In summary, we propose a novel approach to programming with names and binders. The semantics of the system is elementary: it rests upon a number of explicit, low-level primitive operations. No renaming, shifting, or substitution are built into the semantics. The programmer is offered an abstract view of names, independent of the chosen implementation scheme. One original feature of our proposal is that name abstraction is not primitive: it is built out of more elementary notions. This helps understand the essence of name abstraction, and increases the system's expressiveness by allowing programmers to build custom forms of name abstractions.

Overview of the paper In order to control the use of names, we introduce an abstract notion of *world*. The type system associates a world with each name, and allows two names to be compared for equality only if they inhabit a common world. Names, worlds, as well as a number of other types and operations, are introduced in §3. At the same time, the system is explained via examples of increasing complexity.

In §4, we describe our two implementation schemes. In the nominal scheme, worlds are sets of atoms. In the de Bruijn-index-based scheme, worlds are integer bounds. We then justify the soundness of our interface. We do this twice: once for each implementation scheme. In each case, we make novel use of logical relations in order to give richer meaning to worlds: we explain

ICFP'10, September 27–29, 2010, Baltimore, Maryland, USA.

how worlds can be viewed as bijections between names. In this setting, the fundamental theorem of logical relations corresponds to the three desired slogans. The slogans remain informal, though, because our system does not have a notion of "α-equivalence", or "scope", to begin with. We do prove that nominal terms in the style of Pitts are adequately encoded in our system; this yields a formal version of slogan 1 with respect to Pitts' notion of α-equivalence.

In our interface, a number of key primitive operations are provided only at names, and must be explicitly lifted (by the programmer) to user-defined data types. In §5, we show how to do this, and suggest that some of this boilerplate code can be automatically produced via generic programming. We conclude with an advanced example 6 and with discussions of related work (§7) and future work (§8).

2. A brief introduction to Agda notation

Throughout the paper, our definitions are presented in the syntax of Agda. In Agda, Set is the type of small types like Bool, Maybe (List Bool), or $\mathbb{N}$. Set_1 is the type of Set. The function space is written A → B, while the dependent function space is written ∀ (x : A) → B. An implicit parameter, introduced via ∀ {x : A} → B, can be omitted at a call site if its value can be inferred from the context. There are shortcuts for introducing multiple arguments at once or for omitting a type annotation, as in ∀ {A} {i j : A} x → ....

Existential quantification is available via the type constructor ∃, which accepts a type function as its argument, as in $\exists\lambda\alpha \to (\alpha \to \mathbb{N}) \times (\mathbb{N} \to \alpha)$.

A data constructor name can be used in multiple data types. Agda makes use of type annotations to resolve ambiguities.

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

Agda is strict about whitespace: x+y is an identifier, whereas x + y is an application. This allows naming a variable after its type (deprived of any whitespace). We use mixfix declarations, such as _⊕_. We use some definitions from Agda's standard library: operations over functors (_⟨$⟩_), monads (return, _≫=_), and applicative functors (pure, _⊛_).

For the sake of conciseness, the code fragments presented in the paper are sometimes not perfectly self-contained. However, a complete Agda development is available online [19].

3. Working with names and binders

We now present the signature (that is, the abstract types and operations) that our system offers to programmers. For the sake of presentation, we intersperse fragments of this signature (declarations) with examples of their use (code fragments).

Worlds We first introduce *worlds*, which names inhabit. There is an empty world, which no names inhabit. There are no other concrete worlds: most of the time, the programmer uses world variables α, β, γ, δ.

```
World : Set
ø     : World
```

We often use relations Rel, and in particular relations over worlds Rel World.

```
Rel : Set → Set₁
Rel A = ∀ (α β : A) → Set
```

Agda does not have a clear phase distinction, that is, a clear distinction between values and types. Nevertheless, one can also view our system as an extension of a calculus that does have this distinction, such as System F^ω under a type-erasure semantics. In that view, worlds, like types, can be erased at runtime.

Names The type of names, Name, is indexed with a world. The idea is that two names can safely be compared only if they inhabit a common world. This is apparent in the type of the name equality test.

```
Name     : ∀ (α : World) → Set
_≟Name_ : ∀ {α} → Name α → Name α → Bool
```

To witness the fact that no name inhabits the empty world, we introduce a function which produces a contradiction when applied to a name in the empty world. Its codomain is the empty type ⊥. Put differently, this function allows marking some cases as impossible, and instructs the system to statically check that they are indeed so.

```
¬nameø : Name ø → ⊥
```

Weak links Let us go on to our next ingredient: a type for *weak links* between worlds.

```
_↚_ : Rel World
```

If α and β are worlds, then $\alpha \nleftarrow \beta$ is a type. Roughly speaking, a name x has type $\alpha \nleftarrow \beta$ under two conditions: first, x inhabits the world β; second, the world that existed before x was introduced is α. Put another way, $\alpha \nleftarrow \beta$ is a more precise type for names. It keeps track of the worlds just before and just after a name is bound. We usually refer to α as the "outer" world and to β as the "inner" world. Weak links allow keeping track of connections between worlds: intuitively speaking, if x has type $\alpha \nleftarrow \beta$, then the worlds α and β assign the same meaning to every name other than x. The name x itself may have no meaning at all in α, or it may have some meaning in α and a different meaning in β. Our weak links do not require x to be fresh for α: they allow a new binding to shadow an earlier binding. Later on, we introduce strong links, which do imply a freshness condition.

Since a weak link is just a more precise type for a name, we offer a way of converting the former into the latter.

```
nameOf↚ : ∀ {α β} → α ↚ β → Name β
```

Example: representing λ-terms We now have enough elements to declare algebraic data types that involve names and binders. Let us begin with an explicitly typed object-language: the untyped λ-calculus with local (**let**) definitions.

```
data Tm (α : World) : Set where
  V   : ∀ (x : Name α) → Tm α
  _·_ : ∀ (t u : Tm α) → Tm α
  ƛ   : ∀ {β} (x : α ↚ β) (t : Tm β) → Tm α
  Let : ∀ {β} (x : α ↚ β) (t : Tm α) (u : Tm β) → Tm α
```

The type constructor Tm is indexed with a world. The type Tm α can be thought of as a type of terms whose free names inhabit the world α. Accordingly, the constructor V carries a name that inhabits α, and the constructor for applications carries two subterms that inhabit α. The constructor ƛ shows how we build simple name abstractions. It carries a weak link (the name to be bound) between the outer world α and some inner world β. The body of the abstraction inhabits this inner world: it has type Tm β. The abstraction itself inhabits the outer world: it has type Tm α. Since β does not occur in the latter type, it is really existentially quantified: viewed from the outside, an abstraction contains an unknown inner world. In Let, the sub-term t inhabits the outer world α: thus, it is not in the scope of the bound name x. On the other hand, the subterm u inhabits the inner world β: it is in the scope of x. It is easy to see how one would define LetRec.

Using worlds, names, and weak links, it is possible to define a wide range of data structures with binders. The above encoding of λ-terms is but one instance of a general encoding of Pitts' nominal terms and nominal signatures. We describe this general encoding and prove it adequate in §4.3. In fact, our system is more expressive than Pitts': this is illustrated by several examples in this section.

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. This unaltered induction also tells a bit about the expressiveness of such functions. It is also efficient: no renaming, substitution, or shifting is involved. Polymorphic recursion is exploited: the call to size t in the ƛ case is at some inner world.

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

Exporting names When two worlds are connected via a weak link, it is desirable to be able to move names from one world into the other along the link. We introduce the function export$_{\leftarrow}$ for this purpose.

```
export← : ∀ {α β} → α ←− β → Name β → Maybe (Name α)
```

The function export$_{\leftarrow}$ expects two names x and y, whose types are $\alpha \leftarrow\!\!- \beta$ and Name β. It compares (nameOf$_{\leftarrow}$ x) and y for equality. If they are equal, export$_{\leftarrow}$ fails: the name y has meaning in β, but may not have meaning, or may have a different meaning, in α. If they differ, export$_{\leftarrow}$ succeeds and returns y at type Name α: indeed, since y is not x, it has the same meaning in both worlds. export$_{\leftarrow}$ is a partial function: it can fail. It is an injective function: if export$_{\leftarrow}$ x y and export$_{\leftarrow}$ x z are equal, then y and z are equal. Like $\_ \stackrel{?}{=}_{\mathsf{Name}} \_$, export$_{\leftarrow}$ is a name equality test. However, it performs type refinement: in the event that the names differ, the input name is returned with a more precise type.

The reader may wonder whether it is possible to move a name in the other direction, from the outer world α into the inner world β. The answer is negative: this would be unsound. (For a justification, see the discussion of dubious in §4.2.) Later on, we introduce a means of moving in this direction, namely world inclusion witnesses.

Example: working with free and bound names We now have enough tools to present a more interesting example, namely a function that 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—a value of type List (Name β). This list cannot be returned, and this is fortunate, since doing so would let the bound variable leak out of its scope! We define an auxiliary function, rm, which removes all occurrences of a name in a list of names and at the same time performs type refinement in the style of export$_{\leftarrow}$.

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

rm : ∀ {α β} → α ←− β → List (Name β) → List (Name α)
rm _ []                  = []
rm x (y :: ys) with export← x y
...            | just y  = y :: rm x ys
...            | nothing = rm x ys
```

The function rm applies export$_{\leftarrow}$ x to every name y in the list and builds a list of only those that successfully cross the link x. It exhibits a typical way of using export$_{\leftarrow}$ to perform a name comparison together with a type refinement. This idiom is recurrent in the programs that we have written.

The function fv enjoys a free theorem, that is, a theorem that follows directly from its type: every name in the output list must occur free in the input term. This claim is backed up by the typed models in §4.2 and §4.4.

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 ≟Name y)
  where
    OccEnv : World → Set
    OccEnv α = Name α → Bool
    extend : ∀ {α β} → α ←− β → OccEnv α → OccEnv β
    extend x Γ y = maybe Γ false (export← x y)

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

The function occurs tests whether some name x occurs free in a term. An environment Γ is carried down, augmented when a binder is crossed, and looked up at variables. Here, this environment is represented as a function of type Name $\alpha \to$ Bool. Although this is a simple and elegant representation, others exist. For instance, we could represent the environment as a linked list of weak links: the code for this variant is online [19]; see also below.

We claim that this code is standard and uncluttered. There is no hidden cost: no renaming is involved. Admittedly, linked lists are not the most efficient representation of environments. It would be nice to be able to implement environments using, say, balanced binary search trees, while preserving well-typedness. We leave this issue to future study.

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 would cause a type error. In the definition of extend, attempting to check whether y occurs in Γ without first comparing y and x would cause a type error. In our nominal implementation scheme (§4.1), it is permitted for newer bindings to shadow earlier ones; our type discipline guarantees that the code works also in that case.

As suggested previously, one may wish to represent environments as an explicit data structure (a linked list of weak links) rather than as an opaque object (a lookup function). While there exists an appropriate abstraction in Agda's standard library called Star, we define a custom data type. An environment is a chain of weak links. At runtime, it is just a list of names.

```
data _⋆←_ (T : Rel World) : Rel World where
  ε   : ∀ {α} → α ⋆← α
  _◁_ : ∀ {α β γ} (x : β ←− γ) (Γ : α ⋆← β) → α ⋆← γ
```

The export$_{\leftarrow}$ operation is extended to chains of weak links:

```
export⋆← : ∀ {α β} → α ⋆← β → Name β → Maybe (Name α)
export⋆← ε       y = just y
export⋆← (x ◁ Γ) y = export← x y >>= export⋆← Γ
  where open MaybeMonad
```

The type $\alpha \mathrel{\star\!\!\leftarrow} \beta$ is the type of an environment, or environment fragment, whose outer world is α and whose inner world is β. The expression export$_{\leftarrow}$ Γ y looks up the name y in the environment Γ. The name y must make sense in the scope of Γ, that is, y must

inhabit the world β. If y is found among the bindings, then the information associated with y can be returned. (Here, there is no such information, so nothing is returned.) If y is not found among the bindings, then y is returned, with a more precise type: indeed, since y is not among the names introduced by Γ, it makes sense outside Γ, that is, in the world α.

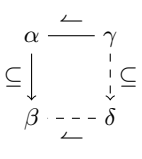
We illustrate the use of chains of weak links with an alternative definition of the function fv. This variant avoids the need to take the bound atoms off the list by not inserting them in the first place. At variables, we use $\mathsf{export}_{\star\leftarrow}$ to check whether the name is free or bound. 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 x.

```
fv' : ∀ {β α} → α ⋆← β → Tm β → List (Name α)
fv' Γ (V x)       = List.fromMaybe (export⋆← Γ x)
fv' Γ (t · u)     = fv' Γ t ++ fv' Γ u
fv' Γ (ƛ x t)     = fv' (x ◁ Γ) t
fv' Γ (Let x t u) = fv' Γ t ++ fv' (x ◁ Γ) u
```

Importing names We now introduce *world inclusion witnesses* $\_\subseteq\_$, whose purpose is to allow moving names from a smaller world into a larger world. In other words, we equip worlds with a system of explicit subtyping. World inclusion is reflexive and transitive; the type constructor Name is covariant; the empty world is the least world.

```
_⊆_       : Rel World
⊆-refl    : ∀ {α} → α ⊆ α
⊆-trans   : ∀ {α β γ} → α ⊆ β → β ⊆ γ → α ⊆ γ
import⊆   : ∀ {α β} → α ⊆ β → Name α → Name β
ø-bottom-⊆ : ∀ {α} → ø ⊆ α
```

Like links, world inclusion witnesses come with an $\mathsf{import}_\subseteq$ function, which moves a name from one world into the other. One major difference with weak links is that this function is total.



While importing names is nice and simple, we are interested, in general, in importing complex terms or data structures from one world into another. This requires, in particular, being able to import abstractions. Upon close examination, we find that we need this commutative diagram to hold.

We make this property available to the programmer by introducing the following primitive operation:

```
_←-commute-⊆_ : ∀ {α β γ} → α ← γ → α ⊆ β
                          → ∃ λ δ    → γ ⊆ δ × β ← δ
```

At this point, it is probably not clear why this commutative diagram is sound, or why it is useful. Its soundness – as well as that of every primitive operation presented here – is justified in §4. Its usefulness is illustrated in §5.2 and §5.3.

Strong links Next, we introduce *strong links*. Again, the type $\alpha \hookleftarrow \beta$ is a precise type for a name: it is more precise than $\alpha \leftarrow \beta$ and (therefore) more precise than Name β. If x has type $\alpha \hookleftarrow \beta$, then x is guaranteed to be fresh for the world α. That is, a strong link represents the introduction of a binding for a fresh name, and (in contrast with a weak link) cannot possibly shadow an earlier binding. As a result, if x has type $\alpha \hookleftarrow \beta$, then $\alpha \subseteq \beta$ must hold: out of a strong link, one can produce a world inclusion witness.

```
_↩_      : Rel World
weaken   : ∀ {α β} → α ↩ β → α ← β
dropName : ∀ {α β} → α ↩ β → α ⊆ β
```

Technically, a strong link comes with an even stricter guarantee: the name x must not just be fresh for α; it must dominate every atom in α, in a sense to be made precise later on (§4.1).

One might wonder why we need both weak links and strong links. Why not use strong links everywhere, since they offer a stronger guarantee? The answer is: precisely because they are stronger than weak links, strong links are also more difficult to construct. In particular, strong links do not enjoy an analogue of the diagram _←-commute-⊆_. Such a diagram would be unsound, because a name that is fresh for a smaller world is not necessarily fresh for a larger world. Yet, a commutative diagram in the style of _←-commute-⊆_ plays a key role in the definition of generalized import operations (§5.3). This explains why we often use weak links in our term representations, such as Tm.

Generating names The alert reader may have noticed that, up to this point, we have not yet introduced a way of producing names or links! To address this issue, we need a mechanism for producing fresh names. We find that it is sufficient to be able to produce strong links, since a strong link can degenerate into a weak link and into a name. We view a *fresh name* with respect to the world α as a strong link into some unspecified next world β, and define the following abbreviation:

```
Fresh    : World → Set
Fresh α = ∃ λ β → α ↩ β
```

We introduce two primitive operations for creating fresh names. freshø is an initial strong link—a name that is fresh for the empty world. $\mathsf{next}_\hookrightarrow$ accepts two names: one is a weak link between two worlds α and β; the other is fresh for α. $\mathsf{next}_\hookrightarrow$ produces a name that is fresh for β.

```
freshø  : Fresh ø
next↪ : ∀ {α β γ} → α ← β → α ↩ γ → Fresh β
```

Together, these two low-level operations allow constructing an infinite stream of fresh names, that is, a name generator.

Packaging up We are done introducing the abstract types and operations that we offer to the users of our library (or programming language). In summary, we have four primitive types (names, weak links, strong links, and world inclusion witnesses), and a number of operations over these types. In the Agda implementation, we find it convenient to package each type together with the operations that it offers. An idiomatic way of doing this involves defining parameterized records, like this:

```
module NamePack {β} (x : Name β) where
  nameOf : Name β
  nameOf = x

module WeakPack {α β} (x : α ← β) where
  open NamePack (nameOf← x) public
  weakOf     : α ← β
  weakOf     = x
  exportWith : Name β → Maybe (Name α)
  exportWith = export← x

module ⊆Pack {α β} (x : α ⊆ β) where
  ⊆Of        : α ⊆ β
  ⊆Of        = x
  importWith : Name α → Name β
  importWith = import⊆ x

module StrongPack {α β} (x : α ↩ β) where
  open WeakPack (weaken x) public
  open ⊆Pack (dropName x) public
  strongOf : α ↩ β
  strongOf = x
  nextOf   : Fresh β
  nextOf   = next↪ weakOf strongOf

module FreshPack {α} (x : Fresh α) where
  open StrongPack (proj₂ x) public
```

The open/public declarations cause one record to be included within another. This permits a limited form of inheritance and overloading. For instance, within the scope of appropriate **open**

declarations, the method nameOf is applicable to names of type Name α, $\alpha \leftharpoonup \beta$, $\alpha \hookrightarrow \beta$, and Fresh α.

Constructing terms Once this boilerplate is set up, we at last show how to construct a term. For example, let us build a representation of the object-level term $\lambda\, x\, y \to x\, y$.

```
app : Tm ø
app = ƛ (weakOf x) (ƛ (weakOf y)
        (V (importWith y (nameOf x)) · V (nameOf y)))
  where open FreshPack
        x = freshø
        y = nextOf x
```

We generate two fresh names x and y. Each of these names is viewed as a weak link (via weakOf) when playing the role of a binding occurrence and is viewed as a name (via nameOf) when playing the role of a regular occurrence. Furthermore, in order to satisfy the type-checker, the regular occurrence of x must be imported into the scope of y.

This is admittedly fairly difficult to read. If our system was implemented as a stand-alone programming language, as opposed to a library within Agda, it seems reasonable to think that one would be able to make the invocations of weakOf, nameOf, and importWith implicit. The omitted information would be reconstructed by a local type inference algorithm.

Towards elaborate uses of worlds The type Tm is just one basic example of an algebraic data type that involves names and binders. As a more challenging example, consider a type C of one-hole contexts associated with Tm. The type C is indexed with two worlds, which respectively play the roles 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:

```
module Context where
  data C : World → 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 γ} α ↚ β → C β γ → C α γ
    Let1  : ∀ {β γ} α ↚ β → C α γ → Tm β → C α γ
    Let2  : ∀ {β γ} α ↚ β → Tm α → C β γ → C α γ
```

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 chain of weak links that connects the outer and inner worlds: these links are carried by the constructors ƛ and Let_2.

Then a context and a term can be paired to produced a term in a context:

```
CTm : World → Set
CTm α = ∃ λ 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 ($\forall\ \{\alpha\} \to$ Tm $\alpha \to$ CTm α), which 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 role played by C α β in this existential type is identical to that played by $\alpha \leftharpoonup \beta$ in the single-name abstraction $\exists\ \lambda\ \beta \to \alpha \leftharpoonup \beta \times$ Tm β. In other words, the type C α β can be viewed as a new, user-defined type of links between worlds, and can be used to build elaborate forms of name abstractions.

As another instance of this idea, if one wished to extend our object language with ML-style patterns, one would index the type Pat of patterns with an outer world and an inner world, and one would use elaborate abstractions of the form $\exists\ \lambda\ \beta \to$ Pat $\alpha\ \beta \times$ Tm β.

Finally, the fact that a type can be indexed with several world parameters can be exploited in other ways. For instance, if one wished to extend our object language with polymorphism, one would index the type Tm with two worlds: one for (names of) term variables, one for (names of) type variables. In other words, worlds can also serve as disjoint name spaces.

4. Two sound implementations

We have axiomatized a number of notions, including worlds, names, and links. Now comes the time to give definitions of these types and terms. We have two versions of these definitions, that is, two Agda implementations of our library. One is in nominal style: it is based on atoms. The other is based on well-scoped de Bruijn indices. Both implementations can be found online [19].

Either of these implementations is well-typed in Agda: this guarantees that well-typed client programs of our library cannot go wrong. However, type soundness is not the whole story: we also wish to prove that well-typed client programs must respect name abstraction. For each of the two models, we establish this property via a logical relations argument.

4.1 The nominal model: implementation

We posit a countably infinite set of atoms $\mathbb{A}$, equipped with a notion of equality. In our Agda implementation, atoms are natural numbers and we make use of ordering of natural numbers for fresh name generation; this is apparent in the semantics of strong links below.

In the nominal model, a world is a set of atoms. In the Agda implementation, such as set is represented as a list without duplicates. A name of type Name α is an atom a together with a proof that a is a member of the world α.

A weak link of type $\alpha \leftharpoonup \beta$ is an atom a together with a proof of the equation $\beta \equiv \alpha \cup \{\mathsf{a}\}$. That is, the world β is the union of the world α and of the atom a. It is important to note that a may or may not be a member of α: a weak link permits shadowing.

Like a weak link, a strong link of type $\alpha \hookrightarrow \beta$ includes an atom a, as well as a proof of the equation $\beta \equiv \alpha \cup \{\mathsf{a}\}$. Furthermore, it contains a proof of the fact that the natural number a is a strict upper bound for the set α. This condition reflects the fact that the name a is fresh for the world α. It implies, and is stronger than, $\mathsf{a} \notin \alpha$. Technically, this extra strength is exploited in the definition of $\mathsf{next}_{\hookrightarrow}$, where we need to guarantee that, if a is fresh for α, then the successors of a form an infinite stream of names that are fresh for α.

A world inclusion witness of type $\alpha \subseteq \beta$ has no computational content: it is just a proof of the set-theoretic inclusion $\alpha \subseteq \beta$.

In the nominal model, the operations $\mathsf{import}_{\subseteq}$, $\mathsf{nameOf}_{\leftharpoonup}$, weaken, dropName and _↚-commute-⊆_ have no computational content. $\_ \stackrel{?}{=}_{\mathrm{Name}} \_$ is an atom equality test. $\mathsf{export}_{\leftharpoonup}$ also involves an atom equality test: it fails if its arguments are equal and returns its second argument otherwise. The function call $\mathsf{next}_{\hookrightarrow}$ a b produces the maximum of the two integers 1 + a and b, so that, if b is fresh for some world α, then $\mathsf{next}_{\hookrightarrow}$ a b is fresh for the world $\alpha \cup \{\mathsf{a}\}$.

4.2 The nominal model: logical relations

Although the implementation described above guarantees type soundness in a traditional sense, this is not sufficient to guarantee that names are handled in a sound way. Indeed, it would be possible to extend this implementation of the library with operations that are well-typed but intuitively do not make sense.

For instance, the above interpretation of worlds validates the fact that, out of a weak link of type $\alpha \leftharpoonup \beta$, one can extract a proof of the inclusion $\alpha \subseteq \beta$. Yet, extending the system with an operation dubious : $\forall\ \{\alpha\ \beta\} \to \alpha \leftharpoonup \beta \to \alpha \subseteq \beta$, implemented

as the identity, would be intuitively unsound. To see this, consider a free atom x and an atom abstraction (λ x y t), whose respective types are Name α and Tm α. The bound atom y has type $\alpha \leftharpoonup \beta$, where β is the inner world of the abstraction. In the presence of dubious, it would become possible to use $\mathsf{import}_{\subseteq}$ to cast the atom x to the type Name β, with undesirable consequences. First, one would then be able to compare the atoms x and y for equality, so that the identity of a bound name would become observable: name abstractions would be violated. Second, one would be able to build a new name abstraction whose bound atom is y and whose body contains a free occurrence of x: this would lead to name capture in the event that x and y happen to be the same atom.

In the following, we remedy this problem by prodiving a richer interpretation of worlds in a nominal setting. We interpret a world no longer as a set of atoms, but as a *partial bijection between atoms*. On top of this, we carry out a standard logical relations construction. These logical relations validate all of the operations of §3, as implemented in §4.1, while rejecting dubious.

The definitions and proofs in this section are informal, in the sense that they have not been machine-checked.

Definition 4.1 *A* relation *between atoms is a subset of* $\mathbb{A} \times \mathbb{A}$. *We write* $a_1\ (\alpha)\ a_2$ *when the pair* (a_1, a_2) *is in the relation* α. *A* partial bijection *between atoms is a relation such that* $a_1\ (\alpha)\ a_2$ *and* $b_1\ (\alpha)\ b_2$ *imply* $(a_1 = b_1 \iff a_2 = b_2)$. ◇

The following notions are used in the interpretation of weak links and strong links, respectively.

Definition 4.2 *The* shadowing extension *of a partial bijection* α *with an atom pair* (b_1, b_2), *written* $(b_1, b_2) \boxplus \alpha$, *is the partial bijection such that* $a_1\ ((b_1, b_2) \boxplus \alpha)\ a_2$ *holds if and only if either* $a_1 \equiv b_1 \wedge a_2 \equiv b_2$ *or* $a_1 \not\equiv b_1 \wedge a_2 \not\equiv b_2 \wedge a_1\ (\alpha)\ a_2$. ◇

The *domain* $\mathrm{dom}(\alpha)$ of a relation α is defined as the set of atoms $\{a_1 \mid \exists a_2, a_1\ (\alpha)\ a_2\}$. Its *codomain* $\mathrm{codom}(\alpha)$ is defined analogously. If A is a set of atoms, we write $b > A$ to indicate that the atom b is a strict upper bound for the set A.

Definition 4.3 *The* fresh extension *of a partial bijection* α *with an atom pair* (b_1, b_2), *written* $(b_1, b_2)\alpha$, *is defined only if* $b_1 > \mathrm{dom}(\alpha)$ *and* $b_2 > \mathrm{codom}(\alpha)$. *When it is defined,* $(b_1, b_2)\alpha$ *is the partial bijection* $\{(b_1, b_2)\} \cup \alpha$. ◇

When the fresh extension exists, it coincides with the shadowing extension.

We assume that the host language of our system supports the construction of logical relations in a standard manner. For instance, the host language may be System F or System F^ω, where logical relations are well-understood [13]. At every type, two relations are defined: a relation between values and a relation between terms. We write $v\ (\tau)\ w$ when the values v and w are related at type τ; we write $t\ (\tau)\ u$ when the terms t and u are related at type τ. We assume that the host language provides the definition of these relations at every standard type-theoretic connective (functions; universal and existential quantifiers; products, sums, unit). We also assume that equivalence of two terms at type τ is defined, independently of τ, in terms of the operational behavior of these terms and in terms of equivalence of two values at type τ.

We now extend this construction by defining what it means for two values to be equivalent at our new primitive types: names, weak links, strong links, and world inclusion witnesses.

Definition 4.4 *At base types, the logical relation is defined by:*

$$\begin{array}{rcl} a_1\ (\mathsf{Name}\ \beta)\ a_2 & \iff & a_1\ (\beta)\ a_2 \\ a_1\ (\alpha \leftharpoonup \beta)\ a_2 & \iff & \beta \equiv (a_1, a_2) \boxplus \alpha \\ a_1\ (\alpha \hookrightarrow \beta)\ a_2 & \iff & \beta \equiv (a_1, a_2)\alpha \\ ()\ (\alpha \subseteq \beta)\ () & \iff & \alpha \subseteq \beta \end{array}$$ ◇

Two atoms a_1 and a_2 are related at type Name β if and only if the pair (a_1, a_2) is in the partial bijection β. They are related at type $\alpha \leftharpoonup \beta$ if and only if β is the shadowing extension of α with the pair (a_1, a_2). They are related at type $\alpha \hookrightarrow \beta$ if and only if β is the fresh extension of α with the pair (a_1, a_2). Last, two unit values are related at type $\alpha \subseteq \beta$ if and only if the relation α is a subset of the relation β.

Note that $a_1\ (\alpha \hookrightarrow \beta)\ a_2$ implies $a_1\ (\alpha \leftharpoonup \beta)\ a_2$, which itself implies $a_1\ (\mathsf{Name}\ \beta)\ a_2$. Thus, it is sound to turn a strong link into a weak one, and a weak link into a name. That is, it is sound to implement the operations $\mathsf{nameOf}_{\leftharpoonup}$ and weaken as the identity. In the case of these two operations, the proof of Theorem 4.5 (below) boils down to this simple remark. Similarly, in view of this interpretation, the operation dropName, which produces an inclusion witness out of a strong link, is clearly sound.

We can now point out why a "binary" interpretation (worlds as partial bijections between atoms) is finer-grained than a "unary" interpretation (worlds as sets of atoms). Indeed, in order to check that the operation dubious, implemented as the identity, is sound, we would need to check that $a_1\ (\alpha \leftharpoonup \beta)\ a_2$ implies $()\ (\alpha \subseteq \beta)\ ()$. That is, we would need to check that $\beta \equiv (a_1, a_2) \boxplus \alpha$ implies $\alpha \subseteq \beta$. However, due to the possibility of shadowing, this is not in general the case: this implication is false when $a_1 \in \mathrm{dom}(\alpha)$ and $a_2 \notin \mathrm{codom}(\beta)$ and when $a_1 \notin \mathrm{dom}(\alpha)$ and $a_2 \in \mathrm{codom}(\beta)$. It is worth noting that $\beta \equiv (a_1, a_2) \boxplus \alpha$ does imply $\mathrm{dom}(\alpha) \subseteq \mathrm{dom}(\beta)$ and $\mathrm{codom}(\alpha) \subseteq \mathrm{codom}(\beta)$. This explains why dubious seemed safe in a unary interpretation.

There remains to establish the fundamental theorem of logical relations. The proof of this theorem is provided by the host language; we need only extend it with one new case for each of our primitive operations.

Theorem 4.5 *Every primitive operation* p *of type* τ *is related to itself at type* τ. ◇

Proof. For the sake of brevity, we provide only one representative case, namely the case of $\mathsf{export}_{\leftharpoonup}$. The goal is to show that $\mathsf{export}_{\leftharpoonup}$ is related to itself at type $\forall\ \{\alpha\ \beta\} \to \alpha \leftharpoonup \beta \to \mathsf{Name}\ \beta \to \mathsf{Maybe}\ (\mathsf{Name}\ \alpha)$. By definition of the logical relation at the standard connectives ($\forall$, $\to$, Maybe) and at our primitive types (Definition 4.4), the goal boils down to:

> if $\beta \equiv (a_1, a_2) \boxplus \alpha$ and $b_1\ (\beta)\ b_2$ hold, then the terms $(\mathsf{export}_{\leftharpoonup}\ a_1\ b_1)$ and $(\mathsf{export}_{\leftharpoonup}\ a_2\ b_2)$ are related at type $\mathsf{Maybe}\ (\mathsf{Name}\ \alpha)$.

Thus, let us assume that $\alpha, \beta, a_1, a_2, b_1, b_2$ are as above. Now, a key remark is this: the hypotheses $\beta \equiv (a_1, a_2) \boxplus \alpha$ and $b_1\ (\beta)\ b_2$, together with the fact that β is bijective, imply $a_1 = b_1 \iff a_2 = b_2$. This remark allows us to distinguish only two cases:

- *Case* $a_1 = b_1 \wedge a_2 = b_2$. Then, the terms $\mathsf{export}_{\leftharpoonup}\ a_1\ b_1$ and $\mathsf{export}_{\leftharpoonup}\ a_2\ b_2$ both reduce to nothing. Because the value nothing is related to itself at type $\mathsf{Maybe}\ (\mathsf{Name}\ \alpha)$, the goal holds.
- *Case* $a_1 \neq b_1 \wedge a_2 \neq b_2$. Then, the terms $\mathsf{export}_{\leftharpoonup}\ a_1\ b_1$ and $\mathsf{export}_{\leftharpoonup}\ a_2\ b_2$ respectively reduce to just b_1 and just b_2. We must prove that these two values are related at type $\mathsf{Maybe}\ (\mathsf{Name}\ \alpha)$. This boils down to proving that b_1 and b_2 are related by α. It is easy to check that this goal does follow from the hypotheses $b_1\ (\beta)\ b_2$, $\beta \equiv (a_1, a_2) \boxplus \alpha$, $a_1 \neq b_1$ and $a_2 \neq b_2$. □

The intuition behind the above proof case is: the success or failure of an $\mathsf{export}_{\leftharpoonup}$ operation does not depend on earlier choices of bound names. More precisely, if we run a single program twice, with different but related inputs, it is impossible for an $\mathsf{export}_{\leftharpoonup}$ operation to succeed in one run and fail in the other.

One implication of Theorem 4.5 is that "choices of fresh names do not matter". Our Agda implementation of the operation $\mathsf{next}_{\hookrightarrow}$,

which we use to produce fresh names, is of course deterministic. However, one could in principle equip $\mathsf{next}_{\hookrightarrow}$ with a non-deterministic semantics, whereby $\mathsf{next}_{\hookrightarrow}\ \mathsf{a}\ \mathsf{b}$ produces an arbitrarily chosen integer that is greater than or equal to the maximum of $1 + \mathsf{a}$ and b. Under this semantics, Theorem 4.5 still holds: related programs produce related results. In other words, non-determinism in the choice of fresh names is not observable by well-typed programs. One could in fact abandon $\mathsf{next}_{\hookrightarrow}$ and introduce an expression fresh, of type $\forall\ \{\alpha\} \to \exists\ \lambda\ \beta \to \alpha\ \hookrightarrow\ \beta$, which reduces to an arbitrary atom that dominates the world α. Under this semantics, again, Theorem 4.5 holds. In a type-erasure implementation of our design, where worlds do not exist at runtime, fresh could be efficiently implemented using global state (that is, a gensym).

Another implication is that "name abstractions cannot be violated". It is perhaps not clear, at first, what this means, especially in light of the fact that our name abstractions are not primitive: they are built out of more elementary constructs. One way of formalizing this statement is to prove that our system permits an adequate encoding of nominal terms [14]: this guarantees that our name abstractions behave as intended. We do so below (§4.3).

This adequacy result shows that our system is able to encode a standard notion of α-equivalence. However, one should keep in mind that our system is more expressive than Pitts' nominal terms and nominal types: it offers many types of data structures with names and binding that do not lie in the image of the encoding. Logical relations tell us what "α-equivalence" means at these types.

4.3 Adequacy of an encoding of nominal terms

Definition 4.6 *The nominal types and nominal terms are:*

$$\begin{array}{rcl} \tau & ::= & \mathsf{atom} \mid \tau \times \tau \mid \tau + \tau \mid \langle \mathsf{atom} \rangle \tau \\ t & ::= & a \mid (t,t) \mid \mathit{inj}_i\ t \mid \langle a \rangle t \end{array} \qquad \diamond$$

For the sake of simplicity, we do not deal with recursive types, but one could extend our argument to do so. Note that the atom a is *not* considered bound in the nominal term $\langle a \rangle t$.

Definition 4.7 *The free atoms of a nominal term are defined by:*

$$\begin{array}{ll} \mathrm{fa}(a) = \{a\} & \mathrm{fa}((t_1,t_2)) = \mathrm{fa}(t_1) \cup \mathrm{fa}(t_2) \\ \mathrm{fa}(\mathit{inj}_i\ t) = \mathrm{fa}(t) & \mathrm{fa}(\langle a \rangle t) = \mathrm{fa}(t) \setminus \{a\} \end{array} \qquad \diamond$$

Definition 4.8 *α-equivalence of two nominal terms at a nominal type is defined as follows:*

$$a \equiv a : \mathsf{atom} \qquad \frac{t_1 \equiv t_2 : \tau \qquad t_1' \equiv t_2' : \tau'}{(t_1, t_1') \equiv (t_2, t_2') : \tau \times \tau'}$$

$$\frac{t_1 \equiv t_2 : \tau_i}{\mathit{inj}_i\ t_1 \equiv \mathit{inj}_i\ t_2 : \tau_1 + \tau_2} \qquad \frac{(a_1\ c)t_1 \equiv (a_2\ c)t_2 : \tau \qquad c \mathbin{\#} \langle a_1 \rangle t_1, \langle a_2 \rangle t_2}{\langle a_1 \rangle t_1 \equiv \langle a_2 \rangle t_2 : \langle \mathsf{atom} \rangle \tau}$$

We use $a \mathbin{\#} t$ as a short-hand for $a \notin \mathrm{fa}(t)$. We write $(a\ b)t$ for the result of swapping all occurrences of the atoms a and b through t. $\diamond$

Definition 4.9 *The encodings of nominal types into our types, and of nominal terms into our extension of System F^ω, are as follows.*

$$\begin{array}{rcl} [\![\mathsf{atom}]\!]_\alpha & = & \mathsf{Name}\ \alpha \\ [\![\tau_1 \times \tau_2]\!]_\alpha & = & [\![\tau_1]\!]_\alpha \times [\![\tau_2]\!]_\alpha \\ [\![\tau_1 + \tau_2]\!]_\alpha & = & [\![\tau_1]\!]_\alpha + [\![\tau_2]\!]_\alpha \\ [\![\langle \mathsf{atom} \rangle \tau]\!]_\alpha & = & \exists \beta.(\alpha \leftharpoonup \beta \times [\![\tau]\!]_\beta) \end{array}$$

$$\begin{array}{rcl} \lceil a \rceil & = & a \\ \lceil (t_1,t_2) \rceil & = & (\lceil t_1 \rceil, \lceil t_2 \rceil) \\ \lceil \mathit{inj}_i\ t \rceil & = & \mathit{inj}_i\ \lceil t \rceil \\ \lceil \langle a \rangle t \rceil & = & \mathsf{pack}(a, \lceil t \rceil) \end{array} \qquad \diamond$$

In order to prove that this encoding is adequate, we wish to prove that two nominal terms are α-equivalent if and only if their encodings are in the logical relation. α-equivalence of nominal terms is defined in terms of total atom permutations, while our worlds are partial atom bijections. The following technical definition helps bridge the gap.

Definition 4.10 *Let A_1 and A_2 be sets of atoms. Let π_1 and π_2 be permutations (that is, total, bijective relations over atoms). Let α be a world, that is, a partial, bijective relation over atoms. We say that the permutations π_1 and π_2 correspond to the world α, with respect to the domains A_1 and A_2, if and only if*

$$(\pi_1; \pi_2^{-1}) \cap (A_1 \times A_2) = \alpha \cap (A_1 \times A_2) \qquad \diamond$$

The following technical lemma shows how our notion of correspondence crosses a name abstraction. The proof of the theorem that follows is then straightforward.

Lemma 4.11 *Let π_1 and π_2 correspond to α with respect to $A_1 \setminus \{a_1\}$ and $A_2 \setminus \{a_2\}$. Let c be fresh for $\pi_1(A_1 \setminus \{a_1\})$ and $\pi_2(A_2 \setminus \{a_2\})$. Let β stand for the world $(a_1, a_2) \boxplus \alpha$. Then, $(\pi_1 a_1\ c) \circ \pi_1$ and $(\pi_2 a_2\ c) \circ \pi_2$ correspond to β with respect to A_1 and A_2.* $\diamond$

Theorem 4.12 *If π_1 and π_2 correspond to α with respect to $\mathrm{fa}(t_1)$ and $\mathrm{fa}(t_2)$, then the following two propositions are equivalent:*

$$\begin{array}{ll} \pi_1 t_1 \equiv \pi_2 t_2 : \tau & \textit{(α-equivalence of nominal terms)} \\ \lceil t_1 \rceil\ ([\![\tau]\!]_\alpha)\ \lceil t_2 \rceil & \textit{(logical relation between terms)} \end{array} \qquad \diamond$$

The proofs appears in the extended version of this paper [20]. As a corollary, if a nominal term t has nominal type τ, and if α is a set of atoms that includes $\mathrm{fa}(t)$, then $\lceil t \rceil$ has type $[\![\tau]\!]_\alpha$ in our unary interpretation (§4.1). Furthermore, we have:

Corollary 4.13 *Let t_1 and t_2 be nominal terms of nominal type τ such that $\mathrm{fa}(t_1) = \mathrm{fa}(t_2) = \emptyset$. Then, t_1 and t_2 are α-equivalent if and only if their encodings are related with respect to the empty world. That is, $t_1 \equiv t_2 : \tau$ holds if and only if $\lceil t_1 \rceil\ ([\![\tau]\!]_\emptyset)\ \lceil t_2 \rceil$ holds.* $\diamond$

Corollary 4.14 *Let t_1 and t_2 be nominal terms of nominal type τ such that $\mathrm{fa}(t_1) = \mathrm{fa}(t_2) = \emptyset$. If $t_1 \equiv t_2 : \tau$ holds, then $\lceil t_1 \rceil$ and $\lceil t_2 \rceil$ are observationally equivalent at type $[\![\tau]\!]_\emptyset$.* $\diamond$

Corollary 4.14 shows that our programming language respects object-level α-equivalence. In other words, our name abstractions behave as intended: the identity of the bound name cannot be observed.

The reverse implication – if t_1 and t_2 are not α-equivalent, then their encodings can be distinguished by some well-typed observer – can be established by implementing an α-equivalence test within the programming language (see §5.1) and by proving that it is correct. We have not yet carried out this proof.

4.4 The de Bruijn model: implementation

In the de Bruijn model, a world is just a natural number n, while a name is just a natural number in the interval $[0, \mathsf{n})$. That is, World is $\mathbb{N}$, and Name is Fin, where $\mathsf{Fin}\ \mathsf{n}$ is the type of the natural numbers that are less than n. In this model, weak and strong links are the same. Indeed, contrary to the nominal model, there is no shadowing: it is impossible for a new binding to hide a previous one. A link of type $\alpha \leftharpoonup \beta$ or $\alpha\ \hookrightarrow\ \beta$ has no computational content: it is just a proof of the equation $\beta \equiv \alpha + 1$. On the other hand, a world inclusion witness of type $\alpha \subseteq \beta$ does have computational content: it is a natural number k such that the equation $\beta \equiv \alpha + \mathsf{k}$ holds. The integer k represents the amount by which one must shift when importing a name from α into β.

Most operations have straightforward semantics: $\_ \stackrel{?}{=}_{\mathsf{Name}\ \_}$ is an integer equality test (in fact, it is an equality test at type $\mathsf{Fin}\ \mathsf{n}$);

$\mathsf{export}_{\leftarrow}$ is the predecessor function, which fails if applied to the index 0; $\mathsf{import}_{\subseteq}$ and $\subseteq$-trans are integer addition; $\mathsf{nameOf}_{\leftarrow}$ is the constant function zero; and $\mathsf{dropName}$ is the constant function one. The functions weaken, $\mathsf{next}_{\hookrightarrow}$ and $\_{\leftarrow}$-$\mathsf{commute}$-$\subseteq\_$ have no computational content.

In the de Bruijn model, importing or exporting a term has a cost. Importing requires copying the term to increment its free names, while exporting requires copying and decrementing. (In the nominal model, in contrast, importing is the identity, while exporting requires an occurs check and is the identity when successful. The price to pay for this is that explicit renamings can be necessary.) Fortunately, thanks to the expressiveness of the type system, the programmer is in control of the import/export machinery, and has access to many of the classic tricks for dealing efficiently with de Bruijn indices. It is possible, for instance, to delay certain imports, and to cheaply combine multiple imports into one, since $\subseteq$-trans is just integer addition.

4.5 The de Bruijn model: logical relations

Like the nominal model, the de Bruijn model is implemented [19] in Agda. This guarantees that our de Bruijn indices range over the expected intervals, and, more generally, that a well-typed client program cannot go wrong. However, as in the nominal model, this is not sufficient to guarantee that our implementation is correct in an intuitive sense. Again, there are operations that are well-typed in this unary interpretation of the de Bruijn model, yet do not make sense.

For instance, imagine $\mathsf{import}_{\subseteq}$ is implemented as $\lambda\ \_\ \mathsf{x} \to \mathsf{x}$, instead of integer addition. This amounts to forgetting to shift, and is clearly a mistake. Yet, this version of $\mathsf{import}_{\subseteq}$ is well-typed, because if x has type $\mathsf{Fin\ m}$ then, for every n greater than or equal to m, x also has type $\mathsf{Fin\ n}$. (This argument is slightly over-simplified. In reality, a coercion is needed: see $\mathsf{Data.Fin.inject+}$ in Agda's standard library.)

As another instance, imagine $\mathsf{export}_{\leftarrow}$ is implemented as the function that fails when applied to the index 0 and returns 0 otherwise. Again, this is meaningless: this function is not even injective! Yet, this version of $\mathsf{export}_{\leftarrow}$ is well-typed, because if x has type $\mathsf{Fin\ m}$ then 0 has type $\mathsf{Fin\ m}$.

In light of these examples, we claim that, perhaps contrary to popular belief, *well-scopedness of de Bruijn indices is not good enough:* it does not guarantee that indices are correctly adjusted where needed.

Again, our solution lies in the construction of logical relations that validate our implementation, while rejecting the incorrect implementations mentioned above.

At this point, the reader may ask: do logical relations have anything non-trivial to say about the de Bruijn model? In the nominal model, logical relations were used to compare two program runs and to show that their outcome is insensitive to choices in the data representation (in particular, choices of bound names) and in the semantics (choices of fresh names). In the de Bruijn model, however, both bound names and fresh names are chosen in a canonical manner, so one might think that there is no interesting comparison to be made. In fact, there is. De Bruijn's representation does carry an arbitrary component in its choice of *free* names. The logical relations argument tells us that a well-typed program is insensitive to choices of free names: if one applies some permutation to the free names in its input, one observes the same permutation in the free names of its output. This property is stronger than well-scopedness of de Bruijn indices; in particular, it is not satisfied by the incorrect implementations mentioned above.

As in the nominal case §4.2, we view a world as a partial bijection between names. This time, names are de Bruijn indices, that is, natural numbers. In the de Bruijn model, there is no shadowing, so there is no analogue of shadowing extension (Definition 4.2). The analogue of fresh extension (Definition 4.3) is the following:

Definition 4.15 *The* shift *of a partial bijection α, written $\alpha\uparrow$, is a partial bijection, characterized as follows:*

$$\alpha\uparrow \quad \stackrel{def}{=} \quad \{(0,0)\} \cup \{(i_1+1, i_2+1) \mid (i_1, i_2) \in \alpha\} \qquad \diamond$$

We write $\alpha\uparrow^k$ for the result of shifting the world α k times.

Definition 4.16 *At base types, the logical relation is defined by:*

$$\begin{array}{rcl} i_1\ (\mathsf{Name}\ \beta)\ i_2 & \iff & i_1\ (\beta)\ i_2 \\ ()\ (\alpha \leftarrow \beta)\ () & \iff & \beta \equiv \alpha\uparrow \\ ()\ (\alpha \hookrightarrow \beta)\ () & \iff & \beta \equiv \alpha\uparrow \\ k\ (\alpha \subseteq \beta)\ k & \iff & \beta \equiv \alpha\uparrow^k \end{array} \qquad \diamond$$

Theorem 4.17 *Every primitive operation p of type τ is related to itself at type τ.* $\diamond$

5. Programmable operations

Several of our primitive operations, such as $\_\stackrel{?}{=}_{\mathsf{Name}\_}$, $\mathsf{export}_{\leftarrow}$, and $\mathsf{import}_{\subseteq}$, operate upon names only. This is a good thing, insofar as it simplifies the meta-theory of our system. However, it is desirable to lift these operations to user-defined data types, such as Tm, so that user-defined terms can be compared for equality (up to α-equivalence) and exported or imported from one world into another. Fortunately, this can be done within the system: for a large class of algebraic data types, these generalized forms of the primitive operations can be programmed up. We now explain how to do so in the particular case of Tm. Where details are omitted, the reader is referred to the code [19].

5.1 Deciding α-equivalence

We sketch how to implement a function that tests whether two terms are α-equivalent. As before (§3), we use environments represented as chains of weak links. We modify the function $\mathsf{export}_{\star\leftarrow}$ (§3) to obtain a new function, $\mathsf{index}\star\leftarrow$, which accepts a name and classifies it as either free or bound in the environment. In the former case, like $\mathsf{export}_{\star\leftarrow}$, $\mathsf{index}\star\leftarrow$ produces a copy of the name at a more precise type. In the latter case, it converts the name to a de Bruijn index.

$$\mathsf{index}\star\leftarrow\ :\ \forall\ \{\alpha\ \beta\} \to \alpha\ \star\leftarrow\ \beta \to \mathsf{Name}\ \beta \to \mathsf{Name}\ \alpha \uplus \mathbb{N}$$

Thus equipped, it is straightforward to write a recursive comparison function of type $\forall\ \{\alpha\ \beta\ \gamma\} \to (\alpha\ \star\leftarrow\ \beta) \times (\alpha\ \star\leftarrow\ \gamma) \to \mathsf{Tm}\ \beta \to \mathsf{Tm}\ \gamma \to \mathsf{Bool}$. At abstractions, the two environments are extended with the two bound names at hand. At variables, $\mathsf{index}\star\leftarrow$ is used to classify each of the two names at hand, and the results produced by $\mathsf{index}\star\leftarrow$ are compared for equality—which is possible because both have type $\mathsf{Name}\ \alpha \uplus \mathbb{N}$.

Once applied to two empty environments, the comparison function has type $\forall\ \{\alpha\} \to \mathsf{Tm}\ \alpha \to \mathsf{Tm}\ \alpha$.

5.2 A generic traversal function

We are interested in several operations that move a term from one world to another. These operations are implemented by traversing the term and building a new term. Much of the traversal code can be shared between these operations, and it is beneficial to do so, as this sheds a more abstract light on the traversal.

In general, a traversal function has a type of the following form:

$$\begin{array}{l} \mathsf{Traverse}\ :\ (\_ \rightsquigarrow \_\ :\ \mathsf{Rel\ World})\ (\mathsf{M}\ :\ \mathsf{Set} \to \mathsf{Set}) \\ \qquad\qquad (\mathsf{F}\ :\ \mathsf{World} \to \mathsf{Set}) \to \mathsf{Set} \\ \mathsf{Traverse}\ \_\rightsquigarrow\_\ \mathsf{M\ F}\ =\ \forall\ \{\alpha\ \beta\} \to \alpha \rightsquigarrow \beta \to \mathsf{F}\ \alpha \to \mathsf{M}\ (\mathsf{F}\ \beta) \end{array}$$

The parameter F describes the data structure that is being traversed and copied: for instance, F could be Tm. The parameter M is

typically either the identity or the Maybe monad. The former is used when implementing an operation that cannot fail; the latter is used when implementing an operation that can fail. More generally, M can be an arbitrary applicative functor [11]. The parameter $\_\rightsquigarrow\_$ indicates what kind of connection is expected between the world α, which describes the input data structure, and the world β, which the output data structure inhabits. For instance and as a first approximation, when implementing an export operation along a weak link, this parameter could be the type constructor for weak links, reversed, flip $\_\leftharpoonup\_$. In reality, the implementation of an export operation needs to maintain an environment that keeps track of the binders that have been entered. In general, $\alpha \rightsquigarrow \beta$ is the type of this environment, so it is more complex than just a single link. Still, it helps to think of it as the type of an abstract link between the input and output worlds.

In the implementation of a traversal function, upon entering a name abstraction, we need this abstract link, whose type is $\alpha \rightsquigarrow \beta$, to commute with the weak link that represents the binding occurrence of the abstraction, whose type is $\alpha \leftharpoonup \gamma$. The following two definitions respectively describe a general commutative diagram and the particular diagram that is needed here:

```
ComposeCommute : (_⇝₁_ _⇝₂_ : Rel World) → Set
ComposeCommute _⇝₁_ _⇝₂_
  = ∀ {α β γ} → α ⇝₁ β → β ⇝₂ γ
      → ∃ λ δ → α ⇝₂ δ × δ ⇝₁ γ
Comm : (_⇝_ : Rel World) → Set
Comm _⇝_ = ComposeCommute (flip _↼_) _⇝_
```

We now present a generic traversal function for the type Tm. In addition to the above parameters, it requires a function onName, which describes what to do at non-binding occurrences of names.

The traversal is straightforward. The parameter Γ is the abstract link between the worlds α and β. At variables, onName is used. At abstractions, the commutative diagram comm is used. This produces a new abstract link Γ' between α' and β', where α' is the inner world of the abstraction that is being deconstructed, and β' is the inner world of the abstraction that is being constructed. This new link is used in the recursive call. The diagram also produces a new weak link x', of type $\beta \leftharpoonup \beta$', which is used in the construction of the new abstraction. The applicative functor machinery is used everywhere, so as to perform effect propagation behind the scenes.

```
module TraverseTm
    {M ⇝} (appli : Cat.RawApplicative M)
    (comm : Comm ⇝)
    (onName : Traverse ⇝ M Name) where
  open Cat.RawApplicative appli

  traverseTm : Traverse ⇝ M Tm
  traverseTm Γ (V x)
    = V ⟨$⟩ onName Γ x
  traverseTm Γ (t · u)
    = _·_ ⟨$⟩ traverseTm Γ t ⊛ traverseTm Γ u
  traverseTm Γ (ƛ x t)
    with comm x Γ
  ... | (_ , Γ' , x')
    = ƛ x' ⟨$⟩ traverseTm Γ' t
  traverseTm Γ (Let x t u)
    with comm x Γ
  ... | (_ , Γ' , x')
    = Let x' ⟨$⟩ traverseTm Γ t ⊛ traverseTm Γ' u
```

Thanks to the generic programming facilities of a language with dependent types, it should be possible to implement this generic traversal not just for Tm, but for any algebraic data type that is composed of unit, pairs, sums, names, and name abstractions. We have not yet done so.

5.3 Applications of the generic traversal

Generalized import Whereas the primitive operation $\mathsf{import}_{\subseteq}$ moves a single name from one world to another, a generalized import function moves a data structure from one world to another. For instance, a generalized import function for Tm has type $\forall\ \{\alpha\ \beta\} \to \alpha \subseteq \beta \to \mathsf{Tm}\ \alpha \to \mathsf{Tm}\ \beta$. In other words, this function witnesses the fact that Tm is covariant in its index.

To implement such a function, we instantiate the generic traversal function traverseTm. Because importing never fails, an appropriate applicative functor is the identity. The type of abstract links $\_\rightsquigarrow\_$ is instantiated with $\_\subseteq\_$, the type of world inclusion witnesses. The parameters comm and onName are instantiated with the primitive operations $\mathsf{import}_{\subseteq}$ and $\_\leftharpoonup$-commute-$\subseteq\_$.

Generalized export A generalized export function for Tm has type $\forall\ \{\alpha\ \beta\} \to \alpha \leftharpoonup \beta \to \mathsf{Tm}\ \beta \to \mathsf{Maybe}\ (\mathsf{Tm}\ \alpha)$. It fails if its first argument, a name, occurs free in its second argument, a term. Otherwise, it returns a copy of its second argument at the outer world α.

We found generalized export more difficult to implement than generalized import. The reason is, the commutative diagram that we would like to use, which involves two weak links and has type Comm (flip $\_\leftharpoonup\_$), appears to be unsound: it is not validated by our logical relations. We do have a work-around, but it involves freshening, that is, replacing the bound names of the input term with fresh names.

Again, we instantiate the generic traversal function traverseTm. Because exporting can fail, an appropriate applicative functor is Maybe. The type of abstract links $\alpha \rightsquigarrow \beta$ is instantiated with $\mathsf{Fresh}\ \beta \times (\mathsf{Name}\ \alpha \to \mathsf{Maybe}\ (\mathsf{Name}\ \beta))$. This means that, during the traversal, we maintain: (i) a fresh name generator for the output world β; and (ii) an environment, that is, a partial mapping of names in the input world α to names in the output world β. Upon entering an abstraction that binds some name x, this environment is extended by mapping x to a fresh name. (This is enough to implement the required commutative diagram, of type Comm $\_\rightsquigarrow\_$, for this particular definition of $\_\rightsquigarrow\_$.) Upon reaching a variable y, the environment is consulted. During this lookup, one of two situations arises. If y is bound in the environment, then the corresponding fresh name is returned. Otherwise, y is a free name of the term that we are attempting to export, so y is submitted to the primitive $\mathsf{export}_{\swarrow}$ operation (which may fail), whose result is returned.

Checking whether a term is closed A closed term inhabits the empty world, and inhabits every world. That is, both of the types $\mathsf{Tm}\ \emptyset$ and $\forall\ \{\alpha\} \to \mathsf{Tm}\ \alpha$ accurately describe closed terms. These types are interconvertible. To convert $\mathsf{Tm}\ \emptyset$ into $\forall\ \{\alpha\} \to \mathsf{Tm}\ \alpha$, one uses the subtyping axiom $\emptyset$-bottom-$\subseteq$ as well as the fact that Tm is covariant in its index. To convert $\forall\ \{\alpha\} \to \mathsf{Tm}\ \alpha$ into $\mathsf{Tm}\ \emptyset$, one instantiates α with $\emptyset$.

Terms that admit the above types are particularly easy to use, because, thanks to polymorphism, they can be freely moved to any world. Of course, the flip side of the coin is, it is somewhat difficult to create such terms. To help in this task, a useful tool is a function closeTm that checks at runtime whether a term is closed and, when it succeeds, returns a term that is statically known to be closed. Such a function should have type $\forall\ \{\alpha\} \to \mathsf{Tm}\ \alpha \to \mathsf{Maybe}\ (\mathsf{Tm}\ \emptyset)$, or equivalently, $\forall\ \{\alpha\ \beta\} \to \mathsf{Tm}\ \alpha \to \mathsf{Maybe}\ (\mathsf{Tm}\ \beta)$.

At the base type Name, such a function is easy to implement. The function const nothing, which always fails, fits the bill: it has type $\forall\ \{\alpha\ \beta\} \to \mathsf{Name}\ \alpha \to \mathsf{Maybe}\ (\mathsf{Name}\ \beta)$. Whereas an export link fails at one particular name and lets every other name through, this function can be viewed as a link that always fails.

With this in mind, implementing closeTm is simple. The construction is identical to that of the generalized export function

above, except in the setup phase, where an export link is replaced with a link that always fails.

6. 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 [10, 14, 22]. The challenge lies in the way the algorithm mixes computational functions, name abstractions, and fresh name generation.

The object language of interest is 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 (we do not ensure normal forms for conciseness reasons), they can be given a common definition, which is parameterized over the representation of 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 $(\mathsf{World} \to \mathsf{Set}) \to (\mathsf{World} \to \mathsf{Set})$: it is an indexed-type transformer.

In order to obtain the syntactic representation, we instantiate Abs with the abstractions that we have used throughout this paper: an abstraction is an existential package of a weak link and of a term that inhabits the inner world. This yields the type Term of syntactic terms.

```
SynAbs : (World → Set) → World → Set
SynAbs F α = ∃ λ β → α ↞ β × F β

open M SynAbs 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. Sem is not an inductive data type; fortunately, with `--no-positivity-check` Agda accepts this type definition, at the cost of breaking strong normalization.

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

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

It is important to note that our semantic name abstractions involve bounded polymorphism in a world: we define SemAbs F α as $\forall\ \{\beta\} \to \alpha \subseteq \beta \to \mathsf{F}\ \beta \to \mathsf{F}\ \beta$, as opposed to the more naïve $\mathsf{F}\ \alpha \to \mathsf{F}\ \alpha$. This provides a more accurate and more flexible description of the behavior of substitution. Furthermore, this has the important effect of making SemAbs (and Sem) covariant with respect to the parameter α, which would not be the case with the naïve definition. In other words, it is possible to define a generalized import operation for semantic terms:

```
impSem : ∀ {α β} → α ⊆ β → Sem α → Sem β
impSem ⊆w (V a)   = V (import⊆ ⊆w a)
impSem ⊆w (ƛ f)   = ƛ (λ ⊆w' v → f (⊆-trans ⊆w ⊆w') v)
impSem ⊆w (t · u) = impSem ⊆w t · impSem ⊆w u
```

At a semantic abstraction, no recursive call is performed, because the body of the abstraction is opaque: it is a computational function f. Instead, we exploit the transitivity of world inclusion and build a new semantic abstraction that inhabits the desired world.

The normalization by evaluation algorithm is parameterized with a representation of environments. The type of environments takes the form Env A α β, where A is the type of the data carried in the environment and α and β are the outer and inner worlds of the environment. Environments must offer the following constants and operations: empty (emptyEnv), lookup (lookupEnv); extension (_,_↦_); map (mapEnv); covariance of Env with respect to its parameter α (importEnv⊆). These requirements are expressed by the type ImportableEnvPack, whose definition is omitted.

```
open ImportableEnvPack envPack
```

The algorithm uses an environment whose type takes the form Env (Sem α) α β. To a name, such an environment associates a semantic term that lies outside the scope of the environment. This type is, again, covariant in α, as witnessed by the following import function:

```
impEnv : ∀ {α β γ} → α ⊆ β → Env (Sem α) α γ
                           → Env (Sem β) β γ
impEnv ⊆w = importEnv⊆ ⊆w ∘ mapEnv (impSem ⊆w)
```

The first part of the algorithm evaluates a syntactic term within an environment to produce a semantic term. When evaluating a ƛ-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 import the environment Γ via impEnv.

```
app : ∀ {α} → Sem α → Sem α → Sem α
app (ƛ f) v = f ⊆-refl v
app n     v = n · v

eval : ∀ {α β} → Env (Sem α) α β → Term β → Sem α
eval Γ (V x)   = [V, id] (lookupEnv Γ x)
eval Γ (t · u) = app (eval Γ t) (eval Γ u)
eval Γ (ƛ (_, a, t))
  = ƛ (λ ⊆w v → eval (impEnv ⊆w Γ [a ↦ v]) t)
```

The second part of the algorithm reifies a semantic term back into a 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 fresh name generator.

```
reify : ∀ {α} → Fresh α → Sem α → Term α
reify g (V a)   = V a
reify g (n · v) = reify g n · reify g v
reify g (ƛ f)   = ƛ (_, weakOf g, t)
  where open FreshPack
        t = reify (nextOf g) (f (⊆Of g) (V (nameOf g)))
```

Evaluation under an empty 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.

```
nf : ∀ {α} → Fresh α → Term α → Term α
nf g = reify g ∘ eval emptyEnv
```

7. Related work

The difficulty of programming with, or reasoning about, names and binders has been known for a long time. It has recently received a lot of attention, due in part to the PoplMark challenge [4]. 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" [9].

In the following, we review several programming language designs that are intended to facilitate the manipulation of names and binders. By lack of space, this review cannot be exhaustive: we focus on relatively recent related work.

Distinctions One traditionally distinguishes several broad approaches to the problem, 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 artificial. In fact, our work presents strong connections with

all three schools of thought. Perhaps more important are the following questions:

- *What properties are enforced by the system?* FreshML [22] offers an adequate encoding of nominal terms, but does not prevent a newly generated atom from escaping its scope. Systems based on well-scoped de Bruijn indices enforce the invariant that every name refers to some binding site; however, as we have pointed out (§4.5), this alone does not imply that indices are correctly adjusted where needed. In systems based on higher-order abstract syntax, and in Pure FreshML [18], name manipulation is hygienic by design: this is built in the syntax and semantics of the programming language. In the present paper, hygiene is not built in. We have used logical relations to find out (and prove) which properties can be expected of a well-typed program.
- *Does the system offer "substitution for free", and if so, at which types? Does it have separate data and computation layers?* In FreshML, Pure FreshML [18] and the present work, the answer to both questions is negative. In several systems in the tradition of higher-order abstract syntax, including Elphin [21], Delphin [16, 17] and Beluga [12], the answer to both questions is positive. In Licata and Harper's work [10], substitution is available for free at many types, even though data and computation are not separated.
- *Do names inhabit a fixed type, or do they inhabit every type?* Usually, systems that provide some form of substitution for free [10, 12, 16, 17, 21] allow names to inhabit every type, while so-called nominal systems [18, 22] as well as the present work offer a separate type of names.
- *How does the system keep track of the context or world in which a name makes sense?* In Pure FreshML, there is effectively just one world, within which every name makes sense; the proof obligations 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, contexts are explicit: a data-layer type, once annotated with a context, becomes a computation-layer type. In the present paper, worlds are explicit, and are built into algebraic data type definitions by the programmer.
- *Which very high-level operations does the semantics of the programming language involve?* The semantics of FreshML and Pure FreshML involve renaming. The semantics of Elphin, Delphin, and Beluga 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 our import and export operations, are obtained (at many, but not all, types) via generic programming.

FreshML and Pure FreshML FreshML [22] 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. This makes it easy to write programs in a style that matches informal mathematical practice. FreshML satisfies a correctness property analogous to our Corollary 4.14 – *name abstractions cannot be broken*. However, 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.

Pure FreshML [18] imposes additional proof obligations, which ensure that freshly created atoms do not escape their scope, and correspond to Pitts' *freshness condition for binders* [14]. 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 (like the original FreshML) or using de Bruijn indices. This is an implementation choice, which the programmer need not know about.

The present paper can be viewed as a different way of constructing a safe variant of FreshML. Whereas Pure FreshML supplements ordinary ML types with logical assertions, we explore the use of richer types and do not rely, for the time being, on a separate logic.

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. Here, in contrast, name abstraction is explained in terms of more basic notions; 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, and possibly to greater runtime efficiency: because our nominal compilation scheme permits shadowing, there is, in some cases, no need to pay a price to enforce the property that all names are distinct.

Nominal System $\mathcal{T}$ [15] follows the tradition of FreshML and so guarantees that name abstractions are not violated. However compared to Pure FreshML and our system it does not statically enforce that names do not escape their scope. Instead the `new` construct is introduced to represent such escaped names, dynamically. This is akin to `nan` in floating point computation, they are not numbers but they still have the `float` type and result of mathematically ill founded operations like dividing by zero. So in some sense names do escape their scope but are dynamically turned into harmless values. Whether such programs should have a semantics or should simply be statically rejected is a matter of design.

Well-scoped de Bruijn indices It is by now well-known that 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 [1, 5]. 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 [6] and Chlipala [7]. However, 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 [8] 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 [10] is implemented on top of well-scoped de Bruijn indices. The system presented in this paper can be compiled down to de Bruijn indices, and could thus be viewed as an abstraction layer on top of well-scoped de Bruijn indices. However, as we have pointed out (§4.5), our system offers a stronger guarantee than raw well-scoped de Bruijn indices do. It does not just guarantee that every index is within range: it also guarantees that a well-typed program component is insensitive to permutations of the free indices in its input. In a scenario where programs are type-checked but not proved correct, this extra guarantee could be welcome.

Licata and Harper's system [10] differs from ours in several ways. Perhaps most notably, Licata and Harper aim to provide substitution for free when possible, whereas we don't; and they expose the use of well-scoped de Bruijn indices to the programmer, who must sometimes reason in terms of zero and successor, whereas we do not reveal the nature of names, thus permitting multiple compilation schemes.

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 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 [20], 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.

Elphin, Delphin, Beluga [12, 16, 17, 21] are closely related to one another in several ways. They separate the data and computation layers, which implies that they do not support first-class functions. At the data level, they 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 extremely simple, as it only extends an existing logical relations argument with a few new primitive types and operations.

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. For instance, our implementation of normalization by evaluation (§6) 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 [2, 3] 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 (§6). An exact connection remains to be investigated.

8. Future work

We have presented an abstract programming model, together with two concrete implementations, in nominal style and de Bruijn style. We have argued separately about the correctness of each implementation. In particular, we have proved that the nominal implementation allows an adequate encoding of nominal terms. Ideally, however, the nominal term encoding should be proved adequate directly with respect to the abstract model, not with respect to its implementations. We do not yet know how to do this, because our abstract model does not have a semantics. A related question is: how to carry out specifications and proofs of programs with respect to our abstract programming model?

Acknowledgements Thanks to Randy Pollack, Andrew Pitts, Benoît Montagu, Jean-Philippe Bernardy and the anonymous reviewers for providing us with very valuable feedback.

References

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

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

[3] Robert Atkey, Sam Lindley, and Jeremy Yallop. Unembedding domain-specific languages. In *Haskell symposium*, pages 37–48, September 2009.

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

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

[6] Chiyan Chen and Hongwei Xi. Implementing typeful program transformations. In *ACM Workshop on Evaluation and Semantics-Based Program Manipulation (PEPM)*, pages 20–28, June 2003.

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

[8] Kevin Donnelly and Hongwei Xi. Combining higher-order abstract syntax with first-order abstract syntax in ATS. In *ACM Workshop on Mechanized Reasoning about Languages with Variable Binding*, pages 58–63, 2005.

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

[10] Daniel R. Licata and Robert Harper. A universe of binding and computation. In *ACM International Conference on Functional Programming (ICFP)*, pages 123–134, September 2009.

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

[12] Brigitte Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 371–382, January 2008.

[13] Andrew M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10:321–359, 2000.

[14] Andrew M. Pitts. Alpha-structural recursion and induction. *Journal of the ACM*, 53:459–506, 2006.

[15] Andrew M. Pitts. Nominal System T. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 159–170, January 2010.

[16] Adam Poswolsky and Carsten Schürmann. Practical programming with higher-order encodings and dependent types. In *European Symposium on Programming (ESOP)*, volume 4960 of *Lecture Notes in Computer Science*, pages 93–107. Springer, March 2008.

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

[18] François Pottier. Static name control for FreshML. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 356–365, July 2007.

[19] Nicolas Pouillard and François Pottier. A fresh look at programming with names and binders (Agda code), March 2010. `http://tiny.nicolaspouillard.fr/FreshLookAgda`.

[20] Nicolas Pouillard and François Pottier. A fresh look at programming with names and binders (extended version). `http://tiny.nicolaspouillard.fr/FreshLookExt`, March 2010.

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

[22] Mark R. Shinwell, Andrew M. Pitts, and Murdoch J. Gabbay. FreshML: Programming with binders made simple. In *ACM International Conference on Functional Programming (ICFP)*, pages 263–274, August 2003.