Phantom Names: a Named Interface for de Bruijn Syntax

Mathis Bouverot-Dupuis
Inria Paris, ENS Paris

Yannick Forster
Inria Paris

François Pottier
Inria Paris

1 Introduction

When implementing and formalizing programs which manipulate abstract syntax in proof assistants such as Rocq [19], a key design decision is the choice of variable representation (see e.g. the POPLMark challenge [5]).

De Bruijn indices [11] are a popular option and are used both in the implementation of systems such as Rocq [19] or Agda [15] and in large scale formalizations, e.g. MetaRocq [18]. However, de Bruijn indices also introduce significant complexity compared to paper definitions: index arithmetic (lifts and shifts) is tricky to get right and pollutes definitions.

A notable alternative is the locally nameless representation [9, 13], which is undeniably easier to program with than de Bruijn indices and is used e.g. in the implementation of the Lean [14] theorem prover, as well as in formalizations such as F-ing modules [17] (albeit with mild success according to the authors). However, reasoning about locally nameless syntax requires a lot of boilerplate, and can be tedious even on simple examples (see Section 4).

In this paper we present *phantom names*, a binder representation which lends itself to both programming and proving, combining the strengths of de Bruijn indices and locally nameless syntax.

Contributions.

1. We present *phantom names*, which build upon the ideas of Bernardy and Pouillard [7] but are both simpler to implement and easier to use thanks to dependent types.
2. We illustrate phantom names on examples that are notoriously tricky to get right when using de Bruijn indices.
3. As a first case study, we prove correct a simple stack reduction machine for the λ-calculus using de Bruijn indices, locally nameless, and phantom names. We compare the proof of correctness across each binder representation, showing that phantom names and de Bruijn indices are superior to locally nameless on this example.
4. As a second case study, we use phantom names to formalize the meta-theory of a variant of the calculus of constructions with existential variables, proving standard results such as confluence and subject reduction.

Our code and case studies [1] are written in Rocq, but we expect our insights to carry over to any dependently-typed language which has type classes.

2 Key ideas

Tags and scopes. The core idea behind phantom names is a variation on intrinsically-scoped syntax [3, 8]: we represent variables using de Bruijn indices, and we index terms with a *scope* (rather than a natural number). A `scope` is simply a list of tags, and `tag` is a unit type:

```
Inductive tag := TAG.
Inductive scope := SNil | SCons (s : scope) (x : tag).
```

We use notations `∅` for the empty scope and `s ▷ x` for the extension of `s` with a single tag `x`. If we think of tags as variables and scopes as lists of variables, we can define scope-indexed de Bruijn indices and lambda terms as follows:

```
Inductive index : scope -> Set :=
| I0 {s x} : index (s ▷ x)
| IS {s x} : index s -> index (s ▷ x).

Inductive term (s : scope) :=
| Var (i : index s)
| App (t u : term s)
| Lam (x : tag) (t : term (s ▷ x))
| LetIn (x : tag) (t : term s) (u : term (s ▷ x)).
```

The astute reader might have noticed that `scope` is isomorphic to `nat`, thus our presentation so far is akin to intrinsically-scoped syntax. In fact, tags are computationally irrelevant. However we can leverage them to guide type class inference: any two Rocq variables `x y : tag` are considered distinct during type class inference [2]. Thus, `x` and `y` can be viewed as distinct *names*.

Scope membership and inclusion. We define two type classes: `scope_mem x s` to witness the membership of the tag `x` in the scope `s`, and `scope_incl s s'` to witness the inclusion of the scope `s` in the scope `s'`. Due to space constraints we only discuss the former in the following. One way to encode `scope_mem` is to use a de Bruijn index, although other encodings are possible:

```
Class scope_mem x s := { idx_of : index s }.
Arguments idx_of : x {s _}.
```

We can define instances to infer such witnesses:

```
Instance here s x : scope_mem x (s ▷ x).
Instance there s x y : scope_mem x s -> scope_mem x (s ▷ y).
```

These instances overlap: we assign a higher priority to `here`.

Similarly, we can use `scope_incl` to define a weakening function: `wk {s s'} `{scope_incl s s'} : term s -> term s'.`

Phantom names. These ingredients allow for a programming style where we treat tags as *phantom names*, i.e. as names which guide the inference of de Bruijn indices. We can build some example lambda terms as follows, using a smart constructor `mk_lam` for abstractions and the notation `#x` for `Var (idx_of x)`:

```
Definition mk_lam {s} (body : forall x, term (s ▷ x)) :=
  Lam TAG (body TAG).
Definition id : term ∅ :=
  mk_lam (fun x => #x).
```

[1] All examples and case studies can be found at https://github.com/MathisBD/phantom-names-rocqshop

[2] Rocq's conversion check does not include an η-law for unit types such as `tag`. The lack of an η-law for unit-like inductives is not fundamental to our technique, we could instead define `tag` as an opaque alias of `unit`.

```
Definition proj1 : term ∅ :=
  mk_lam (fun x => mk_lam (fun y => #x)).
```

The calls to `idx_of` trigger type class inference on the goals `scope_mem x (∅ ▷ x)` and `scope_mem x (∅ ▷ x ▷ y)`, which are resolved using the two instances `here` and `there`.

3 Examples: various translations

We illustrate phantom names on a few examples which are notoriously tricky to get right using de Bruijn indices:

1. A CPS translation [16] for the λ-calculus.
2. A unary parametricity translation for a pure type system, based on the presentation by Bernardy et al. [6].
3. A translation from the λ-calculus to the π-calculus studied by Accattoli et al. [2]

We show the CPS translation below (lambda terms are defined as in Section 2) and refer the reader to our supplementary material for the other translations:

```
Fixpoint cps {s} (t k : term s) : term s :=
  match t with
  | Var i => App k (Var i)
  | Lam x t =>
      App k (Lam x (mk_lam (fun k' => cps (wk t) #k')))
  | App t u =>
      cps t (mk_lam (fun x =>
      cps (wk u) (mk_lam (fun y =>
        App (App #x #y) (wk k)))))
  | LetIn x t u =>
      cps t (mk_lam (fun v =>
        LetIn x #v (cps (wk u) (wk k))))
  end.
```

The CPS function is not structurally recursive, and in fact we disable the guard checker for simplicity. Proving termination is easy but orthogonal to our work.

More importantly, notice that we do not need to do any arithmetic on de Bruijn indices: everything is done using type class inference. The application `wk u` in the `LetIn` case generates a type class goal which is particularly interesting: `scope_incl (s ▷ x) (s ▷ v ▷ x)`. This is where scopes really shine compared to plain natural numbers: there are many thinnings (a.k.a. order-preserving renamings) between scopes of size `n + 1` and `n + 2`, but there is only one obvious thinning from `s ▷ x` to `s ▷ v ▷ x`.

4 Correctness of a stack reduction machine

We implement a simple stack-based reduction machine for the untyped λ-calculus, in the style of Krivine's machine [12]. The machine performs strong call-by-name reduction: when reducing a term t the initial state is $(t, [])$ and we use the following transitions:

$$(t\ u,\ us) \longrightarrow (t,\ u :: us)$$
$$(\lambda x.\, t,\ u :: us) \longrightarrow (t[x := u],\ us)$$
$$(\lambda x.\, t,\ []) \longrightarrow (\lambda x.\, u\, us,\ []) \ \textit{if}\ (t,\ []) \longrightarrow (u,\ us)$$

We prove the machine correct with respect to an inductively defined reduction judgement: if the machine reduces t into u, then indeed t reduces to u according to the reduction judgement. We compare three different binder representations on this task: (unscoped) de Bruijn indices, locally nameless, and phantom names. Implementing the reduction machine is equally easy with every binder representation. However, with de Bruijn indices and phantom names the proof is completely trivial but for locally nameless the proof is difficult and tedious. Cofinite quantification [4] is insufficient on this simple example and we have to resort to a different approach combining universal and existential quantification: see our supplementary material for more details on where cofinite quantification gets stuck and for our successful proof. Perhaps most telling is the number of lines dedicated to proofs in each version: de Bruijn indices and phantom names each require 18 lines of proofs, whereas locally nameless requires 192 lines of proofs!

5 Mechanized meta-theory

Inspired by the MetaRocq project [18], we use phantom names to mechanize the meta-theory of a small dependently typed calculus. Specifically, we choose to formalize a variant of the calculus of constructions [10] which additionally contains existential variables (CC_{ev} in the following):

```
Inductive term (s : scope) :=
| Type_
| Var (i : index s)
| Lam (x : tag) (ty : term s) (body : term (s ▷ x))
| Prod (x : tag) (a : term s) (b : term (s ▷ x))
| App (t u : term s)
| Evar (e : evar_id).
```

There is a single sort `Type_` (CC_{ev} has `Type_` in `Type_`). `Var` represents variables using de Bruijn indices, `Lam` is a lambda abstraction, `Prod` is a dependent product (i.e. a Π-type), `App` is an application, and `Evar` represents an existential variable using a unique identifier (of type `evar_id`).

We mechanize the theory of parallel substitution and σ-calculus [1] and prove many lemmas about CC_{ev}, notably the Church-Rosser lemma (i.e. confluence of reduction), injectivity of Π-types, and subject reduction. Based on the size of our formalization (several thousand lines) and because our proofs using phantom names are extremely similar to those based on pure de Bruijn indices (e.g. in MetaRocq), we are confident that phantom names scale to large formalizations.

6 Related and future work

Our work builds upon the paper *Names for free* by Bernardy and Pouillard [7], in which the authors use Haskell as their language of choice. One of our key insights is that dependent types allow for a streamlined presentation: *Names for free* requires packing/unpacking functions, a clear distinction between universal and existential representations of binders, and intricate type-level encodings of data such as tags and scopes, but we need none of this machinery.

Concerning future work, we plan to use phantom names to implement a metaprogramming framework for Rocq where users can verify their metaprograms. Beyond proofs and programming, a third consideration is *efficiency*: early experiments with extraction to OCaml suggest that using phantom names does not sacrifice any performance compared to de Bruijn indices; scopes and tags can be completely erased at runtime, and well-scoped de Bruijn indices (`index` in Section 2) can be represented by native integers.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. 1991. Explicit substitutions. *Journal of Functional Programming* 1, 4 (1991), 375–416. https://doi.org/10.1017/S0956796800000186

[2] Beniamino Accattoli, Horace Blanc, and Claudio Sacerdoti Coen. 2023. Formalizing Functions as Processes. In *14th International Conference on Interactive Theorem Proving (ITP 2023) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 268)*, Adam Naumowicz and René Thiemann (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 5:1–5:21. https://doi.org/10.4230/LIPIcs.ITP.2023.5

[3] Thorsten Altenkirch and Bernhard Reus. 1999. Monadic Presentations of Lambda Terms Using Generalized Inductive Types. In *Computer Science Logic*, Jörg Flum and Mario Rodriguez-Artalejo (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 453–468.

[4] Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. 2008. Engineering formal metatheory. *SIGPLAN Not.* 43, 1 (Jan. 2008), 3–15. https://doi.org/10.1145/1328897.1328443

[5] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. 2005. Mechanized metatheory for the masses: the PoplMark challenge. In *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics* (Oxford, UK) *(TPHOLs'05)*. Springer-Verlag, Berlin, Heidelberg, 50–65. https://doi.org/10.1007/11541868_4

[6] Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. 2010. Parametricity and dependent types. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming* (Baltimore, Maryland, USA) *(ICFP '10)*. Association for Computing Machinery, New York, NY, USA, 345–356. https://doi.org/10.1145/1863543.1863592

[7] Jean-Philippe Bernardy and Nicolas Pouillard. 2013. Names for free: polymorphic views of names and binders. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell* (Boston, Massachusetts, USA) *(Haskell '13)*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/2503778.2503780

[8] Richard S. Bird and Ross Paterson. 1999. de Bruijn notation as a nested datatype. *J. Funct. Program.* 9, 1 (Jan. 1999), 77–91. https://doi.org/10.1017/S0956796899003366

[9] Arthur Charguéraud. 2012. The Locally Nameless Representation. *Journal of Automated Reasoning - JAR* 49 (10 2012), 1–46. https://doi.org/10.1007/s10817-011-9225-2

[10] Thierry Coquand and Gérard Huet. 1988. The calculus of constructions. *Information and Computation* 76, 2 (1988), 95–120. https://doi.org/10.1016/0890-5401(88)90005-3

[11] N.G de Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)* 75, 5 (1972), 381–392. https://doi.org/10.1016/1385-7258(72)90034-0

[12] Jean-Louis Krivine. 2007. A call-by-name lambda-calculus machine. *Higher Order Symbol. Comput.* 20, 3 (Sept. 2007), 199–207. https://doi.org/10.1007/s10990-007-9018-9

[13] Conor McBride and James McKinna. 2004. Functional pearl: i am not a number–i am a free variable. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell* (Snowbird, Utah, USA) *(Haskell '04)*. Association for Computing Machinery, New York, NY, USA, 1–9. https://doi.org/10.1145/1017472.1017477

[14] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 625–635. https://doi.org/10.1007/978-3-030-79876-5_37

[15] Ulf Norell. 2007. Towards a practical programming language based on dependent type theory. https://api.semanticscholar.org/CorpusID:118357515

[16] Gordon D. Plotkin. 1975. Call-by-Name, Call-by-Value and the lambda-Calculus. *Theor. Comput. Sci.* 1 (1975), 125–159. https://api.semanticscholar.org/CorpusID:205090276

[17] Andreas Rossberg, Claudio V. Russo, and Derek Dreyer. 2010. F-ing modules. In *Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation* (Madrid, Spain) *(TLDI '10)*. Association for Computing Machinery, New York, NY, USA, 89–102. https://doi.org/10.1145/1708016.1708028

[18] Matthieu Sozeau, Yannick Forster, Meven Lennon-Bertrand, Jakob Nielsen, Nicolas Tabareau, and Théo Winterhalter. 2025. Correct and Complete Type Checking and Certified Erasure for Coq, in Coq. *J. ACM* 72, 1, Article 8 (Jan. 2025), 74 pages. https://doi.org/10.1145/3706056

[19] The Coq Development Team. 2024. *The Coq Proof Assistant*. https://doi.org/10.5281/zenodo.14542673