# Syntax with Shifted Names

Stephen Dolan
OCaml Labs, University of Cambridge

Leo White
Jane Street Europe

## Abstract

We propose *shifted names*, a new representation for names and binding. Like representations with explicit names, we can use distinct names for different things and not worry about reordering them, but like de Bruijn indices we have no need for freshness side-conditions.

## 1 Introduction

Representing and reasoning about syntax containing variable bindings is a longstanding source of frustration. When writing code that operates on, say, terms of lambda calculus, it is far too easy to write an incorrect substitution function that accidentally captures variables. If you do manage to write a correct one, it is far too hard to convince a proof assistant of the fact.

The literature contains a wealth of approaches to this problem, including direct representations of names as strings or de Bruijn indices, first-order representations that distinguish free and bound variables (the *locally named* representation of McKinna and Pollack [4], or the *locally nameless* representation (see e.g. Charguéraud [2]), higher-order representations that re-use metalanguage binding [5], and approaches based on nominal sets [6, 7].

Our interest in this problem stems from work on type systems for algebraic effects. These systems allow detailed static tracking of the type of effects that code may perform, as well as the type of its result. For instance, an expression $e$ that returns an integer (after asking the environment for the name of a configuration file and reading it) might be given the following type:

$$\vdash e : \texttt{Int}[\texttt{Reader Filename}, \texttt{FileIO}]$$

A central advantage of algebraic effects is that the row of effects above is *unordered*: the row is grown by performing other effectful operations and shrunk by *handling* them, but it is not necessary to handle effects in exactly the order that the type lists them.

But what happens if the row contains two effects with the same name? This can happen because of a recursive function, because the programmer used the same name for two effects, or for myriad other reasons. Suddenly, the problems of disjointness, freshness and capture-avoiding substitution reappear.

Our original idea was that we would borrow some machinery from the literature on binding representations to help deal with algebraic effects. However, what's happened so far is the opposite: an idea borrowed from algebraic effects turns out to help deal with binding. This talk is about that idea, *shifted names*, and how it helps clarify and simplify the locally nameless representation.

*TyDE'19, August 18, 2019, Berlin, Germany*
2019.

### 1.1 Shifted Names

Leijen's language Koka [3] introduced a novel way of dealing with multiple copies of an effect appearing simultaneously, which was developed further by Biernacki et al. [1]. In a row of effects such as:

$$[\texttt{Reader Filename}, \texttt{FileIO}, \texttt{Reader Filename}]$$

the effects with distinct names can be freely reordered, but the order matters for effects with the same name. References to `Reader Filename` refer to the rightmost occurrence, but shadowed occurrences can be referenced by repeatedly applying an operation known variously as *inject*, *lift* or *shift*. In essence, de Bruijn indices are used to disambiguate between effects with the same name.

Here, we adapt this idea for reasoning about syntax with binding: our *names* $x, y, z$ are of the form $\texttt{foo}_3$, consisting of a pair of a *label* (here `foo`) and an *index* (here 3). We use the indices as de Bruijn indices, to resolve collisions between names with the same label.

## 2 Locally Nameless and Shifted Names

The locally nameless representation draws a distinction between free variables (represented as names) and bound variables (represented as numbers). Terms of the untyped lambda calculus are represented as follows, with natural numbers $n$ representing bound variables:

$$t, u ::= x \mid n \mid t\ u \mid \lambda\, t$$

Note that $\lambda\, t$ does not specify the name of the bound variable: bound variables are numbered in sequence, de Bruijn-style. Compared to the *locally named* representation [4] (using names for both free and bound variables), this choice makes $\alpha$-equivalence trivial.

There are two fundamental operations: $\mathsf{open}_x$ turns bound variables referring to the outermost binder into the free variable $x$, while $\mathsf{close}_x$ turns the free variable $x$ into a reference to the outermost binder.

We think of these operations as moving a cursor around a term: $\mathsf{open}_x$ moves the cursor underneath $\lambda$, giving the name $x$ to the bound variable, while $\mathsf{close}_x$ moves the cursor back out, binding occurrences of $x$.

Our versions of $\mathsf{open}_x$ and $\mathsf{close}_x$ have a more subtle definition, adjusting indices to avoid causing name

collisions. For instance:

$$\mathsf{open}_{\mathtt{foo}_2}\ (\mathtt{foo}_2\ 0) = \mathtt{foo}_3\ \mathtt{foo}_2$$
$$\mathsf{open}_{\mathtt{foo}_2}\ (\mathtt{foo}_3\ 1) = \mathtt{foo}_4\ 0$$

The index in the free variable $\mathtt{foo}_2$ was incremented, to avoid confusing it with the newly-opened $\mathtt{foo}_2$. This adjustment is undone by $\mathsf{close}_x$, giving us the following useful properties:

$$\mathsf{open}_x \circ \mathsf{close}_x = \mathsf{id}$$
$$\mathsf{close}_x \circ \mathsf{open}_x = \mathsf{id}$$

These equations also appeared in Charguéraud's presentation of the locally nameless representation [2], but required technical side-conditions of freshness and local closure. Our versions of $\mathsf{open}_x$ and $\mathsf{close}_x$ do not cause name collisions, and so do not need these side-conditions.

### 2.1 Inserting and removing binders

As well as $\mathsf{open}_x$ and $\mathsf{close}_x$ to move the cursor into and out of a binder, we have two operations $\mathsf{wk}$ and $\mathsf{bind}_u$ to add and remove bound variables.

The operation $\mathsf{wk}$ adds an unused bound variable, so that $\lambda\ (\mathsf{wk}\, t)$ is a constant function returning $t$. Dually, $\mathsf{bind}_u$ removes a bound variable by replacing its occurrences with $u$. For instance, the $\beta$-reduction rule can then be written:

$$(\lambda\ t)\ u \longrightarrow \mathsf{bind}_u\ t$$

We have an equation relating these operations:

$$\mathsf{bind}_u \circ \mathsf{wk} = \mathsf{id}$$

In the standard presentation of the locally nameless representation, these operations are not explicit. Usually, $\mathsf{wk}$ is entirely implicit, and $\mathsf{bind}$ is conflated with $\mathsf{open}$. With shifted names, they are distinct: $\mathsf{bind}_x\, t$ and $\mathsf{open}_x\, t$ are not in general equal, as $\mathsf{open}_x\, t$ adjusts indices to ensure free variables of $t$ do not collide with $x$, while $\mathsf{bind}_x$ does not. Indeed, this conflation of $\mathsf{bind}$ and $\mathsf{open}$ is one of the main reasons that freshness side-conditions are needed in the standard presentation.

## 3 Renaming, substitution and shifting

The operations $\mathsf{wk}$ and $\mathsf{close}_x$ introduce a bound variable, while $\mathsf{bind}_u$ and $\mathsf{open}_x$ remove one. This gives us three derived operations which introduce and then remove a bound variable (the fourth possibility, $\mathsf{bind}_u \circ \mathsf{wk}$, is the identity):

$$\langle y/x\rangle = \mathsf{open}_y \circ \mathsf{close}_x \qquad \text{(renaming)}$$
$$[u/x] = \mathsf{bind}_u \circ \mathsf{close}_x \qquad \text{(substitution)}$$
$$\mathsf{S}_x = \mathsf{open}_x \circ \mathsf{wk} \qquad \text{(shifting)}$$

Several properties of these operations are immediate consequences of the three equations of section 2:

$$\langle x/x\rangle = \mathsf{id}$$
$$\langle x/y\rangle \circ \langle y/z\rangle = \langle x/z\rangle$$
$$\langle x/y\rangle \circ \langle y/x\rangle = \mathsf{id}$$
$$[u/x] \circ \langle x/y\rangle = [u/y]$$
$$[u/x] \circ \mathsf{S}_x = \mathsf{id}$$
$$\mathsf{S}_x \circ \langle x/y\rangle = \mathsf{S}_y$$

Again, note the lack of freshness conditions here.

The $\mathsf{S}_x$ operation is particularly useful for eliding freshness conditions. In our notation, the statement of Barendregt's substitution lemma is, for variables $x, y$ with distinct labels:

$$[t/x] \circ [u/y] = [[t/x]u\ /\ y] \circ [\mathsf{S}_y\, t/x]$$

Note that we do not need Barendregt's side-condition that $y$ does not appear free in $t$. Instead, we can use $\mathsf{S}_y\, t$, relying on the property that $[u/y]\mathsf{S}_y\, t = t$.

### 3.1 Parallel operations

The operations of renaming, substitution and shifting above operate on only a single variable at a time. By composing longer sequences of $\mathsf{open}_x$, $\mathsf{close}_x$, $\mathsf{wk}$ and $\mathsf{bind}_u$, we can operate on multiple variables in parallel. We define a *weakening* $\rho$ as a mixture of renaming and shifting, and a *substitution* $\eta$ as a mixture of renaming and substitution:

$$\rho = \mathsf{id} \mid \mathsf{open}_x \circ \rho \circ \mathsf{close}_y \mid \mathsf{open}_x \circ \rho \circ \mathsf{wk}$$
$$\eta = \mathsf{id} \mid \mathsf{open}_x \circ \eta \circ \mathsf{close}_y \mid \mathsf{bind}_u \circ \eta \circ \mathsf{close}_x$$

As consequences of the equations above, we have several useful properties of weakenings and substitutions. For instance, for every weakening $\rho$ there is a substitution $\rho^*$ such that $\rho^* \circ \rho = \mathsf{id}$, and for every substitution $\eta$ there is a weakening $\eta^*$ such that $\eta \circ \eta^* = \mathsf{id}$.

## 4 Discussion

Shifted names make it possible to represent and reason about syntax with binding without needing to reason about arithmetic (as with de Bruijn representations) or have freshness side-conditions (as with named representations). We try to use distinct names for distinct things, but if collisions do arise they are resolved with numbering.

***Acknowledgements*** Thanks to James McKinna for enlightening discussion.

## References

[1] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2018. Handle with Care: Relational Interpretation

of Algebraic Effects and Handlers. *POPL '18* 2, POPL, Article 8 (Dec. 2018), 30 pages. https://doi.org/10.1145/3158096

[2] Arthur Charguéraud. 2012. The locally nameless representation. *Journal of automated reasoning* 49, 3 (2012), 363–408.

[3] Daan Leijen. 2014. Koka: Programming with row polymorphic effect types. *arXiv preprint arXiv:1406.2061* (2014).

[4] James McKinna and Robert Pollack. 1993. Pure type systems formalized. In *International Conference on Typed Lambda Calculi and Applications*. Springer, 289–305.

[5] Frank Pfenning and Conal Elliott. 1988. Higher-order Abstract Syntax. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (PLDI '88)*. ACM, 199–208. https://doi.org/10.1145/53990.54010

[6] Andrew M. Pitts. 2003. Nominal logic, a first order theory of names and binding. *Information and Computation* 186, 2 (2003), 165 – 193. https://doi.org/10.1016/S0890-5401(03)00138-X Theoretical Aspects of Computer Software (TACS 2001).

[7] Christian Urban. 2008. Nominal techniques in Isabelle/HOL. *Journal of Automated Reasoning* 40, 4 (2008), 327–356.