Thinning Thinnings: Safe and Efficient Binders

April Gonçalves[1] and Wen Kokke[2]

[1] University of Strathclyde, Glasgow, UK
[2] Well-Typed

Abstract

Binding is notoriously difficult to implement both correctly and efficiently. When using names, one must be ever vigilant for name capture. When using de Bruijn indices and thinnings, one must correctly adjust them whenever their scope changes. These invariants have tripped up even the most seasoned compiler writers. Using dependent types, it is straightforward to ensure the correct use of de Bruijn indices and thinnings. Unfortunately, such implementations are often inefficient. Moreover, this requires that the compiler is written in a dependently-typed language.

We present a Haskell library that provides an implementation de Bruijn indices and thinnings that is both correct and efficient. The library exports the usual inductive definitions for de Bruijn indices and thinnings, indexed on the type-level with the size of their scope and scopes, respectively. However, it implements these efficiently as machine words and bit vectors. We test the correctness of the efficient implementation using QuickCheck. We intend to benchmark the library against the usual inductive definitions, to ensure that it is more efficient, and against the unsafe efficient implementation, to ensure that the type-level indices do not incur any runtime overhead.

Background. Logicians, type theorists, and compiler writers have struggled with syntax and binding for over a century [3]. Compiler writers, especially, struggle with representing binding. *Named* representations of binding use matching names—be they strings or numbers—for binding sites and bound variables. Using names, one must work to avoid name capture and ensure α-equivalence. *Nameless* representations of binding got their party started with de Bruijn and his indices [1] which represent bound variables as numbers that index into the list of enclosing scopes, writing, e.g., $\lambda.(\lambda.1)$ for $\lambda x.(\lambda y.x)$. For the remainder of this abstract, we leave our stringly-named friends behind and focus on indices.

De Bruijn Indices and Thinnings, Inductively. A de Bruijn index selects an element from a list—be it a typing context or an evaluation environment, and we *can* represent them in Haskell as we will show… Everybody brace for length-indexed vectors! *Nat* is the kind of type-level natural numbers:

```
type data Nat = Z | S Nat   -- We write 0 for Z, 1 for S 0, 2 for S 1, 3 for S 2, etc.
```

Env n is the type of lists of length n and *Ix n* is the type of indices into such lists:

```
data Env n a where                    data Ix n where
  Nil ::                Env Z     a     FZ ::          Ix (S n)
  Cons :: a → Env n a → Env (S n) a     FS :: Ix n → Ix (S n)
```

Where an index selects a single element from a list, a *thinning* selects a sublist. *Th m n* is the type of thinnings that thins a list of length m out to a list of length n by either keeping or dropping each element:

```
data Th m n where
  Done ::                Th Z       Z
  Keep :: Th m n → Th (S m) (S n)
  Drop :: Th m n → Th (S m)    n
```

```
ex1 :: Th 4 2   -- [A, B, C, D] to [C, D]
ex1 = Drop ∘ Drop ∘ Keep ∘ Keep $ Done
ex2 :: Th 4 2   -- [A, B, C, D] to [A, C]
ex2 = Keep ∘ Drop ∘ Keep ∘ Drop $ Done
```

Both *ex1* and *ex2* are examples of thinnings that select a sublist of length 2 from a list of length 4. Both have type *Th* 4 2. However, they are distinct: *ex1* drops elements 1 and 2 and *ex2* drops elements 2 and 4. The type *Th* 4 2 is inhabited by all combinations of selecting 2 elements from a list 4. This is such a general concept that thinnings, for compiler writers, pop up all over. In evaluation, a thinning can represent a batch of adjustments to the bound variable indices [5]. In dependent type checking, a thinning can represent the portion of its context to which a meta-variable has access. Using co-de Bruijn representation [5], thinnings can even be used to represent binding, obviating the need for indices.

De Bruijn Indices and Thinnings, Efficiently. The inductive definitions *Ix* and *Th* are correct but inefficient. The index 3, represented as *FS* (*FS FZ*), takes up *five* machine words. I hope we can agree that *one* machine word should suffice. If you *really* need more than 2^{64} nested binders[1], you can use Haskell's efficient arbitrary-precision *Integer* type[2]. Likewise, *ex1* and *ex2* each take up *nine* machine words, where 4 bits would suffice to represent each *Keep* and *Drop*. Such optimisations are often found in the wild. Both smalltt [4] and ask [6] represent indices as *Int* and thinnings as bit vectors—smalltt uses *Int*, which is efficient but limits the number of nested binders to 64, and ask uses *Integer* as an infinite bit vector. Unfortunately, these optimisations lose the type-level safety guarantees of the inductive definitions. Anecdotally, the most error-prone parts of both ask and Idris 1 [2] was in their implementations of binders.

Contribution. Our contribution is a library that provides an efficient implementation of de Bruijn indices and thinnings, representing indices as words and thinnings as bit vectors, but whose API matches the inductive definitions, including type-level safety guaratees. Our library leverages bidirectional pattern synonyms, view patterns, and—most importantly—*lies* to mimic the inductive data type definitions down to their constructors. Our technique leverages unsafe projection and embedding functions between the efficient representation and the inductive base functor, generating the inductive constructors on a by-need basis, e.g., indices are defined as:

```
data IxF f n where
  FZF ::        IxF f (S n)
  FSF :: f n → IxF f (S n)
```

```
newtype Ix (n :: Nat) = Ix Word16
unsafeProject :: Ix n          → IxF Ix n
unsafeEmbed   :: IxF Ix (S n) → Ix (S n)
```

```
pattern FZ   ← (unsafeProject → FZF)   where FZ   = unsafeEmbed FZF
pattern FS i ← (unsafeProject → FSF i) where FS i = unsafeEmbed (FSF i)
```

We test the equivalence of the inductive and the efficient definitions using a QuickCheck suite. In the future, we hope to benchmark our library against the inductive definitions. While the data type definitions make it sufficiently clear that our library is more space efficient, such a benchmark would test whether or not our library is more time efficient. Furthermore, we hope to benchmark our library against the efficient definitions *without* type-level safety guaratees, which would test whether or not our type-level indices introduce any runtime overhead.

[1] For simplicity, we assume a 64-bit architecture.

[2] But you should, probably, re-evaluate the decisions that led to you needing 18446744073709551615 variables.

References

[1] N. G de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, January 1972.

[2] The Idris Developers. Idris 1. https://github.com/idris-lang/Idris-dev, 2025.

[3] Gottlob Frege. Uber sinn und bedeutung. *Zeitschrift für Philosophie Und Philosophische Kritik*, 100(1):25–50, 1892.

[4] András Kovács. smalltt. https://github.com/AndrasKovacs/smalltt, 2023.

[5] Conor McBride. Everybody's Got To Be Somewhere. *Electronic Proceedings in Theoretical Computer Science*, 275:53–69, July 2018.

[6] Conor McBride, Guillaume Allais, Fredrik Nordvall Forsberg, and Jules Hedges. ask. https://github.com/msp-strath/ask, 2025.