

Free Foil: Generating Efficient and Scope-Safe Abstract Syntax

[Nikolai Kudasov](#), Renata Shakirova, Egor Shalagin, Karina Tyulebaeva

The 4th International Conference on Code Quality — ICCQ 2024

Innopolis, Russia, June 22, 2024

Lab of Programming Languages and Compilers



Capture-Avoiding Substitution

In presence of bound (local) variables, substitution is not trivial. In λ -calculus:

$$(\lambda x. \lambda y. x)y =_{\beta} [x \mapsto y](\lambda y. x) \neq \lambda y. y$$

This is not just a theoretic issue! Capture-avoiding substitution is used to implement

1. type elaboration (especially with dependent types, e.g. in Agda)
2. inlining and other hygienic transformations in compilers, e.g. in GHC (Peyton Jones and Marlow 2002)
3. hygienic macro expansion, e.g. in Lean (Ullrich and Moura 2020)
4. SMT solvers, theorem provers, and other formal systems

Approaches to Abstract Syntax with Binders

Many approaches exist to prevent name captures, varying in type safety, efficiency, and ability to produce generic algorithms (starting with substitution).

	Safe	Efficient	General
Naïve	NO	NO	NO
de Bruijn 1972	NO	PARTIAL	NO
Bird and Paterson 1999	YES	NO	NO
“Rapier” by Peyton Jones and Marlow 2002	NO	YES	NO
PHOAS by Chlipala 2008	YES	YES	NO
“Foil” by Maclaurin, Radul, and Paszke 2023	YES	YES	NO
“Free Scoped Monads” by Kudasov 2024	YES	NO	YES
“Free Foil” (our work)	YES	YES*	YES

The Rapier and the Foil

The “*Rapier*” (Peyton Jones and Marlow 2002, used in GHC and Agda):

- implements the Barendregt convention (Barendregt 1985)
- does not rely on global state (amenable to parallelization)
- relies on immutable maps (Okasaki and Gill 1998) for efficient scopes and substitutions

The “*Foil*” (Maclaurin, Radul, and Paszke 2023, used in Dex):

- same underlying representation as the “*Rapier*”
- adds phantom types parameters to track scopes statically
- offers safe zero-cost “sinking” (corresponds to “shifting” for de Bruijn indices)
- relies on existential types and rank-2 polymorphism for scope-safety

The Foil: Scope-Safe Types

The “Foil” introduces the following scope-safe types and classes:

1. `Name n` — an identifier in scope `n` (represented as a type variable)
2. `NameBinder n l` — an identifier that extends scope `n` to scope `l`
3. `Scope n` — a set of identifiers in scope `n`
4. `DExt n l` — a constraint ensuring that `l` contains distinct names and extends `n`

To work safely with binders, the “Foil” relies on rank-2 polymorphism:

```
withRefreshed
```

```
  :: Scope o
  -> Name i
  -> (forall o'. DExt o o' => NameBinder o o' -> r)
  -> r
```

The Foil: Sinkability Proof (User Code)

The user can use the “Foil” to construct scope-safe abstract syntax:

```
data Expr n where
  VarE  :: Name n -> Expr n           -- variable: x
  AppE  :: Expr n -> Expr n -> Expr n -- application: (e1 e2)
  LamE  :: NameBinder n l -> Expr l -> Expr n -- abstraction: λx.e
```

The following instance proves that we can “sink” expressions safely:

```
instance Sinkable Expr where
  -- sinkabilityProof :: (Name n -> Name l) -> Expr n -> Expr l
  sinkabilityProof rename (VarE v) = VarE (rename v)
  sinkabilityProof rename (AppE f e) =
    AppE (sinkabilityProof rename f) (sinkabilityProof rename e)
  sinkabilityProof rename (LamE binder body) =
    extendRenaming rename binder $ \rename' binder' ->
      LamE binder' (sinkabilityProof rename' body)
```

The Foil: Substitution (User Code)

The scope-safe types ensure that we do not mix up the scopes and do not forget to extend scopes and substitutions:

```
substitute :: Distinct o => Scope o -> Substitution Expr i o -> Expr i -> Expr o
substitute scope subst = \case
  VarE name -> lookupSubst subst name
  AppE f x -> AppE (substitute scope subst f) (substitute scope subst x)
  LamE binder body -> withRefreshed scope binder $ \extendSubst binder' ->
    let subst' = extendSubst subst
        scope' = extendScope binder' scope
        body' = substitute scope' subst' body
    in LamE binder' body'
```

But the user still has to write all of this (and more) before doing anything useful!

Limitations of the Foil

The “Foil” is safe, but does not provide any of the following for the object language:

1. capture-avoiding substitution
2. α -equivalence check (which is non-trivial to implement efficiently!)
3. scope-safe data type(s) for abstract syntax of the object language
4. conversion functions to/from raw representation (e.g. to connect to a parser)
5. derivation of sinkability proof(s) for the object language

Also, the original “Foil” does not offer general support for patterns (complex binders), but patterns can be relatively easily implemented in the user code.

A “Foil” implementation of $\lambda\Pi$ -interpreter take 251 lines of Haskell code.

Distinct Types for Scope Annotations

Raw representation is usually bound to concrete syntax and can be generated by parser generators like BNF Converter¹ from a grammar file. Our first idea was to extend grammar with some scope annotations, to generate the scope-safe syntax. However, as it turns out it is enough to use 4 distinct types (or non-terminals in the grammar):

```
data Expr
  = Var Ident
  | App Expr Expr
  | Lam Pattern ScopedExpr
```

```
data Pattern = PatternVar Ident
data ScopedExpr = ScopedExpr Expr
```

¹<https://bnfc.digitalgrammars.com>

The Foil + Template Haskell

We use Template Haskell (Sheard and Jones 2002) to generate

1. scope-safe abstract syntax
2. conversion functions to/from raw representation
3. sinkability proof(s)

This approach works well with BNFC-generated types,
(with and without the `--functor` option for generic location annotations).

It is technically possible to generate substitution and α -equivalence, but Template Haskell is a fragile instrument, and we prefer to avoid relying too much on it.

λ II-interpreter with TH takes 98 lines of Haskell code.

Merging the Foil with Free Scoped Monads

For generic algorithms, we merge the “Foil” with free scoped monads (Kudasov 2024):

```
data ScopedAST sig n where
  ScopedAST :: NameBinder n l -> AST sig l -> ScopedAST sig n
```

```
data AST sig n where
  Var  :: Name n -> AST sig n
  Node :: sig (ScopedAST sig n) (AST sig n) -> AST sig n
```

Here, `AST sig n` is the type of scope-safe terms freely generated from a signature `sig`. For example, this signature corresponds for λ -calculus:

```
data ExprSig scope term = App term term | Lam scope
```

Recursive scope-safe AST for λ -calculus is generated with `AST`:

```
type Expr n = AST ExprSig n
```

The Free Foil

The “Free Foil” enables various generic algorithms, which are written once and then can be reused for **any** object language:

1. capture-avoiding substitution
2. α -equivalence checks
3. abstract syntax à la carte
4. (higher-order) unification (not implemented yet)

λ II-interpreter with “Free Foil” takes 181 lines of Haskell code.

The Free Foil + Template Haskell (experimental)

We can also use Template Haskell with the Free Foil, getting almost all the boilerplate and generic algorithms for free:

```
-- ** Signature
mkSignature 'Raw.Term' 'Raw.VarIdent' 'Raw.ScopedTerm' 'Raw.Pattern'
deriveZipMatch 'Term'Sig
deriveBifunctor 'Term'Sig
deriveBifoldable 'Term'Sig
deriveBitraversable 'Term'Sig

-- ** Pattern synonyms
mkPatternSynonyms 'Term'Sig

-- ** Conversion helpers
mkConvertToFreeFoil 'Raw.Term' 'Raw.VarIdent' 'Raw.ScopedTerm' 'Raw.Pattern'
mkConvertFromFreeFoil 'Raw.Term' 'Raw.VarIdent' 'Raw.ScopedTerm' 'Raw.Pattern'
```

λ II-interpreter with “Free Foil” and TH takes 77 lines of Haskell code.

Benchmark Results

We use Stephanie Weirich's benchmark suite for λ -calculus implementations to compare performance of "Free Foil" against other implementations.

Our findings are as follows:

1. "Foil" > "Free Foil" > "free scoped monads"
2. strict and lazy implementations win in different benchmarks
3. the fastest implementations rely on *delayed substitutions*

See our fork of the benchmark suite on GitHub: [KarinaTyulebaeva/lambda-n-ways](https://github.com/KarinaTyulebaeva/lambda-n-ways).

Results

In the paper:

1. `CoSinkable` patterns for the Foil
2. Template Haskell generators for the Foil
3. The Free Foil representation
4. Generic substitution for the Free Foil
5. Performance benchmarks (see [KarinaTyulebaeva/lambda-n-ways](#))
6. $\lambda\Pi$ -calculus interpreter implementations for comparison of approaches

Additionally (on GitHub, see [fizruk/free-foil](#)):




1. Template Haskell generators for the Free Foil signature and conversion functions
2. Generic α -equivalence for the Free Foil
3. Generic conversion helpers for the Free Foil



All Template Haskell generators work with BNFC-generated abstract syntax!




Future work



1. *Generalized binders* (patterns) in the free foil would enable more languages.
2. *General closure representation* (delayed substitutions) should enable normalization by evaluation and other efficient substitution-heavy algorithms.
3. *General (higher-order) unification algorithms* may be implemented now on top of the free foil, adapting a previous approach (Kudasov 2024).
4. *Strict(er) free foil* representation may offer better performance.
5. *Church-encoded Free Foil* representation à la (Voigtländer 2008) should offer better performance for substitution.
6. *Singleton scope types* (Eisenberg and Weirich 2012) may simplify the interface, removing the explicit scope parameter from many functions.
7. *Zero-overhead generic algorithms* probably could be derived safely with “generic programming for all kinds” (Serrano and Miraldo 2018)



Thank you!

-  Barendregt, Hendrik Pieter (1985). **The lambda calculus - its syntax and semantics**. Vol. 103. Studies in logic and the foundations of mathematics. North-Holland. ISBN: 978-0-444-86748-3.
-  Bird, Richard S. and Ross Paterson (1999). “**de Bruijn notation as a nested datatype**”. In: *Journal of Functional Programming* 9.1, pp. 77–91. DOI: [10.1017/S0956796899003366](https://doi.org/10.1017/S0956796899003366).
-  Chlipala, Adam (Sept. 2008). “**Parametric Higher-Order Abstract Syntax for Mechanized Semantics**”. In: *SIGPLAN Not.* 43.9, pp. 143–156. ISSN: 0362-1340. DOI: [10.1145/1411203.1411226](https://doi.org/10.1145/1411203.1411226).

-  de Bruijn, N.G (1972). “**Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem**”. In: *Indagationes Mathematicae (Proceedings)* 75.5, pp. 381–392. ISSN: 1385-7258. DOI: [https://doi.org/10.1016/1385-7258\(72\)90034-0](https://doi.org/10.1016/1385-7258(72)90034-0). URL: <https://www.sciencedirect.com/science/article/pii/1385725872900340>.
-  Eisenberg, Richard A. and Stephanie Weirich (2012). “**Dependently typed programming with singletons**”. In: *Proceedings of the 2012 Haskell Symposium*. Haskell '12. Copenhagen, Denmark: Association for Computing Machinery, pp. 117–130. ISBN: 9781450315746. DOI: [10.1145/2364506.2364522](https://doi.org/10.1145/2364506.2364522). URL: <https://doi.org/10.1145/2364506.2364522>.

-  Kudasov, Nikolai (2024). **Free Monads, Intrinsic Scoping, and Higher-Order Preunification**. To appear in *TFP 2024*. Orange, NJ, USA. arXiv: [2204.05653](https://arxiv.org/abs/2204.05653) [cs.LG].
-  Maclaurin, Dougal, Alexey Radul, and Adam Paszke (2023). **“The Foil: Capture-Avoiding Substitution With No Sharp Edges”**. In: *Proceedings of the 34th Symposium on Implementation and Application of Functional Languages*. IFL '22. Copenhagen, Denmark: Association for Computing Machinery. ISBN: 9781450398312. DOI: [10.1145/3587216.3587224](https://doi.org/10.1145/3587216.3587224). URL: <https://doi.org/10.1145/3587216.3587224>.
-  Okasaki, Chris and Andrew Gill (1998). **“Fast mergeable integer maps”**. In: *Workshop on ML*, pp. 77–86.

-  Peyton Jones, Simon and Simon Marlow (July 2002). “**Secrets of the Glasgow Haskell Compiler inliner**”. In: *Journal of Functional Programming* 12, pp. 393–434. URL: <https://www.microsoft.com/en-us/research/publication/secrets-of-the-glasgow-haskell-compiler-inliner/>.
-  Serrano, Alejandro and Victor Cacciari Miraldo (2018). “**Generic programming of all kinds**”. In: *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell*. Haskell 2018. St. Louis, MO, USA: Association for Computing Machinery, pp. 41–54. ISBN: 9781450358354. DOI: [10.1145/3242744.3242745](https://doi.org/10.1145/3242744.3242745). URL: <https://doi.org/10.1145/3242744.3242745>.

-  Sheard, Tim and Simon Peyton Jones (2002). **“Template meta-programming for Haskell”**. In: *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell, Haskell 2002, Pittsburgh, Pennsylvania, USA, October 3, 2002*. Ed. by Manuel M. T. Chakravarty. ACM, pp. 1–16. DOI: [10.1145/581690.581691](https://doi.org/10.1145/581690.581691). URL: <https://doi.org/10.1145/581690.581691>.
-  Ullrich, Sebastian and Leonardo de Moura (2020). **“Beyond Notations: Hygienic Macro Expansion for Theorem Proving Languages”**. In: *Automated Reasoning*. Ed. by Nicolas Peltier and Viorica Sofronie-Stokkermans. Cham: Springer International Publishing, pp. 167–182. ISBN: 978-3-030-51054-1.

-  Voigtländer, Janis (2008). “**Asymptotic Improvement of Computations over Free Monads**”. In: *Mathematics of Program Construction*. Ed. by Philippe Audebaud and Christine Paulin-Mohring. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 388–403. ISBN: 978-3-540-70594-9. DOI: [10.1007/s10817-011-9225-2](https://doi.org/10.1007/s10817-011-9225-2).