Operations On Syntax Should Not Inspect The Scope

Jesper Cockx

TU Delft, Delft, Netherlands

When implementing or formalizing the syntax of a language with names and binders, one challenging task is establishing and preserving well-scopedness. This is especially true when implementing a dependent type checker, where types bind variables and terms with free variables are evaluated. Luckily, if we implement this type checker itself in a dependently typed language, we can work with *well-scoped syntax*, i.e. syntax that is statically known to be well-scoped by the type system. For example, here is a minimal definition of well-scoped syntax for the untyped lambda calculus in Agda:

```
data Var : (n : ℕ) → Set where
  zero : Var (suc n)
  suc  : Var n → Var (suc n)
```

```
data Term (n : ℕ) : Set where
  var : Var n → Term n
  lam : Term (suc n) → Term n
  app : Term n → Term n → Term n
```

Brady et al. [2003] have taught us that inductive families such as Var and Term need not store their indices: the number n can be safely erased during compilation. However, to produce efficient compiled code we should also ensure that operations on the syntax do not inspect the scope at run-time. In a language with support for runtime irrelevance [McBride, 2016, Atkey, 2018] such as Idris 2 or Agda, we can enforce this property statically. But this reveals a problem: to implement a function right : Var $n \rightarrow$ Var $(k + n)$ that weakens a variable by adding k unused variables to the scope, it must apply the suc constructor k times to its argument, so erasing k is impossible! This example shows that using $\mathbb{N}$ as the type of scopes does not work.

This leads us to the question: is it possible to design types Scope : Set and Var : Scope $\rightarrow$ Set such that all necessary operations on variables can be defined without inspecting the scope. To make this question more concrete, let me list some operations that I consider 'necessary':

1. **Decidable equality** of variables: $\_\overset{?}{=}\_ : (x\ y : \mathsf{Var}\ \alpha) \rightarrow \mathsf{Dec}\ (x \equiv y)$.
2. An **empty scope** $\circ$: Scope such that Var $\circ \simeq \bot$.
3. A **singleton scope** $\bullet$: Scope such that Var $\bullet \simeq \top$.
4. A **disjoint union** $\_\diamond\_$: Scope $\rightarrow$ Scope $\rightarrow$ Scope such that $\mathsf{Var}\ (\alpha \diamond \beta) \simeq \mathsf{Var}\ \alpha \uplus \mathsf{Var}\ \beta$.
5. A **weakening** coerce : $\alpha \subseteq \beta \rightarrow \mathsf{Var}\ \alpha \rightarrow \mathsf{Var}\ \beta$, where $\_\subseteq\_$: Scope $\rightarrow$ Scope $\rightarrow$ Set is a preorder on scopes.
6. For any $p : \alpha \subseteq \beta$, a **complement** p^C : Scope such that $p^C \subseteq \beta$ and $p^C \subseteq (\mathsf{trans}\ p\ q)^C$ for any $q : \beta \subseteq \gamma$.

Instead of using $\mathbb{N}$, let us represent scopes as *binary trees* where each leaf is either an empty scope $\circ$ or a singleton $\bullet$:

```
data Scope : Set where
  ○ ● : Scope
  _◇_ : Scope → Scope → Scope
```

Rather than define Var and _⊆_ directly, we can define both in terms of a *proof relevant separation algebra* [Rouvoet et al., 2020], a ternary relation on scopes that determines how the names in the third scope are distributed over the first two.

```
data _⋈_≡_ : (α β γ : Scope) → Set where
  ∘-l   : ∘ ⋈ β ≡ β
  ∘-r   : α ⋈ ∘ ≡ α
  join  : α ⋈ β ≡ (α ⋄ β)
  swap  : α ⋈ β ≡ (β ⋄ α)
  ⋄-ll  : (α₂ ⋈ β  ≡ δ) → (α₁ ⋈ δ  ≡ γ) → (α₁ ⋄ α₂) ⋈ β        ≡ γ
  ⋄-lr  : (α₁ ⋈ β  ≡ δ) → (δ  ⋈ α₂ ≡ γ) → (α₁ ⋄ α₂) ⋈ β        ≡ γ
  ⋄-rl  : (α  ⋈ β₂ ≡ δ) → (β₁ ⋈ δ  ≡ γ) → α         ⋈ (β₁ ⋄ β₂) ≡ γ
  ⋄-rr  : (α  ⋈ β₁ ≡ δ) → (δ  ⋈ β₂ ≡ γ) → α         ⋈ (β₁ ⋄ β₂) ≡ γ
```

Subscoping and variables can then be defined in terms of separation:

```
α ⊆ β = Σ (Erased Scope) (λ ([ γ ]) → α ⋈ γ ≡ β)
Var α  = • ⊆ α
```

Here, Erased A is a record type with constructor [_] : @0 A → Erased A. This definition of _⊆_ makes it trivial to define the complement operation $\_^{C}$, since it is just the first projection of the subscope proof.

An implementation of the operations listed above can be found at `https://github.com/jespercockx/scopes-n-roses`. Compared to the code here, it follows Pouillard [2012] by providing an abstract interface for working with scopes and support for *named* variables.

There are at least two still unresolved problems with this scope representation. The first one is that separation proofs are not unique. In particular, we can map any proof of $(\alpha_1 \diamond \alpha_2) \bowtie \beta \equiv \gamma$ to another distinct proof of the same type:

```
enlarge : (α₁ ⋄ α₂) ⋈ β ≡ γ → (α₁ ⋄ α₂) ⋈ β ≡ γ
enlarge p = ⋄-ll join (⋄-rr join p)
```

As a result, the functions Var • → ⊤ and Var $(\alpha \diamond \beta)$ → Var α ⊎ Var β are only retractions rather than equivalences.

The second problem is that introduction of scope separation makes additional operations hard or impossible to implement, such as the following property that we would like to have in addition to the six above:

7. For two separations $p : \alpha_1 \bowtie \alpha_2 \equiv \gamma$ and $q : \beta_1 \bowtie \beta_2 \equiv \gamma$ of the same scope γ, a **four-way separation** into scopes γ_1, γ_2, γ_3, and γ_4 such that $\gamma_1 \bowtie \gamma_2 \equiv \alpha_1$, $\gamma_3 \bowtie \gamma_4 \equiv \alpha_2$, $\gamma_1 \bowtie \gamma_3 \equiv \beta_1$, and $\gamma_2 \bowtie \gamma_4 \equiv \beta_2$.

To address these problems, it may be necessary still to switch to a different representation of scopes or scope representations. However, at the moment is is not even clear whether such a representation even exists. This leads us to the following question: *is possible to give an implementation of scopes and scope separation that satisfies all the properties 1-7, while keeping the size of separation proofs bounded by the size of the scopes?* While the representation of scopes presented here does not yet answer this question, the interface it offers provides new insight into the kind of properties we can enforce by using dependent and quantitative types. It is thus a first step towards an unexplored and exciting world of new variable representations.

References

Robert Atkey. Syntax and semantics of quantitative type theory. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 56–65. ACM, 2018. doi: 10.1145/3209108.3209189. URL http://doi.acm.org/10.1145/3209108.3209189.

Edwin Brady, Conor McBride, and James McKinna. Inductive families need not store their indices. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, April 30 - May 4, 2003, Revised Selected Papers*, volume 3085 of *Lecture Notes in Computer Science*, pages 115–129. Springer, 2003. ISBN 3-540-22164-6. URL http://springerlink.metapress.com/openurl.asp?genre=article&issn=0302-9743&volume=3085&spage=115.

Conor McBride. I got plenty o' nuttin'. In Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella, editors, *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *Lecture Notes in Computer Science*, pages 207–233. Springer, 2016. ISBN 978-3-319-30935-4. doi: 10.1007/978-3-319-30936-1_12. URL http://dx.doi.org/10.1007/978-3-319-30936-1_12.

Nicolas Pouillard. *Namely, Painless: A unifying approach to safe programming with first-order syntax with binders. (Une approche unifiante pour programmer sûrement avec de la syntaxe du premier ordre contenant des lieurs)*. PhD thesis, Paris Diderot University, France, 2012. URL https://tel.archives-ouvertes.fr/tel-00759059.

Arjen Rouvoet, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. Intrinsically-typed definitional interpreters for linear, session-typed languages. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 284–298. ACM, 2020. ISBN 978-1-4503-7097-4. doi: 10.1145/3372885.3373818. URL https://doi.org/10.1145/3372885.3373818.