JFP, 11 pages, 2021.
doi:10.1017/xxxxx

FUNCTIONAL PEARL

A correct-by-construction conversion to combinators

WOUTER SWIERSTRA
Utrecht University
(e-mail: w.s.swierstra@uu.nl)

Abstract

This pearl defines a translation from well-typed lambda terms to combinatory logic, where the preservation of types and the correctness of the translation is enforced statically.

1 Introduction

Historically, there is a close connection between the lambda calculus and combinatory logic (Curry *et al.*, 1958; Schönfinkel, 1924). In this pearl, we will show how to implement a translation from the simply-typed lambda terms to combinators such that both the *types* and *semantics* of each lambda term is preserved. While we could do so by defining a translation scheme and proving these facts post-hoc, we instead uncover a solution that is correct by construction and requires no further proofs or postulates.

2 Lambda calculus

To set the scene, we start by defining an evaluator for the simply typed lambda calculus in the dependently typed programming language Agda (Norell, 2007). This evaluator features in numerous papers and introductions on programming with dependent types (McBride, 2004; Norell, 2009, 2013; Abel, 2016), yet we include it here in its entirety for the sake of completeness.

Types

The *types* of our lambda calculus consist of a single base type (ι) and functions between types, denoted using the function space operator (⇒):

```
data U : Set where
  ι     : U
  _⇒_ : U → U → U
```

We can map these types to their Agda counterparts:

```
Val : U → Set
Val ι          = A
Val (u₁ ⇒ u₂) = Val u₁ → Val u₂
```

Here the interpretation of the base type, ι, is mapped to a type $\mathsf{A} : \mathsf{Set}$, that we provide as a module parameter. The remainder of this development does not depend on the interpretation of our base type in any meaningful way.

Finally, we will represent *contexts* as lists of such types:

```
Ctx = List U
```

Typically, we will use variable names drawn from the Greek alphabet to refer to types (such as σ and τ) and contexts (Γ).

Terms

Before we define the *terms* of the simply typed lambda calculus, we need to decide on how to treat variables. Initially, we will define the following inductive family, modelling valid references to a type σ in a given context Γ:

```
data Ref (σ : U) : Ctx → Set where
  Top : Ref σ (σ :: Γ)
  Pop : Ref σ Γ → Ref σ (τ :: Γ)
```

Erasing the type indices, we are left with the Peano natural numbers—corresponding to the typical De Bruijn representation of variable binding.

We can now define the data type for well-typed, well-scoped lambda terms as follows:

```
data Term : Ctx → U → Set where
  App : Term Γ (σ ⇒ τ) → Term Γ σ → Term Γ τ
  Lam : Term (σ :: Γ) τ → Term Γ (σ ⇒ τ)
  Var : Ref σ Γ → Term Γ σ
```

Each constructor mirrors a familiar typing rule: applications require the domain and argument's type to coincide; lambda abstractions introduce a new variable in the context of the lambda's body; the Var constructor may be used to refer to any variable currently in scope.

Evaluation

The dependent types in the definition of Term pay dividends once we try to define an evaluator for lambda terms. Before we can do so, however, we need to introduce a data type for *environments*:

```
data Env : Ctx → Set where
  Nil  : Env []
  Cons : Val σ → Env Γ → Env (σ :: Γ)
```

An environment stores a value for each variable in the context Γ, as witnessed by the following lookup function:

```
lookup : Ref σ Γ → Env Γ → Val σ
lookup Top       (Cons x env) = x
lookup (Pop ref) (Cons x env) = lookup ref env
```

Note that this function is *total*. The type indices ensure that there is no valid variable in the empty context; correspondingly, the lookup function need never worry about returning a value when the environment is empty.

We can now define an evaluator for the simply typed lambda calculus:

```
⟦_⟧ : Term Γ σ → (Env Γ → Val σ)
⟦ App t₁ t₂ ⟧ = λ env → (⟦ t₁ ⟧ env) (⟦ t₂ ⟧ env)
⟦ Lam t ⟧     = λ env → λ x → ⟦ t ⟧ (Cons x env)
⟦ Var x ⟧     = λ env → lookup x env
```

That this code type checks at all is somewhat surprising at first. It maps App constructors to Agda's application and Lam constructors to Agda's built-in lambda construct. Once again, the type indices ensure that the evaluation of the Lam construct must return a function (and hence we may introduce a lambda). Similarly in the case for applications, evaluating t_1 will return a function whose domain coincides with the type of the value arising from the evaluation of t_2. Finally, the environment of type Env Γ passed as an argument contains just the right values for all the variables drawn from the context Γ.

3 Translation to combinatory logic

Before we can define the translation from lambda terms to combinators, we need to fix our target language. As a first attempt, we might try something along the following lines, replacing the Lam constructor in the Term data type with the three familiar combinators from combinatory logic, S, K, and I:

```
data Comb : Set where
  S K I : Comb
  App : Comb → Comb → Comb
  Var : ... → Comb
```

Yet if we aim for our translation to be type-preserving, the very least we can do is decorate our combinators with the same type information as our lambda terms:

```
data Comb (Γ : Ctx) : U → Set where
  S   : Comb Γ ((σ ⇒ τ ⇒ τ') ⇒ (σ ⇒ τ) ⇒ (σ ⇒ τ'))
  K   : Comb Γ (σ ⇒ τ ⇒ σ)
  I   : Comb Γ (σ ⇒ σ)
  App : Comb Γ (σ ⇒ τ) → Comb Γ σ → Comb Γ τ
  Var : Ref σ Γ → Comb Γ σ
```

The types of both the App and Var constructors are the same as we saw for the lambda terms; the types of the S, K, and I combinators is fixed by their intended reduction behaviour:

$$
\begin{aligned}
&\mathsf{S\,f\,g\,x} &&= (\mathsf{f\,x})\,(\mathsf{g\,x})\\
&\mathsf{K\,x\,y} &&= \mathsf{x}\\
&\mathsf{I\,x} &&= \mathsf{x}
\end{aligned}
$$

Note that—as our Comb lacks lambdas and cannot introduce new variables—the context is now a *parameter*, rather than an *index* as we saw for the Term data type. This is the essence of combinatory logic: a language with variables, but without binders.

Yet we will strive to do even better. We will define a translation that preserves both the static and dynamic semantics of our lambda terms. To achieve this, we index our combinators with *both* their types and their intended semantics, given by a function of type $\mathsf{Env}\,\Gamma \to \mathsf{Val}\,\mathsf{u}$, yielding this final version of our combinators:

```
data Comb : (Γ : Ctx) → (u : U) → (Env Γ → Val u) → Set where
  S   : Comb Γ ((σ ⇒ τ ⇒ τ') ⇒ (σ ⇒ τ) ⇒ σ ⇒ τ') (λ env → λ f g x → (f x) (g x))
  K   : Comb Γ (σ ⇒ (τ ⇒ σ)) (λ env → λ x y → x)
  I   : Comb Γ (σ ⇒ σ) (λ env → λ x → x)
  Var : (i : Ref σ Γ) → Comb Γ σ (λ env → lookup i env)
  App : Comb Γ (σ ⇒ τ) f → Comb Γ σ x → Comb Γ τ (λ env → (f env) (x env))
```

Here the type of each base combinator (S, K, and I) contains both its type and semantics. For example, the I combinator has type $\sigma \Rightarrow \sigma$ and corresponds to the lambda term $\lambda\,x \to x$. None of the combinators rely on the additional environment parameter env. This environment is used in the Var constructor; just as we saw in our evaluator for lambda terms, this environment stores a value for each variable. Finally, the App constructor applies one combinator term to another. The type information for both the Var and App constructors coincide with their counterparts from the Term data type; their intended semantics can be read off from the evaluator for lambda terms, $[\![\, t \,]\!]$, that we defined previously.

The key difference between lambda terms and SKI combinators is the the lack of lambdas in the latter. To handle this, we define an auxiliary function, sometimes referred to as *bracket abstraction*, that maps one combinator term to another:

```
lambda : ∀ {f} → Comb (σ :: Γ) τ f → Comb Γ (σ ⇒ τ) (λ env x → f (Cons x env))
lambda S                = App K S
lambda K                = App K K
lambda I                = App K I
lambda (App t₁ t₂)      = App (App S (lambda t₁)) (lambda t₂)
lambda (Var Top)        = I
lambda (Var (Pop i))    = App K (Var i)
```

This behaviour of the lambda function should be clear from its type: given a Comb term of type τ using variables drawn from the context $\sigma :: \Gamma$, the lambda function returns a combinator of type $\sigma \Rightarrow \tau$ using variables drawn from the context Γ. Essentially, any occurrences of the Var Top are replaced with the identity I; the new argument is distributed over applications using the S combinator; any other variables or base combinators discard this new argument by introducing an additional K combinator.

With this definition in place, we can now define our type-preserving correct-by-construction translation. That is, we aim to define a translation with the following type:

$$\mathsf{translate} : (\mathsf{t} : \mathsf{Term}\ \Gamma\ \sigma) \to \mathsf{Comb}\ \Gamma\ \sigma\ [\![\ \mathsf{t}\ ]\!]$$

Here a lambda term of type σ in the context Γ is mapped to a combinator of type σ using variables drawn from the context Γ in such a way that the evaluation of t and semantics of the combinator are identical, namely, $[\![\ \mathsf{t}\ ]\!]$. The definition of this translation is now entirely straightforward:

$$\begin{aligned}
&\mathsf{translate}\ (\mathsf{App}\ \mathsf{t}_1\ \mathsf{t}_2) &&= \mathsf{App}\ (\mathsf{translate}\ \mathsf{t}_1)\ (\mathsf{translate}\ \mathsf{t}_2)\\
&\mathsf{translate}\ (\mathsf{Lam}\ \mathsf{t}) &&= \mathsf{lambda}\ (\mathsf{translate}\ \mathsf{t})\\
&\mathsf{translate}\ (\mathsf{Var}\ \mathsf{i}) &&= \mathsf{Var}\ \mathsf{i}
\end{aligned}$$

To see why this code type checks, note that both the (dynamic) semantics of both the App and Var constructors of the Comb data type coincide precisely with their semantics as lambda terms, $[\![\ \mathsf{App}\ \mathsf{t}_1\ \mathsf{t}_2\ ]\!]$ and $[\![\ \mathsf{Var}\ \mathsf{i}\ ]\!]$ respectively. Finally, if translating the body of a lambda produces some Comb term f, the lambda function produces a combinator term with the semantics $\lambda\ \mathsf{env}\ \mathsf{x} \to \mathsf{f}\ (\mathsf{Cons}\ \mathsf{x}\ \mathsf{env})$. The similarity between the *type* of the lambda function and the Lam branch of our evaluator is no coincidence.

These combinators are not the only possible choice of combinatorial basis. In particular, the S combinator *always* passes its third argument to the first two—even if it discarded. Can we do better?

4 An optimising translation

We can extend our choice of combinatorial basis extending the SKI combinators with two new combinators B and C:

$$\begin{aligned}
\mathsf{B}\ \mathsf{f}\ \mathsf{g}\ \mathsf{x} &= (\mathsf{f}\ \mathsf{x})\ \mathsf{g}\\
\mathsf{C}\ \mathsf{f}\ \mathsf{g}\ \mathsf{x} &= \mathsf{f}\ (\mathsf{g}\ \mathsf{x})
\end{aligned}$$

When translating an application, we now need to select between four possible choices: K, B, C and S, depending on the variables that occur in the arguments of the application. How can we make this choice, while still guaranteeing that types and semantics are preserved accordingly? The key insight is that we need more information about the variables in our lambda terms.

Contexts and subsets

Previously used a single context to capture *all* the variables that *may* be used. To account for the variables that *have been* used, we need to keep track of a subset of this context. To do so, we begin by defining the following subset predicate:

```
data Subset : Ctx → Set where
  Empty : Subset Γ
  Drop  : Subset Γ → Subset (τ :: Γ)
  Keep  : Subset Γ → Subset (τ :: Γ)
```

A subset of some context Γ may be Empty, or it may choose to Keep or Drop the most recently bound variable of type τ. The intended semantics of such subsets can be given by computing the context underlying this subset predicate:

$$
\begin{aligned}
&\lfloor\_\rfloor \;:\; \mathsf{Subset}\,\Gamma \to \mathsf{Ctx}\\
&\lfloor\,\mathsf{Empty}\,\rfloor && = []\\
&\lfloor\,\mathsf{Drop}\,\Delta\,\rfloor && = \lfloor\,\Delta\,\rfloor\\
&\lfloor\,\mathsf{Keep}\,\{\tau = \tau\}\;\Delta\,\rfloor && = \tau :: \lfloor\,\Delta\,\rfloor
\end{aligned}
$$

The Empty subset corresponds to the empty context; the Drop constructor discards the most recently bound variable, whereas the Keep constructor retains it. There is some choice here in the design of the Subset predicate. We could have chosen that the Empty constructor is the only possible subset of the empty context, Nil. Instead, we have allowed for a bit more wiggle room—the Empty subset is a subset of any context Γ. As we shall see later on, this distinction will be important. Note that 'sublist' would be slightly more precise than subset. We will, however, use the more common terminology of subsets to refer to the collection of variables that occur in a given lambda term.

We will need three separate operations for manipulating subsets. Not entirely coincidentally, each of these operations will be used to account for the variables used in each constructor from our Term data type. First of all, we can compute the union of two subsets:

$$
\begin{aligned}
&\_\cup\_ \;:\; \mathsf{Subset}\,\Gamma \to \mathsf{Subset}\,\Gamma \to \mathsf{Subset}\,\Gamma\\
&\mathsf{Empty} && \cup\; \mathsf{sub} && = \mathsf{sub}\\
&\mathsf{Drop}\,\mathsf{xs} && \cup\; \mathsf{Drop}\,\mathsf{ys} && = \mathsf{Drop}\,(\mathsf{xs} \cup \mathsf{ys})\\
&\mathsf{Drop}\,\mathsf{xs} && \cup\; \mathsf{Keep}\,\mathsf{ys} && = \mathsf{Keep}\,(\mathsf{xs} \cup \mathsf{ys})\\
&\mathsf{Drop}\,\mathsf{xs} && \cup\; \mathsf{Empty} && = \mathsf{Drop}\,\mathsf{xs}\\
&\mathsf{Keep}\,\mathsf{xs} && \cup\; \mathsf{Drop}\,\mathsf{ys} && = \mathsf{Keep}\,(\mathsf{xs} \cup \mathsf{ys})\\
&\mathsf{Keep}\,\mathsf{xs} && \cup\; \mathsf{Keep}\,\mathsf{ys} && = \mathsf{Keep}\,(\mathsf{xs} \cup \mathsf{ys})\\
&\mathsf{Keep}\,\mathsf{xs} && \cup\; \mathsf{Empty} && = \mathsf{Keep}\,\mathsf{xs}
\end{aligned}
$$

The first element of a union of two subsets is only discarded, when both subsets discard this fist element. If at least one of the two subsets retains the head element, so does the union. The remaining cases state that the empty set is the left and right identity of the union operation.

Next, we can map any variable reference of type $\mathsf{Ref}\,\tau\,\Gamma$ to a singleton subset containing that reference and nothing else:

$$
\begin{aligned}
&[\_] \;:\; \mathsf{Ref}\,\tau\,\Gamma \to \mathsf{Subset}\,\Gamma\\
&[\,\mathsf{Top}\,] && = \mathsf{Keep}\,\mathsf{Empty}\\
&[\,\mathsf{Pop}\,\mathsf{p}\,] && = \mathsf{Drop}\,[\,\mathsf{p}\,]
\end{aligned}
$$

Finally, any subset of $\sigma :: \Gamma$ determines a unique subset of Γ, by simply ignoring whether the first element is kept or discarded:

$$
\begin{aligned}
&\mathsf{pop} \;:\; \mathsf{Subset}\,(\sigma :: \Gamma) \to \mathsf{Subset}\,\Gamma\\
&\mathsf{pop}\,(\mathsf{Drop}\,\mathsf{s}) && = \mathsf{s}\\
&\mathsf{pop}\,(\mathsf{Keep}\,\mathsf{s}) && = \mathsf{s}\\
&\mathsf{pop}\,(\mathsf{Empty}) && = \mathsf{Empty}
\end{aligned}
$$

This pop operation is reminiscent of the tail operation on lists, discarding the information about the first element and returning the remaining subset.

Co-de Bruijn terms

We can now revisit our Term data type, providing additional information about the variables that occur in a given term. This representation of variables, sometimes referred to as the co-De Bruijn representation (McBride, 2018), extends our previous Term data type with an additional index of type Subset Γ. This subset records the variables used in each (sub)term:

```
data Term (Γ : Ctx) : Subset Γ → U → Set where
  Lam : Term (σ :: Γ) Δ τ → Term Γ (pop Δ) (σ ⇒ τ)
  App : Term Γ Δ₁ (σ ⇒ τ) → Term Γ Δ₂ σ → Term Γ (Δ₁ ∪ Δ₂) τ
  Var : (i : Ref σ Γ) → Term Γ [ i ] σ
```

Each constructor has the same context and type index as we saw previously; the only new type information is in the subset, Δ. Lambda abstractions bind the top variable in the context; applications take the union of the two subsets of variables associated with each subterm; the variables associated with a Var constructor consists of a singleton subset of that variable.

As it stands, this revised Term data type accurately captures the invariant in which we are interested: which variables are used by a term. This does, however, come at a price. The types of all the constructors of the Term data type now contain *functions* (pop, union and singleton), where previously they only used *variables* and *constructors*. One consequence of this design choice, however, is that it may complicate pattern matching: deciding the possible constructors that may inhabit a term of some given type, becomes much more difficult. To illustrate this, suppose we want to define a function with the taking an argument of type:

```
Term (σ :: Γ) (Keep Δ) τ
```

Which constructors of the Term data type should we match against? This is not a trivial question: it involves checking the *implementation* of the singleton and union functions to determine whether or not they can produce a subset of the form Keep Δ. As this is not decidable in general, Agda cannot case split on such terms. To address this, we will sometimes use *relations*, expressing the graph of a given function instead. For example, consider the Union data type defined as follows:

```
data Union : Subset Γ → Subset Γ → Subset Γ → Set where
  Empty₁   : Union Empty Δ Δ
  Empty₂   : Union Δ Empty Δ
  Drop     : Union Δ₁ Δ₂ Δ → Union (Drop Δ₁) (Drop Δ₂) (Drop Δ)
  KeepDrop : Union Δ₁ Δ₂ Δ → Union (Keep Δ₁) (Drop Δ₂) (Keep Δ)
  DropKeep : Union Δ₁ Δ₂ Δ → Union (Drop Δ₁) (Keep Δ₂) (Keep Δ)
  KeepKeep : Union Δ₁ Δ₂ Δ → Union (Keep Δ₁) (Keep Δ₂) (Keep Δ)
```

We can show that the $\mathsf{Union}\ \Delta_1\ \Delta_2\ \Delta$ data type is inhabited precisely when $\mathsf{union}\ \Delta_1\ \Delta_2$ is Δ:

$$\mathsf{union} : (\Delta_1\ \Delta_2 : \mathsf{Subset}\ \Gamma) \to \mathsf{Union}\ \Delta_1\ \Delta_2\ (\Delta_1 \cup \Delta_2)$$

The definition of the union function follows the same structure of recursion as the $\_\cup\_$ function, selecting the appropriate case of the Union relation depending on its inputs. Similarly, the $\mathsf{Singleton}$ relation and $\mathsf{singleton}$ function express that a subset of the context Γ is equal to the singleton $[\ \mathsf{i}\ ]$, for some reference i:

```
data Singleton {σ : U} : Ref σ Γ → Subset Γ → Set where
  Here : Singleton (Top {Γ = Γ}) (Keep Empty)
  There : (i : Ref σ Γ) → Singleton i Δ → Singleton (Pop i) (Drop Δ)
singleton : (i : Ref τ Γ) → Singleton i [ i ]
singleton Top      = Here
singleton (Pop i)  = There i (singleton i)
```

Evaluation

We now turn our attention to defining an evaluator for our new Term data type. Rather than take an environment $\mathsf{Env}\ \Gamma$, storing a value for *all* possible variables drawn from Γ as we did previously, we now choose the following type for our evaluator:

$$[\![\_]\!] : \mathsf{Term}\ \Gamma\ \Delta\ \sigma \to (\mathsf{Env}\ \lfloor \Delta \rfloor \to \mathsf{Val}\ \sigma)$$

Here we take an environment of type $\mathsf{Env}\ \lfloor \Delta \rfloor$ as argument, storing variables for all the variables that *are* used, rather than the variables that *might be* used. To accomplish this, we need to define three functions for projecting the relevant parts of an argument environment:

```
prj₁ : Union Δ₁ Δ₂ Δ → Env ⌊ Δ ⌋ → Env ⌊ Δ₁ ⌋
prj₂ : Union Δ₁ Δ₂ Δ → Env ⌊ Δ ⌋ → Env ⌊ Δ₂ ⌋
prj : {i : Ref σ Γ} → Singleton i Δ → Env ⌊ Δ ⌋ → Val σ
```

Each of these definitions follows the inductive structure of the Union and $\mathsf{Singleton}$ relations, triggering the reduction in the $\mathsf{Env}\ \lfloor \Delta \rfloor$ type.

The evaluator itself is only slightly more complicated than the one we saw initially:

```
⟦_⟧ : Term Γ Δ σ → (Env ⌊ Δ ⌋ → Val σ)
⟦ Lam {Δ = Keep Δ} t ⟧             = λ env → λ x → ⟦ t ⟧ (Cons x env)
⟦ Lam {Δ = Drop Δ} t ⟧             = λ env → λ _ → ⟦ t ⟧ env
⟦ Lam {Δ = Empty} t ⟧              = λ env → λ _ → ⟦ t ⟧ Nil
⟦ Var i ⟧                          = λ env → prj (singleton i) env
⟦ App {Δ₁ = Δ₁} {Δ₂ = Δ₂} t₁ t₂ ⟧  = λ env → let env₁ = prj₁ (union Δ₁ Δ₂) env in
                                              let env₂ = prj₂ (union Δ₁ Δ₂) env in
                                              (⟦ t₁ ⟧ env₁) (⟦ t₂ ⟧ env₂)
```

The Var case projects the single value of type $\mathsf{Val}\ \sigma$ from the argument environment. The case for applications recurses as expected, projecting the relevant values from the input environment. Finally, the case for lambda abstractions is more interesting. Evaluation continues on the body of the lambda t, but the environment is only extended with the freshly bound variable x, when it occurs in t. Otherwise, it can be safely discarded.

Combinators revisited

We can now revisit our language representing the terms of combinatory logic. Initially, our data type declaration had the following form:

```
data Comb : (Γ : Ctx) → (σ : U) → (Env Γ → Val σ) → Set where
```

However, we now want to keep track of an additional subset Δ : Subset Γ, that tracks the variables that occur in each subterm. Hence we may consider defining a type for combinators using the following declaration:

```
data Comb : (Γ : Ctx) → (Δ : Subset Γ) → (σ : U) → (Env Γ → Val σ) → Set where
```

Yet there is one further adaptation necessary. Previously, our evaluator mapped a term in Term Γ σ to a function Env Γ → Val σ. In the preceding pages, however, our evaluator now takes an environment of type Env ⌊ Δ ⌋ as its argument. Correspondingly, we declare our data type for combinatory logic terms as follows:

```
data Comb : (Γ : Ctx) → (Δ : Subset Γ) → (σ : U) → (Env ⌊ Δ ⌋ → Val σ) → Set where
  S : Comb Γ Empty ((σ ⇒ (τ ⇒ τ')) ⇒ ((σ ⇒ τ) ⇒ (σ ⇒ τ'))) λ env f g x → (f x) (g x)
  B : Comb Γ Empty ((σ ⇒ τ) ⇒ (τ' ⇒ σ) ⇒ (τ' ⇒ τ)) λ env f g x → f (g x)
  C : Comb Γ Empty ((σ ⇒ τ ⇒ τ') ⇒ τ ⇒ σ ⇒ τ') λ env f g x → (f x) g
  K : Comb Γ Empty (σ ⇒ (τ ⇒ σ)) λ env x y → x
  I : Comb Γ Empty (σ ⇒ σ) λ env x → x
  App : ∀ {f x} → Comb Γ Δ₁ (σ ⇒ τ) f → Comb Γ Δ₂ σ x →
    (u : Union Δ₁ Δ₂ Δ) →
    Comb Γ Δ τ λ env → (f (prj₁ u env)) (x (prj₂ u env))
  Var : (i : Ref σ Γ) → (s : Singleton i Δ) → Comb Γ Δ σ (λ env → prj s env)
```

In principle, not much has changed. The base combinator, such as S and B, are decorated with an empty subset. The general pattern—adding additional information about their types and corresponding lambda terms—should be familiar by now. The constructors for application, App, and variables, Var, are variations of the ones we saw previously. Each application records how the union of the variables in the subsets Δ_1 and Δ_2 gives rise to Δ. Similarly, the subset associated with a variable is a singleton. The behaviour associated with each of these constructors can be read off of the cases for application and variables from the semantics of terms we defined previously.

To complete our development, we now seek to define a translation from lambda terms to these combinators:

```
translate : (t : Term Γ Δ σ) → Comb Γ Δ σ ⟦ t ⟧
```

Just as we saw previously, we can map variables to variables and applications to applications. The key question is: how do we handle lambda bindings? Previously, we defined a single function:

$$\mathsf{lambda} : \mathsf{Comb}\ (\sigma :: \Gamma)\ \tau\ \mathsf{f} \to \mathsf{Comb}\ \Gamma\ (\sigma \Rightarrow \tau)\ (\lambda\ \mathsf{env}\ \mathsf{x} \to \mathsf{f}\ (\mathsf{Cons}\ \mathsf{x}\ \mathsf{env}))$$

Yet we can now see from the evaluation function, that we will need to distinguish between whether or not the freshly bound variable x is used. Before we can do so, however, we define a pair of functions:

$$\begin{aligned}
&\mathsf{str\text{-}stop} : \forall \{\mathsf{f} : \mathsf{Env}\ [\,] \to \mathsf{Val}\ \tau\} &&\to \mathsf{Comb}\ (\sigma :: \Gamma)\ (\mathsf{Empty})\ \tau\ \mathsf{f} &&\to \mathsf{Comb}\ \Gamma\ \mathsf{Empty}\ \tau\ \mathsf{f}\\
&\mathsf{str\text{-}drop} : \forall \{\mathsf{f} : \mathsf{Env}\ \lfloor \Delta \rfloor \to \mathsf{Val}\ \tau\} &&\to \mathsf{Comb}\ (\sigma :: \Gamma)\ (\mathsf{Drop}\ \Delta)\ \tau\ \mathsf{f} &&\to \mathsf{Comb}\ \Gamma\ \Delta\ \tau\ \mathsf{f}
\end{aligned}$$

These functions are dual to the usual ‘weakening’ operation, where you introduce an unused variable. Instead, they establish a ‘strengthening’ principle, proving that unused arguments may be discarded safely. Their definition is entirely straightforward, pattern matching on the argument combinator and recursing over applications.

We can use such strengthening functions to define our first variation of the lambda function. If we know that an argument is unused, we can introduce the K combinator directly and discard it:

$$\begin{aligned}
&\mathsf{drop\text{-}lambda} : \forall \{\mathsf{f}\} \to\\
&\quad \mathsf{Comb}\ (\sigma :: \Gamma)\ (\mathsf{Drop}\ \Delta)\ \tau\ \mathsf{f} \to \mathsf{Comb}\ \Gamma\ \Delta\ (\sigma \Rightarrow \tau)\ (\lambda\ \mathsf{env}\ _ \to \mathsf{f}\ \mathsf{env})\\
&\mathsf{drop\text{-}lambda}\ \mathsf{t} = \mathsf{App}\ \mathsf{K}\ (\mathsf{str\text{-}drop}\ \mathsf{t})\ \mathsf{Empty}_1
\end{aligned}$$

The more interesting case is when the freshly bound variable *is* used in the combinator we have constructed so far. To handle this case, we introduce a keep-lambda function whose type signature mirrors the lambda function we saw previously:

$$\begin{aligned}
&\mathsf{keep\text{-}lambda} : \forall \{\mathsf{f}\} \to\\
&\quad \mathsf{Comb}\ (\sigma :: \Gamma)\ (\mathsf{Keep}\ \Delta)\ \tau\ \mathsf{f} \to \mathsf{Comb}\ \Gamma\ \Delta\ (\sigma \Rightarrow \tau)\ (\lambda\ \mathsf{env}\ \mathsf{v} \to \mathsf{f}\ (\mathsf{Cons}\ \mathsf{v}\ \mathsf{env}))
\end{aligned}$$

It is here that we must decide which combinator to use, depending on which subterms depend on the freshly bound variable. We can distinguish the following cases:

$$\begin{aligned}
&\mathsf{keep\text{-}lambda}\ (\mathsf{Var}\ \mathsf{Top}\ \mathsf{Here}) \qquad\qquad = \mathsf{I}\\
&\mathsf{keep\text{-}lambda}\ (\mathsf{App}\ \mathsf{t}_1\ \mathsf{t}_2\ (\mathsf{KeepKeep}\ \mathsf{u})) =\\
&\quad \mathsf{App}\ (\mathsf{App}\ \mathsf{S}\ (\mathsf{keep\text{-}lambda}\ \mathsf{t}_1)\ \mathsf{Empty}_1)\ (\mathsf{keep\text{-}lambda}\ \mathsf{t}_2)\ \mathsf{u}\\
&\mathsf{keep\text{-}lambda}\ (\mathsf{App}\ \mathsf{t}_1\ \mathsf{t}_2\ (\mathsf{KeepDrop}\ \mathsf{u})) =\\
&\quad \mathsf{App}\ (\mathsf{App}\ \mathsf{C}\ (\mathsf{keep\text{-}lambda}\ \mathsf{t}_1)\ \mathsf{Empty}_1)\ (\mathsf{str\text{-}drop}\ \mathsf{t}_2)\ \mathsf{u}\\
&\mathsf{keep\text{-}lambda}\ (\mathsf{App}\ \mathsf{t}_1\ \mathsf{t}_2\ (\mathsf{DropKeep}\ \mathsf{u})) =\\
&\quad \mathsf{App}\ (\mathsf{App}\ \mathsf{B}\ (\mathsf{str\text{-}drop}\ \mathsf{t}_1)\ \mathsf{Empty}_1)\ (\mathsf{keep\text{-}lambda}\ \mathsf{t}_2)\ \mathsf{u}
\end{aligned}$$

The first two cases are familiar: we can use the S combinator to pass the freshly bound variable to both arguments of an application; the only possible variable is Var Top, which we map to the I combinator as before. The next two cases are new: depending on which subterm depends on the freshly bound variable, we select the B or C combinator accordingly. We recurse over the subterm that uses the variable; we apply our strengthening principle to the subterm that does not, proving that this variable can be safely discarded. The complete definition of keep-lambda includes two cases, corresponding to the Empty_1 and Empty_2 constructors of the Union data type, which are handled similarly.

The two functions, keep-lambda and drop-lambda, pattern match on a combinator term with a *particular* subset of variables. If we were to use functions rather than relations in the indices of the constructors of the Comb data type, we would run into unification problems during pattern matching. The relations, while introducing some overhead, clarify the three cases that can result in a subset of variables of the from $\mathsf{Keep}\ \Delta$.

These functions rely on the empty set being a left identity of the union of subsets. For example, in the drop-lambda function, we build an application of the form $\mathsf{App\ K\ (str\text{-}drop\ t)}$. To establish this has the desired subset Δ, we need to show that the union arising from the application, also has the index Δ. This follows from the Empty_1 constructor of the Union data type. Note that if we had taken the empty subset Empty to have type $\mathsf{Subset}\ []$, this would no longer hold definitionally and would instead require an additional lemma proving this fact.

Using these two variations of the lambda function, we can revisit our translation to combinatory logic:

$$
\begin{aligned}
&\mathsf{translate} : (\mathsf{t} : \mathsf{Term}\ \Gamma\ \Delta\ \sigma) \to \mathsf{Comb}\ \Gamma\ \Delta\ \sigma\ [\![\ \mathsf{t}\ ]\!] \\
&\mathsf{translate}\ (\mathsf{App}\ \mathsf{t}_1\ \mathsf{t}_2) && = \mathsf{App}\ (\mathsf{translate}\ \mathsf{t}_1)\ (\mathsf{translate}\ \mathsf{t}_2)\ (\mathsf{union}\ \_\ \_) \\
&\mathsf{translate}\ (\mathsf{Lam}\ \{\Delta = \mathsf{Drop}\ \Delta\}\ \mathsf{t}) && = \mathsf{drop\text{-}lambda}\ (\mathsf{translate}\ \mathsf{t}) \\
&\mathsf{translate}\ (\mathsf{Lam}\ \{\Delta = \mathsf{Keep}\ \Delta\}\ \mathsf{t}) && = \mathsf{keep\text{-}lambda}\ (\mathsf{translate}\ \mathsf{t}) \\
&\mathsf{translate}\ (\mathsf{Lam}\ \{\Delta = \mathsf{Empty}\}\ \mathsf{t}) && = \mathsf{App}\ \mathsf{K}\ (\mathsf{str\text{-}stop}\ (\mathsf{translate}\ \mathsf{t}))\ \mathsf{Empty}_2 \\
&\mathsf{translate}\ (\mathsf{Var}\ \mathsf{x}) && = \mathsf{Var}\ \mathsf{x}\ (\mathsf{singleton}\ \mathsf{x})
\end{aligned}
$$

As we saw previously, applications and variables are not particularly interesting. The real work is done in the case for lambda bindings. Depending on if the freshly bound variable is used or not, either keep-lambda or drop-lambda is invoked, after translating the lambda's body.

5 Reflection

Although the translation schemes are reasonably straightforward, finding them was not. Writing dependently typed programs in this style—folding a program's specification into its type—may feel like a bit of a parlour trick, where the right choice of definitions ensure the entire construction is correct. Yet reading through these definitions *post hoc*—like so often with Agda programs—does not always tell how they were written.

In particular, the *type safe* translation from lambda terms to SKI combinators is a question I have set my students in the past. Proving this translation correct, requires defining a semantics for combinatory terms and showing that the translation is semantics preserving. Interestingly, this direct proof requires an axiom—functional extensionality—in the case for lambdas, as we need to prove two functions equal. Yet the *structure* of proof is simple enough: it relies exclusively on induction hypotheses and a property of the lambda function. It is this observation that makes it possible to incorporate the correctness proofs in the definitions themselves. Extending the translation scheme to also use the B and C combinators is harder—but follows naturally once you have the right choice of variable representation.

As our starting point, we have taken the 'traditional' simply-typed lambda calculus. More recent work by Kiselyov (2018), shows how a slight modification to the traditional typing rules allows for a denotation semantics as combinators directly. Formalising this in a proof assistant, however, is left as an exercise for the reader.

References

Abel, A. (2016) Agda tutorial. *13th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings*. Springer.

Curry, H. B., Feys, R., Craig, W., Hindley, J. R. and Seldin, J. P. (1958) *Combinatory logic*. Vol. 1. North-Holland Amsterdam.

Kiselyov, O. (2018) *lambda* to ski, semantically. *International Symposium on Functional and Logic Programming* pp. 33–50. Springer.

McBride, C. (2004) Epigram: Practical programming with dependent types. *International School on Advanced Functional Programming* pp. 130–170. Springer.

McBride, C. (2018) Everybody's got to be somewhere. *Electronic Proceedings in Theoretical Computer Science* **275**(Jul):53–69.

Norell, U. (2007) *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology.

Norell, U. (2009) Dependently typed programming in Agda. *Advanced Functional Programming: 6th International School AFP* pp. 230–266. Springer Berlin Heidelberg.

Norell, U. (2013) Interactive programming with dependent types. *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ICFP '13, pp. 1–2. ACM.

Schönfinkel, M. (1924) Über die bausteine der mathematischen logik. *Mathematische annalen* **92**(3):305–316.