A correct-by-construction conversion to combinators

Wouter Swierstra

Utrecht University

Executing combinator terms – Augustsson edition

```haskell
exec :: Comb -> Any
exec (App f e)  = (unsafeCoerce $ exec f) (unsafeCoerce $ exec e)
exec S          = unsafeCoerce $ \f g x -> (f x) (g x)
exec K          = unsafeCoerce $ const
exec I          = unsafeCoerce $ id
```

Executing combinator terms – Augustsson edition

```
exec :: Comb -> Any
exec (App f e)  = (unsafeCoerce $ exec f) (unsafeCoerce $ exec e)
exec S          = unsafeCoerce $ \f g x -> (f x) (g x)
exec K          = unsafeCoerce $ const
exec I          = unsafeCoerce $ id
```

What happened to static type safety?

Challenge

- Can we show that bracket abstraction is *type preserving*?
- Can we show that bracket abstraction is *semantics preserving*?

Challenge

- Can we show that bracket abstraction is *type preserving*?
- Can we show that bracket abstraction is *semantics preserving*?

And finally…

Can we establish this without writing any proofs?

Evaluation

Well typed lambda terms have a simple evaluator:

```
eval : ∀ {Γ s} → Term Γ s → Env Γ → Val s
eval (App f x) env  = (eval f env) (eval x env)
eval (Lam t) env    = λ x → eval t (x ∷ env)
eval (Var i) env    = lookup i env
```

Problem (again)

- Can we show that this translation to combinators is *type preserving*?

Can we define a 'typed combinatory logic' and a translation that is obviously type preserving:

```
translate : forall {Γ σ} -> (t : Term Γ σ) -> Comb Γ σ
```

Problem (again)

- Can we show that this translation to combinators is *type preserving*?

Can we define a ‘typed combinatory logic’ and a translation that is obviously type preserving:

```
translate : forall {Γ σ} → (t : Term Γ σ) → Comb Γ σ
```

- Can we show that this translation is also *semantics preserving*?

Can we prove that our translation is correct:

```
eval t env ≡ evalComb (translate t) env
```

Well typed combinator 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 Γ σ → Comb σ Γ

translate : Term Γ σ → Comb Γ σ
translate (App t₁ t₂)  = App (translate t₁) (translate t₂)
translate (Lam t₁)     = bracket (translate t₁)
translate (Var x)      = Var x
```

Bracket abstraction

```
bracket : Comb (σ ∷ Γ) τ f ⇾ Comb Γ (σ ⇒ τ)
bracket (App t₁ t₂)    = App (App S (bracket t₁)) (bracket t₂)
bracket S              = App K S
bracket K              = App K K
bracket I              = App K I
bracket (Var Top)      = I
bracket (Var (Pop j)) = App K (Var j)
```

All the types go through easily enough…

Bracket abstraction

```
bracket : Comb (σ :: Γ) τ f → Comb Γ (σ ⇒ τ)
bracket (App t₁ t₂)   = App (App S (bracket t₁)) (bracket t₂)
bracket S             = App K S
bracket K             = App K K
bracket I             = App K I
bracket (Var Top)     = I
bracket (Var (Pop j)) = App K (Var j)
```

All the types go through easily enough…

What about proving correctness?

Correctness – take 1

We define an evaluator for combinatory terms:

```
evalComb : Comb Γ σ -> Env Γ -> Val σ
evalComb K env         = \x y -> x
evalComb I env         = \x ->x
evalComb (App f e) env = (evalComb f env) (evalComb e env)
...
```

And prove the desired property – we need one lemma:

```
bracket-correct : (t : Comb (σ :: Γ) τ) (env : Env Γ) (v : Val σ) ->
  evalComb (bracket t) env v ≡ evalComb t (v :: env)
```

The proof itself is not very interesting – it follows *immediately* from our induction hypotheses.

Correctness – take 1

We define an evaluator for combinatory terms:

```
evalComb : Comb Γ σ -> Env Γ -> Val σ
evalComb K env         = \x y -> x
evalComb I env         = \x ->x
evalComb (App f e) env = (evalComb f env) (evalComb e env)
...
```

And prove the desired property – we need one lemma:

```
bracket-correct : (t : Comb (σ ∷ Γ) τ) (env : Env Γ) (v : Val σ) ->
  evalComb (bracket t) env v ≡ evalComb t (v ∷ env)
```

The proof itself is not very interesting – it follows *immediately* from our induction hypotheses.

If the proof is so obvious – can we make our translation correct by construction?

Correctness - take 2

```
data Comb : (Γ : Ctx) → (u : U) → (Env Γ → Val u) → Set where
  K    : Comb Γ (σ ⇒ (τ ⇒ σ)) (λ env → λ x y → x)
  I    : Comb Γ (σ ⇒ σ) (λ env → λ x → x)
  ...

translate : (t : Term Γ σ) → Comb Γ σ (eval t)
translate (app t₁ t₂)  = app (translate t₁) (translate t₂)
translate (lam t)      = bracket (translate t)
translate (var i)      = var i
```

Correctness - take 2

```
data Comb : (Γ : Ctx) → (u : U) → (Env Γ → Val u) → Set where
  K    : Comb Γ (σ ⇒ (τ ⇒ σ)) (λ env → λ x y → x)
  I    : Comb Γ (σ ⇒ σ) (λ env → λ x → x)
  ...

translate : (t : Term Γ σ) → Comb Γ σ (eval t)
translate (app t₁ t₂)  = app (translate t₁) (translate t₂)
translate (lam t)      = bracket (translate t)
translate (var i)      = var i
```

So what does the new version of the `bracket` function do?

Bracket abstraction - correct by construction

```
bracket : ∀ {f} → Comb (σ ∷ Γ) τ f → Comb Γ (σ ⇒ τ) (λ env x → f (x ∷ env))
bracket (app t₁ t₂)      = app (bracket t₁) (bracket t₂)
bracket I                = app K I
bracket (var zero)       = I
bracket (var (succ i))   = app K (var i)
...
```

Note: the function in the type of `bracket` and the right-hand side of evaluation for lambda terms coincide precisely.

So what?

- These programs may seem like a bit of a ‘parlour trick’ – where you show off your dependently type trickery.

So what?

- These programs may seem like a bit of a ‘parlour trick’ – where you show off your dependently type trickery.

- But it’s the *nature* of the proof – immediate induction – that guarantees we can roll the translation and its correctness proof into one.

So what?

- These programs may seem like a bit of a ‘parlour trick’ – where you show off your dependently type trickery.
- But it’s the *nature* of the proof – immediate induction – that guarantees we can roll the translation and its correctness proof into one.
- **TODO** Port to Haskell.

So what?

- These programs may seem like a bit of a ‘parlour trick’ – where you show off your dependently type trickery.
- But it’s the *nature* of the proof – immediate induction – that guarantees we can roll the translation and its correctness proof into one.
- **TODO** Port to Haskell.

Not quite so easy… Unsaturated type families, no type-level lambda, unclear reduction rules for type families, and many other headaches.

So what?

- These programs may seem like a bit of a ‘parlour trick’ – where you show off your dependently type trickery.

- But it’s the *nature* of the proof – immediate induction – that guarantees we can roll the translation and its correctness proof into one.

- **TODO** Port to Haskell.

Not quite so easy… Unsaturated type families, no type-level lambda, unclear reduction rules for type families, and many other headaches.

- **The end**

Simple types

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

Ctx = List U

Val : U → Set
Val i = Bool
Val (u ⇒ u₁) = Val u → Val u₁
```

Well-typed lambda terms

```
data Ref (s : U) : Ctx → Set where
  Top : ∀ {Γ} → Ref s (s ∷ Γ)
  Pop : ∀ {Γ t} → Ref s Γ → Ref s (t ∷ Γ)

data Term : Ctx → U → Set where
  App : ∀ {Γ t s} → Term Γ (s ⇒ t) → Term Γ s → Term Γ t
  Lam : ∀ {Γ t s} → Term (s ∷ Γ) t → Term Γ (s ⇒ t)
  Var : ∀ {Γ s} → Ref s Γ → Term Γ s
```