A correct-by-construction conversion to combinators

Wouter Swierstra

Utrecht University
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
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?
• 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?
Well typed lambda terms have a simple evaluator:

\[
\begin{align*}
    \text{eval} : \forall \{\Gamma \ s\} & \rightarrow \text{Term} \ \Gamma \ s \ → \ \text{Env} \ \Gamma \ → \ \text{Val} \ s \\
    \text{eval} (\text{App} \ f \ x) \ \text{env} & = (\text{eval} \ f \ \text{env}) \ (\text{eval} \ x \ \text{env}) \\
    \text{eval} (\text{Lam} \ t) \ \text{env} & = \lambda \ x \ → \ \text{eval} \ t \ (x \ ∷ \ \text{env}) \\
    \text{eval} (\text{Var} \ i) \ \text{env} & = \text{lookup} \ i \ \text{env}
\end{align*}
\]
Problem (again)

• Can we show that this translation to combinators is \textit{type preserving}? 

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

\texttt{translate : } \texttt{forall (}\Gamma \sigma\texttt{) } \rightarrow \texttt{(t : Term } \Gamma \sigma \texttt{) } \rightarrow \texttt{Comb } \Gamma \sigma \texttt{)
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:

\[
\text{translate} : \forall \{\Gamma \sigma\} \rightarrow (t : \text{Term } \Gamma \sigma) \rightarrow \text{Comb } \Gamma \sigma
\]

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

Can we prove that our translation is correct:

\[
\text{eval } t \text{ env } = \text{evalComb (translate } t) \text{ env}
\]
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

\[
\text{bracket} : \text{Comb} \ (\sigma :: \Gamma) \ \tau \ f \rightarrow \text{Comb} \ \Gamma \ (\sigma \Rightarrow \tau)
\]

\[
bracket \ (\text{App} \ t_1 \ t_2) = \text{App} \ (\text{App} \ S \ (\text{bracket} \ t_1)) \ (\text{bracket} \ t_2)
\]

\[
bracket \ S = \text{App} \ K \ S
\]

\[
bracket \ K = \text{App} \ K \ K
\]

\[
bracket \ I = \text{App} \ K \ I
\]

\[
bracket \ (\text{Var} \ \text{Top}) = I
\]

\[
bracket \ (\text{Var} \ (\text{Pop} \ j)) = \text{App} \ K \ (\text{Var} \ j)
\]

All the types go through easily enough...
Bracket abstraction

\[
\text{bracket} : \text{Comb} \ (\sigma :: \Gamma) \ \tau \ f \rightarrow \text{Comb} \ \Gamma \ (\sigma \Rightarrow \tau)
\]

\[
\text{bracket} \ (\text{App} \ t_1 \ t_2) = \text{App} \ (\text{App} \ S \ (\text{bracket} \ t_1)) \ (\text{bracket} \ t_2)
\]

\[
\text{bracket} \ S = \text{App} \ K \ S
\]

\[
\text{bracket} \ K = \text{App} \ K \ K
\]

\[
\text{bracket} \ I = \text{App} \ K \ I
\]

\[
\text{bracket} \ (\text{Var} \ \text{Top}) = I
\]

\[
\text{bracket} \ (\text{Var} \ (\text{Pop} \ j)) = \text{App} \ K \ (\text{Var} \ j)
\]

All the types go through easily enough...

What about proving correctness?
We define an evaluator for combinatory terms:

\[
\text{evalComb} : \text{Comb} \Gamma \sigma \rightarrow \text{Env} \Gamma \rightarrow \text{Val} \sigma
\]

\[
\text{evalComb} K \text{ env} = \lambda x y \rightarrow x
\]

\[
\text{evalComb} I \text{ env} = \lambda x \rightarrow x
\]

\[
\text{evalComb} (\text{App} f e) \text{ env} = (\text{evalComb} f \text{ env}) (\text{evalComb} e \text{ env})
\]

...  

And prove the desired property – we need one lemma:

\[
\text{bracket-correct} : (t : \text{Comb} (\sigma :: \Gamma) \tau) (\text{env} : \text{Env} \Gamma) (v : \text{Val} \sigma) \rightarrow \\
\text{evalComb} (\text{bracket} t) \text{ env} v \equiv \text{evalComb} t (v :: \text{env})
\]

The proof itself is not very interesting – it follows immediately from our induction hypotheses.
We define an evaluator for combinatory terms:

\[
\text{evalComb} : \text{Comb} \; \Gamma \; \sigma \rightarrow \text{Env} \; \Gamma \rightarrow \text{Val} \; \sigma
\]
\[
\text{evalComb} \; K \; \text{env} = \lambda x \; y \rightarrow x
\]
\[
\text{evalComb} \; I \; \text{env} = \lambda x \rightarrow x
\]
\[
\text{evalComb} \; (\text{App} \; f \; e) \; \text{env} = (\text{evalComb} \; f \; \text{env}) \; (\text{evalComb} \; e \; \text{env})
\]

... 

And prove the desired property – we need one lemma:

\[
\text{bracket-correct} : (t : \text{Comb} \; (\sigma :: \Gamma) \; \tau) \; (\text{env} : \text{Env} \; \Gamma) \; (v : \text{Val} \; \sigma) \rightarrow
\]
\[
\text{evalComb} \; (\text{bracket} \; t) \; \text{env} \; v \equiv \text{evalComb} \; t \; (v :: \text{env})
\]

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

\[
\begin{align*}
\text{data} & \quad \text{Comb} : (\Gamma : \text{Ctx}) \to (u : U) \to (\text{Env} \Gamma \to \text{Val} u) \to \text{Set where} \\
K & \quad : \text{Comb} \Gamma (\sigma \Rightarrow (\tau \Rightarrow \sigma)) (\lambda \text{env} \to \lambda x y \to x) \\
I & \quad : \text{Comb} \Gamma (\sigma \Rightarrow \sigma) (\lambda \text{env} \to \lambda x \to x) \\
& \quad \ldots
\end{align*}
\]

translate : (t : \text{Term} \Gamma \sigma) \to \text{Comb} \Gamma \sigma (\text{eval} t)
translate (\text{app} t_1 t_2) = \text{app} (\text{translate} t_1) (\text{translate} t_2)
translate (\text{lam} t) = \text{bracket} (\text{translate} t)
translate (\text{var} i) = \text{var} i

So what does the new version of the bracket function do?
Bracket abstraction - correct by construction

\[ \text{bracket} : \forall \{ f \} \to \text{Comb} (\sigma :: \Gamma) \tau f \to \text{Comb} \Gamma (\sigma \Rightarrow \tau) (\lambda \text{env} x \to f (x :: \text{env})) \]

\begin{align*}
\text{bracket} (\text{app } t_1 t_2) &= \text{app} (\text{bracket } t_1) (\text{bracket } t_2) \\
\text{bracket } I &= \text{app } K \ I \\
\text{bracket } (\text{var } \text{zero}) &= I \\
\text{bracket } (\text{var } (\text{succ } i)) &= \text{app } K (\text{var } i)
\end{align*}

\[ \ldots \]

Note: the function in the type of bracket and the right-hand side of evaluation for lambda terms coincide precisely.
• These programs may seem like a bit of a ‘parlour trick’ – where you show off your dependently type trickery.

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
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 \((s : U) : \text{Ctx} \rightarrow \text{Set} \)** where

\[
\text{Top} : \forall \{\Gamma\} \rightarrow \text{Ref } s (s :: \Gamma)
\]

\[
\text{Pop} : \forall \{\Gamma t\} \rightarrow \text{Ref } s \Gamma \rightarrow \text{Ref } s (t :: \Gamma)
\]

**data** Term \(\text{Ctx} \rightarrow U \rightarrow \text{Set} \)** where

\[
\text{App} : \forall \{\Gamma t s\} \rightarrow \text{Term } \Gamma (s \Rightarrow t) \rightarrow \text{Term } \Gamma s \rightarrow \text{Term } \Gamma t
\]

\[
\text{Lam} : \forall \{\Gamma t s\} \rightarrow \text{Term } (s :: \Gamma) t \rightarrow \text{Term } \Gamma (s \Rightarrow t)
\]

\[
\text{Var} : \forall \{\Gamma s\} \rightarrow \text{Ref } s \Gamma \rightarrow \text{Term } \Gamma s
\]