A correct-by-construction conversion to combinators

IFIP 2.1 Online meeting

Wouter Swierstra

Utrecht University
The syntax of the lambda calculus should be familiar:

\[ t ::= x \]
\[ \mid t \, t \]
\[ \mid \lambda x.t \]

There is one key reduction rule, describing evaluation:

\[ (\lambda x.t) \, t' \rightarrow_{\beta} t[x\backslash t'] \]
The syntax of the lambda calculus should be familiar:

\[
 t ::= x \\
    | t t \\
    | \lambda x.t
\]

There is one key reduction rule, describing evaluation:

\[
(\lambda x.t) t' \rightarrow_\beta t[x \backslash t']
\]

The lambda calculus has many applications!
Combinatory logic

\[ c ::= x \mid c \cdot c \mid S \mid K \mid I \]

- Variables, application and three combinators;
- Crucially, there is no lambda abstraction.
Combinatory logic

\[ c := x \mid c \circ c \mid S \mid K \mid I \]

- Variables, application and three combinators;
- Crucially, there is no lambda abstraction.

Yet given the following reduction rules, this language is ‘equally expressive’ as lambda calculus:

- \( K c_1 c_2 \rightarrow c_1 \)
- \( S c_1 c_2 c_3 \rightarrow (c_1 c_3)(c_2 c_3) \)
- \( I c \rightarrow c \)

(And congruence rules for evaluating applications)
To show that these two calculi are equally expressive, we can translate from lambda terms to combinators:

\[
convert : \text{Term} \rightarrow \text{Comb}
\]

\[
\begin{align*}
convert (t_1 \, t_2) &= (convert \, t_1) \, (convert \, t_2) \\
convert \, x &= x \\
convert (\lambda x. \, t) &= \text{abs} \, x \, (convert \, t)
\end{align*}
\]
Bracket abstraction

To show that these two calculi are equally expressive, we can translate from lambda terms to combinators:

\[
\text{convert} : \text{Term} \to \text{Comb} \\
\text{convert} (t_1 t_2) = (\text{convert } t_1) (\text{convert } t_2) \\
\text{convert } x = x \\
\text{convert} (\lambda x. t) = \text{abs } x (\text{convert } t)
\]

The process of ‘bracket abstraction’ modifies the (combinatory) term corresponding to the body of a lambda to have the same reduction behaviour:

\[
\text{abs } x x = I \\
\text{abs } x c = Ky \text{ if } x \notin \text{FV}(c) \\
\text{abs } x (c c') = S (\text{abs } x c) (\text{abs } x c')
\]
Why?

• Reduction in combinatory logic no longer requires substitution.
• In the 1920's, there was a great deal of interest in ‘logical minimalism’ – finding the smallest foundations for mathematics.
• Combinators have been used as the target language for the compiling functional languages.
Why?

• Reduction in combinatory logic no longer requires substitution.
• In the 1920’s, there was a great deal of interest in ‘logical minimalism’ – finding the smallest foundations for mathematics.
• Combinators have been used as the target language for the compiling functional languages.

Today’s challenges

• How can we implement this translation?
Why?

• Reduction in combinatory logic no longer requires substitution.

• In the 1920’s, there was a great deal of interest in ‘logical minimalism’ – finding the smallest foundations for mathematics.

• Combinators have been used as the target language for the compiling functional languages.

Today’s challenges

• How can we implement this translation?

• How do we use types to ensure it is correct?
Naive implementation in Haskell

data Term = Var String
  | App Term Term
  | Lambda String Term

convert :: Lambda → SKI
convert (Var x) = Var x
convert (App t1 t2) = (convert t1) `App` (convert t2)
convert (Lam x t) = abs x (convert t)
Brackets for abstraction

```
abs :: Var → SKI → SKI
abs x c
    | not (x `elem` fv t) = K c
abs x (Var y)
    | x == y = I
abs x (App c1 c2) =
    S `App` (remove x c1)
    `App` (remove x c2)
```

But two bound variables can have the same name – yet refer to different binding sites...
De Bruijn indices (1972)

Now we no longer have named variables, but instead need to do bookkeeping with integers.

```haskell
data Term = Var Int
           | App Term Term
           | Lambda Term
```

This is still all too easy to get wrong.
De Bruijn indices (1972)

Now we no longer have named variables, but instead need to do bookkeeping with integers.

This is still all too easy to get wrong.
Well scoped (Altenkirch-Reus 1999; Bird-Paterson 1999)

```
data Term a = Var a
            | App Term Term
            | Lambda (Maybe Term)
```

```
convert :: Term a → Comb a
abst :: Comb (Maybe a) → Comb a
```

This is clearly better – but the type signature is not (yet) a specification.
Well typed terms (around 2005)

\[
data \ \text{Term} : \ \text{Ctx} \to \ \text{Type} \to \ \text{Set} \ \text{where}\\
\text{app} : \ \text{Term} \ \Gamma \ (\sigma \to \tau) \to \ \text{Term} \ \Gamma \ \sigma \to \ \text{Term} \ \Gamma \ \tau\\
\text{lam} : \ \text{Term} \ (\sigma :: \Gamma) \ \tau \to \ \text{Term} \ \Gamma \ (\sigma \to \tau)\\
\text{var} : \ \text{Ref} \ \sigma \ \Gamma \to \ \text{Term} \ \Gamma \ \sigma\\
\text{convert} : \ \text{Term} \ \Gamma \ a \to \ \text{Comb} \ \Gamma \ a\\
\text{abst} : \ \text{Comb} \ (a :: \Gamma) \ b \to \ \text{Comb} \ \Gamma \ (a \to \ b)\\
\]

We can use this to establish that the translation to combinators is type preserving....

But does it also preserve the intended semantics?
Semantics preservation

• Define an evaluator for well-typed terms;
• Define a type for combinator terms that are also indexed by their semantics;
• Show that we can define the translation to combinators:

\[
\text{convert} : (t : \text{Term } \Gamma \sigma) \rightarrow \text{Comb } \Gamma \sigma (\text{eval } t)
\]

And achieve all of the above without writing any proof terms or type coercions.
There is a well known evaluator for well typed lambda terms:

\[
\begin{align*}
\text{eval} & : \text{Term } \Gamma \sigma \rightarrow (\text{Env } \Gamma \rightarrow \text{Val } \sigma) \\
\text{eval} (\text{App } f \ x) \ env & = (\text{eval } f \ env) (\text{eval } x \ env) \\
\text{eval} (\text{Lam } t) \ env & = \lambda x \rightarrow \text{eval } t \ (\text{Cons } x \ env) \\
\text{eval} (\text{Var } i) \ env & = \text{lookup } i \ env
\end{align*}
\]
data Comb : (Γ : Ctx) → (σ : Type) → (Env Γ → σ) → Set where

S : Comb Γ ... (λ env x y z → (x z) (y z))
K : Comb Γ ... (λ env x y → x)
I : Comb Γ ... (λ env x → x)
Var : (i : Ref σ Γ) → Comb Γ σ (lookup i)
App : Comb Γ (σ → τ) f → Comb Γ σ x → Comb Γ τ (λ env → (f env) (x env))
data Comb : (Γ : Ctx) → (σ : Type) → (Env Γ → σ) → Set where
  S : Comb Γ ... (λ env x y z → (x z) (y z))
  K : Comb Γ ... (λ env x y → x)
  I : Comb Γ ... (λ env x → x)
  Var : (i : Ref σ Γ) → Comb Γ σ (lookup i)
  App : Comb Γ (σ → τ) f → Comb Γ σ x → Comb Γ τ (λ env → (f env) (x env))

Now all that we still need to do is define the desired conversion:

convert : (t : Term Γ σ) → Comb Γ σ (eval t)
Conversion to combinators

\[
\text{convert} : (t : \text{Term } \Gamma \sigma) \rightarrow \text{Comb } \Gamma \sigma \left( \text{eval } t \right)
\]

\[
\text{convert} \left( \text{App } t_1 \ t_2 \right) = \text{App} \left( \text{convert } t_1 \right) \left( \text{convert } t_2 \right)
\]

\[
\text{convert} \left( \text{Var } i \right) = \text{Var } i
\]

\[
\text{convert} \left( \text{Lam } t \right) = \text{abs} \left( \text{convert } t \right)
\]

The first two cases are easy and ‘obviously correct’.

What about the \text{abs} function?
Correct by construction bracket abstraction

\[
\text{abs} : \text{Comb} \ (\sigma :: \Gamma) \tau f \to \text{Comb} \ \Gamma \ (\sigma \to \tau) \ (\lambda \text{env } x \to f \ (\text{Cons } x \text{ env}))
\]

\[
\begin{align*}
\text{abs } S &= \text{App } K \ S \\
\text{abs } K &= \text{App } K \ K \\
\text{abs } I &= \text{App } K \ I \\
\text{abs } (\text{App } f \ x) &= \text{App } (\text{App } S \ (\text{abs } f)) \ (\text{abs } x) \\
\text{abs } (\text{Var } \text{Top}) &= I \\
\text{abs } (\text{Var } (\text{Pop } i)) &= \text{App } K \ (\text{Var } i)
\end{align*}
\]

The abs function turns the body of lambda into a combinator that behaves precisely as the desired lambda abstraction!
Why does this work?

This seems like a parlour trick – a correct by construction conversion without doing any proofs.

This only works because the direct proof appeals *only* to induction hypotheses and a lemma about \texttt{abs} - which we rolled into the correct by construction definition of the \texttt{abs} function.

As a result, we can fold the proof into the entire development.

But surely this breaks for anything more complicated?
The SKI combinators are not the only choice of combinators.

Alternatives are more careful about handling applications:

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

If \(t_1\) or \(t_2\) do not use the most recently bound variable, we can short-cut the translation and discard it immediately.

We can introduce two new combinators:

\[
\begin{align*}
B \ f \ g \ x &= (f \ x) \ g \\
C \ f \ g \ x &= f \ (g \ x)
\end{align*}
\]
The problem

We need to test which combinator (S, B, or C) to use for every application.

Using named variables, we might write:

```haskell
abs x (App t₁ t₂)
  | x `elem` (fv t₁)
    && x `elem` fv t₂  = ... use S
  | x `elem` (fv t₁)  = ... use B
  | x `elem` (fv t₂)  = ... use C
  | otherwise          = ... use K
```

But why does this preserve types? Let alone semantics...
We don’t just care about which variables *may* be in scope – but also need to know *whether* they are used or not.

In Agda, it’s better to shift to a different representation of variables:

```agda
data Term (Γ : Ctx) : Subset Γ → Type → Set where```

What are the constructors?
We don’t just care about which variables *may* be in scope – but also need to know *whether* they are used or not.

In Agda, it’s better to shift to a different representation of variables:

```haskell
data Term (Γ : Ctx) : Subset Γ → Type → Set where

App : Term Γ Δ₁ (σ → τ) → Term Γ Δ₂ σ → Term Γ (Δ₁ ∪ Δ₂) τ
Var : (i : Ref σ Γ) → Term Γ (singleton i) σ
Lam : Term (σ :: Γ) Δ τ → Term Γ (pop Δ) (σ → τ)
```
Choosing the best combinator

Using this representation, we know exactly which variables are used in both branches of the application:

\[
\text{App} : \text{Term } \Gamma \Delta_1 (\sigma \rightarrow \tau) \rightarrow \text{Term } \Gamma \Delta_2 \sigma \rightarrow \text{Term } \Gamma (\Delta_1 \cup \Delta_2) \tau
\]

By inspecting \(\Delta_1\) and \(\Delta_2\), we distinguish four cases:

- both \(\Delta_1\) and \(\Delta_2\) use the bound variable of type \(\sigma\) - use \(S\)
- \(\Delta_1\) uses the freshly bound variable of type \(\sigma\), but \(\Delta_2\) does not - use \(B\)
- \(\Delta_2\) uses the freshly bound variable of type \(\sigma\), but \(\Delta_1\) does not - use \(C\)
- neither \(\Delta_1\) nor \(\Delta_2\) use the freshly bound variable - use \(K\)

We can define a type preserving ‘optimising’ translation in the same style.
Choosing the best combinator

Using this representation, we know exactly which variables are used in both branches of the application:

\[
\text{App} : \text{Term } \Gamma \Delta_1 (\sigma \to \tau) \to \text{Term } \Gamma \Delta_2 \sigma \to \text{Term } \Gamma (\Delta_1 \cup \Delta_2) \tau
\]

By inspecting \( \Delta_1 \) and \( \Delta_2 \), we distinguish four cases:

- both \( \Delta_1 \) and \( \Delta_2 \) use the bound variable of type \( \sigma \) - use \( S \)
- \( \Delta_1 \) uses the freshly bound variable of type \( \sigma \), but \( \Delta_2 \) does not - use \( B \)
- \( \Delta_2 \) uses the freshly bound variable of type \( \sigma \), but \( \Delta_1 \) does not - use \( C \)
- neither \( \Delta_1 \) nor \( \Delta_2 \) use the freshly bound variable - use \( K \)

We can define a type preserving ‘optimising’ translation in the same style.

And establish correctness without using an (external) proof.
Conclusions

• Such correct by construction ‘proofs’ work – but it took me more than one try to find the right definitions;

• This presentation loses how these definitions are found.

• I typically found myself ensuring type preservation first, checking my definitions and starting a proof of correctness, before folding this back into the types themselves.

• The choice of variable binding makes this problem either trivial or very hard.