A correct-by-construction conversion to combinators

Types, Thorsten and Theories

Wouter Swierstra

Utrecht University

Congratulations Thorsten!

BCTCS Swansea 2006

Lambda calculus

The syntax of the lambda calculus should be familiar:

$$
\begin{aligned}
t \;:=\; & x \\
\mid\; & t\;t \\
\mid\; & \lambda x.t
\end{aligned}
$$

There is one key reduction rule, describing evaluation:

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

Lambda calculus

The syntax of the lambda calculus should be familiar:

$$
\begin{aligned}
t := {} & x \\
\mid {} & t\ t \\
\mid {} & \lambda x.t
\end{aligned}
$$

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

$$c := x \mid c\ c \mid S \mid K \mid I$$

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

Combinatory logic

$$c := x \mid c\ c \mid S \mid K \mid I$$

- 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:

- $K\, c_1\, c_2 \;\longrightarrow\; c_1$
- $S\, c_1\, c_2\, c_3 \;\longrightarrow\; (c_1\, c_3)\, (c_2\, c_3)$
- $I\, c \;\longrightarrow\; c$

(And congruence rules for evaluating applications)

Bracket abstraction

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

$$\begin{aligned}
\text{convert} &: \mathit{Term} \to \mathit{Comb} \\
\text{convert}\,(t_1\; t_2) &= (\text{convert}\, t_1)\,(\text{convert}\, t_2) \\
\text{convert}\, x &= x \\
\text{convert}\,(\lambda x.t) &= \text{abs}\, x\,(\text{convert}\, t)
\end{aligned}$$

Bracket abstraction

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

$$\begin{aligned}
\text{convert} &: \mathit{Term} \to \mathit{Comb} \\
\text{convert}\,(t_1\;t_2) &= (\text{convert}\,t_1)\,(\text{convert}\,t_2) \\
\text{convert}\,x &= x \\
\text{convert}\,(\lambda x.t) &= \text{abs}\,x\,(\text{convert}\,t)
\end{aligned}$$

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

$$\begin{aligned}
\text{abs}\,x\,x &= I \\
\text{abs}\,x\,c &= K\,y \quad \text{if } x \notin FV(c) \\
\text{abs}\,x\,(c\,c') &= S\,(\text{abs}\,x\,c)\,(\text{abs}\,x\,c')
\end{aligned}$$

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

```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)
```

Bracket abstraction

```haskell
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)

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

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

De Bruijn indices (1972)

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

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 Term : Ctx → Type → 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

convert : Term Γ a → Comb Γ a
abst : Comb (a : Γ) b → Comb Γ (a → 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 the we can define the translation to combinators:

```
convert : (t : Term Γ σ) → Comb Γ σ (eval t)
```

And achieve all of the above without writing any proof terms or type coercions.

Evaluating well typed lambda terms

There is a well known evaluator for well typed lambda terms:

```
eval : Term Γ σ → (Env Γ → Val σ)
eval (App f x) env  = (eval f env) (eval x env)
eval (Lam t) env    = λ x → eval t (Cons x env)
eval (Var i) env    = lookup i env
```

Combinatory terms - indexed by their semantics

```
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))
```

Combinatory terms - indexed by their semantics

```
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

```
convert : (t : Term Γ σ) → Comb Γ σ (eval t)
convert (App t₁ t₂)  = App (convert t₁) (convert t₂)
convert (Var i)      = Var i
convert (Lam t)      = abs (convert t)
```

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

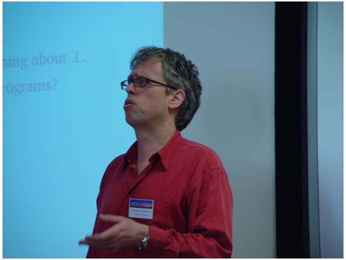
What about the `abs` function?

Correct by construction bracket abstraction

```
abs : Comb (σ ∷ Γ) τ f → Comb Γ (σ → τ) (λ env x → f (Cons x env))
abs S              = App K S
abs K              = App K K
abs I              = App K I
abs (App f x)      = App (App S (abs f)) (abs x)
abs (Var Top)      = I
abs (Var (Pop i))  = App K (Var i)
```

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 `abs` - which we rolled into the correct by construction definition of the `abs` function.

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

But surely this breaks for anything more complicated?

Beyond SKI

The SKI combinators are not the only choice of combinators.

Alternatives are more careful about handling applications:

```
abs (App t₁ t₂)   = App (App S (abs t₁)) (abs t₂)
```

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:

```
B f g x = (f x) g
C f g x = f (g x)
```

The problem

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

Using named variables, we might write:

```
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...

How to handle bound variables?

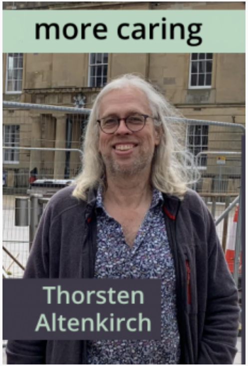
We don't just care about which variables *may* be in scope – but also need to know *whether* they are used or not.

We need a **more caring** treatment of bound variables…

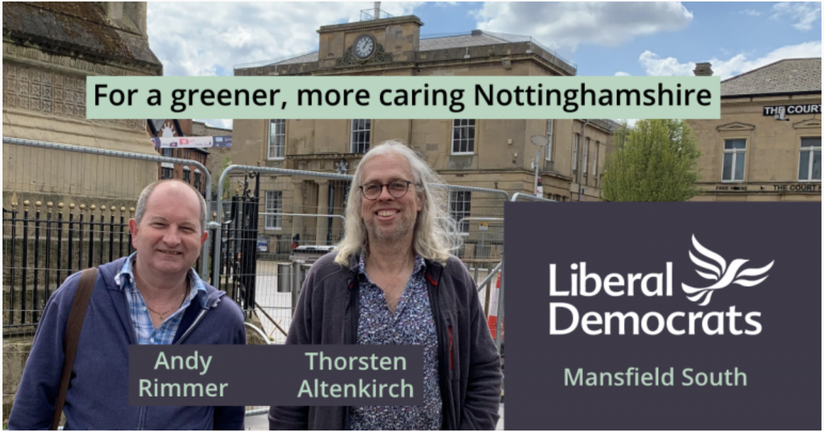
How to handle bound variables?

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

We need a **more caring** treatment of bound variables...



Thorsten – the liberal democrat



co-de Bruijn – a more caring treatment of bound variables

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

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

Each term tracks the context of all variables in scope Γ and the variables that have been used (a subset of Γ).

In each application, we can then choose to introduce an S, B, C, or K combinator depending on if a variable is used in both, the left, the right or neither subterm respectively.

With a bit of cunning, we can adapt the translation accordingly.

Reviewer #2

The paper contains a completely mechanized proof of correctness of two different translation of lambda calculus to combinatory logic. However, to enable the demonstrated proof technique, this result is not as general as the original result by Curry and Feys: the paper includes the simply-typed lambda calculus only, not the full untyped lambda calculus. This should be clarified and motivated.

Reviewer #2

```
The paper contains a completely mechanized proof of correctness of two
different translation of lambda calculus to combinatory logic. However, to
enable the demonstrated proof technique, this result is not as general as
the original result by Curry and Feys: the paper includes the simply-typed
lambda calculus only, not the full untyped lambda calculus. This should be
clarified and motivated.
```

One important lesson I learned from Thorsten:

But it’s the **typed terms** that we **care** about! Any meaning associated with untyped terms is entirely coincidental.

Thank you for everything Thorsten!

FP Lab Away Day – 2007