
# A correct by construction conversion to combinators

A JFP Functional pearl

Wouter Swierstra

Utrecht University

# Logical minimalism

- In the 1920's and 1930's, Moses Schönfinkel and Haskell Curry searched for languages *without variable binding*, yet still capable of describing all of mathematics.
- *Combinatory logic* is a simpler alternative to the lambda calculus, yet capable of expressing the same computations.

# This paper

- How to define the translation from lambda calculus to combinatory logic in Agda...

# This paper

- How to define the translation from lambda calculus to combinatory logic in Agda…
- where the *type* of the translation guarantees that it is type preserving…

# This paper

- How to define the translation from lambda calculus to combinatory logic in Agda…
- where the *type* of the translation guarantees that it is type preserving…
- and *correct*! (In that it preserves the *semantics* of our lambda terms.)

# This paper

- How to define the translation from lambda calculus to combinatory logic in Agda…
- where the *type* of the translation guarantees that it is type preserving…
- and *correct*! (In that it preserves the *semantics* of our lambda terms.)

Hence, this yields a correct by construction conversion to combinators.

# Combinatory logic

A first order language with variables:

$$t \quad ::= \quad x \;\mid\; t\ t \;\mid\; \mathsf{S} \;\mid\; \mathsf{K} \;\mid\; \mathsf{I}$$

Instead of beta reduction or substitution, we have three simple reduction rules:

$$\begin{aligned} \mathsf{I}\ x &\longrightarrow x \\ \mathsf{K}\ x\ y &\longrightarrow x \\ \mathsf{S}\ x\ y\ z &\longrightarrow (x\ z)\ (y\ z) \end{aligned}$$

(Aside: used as a target language for compilers for functional languages.)

# Combinatory logic

A first order language with variables:

$$t \quad ::= \quad x \;\mid\; t\ t \;\mid\; \mathsf{S} \;\mid\; \mathsf{K} \;\mid\; \mathsf{I}$$

Instead of beta reduction or substitution, we have three simple reduction rules:

$$\mathsf{I}\ x \longrightarrow x$$

$$\mathsf{K}\ x\ y \longrightarrow x$$

$$\mathsf{S}\ x\ y\ z \longrightarrow (x\ z)\ (y\ z)$$

(Aside: used as a target language for compilers for functional languages.)

No lambdas or variable binding – yet somehow equivalent?

# Towards the simply typed lambda calculus

$$\sigma, \tau \quad ::= \quad \iota \quad | \quad \sigma \to \tau$$

# Towards the simply typed lambda calculus

$$\sigma, \tau \quad ::= \quad \iota \quad | \quad \sigma \to \tau$$

Syntax of types:

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

# Towards the simply typed lambda calculus

$$\sigma, \tau \quad ::= \quad \iota \quad | \quad \sigma \to \tau$$

Syntax of types:

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

Semantics of types:

```
⟦_⟧ : U → Set
⟦ ι ⟧          = A
⟦ σ → τ ⟧      = ⟦ σ ⟧ → ⟦ τ ⟧
```

Here we have taken the base type to some type `A`, but the choice does not really matter.

# From typing rules to intrinsically typed syntax

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}$$

$$\frac{\Gamma \vdash t_1 : \sigma \to \tau \quad \Gamma \vdash t_2 : \sigma}{\Gamma \vdash t_1\ t_2 : \tau}$$

$$\frac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash \lambda x\ t : \sigma \to \tau}$$

```
data Term : Ctx → U → Set where
    var  : σ ∈ Γ → Term Γ σ
    app  : Term Γ (σ ⇒ τ) → Term Γ σ → Term Γ τ
    lam  : Term (σ :: Γ) τ → Term Γ (σ ⇒ τ)

Ctx = List U
```

# Finally, a tagless evaluator

With all these types and definitions, we complete a ‘tagless’ evaluator:

```
eval : Term Γ σ → Env Γ → ⟦ σ ⟧
eval (app t₁ t₂) env  = (eval t₁ env) (eval t₂ env)
eval (lam t) env      = λ x → eval t (cons x env)
eval (var i) env      = lookup env i
```

Here our environments are a heterogeneous list of values, recording the value associated with each variable in scope.

# Finally, a tagless evaluator

With all these types and definitions, we complete a ‘tagless’ evaluator:

```
eval : Term Γ σ → Env Γ → ⟦ σ ⟧
eval (app t₁ t₂) env  = (eval t₁ env) (eval t₂ env)
eval (lam t) env      = λ x → eval t (cons x env)
eval (var i) env      = lookup env i
```

Here our environments are a heterogeneous list of values, recording the value associated with each variable in scope.

It is amazing that this works at all!

# Combinatory logic

$$t \quad ::= \quad x \;\mid\; t\,t \;\mid\; \mathsf{S} \mid \mathsf{K} \mid \mathsf{I}$$

What should the type of our combinatory terms be?

```
data CL : Set
  var : String → CL
  app : CL → CL → CL
  S K I : CL
```

# Combinatory logic

$$t \quad ::= \quad x \quad | \quad t\,t \quad | \quad \mathsf{S} \quad | \quad \mathsf{K} \quad | \quad \mathsf{I}$$

What should the type of our combinatory terms be?

```
data CL : Set
  var : String → CL
  app : CL → CL → CL
  S K I : CL
```

This representation is not wrong – but it doesn't tell us anything about the types involved!

```
translate : Term Γ σ → CL
```

In particular, there is no guarantee that this function maps well typed terms to well typed combinators.

# Type preserving translation

To ensure types preservation, we define a datatype for well typed terms in combinatory logic:

```
data CL (Γ : Ctx) : U → Set where
  var : σ ∈ Γ → CL Γ σ
  app : CL Γ (σ ⇒ τ) → CL Γ σ → CL Γ τ
  I   : CL Γ (σ ⇒ σ)
  K   : CL Γ (σ ⇒ τ ⇒ σ)
  S   : CL Γ ((σ ⇒ τ ⇒ ρ) ⇒ (σ ⇒ τ) ⇒ (σ ⇒ ρ))
```

# Type preserving translation

To ensure types preservation, we define a datatype for well typed terms in combinatory logic:

```
data CL (Γ : Ctx) : U → Set where
  var : σ ∈ Γ → CL Γ σ
  app : CL Γ (σ ⇒ τ) → CL Γ σ → CL Γ τ
  I   : CL Γ (σ ⇒ σ)
  K   : CL Γ (σ ⇒ τ ⇒ σ)
  S   : CL Γ ((σ ⇒ τ ⇒ ρ) ⇒ (σ ⇒ τ) ⇒ (σ ⇒ ρ))
```

Now it is clear that the translation preserves types:

```
translate : Term Γ σ → CL Γ σ
```

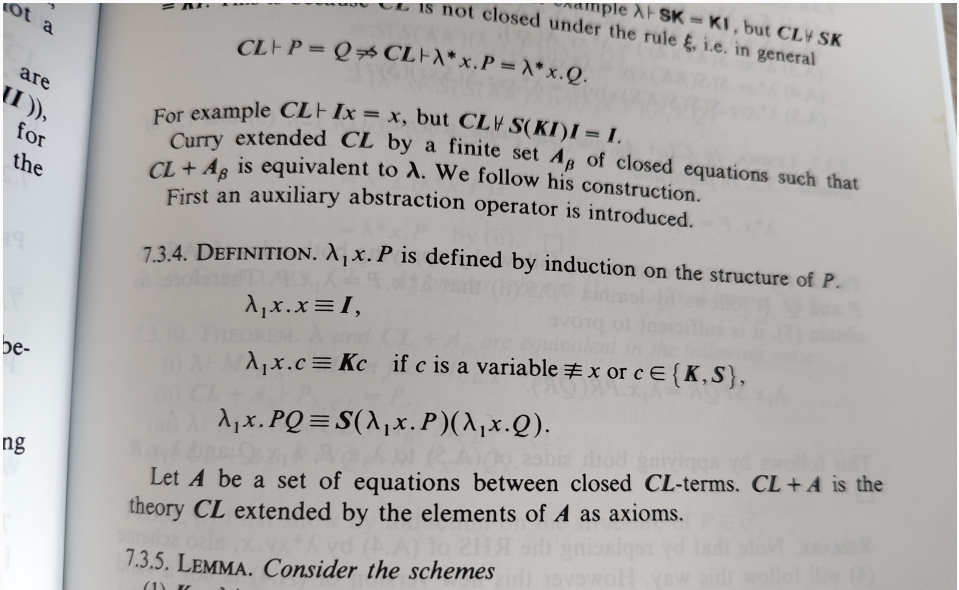
# Translation to combinatory logic

Most of the translation is now trivial:

```
translate : Term Γ σ → CL Γ σ
translate (app t₁ t₂) = app (translate t₁) (translate t₂)
translate (var i)     = var i
translate (lam t)     = abs (translate t)
  where
  abs : CL (σ :: Γ) τ → CL Γ (σ ⇒ τ)
```

The only real work is done by the *bracket abstraction* function, `abs`.

# Bracket abstraction - the theory



is not closed under the rule $\xi$, i.e. in general

$$CL \vdash P = Q \not\Rightarrow CL \vdash \lambda^* x.P = \lambda^* x.Q.$$

For example $CL \vdash Ix = x$, but $CL \nvdash S(KI)I = I$.

Curry extended $CL$ by a finite set $A_\beta$ of closed equations such that $CL + A_\beta$ is equivalent to $\lambda$. We follow his construction.

First an auxiliary abstraction operator is introduced.

7.3.4. DEFINITION. $\lambda_1 x.P$ is defined by induction on the structure of $P$.

$$\lambda_1 x.x \equiv I,$$

$$\lambda_1 x.c \equiv Kc \quad \text{if } c \text{ is a variable} \not\equiv x \text{ or } c \in \{K, S\},$$

$$\lambda_1 x.PQ \equiv S(\lambda_1 x.P)(\lambda_1 x.Q).$$

Let $A$ be a set of equations between closed $CL$-terms. $CL + A$ is the theory $CL$ extended by the elements of $A$ as axioms.

7.3.5. LEMMA. *Consider the schemes*

# Bracket abstraction – the implementation

```
-- the 'same' type as the lam constructor
abs : CL (σ :: Γ) τ → CL Γ (σ ⇒ τ)
-- the most recently bound variable is replaced by I
abs (var top)      = I
-- other variables/combinators are wrapped by K
abs (var (pop j)) = app K (var j)
abs S              = app K S
abs K              = app K K
abs I              = app K I
-- application introduces S and recurses
abs (app t₁ t₂)    = app (app S (abs t₁)) (abs t₂)
```

All the types go through easily enough...

# Bracket abstraction – the implementation

```
-- the 'same' type as the lam constructor
abs : CL (σ :: Γ) τ → CL Γ (σ ⇒ τ)
-- the most recently bound variable is replaced by I
abs (var top)      = I
-- other variables/combinators are wrapped by K
abs (var (pop j)) = app K (var j)
abs S              = app K S
abs K              = app K K
abs I              = app K I
-- application introduces S and recurses
abs (app t₁ t₂)    = app (app S (abs t₁)) (abs t₂)
```

All the types go through easily enough…

*What about proving correctness?*

# Correctness – take 1

We define an evaluator for combinatory terms:

```
evalCL : CL Γ σ → Env Γ → ⟦ σ ⟧
evalCL K env          = λ x y → x
evalCL I env          = λ x → x
evalCL (app f e) env = (evalCL f env) (evalCL e env)
...
```

Now we only need to prove the obvious correctness statement:

```
correctness : (t : Term Γ σ) (env : Env Γ) → evalCL (translate t) env ≡ eval t env
```

# Correctness – take 1

The proof itself is not very interesting – it requires one additional lemma:

```
abs-correct : (t : CL (σ :: Γ) τ) (env : Env Γ) (v : ⟦ σ ⟧) →
  evalCL (abs t) env v ≡ evalCL t (v :: env)
```

This lemma makes precise that ‘bracket abstraction behaves like lambda abstraction’.

The proof follows from immediate induction.

# Correctness – take 1

The proof itself is not very interesting – it requires one additional lemma:

```
abs-correct : (t : CL (σ :: Γ) τ) (env : Env Γ) (v : ⟦ σ ⟧) →
  evalCL (abs t) env v ≡ evalCL t (v :: env)
```

This lemma makes precise that ‘bracket abstraction behaves like lambda abstraction’.

The proof follows from immediate induction.

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

# Correctness - take 2

Let's redefine our datatype for terms in combinatory logic.

Now they not only carry their *type* but also their (dynamic) semantics, i.e., the result of evaluation:

```
data CL : (Γ : Ctx) → (σ : U) → (Env Γ → ⟦ σ ⟧) → Set where
  K   : CL Γ (σ ⇒ (τ ⇒ σ)) (λ env → λ x y → x)
  I   : CL Γ (σ ⇒ σ) (λ env → λ x → x)
  app : CL Γ (σ ⇒ τ) f → CL Γ σ t → CL Γ τ (f t)
  ...
```

# Correctness - take 2

Let's redefine our datatype for terms in combinatory logic.

Now they not only carry their *type* but also their (dynamic) semantics, i.e., the result of evaluation:

```
data CL : (Γ : Ctx) → (σ : U) → (Env Γ → ⟦ σ ⟧) → Set where
  K    : CL Γ (σ ⇒ (τ ⇒ σ)) (λ env → λ x y → x)
  I    : CL Γ (σ ⇒ σ) (λ env → λ x → x)
  app  : CL Γ (σ ⇒ τ) f → CL Γ σ t → CL Γ τ (f t)
  ...
```

Now we can *specify* the translation to combinatory logic that is correct by construction:

```
translate : (t : Term Γ σ) → CL Γ σ (eval t)
```

# Correctness - take 2

Let's redefine our datatype for terms in combinatory logic.

Now they not only carry their *type* but also their (dynamic) semantics, i.e., the result of evaluation:

```
data CL : (Γ : Ctx) → (σ : U) → (Env Γ → ⟦ σ ⟧) → Set where
  K    : CL Γ (σ ⇒ (τ ⇒ σ)) (λ env → λ x y → x)
  I    : CL Γ (σ ⇒ σ) (λ env → λ x → x)
  app  : CL Γ (σ ⇒ τ) f → CL Γ σ t → CL Γ τ (f t)
  ...
```

Now we can *specify* the translation to combinatory logic that is correct by construction:

```
translate : (t : Term Γ σ) → CL Γ σ (eval t)
```

How is this function defined?

# Translation, revisited

The translation remains exactly the same – it is only the types that change!

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

# Translation, revisited

The translation remains exactly the same – it is only the types that change!

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

Of course, we still need to (re)define the bracket abstraction function…

# Translation, revisited

The translation remains exactly the same – it is only the types that change!

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

Of course, we still need to (re)define the bracket abstraction function…

But once again, it is only the *type* that changes!

# Translation, optimized

Adding new combinators helps *reduce* the number of evaluation steps:

$$t \quad ::= \quad x \;\mid\; t\,t \;\mid\; \mathsf{S} \;\mid\; \mathsf{K} \;\mid\; \mathsf{I} \;\mid\; \mathsf{B} \;\mid\; \mathsf{C}$$

Unlike S that distributes its arguments to *both* subterms, B and C pass the argument to one of the two subterms:

$$\mathsf{B}\ x\ y\ z \longrightarrow x\ (y\ z)$$
$$\mathsf{C}\ x\ y\ z \longrightarrow (x\ z)\ y$$

# Translation, optimized

Adding new combinators helps *reduce* the number of evaluation steps:

$t$ ::= $x$ | $t\,t$ | S | K | I | B | C

Unlike S that distributes its arguments to *both* subterms, B and C pass the argument to one of the two subterms:

$$\mathsf{B}\ x\ y\ z \longrightarrow x\ (y\ z)$$
$$\mathsf{C}\ x\ y\ z \longrightarrow (x\ z)\ y$$

How shall we redefine the translation to use B and C (rather than S) whenever possible?

# Correct by construction optimization

I wrote a correct by construction translation, using co-De Bruijn indices to track which variables are used. It works!

# Correct by construction optimization

I wrote a correct by construction translation, using co-De Bruijn indices to track which variables are used. It works!

But Ralf Hinze suggested a much simpler solution...

# Correct by construction optimization

I wrote a correct by construction translation, using co-De Bruijn indices to track which variables are used. It works!

But Ralf Hinze suggested a much simpler solution...

Define a smart constructor for application nodes:

```
sapp : CL Γ (σ ⇒ τ ⇒ ρ) f → CL Γ (σ ⇒ τ) x →
       CL Γ (σ ⇒ ρ) (λ y → (f y) (x y))
```

that checks which combinator to use to route variables to the right place.

# Correct by construction

This seems like a ‘parlour trick’...

# Correct by construction

This seems like a ‘parlour trick’...

But the meta-properties that makes this possible are:

- the translation is defined by simple induction, with one auxiliary function;

# Correct by construction

This seems like a ‘parlour trick’...

But the meta-properties that makes this possible are:

- the translation is defined by simple induction, with one auxiliary function;
- the correctness proof proceeds by immediate induction, using one additional lemma about this auxiliary function;

# Correct by construction

This seems like a ‘parlour trick’...

But the meta-properties that makes this possible are:

- the translation is defined by simple induction, with one auxiliary function;
- the correctness proof proceeds by immediate induction, using one additional lemma about this auxiliary function;
- the structure of the translation and correctness proof coincide *exactly*;

# Correct by construction

This seems like a ‘parlour trick’…

But the meta-properties that makes this possible are:

- the translation is defined by simple induction, with one auxiliary function;
- the correctness proof proceeds by immediate induction, using one additional lemma about this auxiliary function;
- the structure of the translation and correctness proof coincide *exactly*;

*Whenever* these properties hold, you can define and verify your function at the same time – leading to definitions that are *correct by construction*.

**Questions?**