Combinatory logic and lambda calculus are equal, algebraically

Ambrus Kaposi

Eötvös Loránd University, Budapest

j.w.w. Thorsten Altenkirch, Artjoms Šinkarovs, Tamás Végh

FSCD, Roma, 3 July 2023

Combinatory logic and lambda calculus

Moses Schönfinkel

- Combinatory logic: Schönfinkel 1920

Combinatory logic and lambda calculus

Moses Schönfinkel

- Combinatory logic: Schönfinkel 1920
- Lambda calculus: Church 1928

Combinatory logic and lambda calculus

Moses Schönfinkel

- Combinatory logic: Schönfinkel 1920
- Lambda calculus: Church 1928
- Originally developed for logic

Combinatory logic and lambda calculus

Moses Schönfinkel

- Combinatory logic: Schönfinkel 1920 (Hilbert style proof theory for propositional logic)
- Lambda calculus: Church 1928 (Gentzen natural deduction)
- Originally developed for logic

Combinatory logic and lambda calculus

Moses Schönfinkel

- Combinatory logic: Schönfinkel 1920 (Hilbert style proof theory for propositional logic)
- Lambda calculus: Church 1928 (Gentzen natural deduction)
- Originally developed for logic
- They are equivalent (Rosser 1935)

Combinatory logic and lambda calculus

Moses Schönfinkel

- Combinatory logic: Schönfinkel 1920 (Hilbert style proof theory for propositional logic)
- Lambda calculus: Church 1928 (Gentzen natural deduction)
- Originally developed for logic
- They are equivalent (Rosser 1935)
- Spin-off from dependently typed combinatory logic

Traditional presentation

Combinatory logic	Lambda calculus
`t ::= K | S | t·t'`	`t ::= x | λx.t | t·t'`

Traditional presentation

Combinatory logic

```
Tm    : Set
K     : Tm
S     : Tm
_·_   : Tm → Tm → Tm
```

Lambda calculus

```
Tm      : Set
var     : ℕ → Tm
lam     : Tm → Tm
_·_     : Tm → Tm → Tm
```

Traditional presentation

Combinatory logic

```
Tm   : Set
K    : Tm
S    : Tm
_·_  : Tm → Tm → Tm

_∈_  : Tm→Ty→Prop
tyK  : K ∈ A⇒B⇒A
...
```

Lambda calculus

```
Tm     : Set
var    : ℕ → Tm
lam    : Tm → Tm
_·_    : Tm → Tm → Tm

_⊢_∈_ : Con→Tm→Ty→Prop
tylam : Γ,A ⊢ t ∈ B →
        Γ ⊢ t ∈ A⇒B
...
```

Intrinsic presentation

Combinatory logic

```
Tm    : Ty → Set
K     : Tm (A⇒B⇒A)
S     : Tm ((A⇒B⇒C)⇒
              (A⇒B)⇒A⇒C)
_·_   : Tm (A⇒B) →
        Tm A → Tm B
```

Lambda calculus

```
Tm     : Con → Ty → Set
zero   : Tm (Γ,A) A
suc    : Tm Γ A →
         Tm (Γ,B) A
_·_    : Tm Γ (A⇒B)→
         Tm Γ A → Tm Γ B
lam    : Tm (Γ,B) A →
         Tm Γ (A⇒B)
```

Intrinsic presentation

Combinatory logic

```
Tm    : Ty → Set
K     : Tm (A⇒B⇒A)
S     : Tm ((A⇒B⇒C)⇒
              (A⇒B)⇒A⇒C)
_·_   : Tm (A⇒B) →
        Tm A → Tm B
```

Lambda calculus

```
Tm     : Con → Ty → Set
zero   : Tm (Γ,A) A
suc    : Tm Γ A →
         Tm (Γ,B) A
_·_    : Tm Γ (A⇒B)→
         Tm Γ A → Tm Γ B
lam    : Tm (Γ,B) A →
         Tm Γ (A⇒B)
```

```
Parameterised by Ty  : Set
                 _⇒_ : Ty → Ty → Ty
```

Intrinsic presentation

Combinatory logic

```
Tm    : Ty → Set
K     : Tm (A⇒B⇒A)
S     : Tm ((A⇒B⇒C)⇒
             (A⇒B)⇒A⇒C)
_·_   : Tm (A⇒B) →
        Tm A → Tm B
```

Lambda calculus

```
Tm     : Con → Ty → Set
zero   : Tm (Γ,A) A
suc    : Tm Γ A →
         Tm (Γ,B) A
_·_    : Tm Γ (A⇒B)→
         Tm Γ A → Tm Γ B
lam    : Tm (Γ,B) A →
         Tm Γ (A⇒B)
```

```
Parameterised by Ty  : Set
                 _⇒_ : Ty → Ty → Ty
Untyped is a special case.
```

From lambda terms to combinators

```
t·u := t·u
K   := λ x y . x
S   := λ f g x . f·x·(g·x)
```

From lambda terms to combinators

```
t·u := t·u
K   := lam (lam 1)
S   := lam (lam (lam (2·0·(1·0))))
```

From combinators to lambda terms

We extend the language of combinators with variables:

```
Tm   : Con → Ty → Set
```

From combinators to lambda terms

We extend the language of combinators with variables:

```
Tm   : Con → Ty → Set
zero : Tm (Γ,A) A
```

From combinators to lambda terms

We extend the language of combinators with variables:

```
Tm   : Con → Ty → Set
zero : Tm (Γ,A) A
suc  : Tm Γ A → Tm (Γ,B) A
```

From combinators to lambda terms

We extend the language of combinators with variables:

```
Tm   : Con → Ty → Set
zero : Tm (Γ,A) A
suc  : Tm Γ A → Tm (Γ,B) A
K    : Tm Γ (A⇒B⇒A)
S    : Tm Γ ((A⇒B⇒C)⇒(A⇒B)⇒A⇒C)
_·_  : Tm Γ (A⇒B) → Tm Γ A → Tm Γ B
```

From combinators to lambda terms

```
lam : Tm (Γ,A) B → Tm Γ (A⇒B)
```

From combinators to lambda terms

```
lam : Tm (Γ,A) B → Tm Γ (A⇒B)
lam zero    :=
lam (suc x) :=
lam K       :=
lam S       :=
lam (t·u)   :=
```

From combinators to lambda terms

```
lam : Tm (Γ,A) B → Tm Γ (A⇒B)
lam zero     := I
lam (suc x) :=
lam K        :=
lam S        :=
lam (t·u)    :=
```

From combinators to lambda terms

```
lam : Tm (Γ,A) B → Tm Γ (A⇒B)
lam zero     := S·K·K
lam (suc x) :=
lam K        :=
lam S        :=
lam (t·u)    :=
```

From combinators to lambda terms

```
lam : Tm (Γ,A) B → Tm Γ (A⇒B)
lam zero     := S·K·K
lam (suc x)  := K·x
lam K        :=
lam S        :=
lam (t·u)    :=
```

From combinators to lambda terms

```
lam : Tm (Γ,A) B → Tm Γ (A⇒B)
lam zero     := S·K·K
lam (suc x)  := K·x
lam K        := K·K
lam S        :=
lam (t·u)    :=
```

From combinators to lambda terms

```
lam : Tm (Γ,A) B → Tm Γ (A⇒B)
lam zero     := S·K·K
lam (suc x) := K·x
lam K        := K·K
lam S        := K·S
lam (t·u)    :=
```

From combinators to lambda terms

```
lam : Tm (Γ,A) B → Tm Γ (A⇒B)
lam zero    := S·K·K
lam (suc x) := K·x
lam K       := K·K
lam S       := K·S
lam (t·u)   := S·lam t·lam u
```

We add equations

```
Tm    : Ty → Set
K     : Tm (A⇒B⇒A)
S     : Tm ((A⇒B⇒C)⇒(A⇒B)⇒A⇒C)
_·_   : Tm (A⇒B) → Tm A → Tm B
Kβ    : K·u·v = u
Sβ    : S·f·g·u = f·u·(g·u)
```

We add equations

```
Tm   : Ty → Set
K    : Tm (A⇒B⇒A)
S    : Tm ((A⇒B⇒C)⇒(A⇒B)⇒A⇒C)
_·_  : Tm (A⇒B) → Tm A → Tm B
Kβ   : K·u·v = u
Sβ   : S·f·g·u = f·u·(g·u)
```

Typed combinatory algebra.

We add equations

```
Tm    : Ty → Set
K     : Tm (A⇒B⇒A)
S     : Tm ((A⇒B⇒C)⇒(A⇒B)⇒A⇒C)
_·_   : Tm (A⇒B) → Tm A → Tm B
Kβ    : K·u·v = u
Sβ    : S·f·g·u = f·u·(g·u)
```

Typed combinatory algebra.
The (quotiented) syntax is the initial algebra.

We add equations: calculus with variables

```
Tm   : Con → Ty → Set
zero
suc
K
S
_·_
Kβ   : K·u·v = u
Sβ   : S·f·g·u = f·u·(g·u)
sucK : suc K = K
sucS : suc S = S
suc· : suc (t·u) = suc t·suc u
```

From combinators to lambda terms

```
lam : Tm (Γ,A) B → Tm Γ (A⇒B)
lam zero     := S·K·K
lam (suc x)  := K·x
lam K        := K·K
lam S        := K·S
lam (t·u)    := S·lam t·lam u
lam Kβ
lam Sβ
lam sucK
lam sucS
lam suc·
```

From combinators to lambda terms

```
lam : Tm (Γ,A) B → Tm Γ (A⇒B)
lam zero     := S·K·K
lam (suc x) := K·x
lam K        := K·K
lam S        := K·S
lam (t·u)    := S·lam t·lam u
lam Kβ       : lam (K·u·v) = lam u
lam Sβ
lam sucK
lam sucS
lam suc·
```

From combinators to lambda terms

```
lam : Tm (Γ,A) B → Tm Γ (A⇒B)
lam zero     := S·K·K
lam (suc x)  := K·x
lam K        := K·K
lam S        := K·S
lam (t·u)    := S·lam t·lam u
lam Kβ       : S·lam (K·u)·lam v = lam u
lam Sβ
lam sucK
lam sucS
lam suc·
```

From combinators to lambda terms

```
lam : Tm (Γ,A) B → Tm Γ (A⇒B)
lam zero    := S·K·K
lam (suc x) := K·x
lam K       := K·K
lam S       := K·S
lam (t·u)   := S·lam t·lam u
lam Kβ      : S·(S·lam K·lam u)·lam v = lam u
lam Sβ
lam sucK
lam sucS
lam suc·
```

From combinators to lambda terms

```
lam : Tm (Γ,A) B → Tm Γ (A⇒B)
lam zero     := S·K·K
lam (suc x)  := K·x
lam K        := K·K
lam S        := K·S
lam (t·u)    := S·lam t·lam u
lam Kβ       : S·(S·(K·K)·lam u)·lam v = lam u
lam Sβ
lam sucK
lam sucS
lam suc·
```

From combinators to lambda terms

```
lam : Tm (Γ,A) B → Tm Γ (A⇒B)
lam zero     := S·K·K
lam (suc x) := K·x
lam K        := K·K
lam S        := K·S
lam (t·u)    := S·lam t·lam u
lam Kβ       : S·(S·(K·K)·lam u)·lam v = lam u
lam Sβ
lam sucK
lam sucS
lam suc·

We add a new equation to the theory:
lamKβ : S·(S·(K·K)·t)·t' = t
```

From combinators to lambda terms

```
lam : Tm (Γ,A) B → Tm Γ (A⇒B)
lam zero    := S·K·K
lam (suc x) := K·x
lam K       := K·K
lam S       := K·S
lam (t·u)   := S·lam t·lam u
lam Kβ      : S·(S·(K·K)·lam u)·lam v = lam u
lam Sβ
lam sucK
lam sucS
lam suc·
lam lamKβ   : lam (S·(S·(K·K)·t)·t') = lam t
```

From combinators to lambda terms

```
lam : Tm (Γ,A) B → Tm Γ (A⇒B)
lam zero    := S·K·K
lam (suc x) := K·x
lam K       := K·K
lam S       := K·S
lam (t·u)   := S·lam t·lam u
lam Kβ      : S·(S·(K·K)·lam u)·lam v = lam u
lam Sβ
lam sucK
lam sucS
lam suc·

Point free version:
lamKβ : λ t t' . S·(S·(K·K)·t)·t' = λ t t' . t
```

From combinators to lambda terms

```
lam : Tm (Γ,A) B → Tm Γ (A⇒B)
lam zero    := S·K·K
lam (suc x) := K·x
lam K       := K·K
lam S       := K·S
lam (t·u)   := S·lam t·lam u
lam Kβ      : S·(S·(K·K)·lam u)·lam v = lam u
lam Sβ
lam sucK
lam sucS
lam suc·

The λs can be removed:
lamKβ : S·(K·S)·(S·(K·K)) = K
```

From combinators to lambda terms

```
lam : Tm (Γ,A) B → Tm Γ (A⇒B)
lam zero     := S·K·K
lam (suc x)  := K·x
lam K        := K·K
lam S        := K·S
lam (t·u)    := S·lam t·lam u
lam Kβ       : S·(S·(K·K)·lam u)·lam v = lam u
lam Sβ
lam sucK
lam sucS
lam suc·

It only holds in the empty context:
lamKβ : S{◇}·(K{◇}·S{◇})·(S{◇}·(K{◇}·K{◇})) = K{◇}
```

From combinators to lambda terms

```
lam : Tm (Γ,A) B → Tm Γ (A⇒B)
lam zero    := S·K·K
lam (suc x) := K·x
lam K       := K·K
lam S       := K·S
lam (t·u)   := S·lam t·lam u
lam Kβ      : S·(S·(K·K)·lam u)·lam v = lam u
lam Sβ
lam sucK
lam sucS
lam suc·
lam lamKβ   holds vacuously
```

From combinators to lambda terms

```
lam : Tm (Γ,A) B → Tm Γ (A⇒B)
lam zero    := S·K·K
lam (suc x) := K·x
lam K       := K·K
lam S       := K·S
lam (t·u)   := S·lam t·lam u
lam Kβ      := from lamKβ (NEW)
lam Sβ      := from lamSβ (NEW)
lam sucK    := refl
lam sucS    := refl
lam suc·    := from lamsuc· (NEW)
lam lamKβ   holds vacuously
lam lamSβ   holds vacuously
lam lamsuc· holds vacuously
```

Three theories

- C: combinatory logic + 3 new equations needed to define lam
- C-var: combinatory logic with variables + 3 new equations
- L: lambda calculus

Three theories

- C: combinatory logic $+$ 3 new equations $+$ η
  (translated point-free closed version of `t = λx.t·x`)
- C-var: combinatory logic with variables $+$ 3 new equations $+$ η
- L: lambda calculus with β and η

Three theories

- C: combinatory logic $+$ 3 new equations $+ \eta$
  (translated point-free closed version of `t = λx.t·x`)
- C-var: combinatory logic with variables $+$ 3 new equations $+ \eta$
- L: lambda calculus with β and η

$$\mathsf{Tm}_{\mathsf{C}}\,A \cong \mathsf{Tm}_{\mathsf{C\text{-}var}}\,\diamond\,A \qquad\qquad \mathsf{Tm}_{\mathsf{C\text{-}var}}\,\Gamma\,A \cong \mathsf{Tm}_{\mathsf{L}}\,\Gamma\,A$$

Three theories

- C: combinatory logic $+$ 3 new equations $+ \eta$
- C-var: combinatory logic with variables $+$ 3 new equations $+ \eta$
- L: lambda calculus with β and η

$$\mathsf{Tm}_{\mathsf{C}}\, A \cong \mathsf{Tm}_{\mathsf{C\text{-}var}} \diamond A \cong \mathsf{Tm}_{\mathsf{L}} \diamond A$$

Three theories

- C: combinatory logic + 3 new equations + η
- C-var: combinatory logic with variables + 3 new equations + η
- L: lambda calculus with β and η

$$\mathsf{Tm}_{\mathsf{C}}\,(\Gamma \Rightarrow^* A) \cong \mathsf{Tm}_{\mathsf{C\text{-}var}} \diamond (\Gamma \Rightarrow^* A) \cong \mathsf{Tm}_{\mathsf{L}} \diamond (\Gamma \Rightarrow^* A) \cong \mathsf{Tm}_{\mathsf{L}}\,\Gamma\,A$$

Three theories

- C: combinatory logic $+$ 3 new equations $+ \eta$

- C-var: combinatory logic with variables $+$ 3 new equations $+ \eta$
- L: lambda calculus with β and η

$$\mathsf{Tm}_{\mathsf{C}}\,(\Gamma \Rightarrow^* A) = \mathsf{Tm}_{\mathsf{C\text{-}var}} \diamond (\Gamma \Rightarrow^* A) = \mathsf{Tm}_{\mathsf{L}} \diamond (\Gamma \Rightarrow^* A) = \mathsf{Tm}_{\mathsf{L}}\,\Gamma\,A$$

Three theories

- C: combinatory logic $+$ 3 new equations $+ \eta$
- C-var: combinatory logic with variables $+$ 3 new equations $+ \eta$
- L: lambda calculus with β and η

$$\mathsf{Tm}_{\mathsf{C}}\,(\Gamma \Rightarrow^* A) = \mathsf{Tm}_{\mathsf{C\text{-}var}} \diamond (\Gamma \Rightarrow^* A) = \mathsf{Tm}_{\mathsf{L}} \diamond (\Gamma \Rightarrow^* A) = \mathsf{Tm}_{\mathsf{L}}\,\Gamma\,A$$

Open (?) problems in the algebraic setting.

Three theories

- C: combinatory logic + 3 new equations + η
- C-var: combinatory logic with variables + 3 new equations + η
- L: lambda calculus with β and η

$\mathsf{Tm}_{\mathsf{C}}\,(\Gamma \Rightarrow^* A) = \mathsf{Tm}_{\mathsf{C\text{-}var}} \diamond (\Gamma \Rightarrow^* A) = \mathsf{Tm}_{\mathsf{L}} \diamond (\Gamma \Rightarrow^* A) = \mathsf{Tm}_{\mathsf{L}}\,\Gamma\,A$

Open (?) problems in the algebraic setting. What is...

- ...the combinatory equivalent of L without η?

Three theories

- C: combinatory logic $+$ 3 new equations $+ \eta$

- C-var: combinatory logic with variables $+$ 3 new equations $+ \eta$
- L: lambda calculus with β and η

$\mathsf{Tm}_{\mathsf{C}}\,(\Gamma \Rightarrow^* A) = \mathsf{Tm}_{\mathsf{C\text{-}var}} \diamond (\Gamma \Rightarrow^* A) = \mathsf{Tm}_{\mathsf{L}} \diamond (\Gamma \Rightarrow^* A) = \mathsf{Tm}_{\mathsf{L}}\,\Gamma\,A$

Open (?) problems in the algebraic setting. What is...

- ...the combinatory equivalent of L without η?
- ...the lambda equivalent of C without η?

Three theories

- C: combinatory logic $+$ 3 new equations $+\ \eta$
- C-var: combinatory logic with variables $+$ 3 new equations $+\ \eta$
- L: lambda calculus with β and η

$$\mathsf{Tm}_{\mathsf{C}}\,(\Gamma \Rightarrow^{*} A) = \mathsf{Tm}_{\mathsf{C\text{-}var}} \diamond (\Gamma \Rightarrow^{*} A) = \mathsf{Tm}_{\mathsf{L}} \diamond (\Gamma \Rightarrow^{*} A) = \mathsf{Tm}_{\mathsf{L}}\,\Gamma\,A$$

Open (?) problems in the algebraic setting. What is...

- ...the combinatory equivalent of L without η?
- ...the lambda equivalent of C without η?
- ...the lambda equivalent of C without extra equations?

Three theories

- C: combinatory logic + 3 new equations + η
- C-var: combinatory logic with variables + 3 new equations + η
- L: lambda calculus with β and η

$\mathsf{Tm}_{\mathsf{C}}\,(\Gamma \Rightarrow^* A) = \mathsf{Tm}_{\mathsf{C\text{-}var}} \diamond (\Gamma \Rightarrow^* A) = \mathsf{Tm}_{\mathsf{L}} \diamond (\Gamma \Rightarrow^* A) = \mathsf{Tm}_{\mathsf{L}}\,\Gamma\,A$

Open (?) problems in the algebraic setting.   What is...

- ...the combinatory equivalent of L without η?
- ...the lambda equivalent of C without η?
- ...the lambda equivalent of C without extra equations?
- ...a dependently typed version of combinatory logic?

Summary

- We proved that the sets of extensional combinatory terms and lambda terms are equal.

Summary

- We proved that the sets of extensional combinatory terms and lambda terms are equal.
- These are well-typed terms quotiented by conversion (QITs).

Summary

- We proved that the sets of extensional combinatory terms and lambda terms are equal.
- These are well-typed terms quotiented by conversion (QITs).
- The proof is formalised in Cubical Agda.

Summary

- We proved that the sets of extensional combinatory terms and lambda terms are equal.
- These are well-typed terms quotiented by conversion (QITs).
- The proof is formalised in Cubical Agda.
- The simply typed and untyped variants are special cases of the general construction.

Summary

- We proved that the sets of extensional combinatory terms and lambda terms are equal.
- These are well-typed terms quotiented by conversion (QITs).
- The proof is formalised in Cubical Agda.
- The simply typed and untyped variants are special cases of the general construction.
- Related work:
  - Textbooks: Curry-Feys-Craig 1959, Barendregt 1985, Hindley-Seldin 2008, Bimbó 2011, ...
  - Selinger 2002: The lambda calculus is algebraic
  - Hyland 2017: Classical lambda calculus in modern dress
  - Dybjer 2019: Categories with families: Unityped, simply typed, dependently typed