A Characterization of Lambda-terms Transforming Numbers

Paweł Parys

University of Warsaw

Results presented on FLOPS 2014

Representing numbers in λ-terms

$$[n] = \lambda f.\lambda x.\ \underbrace{f\ (f\ (f\ \ldots\ (f}_{n}\ x)\ldots))$$ (Church numerals)

Representing numbers in λ-terms

$[n] = \lambda f.\lambda x.\ \underbrace{f\ (f\ (f\ \ldots\ (f}_{n}\ x)\ldots))$ (Church numerals)

We can implement several functions working on such numbers, e.g. addition:

$\text{add} = \lambda n_1.\lambda n_2.\lambda f.\lambda x.\ n_1\ f\ (n_2\ f\ x)$

Representing numbers in λ-terms

$[n] = \lambda f.\lambda x.\ \underbrace{f\ (f\ (f\ \ldots\ (f}_{n}\ x)\ldots))$ (Church numerals)

We can implement several functions working on such numbers, e.g. addition:

$$\text{add} = \lambda n_1.\lambda n_2.\lambda f.\lambda x.\ n_1\ f\ (n_2\ f\ x)$$

In this talk we consider simply-typed λ-calculus
(sorts are of the form $\tau\to\sigma$ constructed out of a base sort o).
The sort of "numbers" is $\mathbb{N}=(o\to o)\to o\to o$.
In fact every closed β-normalized term of this sort represents some number.

Higher-order functions on numbers

We can construct higher-order functions operating on numbers, for example:

$$g(f) = n_1+f(n_2+f(n_3+f(...+f(n_k)...)))$$

Goal of this work: characterize all such functions.

Higher-order functions on numbers

We can construct higher-order functions operating on numbers, for example:

$$g(f) = n_1+f(n_2+f(n_3+f(...+f(n_k)...)))$$

Goal of this work: characterize all such functions.

In order to know precisely the result of g for each f, we need to remember all the numbers $n_1, \ldots, n_k$ (where k arbitrarily big).

Higher-order functions on numbers

We can construct higher-order functions operating on numbers, for example:

$g(f) = n_1+f(n_2+f(n_3+f(...+f(n_k)...)))$

Goal of this work: characterize all such functions.

In order to know precisely the result of g for each f, we need to remember all the numbers n_1, …, n_k (where k arbitrarily big).

What if it is enough to approximate the result?
We can take

$g'(f) = n_1+f(m)$ where $m=n_2+...+n_k$ (assume $n_i>0$)

It contains only two numbers (n_1, m) and approximates g.

Higher-order functions on numbers

We can construct higher-order functions operating on numbers,
for example:

$$g(f) = n_1+f(n_2+f(n_3+f(...+f(n_k)...)))$$

Goal of this work: characterize all such functions.

In order to know precisely the result of g for each f, we need to remember all the numbers $n_1, \ldots, n_k$ (where k arbitrarily big).

What if it is enough to approximate the result?
We can take

$$g'(f) = n_1+f(m) \quad \text{where } m=n_2+...+n_k \quad \text{(assume } n_i>0)$$

It contains only two numbers (n_1, m) and approximates g.

e.g. for $f(x)=2*x$ we have $g'(f) \leqslant g(f) \leqslant g'(f)*2^{g'(f)}$
We have similar relationship for each fixed f (depending on f, but not on the numbers used in g/g').

Contribution

We prove that for every sort, eg. $(((o \to o) \to o \to o) \to ((o \to o) \to o \to o)) \to ((o \to o) \to o \to o)$
there are finitely many types (shapes) of functions,
each of them using a fixed amount of natural numbers.

term M ⟶ types(M), vec(M)

from finite set — vector of numbers of length determined by types(M)

Contribution

We prove that for every sort, eg. $(((o\to o)\to o\to o)\to((o\to o)\to o\to o))\to((o\to o)\to o\to o)$
there are finitely many types (shapes) of functions,
each of them using a fixed amount of natural numbers.

term M ⟶ types(M), vec(M)

from finite set — vector of numbers of length determined by types(M)

Compositionality

types(M), types(N) ⟶ types(M N)

Contribution

We prove that for every sort, eg. $(((o\to o)\to o\to o)\to((o\to o)\to o\to o))\to((o\to o)\to o\to o)$
there are finitely many types (shapes) of functions,
each of them using a fixed amount of natural numbers.

term M ⟶ types(M), vec(M)

from finite set — vector of numbers of length determined by types(M)

<u>Compositionality</u>

types(M), types(N) ⟶ types(M N), linear transformation L

vec(M N) = L(vec(M), vec(N))

Contribution

We prove that for every sort, eg. $(((o\to o)\to o\to o)\to((o\to o)\to o\to o))\to((o\to o)\to o\to o)$
there are finitely many types (shapes) of functions,
each of them using a fixed amount of natural numbers.

term M ⟶ types(M), vec(M)

from finite set — vector of numbers of length determined by types(M)

Compositionality

types(M), types(N) ⟶ types(M N), linear transformation L

vec(M N) = L(vec(M), vec(N))

For a term M of sort $\mathbb{N}=(o\to o)\to o\to o$ representing a number n,
a number m in vec(M) approximates n:

$$m \leqslant H(n) \quad \text{and} \quad n \leqslant H(m)$$

for a fixed (but fast-growing) function H

Remark: Our result holds for every representation of natural numbers in lambda-terms

Consequences: representing tuples

We can represent pairs of numbers (in terms of type $(\mathbb{N}\rightarrow\mathbb{N}\rightarrow\mathbb{N})\rightarrow\mathbb{N}$):

$$[(n_1, n_2)] = \lambda f.\ f\ [n_1]\ [n_2]$$

Consequences: representing tuples

We can represent pairs of numbers (in terms of type $(\mathbb{N}\rightarrow\mathbb{N}\rightarrow\mathbb{N})\rightarrow\mathbb{N}$):

$$[(n_1, n_2)] = \lambda f.\ f\ [n_1]\ [n_2]$$

constructor of pairs:

$$\text{pair} = \lambda n_1.\lambda n_2.\lambda f.\ f\ n_1\ n_2$$

extractors:

$$\text{ext}_1 = \lambda p.\ p\ (\lambda x.\lambda y.\ x)$$

$$\text{ext}_2 = \lambda p.\ p\ (\lambda x.\lambda y.\ y)$$

Consequences: representing tuples

We can represent pairs of numbers (in terms of type $(\mathbb{N}\to\mathbb{N}\to\mathbb{N})\to\mathbb{N}$):

$$[(n_1, n_2)] = \lambda f.\ f\ [n_1]\ [n_2]$$

constructor of pairs:

$$pair = \lambda n_1.\lambda n_2.\lambda f.\ f\ n_1\ n_2$$

extractors:

$$ext_1 = \lambda p.\ p\ (\lambda x.\lambda y.\ x)$$

$$ext_2 = \lambda p.\ p\ (\lambda x.\lambda y.\ y)$$

it holds:

$$ext_1\ (pair\ n_1\ n_2) \to_\beta n_1$$

$$ext_2\ (pair\ n_1\ n_2) \to_\beta n_2$$

Consequences: representing tuples

We can represent pairs of numbers (in terms of type $(\mathbb{N}\to\mathbb{N}\to\mathbb{N})\to\mathbb{N}$):

$$[(n_1, n_2)] = \lambda f.\ f\ [n_1]\ [n_2]$$

constructor of pairs:

$$\text{pair} = \lambda n_1.\lambda n_2.\lambda f.\ f\ n_1\ n_2$$

extractors:

$$\text{ext}_1 = \lambda p.\ p\ (\lambda x.\lambda y.\ x)$$

$$\text{ext}_2 = \lambda p.\ p\ (\lambda x.\lambda y.\ y)$$

it holds:

$$\text{ext}_1\ (\text{pair}\ n_1\ n_2) \to_\beta n_1$$

$$\text{ext}_2\ (\text{pair}\ n_1\ n_2) \to_\beta n_2$$

In a similar way we can represent triples, quadruples, …

But (with such standard representation) for tuples of bigger arities we need to use terms of a more complicated sorts.

Natural question:
Maybe in terms of some sort τ we can represent arbitrarily long tuples (arrays) of integers?

Consequences: representing tuples

Natural question:
Maybe in terms of some sort τ we can represent arbitrarily long tuples (arrays) of integers?

What would it mean?

Of course we can represent k numbers in this way:
$[(n_1, n_2, \ldots, n_k)] = \lambda f.\ f\ n_1\ (f\ n_2\ (\ldots\ (f\ n_{k-1}\ n_k)\ldots))$
but the numbers cannot be extracted...

Consequences: representing tuples

Natural question:

Maybe in terms of some sort τ we can represent arbitrarily long tuples (arrays) of integers?

It would mean that:

For each *k* there exist closed terms

$k\text{tuple} : \mathbb{N}\to\mathbb{N}\to\ldots\to\mathbb{N}\to\tau$

$k\text{ext}_1, \ldots, k\text{ext}_k : \tau\to\mathbb{N}$

such that

$\forall i \quad k\text{ext}_i\ (k\text{tuple}\ n_1\ n_2\ \ldots\ n_k) \to_\beta n_i$

Consequences: representing tuples

Natural question:

Maybe in terms of some sort τ we can represent arbitrarily long tuples (arrays) of integers?

It would mean that (a weaker statement):

For each *k* there exist closed terms

$k\text{ext}_1, \ldots, k\text{ext}_k : \tau \to \mathbb{N}$

and for all $n_1, n_2, \ldots, n_k \in \mathbb{N}$ there exists a closed term T of type τ (a representation of this tuple) such that

$\forall i \quad k\text{ext}_i\ T \to_\beta n_i$

Consequences: representing tuples

Natural question:

Maybe in terms of some sort τ we can represent arbitrarily long tuples (arrays) of integers?

It would mean that (a weaker statement):

For each *k* there exist closed terms

$k\text{ext}_1, \ldots, k\text{ext}_k : \tau \to \mathbb{N}$

and for all $n_1, n_2, \ldots, n_k \in \mathbb{N}$ there exists a closed term T of type τ (a representation of this tuple) such that

$\forall i \quad k\text{ext}_i \; T \to_\beta n_i$

Theorem 1

The answer is NO – such type τ does not exist.

Another point of view

Consider the equivalence relation ~ on terms of the same sort $\tau \rightarrow \mathbb{N}$:

K~L if for each sequence $N_1, N_2, \ldots$ of terms of sort τ,

seq. KN_1, $KN_2, \ldots$ is bounded $\Leftrightarrow$ seq. LN_1, $LN_2, \ldots$ is bounded

e.g. (λn. n) and (λn. add n n) are equivalent.

Another point of view

Consider the equivalence relation ~ on terms of the same sort $\tau \rightarrow \mathbb{N}$:

K~L if for each sequence $N_1, N_2, \ldots$ of terms of sort τ,

seq. KN_1, $KN_2, \ldots$ is bounded $\Leftrightarrow$ seq. LN_1, $LN_2, \ldots$ is bounded

e.g. (λn. n) and (λn. add n n) are equivalent.

Theorem 2.

For each sort τ the relation ~ has finitely many equivalence classes.

Another point of view

Consider the equivalence relation ~ on terms of the same sort $\tau \rightarrow \mathbb{N}$:

K~L if for each sequence $N_1, N_2, \ldots$ of terms of sort τ,

seq. $KN_1, KN_2, \ldots$ is bounded $\Leftrightarrow$ seq. $LN_1, LN_2, \ldots$ is bounded

e.g. (λn. n) and (λn. add n n) are equivalent.

Theorem 2.

For each sort τ the relation ~ has finitely many equivalence classes.

Theorem 1 follows immediately from Theorem 2: the extractors cannot be equivalent, so length of representable tuples is not greater than the number of equivalence classes of ~.

(Longer tuples cannot be represented even when we allow approximate extraction, up to some error).

Another point of view

Consider the equivalence relation ~ on terms of the same sort $\tau\rightarrow\mathbb{N}$:

K~L if for each sequence $N_1,N_2,\ldots$ of terms of sort τ,

seq. $KN_1, KN_2,\ldots$ is bounded $\Leftrightarrow$ seq. $LN_1, LN_2,\ldots$ is bounded

Theorem 2.
For each sort τ the relation ~ has finitely many equivalence classes.

<u>Proof of Theorem 2</u>: if types(K)=types(L), then K~L.

Take K, L such that types(K)=types(L), and take $N_1,N_2,\ldots$ such that seq. $KN_1, KN_2,\ldots$ is bounded. Goal: seq. $LN_1, LN_2,\ldots$ is bounded.

W.l.o.g. types(N_1)=types(N_2)=...

value of $KN_j \approx$ a number in vec(KN_j),
value of $LN_j \approx$ a number in vec(LN_j),

vec(KN_j) = Lin(vec(K), vec(N_j)) $\approx$ Lin(vec(L), vec(N_j)) = vec(LN_j)
(where Lin is determined by types(K) and types(N_1) – the same for each j)

Thus $LN_1, LN_2,\ldots$ is bounded.

Techniques used

Intersection type system:

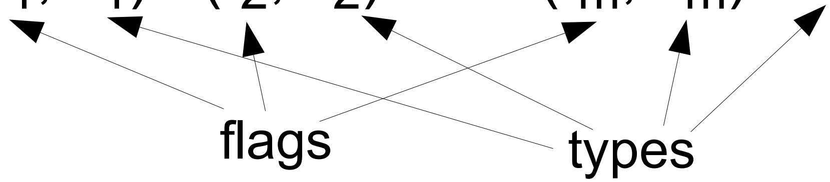
- Intersection types refine sorts (simple types).
- To a term we assign a pair (flag, type), where flag∈{pr, np} (“productive”, “nonproductive”).
- One base type: o.
- The types are of the form $(f_1, \tau_1) \wedge (f_2, \tau_2) \wedge \ldots \wedge (f_m, \tau_m) \rightarrow \tau$.

flags types

To one term we may assign multiple pairs (flag, type).

Intersection types

The types are of the form $(f_1, \tau_1) \wedge (f_2, \tau_2) \wedge \ldots \wedge (f_m, \tau_m) \to \tau$.



When a term M has such type, it means that if to the argument of the function M we can assign all pairs (f_1, τ_1), (f_2, τ_2), ..., (f_m, τ_m), then the result has type τ.

Moreover M is <u>required</u> to use its argument in each of these types (we have type $\top \to \tau$ (with m=0) when the argument is not used at all).

Thus we know precisely which arguments are used and with which types.

Intersection types

Beside of a type, to a term M we also assign a flag.

Flag “productive” means that M adds something to the resulting value (in addition to the value supported by the arguments):

– M is productive when it uses some of its productive arguments more than once (we look at the derivation tree, not at the term itself).

e.g. F=(λf.λx. f (f x)) is productive for productive f

because if f adds 1, then (F f x) is bigger than (f x)

but F=(λf.λx. f x) is nonproductive (even when f is productive),

because (F (F (F f))) = f.

To one term we may assign multiple pairs (flag, type).

Typing rules

$$\frac{\alpha = \overbrace{o \to \cdots \to o}^{k} \to o}{\emptyset \vdash \mathbf{c}^{\alpha} : (\mathsf{pr}, \underbrace{(\mathsf{pr}, o) \to \cdots \to (\mathsf{pr}, o)}_{k} \to o)} \qquad x : (f, \tau) \vdash x : (\mathsf{np}, \tau)$$

$$\frac{\Gamma \cup \{x : (f_i, \tau_i) \mid i \in I\} \vdash M : (f, \tau) \qquad x \notin dom(\Gamma)}{\Gamma \vdash \lambda x.M : (f, \bigwedge_{i \in I}(f_i, \tau_i) \to \tau)}\ (\lambda)$$

$$\frac{\Gamma \vdash M : (f', \bigwedge_{i \in I}(f_i^{\bullet}, \tau_i) \to \tau) \qquad \Gamma_i \vdash N : (f_i^{\circ}, \tau_i) \text{ for each } i \in I}{\Gamma \cup \bigcup_{i \in I} \Gamma_i \vdash MN : (f, \tau)}\ (@)$$

where in the (@) rule we assume that

- each pair $(f_i^{\bullet}, \tau_i)$ is different (where $i \in I$), and
- for each $i \in I$, $f_i^{\bullet} = \mathsf{pr}$ if and only if $f_i^{\circ} = \mathsf{pr}$ or $\Gamma_i{\restriction_{\mathsf{pr}}} \neq \emptyset$, and
- $f = \mathsf{pr}$ if and only if $f' = \mathsf{pr}$, or $f_i^{\circ} = \mathsf{pr}$ for some $i \in I$, or $|\Gamma{\restriction_{\mathsf{pr}}}| + \sum_{i \in I} |\Gamma_i{\restriction_{\mathsf{pr}}}| > |(\Gamma \cup \bigcup_{i \in I} \Gamma_i){\restriction_{\mathsf{pr}}}|$.

Typing rules - example

$$b_x = x : (\mathsf{pr}, o)$$

$$b_y = y : (\mathsf{pr}, (\mathsf{pr}, o) \to o)$$

$$\dfrac{\dfrac{\dfrac{b_y \vdash y : (\mathsf{np}, (\mathsf{pr}, o) \to o) \qquad \dfrac{b_y \vdash y : (\mathsf{np}, (\mathsf{pr}, o) \to o) \qquad b_x \vdash x : (\mathsf{np}, o)}{b_x,\, b_y \vdash y\, x : (\mathsf{np}, o)}\,(@)}{b_x,\, b_y \vdash y\,(y\, x) : (\mathsf{pr}, o)}\,(@)}{b_y \vdash \lambda x.y\,(y\, x) : (\mathsf{pr}, (\mathsf{pr}, o) \to o)}\,(\lambda)}{\vdash \lambda y.\lambda x.y\,(y\, x) : (\mathsf{pr}, (\mathsf{pr}, (\mathsf{pr}, o) \to o) \to (\mathsf{pr}, o) \to o)}\,(\lambda)$$

$$b'_y = y : (\mathsf{pr}, (\mathsf{pr}, o) \to o)$$

$$\dfrac{\dfrac{\dfrac{b'_y \vdash y : (\mathsf{np}, (\mathsf{pr}, o) \to o) \qquad \dfrac{b'_y \vdash y : (\mathsf{np}, (\mathsf{pr}, o) \to o) \qquad b_x \vdash x : (\mathsf{np}, o)}{b_x,\, b'_y \vdash y\, x : (\mathsf{np}, o)}\,(@)}{b_x,\, b'_y \vdash y\,(y\, x) : (\mathsf{np}, o)}\,(@)}{b'_y \vdash \lambda x.y\,(y\, x) : (\mathsf{np}, (\mathsf{pr}, o) \to o)}\,(\lambda)}{\vdash \lambda y.\lambda x.y\,(y\, x) : (\mathsf{np}, (\mathsf{np}, (\mathsf{pr}, o) \to o) \to (\mathsf{pr}, o) \to o)}\,(\lambda)$$

Techniques used

Step 2: count "how much a term is productive".

To each typed term M (in fact to a derivation tree for $M:(f,\tau)$) we assign a number $\mathrm{val}(M,\tau)$, which counts:

– the number of application subterms KL such that a productive variable is used both in K and in L.

Easy <u>observation</u> – compositionality:
For closed terms it holds
$\mathrm{val}(KL,\tau)=\mathrm{val}(K,(f_1, \tau_1)\wedge\ldots\wedge(f_m, \tau_m)\to\tau)+\mathrm{val}(L, \tau_1)+\ldots+\mathrm{val}(L, \tau_m)$.

Quite difficult <u>lemma</u>:
For closed terms $M \to_\beta N$ of base sort it holds

$$\mathrm{val}(M,o) \leq \mathrm{val}(N,o) \leq \underbrace{2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}}_{\mathrm{val}(M,o)}{}^{\mathrm{val}(M,o)}$$

Techniques used

Quite difficult lemma:

For closed terms $M \to_\beta N$ of base sort it holds

$$\mathrm{val}(M,o) \le \mathrm{val}(N,o) \le \underbrace{2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}}_{\mathrm{val}(M,o)}{}^{\mathrm{val}(M,o)}$$

To prove this lemma, we need to:

– isolate closed subterms in M,

– replace the tower of 2^2 by an appropriately defined high(M),

– perform the head β-reduction first (closed subterms remain closed), and prove that val(M) increases and high(M) decreases.

Thank you.