How Many Numbers Can a Lambda-Term Contain?

Paweł Parys

University of Warsaw

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
(types are of the form $\tau \rightarrow \sigma$ constructed out of a base type o).
The type of "numbers" is $\mathbb{N}=(o\rightarrow o)\rightarrow o\rightarrow o$.
In fact each closed β-normalized term of this type represents some number.

Representing pairs

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

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

Representing pairs

We can also represent pairs (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:

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

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

Representing pairs

We can also represent pairs (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:

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

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

it holds:

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

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

Representing pairs

We can also represent pairs (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 natural representation) for tuples of bigger arities we need to use terms of a more complicated type.

Natural question:

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

Representing tuples

Natural question:

Maybe in terms of some type 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...

Representing tuples

Natural question:

Maybe in terms of some type 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

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

Representing tuples

Natural question:

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

It would mean that (a weaker statement):

For each k there exist closed terms

$k\mathrm{ext}_1, ..., k\mathrm{ext}_k : \tau \to \mathbb{N}$

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

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

Representing tuples

Natural question:

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

It would mean that (a weaker statement):

For each *k* there exist closed terms

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

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

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

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 type $\tau \rightarrow \mathbb{N}$:

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

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

e.g. $(\lambda n.\ n)$ and $(\lambda n.\ \text{add } n\ n)$ are equivalent.

Another point of view

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

$K \sim L$ if for each sequence $N_1, N_2, \ldots$ of terms of type τ,

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

e.g. $(\lambda n.\ n)$ and $(\lambda n.\ \text{add } n\ n)$ are equivalent.

Theorem 2.

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

Another point of view

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

K~L if for each sequence $N_1, N_2, \ldots$ of terms of type 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 type 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).

Motivation (related work)

A similar theorem turns out to be useful while proving that all higher-order recursion schemes (that is λY-terms) generate more trees than those of them which are “safe”.

“Safety” is a widely considered syntactic restriction, which simplifies some reasonings.

they generate Böhm trees, which are infinite trees

Techniques used

To simplify the analysis we add constants: **0** : o and **1+** : $o \to o$.

For each n of type $\mathbb{N}$, the term (n **1+ 0**) after normalization is of the form

$$\underbrace{\mathbf{1+}\ (\mathbf{1+}\ (\ldots\ (\mathbf{1+}}_{n}\ \mathbf{0})\ldots))$$

Techniques used

Intersection type system:

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

flags types

- It will turn out that the equivalence class of ~ depends only on the set of such pairs (flag, type) which can be assigned to a term.

Intersection types

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

flags types

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):

– the use of **1+** is productive (a 1+ has to appear in the derivation of a type, which means that it is really used),

– M is productive also when it uses some of its productive arguments more than once (again, we look at the derivation tree).

e.g. F=(λf. f (f **0**)) is productive, because (f **1+**) = (**1+** (**1+** **0**))

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

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

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,τ)) we assign a number val(M), which counts:

– the number of **1+** nodes in the derivation tree, and

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

Easy <u>observation</u> – compositionality:
For closed terms it holds val(KL)=val(K)+val(L).

Quite difficult <u>lemma</u>:
For closed terms of base type it holds

$$\mathrm{val}(M) \leq \text{the number represented by } M \leq \underbrace{2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}}_{\text{the maximal order of a subterm of M}}{}^{\mathrm{val}(M)}$$

Techniques used

Quite difficult lemma:
For closed terms of base type it holds

$\mathrm{val}(M) \leq$ the number represented by $M \leq 2^{2^{2^{\cdot^{\cdot^{\cdot^{2^{\mathrm{val}(M)}}}}}}}$

the maximal order of a subterm of M

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.

Proof of the theorem

Easy <u>observation</u> – compositionality:
For closed terms it holds val(KL)=val(K)+val(L).

Quite difficult <u>lemma</u>:
For closed terms of base type it holds

$$\mathrm{val}(M) \leq \text{the number represented by } M \leq \underbrace{2^{2^{2^{\cdots 2^{\mathrm{val}(M)}}}}}_{\text{the maximal order of a subterm of M}}$$

We want to prove that:

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

The sequences are almost: (lemma)

val(KN_1), val(KN_2), … and val(LN_1), val(LN_2), …

Proof of the theorem

Easy observation – compositionality:
For closed terms it holds val(KL)=val(K)+val(L).

Quite difficult lemma:
For closed terms of base type it holds

$\mathrm{val}(M) \le$ the number represented by $M \le 2^{2^{2^{\cdots 2^{\mathrm{val}(M)}}}}$

the maximal order of a subterm of M

We want to prove that:

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

The sequences are almost: (lemma + observation)

$\mathrm{val}(K)+\mathrm{val}(N_1), \mathrm{val}(K)+\mathrm{val}(N_2), \ldots$ and $\mathrm{val}(L)+\mathrm{val}(N_1), \mathrm{val}(L)+\mathrm{val}(N_2), \ldots$

so they differ only by a constant val(L)-val(K).

Proof of the theorem

Easy observation – compositionality:
For closed terms it holds $\mathrm{val}(KL)=\mathrm{val}(K)+\mathrm{val}(L)$.

Quite difficult lemma:
For closed terms of base type it holds

$$\mathrm{val}(M) \leq \text{the number represented by } M \leq \underbrace{2^{2^{2^{\cdots 2^{\mathrm{val}(M)}}}}}_{\text{the maximal order of a subterm of M}}$$

We want to prove that:
seq. $KN_1, KN_2, \ldots$ is bounded $\Leftrightarrow$ seq. $LN_1, LN_2, \ldots$ is bounded

The sequences are almost: (lemma + observation)
$\mathrm{val}(K)+\mathrm{val}(N_1), \mathrm{val}(K)+\mathrm{val}(N_2), \ldots$ and $\mathrm{val}(L)+\mathrm{val}(N_1), \mathrm{val}(L)+\mathrm{val}(N_2), \ldots$

so they differ only by a constant $\mathrm{val}(L)-\mathrm{val}(K)$.

This is true assuming that we can use the same types for K and L, that is the same type for N_i in both sequences...

Thank you.