Записки научных
семинаров ПОМИ
Том 468, 2018 г.

S. Soloviev, J. Malakhovski

AUTOMORPHISMS OF TYPES AND THEIR APPLICATIONS

Abstract. We outline recent results in the theory of type isomorphisms and automorphisms and present several practical applications of said results that can be useful in the contexts of programming and data security.

§1. Introduction

The aim of this paper is to present an approach and several examples to the creation of systems relying on the recent advances in the study of type isomorphisms and automorphisms [26].

The power of type isomorphisms and automorphisms is illustrated by the fact that they have a fairly simple computational structure closed under composition behind them (finite hereditary permutations) [7] and the groups of automorphisms of higher order types can represent arbitrary finite groups [26]. The former property allows us to efficiently compute answers to several common problems up to isomorphism, while exploiting the latter property allows us to encode many conventional cryptographic primitives directly in Type Theory. Among other examples we consider automated theorem proving and typed library search up to isomorphism, and an instance of **ElGamal** cryptosystem based on type automorphisms.

We omit proofs published elsewhere because the purpose of this article is to present an idea of a method (most relevant proofs may be found in [26]).

As a main system to present our ideas we shall use the second order λ-calculus $\lambda^2\beta\eta$ (system F). Its subsystems and extensions will also play some role. The system that "underlies" all these systems is the untyped λ-calculus Λ. For a detailed presentation of λ-calculus (typed and untyped)

Key words and phrases: lambda calculus, type theory, isomorphisms, automorphisms, automated theorem proving, library search, group theory, cryptography, data security.

This work was partially financially supported by the Goverment of Russian Federation (Grant 08-08) and by EU COST Action CA15123.

see [2, 13]. In the next section we shall present a brief description of these systems, mainly for notation clarity purposes.

§2. Lambda Calculus

2.1. Untyped Lambda Calculus. The syntax of Λ is very simple. The class of λ-terms consists of words constructed from the variables $x, y, z \ldots$ using abstraction operator λ and parentheses (,). The class Λ of λ-terms is the least class such that:

- if x is a variable, $x \in \Lambda$;
- if $M \in \Lambda$ then $(\lambda x.M) \in \Lambda$
- if $M, N \in \Lambda$ then $(MN) \in \Lambda$.

The symbol $\equiv$ denotes syntactic equality. The usual convention is to elide internal . and λ, assume that abstraction associates to the right, application to the left, and application precedes abstraction (that permits to omit some of the parentheses). For example,

$$\lambda x_1 x_2 x_3.MN_1N_2N_3 \equiv (\lambda x_1.(\lambda x_2.(\lambda x_3.(((MN_1)N_2)N_3)))).$$

Sometimes we will need to abbreviate even more. Given a list of indexes $i_1, \ldots, i_n$, instead of $A_{i_1} \ldots A_{i_n}$ we will write using a vector notation $A_{i_1 \div i_n}$. This abbreviation may be used with the convention above, for example $\lambda x_{1\div n}.MN_{1\div n}$ will have the same meaning as $\lambda x_1 \ldots x_n.MN_1 \ldots N_n$.

In the term $\lambda x.M$, M is called the *scope* of λx. All occurrences of a variable x in some term that are not in the scope of any λx are called free. The set of all the free variables of M is denoted $FV(M)$. All the free occurrences of x in M are bound by λx in $\lambda x.M$. The term without free variables is called closed. Now, α-conversion is just a renaming of bound variables. It defines an equivalence relation on terms (α-equivalence). We shall write $\equiv_\alpha$ for syntactic equality extended by α-conversion.

As usual, the *variable convention* (justified by α-equivalence) is used: if terms $M_1, \ldots, M_n$ occur in a certain context then in these terms all bound variables are chosen to be different from free variables.

The syntax form $[N/x]M$ denotes the substitution of N for all the free occurrences of x in M with renaming of bound variables. To shorten the formulas we may use the notation with $\div$, that is, $[N_{1\div n}/x_{1\div n}]M$ will also mean $[N_1/x_1](\ldots([N_n/x_n]M)\ldots)$.

Normalization is the process based on *reductions* (oriented conversions) leading to a *normal form*. In the untyped λ-calculus two reductions are considered:

- (β) $(\lambda x.M)N \to [N/x]M$,
- (η) $\lambda x.(Mx) \to M$, $x \notin FV(M)$.

The α-congruence is not used as a reduction but only to respect the variable convention. Subterms of the form $(\lambda x.M)N$ are called *β-redexes*. Similarly, subterms of the form $\lambda x.(Mx)$ $(x \notin FV(M))$ are called *η-redexes*. When a term contains no β-redexes it is said to be in β-normal form, when a term in β-normal form contains no η-redexes it is said to be in $\beta\eta$-normal form.

Notation $M \to^* N$ means that there exists a reduction sequence from M to N. Term M is said to have a β-normal form ($\beta\eta$-normal form) N if there exists a β $(\beta\eta)$ reduction sequence $M \to^* N$. In the untyped λ-calculus not all the terms have normal forms. For example, the fixed point combinator $\mathbf{Y} \equiv \lambda f.((\lambda x.f(xx))(\lambda x.f(xx)))$ does not have any normal form. Respectively, for some terms the normalization process may not terminate. A reduction sequence can not be extended only if it ends in a normal form.

M is said to be weakly normalizing ($\mathcal{WN}(M)$) if there exists a finite reduction sequence starting with M and leading to some normal term. M is said to be strongly normalizing ($\mathcal{SN}(M)$) if all reduction sequences starting with M are finite.

Definition 1. *Reduction $\to$ is said to have the $\mathcal{CR}$ property iff for each $M \to^* M_1$ and $M \to^* M_2$ there exists N such that $M_1 \to^* N$ and $M_2 \to^* N$.*

Theorem 1. *Church–Rosser.* ([2, Th. 3.3.9]).

- *Both β-reduction and $\beta\eta$-reduction have the $\mathcal{CR}$ property.*
- *If $M_1 = M_2$ then there exists some term N such that $M_1 \to^* N$ and $M_2 \to^* N$.*

This has two important consequences:

- *M has a β-normal form ($\beta\eta$-normal form) if there exists an N such that $M = N$ and N is a β-normal form ($\beta\eta$-normal form);*
- *M does have at most one normal form: all reduction sequences that terminate, do terminate with the same normal form.*

Two terms M, N are said to be *convertible* if there exists a third term K such that both M and N can be reduced to K by applying some reduction sequences $M \to^* K$, $N \to^* K$. The equality $=$ on terms is the reflexive symmetric transitive closure of convertibility relation.

Remark 1. From the point of view of computation, the untyped λ-calculus is very powerful: it has the same computational power as Turing machines or any other equivalent formalism, such as partial recursive functions. It has obvious consequences for decidability and complexity. Roughly speaking, every known "hard problem" can be modeled. In particular, the property $\mathcal{WN}(M)$ is undecidable (equivalent to the Halting Problem).

2.2. Typed Systems. The proof-theoretical presentation of $\lambda^2\beta\eta$ below is based on [7].

The class of types of $\lambda^2\beta\eta$ is constructed from the type variables X, Y, $Z\ldots$ using type constructors $\to, \forall$ and parentheses (,). It is the least class Θ such that:

- if X is a type variable, $X \in \Theta$;
- if $A, B \in \Theta$ then $(A \to B) \in \Theta$;
- if $A \in \Theta$ and X is a type variable then $\forall X.A \in \Theta$.

To omit some of the parentheses, it is assumed that $\to$ is applied first, all operations associate to the right, and the convention that permits to elide internal $\forall$ and . is applied. For example,

$$\forall XY.X \to Y \to X \to X \equiv \big(\forall X.(\forall Y.(X \to (Y \to (X \to X))))\big).$$

Usually the types $A \to B$ are referred to as the arrow (or function) types. The variables in types are bound by $\forall$. In the type $\forall X.A$, the scope of $\forall X$ is A. The usual α-equality and variable convention are extended to types. This permits to define the substitution of types into types in a way similar to the untyped λ-terms.

The class of pre-terms is the smallest class Θ such that:

- the term variables $x \in \Theta$;
- if x is a term variable, A is a type, and $M \in \Theta$ then $\lambda x : A.M \in \Theta$;
- if $M \in \Theta$ and $N \in \Theta$ then $(MN) \in \Theta$;
- if X is a type variable and $M \in \Theta$ then $\lambda X.M \in \Theta$ is a pre-term;
- if $M \in \Theta$ and A is a type then $(MA) \in \Theta$.

In the pre-terms there are two binders, λ and $\forall$ (it may be used inside types), but the notions of α-equality, the variable convention, and the definition of the substitution can be extended to pre-terms [7].

To define well-typed terms we introduce a deductive system closely related to the second-order propositional calculus. Below $\Gamma, \Delta\ldots$ denote the contexts of type declarations, *i.e.*, the lists of typed term variables $x : A, y : B\ldots$ where each variable name $x, y, \ldots$ is used at most once.

The *typing judgements* are the expressions of the form $\Gamma \vdash M : A$ where Γ is a context, M is a pre-term, and A is a type. Well-formed terms (or merely terms) are pre-terms that are part of the typing judgements derivable in the deductive system below.

Axiom: $$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}.$$

Rules:

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.M : A \to B}(\to -intro) \quad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash (MN) : B}(\to -elim)$$

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \lambda X.M : \forall X.A}(\forall - intro) * \quad \frac{\Gamma \vdash M : \forall X.A}{\Gamma \vdash MB : A[B/X]}(\forall - elim) * *$$

* For the type variable X not free in the type of any free term variable that occurs in the term M.

** For any type B.

To define the equality relation, the following basic equalities are considered:

$$(\beta) \quad (\lambda x : A.M)N = M[N/x], \quad (\eta) \quad \lambda x : A.(Mx) = x \; if \; x \notin FV(M),$$

$$(\beta^2) \quad (\lambda X.M)A = M[A/X], \quad (\eta^2) \quad \lambda X.(MX) = M.$$

The equality generated by $\beta, \eta, \beta^2, \eta^2$ is denoted by $=_2$. This system is strongly normalizing and has the Church-Rosser property [7], so to check $=_2$ it is enough to compare normal forms.

Other systems.

(i) The calculus $\lambda^1\beta\eta$. It is the $\lambda^2\beta\eta$ restricted to the first order or simple types (types that do not contain $\forall$). Equality of terms in $\lambda^1\beta\eta$ is generated by β, η and denoted by $=_1$. Equality of types is syntactic identity.

(ii) Dependent type systems. In general, those are systems where types may depend on terms. We shall not go far into details in this case, because most of technical considerations below are valid already in $\lambda^2\beta\eta$. Two rules that introduce dependent product look very similar to $\to$-introduction and abstraction in $\lambda^2\beta\eta$ and $\lambda^1\beta\eta$:

$$\frac{\Gamma, x : K \vdash K' \, kind}{\Gamma \vdash (x : K)K' \, kind} \qquad \frac{\Gamma, x : K \vdash M : K'}{\Gamma \vdash [x : K]M : (x : K)K'}$$

(though x may occur in K'.)

The rules are taken from the system LF considered in [20]. This system contains a special type $Type$, and because of that in general the types of LF are called *kinds*.

In LF the expression $(x : K)K'$ denotes dependent product, and $[x : K]M$ the λ-abstraction. The $\forall X.A(X)$ of $\lambda^2\beta\eta$ can be modeled by $(x : Type)K(x)$.

The following rule (application) shows how the *kinds* in LF may be influenced by terms:

$$\frac{\Gamma \vdash N : (x : K)K' \quad \Gamma \vdash M : K}{\Gamma \vdash (NM) : [M/x]K'}.$$

(iii) Extensions with induction-recursion. All the systems mentioned above ($\lambda^1\beta\eta$, $\lambda^2\beta\eta$, LF) may be extended by adding inductive types, like the type of natural numbers $Nat = Ind\,X.\{0 : X, succ : X \to X\}$ or the type of ω-trees $T_\omega = Ind\,X.\{0 : X, succ : X \to X, \lim : (Nat \to X) \to X\}$.

As usual, inductive types are defined with corresponding recursion operators (constants defined together with appropriate computation rules usualy called ι-reduction).

All of the typed systems mentioned above are $\mathcal{SN}$. For $\lambda^1\beta\eta$ and $\lambda^2\beta\eta$ see [7], for LF and its extension UTT with induction-recursion (and also type universes) see [11]. The kinds (since they contain terms) in LF and UTT also may be normalized.

§3. Isomorphisms and Automorphisms of Types

To define the notion of isomorphism of types, one needs only a sort of partial categorical structure: for all types A, B the class of terms that represent morphisms from A to B (usually one takes the terms $t : A \to B$); for each type A, a term id_A that represents identity (usually $\lambda x : A.x$); a composition $\circ$ (at least for terms $A \to B$ and $B \to A$); and an equivalence relation $\equiv$ on terms (usually the $\beta\eta$-eqivalence). The term t from A to B is an isomorphism iff there exists t^{-1} from B to A such that $t^{-1} \circ t \equiv id_A$ and $t \circ t^{-1} \equiv id_B$. In this case we call the types A, B isomorphic and write $A \sim B$. In a special case when A and B are equal, the isomorphisms $A \to B$ are called automorphisms $A \to A$.

The equality of types may be syntactic identity, α-equality (in second order lambda calculus), or based on equivalence of terms (in case of dependent types that may depend on terms).

Groupoids are defined as small categories where each arrow is invertible [4]. Let K be a category. For $A \in Ob(K)$, let $K_{\mathrm{iso}}(A)$ denote the subcategory of K that contains all $A' \in Ob(K)$ such that $A \sim A'$. Its morphisms are the isomorphisms $f : A' \to A''$ where $A' \sim A \sim A''$. The

category $K_{\mathrm{iso}}(A)$ is a groupoid. The graph of $K_{\mathrm{iso}}(A)$ is a connected component of the graph of K.

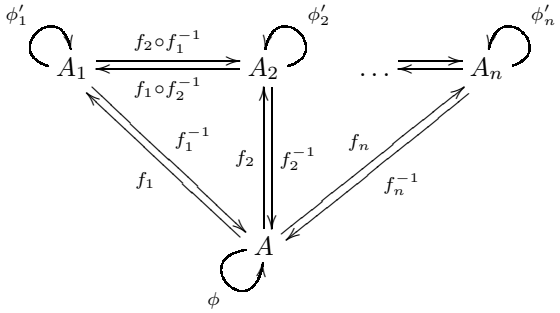
Let $\mathrm{Aut}_K(A)$ denote the group of automorphisms of A, that is, of isomorphisms $A \to A$, with composition as group multiplication, $fg = f \circ g$. It may be seen as a category with one object. It is also a full subcategory of $K_{\mathrm{iso}}(A)$. When K is clear, we shall omit the index K and write $\mathrm{Aut}(A)$.

Lemma 1. *Let $A, A' \in Ob(K)$ be isomorphic. Then a) $K_{\mathrm{iso}}(A) = K_{\mathrm{iso}}(A')$ and b) the groups $\mathrm{Aut}_K(A)$ and $\mathrm{Aut}_K(A')$ are isomorphic as groups.*

Lemma 2. *Let $f : A \to A'$ be an isomorphism in K, any other isomorphism $g : A \to A'$ may be uniquely represented as $g = f \circ h$ and $g = h' \circ f$ where $h : A \to A, h' : A' \to A'$.*

Theorem 2. *For each pair $B, C \in Ob(K_{\mathrm{iso}}(A))$ a distinguished isomorphism f_{BC} may be selected in such a way that every diagram of distinguished isomorphisms commutes. The result obviously holds also for the isomorphisms in $K^{op}_{\mathrm{iso}}(A)$.*

In type theories that we considered above the number of types isomorphic to A is finite, so the groupoid may be represented by the diagram



where $A_1, \ldots, A_n$ are all types that are isomorphic (but not equal) to A (and to each other), f_i, f_i^{-1} denote the fixed isomorphisms and their inverses, ϕ denotes an arbitrary automorphism of A and $\phi'_i = f_i \circ \phi \circ f_i^{-1}$ $(1 \leqslant i \leqslant n)$.

3.1. Finite hereditary permutations.

Definition 2. (*Cf.* [6, p. 323]), *Let M and N be normal terms in Λ. For M to be inverse of N means that both relations $\lambda x.M(Nx) \to^* \lambda x.x$ and $\lambda x.N(Mx) \to^* \lambda x.x$ are valid.*

Definition 3. (*Cf.* [6] *and* [7], *def.* 1.9.2.) *An untyped* λ*-term* M *is a finite hereditary permutation (f.h.p.) iff*

- $M \equiv \lambda x.x$, *or*
- $M \equiv \lambda z.\lambda x_{\sigma(1)\div\sigma(n)}.zM_{1\div n}$ *where* σ *is a permutation of the set* $\{1,\dots,n\}$ *and* $\lambda x_i.M_i$ *is a finite hereditary permutation for all* $1 \leqslant i \leqslant n$.

  The first variable of an f.h.p. after the λ*-prefix will be called its head variable.*

It follows immediately from this definition that f.h.p.'s are closed terms.

Examples 1. The following terms are f.h.p.'s:

- $\lambda z.\lambda x_2 x_1.zx_1x_2$;
- $\lambda z.\lambda x_2 x_1 x_3.zx_1x_2(\lambda y_2 y_1.x_3 y_1 y_2)$.

The terms $M_{1\div n}$ themselves are not f.h.p.'s (an abstraction λx_i has to be applied). In difference from both Di Cosmo and Dezani we apply permutation to the indexes in the prefix and not at the right under the application. When the erasures of typed λ-terms are considered, it helps to reconstruct directly their type from the λ-prefix. In fact both definitions are related by α-conversion via $x_i \mapsto x_{\sigma^{-1}(i)}$ and thus are equivalent.

Let us notice that the f.h.p.'s are not necessarily normal. For example, if in the definition of an f.h.p. M_n is $x_{\sigma(n)}$ then an η-reduction is possible; if $M_{n-1} \equiv x_{\sigma(n-1)}$ then another η-reduction is possible afterwards, and similar η-reductions may be possible inside M_i.

However such η-reductions are the only reductions possible due to the definition of an f.h.p. Via these reductions an f.h.p. always reduces to a unique normal form that is also an f.h.p.

The detailed technical proof may be already found in [6]. As a brief explanation, let us quote [7]: "One may easily show that the f.h.p.'s are typable terms... By the usual abuse of language we may then speak of typed f.h.p.'s. Recall now that all typed terms possess a (unique) normal form (see [2])."

We shall permit us an abuse of language and call f.h.p.'s all terms that possess a normal form and this normal form is an f.h.p.

The main result about invertible untyped λ-terms is given by the following theorem.

Theorem 3. (*See* [7], *theorem* 1.9.1*; cf.* [6], *main theorem.*) *Let* M *be an untyped term that possesses a normal form. Then* M *is invertible iff it is an f.h.p.*

3.2. Typed Isomorphisms. For a type A in each type theory where a notion of isomorphism is defined as above, one may define the groupoid $Gr(A)$ whose objects are the types $A' \sim A$ (all types isomorphic to A) and whose morphisms are the isomorphisms between such types, and the group $\mathrm{Aut}(A)$ of automorphisms $A \to A$. (The elements of this group are λ-terms considered up to $=$ and the group operation is the composition of λ-terms.)

It is easily shown that if $\Gamma \vdash (x : K)K'$ is an isomorphism then x has no free occurrences in (the normal form of) K'. If in the kind $(x : K)K'$ $x \notin FV(K')$ we shall usually write $K \to K'$.

The relation of isomorphism between types is decidable in $\lambda^1\beta\eta$, $\lambda^2\beta\eta$, and LF [25]. Moreover, there exists only finite number of types (kinds) isomorphic to a given type (kind).

Remark 2. For the complexity of this decision problem in LF a lower bound is given by graph isomorphism problem.

Definition 4. (*See* [26], *cf.* [7], *p.*48.) *Let us consider $\lambda^1\beta\eta$, $\lambda^2\beta\eta$ and LF. Let M be a typed λ-term,* i.e., $\Gamma \vdash M : A$ *in one of these systems. The erasure $e(M)$ is defined as follows:*

- *in:*

$\lambda^1\beta\eta$: $e(x) \equiv x$; $e(\lambda x : A.N) \equiv \lambda x.e(N)$; $e(M_1M_2) \equiv e(M_1)e(M_2)$

(one may say merely that all type labels of variables are erased);

- *in LF (we consider M being part of $\Gamma\, M : K$):*

$e(x) \equiv x$; $e([x : K]N) \equiv \lambda x.e(N)$; $e(M_1M_2) \equiv e(M_1)e(M_2)$;

- *in $\lambda^2\beta\eta$ we define first $e(B)$ for types, because types may occur not only as labels of variables: the untyped λ-calculus will contain now two sorts of variables: $x, y, z, \ldots$ and $X, Y, Z \ldots$ (inside the calculus they are treated in exactly the same way, and introduced only to trace the origin of these variables) and be extended by two constants, $K_\forall$ and $K_\to$.*
  - *Let B be a type. We define*[1]

$e(X) \equiv X$, $e(B_1 \to B_2) \equiv K_\to(e(B_1)e(B_2))$, $e(\forall X.B_0) \equiv K_\forall(\lambda X.e(B_0))$.

[1]The abstraction λ is introduced in $e(\forall X.A)$ to respect binding. This definition is inspired by the definition of erasure for second order types in [5]. However, we modified the definition of $e(A \to B)$. Bruce and Longo used $e(A \to B) \equiv K_\to e(A)e(B)$ but with their definition erasure may create redexes, for example $e(\forall X.(Y \to X)) \equiv K_\forall(\lambda X.K_\to YX)$. With our definition $e(A)$ is normal for any type A. In fact, when one is interested only

– *Let M be a term. Now $e(x) \equiv x$, $e(\lambda x : A.N) \equiv \lambda x.e(N)$, $e(\lambda X.N) \equiv \lambda X.e(N)$, $e(NB) \equiv e(N)e(B)$, $e(M_1M_2) \equiv e(M_1)e(M_2)$.*

A fundamental fact is that $\theta : S \to S'$ is an isomorphism iff its erasure $e(\theta)$ is a finite hereditary permutation [6, 7]. In particular it does not contain constants $K_{\to}, K_{\forall}$. The following theorem is given in a stronger formulation than similar theorems in [7] (for $\lambda^1\beta\eta$ and $\lambda^2\beta\eta$) or [6] (for $\lambda^1\beta\eta$). We put together and slightly strengthen our own theorems 3.13, 3.15 and 3.25 from [26].

Theorem 4. *Let $\Gamma \vdash \theta : S \to S'$ (in $\lambda^1\beta\eta$, $\lambda^2\beta\eta$ or LF). Then θ is an isomorphism iff $e(\theta)$ is an f.h.p. Moreover, by $e(\theta)$ and one of the types S, S' it is possible to reconstruct θ.*

3.3. Computing Isomorphisms. We shall now show how to compute isomorphisms between types.

Note that every $\lambda^1\beta\eta$ type A can be written as

$$A \equiv T_1 \to T_2 \to \dots T_n \to Z$$

with Z being a type variable, and $T_{1\div n}$ being some types. Meanwhile, every $\lambda^2\beta\eta$ type B can be transformed by applying the following rewrite rule (half of an isomorphism)

$$(B \to \forall X.C) \;\mapsto\; (\forall X.B \to C), \;\; X \notin FV(A)$$

(renaming bound variables when needed) until this rule can no longer be applied (that is, all $\forall$s on the each level can be moved to the left), which allows us to write the resulting type as

$$D \equiv \forall X_{1\div m}.T_1 \to T_2 \to \dots T_n \to Z.$$

Definition 5. *Q-normal form. Given a total ordering on types (such an ordering can always be defined given an ordering on type variables) we say that a $\lambda^1\beta\eta$ type A written in the above notation is in Q-normal form iff $i < j$ when $T_i \leqslant T_j$ ($\leqslant$ signifies "total-ordering-less or as-types-equal") and each $T_{1\div n}$ is in Q-normal form.*

Definition 6. *S-normal form. Similarly, we say that the $\lambda^1\beta\eta$ type A written in the above notation is in S-normal form iff $i < j$ when*

$$\mathrm{Card}\{T \mid T \in T_{1\div n},\ T \equiv T_i\} < \mathrm{Card}\{T \mid T \in T_{1\div n},\ T \equiv T_j\}$$

in the relationship with f.h.p.'s, there are many other possibilities to define $e(A \to B)$ in such a way that no redexes are created, for example, as $K_{\to}(K_{-}A)(K_{+}B)$.

or $T_i < T_j$ when those are equal, and each $T_{1 \div n}$ is in S-normal form.

In other words, to transform a type into its Q-normal form one needs to recursively sort all premises using a supplied ordering, while to get its S-normal form one needs to first recursively sort arguments by the number of their occurrences and then if two different sub-formulas (on the same level) have a same number of occurrences, sort those using the ordering.

Examples 2. For the type

$$X \to Y \to X \to Z$$

assuming the conventional lexicographic ordering on type variable names

- $X \to X \to Y \to Z$ is the Q-normal form,
- $Y \to X \to X \to Z$ is the S-normal form.

Lemma 3. *A given $\lambda^1\beta\eta$ type has unique Q- and S-normal forms. There exist mechanical procedures to compute each of those normal forms. Those procedures produce isomorphisms.*

Proof. The procedures are defined as outlined above: apply to premises recursively, and either sort the results for the Q-normal form, or first count occurrences and then sort for the S-normal form.

Clearly, both procedures produce the unique solutions (up to the equality on types) because of the total orderings on types and natural numbers.

These procedures simply recursively reorder premises, hence they produce finite hereditary permutations, hence by theorem 4, they produce isomorphisms. □

We shall use notation $QNF(A)$ and $SNF(A)$ to denote the Q- and S-normal forms of A respectively.

Note that QNF and SNF, too, can be represented in a vector notation

$$A_{1 \div i_A} \to B_{1 \div i_B} \to \cdots \to Z$$

or, in plain programming terms, using the following algebraic datatype

$$F = List(\mathcal{N} \times F) \times Var$$

where Var signifies type variable name and the natural number signifies the number of occurrences of the corresponding F. Then, QNF's $List$ is sorted by F and SNF's $List$ sorted by $\mathcal{N}$ and then by F.

Theorem 5. *Two $\lambda^1\beta\eta$ types are isomorphic iff their Q-normal forms coincide. Two $\lambda^1\beta\eta$ types are isomorphic iff their S-normal forms coincide.*

Proof. Lemma 3 provides $A \sim QNF(A)$ and $B \sim QNF(B)$.

- If $A \sim B$, then $QNF(A) \sim QNF(B)$ by transitivity via completion of the diagram.
- If $QNF(A) \equiv QNF(B)$, then, trivially, $QNF(A) \sim QNF(B)$, and hence $A \sim B$ by transitivity via completion of the diagram.

Identically for SNF. □

Note that adapting these results to the $\lambda^2\beta\eta$ case is impossible: renaming of variables bound with $\forall$ can change the order of sub-formulas (which means Q- and S-normal forms stop respecting α-conversion), switching to the nameless (de Bruijn) notation solves that particular issue, but then the resulting normal form is still influenced by the order of quantification, which prevents the property analogous to the theorem 5 from working out. This is only natural as CCC-isomorphism test for Hindley-Milner is known to be graph-isomorphism-complete [3] while the result for $\lambda^1\beta\eta$ is polynomial [10].

Note however, that S-normal-form-like construction can still be useful even for the $\lambda^2\beta\eta$. Assuming Set is an "unordered $List$" (i.e. equality on Set is set equality), we can modify the above datatype as follows

$$F = Qs \times List(\mathcal{N} \times Set\ F) \times Var$$

with Qs signifying the natural number of $\forall$s this type introduces, Var now signifying de Bruijn index of the corresponding variable, and the Set accumulating sub-types of the same number of occurrences. Also note that in the systems that have kinds one has to apply the same structure and sorting method to $\forall$s too. Preprocessing a list of types into this form simplifies later isomorphism checking as one can rapidly reject many non-isomorphic pairs by inspecting only the spines ("S-" in "S-normal form" comes from "spine", and "Q-" comes from "quotient") of natural numbers of the resulting values and, most importantly, these spines stay constant under reordering of variables under $\forall$s. For simplicity, in the following sections we shall use the term "S-normal form" both for the $\lambda^1\beta\eta$ case and for the S-normal-form-like structures of the more general systems.

§4. Programming applications

In this section we outline several applications of the above facts to the programming languages and their tooling.

4.1. Proof Search. As we noted above, S-normal form is, essentially, a representation of a given type directly in a vector notation (with some restrictions). We can apply the same idea to terms thus producing a notation for vector-λ-calculus with types in S-normal form.

Let $\lambda x_{1\div n}$ signify the binding of n variables of the same type in a row to the vector of variables x (with the usual λx now being the syntax sugar for $\lambda x_{1\div 1}$) and let all variable occurrences use two-tier system of a variable name and an index (and similarly for $\forall$s and kinds in systems that have those). For example,

$$\lambda xyz.xyz : (A \to A \to B) \to A \to A \to B$$

becomes

$$\lambda x_{1\div 1}.\lambda y_{1\div 2}.x_1 y_1 y_2 : (A_{1\div 2} \to B)_{1\div 1} \to A_{1\div 2} \to B.$$

Now, extend this system by allowing variable occurrences to refer to any element of the corresponding vector using x_{any} notation. Terms with such $_{any}$-variables in this notation correspond to sets of terms in the conventional notation.

By "forgetting" that bindings are vectors we can straightforwardly adapt many proof search (automated theorem proving) algorithms that take a type as input and produce a term as output (e.g. [8,19]) by giving them the type in S-normal form with all numbers of occurrences stripped as input and reannotating the output as follows: bindings get their size annotations from the S-normal form, variables are annotated with $_{any}$ unless they are bound to vectors of size 1, in which case they are annotated with $_1$. For example, given a type

$$A \to B \to A \to A$$

we take its S-normal form

$$B_{1\div 1} \to A_{1\div 2} \to A$$

strip it

$$B \to A \to A$$

run an inference algorithm that produces a function (a set of functions) of this type

$$\lambda xy.y$$

and then reannotate this term (these terms) into

$$\lambda x_{1\div 1} y_{1\div 2}.y_{any}$$

thus producing a compact form for a set of terms satisfying the original type up to isomorphism.

Finally, note that unification algorithms [9, 16, 18], too, can be (somewhat less straightforwardly, as they have to handle the case when two or more sub-types have the same number of occurrences in the S-normal form) adapted to this representation: vectored arrows, $\forall$s and λs unify with similar vectored things of the same size (and sizes of consequent vectored things must monotonically increase), an $_{any}$-variable unifies with any index of the same binding.

In short, applying this idea to program inference and unification algorithms produces algorithms that solve the problems in question up to isomorphism almost for free with very little changes.

4.2. Typed Library Search. Types in functional programming languages can be used by specialized search tools like Hoogle [21] to search program libraries for functions [7, 22–24]. Early systems tried to unify the query type modulo isomorphism with all the types of functions available in the library. Modern systems have to deal with much larger libraries and so they usually use unification (if at all) on a pre-filtered set of candidates chosen by heuristics like close arity (i.e. query and result should have similar number of arguments) and high result rarity (results with common type signatures are less interesting). We feel that, especially when programming using very generic algebraic structures, the arity pre-filter frequently filters out useful results and hence the user frequently has query the system with non-trivial modifications of the original query (like adding new type variables into seemingly random places in the expression) to get something useful.

Preprocessing all the types in a library into S-normal form and then using those for pre-filtering (if not for the actual unification modulo isomorphism) gives much more control than simple arity comparisons. For instance, one can use edit distance on the spines of the S-normal form in question thus moving the system from an ad-hoc heuristic into a realm of more conventional search engines.

4.3. Rejecting Non-trivial Automorphisms. Finally, yet another interesting application is type checking that rejects function with types that have non-trivial automorphisms. Transforming a given type into its S-normal form reveals the structure of its automorphism group (see the next section), in particular, sub-types that have more than one occurrence in

S-normal form can be shuffled around. In some instances (like when a function of two arguments is commutative) this doesn't make a difference, but in other instances these functions can be "misused" by mistakenly applying arguments in the wrong order. We feel that enforcing this rule of "no non-trivial automorphisms" can be useful to force programmer diligence when designing a truly strictly-typed library API.

On one hand, it's hard to imagine a practical language with non-discriminatory application of this rule (use in select libraries does seem useful, though). On the other hand, in such a utopia of a programming language one can call functions by supplying arguments in arbitrary order.

§5. Automorphism Groups of Types

The necessary notions of the theory of groups, such as wreath product, may be found in [12], see also [27]. The following theorem characterizes the groups of automorphisms of types in $\lambda^1\beta\eta$. A detailed proof may be found in [26].

Theorem 6. *Let S_m denote the symmetric group on m elements, and $S_m|\wr G$ the wreath product of S_m and G. The groups* $\mathrm{Aut}(A)$ (*and* $\mathrm{Aut}^{op}(A)$) *are, up to group isomorphisms, exactly the groups that belong to the class W of finite groups defined inductively as follows:*

- $\{1\} \in W$;
- *if* $G, H \in W$ *then* $G \times H \in W$;
- *if* $G \in W$ *and* $m \geqslant 2$ *then* $S_m| \wr G \in W$.

Corollary 1. (*See* [1], *p.* 1457, *proposition* 1.15.) *The class of groups* $\mathrm{Aut}(A)$ (*considered up to group isomorphisms*) *where A are types of $\lambda^1\beta\eta$ coincides with the class of automorphisms of finite trees.*

The "representation power" of $\lambda^2\beta\eta$ and dependent type systems is much stronger. Every finite group may be represented as $\mathrm{Aut}(A)$ for some A in $\lambda^2\beta\eta$ or in LF where $\forall$ may be replaced by dependent product. This was proved in [26] (theorems 5.5 and 5.6). Here we present these results in a more refined form, that permits to obtain a recursive algorithm to compute the groups $\mathrm{Aut}(A)$ (in [26] no algorithm was proposed).

Let $V = \{X_{1 \div n}\} \subseteq FV(A)$. Let $\sigma(A)$ denote the result of substitution

$$[X_{\sigma(1)}/X_1, \ldots, X_{\sigma(n)}/X_n]A, \quad \sigma \in S_n.$$

One may consider the permutations $\Sigma_V(A) \subseteq S_n$ such that $\forall\sigma \in \Sigma_V(A).(\sigma(A) \sim A)$ and the groupoid $Gr_\Sigma(A) \subseteq Gr(A)$ whose objects are types

$A' \sim A$ such that $\exists \sigma \in \Sigma(A).(A' = \sigma(A))$ and morphisms are the same isomorphisms as in $Gr(A)$, so it is a full subcategory of $Gr(A)$.

Lemma 4. *For any $V \subseteq FV(A)$, $\Sigma_V(A)$ is a group w.r.t. the composition of permutations.*

Lemma 5. *Let $V = \{X_{1\div n}\} \subseteq FV(A)$. There exists a bijection between the group $\mathrm{Aut}(\forall X_{1\div n}.A)$ and the set of isomorphisms $M : A \to A'$ such that $\exists \sigma \in \Sigma_V.(\sigma(A) \equiv A')$.*

Theorem 7. *Let A be some type in $\lambda^1\beta\eta$ and let $\overline{\forall}.A$ denote its universal closure (the type in the second order calculus $\lambda^2\beta\eta$). Then the group of automorphisms $\mathrm{Aut}(\overline{\forall}.A)$ (in $\lambda^2\beta\eta$) is isomorphic to the cartesian product $\mathrm{Aut}(A) \times \Sigma(A)$.*

Theorem 8. *For every finite group G there exists some type A in $\lambda^1\beta\eta$ such that the group $\Sigma_{FV(A)}(A)$ is isomorphic to G. It is possible to construct A in such a way, that at the same time $\mathrm{Aut}(A) = \{id_A\}$.*

As in [26], the idea is to "model" the Cayley colored graph using the structure of the type. In this theorem in fact A may be quantifier-free.

Corollary 2. *For every finite group G there exists some type A in $\lambda^1\beta\eta$ such that the group $\mathrm{Aut}(\overline{\forall}.A)$ (in $\lambda^2\beta\eta$) is isomorphic to G.*

Using theorem 6 we can describe now (up to an isomorphism of groups) the group $\mathrm{Aut}(A)$ for any type A in and $\lambda^2\beta\eta$.

Theorem 9. *Any type A in $\lambda^2\beta\eta$ is isomorphic to a type A' of the form*

$$A' = \forall X_{1\div n}.A_{1\ 1\div i_1} \to \ldots A_{m\ 1\div i_m} \to Z$$

where Z and $X_{1\div n}$ are type variables, the types in each list A_k are identical, types in different lists are not isomorphic, and each $A_{k1\div ki_k}$ are in this form too. In particular, any type A in $\lambda^2\beta\eta$ is isomorphic to its Q-normal form (-like stucture). The groups $\mathrm{Aut}(A)$ and $\mathrm{Aut}(A')$ are isomorphic as groups, and $\mathrm{Aut}(A')$ is obtained from symmetric groups by combination of wreath products, cartesian products and cartesian products with $\Sigma_V(-)$.

In case of LF the universal quantification may be modeled by dependent product. Instead of $\forall X.B(X)$ we write $(X : Type)B(X)$, and we can obtain similar results about the representation of finite groups in LF.

§6. Security Applications

In this section we present several examples of possible practical applications of the results described above.

6.1. Encoding Conventional Cryptography on Finite Groups. For illustrative purposes, let us recall the description of the **ElGamal** cryptosystem [15, 17]:

- Private Key: $m, m \in N$.
- Public Key: g and g^m.
- Encryption: To send a message a Bob computes g^r and g^{mr} for a random $r \in N$. The ciphertext is $(g^r, g^{mr}a)$.
- Decryption: Alice knows m, so if she receives the ciphertext $(g^r, g^{mr}a)$, she computes g^{mr} from g^r, then $(g^{mr})^{-1}$, and then computes a from $g^{mr}a$.

We can use the results described in the previous sections to emulate this protocol in Type Theory:

- select some base type A and the message $a : A$,
- represent g as a distinguished automorphism $g : A \to A$ (f.h.p.),
- represent g^m as $g \circ \cdots \circ g$ (m times), and similarly for g^r and g^{mr},
- run the ElGamal protocol as normal.

By encoding a finite cyclic group of prime order as a group of automorphism of some type we can implement ElGamal (or any other cryptographic protocol based on finite groups) since the composition and inverse of type automorphisms (represented by finite hereditary permutations [7]) can be computed in linear time.

We do not consider here the cryptosystems like **MOR** based on a more sophisticated group theory [17] but they, too, can be represented in type theory using the results of [26].

It is fairly clear that these type-based implementations are going to be less efficient than an equivalent long integer-based ones, which would make them less desirable for conventional applications. But that alone can make them more desirable for other uses like **proof-of-work** algorithms.

Also of note is the fact that these encryption schemes preserve the structure of $a : A$. Which, for instance, means that Alice needs not typecheck the decrypted a if she trusts Bob to typecheck his.

6.2. More General Ideas. Consider some type S. The closed terms $F : S$ represent combinators that may take other terms as arguments.

A possible meaning is that F combines in some way (opaque to external users) the operators and data. All its parameters may be fed externally in a controlled way.

For example, let

$$S \equiv (X_1 \to X_1') \to \dots (X_n \to X_n') \to X_1 \to \dots X_n \to X.$$

Here $X_1, \dots, X_n', X$ are not necessarily different and may be type variables or constants). The terms $F : S$ represent combinators that may take any functions $\phi_1 : X_1 \to X_1', \dots \phi_n : X_n \to X_n'$ and data $\chi_1 : X_1, \dots, \chi_n : X_n$ as arguments.

If we take

$$F \equiv \lambda f_{1 \div n} : X \to X \lambda x : X.(f_{\sigma(1)}(\dots (f_{\sigma(n)} x) \dots)$$

where σ is some permutation of $\{1, \dots, n\}$, it will combine the applications of $\phi_{1 \div n}$ in any desired order. If we take $F \equiv \lambda f_{1 \div n} : X \to X \lambda x : X.f_i x$ then only one of ϕ will be selected, etc. The $\phi_{1 \div n}$ themselves may be, for example, some coding functions.

In an extension of λ-calculus with inductive types and induction-recursion F may include recursion operators and the functions $\phi_1, \dots, \phi_n$ and data $x_1, \dots, x_n$ be the parameters of recursion.

The type S of the combinator F may have many automorphisms which form a subset of all possible isomorphisms to/from this type. Automorphisms, in difference from isomorphisms, do not change the types of parameters (in a given order) that F can be applied to. So, if an automorphism $\theta : S \to S$ is applied to F then $\theta(F)\phi_1 \dots \phi_n \, x_1 \dots x_n$ is valid iff $F\phi_1 \dots \phi_n \, x_1 \dots x_n$ is valid. In difference from automorphisms, an action of an isomorphism $\theta' : S \to S'$ (which is not an automorphism) may make invalid an application of $\theta'(F)$.

The terms and types above belong to $\lambda^1\beta\eta$ possibly extended with inductive types. In $\lambda^2\beta\eta$ we may add a second-order λ and consider

$$\lambda X_1 \dots X_n \, X_1' \dots X_n'.F : \forall X_1 \dots X_n \, X_1' \dots X_n'.S.$$

In this way the types also become controlled parameters, for example one may "feed" Nat, $Bool$ or other types for variables.

If dependent types are admitted, the types X themselves may depend on terms as parameters. A standard example is the type of Boolean vectors of the length $n : Nat$.

The information about the distinction of type variables may be hidden from an intruder and this will permit to use type-flaw detection methods [14] to detect the attacks.

In general S may contain subtypes of any possible form, including constant types, the types of many-variable functions $Y_{1\div i} \to Y$, higher-order functions like $(X \to Y) \to Z$ etc. It may be seen as the type of combinators that assemble a program from the program modules of these types. The automorphisms, since they do not change the type globally, may be used to hide the exact purpose of each module (to "mix" the program modules of the same type).

Examples 3. Consider a family of functions (represented by some closed terms in LF with inductive types) $\{f(n) : G(n) \to H(n)\}$ with $n : Nat$ and $G(n), H(n) : Type$. We may merely consider one function $\{f : (n : Nat)(G(n) \to H(n)\}$ with $G, H : Nat \to Type$. The $f(n)$ may be isomorphisms for some n. Let F be the type of data to be encoded. Consider the following closed term:

$$[n : Nat]f(n) : G(n) \to H(n).$$

Let $k : Nat$ be the smallest k that $G(k) = F$ and $f(k)$ is an isomorphism (we assume that it exists). Then $f(k)$ is the coding function.

We assume also that there exists the smallest $n > k$ such that $G(n) = H(k)$ and $H(n) = F$, and $f(n)$ is the inverse isomorphism for $f(k)$.

To decode the data one has to find n. Obviously this schema permits to represent many cryptosystems, including the ElGamal considered above.

§7. An Outline of a Quantitative Analysis

Let A be a type. The cardinality $|\operatorname{Aut}(A)|$ and the number q of types that are isomorphic to A (including A) are finite. From lemma 2 and theorem 2 it follows that the number of isomorphisms $A \to \ldots$ (or $\cdots \to A$) that are not automorphisms is merely $|\operatorname{Aut}(A)| \cdot (q-1)$.

Consider first the calculus $\lambda^1\beta\eta$. Without loss of generality we may assume that $A \equiv A_{11\div 1i_1} \to \ldots A_{m1\div mi_m} \to X$. Let $n = i_1 + \cdots + i_n$. Taking into account the definition of wreath product [12], the size of symmetry groups and that $|\operatorname{Aut}(A_{k1})| = |\operatorname{Aut}^{op}(A_{k1})|$, we obtain the following recursive formula:

$$|\operatorname{Aut}(A)| = i_1! \cdot |\operatorname{Aut}(A_{11}))|^{i_1} \cdot \cdots \cdot i_m! \cdot |\operatorname{Aut}(A_{m1})|^{i_m}.$$

The number of types that are isomorphic but not identical to A is given by

$$\frac{n!}{i_1! \dots i_m!} \cdot |\operatorname{iso}(A_{11})|^{i_1} \cdot \dots \cdot |\operatorname{iso}(A_{m1}|^{i_m} - 1.$$

In $\lambda^2\beta\eta$ there will be more isomorphisms due to possible permutations in the $\forall$-prefix (each premutation defines an isomorphism), but also the equality of types will become non-trivial because of α-conversion in types, and it will augment the number of automorphisms. In case of LF there is more constraints (not all "premises" of a dependent product can be permuted because of dependencies), so this can reduce both the number of isomorphisms and automorphisms. However the choice of type structure remains very flexible and permits to control the structure of the groupoid of isomorphisms and the automorphism groups.

§8. Conclusion

Some more questions must be resolved to make some of the ideas outlined in this paper more practical. Presented algorithms should be implemented in a code base ready for public consumption, precise communication protocols that would make use of the distinction between iso – and automorphisms, public and private type information, should be elaborated. The complexity of algorithms (for example, for reconstruction of typed isomorphisms from erasure) needs to be investigated much more precisely.

Moreover, we believe that the use of type theory and λ-calculus as a higher-level formal language for data protection (especially software protection) and detection of attacks deserves to be investigated further.

References

1. L. Babai, Automorphism Groups, Isomorphism, Reconstruction. In Handbook of Combinatorics. Elsevier. **2** (1995), 1447–1541.
2. H. Barendregt, *The Lambda Calculus; Its Syntax and Semantics* (*revised edition*). North-Holland Plc. (1984).
3. D. A. Basin, Equality of Terms Containing Associative-Commutative Functions and Commutative Binding Operators is Isomorphism Complete. In M. E. Stickel Ed., lOth Int. Conf. on Automated, Kaiserslautern, Germany, of Lecture Notes in Artificial Intelligence, Springer-Verlag **449** (1990), 251–260.
4. R. Brown, Ph. G. Higgins, R. Sivera, *Nonabelian Algebraic Topology.* — European Math. Soc. (2011) .

5. K. Bruce, G. Longo, *Provable isomorphisms and domain equations in models of typed languages*. In: ACM symposium on theory of computing (STOC 85) (1985), 263–272.
6. M. Dezani-Ciancaglini, *Characterization of normal forms possessing inverse in the* $\lambda - \beta - \eta$*-calculus*. — Theor. Comp. Scie., **2** (1976), 323–337.
7. R. Di Cosmo, *Isomorphisms of types: from lambda-calculus to information retrieval and language design*. Birkhauser (1995).
8. D. Gilles, *A complete proof synthesis method for the cube of type systems*. — J. Logic Comp., **3(3)** (1993), 287–315.
9. D. Gilles, *Higher-order unification and matching*. In Alan Robinson and Andrei Voronkov, editors, Handbook of automated reasoning, Elsevier Science. **2**, No. 16 (2001), 1009–1062.
10. J. (Yossi) Gil, Y. Zibin, *Efficient Algorithms for Isomorphisms of Simple Types*. — Math. Struct. Comp. Science (2003).
11. H. Goguen, *A Typed Operational Semantics for Type Theory*. PhD thesis, University of Edinburgh, 1994.
12. M. Hall, (Jr.) *The Theory of Groups*. The Macmillan Company (1959).
13. Chr. Hankin, *Lambda Calculi: A Guide for Computer Scientists*. Clarendon Press, Oxford (1994).
14. J. Heather, G. Lowe, S. Schneider, *How to prevent type flaw attacks on security protocols*. — J. Comp. Sec., **11(2)** (2003), 217–244.
15. J. Hoffstein, J. Pipher, J. H. Silverman, *An introduction to mathematical cryptography*. Springer, New York, 2008.
16. G. Huet, C. Derek Oppen, Equations and Rewrite Rules: A Survey. Technical report. Stanford University (1980).
17. A. Mahalanobis, *The MOR cryptosystem and finite p-groups*. — Contemp. Math., **633** (2015), 81–95.
18. A. Martelli, U. Montanari, *An Efficient Unification Algorithm*. — ACM Trans. Program. Lang. Syst., **4 (2)** (1982), 258–282.
19. F. Lindblad, M. Benke, A Tool for Automated Theorem Proving in Agda. In: Paulin-Mohring C., Werner B. (eds) Types for Proofs and Programs. TYPES 2004. Lecture Notes in Computer Science, vol 3839. Springer, Berlin, Heidelberg (2006).
20. Z. Luo, *Computation and Reasoning*. — Int. Series Monogr. Comp. Sci., Oxford Science Publications, Clarendon Press, Oxford, UK, **11** (1994).
21. Neil Mitchell et al. Hoogle: Haskell API search engine. `https://github.com/ndmitchell/hoogle`.
22. M.Rittri, *Using Types as Search Keys in Function Libraries*. — Proceedings of the fourth international conference on Functional programming languages and computer architecture (1989), 174–183.
23. M. Rittri, *Retrieving library functions by unifying types modulo linear isomorphism*. Theor. Inform. Applic., (1992).
24. C. Runciman, I. Toyn, *Retrieving reusable software components by polymorphic type*. — J. Func. Progr. **1 (2)** (1991), 191–211.
25. S. Soloviev, *On Isomorphism of Dependent Products in a Typed Logical Framework*. — Post-proceedings of TYPES 2014, LIPICS, Schloss Dagstuhl - Leibniz-Zentrum für Informatik. **39** (2015), 275–288.

26. S. Soloviev, *Automorphisms of Types in Certain Type Theories and Representation of Finite Groups.* — Math. Struct. Comp. Sci., (2018).
27. A. T. White, *Graphs, Groups and Surfaces.* — North-Holland, Amsterdam (1984).

IRIT, Paul Sabatier University,
118 route de Narbonne 31062
Toulouse, France;
ITMO University,
St.Petersburg, Russia
E-mail: soloviev@irit.fr

IRIT, Paul Sabatier University,
118 route de Narbonne 31062
Toulouse, France;
ITMO University,
St.Petersburg, Russia
E-mail: papers@oxij.org

Поступило 20 сентября 2018 г.