Algorithms for Extended Alpha-Equivalence and Complexity*

Manfred Schmidt-Schauß, Conrad Rau, and David Sabel

Goethe-Universität, Frankfurt, Germany
{schauss,rau,sabel}@ki.informatik.uni-frankfurt.de

Abstract

Equality of expressions in lambda-calculi, higher-order programming languages, higher-order programming calculi and process calculi is defined as alpha-equivalence. Permutability of bindings in let-constructs and structural congruence axioms extend alpha-equivalence. We analyse these extended alpha-equivalences and show that there are calculi with polynomial time algorithms, that a multiple-binding "let" may make alpha-equivalence as hard as finding graph-isomorphisms, and that the replication operator in the pi-calculus may lead to an EXPSPACE-hard alpha-equivalence problem.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems, F.4.2 Grammars and Other Rewriting Systems

Keywords and phrases alpha-equivalence, higher-order calculi, deduction, pi-calculus

Digital Object Identifier 10.4230/LIPIcs.RTA.2013.255

1 Introduction

Motivation. Reasoning, rewriting, matching, and automated deduction in higher order calculi often require – as a very basic operation – to identify higher-order expressions *up to alpha-equivalence*. This means expressions are identified if they are syntactically equal up to a renaming of bound variables (which represent the binding structure). As a basic example consider the expressions of the classical lambda calculus $e_1 = \lambda x.\lambda y.x$ and $e_2 = \lambda y.\lambda x.y$. These expressions are alpha-equivalent, since the renaming $\sigma = \{y \mapsto x, x \mapsto y\}$ of *bound* variables makes $\sigma(e_1)$ and e_2 syntactically equally. An approach to handle alpha-equivalence in deduction systems is to use nominal techniques [27], however, the focus is to ease formula specification and deduction rather than speeding up alpha-equivalence checking. In addition nominal techniques consider so-called equivariance between terms, which is a slight extension of alpha-equivalence, since terms e_1, e_2 are equivariant if there exists a finite permutation of variable names π such that e_1 is syntactically equal to πe_2.

For a lot of classical program calculi (e.g. several variants of extensions of the lambda calculus) checking alpha-equivalence can be performed by efficient (and also more or less trivial) algorithms in log-linear time in the size of the expressions. Also deciding equivariance of such terms is known to be in **P** [9].

However, more sophisticated calculi also allow programming primitives that satisfy laws like commutativity and / or associativity in combination with binding primitives, like non-recursive and recursive bindings, which may occur nested (e.g. `let(rec)`-expressions in extended lambda-calculi like [2, 23, 38], the parallel composition and ν-binders in process calculi, like the π-calculus [21, 20, 35]). Equality testing of expressions in such calculi,

* This work was supported by the DFG under grant SCHM 986/9-1 and SCHM 986/9-2

respecting alpha-equivalence and the laws of the primitives, turns out to have a harder decision problem.

In this paper we focus on this problem and describe several algorithms and also determine the complexity of checking *equality of expressions* up to alpha-equivalence under the usual laws of the programming primitives (permutativity, commutativity, associativity, etc.). Algorithms for expressions in higher-order extended lambda-calculi and process-calculi will be discussed, which also includes alpha-equivalence of functional programs in Haskell.

Applications. Our motivation to analyze extended alpha-equivalence stems from recent research that aims to automate the diagram-based proof method for showing correctness of program transformations (see [38] for the method and [29, 28] for the automation). The method is mainly used for call-by-need programming calculi modelling the semantics of functional programming languages like Haskell, however, the method is also applicable to other kinds of calculi like variations of call-by-need calculi, and to process calculi like the π-calculus. The first step is to compute critical overlaps (similar to critical pairs) between reduction rules from the operational semantics and program transformations using a sophisticated unification algorithm [29, 30]. In a second step the overlappings must be "closed", which is similar to show joinability of critical pairs. This requires to find out whether two reduction sequences starting from different expressions lead to expressions that are alpha-equivalent after permutation of bindings. Thus checking expressions for extended alpha-equivalence is an operation that is often performed even on large expressions. Ad-hoc algorithms for checking alpha-equivalence of such expressions are worst-case exponential due to searching for all possible permutations. Indeed, we will show in this paper that this is unavoidable in general. Another potential application of interest in program analysis during compilation of (functional) programs is in "common subexpression elimination" which shares identical subexpressions and where subexpressions must be checked for equality up to alpha-equivalence.

Graph isomorphism. Several extended alpha-equivalence problems will be shown to be graph-isomorphism-complete (**GI**-complete) in this paper. The graph-isomorphism problem as a complexity class **GI** is only known to be between **PTIME** and **NP**. Proving an algorithmic problem as **GI**-complete indicates hardness and (assuming current knowledge) that there is no polynomial time algorithm for it solving all instances. More details from a complexity point of view on the class **GI** can e.g. be found in [39, 16, 15], where it is also show that if **GI** were **NP**-complete, then the polynomial time hierarchy would collapse.

A related problem is term equality including associative-commutative operators which is shown to be **GI**-complete in [6]. However, [6] does not consider the case of (perhaps mutually-recursive) binding-environments as they are provided by the `letrec`-construct. Another result is that deciding structural congruence in the π-calculus *without replication* (but perhaps with recursion) is **GI**-complete [14]. Algorithms for deciding structural congruence in the π-calculus *with replication* were investigated for several variants of the calculus in [10, 11], where the complexity is either not analyzed or shown to be in **EXPSPACE**. As we show – under mild restrictions – the problem is also **EXPSPACE**-hard. A further related result is that syntactic equivalence of Boolean formulas in CNF using associativity/commutativity of Boolean operators is **GI**-complete [5].

Results and structure of the paper. Section 2 contains preliminaries on graphs, the isomorphism problem of various variants of graphs, and a proof of **GI**-completeness for

a special class of graphs (so-called outgoing-ordered labelled directed graphs). Moreover, an efficient decision procedure is presented for a subclass of these graphs. Based on these results, we prove in Section 3 that alpha-equivalence for a class of higher-order languages with letrec-expressions is **GI**-complete (Theorem 3.8), and that there is a polynomial time algorithm provided the expressions are free of garbage (Theorem 3.11) or rewrite rules for performing garbage collection are included in the calculus (Theorem 3.13). We also show **GI**-completeness for languages with non-recursive or non-nested let. We also present several instances of program calculi that are covered by our results. In Section 4 we investigate structural congruence in process calculi. Especially, we summarize known results about the complexity of structural congruence in several variations of the π-calculus and provide a proof of **EXPSPACE**-hardness of structural congruence of Milner's variant (Theorem 4.3).

2 Graphs and Graph Isomorphism

Before considering the (extended) alpha-equivalence problems, as a preliminary we introduce the necessary notions and notation on graphs and the graph isomorphism problem. In this section, we also introduce some specific, restricted graphs and their isomorphism problems together with algorithms and analyses of them. In later sections, these results will be applied to the alpha-equivalence problem in different program calculi.

We define labelled directed graphs as a flexible formalism for several classes of graphs, e.g. including unlabelled, undirected graphs.

▶ **Definition 2.1.** A *labelled directed graph (LDG)* is a tuple $G = (V, E, L, \text{lab})$ where V is a finite set of nodes, $E \subseteq (V \times V \times L)$ are directed labelled edges between nodes, L is a finite set of labels, V, E are disjoint, and $\text{lab} : V \to L$ assigns a label to every node. If $(v_1, v_2, l) \in E \iff (v_2, v_1, l) \in E$, then the graph is *undirected*, and if $|L| = 1$, then we call G *unlabelled*. For convenience, we omit the components L and lab for unlabelled graphs.

Note that this definition does only allow parallel edges with different labels, and forbids parallel edges in unlabelled graphs. An isomorphism between two LDGs is defined as follows:

▶ **Definition 2.2** (Isomorphic LDGs)**.** Two LDGs $G_1 = (V_1, E_1, L, \text{lab}_1)$, $G_2 = (V_2, E_2, L, \text{lab}_2)$ are *isomorphic* iff there is a bijection $\phi : V_1 \to V_2$ such that $(v_1, v_2, l) \in E_1 \iff (\phi(v_1), \phi(v_2), l) \in E_2$ for all edges $(v_1, v_2, l) \in E_1$ and $\text{lab}_1(v) = \text{lab}_2(\phi(v))$ for all nodes $v \in V_1$. The mapping ϕ is called an *isomorphism* in this case.

Note that an isomorphism is completely determined by the mapping on the nodes.

▶ **Definition 2.3** (Graph Isomorphism Problem (GI))**.** Graph-isomorphism (**GI**) is the following problem: Given two finite (unlabelled, undirected) graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, are G_1 and G_2 isomorphic?

It is well-known that the isomorphism problem for *directed* graphs is **GI**-complete [41], which we will use later for our encodings. Also, the labelled directed graph isomorphism problem is equivalent to the unlabelled graph isomorphism problem, i.e. it is known to be **GI**-complete. See [7] for further classes of graphs with a **GI**-complete isomorphism problem. We summarize these known results in the following proposition:

▶ **Proposition 2.4** ([7, 41])**.** *The isomorphism problem for LDGs and the isomorphism problem for unlabelled directed graphs are* **GI***-complete.*

Figure 1 Example for the encoding ρ in the proof of Proposition 2.6 (new nodes are shaded).

The graph-isomorphism problem as a complexity class **GI** is only known to be between **PTIME** and **NP**.

In our application in the next section, the following specific graphs play an important role. The graphs are LDGs where all outgoing edges of a node have unique labels:

▶ **Definition 2.5** (Outgoing-Ordered LDG)**.** We call a labelled directed graph $G = (V, E, L, \text{lab})$ *outgoing-ordered* (an *OOLDG*), iff for every node $v \in V$: whenever $(v, v_1, l), (v, v_2, l) \in E$ then $v_1 = v_2$.

OOLDGs are related to expressions with a restricted use of `let`-expressions, where the intuition is that their alpha-equivalence-check may be more efficient.

The paper [12] describes algorithms for matching directed labelled graphs, but the isomorphism definition is w.r.t. the (infinite) unrolling, and thus different. The isomorphism problem for so-called ordered directed graphs was shown to be solvable in polynomial time by [13]. However, this result is only applicable to labelled directed graphs where for every node the – outgoing as well as the incoming – edges are uniquely labelled for every node. For our application, the incoming edges have no restrictions. Hence their result cannot be used. The following proposition even shows that these questions are different:

▶ **Proposition 2.6.** *The isomorphism problem for OOLDGs is* **GI***-complete.*

Proof. The problem is in **GI** (this trivially follows from Proposition 2.4). For proving **GI**-hardness, we define an encoding ρ of arbitrary unlabelled directed graphs into OOLDGs: Given an unlabelled directed graph G, we construct an OOLDG $\rho(G)$ from G by the following operation: Every edge (v_1, v_2) of G is replaced by two edges $(v, v_1, 1)$ and $(v, v_2, 2)$, where v is a new node: i.e. $v_1 \to v_2$ is replaced by $v_1 \xleftarrow{1} v \xrightarrow{2} v_2$, where for every edge a fresh node v is constructed. An example of the encoding is shown in Figure 1.

Given two unlabelled directed graphs G_1, G_2, these are isomorphic iff $\rho(G_1)$ and $\rho(G_2)$ are isomorphic: Any isomorphism of $\rho(G_1)$ and $\rho(G_2)$ only maps old nodes to old ones and fresh nodes to fresh ones due to the direction of the new edges (i.e. only new nodes have outgoing edges, and only old nodes have incoming edges). The direction of the edges is preserved (i.e. encoded) due to the labels $1, 2$. Since the encoding ρ at most doubles the size of the graphs and the isomorphism problem for (unlabelled) directed graphs is **GI**-complete (Proposition 2.4), OOLDG-isomorphism checking is **GI**-hard. ◀

Since the encoding generates an *acyclic* directed graph, this also implies:

▶ **Proposition 2.7.** *The isomorphism problem for acyclic OOLDGs is* **GI***-complete.*

However, as we show in the remainder of this section, there are also some complexity results for OOLDGs, which are positive under various reachability restrictions. First, we define some notation:

▶ **Definition 2.8.** Let $G = (V, E, L, \text{lab})$ be a (labelled or unlabelled) directed graph. We say a node $v \in V$ is a *root*, if it has no incoming edges (there exists no edge $(w, v) \in E$), and it is a *leaf* if it has no outgoing edges (there exists no edge $(v, w) \in E$).

For a node v of G, let $\text{reach}(v, G)$ be the set of nodes in G reachable from v via directed edges. A node v is called *initial*, iff every other node $w \in V \setminus \{v\}$ can be reached from v (i.e. $w \in \text{reach}(v, G)$). A set S of nodes is *initial*, iff every other node $w \in V \setminus S$ can be reached from some $v \in S$ (i.e. $w \in \bigcup_{v \in S} \text{reach}(v, G)$).

G is *weakly connected*, iff its corresponding undirected graph is connected. Hence G can be split into its weakly connected components. We say G is *k-initial*, if for every weakly connected component G' of G, there is an initial set $S_{G'}$ within G' of at most k nodes (i.e. $|S_{G'}| \leq k$). We say G is *k-rooted*, if every weakly connected component G' of G has at most k roots, and if the set of roots within G' is initial. We say G is *rooted*, if it is 1-rooted, and G is weakly connected. For an LDG $G = (V, E, L, \text{lab})$ its *size* $|G|$ is the sum of the cardinalities of V and E.

▶ **Proposition 2.9.** *Let G_1, G_2 be rooted OOLDGs. Then isomorphism between G_1 and G_2 can be tested in time $O(n \log(n))$ where $n = |G_1| + |G_2|$. Moreover, there is at most one isomorphism between G_1 and G_2.*

Proof. Let $G_i = (V_i, E_i, L_i, \text{lab}_i)$ for $i = 1, 2$. W.l.o.g, we can assume that for $i = 1, 2$: every $l \in L_i$ is used in G_i. If $L_1 \neq L_2$, then G_1 and G_2 are not isomorphic, hence we assume $L_1 = L_2$. We construct a mapping $\phi : V_1 \to V_2$ such that either ϕ is an isomorphism between G_1 and G_2 or the construction fails. In the latter case G_1 and G_2 are not isomorphic. Let r_i be the root nodes of G_i, for $i = 1, 2$. We set $\phi(r_1) := r_2$. We iteratively process all nodes in V_1: Assume $\phi(v_1) = w_1$ is already constructed, and the outgoing edges are $(v_1, v_{1,j}, l_j)$ for $j = 1, \ldots, m$. Then the outgoing edges from w_1 either can be ordered as $(w_1, w_{1,j}, l_j)$ for $j = 1, \ldots, m$, or the ϕ-construction fails and G_1, G_2 are not isomorphic. We set $\phi(v_{1,j}) := w_{1,j}$ for $j = 1, \ldots, m$. If there is a conflict, for example since ϕ is already differently defined on some $v_{1,k}$, then again the construction fails, and G_1, G_2 are not isomorphic. If the construction goes through without failing, then an isomorphism has been found, since the graphs are rooted, and hence every node is reachable. Obviously, the construction leads to a unique ϕ, if the construction halts successfully. As a preprocessing we store all edges (w_1, w_2, l_j) of E_2 in an efficient data structure with key (w_1, l_j) and value w_2. This can be done in $O(|E_2| \log |E_2|)$ time. During the construction of ϕ we lookup the corresponding edge in $O(\log |E_2|)$ time. The mapping ϕ can also be stored in an efficient data structure, which shows that the whole procedure requires $O(n \log n)$ time. ◀

▶ **Proposition 2.10.** *Let G_1, G_2 be OOLDGs with $n = |G_1| + |G_2|$. Estimations for the complexity of isomorphism checking of G_1, G_2 are:*

1. *$O(k!\ n \log(n))$, if G_1, G_2 are weakly connected and k-rooted.*
2. *$O(k!\ n^3 \log(n))$, if G_1, G_2 are k-rooted.*
3. *$O(k!\ n^2 \log(n))$, if G_1, G_2 are weakly connected and k-initial.*
4. *$O(k!\ n^4 \log(n))$, if G_1, G_2 are k-initial.*

Proof. The first item for $k = 1$ is exactly Proposition 2.9. For $k > 1$, all possible bijections of the roots have to be tried, which justifies the factor $k!$. The complexity in the second item is derived from the previous item, since $O(n^2)$ isomorphism checks between the weakly-connected components have to be performed followed by a comparison of the equivalence classes. Since the initial nodes are not unique in contrast to the roots, also all possibilities for other nodes have to be tried, which increases the exponent of n by 1 in the corresponding cases (3) and (4). ◀

▶ **Definition 2.11.** For an LDG $G = (V, E, L, \text{lab})$, we define the *outgoing-ordered subgraph (OO-subgraph)* $OO(G) = (V, E', L, \text{lab})$, which is constructed from G by removing all the edges with ambiguous labels. More rigorously: for every node $v \in V$, and every label $l \in L$, if $e_1, \ldots, e_n$ are all the outgoing edges from v, labelled with l, and $n \geq 2$, then remove $e_1, \ldots, e_n$, but not the nodes. The resulting graph is denoted as $OO(G)$.

▶ **Proposition 2.12.** *Let G_1, G_2 be two LDGs such that $G_i' := OO(G_i)$ for $i = 1, 2$ are rooted. Then isomorphism of G_1, G_2 can be tested in time $O(n \log(n))$ where $n = |G_1| + |G_2|$. Moreover, there is at most one isomorphism.*

Proof. Any isomorphism ϕ between G_1 and G_2 is also an isomorphism between G_1' and G_2', since $OO(\cdot)$ does not remove nodes. Moreover, any isomorphism ϕ between G_1' and G_2' is a bijective mapping between the nodes of G_1 and G_2. Using Proposition 2.9, the isomorphism test of G_1', G_2' can be performed in time $O(n \log(n))$, where the mapping ϕ is also constructed on the fly. Then we test whether ϕ is an isomorphism of G_1, G_2 which can also be performed in time $O(n \log n)$. ◀

3 Alpha-Equivalence for Higher-Order Languages with Let

We define a fragment of the core language of higher order extended lambda-calculi with a recursive or non-recursive let that captures the essence at least of its alpha-equivalence issues and is nevertheless general enough such that the results grade up to the full language.

A *signature* Σ is a finite set of ranked constructor or function symbols c, c_i equipped with an arity $ar(c) \in \mathbb{N}_0$. We assume a countable infinite set of variables denoted by x, y, z (possibly indexed by numbers). The language CH (over the signature Σ) has the syntax

$$r, s, t \in \mathcal{L}_{\mathsf{CH}} ::= x \mid c(s_1, \ldots, s_{ar(c)}) \mid \lambda x.s \mid \texttt{letrec}\ x_1 = s_1; \ldots; x_n = s_n\ \texttt{in}\ s.$$

The constructs that bind variables are abstractions (λ-expressions) and `letrec`-expressions. In a letrec-construct $\texttt{letrec}\ x_1 = s_1; \ldots; x_n = s_n\ \texttt{in}\ s$, the bindings $x_i = s_i$ may be interchanged; the variables $x_1, \ldots, x_n$ must be distinct; the scope of the bound variables x_i is in every $s_1, \ldots, s_n$ as well as in s. As an abbreviation we use *Env* (as a meta symbol) for a (non-empty) set of `letrec`-bindings, e.g. we sometimes write $\texttt{letrec}\ Env\ \texttt{in}\ s$, or also $\texttt{letrec}\ Env_1, Env_2\ \texttt{in}\ s$. As another abbreviation we use $\lambda x_1, \ldots, x_n.s$ instead of $\lambda x_1. \ldots . \lambda x_n.s$. For a CH-expression s we use $FV(s)$ for the set of *free variables* of s.

Several usual programming language constructs of extended lambda calculi like application, `seq`-expression, non-deterministic choice fit into this syntax by choosing a corresponding function symbol c for the construct. Also calculi with case-expressions are covered, where the usual case-expression $\texttt{case}\ s\ \texttt{of}\ (c_1(x_{1,1}, \ldots, x_{1,ar(c_1)}) \to t_1)\ \ldots\ (c_n(x_{1,1}, \ldots, x_{n,ar(c_n)}) \to t_n)$ is represented as $\texttt{case}(s, \lambda x_{1,1}, \ldots, x_{1,ar(c_1)}.t_1, \ldots, \lambda x_{n,1}, \ldots, x_{n,ar(c_n)}.t_n)$. *Alpha-equivalence* of CH-expressions is defined as follows and includes permutation of the letrec-bindings:

▶ **Definition 3.1.** Two CH-expressions s, t are *alpha-equivalent*, $s \simeq_{\alpha,\mathsf{CH}} t$ iff s can be transformed into t using perhaps several of the two operations: (i) renaming of bound variables without capture, (ii) permuting the bindings in the letrec-environments.

For example, $(\texttt{letrec}\ x = y; y = z\ \texttt{in}\ x) \simeq_{\alpha,\mathsf{CH}} (\texttt{letrec}\ x = z; y = x\ \texttt{in}\ y)$, where we have to exchange x, y by a substitution and then to permute the bindings in the environment.

The language CHNR is the same language as CH except that `letrec` is non-recursive. For convenience, in this case we write `let` instead of `letrec`. The scope of x_i in $\texttt{let}\ x_1 =$

$s_1; \ldots; x_n = s_n$ `in` t is only t, and in CHNR the alpha-equivalence is defined accordingly, where only the let-renamings are different due to the scoping.

It is known that alpha-equivalence of expressions in the lambda-calculus and its extensions without permutations of bindings is decidable in polynomial time, even in time $O(n \log n)$, where n is the size of the expressions [8].

The issue when checking alpha-equivalence of CH-expressions is the potential exponential time requirement in searching for all possible permutations of letrec-bindings. It is easy to see that the alpha-equivalence problem is in **NP**: A single guess for the permutation of every letrec-expression and then checking alpha-equivalence in polynomial time are sufficient.

3.1 Complexity in the General Case of CH and CHNR

First we show that alpha-equivalence in CH is **GI**-hard.

▶ **Proposition 3.2.** *Even if the signature Σ is empty, deciding* CH*-alpha-equivalence as well as* CHNR*-alpha-equivalence is* **GI***-hard.*

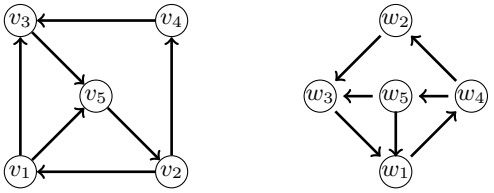
Proof. We encode the graph isomorphism problem for unlabelled directed graphs into the CHNR-alpha-equivalence problem. Hence, we encode two arbitrary unlabelled directed graphs $G_i = (V_i, E_i)$ for $i = 1, 2$ into CHNR-expressions $exp(G_i)$. We first assume that Σ contains a constant a and a binary constructor c, later we cover the case that Σ is empty. W.l.o.g. we assume $V_1 = \{v_{1,1}, \ldots, v_{1,n}\}$ and $V_2 = \{v_{2,1}, \ldots, v_{2,n}\}$ such that $V_1 \cap V_2 = \emptyset$.

The expression $exp(G_i)$ is `let` $Env_{i,A}$ `in` (`let` $Env_{i,B}$ `in` a), where the environments are as follows: For every node $v_{i,j}$ of V_i we have a component $v_{i,j} = a$ in $Env_{i,A}$, and for every edge $(v_{i,j}, v_{i,k})$ in E_i, we have the component $x_{i,j,k} = c(v_{i,j}, v_{i,k})$ in $Env_{i,B}$.

Assume that $exp(G_1) \simeq_{\alpha,\text{CHNR}} exp(G_2)$. Then there is a renaming $\sigma : V_1 \cup \bigcup\{x_{1,j,k}\} \to V_2 \cup \bigcup\{x_{2,j,k}\}$, such that $\sigma(exp(G_1))$ is syntactical equal to $exp(G_2)$ after some permutations of `let`-bindings. Let σ' be the restriction of σ to V_1. Obviously, σ' must be a bijection between V_1 and V_2; and $(v_{1,j}, v_{1,k}) \in E_1$ whenever $(\sigma'(v_{1,j}), \sigma'(v_{1,k})) \in E_2$. Thus σ' is an isomorphism between G_1 and G_2. Now assume that G_1 and G_2 are isomorphic. Then there exists a bijection $\sigma : V_1 \to V_2$ such that $(v_{1,j}, v_{1,k}) \in E_1$ iff $(\sigma(v_{1,j}), \sigma(v_{1,k})) \in E_2$. Then $exp(G_1)$ and $exp(G_2)$ are alpha-equivalent: σ can be used as a renaming, but has to be extended on the variables $x_{1,j,k}$ which is always possible. Thus G_1 and G_2 are isomorphic iff $exp(G_1) \simeq_{\alpha,\text{CHNR}} exp(G_2)$. Since every CHNR-expression is also a CH-expression, this also shows **GI**-hardness of CH-alpha-equivalence.

Now assume $\Sigma = \emptyset$: Then the same proof can be performed by replacing a by a free variable x_a and replacing $c(v_i, v_j)$ by `let` $y = v_i$ `in` v_j. ◀

▶ **Example 3.3.** We encode the following directed graphs as CHNR-expressions:



G_1 G_2

The encodings are $s_i = exp(G_i)$ with:

$$\begin{aligned} s_1 &= \texttt{let } v_1 = a; v_2 = a; v_3 = a; v_4 = a; v_5 = a \texttt{ in} \\ &\quad \texttt{let } x_1 = c(v_1, v_3); x_2 = c(v_1, v_5); x_3 = c(v_2, v_4); x_4 = c(v_2, v_1); \\ &\quad\quad x_5 = c(v_3, v_5); x_6 = c(v_4, v_3); x_7 = c(v_5, v_2) \texttt{ in } a \\ s_2 &= \texttt{let } w_1 = a; w_2 = a; w_3 = a; w_4 = a; w_5 = a \texttt{ in} \\ &\quad \texttt{let } x_1 = c(w_1, w_4); x_2 = c(w_2, w_3); x_3 = c(w_3, w_1); x_4 = c(w_4, w_2); \\ &\quad\quad x_5 = c(w_4, w_5); x_6 = c(w_5, w_1); x_7 = c(w_5, w_3) \texttt{ in } a \end{aligned}$$

The expressions are alpha-equivalent: For $\sigma = \{v_1 \mapsto w_5, v_5 \mapsto w_1, v_2 \mapsto w_4, v_4 \mapsto w_2, v_3 \mapsto w_3, x_1 \mapsto x_7, x_2 \mapsto x_6, x_5 \mapsto x_3, x_3 \mapsto x_4, x_4 \mapsto x_5, x_6 \mapsto x_2, x_7 \mapsto x_1\}$ the expression $\sigma(s_1)$ is syntactically equal to s_2 after "sorting" the `let`-environments.

The mapping σ restricted to $v_1, \ldots, v_5$ is an isomorphism between the graphs G_1, G_2.

▶ **Proposition 3.4.** *If Σ contains a binary constructor or function symbol, deciding alpha-equivalence in* CH *is* **GI***-hard, even if the expressions are restricted s.t.* `letrec` *is only allowed on the top-level of any expression, i.e. nested* `letrec`*-expressions are not permitted.*

Proof. The same proof as of Theorem 3.2 can be used, except that the expression encoding is `letrec` $Env_{i,A}, Env_{i,B}$ `in` a. ◀

Proposition 3.4 does not hold in CHNR, i.e., for non-recursive `let`-bindings. In this case alpha-equivalence can be decided in polynomial time which we show in Corollary 3.15.

Now we show that every alpha-equivalence problem in CH can be encoded as a directed graph-isomorphism problem, where the encoding can be done in polynomial time, and even in logarithmic space, i.e. deciding $\simeq_{\alpha,\mathsf{CH}}$ is in **GI**. First we define a graph construction from an expression:

▶ **Definition 3.5.** Given a CH-expression s, we describe the construction of $G(s)$, the labelled directed graph corresponding to s. We assume that s fulfills the distinct variable convention, i.e. the set of bound variables is distinct from free variables and bound variables are pairwise distinct. During the construction we use (and construct) a helper function `node`(.) which computes a node of the graph for every subexpression (with its position) of s. First the variables in s are partitioned into three sets: Let $\{x_1, \ldots, x_k\}$ be the free variables, $\{y_1, \ldots, y_m\}$ be the lambda-bound variables, $\{z_1, \ldots, z_n\}$ be the letrec-bound variables, and let $Var(s)$ be the union of the three sets.

Let $c_1, \ldots, c_k$ be the constructor and function symbols occurring in s, and q be the maximum arity of these symbols. Then the LDG $G(s)$ has the label set $L = \{\mathsf{var}, \mathsf{lamvar}, \mathsf{lvar}, \mathsf{body}, \mathsf{letvar}, \mathsf{bind}, \mathsf{letrec}, \mathsf{in}, \lambda\} \cup \{c_1, \ldots, c_k\} \cup \{1, \ldots, q\} \cup \{x_1, \ldots, x_k\}$.

For every variable $w \in Var(s)$ there is a node $N(w)$ in the graph $G(s)$. We set $\texttt{node}(w) := N(w)$. For every free variable x_i we set $\mathrm{lab}(N(x_i)) := x_i$, for every lambda-bound variable y_i we set $\mathrm{lab}(N(y_i)) = \mathsf{lamvar}$, and for every letrec-bound variable z_i we set $\mathrm{lab}(N(z_i)) = \mathsf{letvar}$.

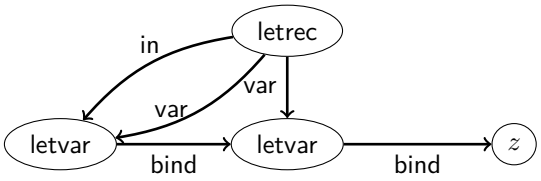
For the construction of the graph every subexpression is inspected (performed bottom up), according to the following cases (variables are already treated above):

- If the subexpression is $\lambda x.r$, then construct a new node v with $\mathrm{lab}(v) := \lambda$, and add edges $(v, \texttt{node}(x), \mathsf{lvar})$, $(v, \texttt{node}(r), \mathsf{body})$. Set $\texttt{node}(\lambda x.r) := v$.
- If the subexpression is ($\texttt{letrec } x_1 = s_1; \ldots; x_n = s_n \texttt{ in } t$), then we add one new node u with $\mathrm{lab}(u) := \mathsf{letrec}$, and let $\texttt{node}(\texttt{letrec } x_1 = s_1; \ldots; x_n = s_n \texttt{ in } t) := u$. For all $i = 1, \ldots, n$ we add edges $(u, \texttt{node}(x_i), \mathsf{var})$, $(\texttt{node}(x_i), \texttt{node}(s_i), \mathsf{bind})$, and an edge $(u, \texttt{node}(t), \mathsf{in})$.

- If the subexpression is $c(s_1, \ldots, s_n)$ then add one new node u with $\mathrm{lab}(u) = c$, and set $\mathtt{node}(c(s_1, \ldots, s_n)) := u$. For $i = 1, \ldots n$ add the edges (u, v_i, i).

Note that the construction of $\mathtt{node}(.)$ is meant together with the position of the subexpression, with the exception that for variables the position does not play any role, since there is only one node for every variable.

▶ **Example 3.6.** For the CH-expression $\mathtt{letrec}\ x = y, y = z\ \mathtt{in}\ x$ its corresponding LDG is $G = (V, E, L, \mathrm{lab})$ where $L = \{\mathsf{var}, \mathsf{lamvar}, \mathsf{lvar}, \mathsf{body}, \mathsf{letvar}, \mathsf{bind}, \mathsf{letrec}, \mathsf{in}, \lambda, z\}$, $V = \{v_1, v_2, v_3, v_4\}$, $E = \{(v_4, v_1, \mathsf{var}), (v_4, v_2, \mathsf{var}), (v_1, v_2, \mathsf{bind}), (v_2, v_3, \mathsf{bind}), (v_4, v_1, \mathsf{in})\}$, and $\mathrm{lab} = \{v_1 \mapsto \mathsf{letvar}, v_2 \mapsto \mathsf{letvar}, v_3 \mapsto z, v_4 \mapsto \mathsf{letrec}\}$. The graph can be depicted as follows:



▶ **Proposition 3.7.** CH-*alpha-equivalence is in* **GI**.

Proof. We encode alpha-equivalence of CH-expressions into the isomorphism problem of labelled directed graphs (which is GI-complete, see Proposition 2.4).

Let t_1, t_2 be CH-expressions. W.l.o.g. we assume that the expressions fulfill the distinct variable convention (if not, this can be achieved in time $O(n \log m)$ where n is the size of the term, and m is the number of variables). Let $G(t_1), G(t_2)$ be the graphs constructed according to Definition 3.5.

One can verify that t_1 and t_2 are alpha-equivalent if, and only if $G(t_1)$ and $G(t_2)$ are isomorphic: The whole term structure is preserved, `letrec`-bindings are commutable in the graph, and variable occurrences are represented by edges in the graph. Given an isomorphism ϕ from $G(t_1)$ to $G(t_2)$, the variable renaming making t_1 and t_2 syntactically equivalent (modulo commutation of `letrec`-bindings) can be derived by inspecting $\phi(\mathtt{node}(w))$ for every lambda- or letrec-bound variable w of t_1. This gives a node $v \in G(t_2)$ which must correspond to the according lambda- or letrec-bound variable in t_2. ◀

Obviously, CH-alpha-equivalence also solves CHNR-alpha-equivalence. Thus Propositions 3.7 and 3.2 imply:

▶ **Theorem 3.8.** CH-*alpha-equivalence and* CHNR-*alpha-equivalence are* **GI**-*complete.*

3.2 An Efficient Algorithm for Expressions without Garbage

The results up to now show that general CH-expressions have a hard alpha-equivalence problem. However, the question whether two CH-expressions are alpha-equivalent up to removing unused bindings can be answered efficiently, as we will see. Concretely, we define the following rewriting rules on CH-expressions for garbage collection (gc) that iteratively remove unused bindings:

$$(gc1)\quad \mathtt{letrec}\ x_1 = s_1; \ldots; x_n = s_n; y_1 = t_1; \ldots; y_m = t_m\ \mathtt{in}\ t_{m+1} \xrightarrow{gc} \mathtt{letrec}\ y_1 = t_1; \ldots; y_m = t_m\ \mathtt{in}\ t_{m+1} \quad \text{if } \bigcup_{i=1}^{m+1} FV(t_i) \cap \{x_1, \ldots, x_n\} = \emptyset$$

$$(gc2)\quad \mathtt{letrec}\ x_1 = s_1; \ldots; x_n = s_n\ \mathtt{in}\ t \xrightarrow{gc} t \quad \text{if } FV(t) \cap \{x_1, \ldots, x_n\} = \emptyset$$

It is easy to verify that the rewriting system on CH-expressions induced by $\xrightarrow{gc}$ is confluent and terminating. Thus unique normal forms w.r.t. (gc) exist. We say a CH-expression s is *without garbage* (or *garbage free*), if it is such a normal form, i.e. the rule (gc) is not applicable to any subexpression of s. Computing the normal form w.r.t. (gc) can be done in polynomial time, since (gc)-redexes can be detected efficiently and the rewriting can be performed by an innermost-strategy, inspecting every `letrec`-expression once. The complexity of computing the (gc)-normal form using the (gc)-rewriting steps is polynomial in n where n is the size of the expressions. However, by a global procedure an (alpha-equivalent) (gc)-normal form can also be computed in time $O(n \log n)$:

▶ **Lemma 3.9.** *Let s be a* CH*-expression. Then an alpha-equivalent (gc)-normal form of s can be computed in time $O(n \log n)$ where n is the size of s.*

Proof. W.l.o.g. we assume that s fulfills the distinct variable convention (otherwise a renamed expression can be computed in time $O(n \log n)$). As a first step, construct the LDG $G(s)$ according to Definition 3.5. Then mark all nodes of $G(s)$ that are reachable from $\texttt{node}(s)$ by never using any edge labelled var. This requires time $O(|G(s)| \log |G(s)|)$ if we store the nodes in an efficient data structure. Clearly, unmarked nodes are garbage. Hence, we delete all unmarked nodes, all corresponding edges from $G(s)$, and finally letrec-nodes that have no outgoing edges marked with var, where incoming edges are redirected to the node corresponding to the `in`-expression. Let the result be the LDG G'. The deletion procedure requires $O(|G(s)| \log |G(s)|)$ time, since a traversal of the graph (with some lookups whether nodes are marked or not marked) is sufficient. Finally, reconstruct the garbage-free expression corresponding to G' which is always possible. ◀

Below we show that the alpha-equivalence-problem for garbage free CH-expressions can be decided efficiently, and thus also the question whether for two given expressions s_1, s_2 their (gc)-normal forms s_1', s_2' are alpha-equivalent can be answered in polynomial time.

Hence, the conclusion is that the worst-case high complexity for checking alpha-equivalence in CH is due to comparing the garbage subexpressions.

▶ **Lemma 3.10.** *Let s be a garbage free* CH*-expression, and $G(s)$ the LDG constructed from s (see Definition 3.5). Then the subgraph $G' := OO(G(s))$ satisfies the assumptions of Proposition 2.12: G' is rooted and an OOLDG.*

Proof. It is easy to see that garbage-freeness of s implies that every node of $G(s)$ is reachable from the root via edges that are not labelled with var. The subgraph $G' := OO(G(s))$ is the subgraph of $G(s)$ where for the nodes all var-edges are removed, if there are at least two outgoing var-edges from this node. ◀

▶ **Theorem 3.11.** *Given* CH*-expressions s_1, s_2, where at least one of s_1, s_2 is free of garbage, their alpha-equivalence can be checked in time $O(n \log(n))$ where $n = |G(s_1)| + |G(s_2)|$.*

Proof. It can be checked in log-linear time whether (gc) is applicable to an expression by constructing $G(s_i)$ and checking whether every node is reachable via edges not labelled with var. The expressions can only be alpha-equivalent, if both are free of garbage. Thus by Lemma 3.10 and Proposition 2.12 the isomorphism-test of $G(s_1)$ and $G(s_2)$ can be performed in time $O(n \log(n))$ where $n = |G(s_1)| + |G(s_2)|$. Since this is equivalent to alpha-equivalence of s_1, s_2, the theorem holds. ◀

▶ **Definition 3.12.** CH-expressions s, t are *alpha-equivalent up to garbage-collection*, denoted by $s \simeq_{\alpha,gc,\mathsf{CH}} t$, iff the (gc)-normal forms s' and t' of s and t are alpha-equivalent.

▶ **Theorem 3.13.** *For* CH*-expressions* s_1, s_2 *it is possible to decide whether* $s_1 \simeq_{\alpha,gc,\mathsf{CH}} s_2$ *in time* $O(n \log n)$ *where* $n = |s_1| + |s_2|$.

Proof. First, alpha-equivalent (gc)-normal forms of s_1, s_2 are computed by Lemma 3.9. Finally, Theorem 3.11 is applied to the (gc)-normal forms. ◀

Note that checking $\simeq_{\alpha,gc,\mathsf{CH}}$ instead of $\simeq_{\alpha,\mathsf{CH}}$ is an option for the automation of the diagram method, but requires the transformation (gc) in the set of permitted transformations/reductions in the reduction sequences, which is often not the case.
After adapting the (gc)-definition to CHNR, very similar reasoning and arguments show:

▶ **Theorem 3.14.** *Theorem 3.11 and 3.13 also hold for* CHNR.

▶ **Corollary 3.15.** *For* CHNR*-expressions* s_1, s_2 *where* `let`*-expressions are only allowed on the top-level of expressions, it is possible to decide whether* $s_1 \simeq_{\alpha,\mathsf{CHNR}} s_2$ *in time* $O(n^3 \log n)$ *where* $n = |s_1| + |s_2|$.

Proof. First decide whether $s_1 \simeq_{\alpha,gc,\mathsf{CHNR}} s_2$ using Theorem 3.14. Then compute the bindings of s_1 and s_2 that are garbage. Testing whether these bindings are alpha-equivalent is possible in time $O(n^3 \log n)$: alpha-equivalence of a single binding $x = t_1$ of s_1 and a single binding $y = t_2$ of s_2 can be tested in time $O(n \log n)$, since t_1, t_2 are usual terms (i.e. ranked trees). For testing the sets of bindings we have to compare $O(n^2)$ bindings. ◀

For CH-expressions s, t with garbage, but without nested letrecs, one can test first the expressions after removing the garbage bindings, and then test all combinations of the garbage-bindings. This implies the following complexity estimation:

▶ **Corollary 3.16.** *Let* s_1, s_2 *be* CH*-expressions with a one-level top letrec. Let* $n = |s_1| + |s_2|$ *and* k *be the number of let-variables that are garbage in* s_1*. Then* $s_1 \simeq_{\alpha,\mathsf{CH}} s_2$ *can be decided in time* $O(k!\, n \log n)$.

3.3 Applications: Lambda-Calculi with Bindings and Haskell

In this section we analyze and list several program calculi and programming languages which fit into the syntax of CH or CHNR and thus have a **GI**-complete alpha-equivalence problem.

The first class of calculi covers several call-by-need lambda calculi with letrec, all of them fit into the syntax of the language CH.

▶ **Proposition 3.17.** *Deciding alpha-equivalence in lambda-calculi with letrec is* **GI***-complete. This includes the following calculi: The call-by-need lambda calculi in [3, 2] (in the variants with letrec), the call-by-need lambda calculus with letrec in [37], and the cyclic lambda calculi [4, 1]. Extended call-by-need letrec calculi as e.g. used in [23, 38], and also extensions with non-deterministic operators [22, 24, 33, 36].*

In all the mentioned calculi a rule for garbage collection can be defined such that Theorem 3.13 is applicable.

Note that in call-by-need lambda calculi with non-recursive, single-binding let-expressions (as e.g. [18]) alpha-equivalence can be decided in log-linear time.

We analyze the following fragment of Haskell. Data definitions are permitted where the names of types and constructors are not renameable. Supercombinator definitions are also permitted, but no modules, expressions are built by the usual constructs: variables, abstractions, applications, constructor applications and case-expressions, also `letrec`-expressions may be present. We also assume that `main` is a distinguished name of a supercombinator and

that the other supercombinator names may be renamed. Then clearly, a Haskell program with supercombinators $x_i\ y_{1,i} \ldots y_{1,m_i} = s_i$ for $i = 1, \ldots, n$ can be expressed in CH as `letrec` $x_1 = \lambda y_{1,1}, \ldots, y_{1,m_1}.s_1; \ldots; x_n = \lambda y_{n,1}, \ldots y_{n,m_n}.s_n$ `in main` and thus we have:

▶ **Corollary 3.18.** *Haskell with only data definitions and renamble supercombinator definitions has a* **GI***-complete alpha-equivalence problem, even if* `letrec`*-expressions are forbidden. Also a rule for garbage collection can be defined such that Theorem 3.13 is applicable.*

In the record calculus [17] records are sets of bindings $l_i \mapsto t_i$ where t_i are expressions of an extended lambda calculus (no `letrec`-bindings). This is like a one-level `letrec`, hence:

▶ **Proposition 3.19.** *The alpha-equivalence problem in the record calculus [17] is* **GI***-complete, provided renaming of the record names is permitted.*

Proof. This follows from Proposition 3.4. ◀

In the record calculus there is no distinguished `main`-label and thus a sufficient rule for garbage collection is not definable in this calculus, i.e. Theorem 3.13 is not applicable.

3.4 Comparison of Types

Another application of our methods is comparison of types under various equivalences, which for example is a useful feature for searching functions with similar types in function libraries [31]. Fix a finite, non-empty set of type constructors tc_i where every type constructor has a fixed arity $ar(tc_i) \in \mathbf{N}_0$. Let α, α_i be type variables of a countably-infinite set of type variables. The syntax of polymorphic types $\tau \in T$ is $\tau, \tau_i \in T ::= \alpha \mid \tau_1 \to \tau_2 \mid tc(\tau_1, \ldots, \tau_{ar(tc)})$. As usual we assume arrow-types to be right-associative. Let similarity, $\sim$, of types be $\sim_1 \circ \sim_2$ where $\sim_1$ allows renaming of type variables and $\sim_2$ is the least congruence on types that respects the axiom $\tau_1 \to \tau_2 \to \tau_3 \sim \tau_2 \to \tau_1 \to \tau_3$. For instance, $(\alpha_1 \to \alpha_2) \to List(\alpha_1) \to List(\alpha_2) \sim List(\alpha_4) \to (\alpha_4 \to \alpha_3) \to List(\alpha_3)$. A type is *positive*, if it is a type variable, or of the form $\tau_1 \to \ldots \to \tau_n$ or $tc(\tau_1, \ldots, \tau_{ar(tc)})$ where every type τ_i does not contain arrow-types.

▶ **Theorem 3.20.** *Similarity of Hindley-Milner polymorphic types (as defined above), even for only positive types, is* **GI***-complete.*

Proof. In **GI** follows using standard methods. **GI**-hardness follows, where the encoding of a directed graph G is as follows: For nodes $v_i, i = 1, \ldots, n$, edges $\{(v_{i_1}, v_{j_1}), \ldots, (v_{i_k}, v_{j_k})\}$, $1 \le i_h, j_h \le n$, let the constructed (positive) type be $v_1 \to \ldots \to v_n \to tc(v_{i_1}, v_{j_1}) \to tc(v_{i_2}, v_{j_2}) \to \ldots \to tc(v_{i_k}, v_{j_k}) \to \alpha$ where v_{i_1}, v_{j_1}, α are now type variables. ◀

4 Structural Congruence in Process Calculi

In this section we consider structural congruence in process calculi. In particular the π-calculus and several fragments are analyzed that not only include an associative and commutative binary operator $\mid$ for parallel composition but also axioms for moving ν-binders, like $\nu x.\nu y.P \equiv \nu y.\nu x.P$ and $(\nu x.P) \mid Q \equiv \nu x.(P \mid Q)$ if $x \notin FV(Q)$.

4.1 Process Calculi with Bindings

We first consider two extended lambda calculi that have a recursive binding scope only at top-level, where bindings may be permutable, and where a structural congruence is defined

extending alpha-equivalence. This covers the call-by-value lambda calculus with futures in [26, 25, 40] and also CHF [34], a concurrent process-extension of Haskell languages.

▶ **Proposition 4.1.** *Deciding structural congruence $\equiv$ in the variants of the lambda-calculus with futures in [25, 40] is* **GI***-complete, as well as of the language CHF modelling Concurrent Haskell with futures [34].*

Proof. **GI**-hardness follows from Proposition 3.4, since these calculi can express renameable, recursive one-level `letrec`-bindings. The proof that deciding structural congruence is in **GI** does not directly follow from results for CH: First one has to verify that a prenex-normal form can be computed, where all ν-binders are on the top of the process, i.e. a process in prenex-normal form is of the form $\nu x_1 \ldots \nu x_n.P$ where P does not contain ν-binders. For encoding these normal forms into graphs, the nested ν-binders are treated like a single binding-operator which binds a *set* of variables, i.e. $\nu X.P$ where $X = \{x_1, \ldots, x_n\}$. Also nested parallel compositions $P_1 \mid P_2 \mid \ldots$ are treated like a (multi-)set of processes $\{P_1, \ldots, P_n\}$. With this preparations the corresponding LDG of a process can be encoded such that isomorphism of the LDGs is equivalent to structural congruence of the processes. ◀

Note that in CHF a garbage collection rule can be defined which is sufficient to apply Theorem 3.13, since there is a distinguished main-thread, while in the lambda calculus with futures there is no such thread and thus Theorem 3.13 is not applicable.

4.2 Structural Congruence in the π-Calculus

We analyze deciding structural congruence in the π-calculus [21, 20, 35]. Note that in the π-calculus a prenex-normal form does not exist, since not every ν-binder can be moved to the top.

Note that in calculi with commutable ν-binders and commutative-associative composition, **GI**-hardness of structural congruence can also be concluded by encoding directed graphs (V, E) with $V = \{v_1, \ldots, v_n\}$ as $\nu v_1, \ldots v_n.P$ where P is a (nested) parallel composition of the components: $c_1(v)$ for any $v \in V$ and $c_2(v, w)$ for every directed edge $(v, w) \in E$ where c_1 is a unary, and c_2 a binary constructor or function symbol. E.g. in the π-calculus with input and output prefixes defined by $\pi ::= x(y) \mid \overline{x}\langle y\rangle$ we can use $\overline{v}\langle w\rangle.0$ for $c_2(v, w)$ and $v(z).0$ for $c_1(v)$ for some name z. These encodings can be found in [14] and are similar to the encodings given in [6]. This shows that structural congruence in several fragments of the π-calculus is at least **GI**-hard. In [14] it was also shown that in the π-calculus with sums and non-renameable defined function symbols (perhaps recursive) deciding structural equivalence is **GI**-complete.

However, in the π-calculus with a replication operator **!** structural congruence includes the axiom $!P \equiv P \mid !P$. With this axiom deciding structural congruence is much harder. We consider the following fragment – called $\mathcal{PIR}$ – which covers several variants of the π-calculus with replication. Let C be an infinite set of constants (atomic actions), then the syntax of $\mathcal{PIR}$ is: $s, s_i \in \mathcal{PIR} := C \mid (s_1 \mid s_2) \mid !s$. We assume that structural congruence is defined by the axioms $(s_1 \mid s_2) \equiv (s_2 \mid s_1)$, $(s_1 \mid (s_2 \mid s_3)) \equiv ((s_1 \mid s_2) \mid s_3)$ and $!s \equiv s \mid !s$.

In [11] it is shown that structural congruence in $\mathcal{PIR}$ can be decided in **EXPSPACE**. Indeed the problem is also **EXPSPACE**-hard, which we show in the following.

First we consider commutative semigroups. Let Σ be an alphabet of constants, written a, b, c, perhaps with indices.

▶ **Lemma 4.2.** *Given equations $s_1 = 1, \ldots, s_n = 1$ where s_i are (commutative) words over Σ, and two further commutative words s, t over Σ, then the decision problem of the word problem $s_1 = 1 \wedge \ldots \wedge s_n = 1 \models s = t$ over commutative monoids is* **EXPSPACE***-complete.*

Proof. Given equations $s_1 = t_1, \ldots, s_n = t_n$ where s_i, t_i are (commutative) words over Σ, and two further commutative words s, t over Σ, then the decision problem of the word problem $s_1 = t_1 \wedge \ldots \wedge s_n = t_n \models s = t$ over commutative semigroups is **EXPSPACE**-complete [19]. We apply this result. Encoding the commutative monoid word problem is easy by adding the axioms for the unit to the given equations. For the other direction we add n fresh constants $g_1, \ldots, g_n$ to the signature and encode the equations as $s_1 g_1 = 1, t_1 g_1 = 1, \ldots, s_n g_n = 1, t_n g_n = 1$. The equation $s_i = s_i g_i t_i = t_i$ for $i = 1, \ldots, n$ is then derivable from these equations. It is also not hard to see that the extension is conservative, i.e., the equational theories are equivalent on words free of $g_1, \ldots g_n$. Since the encodings are polynomial, we are done. ◀

▶ **Theorem 4.3.** *Structural congruence of expressions in $\mathcal{PIR}$ is* **EXPSPACE***-complete.*

Proof. Due to the results of [11] it suffices to show **EXPSPACE**-hardness. For convenience, let us write the expressions without the parallel-operator, and assume that we have commutative words in C^*. Then $w_1!(v_1)\ldots!(v_n) \equiv w_2!(v_1)\ldots!(v_n)$ is equivalent to $v_1 = 1 \wedge \ldots \wedge v_n = 1 \models w_1 = w_2$ in a commutative monoid. Thus Lemma 4.2 implies **EXPSPACE**-hardness. ◀

Since $\mathcal{PIR}$ can be embedded in the π-calculus with replication by simulating the constants with different input-expressions, and omitting ν-binders, we have the following result:

▶ **Corollary 4.4.** *Structural congruence in the π-calculus with replication is* **EXPSPACE***-hard.*

This high complexity is a hint that it is not a good idea to include the replication axiom in the congruence relation. It would be better to include it in the operational semantics and only copy expressions "by need".

Engelfriet and Gelsema [10] investigate variants of the congruence axioms for the replication operator and show decidability of the congruence, however, they do not mention complexity bounds. An application of congruence and complexity of these variants is in [32].

Note that Theorem 4.3 does not apply to variants of congruence for the replication operator as proposed in [10], since the respective congruences are different.

▶ Remark. The complexity (and even the decidability) of the congruence problem for Milner's variant of the π-calculus is still open. We show an example for one of the problematic cases (using term notation) that are not covered by [11]: Let $s = \nu x.(f(x) \mid !(f(x) \mid a)) \mid \nu x.!(g(x) \mid a)$, and $t = \nu x.!(f(x) \mid a) \mid \nu x.(g(x) \mid !(g(x) \mid a))$, then $s \equiv t$ by exchanging the action a and using the replication axiom in both directions.

5 Summary and Conclusion

We have shown that alpha-equivalence of expressions in higher-order core functional programming languages with a recursive let-construct that permits to permute the bindings, and thus also for Haskell-expressions, is **GI**-complete, and consequently, algorithms require exponential time in the worst case. If there is no garbage, then the worst case time complexity is polynomial. This fact allows deduction systems a choice: if the application of algorithms for alpha-equivalence is infeasible in the deduction, then the program transformation (gc) could

be added to the set of transformations, which opens the possibility to use the polynomial time algorithm for garbage free expression as proposed above.

Acknowledgments

We thank the anonymous reviewers for their valuable and helpful comments.

References

1 Z. M. Ariola and S. Blom. Skew confluence and the lambda calculus with letrec. *Ann. Pure Appl. Logic*, 117(1-3):95–168, 2002.

2 Z. M. Ariola and M. Felleisen. The call-by-need lambda calculus. *J. Funct. Programming*, 7(3):265–301, 1997.

3 Z. M. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. A call-by-need lambda calculus. In *POPL'95*, pp. 233–246. ACM Press, 1995.

4 Z. M. Ariola and Stefan Blom. Cyclic lambda calculi. In *Proc. 3rd TACS '97*, LNCS 1281, pp. 77–106. Springer, 1997.

5 G. Ausiello, F. Cristiano, and L. Laura. Syntactic isomorphism of CNF Boolean formulas is graph isomorphism complete. *Electronic Colloquium on Computational Complexity*, 19:122, 2012.

6 D. A. Basin. A term equality problem equivalent to graph isomorphism. *Inf. Process. Lett.*, 51(2):61–66, 1994.

7 K. S. Booth and C. J. Colboum. Problems polynomially equivalent to graph isomorphism. Technical Report CS-77-04, University of Waterloo, 1979.

8 C. Calvès and M. Fernández. Matching and alpha-equivalence check for nominal terms. *J. Comput. System Sci.*, 76(5):283–301, 2010.

9 J. Cheney. The complexity of equivariant unification. In *Proc. 31st ICALP*, *LNCS* 3142, pp. 332–344. Springer, 2004.

10 J. Engelfriet and T. Gelsema. A new natural structural congruence in the pi-calculus with replication. *Acta Inf.*, 40(6-7):385–430, 2004.

11 J. Engelfriet and T. Gelsema. An exercise in structural congruence. *Inf. Process. Lett.*, 101(1):1–5, 2007.

12 J. J. Fu. Directed graph pattern matching and topological embedding. *J. Algorithms*, 22(2):372–391, 1997.

13 X. Jiang and H. Bunke. Optimal quadratic-time isomorphism of ordered graphs. *Pattern Recognition*, 32(7):1273–1283, 1999.

14 V. Khomenko and R. Meyer. Checking pi-calculus structural congruence is graph isomorphism complete. In *Proc. 9th ACSD*, pp. 70–79. IEEE Computer Society, 2009.

15 Johannes Köbler. On graph isomorphism for restricted graph classes. In *CiE*, *LNCS* 3988, pp. 241–256. Springer, 2006.

16 J. Köbler, U. Schöning, and J. Torán. Graph isomorphism is low for PP. *Comput. Complexity*, 2:301–330, 1992.

17 E. Machkasova. Computational soundness of a call by name calculus of recursively-scoped records. *ENTCS*, 204:147–162, 2008.

18 J. Maraist, M. Odersky, and P. Wadler. The call-by-need lambda calculus. *J. Funct. Programming*, 8:275–317, 1998.

19 E. W. Mayr and A. R. Meyer. The complexity of the word problems for commutative semigroups and polynomial ideals. *Adv. in Math.*, 46(3):305–329, 1982.

20 R. Milner. Functions as processes. *Math. Structures Comput. Sci.*, 2(2):119–141, 1992.

21 R. Milner. *Communicating and Mobile Systems: the π-calculus.* Cambridge university press, 1999.

22 A. K. D. Moran. *Call-by-name, call-by-need, and McCarthy's Amb.* PhD thesis, Dept. of Comp. Science, Chalmers university, Sweden, 1998.

23 A. K. D. Moran and D. Sands. Improvement in a lazy context: An operational theory for call-by-need. In *Proc. 26th POPL*, pp. 43–56. ACM Press, 1999.

24 A. K. D. Moran, D. Sands, and M. Carlsson. Erratic fudgets: A semantic theory for an embedded coordination language. *Sci. Comput. Program.*, 46(1-2):99–135, 2003.

25 J. Niehren, D. Sabel, M. Schmidt-Schauß, and J. Schwinghammer. Observational semantics for a concurrent lambda calculus with reference cells and futures. *ENTCS*, 173:313–337, 2007.

26 J. Niehren, J. Schwinghammer, and G. Smolka. A concurrent lambda calculus with futures. *Theoret. Comput. Sci.*, 364(3):338–356, 2006.

27 A. M. Pitts. Nominal logic, a first order theory of names and binding. *Inf. Comput.*, 186(2):165–193, 2003.

28 C. Rau, D. Sabel, and M. Schmidt-Schauß. Correctness of program transformations as a termination problem. In *Proc. 6th IJCAR*, LNCS 7364, pp. 462–476. Springer, 2012.

29 C. Rau and M. Schmidt-Schauß. A unification algorithm to compute overlaps in a call-by-need lambda-calculus with variable-binding chains. In *Proc. 25th UNIF*, pp. 35–41, 2011.

30 C. Rau and M. Schmidt-Schauß. Computing overlappings by unification in the deterministic lambda calculus LR with letrec, case, constructors, seq and variable chains. Frank Report 46, Institut für Informatik. Goethe-Universität Frankfurt, 2011.

31 M. Rittri. Using types as search keys in function libraries. *J. Funct. Program.*, 1(1):71–89, 1991.

32 A. Romanel and C. Priami. On the decidability and complexity of the structural congruence for beta-binders. *Theoret. Comput. Sci.*, 404(1-2):156–169, 2008.

33 D. Sabel and M. Schmidt-Schauß. A call-by-need lambda-calculus with locally bottom-avoiding choice: Context lemma and correctness of transformations. *Math. Structures Comput. Sci.*, 18(03):501–553, 2008.

34 D. Sabel and M. Schmidt-Schauß. A contextual semantics for Concurrent Haskell with futures. In *Proc. 13th PPDP*, pp. 101–112. ACM, 2011.

35 D. Sangiorgi and D. Walker. *The π-calculus: a theory of mobile processes.* Cambridge university press, 2001.

36 M. Schmidt-Schauß and E. Machkasova. A finite simulation method in a non-deterministic call-by-need calculus with letrec, constructors and case. In *Proc. 19th RTA*, LNCS 5117, pp. 321–335. Springer-Verlag, 2008.

37 M. Schmidt-Schauß, D. Sabel, and E. Machkasova. Simulation in the call-by-need lambda-calculus with letrec. In *Proc. 21st RTA*, LIPIcs 6, pp. 295–310. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2010.

38 M. Schmidt-Schauß, M. Schütz, and D. Sabel. Safety of Nöcker's strictness analysis. *J. Funct. Programming*, 18(04):503–551, 2008.

39 U. Schöning. Graph isomorphism is in the low hierarchy. *J. Comput. Syst. Sci.*, 37(3):312–323, 1988.

40 J. Schwinghammer, D. Sabel, M. Schmidt-Schauß, and J. Niehren. Correctly translating concurrency primitives. In *Proc. ML '09*, pp. 27–38. ACM, 2009.

41 V.N. Zemlyachenko, N.M. Korneenko, and R.I.Tyshkevich. Graph isomorphism problem. *J. Math. Sci. (N. Y.)*, 29:1426–1481, 1985.