Hashing Modulo Context-Sensitive α-Equivalence

LASSE BLAAUWBROEK and MIROSLAV OLŠÁK, Institut des Hautes Études Scientifiques, France
HERMAN GEUVERS, Radboud University, The Netherlands

The notion of α-equivalence between λ-terms is commonly used to identify terms that are considered equal. However, due to the primitive treatment of free variables, this notion falls short when comparing subterms occurring within a larger context. Depending on the usage of the Barendregt convention (choosing different variable names for all involved binders), it will equate either too few or too many subterms. We introduce a formal notion of context-sensitive α-equivalence, where two open terms can be compared within a context that resolves their free variables. We show that this equivalence coincides exactly with the notion of bisimulation equivalence. Furthermore, we present an efficient $O(n \log n)$ runtime hashing scheme that identifies λ-terms modulo context-sensitive α-equivalence, generalizing over traditional bisimulation partitioning algorithms and improving upon a previously established $O(n \log^2 n)$ bound for a hashing modulo ordinary α-equivalence by Maziarz et al [20]. Hashing λ-terms is useful in many applications that require common subterm elimination and structure sharing. We have employed the algorithm to obtain a large-scale, densely packed, interconnected graph of mathematical knowledge from the Coq proof assistant for machine learning purposes.

CCS Concepts: • **Software and its engineering** → **Compilers**; • **Theory of computation** → **Lambda calculus**; **Design and analysis of algorithms**.

Additional Key Words and Phrases: Lambda Calculus, Hashing, Alpha Equivalence, Bisimilarity, Syntax Tree

1 INTRODUCTION

This paper studies equivalence of λ-terms modulo renaming of bound variables, so called α-equivalence. This has been studied extensively in the history of λ-calculus, starting with Church [11]. The overview book by Barendregt [3] also defines and studies it in detail. There, α-equivalence is defined as a relation $t =_\alpha u$ between two λ-terms that captures the idea that t can be obtained from u by renaming bound variables.

In the present paper we study a more general situation where t and u are accompanied by a *context* that binds their free variables. Hence, we study the notion of *context-sensitive α-equivalence*, which we will show to coincide with bisimulation when interpreting λ-terms as graphs. This notion has particular importance in case one wants to semantically compare and de-duplicate the subterms of huge λ-terms (e.g. a dataset extracted from the Coq proof assistant [25]). To this end, we also define an efficient $O(n \log n)$-time hashing algorithm that respects this equivalence, in the sense that sub-terms receive the same hash if and only if they are context-sensitive α-equivalent.

1.1 Problem Description

The α-equivalence relation equates λ-terms modulo the names of binders. For example, $\lambda x.\lambda y.x =_\alpha \lambda y.\lambda x.y$. De Bruijn indices are a way to make the syntax of λ-terms invariant w.r.t. the α-equivalence relation. Using de Bruijn indices, both terms above would be represented by $\lambda\lambda \underline{1}$, and would hence be syntactically equal.

The α-equivalence relation becomes less clear when free variables are involved. Usually, it is understood that terms are only equal when all occurrences of free variables are equal. However, the situation is more complicated when considering free variables within a known context. Consider the example

$$\lambda t.\, Q\ (\lambda z.\lambda f.\, f\ t)\ (\lambda g.\, g\ t). \tag{1.0.1}$$

Authors' addresses: Lasse Blaauwbroek, lasse@blaauwbroek.eu; Miroslav Olšák, mirek@olsak.net, Institut des Hautes Études Scientifiques, Bures-sur-Yvette, France; Herman Geuvers, herman@cs.ru.nl, Radboud University, Nijmegen, The Netherlands.

In this term, are the subterms $\lambda f.\ f\ t$ and $\lambda g.\ g\ t$ considered α-equivalent? Most would agree they are equivalent because we can share these terms with a let-construct without changing the meaning of the term:

$$\lambda t.\ \text{let}\ s := \lambda f.\ f\ t\ \text{in}\ Q\ (\lambda z.s)\ s \qquad\qquad (1.0.2)$$

Here, the justification that we are "not changing the meaning of the term" is that one β-reduction of the introduced *let* will give us the original term (modulo renaming of bound variables). However, when we represent the original term with de Bruijn indices, these two sub-terms are not syntactically equal.

$$\lambda\ Q\ (\lambda\lambda\ \underline{0}\ \underline{2})\ (\lambda\ \underline{0}\ \underline{1})$$

The promise of de Bruijn indices has failed us! If we want to find the common sub-terms of a program, we cannot simply convert the program to use de Bruijn indices, hash the program into a Merkle tree, and call it a day. Other representations, like de Bruijn levels, suffer from similar issues.

In addition to false negatives, de Bruijn indices can also lead to false positives. Consider the example

$$\lambda t.\ Q\ (\lambda z.\lambda f.\ f\ z)\ (\lambda g.\ g\ t). \qquad\qquad (1.0.3)$$

Here, the subterms $\lambda f.\ f\ p$ and $\lambda g.\ g\ t$ are not α-equivalent. However, when expressed using de Bruijn indices they become equal.

$$\lambda\ Q\ (\lambda\lambda\ \underline{0}\ \underline{1})\ (\lambda\ \underline{0}\ \underline{1})$$

Given these counter-examples, one might conclude that de Bruijn indices are not as useful in deciding equality between (sub)-terms as is commonly thought. Unfortunately, the situation is not much better for λ-terms with ordinary named variables. Take for example this naive attempt at defining context-sensitive α-equivalence:

> Two subterms of t are α-equivalent in the context of t if the bound variables in t can be renamed such that the subterms become syntactically equal.

An immediate counter-example to this definition is the term $Q\ (\lambda x.x)\ (\lambda xy.\ x)$ and the question whether the two occurrences of the variable x are α-equivalent. According to the definition, yes, but the variables correspond to binders that cannot possibly be considered equivalent.

At this point, it is not even clear what precise equivalence relation we are looking for, even though many people would "know an equivalence between subterms when they see one." The only intuitive idea we have to build on is the introduction of a *let*-abstraction in Formula 1.0.2. But, as we will see, this is not sufficient on its own.

1.2 Fork Equivalence

Let us return to Example 1.0.1, where we used *let*-abstraction to "show" that two subterms are equal. We make this more precise (a fully formal treatment can be found in Section 2). To conveniently talk about the equality of subterms within a context, we underline the two terms of interest. This allows us to restate the question in Example 1.0.1 as follows.

$$\lambda t.\ Q\ (\lambda z.\underline{\lambda f.\ f\ t})\ (\underline{\lambda g.\ g\ t})$$
$$\lambda\ Q\ (\lambda\lambda\ \underline{\underline{0}\ \underline{2}})\ (\lambda\ \underline{\underline{0}\ \underline{1}}) \qquad\qquad (1.0.4)$$

The underlined subterms are the *subject* of the context-sensitive α-equivalence. The remainder of the term, which we can write as $\lambda t.\ Q\ (\lambda z.\ \_)\ \_$ or $\lambda\ Q\ (\lambda\ \_)\ \_$, is the *context* in which the equivalence is to be shown. Now, in order to perform the *let*-abstraction, we must split the context into two pieces, between which we can insert the *let*. In this case, the split we make is $\lambda t.\ \_$ and $Q\ (\lambda z.\ \_)\ \_$. We can then show that the term $\lambda f.\ f\ t$ is closed under the outer context and when we

(a) The single fork in Example 1.0.4.   (b) Illustration of sub-forks and transitivity from Example 1.0.5.
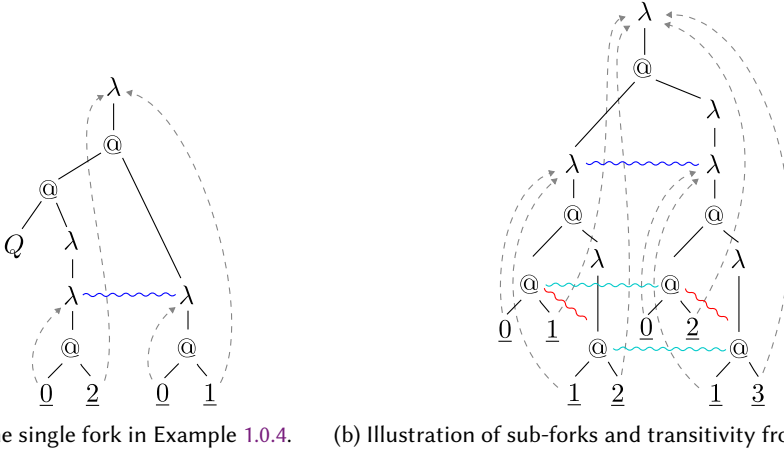
Fig. 1. Illustrations of forks in terms built from de Bruijn indices. Equivalent sub-terms are related through a squiggly line. Back-edges from variables to binders are illustrative only.

substitute it into the holes in the inner context (while avoiding variable capture), we obtain the original term. Hence, we can reassemble everything to obtain the same term from Example 1.0.2.

$$\lambda t. \text{ let } s := \lambda f.\, f\, t \text{ in } Q\ (\lambda z.s)\ s$$

Let-abstractable subterms are the first ingredient of the fork-equivalence relation[1]. This relation is named so because the relation is built upon a fork, starting with a single outer context that forms the base of the fork. Then follows an inner context that acts as the bifurcation of the base towards the two subjects. The fork is shown in Figure 1a.

The definition of fork equivalence is not yet complete. *Let*-abstraction only allows us to compare subjects that share a common context. However, when two subjects are closed, their context is irrelevant. In that situation, both subjects may occur in a completely different context while still being equivalent. This gives rise to an equivalence relation $\sim_f$ formed between two pairs of a subject and context. This allows establishing equivalences between closed subjects such as

$$P\ (\underline{\lambda x.x})\ \sim_f Q\ (\underline{\lambda y.y}).$$

In general, in addition to *let*-abstraction, we say that a fork can also be established between any two closed subjects that are equal modulo variable renaming (or syntactically equal in the case of de Bruijn indices). Closed subjects can be accompanied by arbitrary contexts, as they can never influence the meaning of the subjects. *Let*-abstraction, when phrased as a relation $\sim_f$, would still require a common context even though that common context is stated twice. Our running Example 1.0.4 would be phrased as

$$\lambda t.\ Q\ (\lambda z.\underline{\lambda f.\, f\, t})\ (\lambda g.\, g\, t)\ \sim_f \lambda t.\ Q\ (\lambda z.\lambda f.\, f\, t)\ (\underline{\lambda g.\, g\, t}).$$

---

[1]The formal definition we introduce later does not involve actual *let*-expressions, but rather identifies the circumstances where a *let*-abstraction can be performed.

To complete the definition of fork equivalence, we must also extend it with both the *sub-fork* and with transitivity of forks. The following example illustrates the need for these extensions:

$$\lambda t. \, (\lambda x. \, \underline{x \, t} \, (\lambda y. \, x \, t)) \, (\lambda z. \, \lambda x. \, x \, t \, (\lambda y. \, x \, t))$$
$$\sim_{\mathrm{f}} \lambda t. \, (\lambda x. \, x \, t \, (\lambda y. \, \underline{x \, t})) \, (\lambda z. \, \lambda x. \, x \, t \, (\lambda y. \, x \, t)) \qquad\qquad (1.0.5)$$
$$\sim_{\mathrm{f}} \lambda t. \, (\lambda x. \, x \, t \, (\lambda y. \, x \, t)) \, (\lambda z. \, \lambda x. \, x \, t \, (\lambda y. \, \underline{x \, t}))$$

Notice how the first equivalence can be established because the subjects are *let*-abstractable. For the second equivalence, we have no such luck. There is no place where we could split the context such that both instances of $x \, t$ would be closed under the outer context. Note, however, that if we widen the subject $x \, t$ to $\lambda x. \, x \, t \, (\lambda y. \, x \, t)$, a *let*-abstraction can indeed be performed. Because both underscored instances of $x \, t$ occur in the same position in this wider term, we allow for the creation of a sub-fork between them. Finally, to demonstrate the transitivity concept, we can combine both *let*-abstractions in the following term, such that all underscored locations coincide:
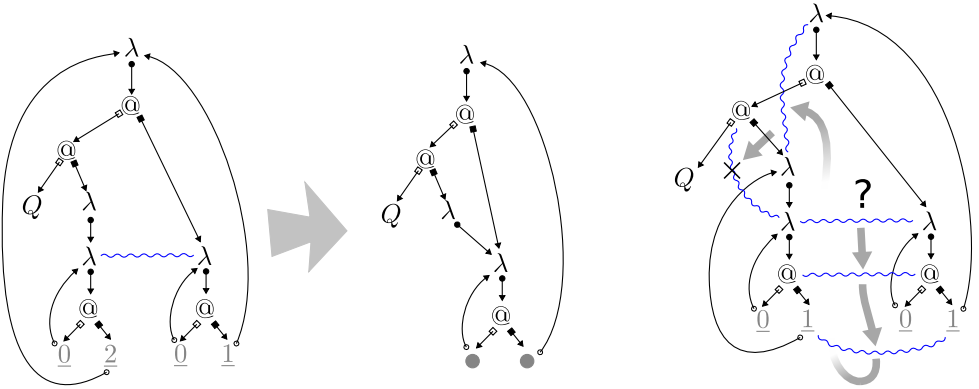
$$\lambda t. \, \text{let } u \coloneqq (\lambda x. \, \text{let } v \coloneqq x \, t \, \text{in } v \, \lambda y.v) \, \text{in } u \, \lambda z.u$$

This is further illustrated in Figure 1b. The two red forks are created by *let*-abstraction, as well as the dark blue fork. The two teal lines are sub-forks of the dark blue fork. Finally, using transitivity, fork equivalence is formed between all four connected sub-terms.

At this point, the definition of fork equivalence includes (1) *let*-abstraction, (2) equivalence between α-equal closed terms, (3) the sub-fork and (4) a transitive closure. It is not obvious that this relation is indeed sound and complete w.r.t. the intuitive notion of equivalence between subterms. To mitigate such doubts, we will now introduce a completely separate equivalence relation that will be shown equal to fork equivalence in Section 4.

1.3 Equivalence through Bisimulation

For our second equivalence relation, we will interpret λ-terms as directed graphs. The skeleton of the graph is formed by the abstract syntax tree of the λ-term, but instead of having variables or de Bruijn indices in the leaves, they have a pointer back to the location where the variable was bound.

(a) Subterms of Example 1.0.1 are bisimilar. Their nodes can be merged, resulting in variables without a unique de Bruijn index.

(b) Subterms of Example 1.0.3 are not bisimilar, shown by a counter-example.

Fig. 2. Illustrations of λ-terms as unordered graphs with labeled edges. Subject terms are related through a squiggly line. De Bruijn indices in variables are for illustrative purposes only.

With such an interpretation, it becomes possible to define context-sensitive α-equivalence as the well-known notion of the *bisimilarity relation* that is common in the analysis of labeled transition systems and many other (potentially) infinite structures [2]. As a refresher, we restate the definition of bisimilarity on a directed graph with labeled nodes and edges: A relation R between nodes is a bisimulation relation when for all nodes $(p_1, p_2) \in R$ the labels of p_1 and p_2 are equal and

$$\text{if } p_1 \xrightarrow{a} q_1 \text{ then there exists } q_2 \text{ such that } p_2 \xrightarrow{a} q_2 \text{ and } (q_1, q_2) \in R$$
$$\text{if } p_2 \xrightarrow{a} q_2 \text{ then there exists } q_1 \text{ such that } p_1 \xrightarrow{a} q_1 \text{ and } (q_1, q_2) \in R.$$

Two nodes p_1 and p_2 in a graph are then considered bisimilar if there exists a bisimulation relation R such that $(p_1, p_2) \in R$. The bisimilarity relation is the union of all bisimulation relations, and is itself a bisimulation.

Figure 2a shows the term in Example 1.0.1 as a graph. Due to the unordered nature of graphs, in contrast to more common presentation of syntax trees as ordered trees, edges are annotated with a shape that represents their label. It is straight-forward to build a relation R that includes the two subject terms and satisfies the requirements of a bisimulation relation. The existence of the bisimulation certifies that we can de-duplicate the two subject terms in the graph structure without changing their meaning. Note that as a result, variables can no longer be assigned a unique de Bruijn index.

Figure 2b shows how the subterms in Example 1.0.3 are not bisimilar. This is done by assuming a valid bisimulation relation R that contains the two subject terms. One can then simultaneously follow equal edges on both sides until two nodes with different labels are encountered that are not allowed to be in R (marked with a cross in the figure).

Deciding whether two subterms are equal is not always as easy as the illustrations in Figure 2 suggest. Consider the bisimulation question in Figure 3 between two variables. To decide whether the variables are bisimilar, we must repeatedly travel up and down into the term jumping between variables and their binders, until we reach the root. From there, we must travel into the subterms A and B. The two variables will be bisimilar if and only if A and B are bisimilar. This shows that no matter the size of the subject terms, one might need to traverse and examine the entire context in order to decide their bisimilarity. A decision procedure or hashing scheme must take all of this information into account while still being efficient.

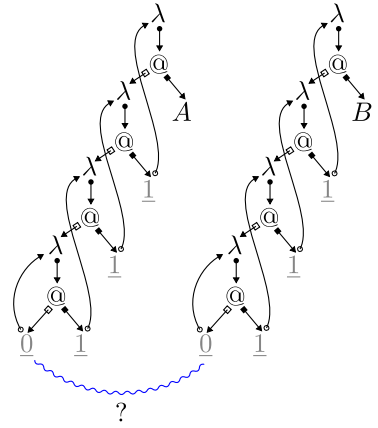
Fig. 3. To decide whether two variables are bisimilar, we must examine bisimilarity between far-away terms A and B.

Finally, we note the importance of having variable nodes in the embedding of λ-terms as graphs. Since variable nodes only have a single incoming edge and a single outgoing back-edge to the corresponding binder, one might be tempted to skip the variable node altogether. However, such an embedding would lead to a situation where we share too many terms. One example of such problems is the following sequence of terms that should not be bisimilar.

$$\lambda x.x \sim_{\text{b}} \lambda y.\lambda x.x \sim_{\text{b}} \lambda z.\lambda y.\lambda x.x \sim_{\text{b}} \ldots$$

However, under the following encoding, which omits variables, these terms would be bisimilar.

$$\lambda\,\curvearrowright\!\!\rightsquigarrow\quad \lambda\,\cdot\!\!\mapsto\lambda\,\curvearrowright\!\!\rightsquigarrow\quad \lambda\,\cdot\!\!\mapsto\lambda\,\cdot\!\!\mapsto\lambda\,\curvearrowright\quad \cdots$$

Fork equivalence and bisimilarity are defined quite differently. Indeed, it might be surprising that they end up being exactly equal. These two dissimilar characterizations of context-sensitive α-equivalence are crucial in the correctness proof of the hashing algorithm we present in Section 3. Figure 4 outlines how we will prove the equality between the algorithm and the two equivalences. Section 4 will show that if two subterms are fork-equivalent, they will receive the same hash. Conversely, if two subterms receive the same hash, they must be bisim-

Fig. 4. Proof strategy.

ilar. The first direction is considerably simplified because fork equivalence is characterized as a sequence of syntactically simple single forks. The other direction can be proven by showing that every λ-term is bisimilar to its hashed version (given an appropriate extension of bisimulation).
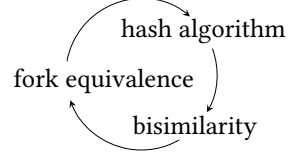
1.4 Hashing versus Partitioning Modulo Bisimulation

Given that λ-terms can be expressed as a deterministic labeled transition systems such that context-sensitive α-equivalence corresponds to bisimilarity, we can draw apply the large body of research in this area to λ-terms. Indeed, there are off-the-shelf algorithms that partition a deterministic graph $G = (E, V)$ modulo bisimulation in $O(|E| \log |V|)$ time [15, 19, 22, 27]. For a λ-term of size n, this translates into a $O(n \log n)$ runtime.

The algorithm we present in Section 3 does not partition a λ-term but rather assigns a hash-code to each position in the term. If no hash-collisions occur, two positions are context-sensitively α-equivalent if and only if their hash is equal. If desired, collisions can be detected by combining our algorithm with hash-consing [16]. As such, our algorithm is strictly more powerful than a partitioning algorithm. The crucial advantage is that λ-terms may be hashed independently of each other while the resulting hashes can still be meaningfully compared. This enables one to efficiently find duplicate terms as they are created by inserting their hashes in a table. Partitioning schemes do not allow this, because all terms must be simultaneously input into the algorithm. We are not aware of any existing graph-hashing algorithms that operate modulo bisimilarity.

Our hashing scheme is straight-forward to implement compared to general-purpose partitioning algorithms, and relies only on well-known λ-calculus operations such as capture-avoiding substitution. Section 5 shows it can outperform partitioning algorithms in all realistic scenarios.

1.5 Context-Sensitive α-Equivalence versus Ordinary α-Equivalence

We will now explore some of the differences between context-sensitive α-equivalence and ordinary α-equivalence. Previously, Maziarz et al [20] have constructed a hashing scheme modulo ordinary α-equivalence. How does this differ from our scheme and when is one relation or algorithm to be preferred over another?

To examine this question, we again look at the term in Example 1.0.5 as visualized in Figure 1b.

$$\lambda t.\ (\lambda x.\ x\ t\ (\lambda y.\ x\ t))\ (\lambda z.\ \lambda x.\ x\ t\ (\lambda y.\ x\ t))$$
$$\lambda t.\ (\lambda x.\ \underline{0}\ \underline{1}\ (\lambda y.\ \underline{1}\ \underline{2}))\ (\lambda z.\ \lambda x.\ \underline{0}\ \underline{2}\ (\lambda y.\ \underline{1}\ \underline{3}))$$

This term contains four instances of the term $(x\ t)$, all of whom are represented differently using de Bruijn indices, and all of whom are considered equal modulo context-sensitive α-equivalence. Which of these instances of $x\ t$ are equal under ordinary α-equivalence? This is a tricky question to answer. Both the variables x and t are free in the term $(x\ t)$. As such, under ordinary α-equivalence, they would be compared syntactically. This would indeed make all four instances α-equivalent. However, such an interpretation would get us into trouble when we rename the bound variables in the term. This may change the variable names in the subterms, which in turn may cause them to no longer be α-equivalent. Hence, α-equality between subterms is contingent on the particular choice

of bound variable names we have made. That is not acceptable. Maziarz et al. solve this ambiguity by globally enforcing the Barendregt convention on the entire universe of λ-terms. That is, no two binders may ever have the same name. The term in our example does not satisfy the Barendregt convention. We must rewrite it to

$$\lambda t.\ (\lambda x_1.\ x_1\ t\ (\lambda y_1.\ x_1\ t))\ (\lambda z.\ \lambda x_2.\ x_2\ t\ (\lambda y_2.\ x_2\ t)).$$

We now have two α-equivalent instances of $(x_1\ t)$ and two α-equivalent instances of $(x_2\ t)$. This is contrasted by context-sensitive α-equivalence, where all four terms are equal. In general, we have that ordinary α-equivalence is a sub-relation of context-sensitive α-equivalence[2]. For ordinary α-equivalence, this can lead to some counter-intuitive situations: The subterms $\lambda x_1.\ x_1\ t\ (\lambda y_1.\ x_1\ t)$ and $\lambda x_2.\ x_2\ t\ (\lambda y_2.\ x_2\ t)$ are considered α-equivalent according to Maziarz et al., but not all their constituent parts are mutually α-equivalent. No such issues exist for the context-sensitive variant.

Trade-offs between the two relations are as follows:

- Ordinary α-equivalence is simple. It is defined on λ-terms, while context-sensitive α-equivalence needs an additional context.
- Ordinary α-equivalence cannot be defined properly on open terms encoded with de Bruijn indices. Syntactic comparisons between open de Bruijn indices leads to incorrect results (see Example 1.0.3). A hybrid approach, such as a locally-nameless representation [9], is required to resolve this. Context-sensitive α-equivalence can be properly defined for any representation of λ-terms.
- When interpreting λ-terms as a graph, one should use context-sensitive α-equivalence because it equals the graph-theoretic notion of bisimilarity.
- For tasks like common subterm elimination in compilers, both relations may be appropriate because there one usually seeks to find the largest α-equivalent subterms and not their descendants. Both relations achieve this.

Furthermore, the trade-offs between the hashing algorithm in this paper and the one by Maziarz et al. are as follows:

- Our algorithm hashes all nodes in a term in $O(n \log n)$ time while Maziarz et al. require $O(n \log^2 n)$ time.
- Maziarz et al. require a global preprocessing step to enforce the Barendregt convention. No such step is required for our algorithm.
- The algorithm by Maziarz et al. is compositional. That is, given two hashed terms t and u, one can derive the hash for $(t\ u)$ from the hash of the children. This may be a desirable property in a compiler, allowing one to maintain correct hashes while rewriting terms during optimization passes[3]. Compositionality is not possible for context-sensitive α-equivalence, because changing the context may require a change in the hash of all subterms (see Figure 3).
- Maziarz et al. rely fundamentally on named variables, while we rely fundamentally on de Bruijn indices. In both cases, it is possible to do a representation conversion before hashing, but both algorithms have a clear "native" representation. Much has been said around the relative merits of λ-term representations [5], and we do not wish to make a value judgement here. It is good to know there are viable algorithms for both representational approaches, even if those algorithms have subtle differences in how they operate.

---

[2]A direct comparison of the two relations is impossible because one is defined on the domain of λ-terms, while the other is defined on λ-terms with a context. However, one can imagine a trivial lift of ordinary α-equivalence to λ-terms with a context, such that the context is entirely ignored.

[3]Re-hashing a term after rewriting a term at depth h may still be expensive, taking up to $O(h^2)$ time.
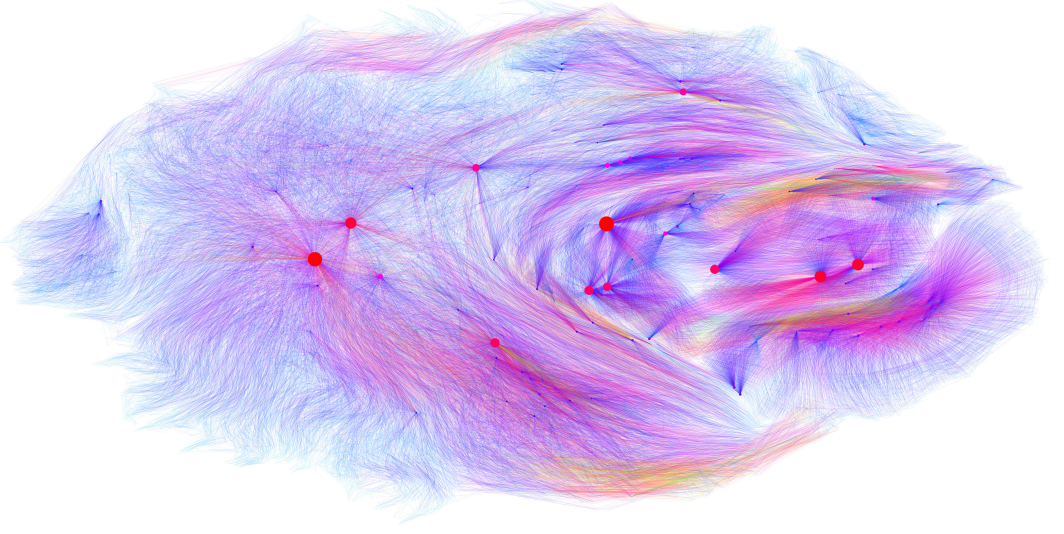
Fig. 5. A visualization of a maximally shared graph of CIC terms extracted from Coq's Prelude.

1.6 Applications

The original motivation for the subterm sharing algorithm in this paper was the creation of a large-scale, graph-based machine learning dataset of terms in the calculus of inductive constructions, exported from the Coq Proof Assistant [25]. Extracting data from over a hundred Coq developments leads to a single, interconnected graph containing over 520k definitions, theorems and proofs. For each node in this graph we calculate a hash modulo context-sensitive α-equivalence. The hash is then used to maximally share all subterms, resulting in a dense graph with approximately 250 million nodes. A very small section of this graph is visualized in Figure 5. More details on the construction can be found elsewhere [7]. Experiments show that subterm sharing allows for an 88% reduction in the number of nodes. Hence, without sharing, such a graph would have over 2 billion nodes. Identifying identical subterms in a graph is helpful semantic information that can be used by machine learning algorithms to make predictions. The graph dataset has been leveraged to train a state of the art graph neural network to synthesize proofs in Coq [23].

In addition to our original motivation, we expect our algorithm to be useful in other applications as well. In compilers, common sub-expression elimination is a common optimization pass [12] that can be performed quickly using our algorithm. This was the original motivation for the algorithm by Maziarz et al. [20]. Hashes can also be used by content addressable programming languages like Unison [10] and to build indexes of program libraries that can then be queried to find opportunities for code sharing [26] or plagiarism detection [28].

1.7 Contributions

In this paper, we develop a notion of context-sensitive α-equivalence that compares potentially open terms within a context. We define this notion both through fork equivalence and bisimulation, and prove that these approaches are equivalent. Note that we are not the first to study bisimilarity on the syntax of λ-terms. In the context of optimal sharing for efficient evaluation this is common [13, 17]. However, our relational characterization between subterms with a context appears to be novel. We present an efficient decision procedure and hashing algorithm that identifies terms modulo context-sensitive α-equivalence. These algorithms have been successfully used to efficiently encode

a graph of Coq terms for machine learning purposes. A reference implementation written in OCaml is available [6].

The remainder of the paper is organized as follows. Section 2 first introduces the preliminaries followed by the formal definitions of fork equivalence in Section 2.4 and bisimilarity in Section 2.5. Then, Section 3.1 presents a simple, naive algorithm for hashing λ-terms in $O(n^2)$ time. This is then refined in Section 3.2 and finally a concrete, $O(n \log n)$ hashing algorithm implemented in OCaml is presented in Section 3.3. Proofs of equality between the two relations and the algorithms can be found in Section 4.

2 DEFINITIONS

In this section, we will further formalize the equivalence relations presented in the introduction. From this point, we will only consider λ-terms encoded with de Bruijn indices. The algorithms rely heavily on the fact that two α-equivalent closed terms encoded using de Bruijn indices are syntactically equal. That said, the equivalence relations and the proofs of equality between them also work when one uses λ-terms with named variables modulo ordinary α-equivalence. For the sake of legibility, we will frequently still give examples using λ-terms with named variables.

2.1 Terms, Positions and Indexing

Definition 2.1 (λ-terms). λ-terms with de Bruijn indices are generated by the grammar

$$t ::= \underline{i} \mid t \, t \mid \lambda \, t,$$

where a de Bruijn index $\underline{i}$ is a nonnegative integer. We denote $t[\underline{i} := u]$ to be the well-known capture-avoiding substitution of variable $\underline{i}$ by u in t. Furthermore, let $\sigma = [u_0, u_1, \ldots, u_{n-1}]$ be a list of terms. Then $t\sigma$ denotes the simultaneous subtitution of all variables $\underline{i}$ by u_i in t.

Definition 2.2 (term indexing). Let p be a *term position* generated by the grammar $\{\downarrow, \nearrow, \searrow\}^*$. The indexing operation $t[p]$ is a partial function defined by

$$(t \, u)[\nearrow p] = t[p] \qquad (t \, u)[\searrow p] = u[p] \qquad (\lambda \, t)[\downarrow p] = t[p] \qquad t[\varepsilon] = t.$$

Example 2.3. Consider a term $t = \lambda \, (\lambda \, A \, B) \, (\lambda \, C \, D)$ and positions $p = \downarrow \nearrow$ and $q = \downarrow \searrow$. Then

$$t[\varepsilon] = \lambda \, (\lambda \, A \, B) \, (\lambda \, C \, D) \qquad t[pp] = A \qquad t[pq] = B \qquad t[qp] = C \qquad t[qq] = D.$$

Definition 2.4 (position sets). Let $|p|_\lambda$ denote the number of $\downarrow$s in a term position p. We define the following sets of positions for a λ-term t.

$\mathbb{P}(t) = \{p \mid t[p] \text{ is defined}\}$	The set of all valid positions in t.
$\mathbb{V}(t) = \{p \in \mathbb{P}(t) \mid t[p] = \underline{i}\}$	Positions that represent a variable in t.
$\mathbb{B}(t) = \{p \in \mathbb{V}(t) \mid t[p] < |p|_\lambda\}$	Positions that represent a bound variable.
$\mathbb{F}(t) = \{p \in \mathbb{V}(t) \mid t[p] \geq |p|_\lambda\}$	Positions that represent a free variable.

2.2 Locally Closed Subterms

A subterm rooted at position p in t is considered to be *locally closed* in t if all of the free variables in $t[p]$ are also free in t. We will later use this concept while formalizing *let*-abstraction.

Definition 2.5. Position $p \in \mathbb{P}(t)$ is *locally closed* in t if for every $q \in \mathbb{F}(t[p])$ we have $pq \in \mathbb{F}(t)$.

Example 2.6. Consider again Example 1.0.1 as visualized in Figure 1a:

$$u = \lambda t. \, Q \, (\lambda z. \lambda f. \, f \, t) \, (\lambda g. \, g \, t).$$

Let $p = \downarrow\nearrow\searrow$, giving us $u[p] = \lambda z.\lambda f.\ f\ t$ and $u[p\downarrow] = \lambda f.\ f\ t$. The position $p\downarrow$ is locally closed in $u[p]$, because the variable f is bound in $u[p\downarrow]$, and t is free in both $u[p]$ and $u[p\downarrow]$. On the other hand, $p\downarrow$ is not locally closed in u, because variable t is free in $u[p\downarrow]$ but bound in u.

Using this notion, we can introduce a more semantic version of term indexing. Intuitively, $t\langle p\rangle$ will denote the subterm $t[p]$ with its de Bruijn indices shifted to skip the context given by p. This way, we "lift" $t[p]$ out of its context. This can only be done if p is locally closed in t.

Definition 2.7. For $p \in \mathbb{P}(t)$, define $t\langle p\rangle$ to be equal to $t[p]$, except that for all $q \in \mathbb{F}(t[p])$ we have $t\langle p\rangle[q] = t[pq] - |p|_\lambda$.

The term $t\langle p\rangle$ is a valid term (with non-negative indices) only when p is locally closed in t. The operation $t\langle p\rangle$ is crucial in much of the analysis below, as it allows us to abstract away from manually doing arithmetic on de Bruijn indices. Another natural view of $t\langle p\rangle$ is that it reverses capture-avoiding substitution, as shown in the following observation.

Observation 2.8. Let t be a λ-term with a free variable $\underline{i}$ at position p. That is, $p \in \mathbb{F}(t)$ such that $t[p] = \underline{i}$. Then $t[\underline{i} := u]\langle p\rangle = u$.

Note that $t[\underline{i} := u][p] = u$ does not hold, because the capture-avoiding substitution may have shifted the free de Bruijn indices in u. The semantic indexing operation reverses these shifts.

Example 2.9. Consider the term $t = \lambda\ \underline{0}\ (\lambda\ \underline{0}\ \underline{2})$, where $\underline{2}$ is a free variable. The position $\downarrow\searrow$ is locally closed in t because the outer-most λ is not referenced. We then have $t\langle\downarrow\searrow\rangle = \lambda\ \underline{0}\ \underline{1}$ compared to $t[\downarrow\searrow] = \lambda\ \underline{0}\ \underline{2}$. On the other hand, neither $\downarrow$ nor $\downarrow\nearrow$ are locally closed positions.

2.3 Term Nodes

We now formally define the notion of a term with a context as introduced in Section 1.2.

Definition 2.10 (term node). Let $t[\![p]\!]$ denote a pair (t, p) such that t is a closed term, and $p \in \mathbb{P}(t)$.

In a pair $t[\![p]\!]$, the term $t[p]$ represents the *subject*, while the remaining part of t is the *context*. Intuitively, we can think of $t[\![p]\!]$ as the subterm at $t[p]$ but without losing the information about the context.

We call a pair $t[\![p]\!]$ a *term node* because it represents a node in the graph induced on λ-terms that we introduced in Section 1.3. In the graph visualizations in Section 1.3, each node is labeled only with the top-most symbol of the subject term, either $\downarrow$, @, or $\uparrow$. Although this is convenient from a visualization perspective, it does not work from a mathematical perspective where a graph is defined as a pair of sets $G = (V, E)$ that determine the nodes and edges. Taking the set of nodes to be $V = \{\downarrow, @, \uparrow\}$ would give us trivial graphs with three nodes. Instead, a node $n \in V$ must uniquely represent the subject term and the context in which it occurs. This is exactly what a term node is.

In order to formally define the graph on λ-terms, we must also define the set E of transitions between term nodes.

Definition 2.11 (term node transitions). We define the transitions between term nodes as follows. Let $t[\![p]\!]$ be a term node, and $x \in \{\downarrow, \nearrow, \searrow\}$ such that $x \in \mathbb{P}(t[p])$. Then,

$$t[\![p]\!] \xrightarrow{x} t[\![px]\!].$$

In addition, a term node whose subject is a variable also has a transition to the corresponding binder. Formally, if $q\downarrow r \in \mathbb{V}(t)$ and $t[q\downarrow r] = \underline{|r|_\lambda}$, then

$$t[\![q\downarrow r]\!] \xrightarrow{\uparrow} t[\![q]\!].$$

Note that for a term node $t[\![p]\!]$ such that $p \in \mathbb{V}(t)$, we can always make a split $p = q \downarrow r$ such that $t[p] = \lfloor r \rfloor_\lambda$. This is because the definition of a term node $t[\![p]\!]$ stipulates that t is closed and hence $|p|_\lambda > t[p]$.

Now, we have all the tools needed to formally specify the two equal notions of context-sensitive α-equivalences outlined in the introduction.

2.4 Fork Equivalence

Here, we formalize the concepts introduced in Section 1.2, starting with the notion of a single fork.

Definition 2.12 (single fork). A *single fork* between term nodes, denoted by $t_1[\![p_1]\!] \sim_{\mathrm{sf}} t_2[\![p_2]\!]$, exists when one of the following rules is satisfied.

$$\frac{q_1 \text{ locally closed in } t[p] \quad t[p]\langle q_1 \rangle = t[p]\langle q_2 \rangle}{t[\![pq_1r]\!] \sim_{\mathrm{sf}} t[\![pq_2r]\!]} \; let\text{-abs} \qquad \frac{t_1[p_1] \text{ closed} \quad t_1[p_1] = t_2[p_2]}{t_1[\![p_1r]\!] \sim_{\mathrm{sf}} t_2[\![p_2r]\!]} \; \text{closed}$$

It is assumed that $t[\![pq_1r]\!]$, $t[\![pq_2r]\!]$, $t_1[\![p_1r]\!]$ and $t_2[\![p_2r]\!]$ are valid term nodes in the rules above.

When $t[\![pq_1r]\!] \sim_{\mathrm{sf}} t[\![pq_2r]\!]$ is satisfied by the *let*-abs rule, this means that a *let* can be introduced in t at position p. The *let* binds the term $t[p]\langle q_1 \rangle$, and the terms at position pq_1 and pq_2 can be changed into a variable pointing to the *let*. Example 1.0.2 illustrates this. The rule for closed terms is simpler. It states that a closed term is equivalent to itself in an arbitrary context. Finally, note that the conclusion of both rules allow for an arbitrary position r that "extends" the prongs of a known fork, as illustrated in the following observation:

Observation 2.13 (sub-fork). $t_1[\![p_1]\!] \sim_{\mathrm{sf}} t_2[\![p_2]\!]$ implies $t_1[\![p_1r]\!] \sim_{\mathrm{sf}} t_2[\![p_2r]\!]$ for $r \in \mathbb{P}(t_1[p_1])$.

The relation for a single fork is reflexive. For every term node $t[\![p]\!]$, a self-fork can be constructed using the *let*-abs rule by taking $q_1 = q_2 = \varepsilon$. In addition, a fork is symmetric. Transitivity does not hold, however. To obtain an equivalence, we take the transitive closure.

Definition 2.14 (fork equivalence). $t_1[\![p_1]\!] \sim_{\mathrm{f}} t_2[\![p_2]\!]$ is the transitive closure of $t_1[\![p_1]\!] \sim_{\mathrm{sf}} t_2[\![p_2]\!]$.

2.5 Bisimilarity

In addition to fork equivalence, we also formalize the bisimulation relation introduced in Section 1.3.

Definition 2.15 (bisimilarity). A binary relation R on term nodes is called a *bisimulation* if for every pair of term nodes $(n_1, n_2) \in R$ and every $x \in \{\downarrow, \nearrow, \searrow, \uparrow\}$ the following holds:

- If $n_1 \xrightarrow{x} n_1'$, then there exists n_2' such that $n_2 \xrightarrow{x} n_2'$ and $(n_1', n_2') \in R$.
- If $n_2 \xrightarrow{x} n_2'$, then there exists n_1' such that $n_1 \xrightarrow{x} n_1'$ and $(n_1', n_2') \in R$.

Two term nodes $t_1[\![p_1]\!]$ and $t_2[\![p_2]\!]$ are *bisimilar*, denoted $t_1[\![p_1]\!] \sim_{\mathrm{b}} t_2[\![p_2]\!]$, if there exists a bisimulation R such that $(t_1[\![p_1]\!], t_2[\![p_2]\!]) \in R$.

It is well-known that bisimilarity is an equivalence relation, and that it is itself a bisimulation. Note that unlike the informal definition in Section 1.3, this bisimulation relation does not directly compare the labels of term nodes (the label of a node $t[\![p]\!]$ would be the root symbol of $t[p]$). This is not needed, because the label of a node is fully determined by the labels of its outgoing edges. This, together with the fact that the transition system is deterministic, considerably simplifies our setup compared to arbitrary transition systems. In particular, the notion of *bisimulation* coincides with the notion of *simulation*, in which the second clause of Definition 2.15 is omitted.

One of the main results of this paper, proved in Section 4, is that fork equivalence and bisimilarity are equal.

Theorem 2.16. $t_1[\![p_1]\!] \sim_\text{b} t_2[\![p_2]\!]$ *if and only if* $t_1[\![p_1]\!] \sim_\text{f} t_2[\![p_2]\!]$.

Definition 2.17 (context-sensitive α-equivalence)**.** Since the main equivalence notion is captured both by $\sim_\text{f}$ and $\sim_\text{b}$, we will use $t_1[\![p_1]\!] \sim t_2[\![p_2]\!]$ to denote context-sensitive α-equivalence and switch between the two interpretations at will.

3 DECIDING CONTEXT-SENSITIVE α-EQUIVALENCE THROUGH GLOBALIZATION

As we saw in the introduction, using syntactic equality on λ-terms with de Bruijn indices is problematic in the presence of free variables. Such variables need to be interpreted within a context in order to be meaningful. Our approach to deciding whether or not two terms are α-equivalent in a given context is to *globalize* the variables. We replace all de Bruijn indices in a term with *global variables*, which are structures that contain exactly the required information to capture the context that is relevant to the variable. After globalization, we can indeed compare two subterms syntactically without having to consider the context in which they exist, because that context has been internalized into the variables.

As it happens, the structure associated to a global variable is itself a λ-term that may contain de Bruijn indices or further global variables. This leads us to extend the grammar of λ-terms into that of g-terms.

Definition 3.1 (g-terms)**.** A g-term is generated by the grammar

$$t ::= \underline{i} \mid t\ t \mid \lambda\ t \mid \text{g}(t)$$

where a term of the form $\text{g}(t)$ represents a *global variable* labeled by a g-term t. We consider any λ-term to also be a g-term, and trivially lift all operations and relations defined on λ-terms:

- Substitution is extended such that $\text{g}(t)[\underline{i} := u] = \text{g}(t)$, without traversing into t.
- Term indexing $t[p]$ behaves identical to λ-terms. Indexing does not extend into the structure of a global variable. The functions $t\langle p\rangle$, $\mathbb{P}(t)$, $\mathbb{V}(t)$, $\mathbb{F}(t)$ and $\mathbb{B}(t))$ remain defined as before. Global variables are not part of the set $\mathbb{V}(t)$. A global variable $\text{g}(u)$ is always closed.
- The definition for $\sim_\text{f}$ remains as before. The definition of $\sim_\text{b}$ is extended, but we postpone this until Section 4.

We are now ready to describe algorithms that transform a λ-term into a g-term that respects context-sensitive α-equivalence. We start with a slow, naive algorithm which is then made more efficient. Finally, we present a practical OCaml program that processes a term and attaches an appropriate hash to every subterm.

3.1 A Naive Globalization Procedure

Contrary to the relation $t_1[\![p_1]\!] \sim t_2[\![p_2]\!]$, the globalization procedure does not operate on term nodes but rather on closed g-terms. This works, because closed terms do not require a context.

Definition 3.2 (naive globalization)**.** Recursively define $\text{globalize}_\text{naive}(t)$ from closed g-terms to closed g-terms as follows.

$$
\begin{aligned}
\text{globalize}_\text{naive}(\lambda\ t) &= \lambda\ \text{globalize}_\text{naive}(t[\underline{0} := \text{g}(\lambda\ t)]) \\
\text{globalize}_\text{naive}(t\ u) &= \text{globalize}_\text{naive}(t)\ \text{globalize}_\text{naive}(u) \\
\text{globalize}_\text{naive}(\text{g}(t)) &= \text{g}(t)
\end{aligned}
$$

Due to the pre-condition on $\text{globalize}_\text{naive}(t)$ that t must be closed, there is no need for a case for de Bruijn indices in the equations above (a bare de Bruijn index is not closed). The pre-condition is maintained in the recursion due to a substitution in case a λ is encountered.

Example 3.3. The globalization of the term $\lambda\,\lambda\,\underline{0}\,\underline{1}$ proceeds as follows:

$$\text{globalize}_{\text{naive}}(\lambda\,\lambda\,\underline{0}\,\underline{1}) = \lambda\,\text{globalize}_{\text{naive}}(\lambda\,\underline{0}\,g(\lambda\,\lambda\,\underline{0}\,\underline{1})) = \lambda\,\lambda\,g(\lambda\,\underline{0}\,g(\lambda\,\lambda\,\underline{0}\,\underline{1}))\,g(\lambda\,\lambda\,\underline{0}\,\underline{1})$$

In order to understand how this algorithm works, we will first state the final theorem that relates the algorithm to context-sensitive α-equivalence.

Theorem 3.4 (correctness of globalize$_{\text{naive}}$). *For λ-term nodes $t_1[\![p_1]\!]$ and $t_2[\![p_2]\!]$ we have $t_1[\![p_1]\!] \sim t_2[\![p_2]\!]$ if and only if $\text{globalize}_{\text{naive}}(t_1)[p_1] = \text{globalize}_{\text{naive}}(t_2)[p_2]$.*

The full proof of this theorem is postponed until Section 4. Here, we present a intuitive argument for why the algorithm works. The crux of the algorithm lies in the property that closed λ-terms encoded with de Bruijn indices are α-equivalent if and only if they are syntactically equal.

Lemma 3.5 (correctness of closed terms). *For closed terms t_1, t_2 we have $t_1[\![\varepsilon]\!] \sim t_2[\![\varepsilon]\!]$ iff $t_1 = t_2$.*

See Lemma A.8 for a proof. Because the input of globalize$_{\text{naive}}$ is always closed, this lemma guarantees that the input term can already be correctly compared. When a binder is encountered, we simply embed this known-correct structure into a global variable and substitute it for any de Bruijn index that references the binder. After the substitution, the subterm of the binder is again closed and correct with respect to α-equivalence. By processing the entire term, all de Bruijn indices are replaced with a global variable. At this point, every subterm is closed and can therefore be compared syntactically with other (properly globalized) terms in order to determine equality.

3.2 Efficient Globalization

The speed of globalize$_{\text{naive}}(t)$ is dominated by the substitution we must perform when we encounter a binder. Performing a substitution takes a linear amount of time for a given term. Furthermore, a term of size n may contain up to $O(n)$ binders. Therefore, in the worst case, globalize$_{\text{naive}}(t)$ takes quadratic time.

Example 3.6. Consider the following pathological term of size $3n - 1$, containing n binders.

$$\lambda x_1.\,\lambda x_2.\,\lambda x_3.\,\ldots\,\lambda x_n.\,x_n\,\ldots\,x_3\,x_2\,x_1.$$

The algorithm performs a substitution every time it encounters one of the n binders. Further, each substitution must traverse a term whose size is at least n, resulting in at least n^2 steps.

To speed this up, we would like to perform substitutions more lazily. If we accumulate substitutions in a list σ, we can delay performing the substitution $t\sigma$ until it is absolutely necessary.

How do we determine when σ needs to be substituted? In the naive algorithm, we rely on the property that the input term is always closed. Due to lazy substitutions, we can no longer guarantee this. However, if a term is known to not be α-equivalent to any other term we might be interested in comparing it to, even without the globalization substitutions, we can postpone the substitution. Speeding up the algorithm relies on finding sufficiently many subterms where we can skip substitution. Indeed, there are numerous simple summaries that can be used to determine when a term is "unique enough" among a set of other terms to skip the substitution step.

Definition 3.7 (term summary). *Let $|\cdot|$ be a function on g-terms to an arbitrary co-domain such that $t_1[\![p_1]\!] \sim_{\text{f}} t_2[\![p_2]\!]$ implies $|t_1[p_1]| = |t_2[p_2]|$.*

We will use term summaries in the contrapositive. That is, if the summary $|t[p]|$ of a subterm is unique among the summaries of any other relevant subterm, then it is not α-equivalent to any of these subterms. We use the notation $|\cdot|$ for term summaries because a rather useful example of a summary is the size of the term: Any two α-equivalent subterms are guaranteed to have the same

size. A stronger example of a term summary is the set of paths $\mathbb{P}(\cdot)$ of a term. Conversely, a rather weak example is the constant function that maps every term to the same object. We need a summary that is cheap to compute and compare, while distinguishing as many terms as possible. The constant function is cheap but clearly distinguishes nothing. On the other hand, $\mathbb{P}(\cdot)$ distinguishes many terms but is expensive to compute and compare. The size of a term is a good middle ground. It is cheap to compute, cache, and compare while still distinguishing many terms.

Lemma 3.8. The constant function, $\mathbb{P}(\cdot)$, and the size of a term are valid term summaries.

PROOF. Straightforward from the definition of fork equivalence and Definition 2.7.   □

We will use the term summary to find unique and non-unique terms. However, we have not yet specified the background set to which a term should be compared for uniqueness. Initially, one might think that we need to compare against the entire infinite universe of potential λ-terms. Fortunately, that is not the case. We can limit the set among which we need to compare to the *strongly connected component* of a term node.
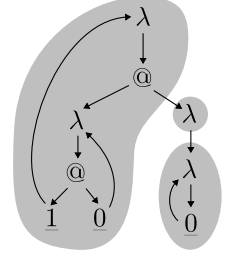
Definition 3.9 (strongly connected component). For a closed g-term t, define the *strongly connected component* SCC(t) as the set of all positions $p \in \mathbb{P}(t)$ where for every nonempty prefix p' of p, $t[p']$ is not closed.

We borrow the name "strongly connected component" from graph theory. In particular, given a closed term t, the set

$$\{t[\![q]\!] \mid q \in \mathrm{SCC}(t)\}$$

forms the strongly connected sub-graph of nodes rooted in t.

Example 3.10. The term $t = \lambda x.(\lambda y.\ x\ y)\ (\lambda y.\lambda z.z)$ is closed, and therefore we can ask what its strongly connected component is. Also note that t contains two other closed subterms, for which we also have a strongly connected component. As such, there are three SCCs associated

to t and its subterms, as visualized on the right. Strongly connected components are disjoint and form a tree. SCCs may be singletons if the children of the root are closed.

Notice that because SCCs form a tree of closed terms, they can be processed independently. If we know a procedure to globalize a single SCC, then we can invoke this procedure recursively, either starting from the top-most SCC and working our way down or the other way around. As such, we have reduced the problem of globalization to individual strongly connected components. (This is a common reduction for bisimulation algorithms.)

To efficiently globalize a SCC, we will calculate the set of subterms in the SCC whose summary has one or more duplicate in the SCC. We will only need to perform globalization substitutions in duplicate subterms because syntactic equality is already strong enough to properly distinguish a non-duplicate term from all other terms in the SCC.

Definition 3.11 (duplicate SCC subterms). Let duplicates(t) denote all strict subterms in the strongly connected component of t whose term summary is not unique within the SCC.

$$\mathrm{duplicates}(t) = \{t[p] \mid p,q \in \mathrm{SCC}(t) \land p,q \neq \varepsilon \land p \neq q \land |t[p]| = |t[q]|\}$$

One might object that this only guarantees globalization to give correct results when comparing two subterms within the same SCC. What about two non-equivalent subterms that do not share the same SCC? How do we guarantee that such terms are not syntactically equal? For this, notice that two subterms can only be α-equivalent if all terms in their strongly connected component are

also pairwise α-equivalent. In particular, the root of their respective SCCs must be α-equivalent. Because the root of a SCC is closed, it may be safely syntactically compared. As such, where the naive algorithm would substitute a g-var $g(t)$, we can safely amend this to $g(r\ t)$, where r is the root of the strongly connected component of t. This will prevent us from inappropriately declaring two terms with different SCCs to be α-equivalent. We can now state an efficient globalization procedure.

Definition 3.12 (efficient globalization). Recursively define globalize, globalize$_{\text{scc}}$ and globalize$_{\text{step}}$:

$$\text{globalize}(r) \qquad\qquad = \text{globalize}_{\text{scc}}(r, [], r)$$

$$\text{globalize}_{\text{scc}}(r, \sigma, \lambda\ t) = \lambda\ \text{globalize}_{\text{step}}(r, (g(r\ t) : \sigma), t)$$

$$\text{globalize}_{\text{scc}}(r, \sigma, t\ u) = \text{globalize}_{\text{step}}(r, \sigma, t)\ \text{globalize}_{\text{step}}(r, \sigma, u)$$

$$\text{globalize}_{\text{scc}}(r, \sigma, g(t)) = g(t)$$

$$\text{globalize}_{\text{scc}}(r, \sigma, \underline{i}) \qquad = \underline{i}\sigma$$

$$\text{globalize}_{\text{step}}(r, \sigma, t) \quad = \begin{cases} \text{globalize}(t\sigma) & \text{if } t \text{ is closed or } t \in \text{duplicates}(r) \\ \text{globalize}_{\text{scc}}(r, \sigma, t) & \text{otherwise} \end{cases}$$

For globalize(r), we maintain the precondition that r is closed, while for globalize$_{\text{scc}}(r, \sigma, t)$ and globalize$_{\text{step}}(r, \sigma, t)$ we maintain the precondition that $t\sigma$ is closed. Furthermore, it holds that there exists a position $p \in \text{SCC}(r)$ such that $r[p] = t$. That is, r is the root of the SCC in which t resides. Finally, for globalize$_{\text{scc}}(r, \sigma, t)$, we expect that $t \notin \text{duplicates}(r)$.

Notice how the algorithm is defined mutually recursively between globalize, globalize$_{\text{scc}}$ and globalize$_{\text{step}}$. There are two possible recursive paths, either back and forth between globalize$_{\text{scc}}$ and globalize$_{\text{step}}$, or with an detour through globalize. Every time the algorithm calls globalize$(t\sigma)$, it crosses from one SCC to another. This either happens because t was already closed (and hence the start of a new SCC), or a new SCC was created by performing the substitution $t\sigma$ because a duplicate was found. The subsitution closes the term, creating a new SCC.

Similar to globalize$_{\text{naive}}(t)$, we now claim that globalize(t) behaves correctly with respect to context-sensitive α-equivalence.

Theorem 3.13 (correctness of globalize). For λ-term nodes $t_1[\![p_1]\!]$ and $t_2[\![p_2]\!]$ we have $t_1[\![p_1]\!] \sim t_2[\![p_2]\!]$ if and only if globalize$(t_1)[p_1] = $ globalize$(t_2)[p_2]$.

We again postpone the full proof of this theorem to Section 4. Although the explanations around the algorithm in this section should provide a solid intuition, an airtight correctness proof requires more extended reasoning. To further build intuition about this more elaborate algorithm, the following observation shows that it is (nearly) a generalization of the naive algorithm.

Observation 3.14. When one instantiates the term summary $|\cdot|$ with a constant function, the globalize(t) function reduces to a function that is very similar to globalize$_{\text{naive}}(t)$. The only difference is the precise substitution being performed when a binder is encountered:

$$\text{globalize}_{\text{naive}}(\lambda\ t) = \lambda\ \text{globalize}_{\text{naive}}(t[\underline{0} := g(\lambda\ t)])$$

$$\text{globalize}(\lambda\ t) = \lambda\ \text{globalize}(t[\underline{0} := g(t\ t)])$$

Both substitutions lead to correct results. In fact, it is sufficient to simply substitute the g-var $g(t)$. The variations $g(\lambda\ t)$ and $g(t\ t)$ do not change the distinguishing power of the g-var.

Now, given that the algorithm is known to behave correctly, we must ask the question whether we have actually gained a substantial speed improvement. In the beginning of this section, we attributed the source of inefficiency for the naive algorithm to excessive substitutions. Interestingly, assuming that we instantiate $|\cdot|$ to be term-size, the worst-case scenario presented in Example 3.6 has now become a best-case scenario. This is because the tree-structure of the example is almost entirely linear. The only subterms with equal size are the variables (which all have size 1). Therefore, a non-trivial duplicate is never encountered and substitution is only trivially triggered when reaching a variable. Now, the substitutions take $O(n)$ time instead of $O(n^2)$.[4]

Conversely, the best-case (non-trivial) scenario for $\text{globalize}_{\text{naive}}$ has now become the worst-case scenario. Such a scenario involves a λ-term that forms a perfectly balanced tree, where (almost) all subterms have a direct sibling that is equal in size. This would cause the efficient algorithm to trigger a substitution on every step. However, because the tree is now balanced, most substitutions are performed on a small subterm. The substitutions then take at most $O(n \log n)$ time.

To formalize this worst-case bound, we will show that each node in the syntax tree is visited at most $O(\log n)$ times by the substitution function.[5] This is facilitated by assuming that every g-term t is annotated with a counter that is increased whenever the substitution function traverses it. The visit count is retrieved using $\text{sv}(t)$ and reset to zero (for all subterms) using $\text{sr}(t)$. We can then prove the following efficiency lemma.

Lemma 3.15. Let $n = \text{sv}(\text{globalize}(\text{sr}(t))[p])$. Then $|t| \geq 2^n$.

PROOF. By induction on n. The base case is trivial. For $n > 0$, we must unfold the algorithm until we reach the point where the first substitution occurs. Indeed, assuming that $u\sigma$ does not traverse into u when u is closed, a simple helper lemma can show the existence of r, σ, q and s such that

$$\text{globalize}(t)[p] = \text{globalize}_{\text{step}}(r, \sigma, r[q])[s] = \text{globalize}(r[q]\sigma)[s]$$

$$|t| \geq |r| \qquad r[q] \in \text{duplicates}(r) \qquad \text{sv}(\text{globalize}(\text{sr}(r[q]\sigma))[s]) = n - 1.$$

From the induction hypothesis, we then know that $|r[q]| = |r[q]\sigma| \geq 2^{n-1}$. Furthermore, we know there exists q' different from q such that $|r[q']| = |r[q]|$. It then follows easily that $|t| \geq |r| \geq 2^n$. □

Observation 3.16. The function $\text{globalize}_{\text{step}}(r, \sigma, t)$ may sometimes substitute σ even in cases where t is not a binder. This is unnecessary. We can amend the algorithm to only perform a substitution when a binder is encountered. This will speed up the algorithm, but not asymptotically so. This optimization does somewhat complicate the proof of correctness. We omit these details.

3.3 A Concrete Hashing Implementation

Although the efficient algorithm from the previous section can be shown to be correct, there are some practical and theoretical shortcomings:

- The algorithm is not concrete enough to fully analyze its asymptotic complexity. In particular, the function $\text{duplicates}(t)$ is too abstract.
- The use of g-terms to compare subterms for α-equivalence is unsatisfactory because equality checking on g-terms takes $O(n)$ time. Instead, we would like to calculate a hash that can be compared in $O(1)$ time (at the expense of potential collisions).
- The globalization process transforms λ-terms into g-terms, destroying any de Bruijn indices. This makes it difficult to further use the term as a normal λ-term. (Even though it is

---

[4]If substitution lists σ are implemented using arrays, lookup and push operations take $O(1)$ amortized time.

[5]We analyze the cost of other functions, such as $\text{duplicates}(\cdot)$ in Section 3.3.4.

technically possible to recover the original λ-term from a globalized term, this is a non-trivial operation). We would like a globalization function that assigns appropriate hashes to each node of an λ-term, without modifying the term itself.

Here we present a more concrete algorithm implemented in the OCaml programming language. A complete, executable reference implementation is available [6].

3.3.1 Datastructures. We start with the definition of terms. We will need several variants of λ-terms. To easily define them in a common framework, we define them using a *term functor*:

```
type 'a termf = Lam of 'a | Var of int | App of 'a * 'a [@@deriving map, fold]
```

Instead of defining a term through direct recursion, we rather "tie the knot" in this term functor. This allows us to decorate a term with additional information when we need it by "adding it to the knot." As an example, the simplest recursive knot we can tie represents a pure, ordinary λ-term with no additional information:

```
type pure_term = pure_term termf
```

By adding an extra constructor GVar to the knot, we can also define a structure that is isomorphic to *g*-terms:

```
type gterm = Term of gterm termf | GVar of gterm
```

For our algorithm, we must efficiently calculate quite a few properties of terms, including whether they are closed, their size and a hash. Information related to this must be stored in each node of a term. Instead of providing a concrete implementation for this, we rather posit the existence of an abstract type term that is assumed to store all the required information. A concrete implementation of this type can be found in supplementary material [6]. It comes with functions lift and case that allows us to convert it to and from the term functor, so that we can pattern match on it.

```
type term     val lift : term termf -> term     val case : term -> term termf
```

The function case is the left inverse of lift, that is case (lift t) = t. We do not have lift (case t) = t, because information stored in t may be thrown away by case. To illustrate how lift and case are used, we will write the functions from_pure and to_pure that convert a pure_term into a term and vice versa. For this, we will need the map_termf function that has been automatically derived for the term functor along with a fold_termf function. They have the following signature.

```
val map_termf  : ('a -> 'b) -> 'a termf -> 'b termf
val fold_termf : ('a -> 'b -> 'a) -> 'a -> 'b termf -> 'a
```

We can use map_termf, lift and case to write the following recursive conversion functions.

```
let rec from_pure (t : pure_term) : term = lift (map_termf from_pure t)
let rec to_pure   (t : term) : pure_term = map_termf to_pure (case t)
```

The from_pure takes an ordinary λ-term, and lifts it into a term decorated with information about term size, closedness and more. Calculating the required information for this decoration happens in lift. The to_pure function does the opposite, because case will forget any decorations that may be stored in the term.

The most important decoration of term is the hash we will assign to each node through globalization. We consider two possible datatypes for a hash. We can use integers as a hash if we want a datatype that is fast to compare, but with the risk of encountering a collisions. When a collision is not acceptable, we can use gterm as a hash. Here, we keep the datatype for hash abstract (but keeping in mind our two target implementations):

```
type hash     val lift_hash : hash termf -> hash     val hash_gvar : hash -> hash
```

In case hash is instantiated to be a gterm, we implement lift_hash and hash_gvar as follows.

```
let lift_hash h = Term h and hash_gvar h = GVar h
```

We assume a hash can be retrieved from any term via function hash, with the following contract.

```
val hash : term -> hash
hash (lift t) = lift_hash (map_termf hash) t
```

This means that when we convert a pure_term into a term, the hash for that term corresponds to the Merkle-style hash of its syntactic structure (including de Bruijn indices). The idea of the globalization algorithm is to adjust these hashes by annotating de Bruijn indices with a corrected, globalized hash. To this end, we stipulate an alternative function for building a variable term with a custom hash.

```
val gvar : hash -> int -> term
case (gvar h i) = Var i    &&    hash (gvar h i) = hash_gvar h
```

Finally, we assume that we can retrieve the size of a term, and check if a term is gclosed. This function returns false if and only if the given term contains any free variable which was not built with gvar.

```
val size : term -> int          val gclosed : term -> bool
```

Figure 6 summarizes the datastructures we have built. A pure_term is isomorphic to a mathematical λ-term, and a hash is isomorphic to a g-term (if the hash is instantiated as a gterm and not an int). One can see a term as a pair that contains a pure_term and a hash.

3.3.2 Calculating Duplicates.
We now turn our attention to the efficient calculation of duplicates(r) from Definition 3.11. Note that this set actually contains more terms than we need. In particular, for any $t \in$ duplicates(r), we have no need for further sub-terms of t to be included in the set. This is because the algorithm is guaranteed to transition from globalize$_{\text{scc}}$ to globalize once it encounters t, which means that a new SCC with different duplicates will become active. It is not difficult to show that the algorithm behaves identically when we omit these irrelevant terms.

Fig. 6. OCaml datastructures and their mathematical counterparts as a commutative diagram.

To efficiently calculate this reduced set of duplicate node terms, we require the property that $|t| > |t[p]|$ for any $p \in \mathbb{P}(t)$. This is satisfied by instantiating the term summary with term size: The size of a subterm of t is smaller than the size of t itself. We can now find duplicates by inserting terms into a priority queue keyed to the size of the terms. We start with a singleton queue that only contains the root of an SCC. Then, we retrieve all terms whose key is equal to the largest key in the queue. Initially, this is only the root of the SCC. If we have retrieved multiple terms, we know that they are duplicates of each other. If we have retrieved only a single term, we know that it cannot have a duplicate because all other terms in the queue are smaller. We then insert the children of that unique term into the queue. We iteratively retrieve and re-insert items into the queue until we have exhausted all terms in the SCC. In OCaml code, this procedure is as follows.

```
val Heap.pop_multiple : Heap.t -> int * term list * Heap.t

let calc_duplicates (t : term) : IntSet.t =
  let step q t = if gclosed t then q else Heap.insert t q in
```
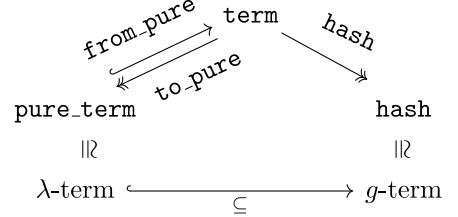
```
let rec aux queue =
  match Heap.pop_multiple queue with
  | None -> IntSet.empty
  | Some (_, [t], queue) -> aux (fold_termf step queue (case t))
  | Some (size, _, queue) -> IntSet.add size (aux queue)
in aux (Heap.insert t Heap.empty)
```

Note that unlike duplicates(t), calc_duplicates(t) does not output a set of duplicate terms. Instead, it outputs a set of duplicate sizes. To check if a term is duplicated in a SCC, one can simply check if the size of that term is duplicated.

3.3.3   Globalization. Before we can define our globalize function, we must first define an OCaml equivalent to substitutions. On the mathematical level, we substitute g-vars for de Bruijn indices. The corresponding concept on the OCaml level is to decorate a de Bruijn index with a hash. This is done through the function set_hash:

```
let rec set_hash (n : int) (h : hash) (t : term) : term =
  if gclosed t then t (* do not modify existing g-vars *)
  else match case t with
    | Lam t -> lift (Lam (set_hash (n+1) h t))
    | Var i -> if n = i then gvar h i else t
    | t      -> lift (map_termf (set_hash n h) t)
```

A substitution $t[\underline{i} := g(u)]$ can be seen as roughly equivalent to set_hash i u t. In addition to setting a single hash, we must have the ability to set a sequence of hashes, similar to a substitution $t\sigma$. For this, we have a datatype hashes, which is morally just a list of hashes. However, a naive linked list would be too inefficient for lookup. A more efficient implementation based on sets is out of scope of this text. Instead, we specify hashes as an abstract datatype with the following functions.

```
type hashes                  val push_hash  : hashes -> hash -> hashes
val empty_hashes : hashes    val set_hashes : hashes -> term -> term
```

A simultaneous substitution $t\sigma$ can be seen as roughly equivalent to set_hashes sigma t.

We are now ready to write our globalization function in OCaml. The following is essentially a direct transliteration of the equations from Definition 3.12 to OCaml.

```
let rec globalize (r : term) : term =
  let duplicates = calc_duplicates r in
  let rec globalize_scc (s : hashes) (t : term) =
    match case t with
    | Lam t' ->
      let s = push_hash s (hash (lift (App (r, t)))) in
      lift (Lam (globalize_step s t'))
    | Var _ -> set_hashes s t
    | t -> lift (map_termf (globalize_step s) t)
  and globalize_step s t =
    let t = if IntSet.mem (size t) duplicates then set_hashes s t else t in
    if gclosed t then globalize t else globalize_scc s t
  in globalize_scc empty_hashes r
```

Following the correctness statement for the mathematical version of the algorithm in Theorem 3.13, we can state the correctness of the OCaml version as follows. Note that this theorem relies on extending term indexing $t[p]$ to the OCaml realm.

Theorem 3.17. Let $t_1[\![p_1]\!]$, $t_2[\![p_2]\!]$ be two term nodes, and t1, t2 the canonical embeddings of t_1, t_2 as an OCaml term. If $t_1[\![p_1]\!] \sim_b t_2[\![p_2]\!]$, then

$$\text{hash } ((\text{globalize } t1)[p_1]) = \text{hash } ((\text{globalize } t2)[p_2])$$

The reverse implication is true if lift_hash and hash_gvar are injective, and have disjoint images.

We state this theorem without further proof. However, it is straightforward to verify that

$$\text{hash } ((\text{globalize } (\text{from\_pure } t))[p]) = \text{globalize}(t)[p]$$

if one instantiates the type hash with gterm. This provides a clear link between the mathematical algorithm and the OCaml algorithm.

3.3.4 Algorithmic Complexity. We will now show that the algorithm presented in Section 3.3.3 runs in $O(n \log n)$ time, where n is the size of the term being globalized. In Lemma 3.15 we already showed that the set_hashes function touches each node at most $O(\log n)$ times. Furthermore, when a variable is encountered for which a hash should be set, the lookup for the correct hash can be done in $O(\log n)$ time. There are at most n variables, and each variable is assigned a hash exactly once. This demonstrates that the total cost of set_hashes remains within the budget.

To analyze the remaining functions, note that the traversal performed by the mutually recursive functions globalize, globalize_scc and globalize_step visits every node of a term exactly once. As such, it suffices to verify that each invoked helper function spends no more than $O(\log n)$ time per node. For most helper functions, like gclosed, size and IntSet.mem this is easy to verify.

The function calc_duplicates is a bit more tricky. This function is invoked once each time globalize is called with a fresh SCC. Its goal is to calculate the set of nodes where we transition back from globalize_scc to globalize. As such, it touches exactly the same set of nodes as the subsequent call to globalize_scc. Therefore, we can attribute the time taken for each node by calc_duplicates to this function call. Processing a node entails inserting it into a queue in $O(\log n)$ time and eventually retrieving it from the queue in $O(\log n)$ time. Therefore, we stay within the available $O(\log n)$ time budget.

4 SKETCH OF CORRECTNESS PROOFS

The next three theorems give a sketch how to prove that bisimulation is equal to fork-equivalence, and that the globalization algorithm is correct. More fleshed out versions of these theorems can be found in Appendix A. Each theorem is responsible for one of the implication arrows in Figure 4. We assume that $t_1[\![p_1]\!]$, $t_2[\![p_2]\!]$ are two λ-term nodes.

Theorem 4.1. If $t_1[\![p_1]\!] \sim_b t_2[\![p_2]\!]$, then $t_1[\![p_1]\!] \sim_f t_2[\![p_2]\!]$.

PROOF SKETCH. When two term nodes are bisimilar, we must find a sequence of places in their contexts where *let*-abstractions can be introduced until the subjects become equal. Each *let*-abstraction represents a single fork. Then, through transitivity, this sequence of single forks demonstrates fork equivalence. Finding this sequence of single forks proceeds by strong induction on the path p_1. That is, we assume the theorem holds for all strict prefixes of p_1. Then we make a split $p_1 = p_{1,0} \downarrow p_{1,1}$ such that $p_{1,1}$ is locally closed in $t_1[p_{1,0}]$ and there exists a free variable $v \in \mathbb{F}(t_1[p_1])$ that references the binder
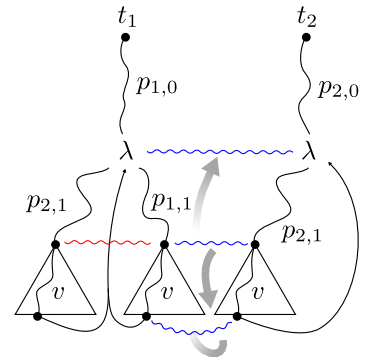

Fig. 7. Schematic proof of Theorem 4.1. Blue connections denote bisimilarity, red a single fork.

at $t_1[p_{1,0}]$.[6] This situation is illustrated in Figure 7. The bisimulation relation then guarantees that a similar split $p_2 = p_{2,0} \downarrow p_{2,1}$ can be made such that $t_1[\![p_{1,0}]\!] \sim_{\mathrm{b}} t_2[\![p_{2,0}]\!]$. These two splits represent the bottom-most location where we introduce a *let*-abstraction. All the remaining *let*-abstractions that need to be introduced along the paths $p_{1,0}$ and $p_{2,0}$ are established through the induction hypothesis, which allows us to obtain $t_1[\![p_{1,0}]\!] \sim_{\mathrm{f}} t_2[\![p_{2,0}]\!]$ and hence also $t_1[\![p_{1,0} \downarrow p_{2,1}]\!] \sim_{\mathrm{f}} t_2[\![p_{2,0} \downarrow p_{2,1}]\!]$. We now only need to establish the single fork $t_1[\![p_{1,0} \downarrow p_{1,1}]\!] \sim_{\mathrm{sf}} t_1[\![p_{1,0} \downarrow p_{2,1}]\!]$ (illustrated by a red connection in Figure 7). This fork can be established using a technical lemma that relies on the fact that $p_{1,1}$ is locally closed in $t_1[p_{1,0}]$. □

Theorem 4.2. If $t_1[\![p_1]\!] \sim_{\mathrm{f}} t_2[\![p_2]\!]$, then $\mathrm{globalize}(t_1)[p_1] = \mathrm{globalize}(t_2)[p_2]$.

PROOF SKETCH. It suffices to show that two term nodes related through a single fork become equal after globalization. The full theorem then follows from transitivity of Leibniz equality. Hence, we need to show the conclusion assuming that the term nodes follow one of the two rules in Definition 2.12. For the second rule, where both subjects $t_1[p_1]$ and $t_2[p_2]$ are closed and equal, the conclusion follows readily because

$$\mathrm{globalize}(t_1)[p_1] = \mathrm{globalize}(t_1[p_1]) = \mathrm{globalize}(t_2[p_2]) = \mathrm{globalize}(t_2)[p_2].$$

This holds, because the globalization procedure only modifies the terms through substitutions, which cannot influence the closed subjects.

Proving correctness for the first rule is more technical, but ultimately relies on the same strategy, where we move the indexing of positions p_1 and p_2 from outside globalize to inside globalize. □

Theorem 4.3. If $\mathrm{globalize}(t_1)[p_1] = \mathrm{globalize}(t_2)[p_2]$, then $t_1[\![p_1]\!] \sim_{\mathrm{b}} t_2[\![p_2]\!]$.

PROOF SKETCH. Here, we rely on a conservative extension of the bisimulation relation to g-terms such that we can show

$$t[\![p]\!] \sim_{\mathrm{b}} \mathrm{globalize}(t)[p][\![\varepsilon]\!].$$

That is, modulo this new bisimulation relation, the globalization algorithm does not modify the term at all. The final theorem then follows trivially. To make this work, we add an extra transition from nodes whose subject are a g-var to their corresponding binder. This transition does not use the context, but rather the knowledge about the context that has been stored inside the g-var by the globalization procedure. □

5 EXPERIMENTAL EVALUATION

We evaluate and compare the runtime of our algorithm with three kinds of synthetic λ-term. First, we uniformly sample closed terms of a fixed size [4, 18]. Second, we generate the unbalanced terms from Example 3.6. Third, are perfectly balanced terms such that binders and applications are alternated. These latter two represent two extreme cases an algorithm must handle.

The left plot of Figure 8 compares the naive algorithm of Section 3.1 to the efficient algorithm of Section 3.2. The trend shows that all terms can be globalized in roughly $O(n \log n)$ time. The naive algorithm takes $O(n^2)$ time, with the exception of the best-case scenario of balanced terms.

The right plot of Figure 8 compares our algorithm with Valmari's deterministic finite automaton minimization algorithm [27] and the hashing algorithm of Maziarz et al. [20]. One should note that these comparisons are not apples-to-apples, see Section 1.4 and 1.5. The plot includes a version of our algorithm with and without hash-consing. The hash-consing version should be compared to Valmari's algorithm, as it can be used to assign equivalence classes to term nodes without collisions.

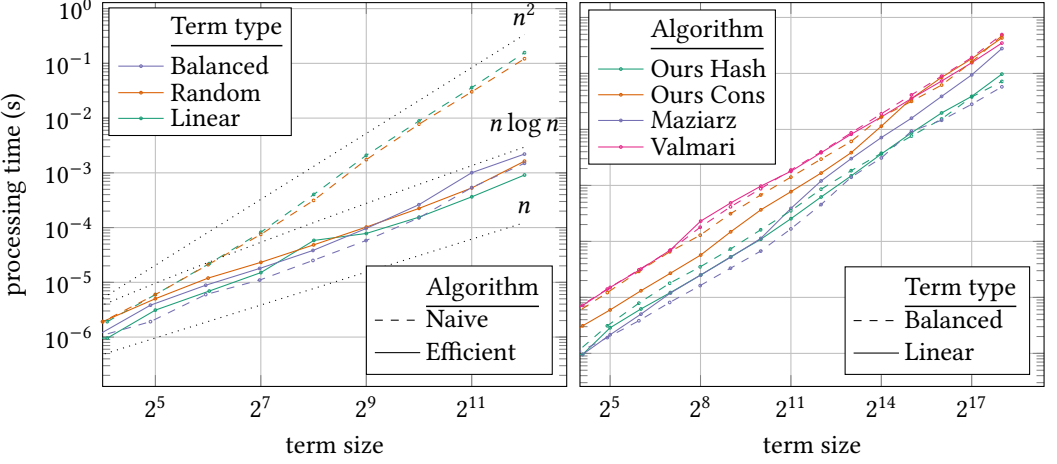[6]If no such split exists, $t_1[p_1]$ is closed, making the theorem trivial.

Fig. 8. Performance of several algorithms on synthetic λ-terms.

We see that hash-consing g-terms imposes a significant performance overhead. Nevertheless, it is still competitive with Valmari's algorithm. The performance of our algorithm is close to Maziarz'. For linear terms, we can see the extra $O(\log n)$ runtime factor emerge in Maziarz' algorithm.

6 RELATED AND FUTURE WORK

Our work should primarily be compared to previous work by Maziarz et al. [20] and bisimulation algorithms [15, 19, 22, 27]. This comparison can be found in Section 1.4 and 1.5. Here we give an overview of further related work and future research.

Term Sharing Algorithms. Term sharing is a common approach as a means of memory saving. However, in most cases, these techniques do not take into account α-equality. In compilers, sharing the structure of a languages AST is often achieved using hash-consing [16]. Hash-consing allows for sub-structure sharing between terms, but shared terms are not guaranteed to be "equal" according to any reasonable equivalence relation. The FLINT compiler [24] is an example where hash-consing is employed aggressively to save space.

The literature is rather sparse with respect to term-sharing modulo α-equivalence. Condoluci, Accattoli and Coen present a decision procedure to check α-equivalence of two terms in which sub-terms may be shared in linear time [13]. This is an important result that may be used for efficient convertibility checking in dependently typed proof assistants such as Coq, LEAN and Agda [14, 21, 25] in combination with efficient reduction algorithms that employ sharing [1, 8]. However, their algorithm only allows pairwise comparisons of terms. It does not show how to efficiently find all α-equivalent subterms.

Hashing of Graphs. We are not aware of existing work in labeled transition systems that calculates a bsimimulation-respecting hash for each node. Such a hash would be useful in the analysis of large-scale graphs, in which calculating the entire bisimulation relation at once may not be feasible. As such, an interesting open question is how far our algorithm can be generalized for arbitrary graphs. The graphs induced by λ-calculus are only a subset of the set of di-graphs. It is guaranteed that during a traversal of a graph from the root, any binder is reached before a variable that refers to that binder. Extending λ-calculus with mutually recursive fixpoints eliminates this property. In such an extension, variables can no longer be represented with de Bruijn indices, invalidating

our algorithm. An algorithm capable of hashing such terms is future work, as is extending the algorithm to arbitrary (non-)deterministic transition systems.

ACKNOWLEDGMENTS

REFERENCES

[1] Beniamino Accattoli and Ugo Dal Lago. 2016. (Leftmost-Outermost) Beta Reduction is Invariant, Indeed. *Log. Methods Comput. Sci.* 12, 1 (2016). https://doi.org/10.2168/LMCS-12(1:4)2016

[2] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of model checking.* MIT Press.

[3] Hendrik Pieter Barendregt. 1985. *The lambda calculus - its syntax and semantics.* Studies in logic and the foundations of mathematics, Vol. 103. Elsevier, North-Holland.

[4] Maciej Bendkowski. 2020. How to generate random lambda terms? *CoRR* abs/2005.08856 (2020). arXiv:2005.08856 https://arxiv.org/abs/2005.08856

[5] Stefan Berghofer and Christian Urban. 2006. A Head-to-Head Comparison of de Bruijn Indices and Names. In *Proceedings of the First International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, LFMTP@FLoC 2006, Seattle, WA, USA, August 16, 2006 (Electronic Notes in Theoretical Computer Science, Vol. 174),* Alberto Momigliano and Brigitte Pientka (Eds.). Elsevier, 53–67. https://doi.org/10.1016/j.entcs.2007.01.018

[6] Lasse Blaauwbroek. 2023. *Reference Implementation for Hashing Modulo Context-Sensitive Alpha-Equivalence.* https://doi.org/10.5281/zenodo.11097757

[7] Lasse Blaauwbroek. 2024. The Tactician's Web of Large-Scale Formal Knowledge. *arXiv preprint* (Jan. 2024). https://doi.org/10.48550/arXiv.2401.02950 arXiv:2401.02950 [cs.LO]

[8] Guy E. Blelloch and John Greiner. 1995. Parallelism in Sequential Functional Languages. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture, FPCA 1995, La Jolla, California, USA, June 25-28, 1995,* John Williams (Ed.). ACM, 226–237. https://doi.org/10.1145/224164.224210

[9] Arthur Charguéraud. 2012. The Locally Nameless Representation. *J. Autom. Reason.* 49, 3 (2012), 363–408. https://doi.org/10.1007/S10817-011-9225-2

[10] Paul Chiusano, Rúnar Bjarnason, and Arya Irani. [n. d.]. *Unison: A friendly, statically-typed, functional programming language from the future · UNISON programming language.* https://www.unison-lang.org/

[11] Alonzo Church. 1941. *The Calculi of Lambda-Conversion.* Princeton: Princeton University Press.

[12] John Cocke. 1970. Global common subexpression elimination. In *Proceedings of a Symposium on Compiler Optimization, Urbana-Champaign, Illinois, USA, July 27-28, 1970,* Robert S. Northcote (Ed.). ACM, 20–24. https://doi.org/10.1145/800028.808480

[13] Andrea Condoluci, Beniamino Accattoli, and Claudio Sacerdoti Coen. 2019. Sharing Equality is Linear. In *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019,* Ekaterina Komendantskaya (Ed.). ACM, 9:1–9:14. https://doi.org/10.1145/3354166.3354174

[14] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9195),* Amy P. Felty and Aart Middeldorp (Eds.). Springer, 378–388. https://doi.org/10.1007/978-3-319-21401-6_26

[15] Agostino Dovier, Carla Piazza, and Alberto Policriti. 2004. An efficient algorithm for computing bisimulation equivalence. *Theor. Comput. Sci.* 311, 1-3 (2004), 221–256. https://doi.org/10.1016/S0304-3975(03)00361-X

[16] Jean-Christophe Filliâtre and Sylvain Conchon. 2006. Type-safe modular hash-consing. In *Proceedings of the ACM Workshop on ML, 2006, Portland, Oregon, USA, September 16, 2006,* Andrew Kennedy and François Pottier (Eds.). ACM, 12–19. https://doi.org/10.1145/1159876.1159880

[17] Clemens Grabmayer and Jan Rochel. 2014. Maximal sharing in the Lambda calculus with letrec. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014,* Johan Jeuring and Manuel M. T. Chakravarty (Eds.). ACM, 67–80. https://doi.org/10.1145/2628136.2628148

[18] Katarzyna Grygiel and Pierre Lescanne. 2013. Counting and generating lambda terms. *J. Funct. Program.* 23, 5 (2013), 594–628. https://doi.org/10.1017/S0956796813000178

[19] John Hopcroft. 1971. An n log n algorithm for minimizing states in a finite automaton. In *Theory of machines and computations*. Elsevier, 189–196.

[20] Krzysztof Maziarz, Tom Ellis, Alan Lawrence, Andrew W. Fitzgibbon, and Simon Peyton Jones. 2021. Hashing modulo alpha-equivalence. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 960–973. https://doi.org/10.1145/3453483.3454088

[21] Ulf Norell. 2009. Dependently typed programming in Agda. In *Proceedings of TLDI'09: 2009 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, GA, USA, January 24, 2009*, Andrew Kennedy and Amal Ahmed (Eds.). ACM, 1–2. https://doi.org/10.1145/1481861.1481862

[22] Robert Paige and Robert Endre Tarjan. 1987. Three Partition Refinement Algorithms. *SIAM J. Comput.* 16, 6 (1987), 973–989. https://doi.org/10.1137/0216062

[23] Jason Rute, Miroslav Olšák, Lasse Blaauwbroek, Fidel Ivan Schaposnik Massolo, Jelle Piepenbrock, and Vasily Pestun. 2024. Graph2Tac: Learning Hierarchical Representations of Math Concepts in Theorem proving. *arXiv preprint* (Jan. 2024). https://doi.org/10.48550/arXiv.2401.02949 arXiv:2401.02949 [cs.LG]

[24] Zhong Shao, Christopher League, and Stefan Monnier. 1998. Implementing Typed Intermediate Languages. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998*, Matthias Felleisen, Paul Hudak, and Christian Queinnec (Eds.). ACM, 313–323. https://doi.org/10.1145/289423.289460

[25] The Coq Development Team. 2020. *The Coq Proof Assistant.* https://doi.org/10.5281/zenodo.4021912

[26] Mikkel Jonsson Thomsen and Fritz Henglein. 2012. Clone detection using rolling hashing, suffix trees and dagification: A case study. In *Proceeding of the 6th International Workshop on Software Clones, IWSC 2012, Zurich, Switzerland, June 4, 2012*, James R. Cordy, Katsuro Inoue, Rainer Koschke, Jens Krinke, and Chanchal K. Roy (Eds.). IEEE Computer Society, 22–28. https://doi.org/10.1109/IWSC.2012.6227862

[27] Antti Valmari. 2012. Fast brief practical DFA minimization. *Inf. Process. Lett.* 112, 6 (2012), 213–217. https://doi.org/10.1016/J.IPL.2011.12.004

[28] Jingling Zhao, Kunfeng Xia, Yilun Fu, and Baojiang Cui. 2015. An AST-based Code Plagiarism Detection Algorithm. In *10th International Conference on Broadband and Wireless Computing, Communication and Applications, BWCCA 2015, Krakow, Poland, November 4-6, 2015*, Leonard Barolli, Fatos Xhafa, Marek R. Ogiela, and Lidia Ogiela (Eds.). IEEE Computer Society, 178–182. https://doi.org/10.1109/BWCCA.2015.52

A PROOFS

The three sections that follow give a more detailed proof of each of the theorems sketched in Section 4.

A.1 Bisimilarity implies Fork Equivalence

We start with some preliminary observations and lemmas about the sets $\mathbb{P}(\cdot)$, $\mathbb{V}(\cdot)$ and $\mathbb{F}(\cdot)$ and how they relate to term indexing.

Observation A.1. Various subsets of term paths can be derived from the paths of its subterms:

$$\mathbb{P}(\lambda\, t) = \{\downarrow p \mid p \in \mathbb{P}(t)\} \cup \{\varepsilon\} \qquad \mathbb{P}(t\, u) = \{\swarrow p \mid p \in \mathbb{P}(t)\} \cup \{\searrow p \mid p \in \mathbb{P}(u)\} \cup \{\varepsilon\}$$

$$\mathbb{V}(\lambda\, t) = \{\downarrow p \mid p \in \mathbb{V}(t)\} \qquad \mathbb{V}(t\, u) = \{\swarrow p \mid p \in \mathbb{V}(t)\} \cup \{\searrow p \mid p \in \mathbb{V}(u)\}$$

$$\mathbb{F}(\lambda\, t) = \{\downarrow p \mid p \in \mathbb{F}(t) \wedge t[p] \neq \underline{|p|_\lambda}\} \qquad \mathbb{F}(t\, u) = \{\swarrow p \mid p \in \mathbb{F}(t)\} \cup \{\searrow p \mid p \in \mathbb{F}(u)\}$$

Lemma A.2. If $p \in \mathbb{P}(t)$ then $\mathbb{P}(t) = \mathbb{P}(t[p])$ implies $p = \varepsilon$.

  Proof. By induction on p, using Observation A.1. □

  The next three preliminary lemmas state that if two term nodes are bisimilar, then the position sets $\mathbb{P}(\cdot)$, $\mathbb{V}(\cdot)$, $\mathbb{F}(\cdot)$ and $\mathbb{B}(\cdot)$ of their subjects must be equal. These lemmas are important, because they show that bisimilar nodes have largely the same structure. One can see the set $\mathbb{P}(t)$ as the "skeleton" of t, where the contents of leafs (variables) are ignored. If subjects do not have the same skeleton, there is no hope of forming a bisimulation between them.

Lemma A.3. If $t_1[\![p_1]\!] \sim_b t_2[\![p_2]\!]$ then $\mathbb{P}(t_1[p_1]) = \mathbb{P}(t_2[p_2])$.

  Proof. We have a bisimulation R with $(t_1[\![p_1]\!], t_2[\![p_2]\!]) \in R$. Proceed by induction on $t_1[p_1]$.

  Case $t_1[p_1] = \underline{i}$: Relation R mandates that there exists j such that $t_2[p_2] = \underline{j}$. The conclusion is then trivially true.

  Case $t_1[p_1] = \lambda\, u$: We have $t_1[\![p_1]\!] \xrightarrow{\downarrow} t_1[\![p_1 \downarrow]\!]$. Furthermore, from R, we have that $t_2[\![p_2]\!] \xrightarrow{\downarrow} t_2[\![p_2 \downarrow]\!]$. The induction hypothesis then gives us $\mathbb{P}(t[p_1 \downarrow]) = \mathbb{P}(t[p_2 \downarrow])$. Finally, we conclude $\mathbb{P}(t[p_1]) = \mathbb{P}(t[p_2])$ with the help of Observation A.1.

  Case $t_1[p_1] = u\, v$: Analogous to the previous case. □

  Whereas the previous lemma shows that bisimilar subjects must have equal skeletons, the next lemma shows that their variables must also be related. In the introduction, we showed that the de Bruijn indices of bisimilar subjects are not always equal. Nevertheless, positions that represent free variables in one subject must also be free positions in the other subject. (The same fact holds for bound variables because $\mathbb{B}(\cdot)$ is the complement of $\mathbb{F}(\cdot)$.)

Lemma A.4. If $t_1[\![p_1]\!] \sim_b t_2[\![p_2]\!]$ then $\mathbb{F}(t_1[p_1]) = \mathbb{F}(t_2[p_2])$.

  Proof. We have a bisimulation R with $(t_1[\![p_1]\!], t_2[\![p_2]\!]) \in R$. Proceed by induction on $t_1[p_1]$. Cases for variables and application proceed straightforward. The interesting case occurs when $t_1[p_1] = \lambda\, u$. From R and the induction hypothesis, we obtain $\mathbb{F}(t_1[p_1 \downarrow]) = \mathbb{F}(t_2[p_2 \downarrow])$. Observation A.1 shows that it now suffices to prove

$$\{q \mid q \in \mathbb{F}(t_1[p_1 \downarrow]) \wedge t_1[p_1 \downarrow q] \neq \underline{|q|_\lambda}\} = \{q \mid q \in \mathbb{F}(t_2[p_2 \downarrow]) \wedge t_2[p_2 \downarrow q] \neq \underline{|q|_\lambda}\}.$$

In other words, it suffices to prove that if $q \in \mathbb{F}(t_1[p_1 \downarrow])$ and $t_1[p_1 \downarrow q] = \underline{|q|_\lambda}$, then $t_2[p_2 \downarrow q] = \underline{|q|_\lambda}$. Without loss of generality, we assume $t_2[p_2 \downarrow q] < \underline{|q|_\lambda}$ to obtain a contradiction. We can then make a split $q = q_0 \downarrow q_1$ such that

$$t_2[\![p_2]\!] \sim_b t_1[\![p_1]\!] \sim_b t_2[\![p_2 \downarrow q_0 \downarrow]\!].$$

Then, from Lemma A.3 we have $\mathbb{P}(t_2[p_2]) = \mathbb{P}(t_2[p_2 \downarrow q_0 \downarrow])$. Finally, Lemma A.2 gives a contradiction. □

Next is a lemma that shows that if two subjects are bisimilar, their sub-structures must also be bisimilar. This is the analogous lemma to Observation 2.13 for fork equivalence.

Lemma A.5. If $q \in \mathbb{P}(t_1[p_1])$ and $t_1[\![p_1]\!] \sim_{\mathrm{b}} t_2[\![p_2]\!]$ then $t_1[\![p_1q]\!] \sim_{\mathrm{b}} t_2[\![p_2q]\!]$.

PROOF. Straightforward by induction on q. □

As a final preliminary lemma, we note that if the subject of a term node is closed, then its context is irrelevant. This is shown by establishing a bisimulation relation between the node, and a modification such that the context is thrown away.

Lemma A.6. If t and $t[p]$ are closed, then $t[\![p]\!] \sim_{\mathrm{b}} t[p][\![\varepsilon]\!]$.

PROOF. Construct the relation

$$R = \{(t[\![pq]\!], t[p][\![q]\!]) \mid q \in \mathbb{P}(t[p])\}.$$

Verifying that R is a bisimulation relation is straightforward. The only case of note is when $t[pq]$ is a variable. We know that the binder corresponding to the variable is a subterm of $t[p]$, because that term is closed. Hence, we can verify that this binder is bisimilar to itself under R. □

We are now ready to prove the main technical "workhorse" lemma for this section. The following lemma extracts the required information for a bisimulation relation in order to establish a single fork. The conclusion of this lemma corresponds closely to the required conditions in Definition 2.12 to build a single fork. Note that the addition of the index into position r is a technical requirement to make the induction hypothesis sufficiently strong. When the lemma is used, we always set $r = \varepsilon$.

Lemma A.7. If $t[\![pq_1]\!] \sim_{\mathrm{b}} t[\![pq_2]\!]$ and q_1 is locally closed in $t[p]$, then $t[p]\langle q_1\rangle[r] = t[p]\langle q_2\rangle[r]$.

PROOF. Note that from Lemma A.5 we have

$$t[\![pq_1r]\!] \sim_{\mathrm{b}} t[\![pq_2r]\!]. \tag{A.7.1}$$

Proceed by induction on $t[pq_1r]$. In case $t[pq_1r]$ is a variable, we perform additional case analysis on whether r is bound or free.

Case $t[pq_1r] = \underline{i}$ **such that** $r \in \mathbb{F}(t[pq_1])$**:** Because r is free, and q_1 is locally closed in $t[p]$, we know that $q_1r \in \mathbb{F}(t[p])$. Therefore, there exists a split $p = p_1 \downarrow p_2$ such that

$$t[\![p_1 \downarrow p_2q_1r]\!] \overset{\uparrow}{\longrightarrow} t[\![p_1]\!].$$

The bisimulation relation from Equation A.7.1 additionally mandates that $t[pq_2r] = \underline{j}$ for some j. Without loss of generality, assume $i \leq j$. We then know that there exists s such that

$$t[\![pq_2r]\!] \overset{\uparrow}{\longrightarrow} t[\![p_1s]\!] \qquad\qquad t[\![p_1]\!] \sim_{\mathrm{b}} t[\![p_1s]\!].$$

From Lemma A.3 and Lemma A.2 we then have $s = \varepsilon$, and as such

$$t[pq_1r] = \underline{|p_2q_1r|}_\lambda \qquad\qquad t[pq_2r] = \underline{|p_2q_2r|}_\lambda.$$

We then conclude that

$$t[p]\langle q_1\rangle[r] = \underline{|p_2q_1r|}_\lambda - |q_1|_\lambda = \underline{|p_2q_2r|}_\lambda - |q_2|_\lambda = t[p]\langle q_2\rangle[r].$$

Case $t[pq_1r] = \underline{i}$ **such that** $r \in \mathbb{B}(t[pq_1])$**:** From the main bisimulation hypothesis and Lemma A.4 we have $\mathbb{F}(t[pq_1]) = \mathbb{F}(t[pq_2])$. Since $\mathbb{B}(\cdot)$ is the complement of $\mathbb{F}(\cdot)$, we know that $r \in \mathbb{B}(t[pq_2])$. The bisimulation relation from Equation A.7.1 then mandates that there exist two splits $r = r_{1A} \downarrow r_{1B} = r_{2A} \downarrow r_{2B}$ such that

$$t[pq_1r] = \underline{|r_{1B}|_\lambda} \qquad t[pq_2r] = \underline{|r_{2B}|_\lambda} \qquad t[\![pq_1r_{1A}]\!] \sim_b t[\![pq_2r_{2A}]\!].$$

Moreover, from Lemma A.3 we have $\mathbb{P}(t[pq_1r_{1A}]) = \mathbb{P}(t[pq_2r_{2A}])$. Now, without loss of generality, assume $|r_{1A}| \geq |r_{2A}|$. We can then make an additional split $r_{1A} = r_{2A}s$, giving us

$$\mathbb{P}(t[pq_1r_{2A}s]) = \mathbb{P}(t[pq_2r_{2A}]) \qquad r = r_{2A}s \downarrow r_{1B} = r_{2A} \downarrow r_{2B} \qquad s \downarrow r_{1B} =\downarrow r_{2B}.$$

Now, using the hypothesis $t[\![pq_1]\!] \sim_b t[\![pq_2]\!]$ and Lemma A.5 we also have

$$\mathbb{P}(t[pq_1r_{2A}]) = \mathbb{P}(t[pq_2r_{2A}]).$$

Putting this together, we get

$$\mathbb{P}(t[pq_1r_{2A}]) = \mathbb{P}(t[pq_1r_{2A}s]).$$

Lemma A.2 then mandates $s = \varepsilon$. This concludes the case, because it implies $r_{1B} = r_{2B}$ and hence

$$t[p]\langle q_1 \rangle[r] = t[pq_1r] = \underline{|r_{1B}|_\lambda} = \underline{|r_{2B}|_\lambda} = t[pq_2r] = t[p]\langle q_2 \rangle[r].$$

Case $t[p] = \lambda\, u$ **and** $t[p] = u\, v$**:** These cases follow by straightforward application of the induction hypothesis. □

In the informal discussion of the algorithm, we have repeatedly referenced the fact that two closed subjects without a context are α-equivalent if and only the subjects are equal. This fact is a corollary of the technical lemma above.

Lemma A.8. For closed terms t_1, t_2 we have $t_1[\![\varepsilon]\!] \sim t_2[\![\varepsilon]\!]$ iff $t_1 = t_2$.

PROOF. The right-to-left implication follows directly from the fact the bisimulation relation is reflexive. For the left-to-right implication, we will use Lemma A.7 instantiated with

$$t := t_1\, t_2 \qquad p := \varepsilon \qquad q_1 := \nearrow \qquad q_2 := \searrow \qquad r := \varepsilon$$

The bisimilarity precondition is obtained with the help of Lemma A.6:

$$(t_1\, t_2)[\![\nearrow]\!] \sim_b t_1[\![\varepsilon]\!] \sim_b t_2[\![\varepsilon]\!] \sim_b (t_1\, t_2)[\![\searrow]\!].$$

Lemma A.7 then lets us conclude

$$t_1 = (t_1\, t_2)[\nearrow] = (t_1\, t_2)\langle \nearrow \rangle = (t_1\, t_2)\langle \searrow \rangle = (t_1\, t_2)[\searrow] = t_2. \qquad \Box$$

Now, we have all the basic ingredients to prove Theorem 4.1. As shown in Figure 1b in the introduction, sometimes, we need multiple different subforks to establish a fork equivalence. In the proof of Theorem 4.1, we use strong induction to decompose a bisimilar pair $t_1[\![p_1]\!] \sim_b t_2[\![p_2]\!]$ into a sequence of single forks. In the base case, the information required to form the single fork comes from Lemma A.8. In the step case, the required information is extracted using Lemma A.7.

Theorem 4.1. If $t_1[\![p_1]\!] \sim_b t_2[\![p_2]\!]$, then $t_1[\![p_1]\!] \sim_f t_2[\![p_2]\!]$.

PROOF. Proceed by strong induction on p_1. That is, we suppose that the claim is true for any strict prefix of p_1 and other arguments are changed arbitrarily. Let us split $p_1 = p'_{1,0}p_{1,1}$ so that $p_{1,1}$ is locally closed in $t[p'_{1,0}]$, and $p'_{1,0}$ is as short as possible. The proof proceeds differently whether $p'_{1,0}$ is empty or not.

Case $p'_{1,0} = \varepsilon$: We know that $t_1[p_1]$ is closed. Furthermore, from Lemma A.4 we know that $t_2[p_2]$ is also closed. Therefore, using Lemma A.6 we have

$$t_1[p_1][\![\varepsilon]\!] \sim_b t_1[\![p_1]\!] \sim_b t_2[\![p_2]\!] \sim_b t_2[p_2][\![\varepsilon]\!].$$

Using Lemma A.8 we then obtain $t_1[p_1] = t_2[p_2]$. We can then directly establish $t_1[\![p_1]\!] \sim_{sf}$ $t_2[\![p_2]\!]$ using the second rule of Definition 2.12.

Case $p'_{q,0} \neq \varepsilon$: By definition of the split $p'_{1,0}p_{1,1}$, if we move the last symbol from $p'_{1,0}$ to the beginning of $p_{1,1}$, $p_{1,1}$ stops being locally closed in $t[p'_{1,0}]$. Therefore, this symbol is $\downarrow$. Let us then denote $p'_{1,0}$ without it as $p_{1,0}$, so that $p = p_{1,0} \downarrow p_{1,1}$. Since $\downarrow p_{1,1}$ is not locally closed in $t[p_{1,0}]$, there is a $v \in \mathbb{F}(t_1[p_1])$ such that $t_1[\![p_1 v]\!] \xrightarrow{\uparrow} t_1[\![p_{1,0}]\!]$. By Lemma A.4, also $v \in \mathbb{F}(t_2[p_2])$, and there is a split $p_2 = p_{2,0} \downarrow p_{2,1}$ such that

$$t_2[\![p_2 v]\!] \xrightarrow{\uparrow} t_2[\![p_{2,0}]\!] \qquad\qquad t_1[\![p_{1,0}]\!] \sim_b t_2[\![p_{2,0}]\!].$$

We use induction assumption on $t_1[\![p_{1,0}]\!] \sim_b t_2[\![p_{2,0}]\!]$, and together with Observation 2.13 obtain

$$t_1[\![p_{1,0} \downarrow p_{2,1}]\!] \sim_f t_2[\![p_{2,0} \downarrow p_{2,1}]\!].$$

To finish the proof, we need to prove that $t_1[\![p_{1,0} \downarrow p_{1,1}]\!] \sim_f t_1[\![p_{1,0} \downarrow p_{2,1}]\!]$. We know that these two term nodes are bisimilar by

$$t_1[\![p_{1,0} \downarrow p_{1,1}]\!] \sim_b t_2[\![p_{2,0} \downarrow p_{2,1}]\!] \sim_b t_1[\![p_{1,0} \downarrow p_{2,1}]\!].$$

Lemma A.7 then gives us

$$t_1[p_{1,0}]\langle\downarrow p_{1,1}\rangle = t_1[p_{1,0}]\langle\downarrow p_{2,1}\rangle.$$

The required fork can then be established by using the first rule of Definition 2.12. A schematic overview of this case can be found in Figure 7. □

A.2 Fork Equivalence implies Algorithm

Before we start analyzing the behavior of the globalization algorithm, we first make some preliminary observations about the interaction between term indexing, closed terms, substitutions, locally closed positions, and strongly connected components.

Observation A.9. Let $p \in \mathbb{P}(t)$ be locally closed in t. If either t is closed or $t\langle p\rangle$ is closed, then $t\langle p\rangle = t[p]$.

Observation A.10. If p is locally closed in t then p is locally closed in $t\sigma$. Further, $t\sigma\langle p\rangle = t\langle p\rangle\sigma$.

Observation A.11. Let t be a closed term such that $p \in \text{SCC}(t)$ and q is locally closed in $t[p]$. If $t[p]\langle q\rangle$ is open, then $pq \in \text{SCC}(t)$.

Now we can start analyzing the behavior of the globalization algorithm. We start with a simple lemma that states that the context around a closed subject does not influence the behavior of globalize and its helper functions on that subject. This lemma will later be used to show if there is a single fork between term nodes formed through the second rule of Definition 2.12, then the closed subjects of those nodes must be equal after globalization.

Lemma A.12. If $p \in \mathbb{P}(t)$ and $t[p]$ is closed, then for all r and σ,

$$\text{globalize}(t)[p] = \text{globalize}_{scc}(r, \sigma, t)[p] = \text{globalize}_{step}(r, \sigma, t)[p] = \text{globalize}(t[p]).$$

PROOF. We proceed by induction on p. If $p = \varepsilon$ then t is closed, and hence $t\sigma = t$. The conclusion follows directly from the definitions. Otherwise, when $p = xp_0$, we can perform a single unfolding of the definitions. The conclusion then follows directly from the induction hypothesis. □

The remaining lemmas are meant to analyze the behavior of globalize for term nodes with a single fork formed through the first rule of Definition 2.12. Eventually, we will argue that a this rule gives rise to a term whose SCC contains a duplicate term. The following lemma demonstrates that this allows us to move an indexing operation that selects this duplicate term from outside globalize$_{\text{step}}$ to inside globalize. This is a similar idea to the previous lemma, but now following a different path of the algorithm, were we know that a non-trivial substitution will occur.

Lemma A.13. Let $p \in \mathbb{P}(t)$, $t\sigma[p]$ be closed, and $t[p] \in \text{duplicates}(r)$. Then

$$\text{globalize}_{\text{step}}(r, \sigma, t)[p] = \text{globalize}(t\sigma[p]).$$

PROOF. By induction on p. When $p = \varepsilon$, the equality holds trivially. Furthermore, if t is closed or $t \in \text{duplicates}(r)$, we have

$$\text{globalize}_{\text{step}}(r, \sigma, t)[p] = \text{globalize}(t\sigma)[p].$$

The problem then reduces to Lemma A.12. Otherwise, the most interesting case is $p = \downarrow p_0$. From the induction hypothesis, we get

$$\text{globalize}_{\text{step}}(r, \sigma, t)[\downarrow p_0] = \text{globalize}_{\text{step}}(r, (g(r\ t) : \sigma), t[\downarrow])[p_0] = \text{globalize}(t[\downarrow](g(r\ t) : \sigma)[p_0]).$$

The proof is then completed by realizing that

$$t[\downarrow](g(r\ t) : \sigma)[p_0] = t\sigma[\downarrow p_0]$$

This is true because $t\sigma[\downarrow p_0]$ is closed, and therefore the topmost λ of t is never referenced. □

The following lemma contains the core argument that allows us to conclude correct behavior of globalize on subjects of nodes with a single fork formed through the first rule. Note how some of the assumptions of this lemma correspond closely to the preconditions of this rule.

Lemma A.14. Let r be closed, $p \in \text{SCC}(r)$ and $r[p]\sigma$ be closed. Let $q_1, q_2 \in \mathbb{P}(r[p])$ be locally closed positions in $r[p]$ such that $r[p]\langle q_1 \rangle = r[p]\langle q_2 \rangle$. Then

$$\text{globalize}_{\text{step}}(r, \sigma, r[p])[q_1] = \text{globalize}_{\text{step}}(r, \sigma, r[p])[q_2].$$

PROOF. First, consider the case where $r[p]\langle q_1 \rangle$ is closed. From Observation A.9 we then have

$$r[p][q_1] = r[p]\langle q_1 \rangle = r[p]\langle q_2 \rangle = r[p][q_2].$$

By Lemma A.12, both sides of the desired equation now reduce to $\text{globalize}(r[pq_1])$, making it true by reflexivity.

We can now assume that neither $r[p]\langle q_1 \rangle$ nor $r[p]\langle q_2 \rangle$ is closed. From Observation A.11 we then have $pq_1, pq_2 \in \text{SCC}(r)$. Furthermore, the definition of a single fork (2.12) gives us $r[p]\llbracket q_1 \rrbracket \sim_{\text{sf}} r[p]\llbracket q_1 \rrbracket$, and hence, according to the definition of term summaries (3.7), $|r[pq_1]| = |r[pq_2]|$. These facts give us

$$r[pq_1], r[pq_2] \in \text{duplicates}(r).$$

Using Lemma A.13 we complete the lemma as follows:

$$\text{globalize}_{\text{step}}(r, \sigma, r[p])[q_1] = \text{globalize}(r[p]\sigma[q_1])$$
$$= \text{globalize}(r[p]\sigma[q_2]) = \text{globalize}_{\text{step}}(r, \sigma, r[p])[q_2]$$

The middle step in this reasoning chain is justified using Observation A.9 and A.10 by

$$r[p]\sigma[q_1] = r[p]\sigma\langle q_1 \rangle = r[p]\langle q_1 \rangle\sigma = r[p]\langle q_2 \rangle\sigma = r[p]\sigma\langle q_2 \rangle = r[p]\sigma[q_2]. \qquad \square$$

Although the hypotheses in the lemma above are similar to the preconditions of a single fork, there are some additional assumptions. This includes the existence of a substitution list σ and an assumption that $p \in \mathrm{SCC}(r)$. The conclusion is also slightly off. Eventually, we need to conclude with a statement of the form

$$\mathrm{globalize}_{\mathrm{step}}(r, [], r)[pq_1] = \mathrm{globalize}_{\mathrm{step}}(r, [], r)[pq_2],$$

where we have an empty substitution list, and the index p is outside of $\mathrm{globalize}_{\mathrm{step}}$. The following technical lemma shows that if we perform enough reduction steps of the globalization algorithm, this equation will eventually take a shape suitable for Lemma A.14.

Lemma A.15. Let r be closed term, σ a substitution list, $p \in \mathrm{SCC}(r)$ and $q \in \mathbb{P}(r[p])$. Then there exist $\sigma_1, \sigma_2, r', p'$ and q' such that

$$pq = p'q' \qquad r' = r[p']\sigma_1 \qquad r' \text{ is closed} \qquad r'[q']\sigma_2 \text{ is closed} \qquad q' \in \mathrm{SCC}(r')$$

$$\mathrm{globalize}_{\mathrm{step}}(r, \sigma, r[p])[q] = \mathrm{globalize}_{\mathrm{step}}(r', \sigma_2, r'[q']).$$

Proof. Follows trivially by induction on q.   □

Now follows the main fact that the existence of single fork between term nodes means that their subjects are equal after globalization. The final theorem then follows trivially from this.

Lemma A.16. If $t_1 [\![p_1]\!] \sim_{\mathrm{sf}} t_2 [\![p_2]\!]$, then $\mathrm{globalize}(t_1)[p_1] = \mathrm{globalize}(t_2)[p_2]$.

Proof. Recall that the single fork $t_1 [\![p_1]\!] \sim_{\mathrm{sf}} t_2 [\![p_2]\!]$ can be built using two rules. We will provide a separate proof for each rule.

$$\frac{t_1[p_1] \text{ closed} \quad t_1[p_1] = t_2[p_2]}{t_1 [\![p_1 r]\!] \sim_{\mathrm{sf}} t_2 [\![p_2 r]\!]} \text{ closed}$$

For this rule, the conclusion reduces to $\mathrm{globalize}(t_1)[p_1 r] = \mathrm{globalize}(t_2)[p_2 r]$. Note that it is sufficient to prove $\mathrm{globalize}(t_1)[p_1] = \mathrm{globalize}(t_2)[p_2]$. This follows directly from Lemma A.12 and the assumption $t_1[p_1] = t_2[p_2]$.

$$\frac{q_1 \text{ locally closed in } t[p] \quad t[p]\langle q_1 \rangle = t[p]\langle q_2 \rangle}{t [\![pq_1 r]\!] \sim_{\mathrm{sf}} t [\![pq_2 r]\!]} \text{ let-abs}$$

For this rule, the conclusion reduces to $\mathrm{globalize}(t)[pq_1] = \mathrm{globalize}(t)[pq_2]$. Because t is closed, this can be expanded into $\mathrm{globalize}_{\mathrm{step}}(t, [], t)[pq_1] = \mathrm{globalize}_{\mathrm{step}}(t, [], t)[pq_2]$. We then use Lemma A.15 to obtain $r, \sigma_1, \sigma_2, p_1$ and p_2 such that

$$p = p_1 p_2 \qquad r = t[p_1]\sigma_1 \qquad r \text{ is closed} \qquad r[p_2]\sigma_2 \text{ is closed} \qquad p_2 \in \mathrm{SCC}(r)$$

$$\mathrm{globalize}_{\mathrm{step}}(t, [], t)[p] = \mathrm{globalize}_{\mathrm{step}}(r, \sigma_2, r[p_2]).$$

Note that from Observation A.10 we have that q_1 is locally closed in $r[p_2]$ and $r[p_2]\langle q_1 \rangle = r[p_2]\langle q_2 \rangle$. Lemma A.14 then completes the proof by showing

$$\mathrm{globalize}_{\mathrm{step}}(r, \sigma_2, r[p_2])[q_1] = \mathrm{globalize}_{\mathrm{step}}(r, \sigma_2, r[p_2])[q_2]. \qquad \qquad □$$

The hardest part of proving Theorem 4.2 is already proven as Lemma A.16, now we finish it by considering arbitrary fork-equivalent pair instead of a single fork.

Theorem 4.2. If $t_1 [\![p_1]\!] \sim_{\mathrm{f}} t_2 [\![p_2]\!]$, then $\mathrm{globalize}(t_1)[p_1] = \mathrm{globalize}(t_2)[p_2]$.

Proof. A fork consists of a sequence of single forks. Each part of the sequence is proven correct by Lemma A.16. The correctness of the complete sequence follows from the transitivity of Leibniz equality.   □

A.3 Algorithm implies Bisimilarity

The main task in this section is to show that

$$r[\![\varepsilon]\!] \sim_{\mathrm{b}} \mathrm{globalize}(r)[\![\varepsilon]\!].$$

The main theorem then readily follows. To show this, we must first define the bisimulation relation on g-terms. The transitions defined in Definition 2.11 for λ-term nodes are lifted verbatim to g-term nodes. Additionally, we extend the transition system by adding appropriate outgoing edges to global variables. In particular, for any g-term node $t_1[\![p_1]\!]$ such that $t_1[p_1]$ is of the form $g(t_2\ t_2[p_2])$ and $t_2[p_2] \notin \mathrm{duplicates}(t_2)$ we have

$$t_1[\![p_1]\!] \xrightarrow{\uparrow} t_2[\![p_2]\!].$$

Remark A.17. This definition of extra $\uparrow$ edges above is made specifically to match the efficient globalization algorithm. To reason about $\mathrm{globalize}_{\mathrm{naive}}$, we would instead add a transition

$$t_1[\![p_1]\!] \xrightarrow{\uparrow} t_2[\![\varepsilon]\!]$$

for all g-term nodes such that $t_1[\![p_1]\!] = g(t_2)$.

Remark A.18. Since we added extra transitions, bimisilarity on g-terms does not match fork-equivalence on g-terms as it does for λ-terms. For example, the following two term nodes are bisimilar but not fork-equivalent.

$$(\lambda\ \underline{0})[\![\downarrow]\!] \sim_{\mathrm{b}} g((\lambda\ \underline{0})\ (\lambda\ \underline{0}))[\![\varepsilon]\!]$$

Having defined a suitable transition system to use for the bisimulation relation, we can now start working towards a proof. We start by observing a few technical facts about bisimilarity. The following two observations are variants of Lemma A.6 and Lemma A.5, trivially lifted from λ-terms to g-terms.

Observation A.19. Let t and $t[p]$ be closed g-terms, then $t[\![p]\!] \sim_{\mathrm{b}} t[p][\![\varepsilon]\!]$.

Observation A.20. If $q \in \mathbb{P}(t_1[p_1])$ and $t_1[\![p_1]\!] \sim_{\mathrm{b}} t_2[\![p_2]\!]$ then $t_1[\![p_1 q]\!] \sim_{\mathrm{b}} t_2[\![p_2 q]\!]$.

And the following fact will be useful to prove bisimilarity by induction.

Observation A.21. Bisimulation between to g-term nodes can be established if their corresponding subterms are known to be bisimilar.

$$t_1[\![p_1\ \nearrow]\!] \sim_{\mathrm{b}} t_2[\![p_2\ \nearrow]\!] \wedge t_1[\![p_1\ \searrow]\!] \sim_{\mathrm{b}} t_2[\![p_2\ \searrow]\!] \implies t_1[\![p_1]\!] \sim_{\mathrm{b}} t_2[\![p_2]\!]$$
$$t_1[\![p_1\ \downarrow]\!] \sim_{\mathrm{b}} t_2[\![p_2\ \downarrow]\!] \implies t_1[\![p_1]\!] \sim_{\mathrm{b}} t_2[\![p_2]\!]$$

In the following two technical lemmata, we prove that a term is bisimilar to itself, even if some of its de Bruijn indices have been replaced by appropriate global variables.

Lemma A.22. Let t be a closed g-term, h be a g-term, and n a g-term node such that

$$g(h) \xrightarrow{\uparrow} n, \quad n \sim_{\mathrm{b}} (\lambda\ t)[\![\varepsilon]\!].$$

Then

$$t[\underline{0} := g(h)][\![\varepsilon]\!] \sim_{\mathrm{b}} (\lambda\ t)[\![\downarrow]\!]$$

Proof. The bisimulation $n \sim_{\mathrm{b}} (\lambda\ t)[\![\varepsilon]\!]$ gives us a bisimulation relation R. From that, we construct a new relation R' as follows.

$$R' = R \cup \{(t[\underline{0} := g(h)][\![p]\!], (\lambda\ t)[\![\downarrow p]\!]) \mid p \in \mathbb{P}(t)\}$$

It is easy to check that this relation is a bisimulation, therefore R is included in bisimilarity, and $t[\underline{0} := g(h)][\![\varepsilon]\!] \sim_{\mathrm{b}} (\lambda\ t)[\![\downarrow]\!]$.   □

Lemma A.23. Let σ be a substitution into global variables, h, t be g-terms, and n be a g-term node such that

$$g(h) \xrightarrow{\uparrow} n, \quad n \sim_b (\lambda\ t)\sigma[\![\varepsilon]\!].$$

Then

$$t(g(h) : \sigma)[\![\varepsilon]\!] \sim_b (\lambda\ t)\sigma[\![\downarrow]\!].$$

PROOF. Let $t' = (\lambda\ t)\sigma[\![\downarrow]\!]$. Then $t(g(h) : \sigma) = t'[\underline{0} := g(h)]$, and we obtain the result by applying Lemma A.22 to t'. □

Now we have all the tools needed to prove the key fact – that the globalization algorithm does not change the term modulo bisimilarity. We cannot prove this directly for the globalize function. The induction hypothesis would be too weak. Instead we prove a stronger statement for globalize$_{scc}$.

Lemma A.24. let r be a closed g-term, $p \in SCC(r)$, and let σ be a list of global variables. Assume

$$r[\![p]\!] \sim_b r[p]\sigma[\![\varepsilon]\!], \tag{A.24.1}$$

and $r[p] \notin \text{duplicates}(r)$. Then we obtain

$$r[\![p]\!] \sim_b \text{globalize}_{scc}(r, \sigma, r[p])[\![\varepsilon]\!]. \tag{A.24.2}$$

PROOF. Proceed by induction on $r[p]$.

Case $r[p] = \underline{i}$: We have globalize$_{scc}(r, \sigma, r[p]) = \underline{i}\sigma = r[p]\sigma$. The conclusion the follows directly from Equation A.24.1.

Case $r[p] = g(h)$: We have globalize$_{scc}(r, \sigma, r[p]) = g(h) = r[p]\sigma$. The conclusion again follows from Equation A.24.1.

Case $r[p] = \lambda\ t$: We have

$$\text{globalize}_{scc}(r, \sigma, r[p]) = \lambda\ \text{globalize}_{step}(r, (g(r\ r[p]) : \sigma), r[p \downarrow]). \tag{A.24.3}$$

Further reduction of the algorithm depends on whether globalize$_{step}$ transitions to globalize or globalize$_{scc}$. We will consider both cases separately. However, in both cases we will require the following fact:

$$r[\![p \downarrow]\!] \sim_b r[p \downarrow](g(r\ r[p]) : \sigma)[\![\varepsilon]\!]. \tag{A.24.4}$$

This follows from Lemma A.23, instantiating node n to $r[\![p]\!]$.

Now, Equation A.24.3 can reduce further according to two possibilities:

(1) If $r[p \downarrow]$ is closed or $r[p \downarrow] \in \text{duplicates}(r)$, it reduces to

$$\lambda\ \text{globalize}(r') = \lambda\ \text{globalize}_{scc}(r', [], r'),$$

where $r' = r[p \downarrow](g(r\ r[p]) : \sigma)$. From the induction hypothesis, using $r'[\![\varepsilon]\!] \sim_b r'[\varepsilon][\,][\![\varepsilon]\!]$, we then obtain

$$r'[\![\varepsilon]\!] \sim_b \text{globalize}_{scc}(r', [], r')[\![\varepsilon]\!].$$

Combining this with with Equation A.24.4 yields

$$r[\![p \downarrow]\!] \sim_b \text{globalize}_{scc}(r', [], r')[\![\varepsilon]\!].$$

The final conclusion then follows from Observation A.21.

(2) Otherwise, if $r[p \downarrow]$ is neither closed nor duplicate, then Equation A.24.3 reduces to

$$\lambda\ \text{globalize}_{scc}(r, (g(r\ r[p]) : \sigma), r[p \downarrow]).$$

By applying Equation A.24.4 on the induction hypothesis, we obtain

$$r[\![p \downarrow]\!] \sim_b \text{globalize}_{scc}(r, (g(r\ r[p]) : \sigma), r[p \downarrow])[\![\varepsilon]\!].$$

Finally, the conclusion again follows from Observation A.21.

Case $r[p] = t\,u$**:** We have

$$\text{globalize}_{\text{scc}}(r, \sigma, r[p]) = \text{globalize}_{\text{step}}(r, \sigma, r[p \nearrow]) \,\text{globalize}_{\text{step}}(r, \sigma, r[p \searrow]).$$

Similar to the two reduction options of the previous case, the two instances of $\text{globalize}_{\text{step}}$ either reduce further to $\text{globalize}_{\text{scc}}$ or globalize. Using similar reasoning to the previous case, we can use the induction hypothesis together with

$$r[\![p \nearrow]\!] \sim_{\text{b}} r[p \nearrow]\sigma[\![\varepsilon]\!] \qquad\qquad r[\![p \searrow]\!] \sim_{\text{b}} r[p \searrow]\sigma[\![\varepsilon]\!]$$

to obtain

$$r[\![p \nearrow]\!] \sim_{\text{b}} \text{globalize}_{\text{step}}(r, \sigma, r[p \nearrow])[\![\varepsilon]\!] \qquad r[\![p \searrow]\!] \sim_{\text{b}} \text{globalize}_{\text{step}}(r, \sigma, r[p \searrow])[\![\varepsilon]\!].$$

Finally, the conclusion again follows from Observation A.21. □

Corollary A.25.

$$r[\![\varepsilon]\!] \sim_{\text{b}} \text{globalize}(r)[\![\varepsilon]\!].$$

PROOF. Follows directly from Lemma A.24. □

Proving Theorem 4.3 is now straightforward.

Theorem 4.3. If $\text{globalize}(t_1)[p_1] = \text{globalize}(t_2)[p_2]$, then $t_1[\![p_1]\!] \sim_{\text{b}} t_2[\![p_2]\!]$.

PROOF. Since $\text{globalize}(t_1)$, and $\text{globalize}(t_1)$ are closed g-terms, this result is obtained from the following equivalence chain provided by Lemma A.25, Observation A.20, and Observation A.19.

$$t_1[\![p_1]\!] \sim_{\text{b}} \text{globalize}(t_1)[\![p_1]\!] \sim_{\text{b}} \text{globalize}(t_1)[p_1][\![\varepsilon]\!]$$
$$= \text{globalize}(t_2)[p_2][\![\varepsilon]\!] \sim_{\text{b}} \text{globalize}(t_2)[\![p_2]\!] \sim_{\text{b}} t_1[\![p_1]\!]$$

□