Sharing Equality is Linear

Beniamino Accattoli[1], Andrea Condoluci[2], and Claudio Sacerdoti Coen[2]

1 **INRIA & LIX, École Polytechnique, France,** beniamino.accattoli@inria.fr
2,3 **Department of Computer Science and Engineering, University of Bologna, Italy,** claudio.sacerdoticoen@unibo.it, andrea.condoluci@unibo.it

Abstract

The λ-calculus is a handy formalism to specify the evaluation of higher-order programs. It is not very handy, however, when one interprets the specification as an execution mechanism, because terms can grow exponentially with the number of β-steps. This is why implementations of functional languages and proof assistants always rely on some form of sharing of subterms.

These frameworks however do not only evaluate λ-terms, they also have to compare them for equality. In presence of sharing, one is actually interested in equality—or more precisely α-conversion—of the underlying *unshared* λ-terms. The literature contains algorithms for such a *sharing equality*, that are polynomial in the sizes of the shared terms.

This paper improves the bounds in the literature by presenting the first *linear time* algorithm. As others before us, we are inspired by Paterson and Wegman's algorithm for first-order unification, itself based on representing terms with sharing as DAGs, and sharing equality as bisimulation of DAGs. Beyond the improved complexity, a distinguishing point of our work is a dissection of the involved concepts. In particular, we show that the algorithm computes the smallest bisimulation between the given DAGs, if any.

Digital Object Identifier 10.4230/LIPIcs.DICE.2018.

This work is currently submitted to LICS 2018.
The full version can be found on the first author's webpage.

Origin and Downfall of the Problem

For as strange as it may sound, the λ-calculus is not a good setting for evaluating and representing higher-order programs. It is an excellent specification framework, but—it is simply a matter of fact—no tool based on the λ-calculus implements it as it is.

Reasonable evaluation and sharing. Fix a dialect λ_X of the λ-calculus with a deterministic evaluation strategy $\to_X$, and note $\mathtt{nf}_X(t)$ the normal form of t with respect to $\to_X$. If the λ-calculus were a reasonable execution model then one would at least expect that mechanizing an evaluation sequence $t \to_X^n \mathtt{nf}_X(t)$ on random access machines (RAM) would have a cost polynomial in the size of t and in the number n of β-steps. In this way a program of λ_X evaluating in a polynomial number of steps can indeed be considered as having polynomial cost.

Unfortunately, this is not the case, at least not literally. The problem is called *size explosion*: there are families of terms whose size grows exponentially with the number of evaluation steps, obtained by nesting duplications one inside the other—simply writing down the result $\mathtt{nf}_X(t)$ may then require cost exponential in n.

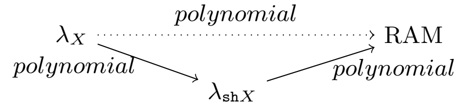
In many cases sharing is the cure because size explosion is based on unnecessary duplications of subterms, that can be avoided if such subterms are instead shared, and evaluation is modified accordingly.

The idea is to introduce an intermediate setting $\lambda_{\mathtt{sh}X}$ where λ_X is refined with sharing (we are vague about sharing on purpose) and evaluation in λ_X is simulated by some refinement

© B. Accattoli, A. Condoluci, and C. Sacerdoti Coen;
licensed under Creative Commons License CC-BY
Developments in Implicit Computational Complexity.

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

$\to_{\mathtt{sh}X}$ of $\to_X$. A term with sharing t represents the ordinary term $t\downarrow$ obtained by unfolding the sharing in t—the key point is that t can be exponentially smaller than $t\downarrow$. Evaluation in $\lambda_{\mathtt{sh}X}$ produces a shared normal form $\mathtt{nf}_{\mathtt{sh}X}(t)$ that is a compact representation of the ordinary result, that is, such that $\mathtt{nf}_{\mathtt{sh}X}(t)\downarrow = \mathtt{nf}_X(t)$. The situation can then be refined as in the following diagram:



Let us explain it. One says that λ_X is *reasonably implementable* if both the simulation of λ_X in $\lambda_{\mathtt{sh}X}$ up to sharing and the mechanization of $\lambda_{\mathtt{sh}X}$ can be done in time polynomial in the size of the initial term t and of the number n of β-steps. If λ_X is reasonably implementable then it is possible to reason about it as if it were not suffering of size explosion. The main consequence of such a schema is that the number of β-steps in λ_X then becomes a reasonable complexity measure—essentially the complexity class P defined in λ_X coincides with the one defined by RAM or Turing machines.

The first result in this area appeared only in the nineties and for a special case—Blelloch and Greiner showed that weak (that is, not under abstraction) call-by-value evaluation is reasonably implementable [5]. The strong case, where reduction is allowed everywhere, has received a positive answer only in 2014, when Accattoli and Dal Lago have shown that leftmost-outermost evaluation is reasonably implementable [4].

Reasonable conversion and sharing. Some higher-order settings need more than evaluation of a single term. They often also have to check whether two terms t and s are $\to_X$-convertible—for instance to implement the equality predicate, as in Ocaml, or for type checking in settings using dependent types, typically in Coq. These settings usually rely on a set of folklore and ad-hoc heuristics for conversion, that quickly solve many frequent special cases. In the general case, however, the only known algorithm is to first evaluate t and s to their normal forms $\mathtt{nf}_X(t)$ and $\mathtt{nf}_X(s)$ and then check $\mathtt{nf}_X(t)$ and $\mathtt{nf}_X(s)$ for equality—actually, for α-equivalence because terms in the λ-calculus are identified up to α. One can then say that conversion in λ_X is *reasonable* if checking $\mathtt{nf}_X(t) =_\alpha \mathtt{nf}_X(s)$ can be done in time polynomial in the sizes of t and s and in the number of β steps to evaluate them.

Sharing is the cure for size explosion during evaluation... but what about conversion? Size explosion forces reasonable evaluations to produce shared results. Equality in λ_X unfortunately does not trivially reduce to equality in $\lambda_{\mathtt{sh}X}$, because a single term admits many different shared representations in general. Therefore, one needs to be able to test *sharing equality*, that is to decide whether $t\downarrow =_\alpha s\downarrow$ given two shared terms t and s.

For conversion to be reasonable, sharing equality has to be testable in time polynomial in the sizes of t and s. The obvious algorithm that extracts the unfoldings $t\downarrow$ and $s\downarrow$ and then checks α-equivalence is of course too naïve, because computing the unfolding is exponential. The tricky point therefore is that sharing equality has to be checked without unfolding the sharing.

In these terms, the question has first been addressed by Accattoli and Dal Lago in [2], where they provide a quadratic algorithm for sharing equality. Consequently, conversion is reasonable.

A closer look to the costs. Once established that strong evaluation and conversion are both reasonable it is natural to wonder how efficiently can they be implemented. Accattoli and Sacerdoti Coen in [1] essentially show that strong evaluation can be implemented within

a bilinear overhead, *i.e.* with overhead linear in the size of the initial term and in the number of β-steps. Their technique has then been simplified by Accattoli and Guerrieri in [3]. Both works actually address *open* evaluation, which is a bit simpler than strong evaluation—the moral however is that evaluation is bilinear. Consequently, the size of the computed result is bilinear.

The bottleneck for conversion then seemed to be Accattoli and Dal Lago's quadratic algorithm for sharing equality. The literature actually contains also other algorithms, studied with different motivations or for slightly different problems (discussed below). None of these algorithms however matches the complexity of evaluation.

In this work we provide the first algorithm for sharing equality that is linear in the size of the shared terms, improving over the literature. Therefore, the complexity of sharing equality matches the one of evaluation, providing a combined bilinear algorithm for conversion, that is the real motivation behind this work.

Computing Sharing Equality

Sharing as DAGs. Sharing can be added to λ-terms in different forms. In this work we adopt a graphical approach. Roughly, a λ-term can be seen as a (sort of) directed tree whose root is the topmost constructor and whose leaves are the (free) variables. A λ-term with sharing is more generally a DAG. Sharing of a subterm t is then the fact that the root node r of t is the child of more than one node.

This is essentially the same sharing of calculi with explicit substitution, environment-based abstract machines, or linear logic—the details are different but all these approaches provide different incarnations of the same notion of sharing. It is instead different of so called *sharing graphs* [12] that are graphs implementing Lévy's optimal evaluation and providing a deeper form of sharing than our DAGs. To our knowledge, sharing equality for sharing graphs has never been studied—it is not even known whether it is reasonable.

Sharing equality as bisimilarity. When λ-terms with sharing are represented as DAGs, a natural way of checking sharing equality is to test DAGs for bisimilarity. Careful here: the transition system under study is the one given by the directed edges of the DAG, and not the one given by β-reduction steps, as in applicative bisimilarity—our DAGs may have β-redexes but we do not reduce them here, that is an orthogonal issue (namely, evaluation). Essentially, two DAGs represent the same unfolded λ-term if they have the same structural paths, just arranged differently.

To be precise, sharing equality is based on what we call *sharing equivalences*, that are bisimulations plus some additional requirements about names—for α-equivalence—and the requirement that they are equivalence relations.

Binders, cycles, and domination. A key point of our problem is the presence of binders, *i.e.* abstractions, and the fact that equality on λ-terms is α-equivalence. Graphically, it is standard to see abstractions as getting a backward edge from the variable they bound—this approach is also supported by the strong relationship between λ-calculus and linear logic proof nets.

Therefore, binders introduce a form of cycle in DAGs. Technically speaking these are only *half-cycles*: the cycle can be easily avoided by reversing the backward edge (and we shall do so), but its essence does not disappear: while two free variables are bisimilar only if they coincide, two bound variables are bisimilar only when also their binders are bisimilar, suggesting that λ-terms with sharing are, as directed graphs, structurally closer to deterministic finite automata (DFA), that may have cycles, than to DAGs. The problem

with cycles is that in general bisimilarity is not linear—Hopcroft and Karp's algorithm [11], the best one, is only pseudo-linear, that is, with an inverse Ackermann factor.

At the same time, these half-cycles induced by binders are of a very special form, being a graphical representation of *scopes*. They are indeed characterized by a structural property called *domination*—exploring the DAG from the root one necessarily visits the binder before the bound variable. Domination turns out to be the key ingredient for a linear algorithm in presence of binders.

Related problems. There are various problems that are closely related to sharing equality, and that are also treated with bisimilarity-based algorithms. Let us list similarities and differences:

- *First-order unification.* On the one hand the problem is more general, because unification roughly allows to substitute variables with terms not present in the original DAGs, while in sharing equality this is not possible. On the other hand, the problem is less general, because it does not allow binders and does not test α-equivalence. There are basically two linear algorithm for first-order unification, Paterson and Wegman's (shortened PW) [17] and Martelli and Montanari's (MM) [15]. Both rely on sharing to be linear. PW even takes terms with sharing as inputs, while MM deals with sharing in a less direct way, except in its less known variant [14] that takes in input terms shared using the Boyer-Moore technique [6].
- *Nominal unification.* This is unification up to α-equivalence (but not up to β or η equivalence) of λ-calculi extended with name swapping, in the nominal tradition. It has been studied by two groups, Calvès & Fernández and Levy & Villaret, adapting PW and MM form first-order unification. It is very close to sharing equality, but the known best algorithms [8, 13] are only quadratic. See [7] for a unifying presentation.
- *Pattern unification.* Miller's pattern unification [16] can also be stripped down to test sharing equality. Qian presents a PW-inspired algorithm, claiming linear complexity [18], that seems to work only on unshared terms. We say *claiming* because the algorithm is very involved and the proofs are far from being clear. Moreover, according to Levy and Villaret in [13]: *it is really difficult to obtain a practical algorithm from the proof described in [18].* We believe that is fair to say that Qian's work is hermetic (please try to read it!).
- *Nominal Matching.* Calvès & Fernández in [9] present an algorithm for nominal matching (a special case of unification) that is linear, but *only* on unshared input terms.
- *Equivalence of DFA.* Automata do not have binders, and yet they are structurally more general than λ-terms with sharing, since they allow arbitrary directed cycles, not necessarily dominated. As already pointed out, the best equivalence algorithm is only pseudo-linear [11].

Previous work. For what concerns sharing equality itself, in the literature there are only two algorithms explicitly addressing it. First, the already cited quadratic one by Accattoli and Dal Lago. Second, a $O(n \log n)$ algorithm by Grabmayer and Rochel [10] (where n is the sum of the sizes of the shared terms to compare, and the input of the algorithm is a graph), obtained by a reduction to equivalence of DFAs and treating the more general case of λ-terms with `letrec`.

Contributions: a theory and a 2-levels linear algorithm. This work is divided in two parts. The first part develops a re-usable, self-contained, and clean theory of sharing equality, independent of the algorithm that computes it. Some of its concepts are implicitly used by other authors, but never emerged from the collective unconscious before (*propagated queries* in particular)—others instead are new. The theory culminates with the sharing

equality theorem that connects α-equivalence on terms with sharing equivalences for DAG-based sharing of λ-terms, under suitable conditions.

The second part studies a linear algorithm for sharing equality by adapting PW linear algorithm for first-order unification to λ-terms with sharing. Our algorithm is actually composed by a 2-levels, modular approach, pushing further the modularity suggested—but not implemented—by Calvès & Fernández in [8]:

- *Blind sharing check*: a reformulation of PW from which we removed the management of meta-variables for unification. It is used as a first-order test on λ-terms with sharing, to check that the unfolded terms have the same skeleton, ignoring variable names.
- *Name check*: a straightforward algorithm executed after the previous one, testing α-equivalence by checking that bisimilar bound variables have bisimilar binders and that two different free variables are never shared.

The decomposition plus the correctness and the completeness of the checks crucially rely on the theory developed in the first part.

The value of the work. It is delicate to explain the value of our work. Three features are obvious: 1) the improved complexity of the problem, 2) the consequent downfall on the complexity of β-conversion, and 3) the isolation of a theory of sharing equality. At the same time, however, our algorithm looks as an easy adaptation of PW, and binders do not seem to play much of a role. Let us then draw attention to the following points:

- *Identification of the problem*: the literature presents similar studied and techniques, and yet we are the first to formulate and study the problem *per se* (unification is different, and it is usually not formulated on terms with sharing), directly (*i.e.* without reducing it to DFAs, like in Grabmayer and Rochel), and with a fine-grained look at the complexity (Accattoli and Dal Lago only tried not to be exponential).
- *The role of binders*: the fact that binders can be treated straightforwardly is—we believe—an insight and not a weakness of our work. Essentially, domination allows to reduce sharing equality in presence of binders to the blind sharing check, under mild but key assumptions on the context in which terms are tested.
- *Minimality.* The set of shared representations of an ordinary λ-term t is a lattice: the bottom element is t itself, the top element is the (always existing) maximally sharing of t, and for any two terms with sharing there exist *inf* and *sup*. Essentially, Accattoli & Dal Lago and Grabmayer & Rochel address sharing equality by computing the top elements of the lattices of the two λ-terms with sharing, and then comparing them for α-equivalence. We show that our blind sharing check—and morally every PW-based algorithm—computes the *sup* of t and s, that is, the term having all and only the sharing in t or s, that is the smallest sharing equivalence between the two DAGs. This insight, first pointed out in PW's original paper to caracterize most general unifiers, is a prominent concept in our theory of sharing equality as well.
- *Proofs, invariants, and detailed development.* We provide detailed correctness, completeness, and linearity proofs, based on finely tuned invariants of the algorithm, to a level of preciseness that is unmatched in the literature. We also provide detailed treatment of the relationship between α-equivalence on terms and sharing equivalences on DAGs. Our work is therefore self-contained, but for the fact that most details are in the Appendix.
- *Concrete implementation.* We implemented our algorithm and verified its linear complexity. The code is available on the third author's webpage.

Acknowledgements. This work has been partially funded by the ANR JCJC grant COCA HOLA (ANR-16-CE40-004-01).

References

1. B. Accattoli and C. S. Coen. On the relative usefulness of fireballs. In *LICS 2015*, pages 141–155, 2015.
2. B. Accattoli and U. Dal Lago. On the invariance of the unitary cost model for head reduction. In *RTA*, pages 22–37, 2012.
3. B. Accattoli and G. Guerrieri. Implementing open call-by-value. In *FSEN 2017, Tehran, Iran, April 26-28, 2017, Revised Selected Papers*, pages 1–19, 2017.
4. B. Accattoli and U. D. Lago. Beta reduction is invariant, indeed. In *CSL-LICS '14*, pages 8:1–8:10, 2014.
5. G. E. Blelloch and J. Greiner. Parallelism in sequential functional languages. In *FPCA*, pages 226–237, 1995.
6. R. S. Boyer and J. S. Moore. The sharing of structure in theorem-proving programs. *Machine intelligence*, 7:101–116, 1972.
7. C. Calvès. Unifying Nominal Unification. In *RTA 2013*, volume 21, pages 143–157, 2013.
8. C. Calvès and M. Fernández. The first-order nominal link. LOPSTR'10, pages 234–248, 2011.
9. C. Calvès and M. Fernández. Matching and alpha-equivalence check for nominal terms. *Journal of Computer and System Sciences*, 76(5):283 – 301, 2010.
10. C. Grabmayer and J. Rochel. Maximal sharing in the lambda calculus with letrec. In *ICFP 2014*, pages 67–80, 2014.
11. J. Hopcroft and R. Karp. A linear algorithm for testing equivalence of finite automata. Technical Report 0, Dept. of Computer Science, Cornell U, December 1971.
12. J. Lamping. An algorithm for optimal lambda calculus reduction. In *POPL*, pages 16–30, 1990.
13. J. Levy and M. Villaret. An efficient nominal unification algorithm. In *RTA 2010*, pages 209–226, Edinburgh, Scottland, UK, 2010.
14. A. Martelli and U. Montanari. Theorem proving with structure sharing and efficient unification. IJCAI'77, pages 543–543, 1977.
15. A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, Apr. 1982.
16. D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. Log. Comput.*, 1(4):497–536, 1991.
17. M. Paterson and M. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16(2):158 – 167, 1978.
18. Z. Qian. Linear unification of higher-order patterns. In *TAPSOFT'93: Theory and Practice of Software Development*, pages 391–405, 1993.