Towards Atomic Graphs

David R. Sherratt[1] and Marco Solieri[1]

Department of Computer Science, University of Bath, U.K.
d.r.sherratt@bath.ac.uk, m.solieri@bath.ac.uk

1 Atomic lambda calculus

Sharing in the λ-calculus is the use of a single representation for multiple instances of some portion of a term. The purpose of sharing is to obtain more control over the duplication of terms, which is a costly operation and may cause redundant computations. By sharing a term, we can virtually evaluate all the copies of the subterm simultaneously by evaluating the shared representation. Among the many notions of sharing, *full laziness* is a powerful and notable example of sharing. Given a term t which needs to be duplicated, it allows to share all maximal sub-terms $u_1, \dots, u_k$ of t that does not contain occurrences of a variable bound in t outside u_i [9].

The atomic λ-calculus [5] is an extension of the ordinary λ-calculus that include sharing constructs in an explicit-substitution-like syntax. Duplication, i.e. the extraction of an object from a shared portion to its context, is implemented by those constructs atomically (indeed). For instance, we can keep a subterm t shared in u using the occurrences of x as a binder, writing $u[x_1, \dots, x_n \leftarrow \lambda x.t]$. Also, we can copy an abstraction without its body, thanks to the distributor construct (cf. (1)), or duplicate an application without its function nor its argument (cf. (2)). When sharing and distributor meet, their duplication work is finished and they can be removed, recovering the broken bindings of the abstractions (cf. (3)).

$$u[x_1, \dots, x_n \leftarrow \lambda x.t] \rightsquigarrow u[x_1, \dots, x_n \twoheadleftarrow \lambda x.\langle y_1, \dots y_n\rangle[y_1, \dots y_n \leftarrow t]] \tag{1}$$

$$u[x_1, \dots, x_n \leftarrow (vt)] \rightsquigarrow u\{(y_1 z_1)/x_1, \dots, (y_n z_n)/x_n\}[y_1, \dots y_n \leftarrow v][z_1, \dots z_n \leftarrow t] \tag{2}$$

$$u[x_1, \dots, x_n \twoheadleftarrow \lambda x.\langle t_1, \dots t_n\rangle[\overline{z} \leftarrow t]] \rightsquigarrow u\{\lambda y_1.t_i[\overline{z_1} \leftarrow y_1]/x_1, \dots, \lambda y_n.t_n[\overline{z_n} \leftarrow y_n]/x_n\} \tag{3}$$

where $\overline{z_i}$ is the subset of $\overline{z}$ containing all the variables free in t_i.

This calculus is a Curry-Howard interpretation of a deep inference [4] proof system for intuitionistic logic. In particular, the distributor is the computational interpretation of the distribution rule in this system.

$$\frac{A \to (B \wedge C)}{(A \to B) \wedge (A \to C)}\ d$$

It is readily seen that the atomic λ-calculus Λ_a allows a more powerful form of sharing than fully-laziness, although fixing an appropriate strategy it can be limited to this latter.

The distributor rule features a non-local behaviour, since it requires the inspection of an arbitrarily large subterm. Namely, and referring to (3), the set $\overline{z}$ has to partitioned by inspecting some of $t_1, \dots, t_n$. *Can Λ_a be defined in purely local way?*

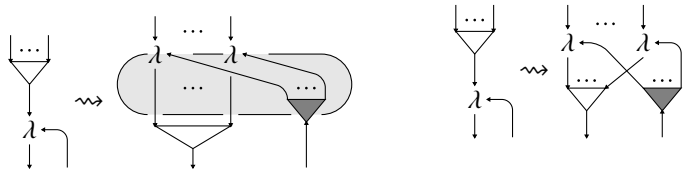
2 Sharing graphs and atomic lambda calculus

An informal graphical intuition of the atomic λ-calculus was already provided by its authors [5]. This highlighted that the sharing technique of the atomic λ-calculus is similar to the *Lévy optimal* notion

of sharing [8], implemented by sharing graphs [6, 1]. Both systems aim at the most fine-grained deconstruction of the duplication, by refining it into smaller operations for the purpose of gaining more control, and thus more laziness. In spite of this, there are major differences. Let us explore them graphically, assuming to work on graphs that include nodes of the lambda calculus (λ and @), and sharing nodes (depicted as a triangles), whose ports may be connected by wires. We shall call *atomic graphs* the graphical counterpart of Λ_a.

Both sharing graphs and atomic graphs feature two kinds of sharing nodes: a positive one, here depicted in white, whose principal port points down towards variables occurrences; and a negative one, called *co-sharing* and coloured in black, pointing towards the graph's root.

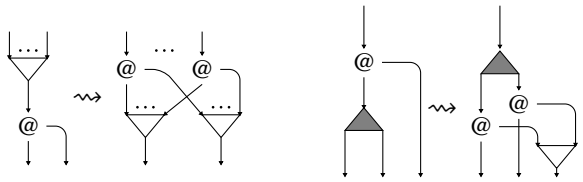
Co-sharing In atomic graphs, the co-sharing node is part of the wider notion of distributor, introduced by the duplication of an abstraction node. The duplicated abstractions are frozen, and the only role played by the co-sharing is waiting for a *rendez-vous* with a sharing node. In Fig. 1a, the distributor is depicted as the gray rounded box. In sharing graphs, the co-sharing has no inferiority complex with its dual: it has no additional restriction (cf. Fig. 1b) and it can act essentially the same way as a sharing node, as the next paragraph shall clarify.



(a) Atomic graphs (b) Sharing graphs

Figure 1: Duplication of abstraction

Duplication of application It can be performed when the principal port of a sharing node is connected to an application node. In sharing graphs, this is allowed only when the sharing node faces the function port of the application (cf. Fig. 2b), whilst in atomic graphs only for the root port of the application, (cf. Fig. 2a). To be precise, the latter redex is consistent also in sharing graphs, but it is not optimal in the sense of [8], since it may duplicate a β-redex.

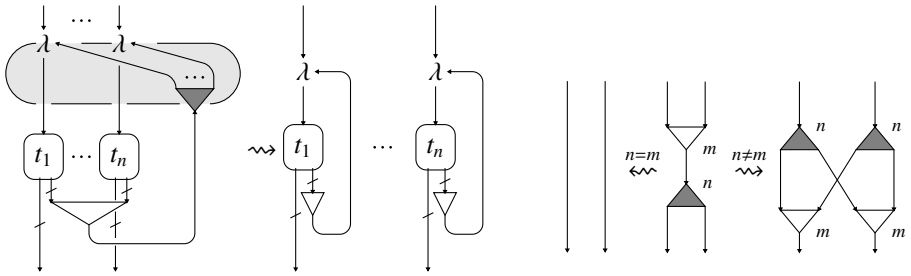


(a) Atomic graphs (b) Sharing graphs

Figure 2: Duplication of application

Sharing vs co-sharing Eventually, a sharing node will meet a co-sharing. In atomic graphs, the distributor that contains the latter node marks the copies of some lambda abstraction for which the duplication process on the body can be considered complete. The two dual nodes rewrite as a set of

sharing nodes with smaller arity, one for each duplicated, as illustrated in Fig. 3a. As we mentioned before about the corresponding term rule (3), and as we shall clarify also later, this rule is a delicate point if one wants local definition.
However, in sharing graphs, a sharing and a co-sharing nodes may also not be friends, i.e. they may are duplicating unrelated, but overlapping sub-graphs. Hence, when they may not only annihilate one with the other (when their common job is done), by also duplicate each other, as in Fig. 3b. To handle this decision, sharing graphs need to name (co-)sharing nodes, and to perform an additional bookkeeping work to maintain coherence on such naming, by dynamically adjusting names during reduction — the so-called oracle. Now, the number of bookkeeping operations can grow as large as an elementary function (a fixed-height tower of exponentials) in the number of β-steps [7]. Therefore, the oracle might introduce overhead causing an overall inefficient reduction. This is still an open question, though, since the number of β-steps, in turn, is considerably shrunk by the optimal sharing. The local duplication itself, instead, has been very recently proven to not cause any overhead to sharing graphs reduction [3]. Since the atomic λ-calculus does not need any bookkeeping, and not even names for sharing, it is interesting to see if this can be preserved in atomic graphs.



(a) Atomic graphs (b) Sharing graphs

Figure 3: Sharing vs. co-sharing

3 Atomic graphs

Aim Let us summarise the question of interests for the ongoing investigation we are reporting about. We are looking for a local formulation of the atomic λ-calculus as a graph rewriting system called atomic graphs ($\mathfrak{Q}_a$), so that, at the same time, we can also discover at full depth its relationship with sharing graphs. The minimum, natural requirement of such a definition is that it is accompanied with a soundness and completeness result. Namely, provided a translation $[\![\cdot]\!]$ of atomic terms to atomic graphs, and a read-back $(\!|\cdot|\!)$ of graphs in terms, we expect to obtain two simulations.

Conjecture 1. *Given $t \in \Lambda_a$:*

1. *if $[\![t]\!] \rightsquigarrow G$ for some $G \in \mathfrak{Q}_a$, then $t \rightsquigarrow^* (\!|G|\!)$;*
2. *if $t \rightsquigarrow t'$ for some $t \in \Lambda_a$, then $[\![t]\!] \rightsquigarrow^* [\![t']\!]$.*

Technique The key issue is to formulate locally the already mentioned sharing/co-sharing rule. If the arities of two nodes of this kind respectively are m, n (cf. Fig. 4) then we know that they rewrite in n sharing nodes. Quite less clear is how should the original sharing node be split. What are the arities of reduct sharing nodes? To which sub-graph should each of the resulting sharing node be connected?

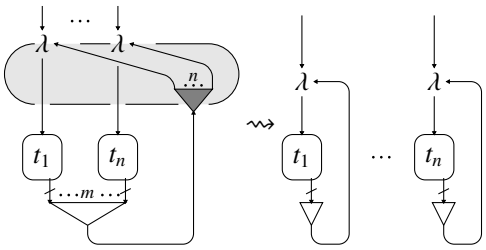
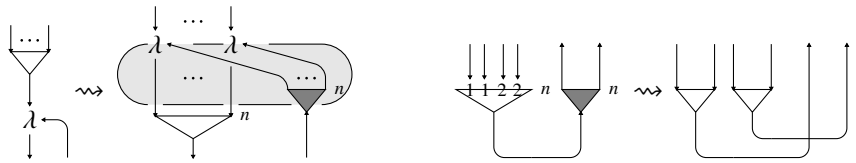


Figure 4: Sharing vs. co-sharing in atomic graphs

Let us briefly outline two techniques currently under our consideration, while mentioning the most critical technical points.

Recording sharing information equip sharing and co-sharing nodes with labels and enrich reduction rules to manage labeling. Labels are first introduced when duplicating a λ as shown in Figure 5a, then they are properly manipulated as the sharing node propogates through the graph. The labels record a pairing index for the sharing and co-sharing, and the ports of the sharing node are individually labelled. Thus finally, labels allow deciding – locally and syntactically – the shape of the sharing/co-sharing reduction rule, as illustrated in Figure 5b.



(a) Labels introduced (b) Labels Eliminated

Figure 5: Recording sharing information

Sharing nodes synchronisation assume that any atomic λ-term we consider is a reduct of the translation of an ordinary λ-term; then observe a uniformity property on sharing nodes. Namely, in the sharing/co-sharing rule it must be the case that arities of reduct sharing nodes are all equal to $\frac{n}{m}$.

Perspectives The construction of atomic graphs is not only prompted by the need of a truly local, graphical formulation of the atomic calculus. It also allows determining the degree of expressive power that is actually needed to do so. More generally, this paves the way (already partially traveled by Balabonski [2]) towards qualitative and quantitative classifications of the many notions of laziness and sharing in the implementation of the λ-calculus.

References

[1] Andrea Asperti and Stefano Guerrini. *The Optimal Implementation of Functional Programming Languages*, volume 45 of *Cambridge tracts in theoretical computer science*. Cambridge University Press, 1998.

[2] Thibaut Balabonski. A unified approach to fully lazy sharing. In *Proceedings of the 39th annual ACM SIGPLAN SIGACT symposium on Principles of programming languages*, POPL '12, pages 469–480, Philadelphia, PA, USA, 2012. ACM.

[3] Stefano Guerrini and Marco Solieri. Is the optimal implementation inefficient? elementary not. To appear in *Second International Conference on Formal Structures for Computation and Deduction* (FSCD17). Preliminary version available on the author's homepage, 2017.

[4] Alessio Guglielmi, Tom Gundersen, and Michel Parigot. A proof calculus which reduces syntactic bureaucracy. 2010.

[5] Tom Gundersen, Willem Heijltjes, and Michel Parigot. Atomic lambda calculus: A typed lambda-calculus with explicit sharing. In *Logic in Computer Science (LICS), 2013 28th Annual IEEE/ACM Symposium on*, pages 311–320. IEEE, 2013.

[6] John Lamping. An algorithm for optimal lambda calculus reduction. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 16–30. ACM, 1989.

[7] Julia L Lawall and Harry G Mairson. Optimality and inefficiency: what isn't a cost model of the lambda calculus? *ACM SIGPLAN Notices*, 31(6):92–101, 1996.

[8] J-J Lévy. Optimal reductions in the lambda calculus. *To HB Curry: Essays on Combinatory Logic, Lambda Coalculus and Formalism*, pages 159–191, 1980.

[9] Christopher Peter Wadsworth. *Semantics and Pragmatics of the Lambda-Calculus.* PhD thesis, University of Oxford, 1971.