Confluence of a Computational Lambda Calculus for Higher-Order Relational Queries

Claudio Sacerdoti Coen and Riccardo Treglia

Università di Bologna, Bologna, Italy
claudio.sacerdoticoen@unibo.it
riccardo.treglia@unibo.it

Abstract

We study the operational semantics of an untyped computational lambda-calculus whose normal forms represent queries on databases. The calculus extends the computational core with additional operations and rewriting rules whose effect is to turn the monadic type of computations into a multiset monad that capture tables. Moreover, we introduce comonadic constructs and additional rewriting rules to be able to form tables of tables. Proving confluence becomes tricky: we succeed exploiting decreasing diagrams.

1 Introduction to the Calculus: Syntax and Reduction Relation

The second author et al. have introduced and studied in [dLT20, FGdLT22] the computational core $\lambda_{\copyright}$, a λ-calculus inspired by Moggi's computational one [Mog89, Mog91]. The calculus differentiates between values and computations, the latter obtained via return/bind constructs for a generic monad. The strong operational semantics is obtained simply orienting the monadic laws, and confluence was proved among other properties.

In this work, we extend $\lambda_{\copyright}$ with specific additional operations and rewriting rules over computations that turn the generic monad into a multiset monad: the 0-ary operation $\emptyset$ represents the empty multiset, $\uplus$ the union of multisets, and the monadic return, denoted by $[V]$, is now interpreted as forming a singleton. The rewriting rules partially capture the algebraicity of the operations in the sense of Plotking and Power [PP02, PP03] by letting the operators commute with those rewriting contexts that are built from bind operators, only. Because in $\lambda_{\copyright}$, contrary to Moggi's computational λ-calculus, values and computations are rigidly split, the extension obtained so far does not allow formation of multisets of multisets, because multisets are not values. To overcome the issue, we add two more co-monadic constructs to reflect computations into values, following ideas by [Fil94]. These constructs are the thunk/force constructs of Levy's call-by-push-value [Lev99]; however, our calculus is strong, i.e. it allows reduction inside values as well. Finally, we introduce an equational theory over computations to capture associativity and commutativity of $\uplus$ and idempotency of $\emptyset$: this is the minimal theory that turns the calculus into a confluent one.

The exact choice of rewriting and equational rules that we pick seems rather arbitrary at first: the empty set is not the neutral element of $\uplus$ and the monadic operations are not forced to be completely algebraic (e.g. $\uplus$ does not commute with contexts that include thunks or force). It is to the (untyped) NRCλ calculus [RC20] as $\lambda_{\copyright}$ is to the (untyped) λ-calculus, and indeed we are introducing it with the intent of studying semantic properties of the NRCλ-calculus via intersection types, trying to scale what the second author already did for $\lambda_{\copyright}$. The NRCλ calculus is an example of nested, higher-order relational calculus that provides a principled foundation for integrating database queries into programming languages. In NRCλ, a database

table is represented by the multiset of its rows, where each row is just a value (NRCλ has tuples). The main properties of the calculus are that it is confluent and strongly normalizing and, moreover, some normal forms can be directly interpreted as SQL queries (those such that the types of the free variables and of the result are just tables of base types and not tables of tables). In particular, the set of rewriting and equational rules that our calculus inherits from the NRCλ-calculus is the minimal set that grants the previous properties.

Because of the important application to database, from now on we call our extension of the $\lambda_{\copyright}$ calculus the λ_{SQL} calculus.

Contributions The first contribution of the work is the design of the λ_{SQL} calculus, which goes beyond the mere effort to fit the NRCλ into a well-assessed monadic frame. Indeed, this can be considered as an experiment of extending $\lambda_{\copyright}$ with algebraic operators (other cases are [dT21, AKR23]), but here it immediately highlights, for example, the need to introduce other kind of constructs, such as the comonadic unit, that could be added to $\lambda_{\copyright}$ independently of the algebraic operators.

The second contribution is the proof of a fundamental property of the calculus: confluence. The proof is labour-intensive because the rewriting rules associated to algebraicity of the operators turn them into control operators: each operator can capture its context and then erase or duplicate it, and many critical pairs arise. Moreover, there is also the issue of the interplay between the equational theory and the rewriting theory. Technically, we make strong use of von Oostrom's decreasing diagram technique [vO94], the most difficult point of which is to find the order relation between the labels of the calculus reduction rule. This will be done by considering orthogonal and nested closures of certain reduction rules, inspired by the work in [ADJL17], postponing in a final step the commutation with respect to the union operator.

Long-term perspectives Our long-term goal is to extract from the confuence proof based on decreasing diagrams an order over reduction rules to design a well-behaved normalizing strategy. We will then define an appropriate intersection type system based on tight multi-types [AGK20] to capture quantitatively the set of terminating queries according to that strategy, the length of their reduction and the size of the normal forms, i.e. the size of the computed SQL queries. Ultimately we want to capture even more quantitative information over the queries itself.

Syntax and Reduction The syntax of the untyped computational SQL λ-calculus, shortly λ_{SQL}, and its reduction relation are reported below:

Definition 1.1 (Term syntax)**.**

$$\begin{array}{rrcl} \mathit{Val}: & V, W & ::= & x \mid \lambda x.M \mid \langle\!\langle M \rangle\!\rangle \\ \mathit{Com}: & M, N & ::= & [V] \mid M \star V \mid M \uplus M \mid \emptyset \mid !V \end{array}$$

Like in $\lambda_{\copyright}$, terms are of either sorts *Val* and *Com*, representing values and computations, respectively. Variables x, abstractions $\lambda x.M$ — where x is bound in M — and the constructors $[V]$ and $M \star V$, written `return` V and M `>>=` V in Haskell-like syntax, respectively, form the syntax of $\lambda_{\copyright}$, which is agnostic on the interpretation of computations. In λ_{SQL}, instead, computations are meant to be understood as tables, i.e. multisets of values, and therefore $[V]$ is interpreted as the singleton whose only element is V and $\star$ as the bind operator of the list monad. The binary and 0-ary operators $\uplus$ and $\emptyset$ are additionally used to construct tables. The pair of constructs $\langle\!\langle \cdot \rangle\!\rangle$ and ! are used to reflect computations into labels, allowing to form tables of (reflected) tables. Note that $\langle\!\langle \cdot \rangle\!\rangle$ can be understood as the unit of a comonad. Terms are identified up to renaming of bound variables so that the capture avoiding substitution $M\{V/x\}$ is always well defined; $FV(M)$ denotes the set of free variables in M. Finally, like in $\lambda_{\copyright}$, application among computations can be encoded by $MN \equiv M \star (\lambda z.\ N \star z)$, where z is fresh.

Wrapping up, the syntax can be condensed in the motto:

$$\lambda_{\text{SQL}} \approx \lambda_{\copyright} + \text{operations over tables} + \text{monadic reification/reflection}$$

with the latter extension being orthogonal to the second one.

We are now in place to introduce the λ_{SQL} **reduction relation**, later closed under contexts:

Definition 1.2 (Reduction)**.** *The reduction relation is the union of the following binary relations over Com:*

$$\begin{array}{rrcll}
\beta_c) & [V] \star \lambda x.M & \mapsto_{\beta_c} & M\{V/x\} & \\
\sigma) & (L \star \lambda x.M) \star \lambda y.N & \mapsto_{\sigma} & L \star \lambda x.(M \star \lambda y.N) & \textit{for } x \notin \mathsf{fv}(N) \\
\uplus_1) & (M \uplus N) \star \lambda x.P & \mapsto_{\uplus_1} & (M \star \lambda x.P) \uplus (N \star \lambda x.P) & \\
\uplus_2) & M \star \lambda x.(N \uplus P) & \mapsto_{\uplus_2} & (M \star \lambda x.N) \uplus (M \star \lambda x.P) & \\
\emptyset_1) & \emptyset \star \lambda x.M & \mapsto_{\emptyset_1} & \emptyset & \\
\emptyset_2) & M \star \lambda x.\emptyset & \mapsto_{\emptyset_2} & \emptyset & \\
!) & !\langle\!\langle M \rangle\!\rangle & \mapsto_{!} & M &
\end{array}$$

The first two rules, taken from $\lambda_{\copyright}$, are oriented monadic equations. The next four rules capture algebraicity of the $\uplus$ operator, but only w.r.t. contexts made of $\star$ only (e.g. there is no rule $(M \uplus N) \uplus P \mapsto (M \uplus P) \uplus (N \uplus P)$ because that would be unsound for tables). The latter rule is the usual rule for the thunk/force redex in call-by-push-value.

The *reduction* $\rightarrow_{\lambda_{\text{SQL}}}$ (when it is clear from the context we omit the subscript) is the contextual closure of λ_{SQL} under *computational contexts*, where such contexts are mutually defined with values contexts as follows:

$$\begin{array}{lr}
\mathsf{V} ::= \langle\cdot_{Val}\rangle \mid \lambda x.\mathsf{C} \mid \langle\!\langle \mathsf{C} \rangle\!\rangle & \text{Value Contexts} \\
\mathsf{C} ::= \langle\cdot_{Com}\rangle \mid [\mathsf{V}] \mid \mathsf{C} \star V \mid M \star \mathsf{V} \mid \mathsf{C} \uplus M \mid M \uplus \mathsf{C} \mid !\mathsf{V} & \text{Computation Contexts}
\end{array}$$

Notice that the hole of each kind of context has to be filled in with a proper kind of term.

We equip the calculus with a sound, but not complete, equational theory for multisets, taken from [RC20].

Definition 1.3 (Equational theory E)**.**

$$\begin{array}{rrcl}
Comm) & M \uplus N & = & N \uplus M \\
Empty) & \emptyset \uplus \emptyset & = & \emptyset
\end{array}$$

2 Route to Confluence

We modularize the proof of confluence by first showing that the equational part can be postponed.

Getting rid of the equational theory. A classic tool to modularize a proof of confluence is Hindley-Rosen lemma, stating that the union of confluent reductions is itself confluent if they all commute with each other. Let us first define what commutation between a reduction relation and an equational theory means, and then state that result properly.

Definition 2.1. *Given a reduction relation $\rightarrow$ and an equational theory $=_E$, we say that $\rightarrow$ commutes over $=_E$ if for all M, N, L such that $M =_E N \rightarrow L$, there exists P such that $M \rightarrow P =_E L$.*

Lemma 2.2 (Hindley-Rosen)**.** *Let $\mathcal{R}_1$ and $\mathcal{R}_2$ be relations on the set A. If $\mathcal{R}_1$ and $\mathcal{R}_2$ are confluent and commute with each other, then $\mathcal{R}_1 \cup \mathcal{R}_2$ is confluent.*

We will exploit that to focus just on the reduction relation while proving confluence.

Lemma 2.3. *$=_E$ commutes with $\to$.*

Hence, by Lemma 2.3 one needs just the confluence of $\to$ to assert the confluence of $\to$ modulo E.

Decreasing diagram. Now that is possible to omit the equational theory induced by Definition 1.3, we need to prove the commutation of all the reduction rules, and in this intent we use decreasing diagrams by van Oostrom [vO94, vO08]. This is a powerful and general tool to establish commutation properties, which reduces the problem of showing commutation to a local test; in exchange of localization, the diagrams need to be decreasing with respect to some labelling.

Definition 2.4 (Decreasing, [vO94])**.** *An rewriting relation $\mathcal{R}$ is* locally decreasing *if there exist a presentation $(R, \{\to_i\}_{i\in I})$ of $\mathcal{R}$ and a well-founded strict order $>$ on I such that:*

$$\underset{i}{\leftarrow} \cdot \underset{j}{\rightarrow} \subseteq \overset{*}{\underset{\vee i}{\longleftrightarrow}} \cdot \overset{=}{\underset{j}{\longrightarrow}} \cdot \overset{*}{\underset{\vee\{ij\}}{\longleftrightarrow}} \cdot \overset{=}{\underset{i}{\longleftarrow}} \cdot \overset{*}{\underset{\vee j}{\longleftrightarrow}},$$

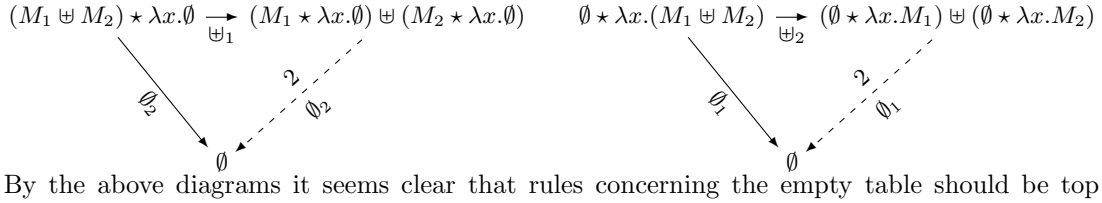
where $\vee\bar{I} = \{i \in I \mid \exists k \in \bar{I}.\ k > i\}$, $\vee i$ abbreviates $\vee\{i\}$, and $\overset{*}{\to}$ (resp. $\overset{*}{\longleftrightarrow}$) and $\overset{=}{\to}$ (resp. $\overset{=}{\longleftrightarrow}$) are the transitive and reflexive closures of the relation $\to$ (resp. $\leftrightarrow$).

Let us give a hint the above definition. The property of decreasiness is stated for a relations, seen as a family of labelled binary relations. Such labels are equipped with a well-founded, strict, order such that every *peak* can be rejoined in a particular way, regulated by that specific order on labels.

The following theorem, due to van Oostrom, states that decreasiness implies confluence.

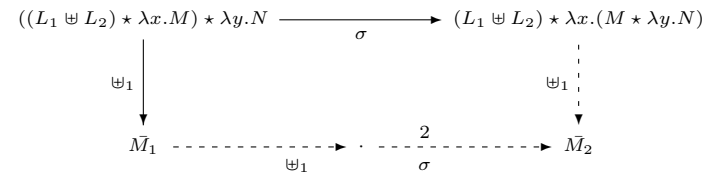
Theorem 2.5 (van Oostrom [vO94, vO08])**.** *Every locally decreasing rewriting relation $\mathcal{R}$ is confluent.*

Which order? Now the point is to find a proper labelling and a strict order on that labelling that satisfies the property of decreasiness. Let's start with harmless reductions, involving rules of union and empty.



By the above diagrams it seems clear that rules concerning the empty table should be top elements of the labelling we are searching for.

When it comes to comparing $\uplus_1$ vs. σ, the situation is a bit tricker because $\uplus_1$ only quasi-commutes over σ. The following diagrams shows that $\uplus_1$ must be made greater than σ.



where $\bar{M_1} = ((L_1 \star \lambda x.M) \uplus (L_2 \star \lambda x.M)) \star \lambda y.N)$, $\bar{M_2} = (L_1 \star \lambda x.(M \star \lambda y.N)) \uplus (L_2 \star \lambda x.(M \star \lambda y.N))$

The case for β_c vs $\uplus_2$, however, shows the need for a non-trivial approach, since depending in which context the rules are applied, we need either $\beta_c > \uplus_2$ or $\beta_c < \uplus_2$. Indeed,

$$\begin{array}{ccc} [V] \star \lambda x.(N \uplus P) & \xrightarrow{\beta_c} & (N \uplus P)\{V/x\} \\ \Big\downarrow{\scriptstyle \uplus_2} & & \Big\Vert \\ ([V] \star \lambda x.N) \uplus ([V] \star \lambda x.P) & \xrightarrow[2]{\beta_c} & N\{V/x\} \uplus P\{V/x\} \end{array}$$

... but ...

$V_1 = \lambda x.(M \star \lambda y.(N_1 \uplus N_2))$
$V_2 = \lambda x.((M \star \lambda y.N_1) \uplus (M \star \lambda y.N_2))$

$$\begin{array}{ccc} [V_1] \star \lambda z.([z] \star z) & \xrightarrow{\uplus_2} & [V_2] \star \lambda z.([z] \star z) \\ \Big\downarrow{\scriptstyle \beta_c} & & \Big\vdots{\scriptstyle \beta_c} \\ [V_1] \star V_1 & \xrightarrow[2]{\uplus_2} & [V_2] \star V_2 \end{array}$$

Generalized version of unions and Multi-reduction. Before stating the main result we have to introduce two new notion of reduction that will lead to a right labelling order to prove the decreasiness. The first one is a *generalized* version of rules $\uplus_1$ and $\uplus_2$, taking into account not just union symbols in the scope of the rule one by one, but all together.

Definition 2.6 (Generalized union step)**.** *Let us define as* generalized $\uplus_1$ *and* $\uplus_2$ *steps, notation* $\boldsymbol{Gen}\uplus_1$ *and* $\boldsymbol{Gen}\uplus_2$*, as follows*

$$\begin{array}{lll} \boldsymbol{Gen}\uplus_1) & (\ldots(M_1 \uplus M_2) \uplus \ldots \uplus M_n) \star \lambda x.N & \mapsto_{\boldsymbol{Gen}\uplus_1} \quad (M \star \lambda x.N) \uplus (M_2 \star \lambda x.N) \uplus \ldots \uplus (M_n \star \lambda x.N) \\ \boldsymbol{Gen}\uplus_2) & M \star \lambda x.(\ldots(N_1 \uplus N_2) \uplus \ldots \uplus N_n) & \mapsto_{\boldsymbol{Gen}\uplus_2} \quad (M \star \lambda x.N_1) \uplus (M \star \lambda x.N_2) \uplus \ldots \uplus (M \star \lambda x.N_n) \end{array}$$

Second, the confluence proof we are going to sketch avoids the issue with β_c vs. $\uplus_2$ reported above by considering *multiple reductions*. Roughly speaking, this means that we consider a labelling that comprehends reduction rules that can perform simultaneously in many 'part' of the term, called formally *positions*. For a fair formalization of these basic notions of rewriting theory, please see, *e.g.*, [BN98].

A **parallel rewrite step** is a sequence of reductions at a set P of **parallel** positions, ensuring that the result does not depend upon a particular sequentialization of P. Given a reduction step γ we define its parallel version as $\mathbf{Par}\gamma$.

We are now ready to state our main result:

Theorem 2.7 (Confluence)**.** λ_{SQL} *is confluent.*

Proof sketch.

1. All reduction rules strongly commute with !: proved by tedious inspection of all cases.
2. Under the following order for parallel rewriting steps, all remaining rules are decreasing as well: also proved by tedious inspection of all cases.

$$\mathbf{Par}\beta_c > \mathbf{Par}\sigma > \mathbf{ParGen}\uplus_2 > \mathbf{ParGen}\uplus_2 > \emptyset_1 > \emptyset_2$$

   The diagrams for the cases $\mathbf{Par}\uplus_1$ vs $\mathbf{Par}\uplus_2$ and $\mathbf{Par}\uplus_2$ vs $\emptyset_1$ only hold up to E. E.g., $\emptyset \; {}_{\emptyset_1}\!\leftarrow \emptyset \star \lambda x.M \uplus N \rightarrow_{\uplus_2} \rightarrow^2_{\emptyset_1} \emptyset \uplus \emptyset$.
3. Confluence is obtained combining the previous points with Lemma 2.3 and Theorem 2.5, following [ADJL17].

□

References

[ADJL17] Ali Assaf, Gilles Dowek, Jean-Pierre Jouannaud, and Jiaxiang Liu. Untyped Confluence In Dependent Type Theories. working paper or preprint, April 2017.

[AGK20] Beniamino Accattoli, Stéphane Graham-Lengrand, and Delia Kesner. Tight typings and split bounds, fully developed. *J. Funct. Program.*, 30:e14, 2020.

[AKR23] Sandra Alves, Delia Kesner, and Miguel Ramos. Quantitative global memory. arXiv:2303.08940, 2023.

[BN98] Franz Baader and Tobias Nipkow. *Term rewriting and all that.* Cambridge University Press, 1998.

[dLT20] U. de' Liguoro and R. Treglia. The untyped computational λ-calculus and its intersection type discipline. *Theor. Comput. Sci.*, 846:141–159, 2020.

[dT21] Ugo de'Liguoro and Riccardo Treglia. Intersection types for a λ-calculus with global store. In Niccolò Veltri, Nick Benton, and Silvia Ghilezan, editors, *PPDP 2021: 23rd International Symposium on Principles and Practice of Declarative Programming, Tallinn, Estonia, September 6-8, 2021*, pages 5:1–5:11. ACM, 2021.

[FGdLT22] Claudia Faggian, Giulio Guerrieri, Ugo de' Liguoro, and Riccardo Treglia. On reduction and normalization in the computational core. *Mathematical Structures in Computer Science*, 32(7):934–981, 2022.

[Fil94] A. Filinski. Representing monads. In *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 446–457. ACM Press, 1994.

[Lev99] Paul Blain Levy. Call-by-push-value: A subsuming paradigm. In *Typed Lambda Calculi and Applications, 4th International Conference (TLCA'99)*, volume 1581 of *Lecture Notes in Computer Science*, pages 228–242, 1999.

[Mog89] E. Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89)*, pages 14–23. IEEE Computer Society, 1989.

[Mog91] E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.

[PP02] G. D. Plotkin and J. Power. Notions of computation determine monads. In *FOSSACS 2002*, volume 2303 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2002.

[PP03] G. D. Plotkin and J. Power. Algebraic operations and generic effects. *Appl. Categorical Struct.*, 11(1):69–94, 2003.

[RC20] Wilmer Ricciotti and James Cheney. Strongly normalizing higher-order relational queries. In *5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020, June 29-July 6, 2020, Paris, France (Virtual Conference)*, pages 28:1–28:22, 2020.

[Ter03] Terese. *Term rewriting systems*, volume 55 of *Cambridge tracts in theoretical computer science*. Cambridge University Press, 2003.

[vO94] Vincent van Oostrom. Confluence by decreasing diagrams. *Theor. Comput. Sci.*, 126(2):259–280, 1994.

[vO08] Vincent van Oostrom. Confluence by decreasing diagrams converted. In *Rewriting Techniques and Applications, 19th International Conference, RTA 2008,*, volume 5117 of *Lecture Notes in Computer Science*, pages 306–320. Springer, 2008.