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

Claudio Sacerdoti Coen[1], Riccardo Treglia[1,*]

[1] *Università di Bologna, Bologna, Italy, Mura Anteo Zamboni, 7, 40126 Bologna (BO), Italy*

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 captures 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. In the second part, we study a Curry style type assignment system for the calculus. We introduce an idempotent intersection type system establishing type invariance under conversion.

Keywords

Lambda calculus, Monads, Confluence, Intersection Types, Databases

1. Introduction to the Calculus: Syntax and Reduction Relation

The second author et al. have introduced and studied in [1, 2] the computational core $\lambda_{\copyright}$, a λ-calculus inspired by Moggi's computational one [3], [4]. The calculus differentiates between values and computations, the latter obtained via return/bind constructs for a generic monad. The operational semantics is obtained simply by 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 $[\cdot]$, is now interpreted as forming a singleton. The rewriting rules partially capture the algebraicity of the operations in the sense of Plotkin and Power [5, 6] 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 described so far does not allow the formation of multisets of multisets, because multisets are not values. To overcome the issue, we

ICTCS'23: 24th Italian Conference on Theoretical Computer Science, September 13–15, 2023, Palermo, Italy

*Corresponding author.

claudio.sacerdoticoen@unibo.it (C. S. Coen); riccardo.treglia@unibo.it (R. Treglia)

https://www.cs.unibo.it/~sacerdot/ (C. S. Coen); https://sites.google.com/view/riccardotreglia/home (R. Treglia)

0000-0002-4360-6016 (C. S. Coen); 0000-0002-9731-1248 (R. Treglia)

add two more co-monadic constructs to reflect computations into values, following ideas by [7]. These constructs are the thunk/force constructs of Levy's call-by-push-value calculus [8]; 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 calculus we are going to introduce takes inspiration from the (untyped) NRCλ calculus [9], and it is to it as $\lambda_{\copyright}$ is to the (untyped) λ-calculus. Indeed, we introduce 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 $\lambda_{\copyright}$, thus providing an explicit monadic formulation of Ricciotti et al's calculus.

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

Intersection Types were introduced by Coppo and Dezani-Ciancaglini in the late 70's [10] to overcome the limitations of Curry's type discipline and enlarge the class of terms that can be typed. This is reached by means of a new type constructor, the *intersection*. Thus, one can assign a finite set of types to a term, thus providing a form of finite, *ad hoc* polymorphism.

In the same way that simple types guarantee termination, intersection types do the same. However, they also *characterize* termination, that is, they type *all* terminating λ-terms. Intersection types have also a very elegant semantic flavour, since they may be seen as a syntactic presentation of denotational models. Notwithstanding, here we do not delve into the denotational semantics of the calculus, and keep the treatment at the syntactical level. In fact, intersection types have shown to be remarkably flexible, since different termination forms can be characterized by tuning details of the type system (*e.g.*, weak/strong normalization, head/weak/call-by-value evaluation). In the present work, we introduce an idempotent intersection type discipline and prove it enjoys the subject convertibility, *i.e.*, the type is preserved not only by reduction, but also by expansion. This is the key property to reach the soundness and completeness results of the type system with respect to termination.

To get an account of the history and expressivity of intersection types, see for example the recent survey by Bono and Dezani-Ciancaglini [11].

Contributions. The first contribution of the work is the design of the λ_{SQL}-calculus in Section 2, 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 [12, 13]), but here it immediately highlights, for example, the need to introduce other kinds of constructs, such as the comonadic unit, that could be added to $\lambda_{\copyright}$ independently of the algebraic operators.

The second contribution, treated in Section 3, 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 van Oostrom's decreasing diagram technique [14], the most difficult point of which is to find the order relation between the labels of the calculus reduction rules. This will be done by considering orthogonal and nested closures of certain reduction rules, inspired by the work in [15], postponing in a final step the commutation with respect to the union operator.

Section 4 is devoted to the third contribution: an idempotent, monadic, intersection types assignment system, proved to enjoy subject convertibility. The intersection type theory we present is monadic version of strict intersection types in the case the monad into account is the multiset monad equipped with the possibility of reflect and reify types. This contribution can be seen as a first step to characterize convergent terms of the calculus and as a first move in obtaining a resource-aware type system for the calculus into consideration.

Future work and related ones are discussed in Section 5.

2. Syntax and Reduction

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

Definition 2.1 (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 multiset 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 2.2 (Reduction)**.** *The reduction relation is the union of the following binary relations over Com:*

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

The first two rules, taken from $\lambda_{\copyright}$, are oriented monadic equations. The next two 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* $\rightarrow_{\lambda_{\mathrm{SQL}}}$ (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 value contexts as follows:

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

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 [9].

Definition 2.3 (Equational theory E)**.**

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

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). This choice was made in order to keep the calculus as close as possible to the reference NRCλ calculus in [9].

3. 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 3.1. *Given a reduction relation $\to$ and an equational theory $=_E$, we say that $\to$ commutes over $=_E$ if for all M, N, L such that $M =_E N \to L$, there exists P such that $M \to P =_E L$.*

Lemma 3.2 (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 3.3. *$=_E$ commutes with $\to$.*

Hence, by Lemma 3.3 one needs just the confluence of $\to$ to assert the confluence of $\to$ modulo E.

Remark 3.4. One can be also interested in the modulo confluence, which is in general different from the confluence modulo (see ch. 14 of [16]). In fact, the equational theory E induces an equivalence relation on computations *Com*, where the equivalence class $[M]_E$ of an element a M consists of all elements N such that $M =_E N$. The set of all equivalence classes, denoted by $Com/=_E$, is called the quotient set of *Com* modulo E. It is easy to see that in our specific case, even if we are not interested in it, confluence modulo E implies the confluence of $Com/=_E$.

Decreasing diagram. Now that is possible to omit the equational theory induced by Definition 2.3, we need to prove the commutation of all the reduction rules, and in this intent, we use decreasing diagrams by van Oostrom [14, 17]. 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 3.5 (Decreasing, [14])**.** *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:*

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

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. $\leftrightarrow$).

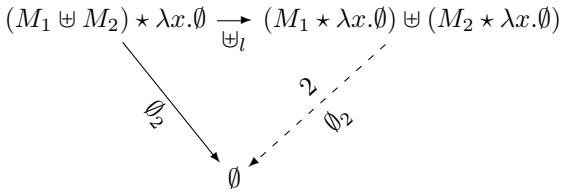
Let us give a hint about the above definition. The property of decreasiness is stated for relations, seen as a family of labelled binary relations. Such labels are equipped with a well-founded, strict, order such that every *peak* can be rejoined in a particular way, regulated by that specific order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

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

Theorem 3.6 (van Oostrom [14, 17])**.** *Every locally decreasing rewriting relation $\mathcal{R}$ is confluent.*

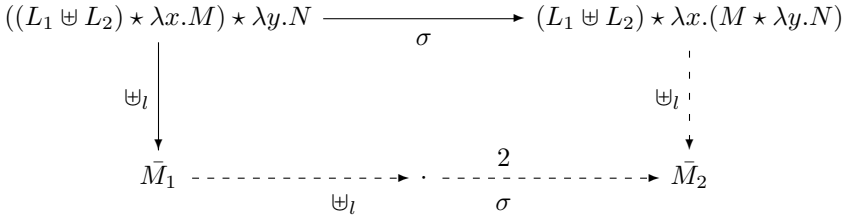
In [14], the method is guaranteed to be complete in the sense that any (countable) confluent rewrite relation can be equipped with such a labelling. But by undecidability of confluence completeness also entails that finding such a labelling is, in general, difficult. So to prove confluence of the relation in Definition 2.2 one needs to prove it decreasing with respect to some labelling. This means rearranging the family in such a way that the union is still the relationship we want to prove the confluence of, but the indices of the family are rearranged to comply with a labelling that fits the definition of decreasiness.

Which order? Now the point is to find a proper labelling and a strict order on that labelling that satisfies the property of decreasiness. If one considers diagrams involving rules of $\uplus_l$ or $\uplus_r$ vs. $\emptyset_1$ and $\emptyset_2$, it is easy to perceive how these rules should be ordered as labels of potential labellings. Consider, for instance, the following diagram:



In fact, the rules concerning the empty table, $\emptyset_1$ and $\emptyset_2$, can be bottom elements of the order over labels we are searching for (read it as: these rules can always be postponed at the end of a reduction sequence).

When it comes to comparing $\uplus_l$ vs. σ, the situation is a bit trickier because $\uplus_l$ only quasi-commutes over σ. The following diagrams show that $\uplus_l$ 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 $\uplus_r$, however, shows the need for a non-trivial approach, since depending on which context the rules are applied, we need either $\beta_c > \uplus_r$ or $\beta_c < \uplus_r$. Indeed,

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

... but ...

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

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

Generalized version of $\uplus_l$ and $\uplus_r$ The case for $\uplus_l$ vs. $\uplus_r$ can seem innocent, for example:

$$\begin{array}{ccc}
(M_1 \uplus M_2) \star \lambda x.(N_1 \uplus N_2) & \xrightarrow[\uplus_l]{\qquad} & (M_1 \star \lambda x.(N_1 \uplus N_2)) \uplus (M_2 \star \lambda x.(N_1 \uplus N_2)) \\
\Big\downarrow{\scriptstyle \uplus_r} & & \Big\vdots{\scriptstyle \uplus_r}\, 2 \\
((M_1 \uplus M_2) \star \lambda x.N_1) \uplus ((M_1 \uplus M_2) \star \lambda x.N_2) & \overset{2}{\underset{\uplus_l}{\dashrightarrow}} & \bar{M} =_E \hat{M}
\end{array}$$

where $\bar{M} \equiv (M_1 \star \lambda x.N_1) \uplus (M_2 \star \lambda x.N_1) \uplus (M_1 \star \lambda x.N_2) \uplus (M_2 \star \lambda x.N_2)$ and $\hat{M} \equiv (M_1 \star \lambda x.N_1) \uplus (M_1 \star \lambda x.N_2) \uplus (M_2 \star \lambda x.N_1) \uplus (M_2 \star \lambda x.N_2)$

Remark 3.7. Actually, for more complex terms the diagram is not so elegant. Consider for example the term $M \equiv (M_1 \uplus M_2) \star \lambda x((N_1 \uplus N_2) \uplus N_3)$. If one is seeking a modular proof of confluence, it is possible to consider the rewriting system made by just the rules $\uplus_l$ and $\uplus_r$. Both together are weakly Church-Rosser (provable by easy induction) and terminating (since the number of $\uplus$ symbols in a term is finite), thus by Newman's Lemma, the rewrite system is Church-Rosser. By the way, we are searching for the right application of van Oostrom's decreasing diagram, hence we are to introduce a generalized version of $\uplus_l$ and $\uplus_r$, respectively.

Definition 3.8 (Generalized union step)**.** *Let us define as* generalized $\uplus_l$ *and* $\uplus_r$ *steps as follows*

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

Multi-reduction. The confluence proof we are going to sketch avoids the issue with β_c vs. $\uplus_r$ 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 'parts' of the term, called formally *positions*. For a fair formalization of these basic notions of rewriting theory, please see, *e.g.*, [18].

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 3.9 (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.

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

   The diagrams for the cases $\mathbf{Par}\uplus_l$ vs $\mathbf{Par}\uplus_r$ and $\mathbf{Par}\uplus_r$ vs $\emptyset_1$ only hold up to E. E.g., $\emptyset \; {}_{\emptyset_1}\!\leftarrow \emptyset \star \lambda x.M \uplus N \rightarrow_{\uplus_r} \rightarrow^2_{\emptyset_1} \emptyset \uplus \emptyset$.
3. Confluence is obtained combining the previous points with Lemma 3.3 and Theorem 3.6, following [15].

□

4. The Intersection Type Assignment System

Intersection types are an extension of Curry's simple type assignment system to untyped λ-terms, obtained by adding new types $\sigma \wedge \sigma'$ to be assigned to terms that have both type σ and σ'. Intersection type assignment systems form a whole family in the literature; see [19] part III. What all these families have in common is that intersection types embody a sort of *ad hoc* polymorphism, in which the conjunction of semantically unrelated types can be contemplated. An advantage is having the ability to assign two different types to two distinct occurrences of a variable, which allows more terms to be typed, thus enlarging the class of terms that can be typed by Curry's type discipline.

In building an intersection type discipline for λ_{SQL}, we actually do not extend the system in [1] (*i.e.*, we do not consider a type theory with subtyping in the BCD fashion, [20]). In fact, we present a syntax-directed type assignment system, without subtyping, to type tables and operations of merging tables, by specializing the generic monad T to the *multiset monad* (the semantical understanding of this will be the focus of a companion paper). Specifically, we adapt strict intersection type theory as in [21, 22] to the case of λ_{SQL}, meaning that the introduction of intersection is restricted merely to values. We introduce two sorts of types corresponding to the two sorts of terms in Definition 2.1, and a third kind, called *blind types*, inspired by [23], typing particular occurrences

of values. The choices to avoid subtyping, and hence the proof of a generation lemma (also named 'inversion lemma', typical lemma needed in intersection type disciplines with subtyping, see [19]), and to introduce a third layer of types, the blind types ones, that at this stage are pleonastic, have been made to smoothly move in a future step to a *quantitative version* of the present type system obtained via non-idempotent, strict intersection [24].

Definition 4.1 (Intersection Types Syntax)**.** *Let α range over a countable set of type variables (the atoms); then we define three sorts of types by mutual induction as follows:*

$$\begin{array}{rrcll} \mathit{ValType}: & \delta & ::= & \alpha \mid \delta \to \tau \mid \bigwedge_{n\geq 0} \delta \mid \langle\!\langle \tau \rangle\!\rangle & \textit{(}\text{value types}\textit{)} \\ \mathit{ComType}: & \tau & ::= & [\delta] \mid \hat{\varnothing} \mid \tau \uplus \tau & \textit{(}\text{computation types}\textit{)} \\ \mathit{BlindType}: & \xi & ::= & \wedge\varnothing \to \tau & \textit{(}\text{blind types}\textit{)} \end{array}$$

We assume that $\wedge$ and $\uplus$ take precedence over $\to$ and that $\to$ associates to the right, so that $\delta \to \tau \wedge \tau'$ reads as $\delta \to (\tau \wedge \tau')$, and $\delta \to \tau \uplus \tau'$ reads as $\delta \to (\tau \uplus \tau')$. We note the empty intersection as $\wedge\varnothing$.

We introduce the emptyset type $\hat{\varnothing}$, as a constant, and should not be mistaken as the symbol for an empty union. As for the terms, we equip computational types in Definition 4.1 with an equational theory stating the commutativity of union and the idempotency of the emptyset type $\hat{\varnothing}$ w.r.t. the union of types. We then introduce for all the computational types τ a boolean predicate $\texttt{emptyin}(\tau)$, that is true if and only if $\hat{\varnothing} \in \tau$. In such a way, we can speak of *canonical form of computational types* for λ_{SQL}: it is easy to see that for every $\tau \in \mathit{ComType}$ there exists a *canonical form* (up to commutativity of union) $\biguplus_{n\geq 0}[\delta_i] \uplus \hat{\varnothing}$ if $\texttt{emptyin}(\tau)$, or $\biguplus_{n>0}[\delta_i]$, otherwise. These forms will be used in type assignment systems to clarify and handle generic computational types when needed.

Type Assignment System. We are now in place to introduce the type assignment system for λ_{SQL}.

Definition 4.2 (Type assignment)**.** *A* basis *is a finite set of typings $\Gamma = \{x_1 : \delta_1, \dots x_n : \delta_n\}$ with pairwise distinct variables x_i, whose* domain *is the set $Dom(\Gamma) = \{x_1, \dots, x_n\}$. A basis determines a function from variables to types such that $\Gamma(x) = \delta$ if $x : \delta \in \Gamma$, $\Gamma(x)$ is the empty intersection $\wedge\varnothing$, otherwise.*

A judgment *is an expression of either shapes: $\Gamma \vdash V : \delta$ or $\Gamma \vdash M : \tau$. It is* derivable *if it is the conclusion of a derivation according to the rules in Figure 1.*

The blind types to which we are taking inspiration are the ones presented by Kesner et al. in [23, 25] to grant strong normalization even in the presence of terms whose occurrences will all be erased during computation. The intersection types discipline assigns to terms being erased $\wedge\varnothing$ and therefore the terms would not be typed, not capturing their divergence. Instead, we type them with a blind type, which states that the term will not receive at runtime any input to be used and it can return any output.

$$\frac{x : \delta \in \Gamma}{\Gamma \vdash x : \delta}\,(\mathsf{Ax}) \qquad \frac{}{\vdash \emptyset : \hat{\varnothing}}\,(\mathsf{Ax}_{\varnothing})$$

$$\frac{\Gamma \vdash M : \tau}{\Gamma \vdash \langle\!\langle M \rangle\!\rangle : \langle\!\langle \tau \rangle\!\rangle}\,(\langle\!\langle\cdot\rangle\!\rangle\ \mathsf{I}) \qquad \frac{\Gamma \vdash V : \langle\!\langle \tau \rangle\!\rangle}{\Gamma \vdash\, !V : \tau}\,(\langle\!\langle\cdot\rangle\!\rangle\ \mathsf{E})$$

$$\frac{\Gamma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \Gamma(x) \to \tau}\,(\to \mathsf{I}) \qquad \frac{\Gamma \vdash \lambda x.M : \xi}{\Gamma \vdash \lambda x.M : \wedge\varnothing}\,(\mathit{Blind})$$

$$\frac{(\Gamma \vdash V : \delta_i)_{i=1,\ldots,n}}{\Gamma \vdash V : \bigwedge_{i=1}^{n} \delta_i}\,(\wedge I) \qquad \frac{\Gamma \vdash V : \delta}{\Gamma \vdash [V] : [\delta]}\,(\mathit{unit}\ \mathsf{I})$$

$$\frac{\Gamma \vdash M : \tau \quad \Gamma \vdash N : \tau'}{\Gamma \vdash M \uplus N : \tau \uplus \tau'}\,(\uplus\ \mathsf{I})$$

$$\frac{\Gamma \vdash M : \biguplus_{n>0} [\delta_i] \biguplus_{m\leq 1} \hat{\varnothing} \qquad \Gamma \vdash V : \bigwedge_{n>0} \delta_i \to \tau_i}{\Gamma \vdash M \star V : \biguplus_{n>0} \tau_i \biguplus_{m\leq 1} \hat{\varnothing}}\,(\star\ \mathsf{I})$$

$$\frac{\Gamma \vdash M : \hat{\varnothing} \qquad \Gamma \vdash V : \wedge\varnothing}{\Gamma \vdash M \star V : \hat{\varnothing}}\,(\star_{\varnothing}\ \mathsf{I})$$

Figure 1: Intersection types assignment system.

The union operator for types is introduced just for computations as regulated in rule ($\uplus$ I). The introduction of bind is split in two, depending on the cases in which the type associated with the computation is just an empty union type or not. In fact, in the case of ($\star_{\varnothing}$ I) we explicitly ask for the value to have an empty intersection to deal with the case the computation has just the empty union type. The rule ($\star$ I), instead, takes care of the case in which the computation has non empty union type (n is strictly greater than 0) and possibly the empty union type (this is the meaning of $\biguplus_{m\leq 1} \hat{\varnothing}$); in such a case, the empty union type, if it is in the premise, is propagated to the resulting type.

Results. We are now in place to state the main results of the presented type system.

Theorem 4.3 (Subject Reduction)**.** *If* $\Gamma \vdash M : \tau$ *and* $M \to N$*, then* $\Gamma \vdash N : \tau$*.*

Theorem 4.4 (Subject Expansion)**.** *If* $\Gamma \vdash N : \tau$ *and* $M \to N$*, then* $\Gamma \vdash M : \tau$*.*

The above theorems state two results: the first is the subject reduction, a desirable property for all type systems. The second property, the subject expansion, on the other

hand, is typical of intersection types and the key to proving characterisation results. This property, indeed, states that the type of a term is preserved even going backwards in its reduction.

5. Conclusions

In the present work, we have introduced a computational λ-calculus λ_{SQL} as an extension of the computational core $\lambda_{\copyright}$ [1]. We have proved the confluence of the calculus and presented an intersection type assignment system enjoying subject reduction and expansion.

We leave for future works the full characterization of convergent terms, via soundness and completeness of the type system, plus a deep investigation on the semantical level via a filter model construction as in [26].

Related works. As mentioned in the Introduction, λ_{SQL} stems from NRCλ calculus in [9]. The first 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.

In designing our type system, we were deeply influenced by the desire to reach a quantitative type system for λ_{SQL}, and in doing so we took inspiration from [24, 13], since our splitting rules in the presence/absence of the empty union type are reminiscent of the persistent/consuming rules in the cited works.

We find that our calculus and type system have substantial connections with dependent type systems, even if not yet explored in detail. This is the case with the intuition borrowed from [15] in proving confluence and with [27] where a dependent intersection type system is presented. Linking our types to the last cited work, it is easy to notice that our types stand to that system as the lists stand to the natural numbers, where the bind in fact is the iterator. As said, we leave a deepening of these insights for future work.

Long-term perspectives. Considering the structure of the present work, we set two long-term goals. The first is related to the confluence proof based on decreasing diagrams in Section 3, since the labelling extracts an order over reduction rules to design a well-behaved normalizing strategy.

Concerning the type system, our idempotent intersection type one is just a mid-term objective, since we are interested in defining an appropriate intersection type system based on tight multi-types [28] to capture quantitatively the set of terminating queries according to strategy extracted from the confluence proof. Such non-idempotent type systems have

been proven to be useful in detecting the length of normalizing reductions and the size of the normal forms, which in our case is the size of the computed SQL queries, via the type system itself. As a result, extending such non-idempotent intersection type disciplines for λ_{SQL} should capture even more quantitative, computational information about the queries themselves.

Acknowledgments

We would like to thank th reviewers for taking the necessary time and effort to review the manuscript. We sincerely appreciate all your valuable comments and suggestions, which helped us in improving the quality of the manuscript.

References

[1] U. de'Liguoro, R. Treglia, The untyped computational λ-calculus and its intersection type discipline, Theor. Comput. Sci. 846 (2020) 141–159. URL: https://doi.org/10.1016/j.tcs.2020.09.029. doi:10.1016/j.tcs.2020.09.029.

[2] C. Faggian, G. Guerrieri, U. de' Liguoro, R. Treglia, On reduction and normalization in the computational core, Mathematical Structures in Computer Science 32 (2022) 934–981. doi:10.1017/S0960129522000433.

[3] E. Moggi, Computational lambda-calculus and monads, in: Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), IEEE Computer Society, 1989, pp. 14–23. doi:10.1109/LICS.1989.39155.

[4] E. Moggi, Notions of computation and monads, Inf. Comput. 93 (1991) 55–92. doi:10.1016/0890-5401(91)90052-4.

[5] G. D. Plotkin, J. Power, Notions of computation determine monads, in: FOSSACS 2002, volume 2303 of *Lecture Notes in Computer Science*, Springer, 2002, pp. 342–356. URL: https://doi.org/10.1007/3-540-45931-6_24. doi:10.1007/3-540-45931-6_24.

[6] G. D. Plotkin, J. Power, Algebraic operations and generic effects, Appl. Categorical Struct. 11 (2003) 69–94. doi:10.1023/A:1023064908962.

[7] A. Filinski, Representing monads, in: H. Boehm, B. Lang, D. M. Yellin (Eds.), Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994, ACM Press, 1994, pp. 446–457. URL: https://doi.org/10.1145/174675.178047. doi:10.1145/174675.178047.

[8] P. B. 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*, 1999, pp. 228–242. URL: https://doi.org/10.1007/3-540-48959-2_17. doi:10.1007/3-540-48959-2_17.

[9] W. Ricciotti, J. 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), 2020, pp. 28:1–

28:22. URL: https://doi.org/10.4230/LIPIcs.FSCD.2020.28. doi:10.4230/LIPIcs.FSCD.2020.28.

[10] M. Coppo, An extended polymorphic type system for applicative languages, in: P. Dembinski (Ed.), Mathematical Foundations of Computer Science 1980 (MFCS'80), Proceedings of the 9th Symposium, Rydzyna, Poland, September 1-5, 1980, volume 88 of *Lecture Notes in Computer Science*, Springer, 1980, pp. 194–204. URL: https://doi.org/10.1007/BFb0022505. doi:10.1007/BFb0022505.

[11] V. Bono, M. Dezani-Ciancaglini, A tale of intersection types, in: H. Hermanns, L. Zhang, N. Kobayashi, D. Miller (Eds.), LICS '20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany, July 8-11, 2020, ACM, 2020, pp. 7–20. URL: https://doi.org/10.1145/3373718.3394733. doi:10.1145/3373718.3394733.

[12] U. de'Liguoro, R. Treglia, Intersection types for a λ-calculus with global store, in: N. Veltri, N. Benton, S. Ghilezan (Eds.), PPDP 2021: 23rd International Symposium on Principles and Practice of Declarative Programming, Tallinn, Estonia, September 6-8, 2021, ACM, 2021, pp. 5:1–5:11. URL: https://doi.org/10.1145/3479394.3479400. doi:10.1145/3479394.3479400.

[13] S. Alves, D. Kesner, M. Ramos, Quantitative global memory 13923 (2023) 53–68. URL: https://doi.org/10.1007/978-3-031-39784-4_4. doi:10.1007/978-3-031-39784-4_4.

[14] V. van Oostrom, Confluence by decreasing diagrams, Theor. Comput. Sci. 126 (1994) 259–280. URL: https://doi.org/10.1016/0304-3975(92)00023-K. doi:10.1016/0304-3975(92)00023-K.

[15] A. Assaf, G. Dowek, J.-P. Jouannaud, J. Liu, Untyped Confluence In Dependent Type Theories (2016). URL: https://inria.hal.science/hal-01515505.

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

[17] V. 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*, Springer, 2008, pp. 306–320. doi:10.1007/978-3-540-70590-1_21.

[18] F. Baader, T. Nipkow, Term rewriting and all that, Cambridge University Press, 1998.

[19] H. Barendregt, W. Dekkers, R. Statman, Lambda Calculus with Types, Perspectives in Logic, Cambridge University Press, 2013. doi:10.1017/CBO9781139032636.

[20] H. Barendregt, M. Coppo, M. Dezani-Ciancaglini, A filter lambda model and the completeness of type assignment, Journal of Symbolic Logic 48 (1983) 931–940. doi:10.2307/2273659.

[21] S. van Bakel, Intersection Type Disciplines in Lambda Calculus and Applicative Term Rewriting Systems, Ph.D. thesis, Department of Computer Science, University of Nijmegen, 1993.

[22] S. van Bakel, Strict intersection types for the lambda calculus, ACM Comput. Surv. 43 (2011) 20:1–20:49. URL: https://doi.org/10.1145/1922649.1922657. doi:10.1145/1922649.1922657.

[23] D. Kesner, P. Vial, Consuming and persistent types for classical logic, in: H. Hermanns, L. Zhang, N. Kobayashi, D. Miller (Eds.), LICS '20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany, July 8-11, 2020, ACM, 2020, pp. 619–632. URL: https://doi.org/10.1145/3373718.3394774. doi:10.1145/3373718.3394774.

[24] D. Kesner, A. Viso, Encoding tight typing in a unified framework, in: F. Manea, A. Simpson (Eds.), 30th EACSL Annual Conference on Computer Science Logic, CSL 2022, February 14-19, 2022, Göttingen, Germany (Virtual Conference), volume 216 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, pp. 27:1–27:20. URL: https://doi.org/10.4230/LIPIcs.CSL.2022.27. doi:10.4230/LIPIcs.CSL.2022.27.

[25] D. Kesner, P. Vial, Non-idempotent types for classical calculi in natural deduction style, Log. Methods Comput. Sci. 16 (2020). URL: https://doi.org/10.23638/LMCS-16(1:3)2020. doi:10.23638/LMCS-16(1:3)2020.

[26] U. de'Liguoro, R. Treglia, From semantics to types: The case of the imperative λ-calculus, Theor. Comput. Sci. 973 (2023) 114082. URL: https://doi.org/10.1016/j.tcs.2023.114082. doi:10.1016/j.tcs.2023.114082.

[27] U. D. Lago, M. Gaboardi, Linear dependent types and relative completeness, Log. Methods Comput. Sci. 8 (2011). URL: https://doi.org/10.2168/LMCS-8(4:11)2012. doi:10.2168/LMCS-8(4:11)2012.

[28] B. Accattoli, S. Graham-Lengrand, D. Kesner, Tight typings and split bounds, fully developed, J. Funct. Program. 30 (2020) e14. URL: https://doi.org/10.1017/S095679682000012X. doi:10.1017/S095679682000012X.