A Simply Numbered Lambda Calculus

Friedrich Steimann
Fernuniversität in Hagen, Germany

Abstract

While programming languages traditionally lean towards functions, query languages are often relational in character. Taking the *relations language* of Harkes and Visser as a starting point, I explore how the functional paradigm, represented by the lambda calculus, can be extended to form the basis of a relational language. It turns out that a straightforward extension with strings of terms not only supports surprisingly many features of the relations language, but also opens it up for higher-order relations, one prominent feature the relations language does not offer.

2012 ACM Subject Classification Software and its engineering → Multiparadigm languages; Theory of computation → Functional constructs

Keywords and phrases multiplicities, strings, lambda calculus, relational programming

Digital Object Identifier 10.4230/OASIcs.EVCS.2023.24

Related Version *Full Version*: `https://feu.de/ps/pubs/SNLC-TR.pdf` [7]

Acknowledgements I thank Sebastian Erdweg for the joint exploration of a lambda calculus with plural terms on the last day of Dagstuhl Research Meeting 21183. My anonymous reviewers greatly helped improve this paper.

I am forever grateful for the friendliness and collegiality of Eelco Visser that I enjoyed.

1 Introduction

At 2014's instalment of the Software Language Engineering (SLE) conference[1], Eelco Visser and his then PhD student Daco Harkes presented their *relations language* [1], a language designed for querying the object graphs that represent the data of various web-based information systems [2]. The language features first-class, n-ary, bidirectional relations between attributed objects and offers concise path expressions (using the usual dot notation; e.g., `x.children.age`) for convenient querying and navigation. To harden the language with static guarantees, the places of relations are constrained not only by types, but also by so-called *native multiplicities*, which abstract from the number of objects other objects may relate to in each place, and which are said to be *orthogonal to types* [1, 2].

To grant some generality in query expressions, the relations language defines an arithmetic sublanguage whose operators accept arbitrarily many operands in each place. For instance, addition can handle multiple numbers in the places of both summands, and even no numbers (in other languages represented by a null value). This is expressed by (big-step) evaluation rules of the kind

$$[\textsc{Add}]\ \frac{e_1 \Downarrow V_1 \qquad e_2 \Downarrow V_2}{e_1 \oplus e_2 \Downarrow \{\!|\, v_1 + v_2 \mid v_1 \in V_1, v_2 \in V_2 \,|\!\}},$$

in which V_1 and V_2 are (flat) bags (here delimited by braces $\{\!|$ and $|\!\}$) and which means that if e_1 evaluates to $\{\!|\, 1, 2 \,|\!\}$ and e_2 to $\{\!|\, 3, 4 \,|\!\}$, then $e_1 \oplus e_2$ evaluates to $\{\!|\, 4, 5, 5, 6 \,|\!\}$. Multiple numbers naturally result from accessing numeric attributes (such as `age`) on multiple objects

[1] Starting with its inaugural issue in 2008, Eelco Visser contributed a total of 18 papers to the SLE conference series, and served as its General Chair only last year (2021).

(as they may result from query expressions such as `x.children`) in one expression (e.g., `x.children.age`). Note that since multiplicity is not encoded in type (the two are thought to be orthogonal), the type system of the relations language grants addition of pairs of multiple numbers without further measures (such as coercions). And yet, these kinds of additions appear to be of little use in a query language[2], unless multiplicities are constrained to "zero or one" (multiplicity ? in the relations language), in which case it amounts to adding under the conditions of an `Optional` type [1, 2]. For instance, if e_1 evaluates to $\{\!|\, 1 \,|\!\}$ (corresponding to `Some 1`) and e_2 to $\{\!|\, |\!\}$ (corresponding to `None`), then, by above rule Add, $e_1 \oplus e_2$ evaluates to $\{\!|\, |\!\}$. Having the semantics of `Optional` for free is certainly a commendable feature of the relations language.

While adding multiple numbers through Add appears to be of little use in querying, aggregating multiple numbers is certainly central. This is also acknowledged by the relations language, which introduces special aggregation operations for this purpose. Evaluation of these operations is defined through pairs of rules like

$$[\textsc{Sum}]\ \frac{e \Downarrow \{\!|\, v_1, \ldots, v_n \,|\!\} \qquad n \geq 1}{\mathrm{sum}(e) \Downarrow \{\!|\, \Sigma_{i=1}^{n} v_i \,|\!\}} \qquad\qquad [\textsc{Sum0}]\ \frac{e \Downarrow \{\!|\, |\!\}}{\mathrm{sum}(e) \Downarrow \{\!|\, 0 \,|\!\}}\ .$$

Note that while Add distributes addition over its multiple operands and collects the results, Sum must treat the multiple as a whole.

While the relations language is very general in its coverage of arithmetic (and logical) operations (see, e.g., [6] for a more constrained approach), it is somewhat less so in other respects. Specifically, even though relations are first-class, it does not feature higher-order relations (relations of relations). Like higher-order functions, higher-order relations would not only let users define their own operations, but would also allow the representation of higher-degree (n-ary) relations as nested binary relations (in analogy to the currying of functions), thereby increasing the expressiveness of the language while at the same time reducing its size.

With this tribute to Eelco Visser's work, I aim to provide a common basis for the core constructs of the relations language, navigation of relations and computing with multiple numbers. I do this by extending the lambda calculus (LC) with strings of elementary terms and values. By associating different multiplicities (in this work called *numbers*, alluding to the grammatical category *number* with values *singular* and *plural* [6]) with strings, I not only show that the resulting calculus, which I have pretentiously dubbed "simply *numbered* lambda calculus" (SNLC), has interesting safety properties, but also shed new light on the relation of typing and numbering which, when moving to higher-order functions (or relations), appear to be parallel instead of orthogonal as previously thought: specifically, while number and type may be independent, the structure of the here introduced number specifiers parallels that of function types.

2 Extending the Lambda Calculus with Strings of Terms

Rather than using lists (which can be encoded in the pure LC, but lead to a polymorphic typing discipline) to represent multiples, I will extend the LC with strings. Compared to other monoids [8], strings have the advantage of being purely syntactical (with concatenation as the monoid operation); unlike lists, strings are inherently flat (there are no strings of strings; Section 5 will spell out what this means).

[2] see [8] for a discussion

2.1 Syntax

The syntax of my extended LC has elementary terms, e, and strings of elementary terms, t, which are terms, too:

$$t ::= \overline{e}$$
$$e ::= x \mid \lambda x\,.\,t \mid t\ t$$

To spell out a string $\overline{e}$, I write $e_1 \cdot \ldots \cdot e_n$ instead of $e_1 \ldots e_n$; I do so to distinguish the string $e_1 \cdot e_2$ unmistakably from the function application $e_1\ e_2$.[3] Note that in a function application $t_2\ t_1$, (non-elementary) strings can occur in three places: at the argument position (t_1), at the function position (t_2), and at the function body positions.

For a term $t = e_1 \cdot \ldots \cdot e_n$, I say that t has length n. For $n = 0$ (the empty string of terms), I write ϵ, to which I refer as *nothing*; e.g., the function $\lambda x\,.\,\epsilon$ is said to return nothing. For terms $t_1 \cdot \ldots \cdot t_n$ (which have the shape of $\overline{e}$ and hence that of t), I sometimes write

$$\Pi_{i=1}^{n} t_i\ .$$

Note that e also has the shape of t; an elementary term is a string term with length 1 and vice versa.

2.2 Operational Semantics

I define small-step operational semantics of my extended LC using a reduction relation $\longrightarrow$. Using this relation, terms t are reduced to values, which can be elementary (u) or strings of elementary values (v):

$$v ::= \overline{u}$$
$$u ::= \lambda x\,.\,t$$

According to this grammar, ϵ is a (non-elementary) value. Note that the fact that a term is elementary does not mean that it reduces to an elementary value; specifically, $t_2\ t_1$ may reduce to a non-elementary value. Constraining terms so that they are guaranteed to reduce to elementary values is achieved by a *number system*, to be introduced in Section 3.

The rules defining the reduction relation are the following:

$$\text{[R-Str]}\ \frac{v \cdot t \neq \epsilon \qquad e \longrightarrow t'}{v \cdot e \cdot t \longrightarrow v \cdot t' \cdot t} \qquad \text{[R-App1]}\ \frac{t_2 \longrightarrow t_2'}{t_2\ t_1 \longrightarrow t_2'\ t_1} \qquad \text{[R-App2]}\ \frac{t \longrightarrow t'}{v\ t \longrightarrow v\ t'}$$

$$\text{[R-AppS]}\ (\Pi_{i=1}^{n} \lambda x\,.\,t_i)\ (\Pi_{j=1}^{m} u_j) \longrightarrow \Pi_{i=1}^{n}\Pi_{j=1}^{m} [u_j/x]t_i$$

$$\text{[R-AppP]}\ (\Pi_{i=1}^{n} \lambda x\,.\,t_i)\ v \longrightarrow \Pi_{i=1}^{n} [v/x]t_i$$

As usual, I write $\xrightarrow{*}$ for the transitive closure of $\longrightarrow$. The following is of note:

- R-Str, R-App1 and R-App2 are congruence rules [3]: R-Str transforms strings having lengths greater than 1 to value strings, while R-App1 and R-App2 transform (elementary) applications to (elementary) applications comprised of value strings. Note that R-App2 means that I rely on call-by-value.

[3] Note that $\cdot$ is not an operator of the object language here; rather, it may be read as the concatenation operator of the metalanguage (and a such is a sibling of substitution).

- For reducing a term $v_2\ v_1$ (the application of a value string to a value string), the choice between the computation rules R-AppS and R-AppP is ambiguous. For instance, $(\lambda x\,.\,x)\ u \longrightarrow u$ both by R-AppS and by R-AppP. This ambiguity will be resolved below (via numbering of terms). Note here that R-AppP substitutes a string for x, whereas R-AppS substitutes each element of the string separately. For instance,
  - using R-AppP, $(\lambda x\,.\,t_1)\cdot(\lambda x\,.\,t_2)\ u_1\cdot u_2 \longrightarrow [u_1\cdot u_2/x]t_1\cdot[u_1\cdot u_2/x]t_2$, while
  - using R-AppS, $(\lambda x\,.\,t_1)\cdot(\lambda x\,.\,t_2)\ u_1\cdot u_2 \longrightarrow [u_1/x]t_1\cdot[u_2/x]t_1\cdot[u_1/x]t_2\cdot[u_2/x]t_2$.

  This difference, which parallels that between the the holistic application of sum and the distributive application of $\oplus$ from Section 1 and which, in the context of nondeterminism, has been described as that between *plural* and *singular semantics* [4, 8], is perhaps most prominent when applying a function to ϵ: under singular semantics (R-AppS), application is strict with respect to ϵ (using R-AppS, $(\lambda x\,.\,t)\ \epsilon \longrightarrow \epsilon$ for every term t), whereas under plural semantics (R-AppP), it is not generally (using R-AppP, $(\lambda x\,.\,t)\ \epsilon \longrightarrow [\epsilon/x]t$).
- As an immediate consequence of the above, using R-AppP, a function application reduces to an elementary value if $n = 1$ (i.e., there is only one function to be applied) and if $[v/x]t_1$ reduces to an elementary value; using R-AppS, the same result additionally requires that $m = 1$, i.e., that the argument of the application is elementary, too. This will be reflected in the numbering of terms as introduced next.

3 Numbering of Terms

Rather than the typing that leads to the simply typed lambda calculus (STLC), I introduce numbering to abstract from the results of computations, and to specify the well-formedness of terms.

3.1 Numbers

Rather than types, I introduce *numbers* η as abstractions of the values terms t reduce to. Number has two possible instances: !, meaning "length 1" (an elementary value; also referred to as "exactly one"), and $*$, meaning "any length" ("any number"). To express that the latter includes the former, I let $! < *$ and define max on numbers η accordingly: $\max(\eta_1, \ldots, \eta_n)$ equals $*$ if $\eta_i = *$ for any $1 \leq i \leq n$, and equals ! otherwise. Numbers are thus *numeric* abstractions: they do not abstract from the kind of a value as types do (e.g., integer or boolean), but from its length. For a term t having number !, one may expect that the value it reduces to has length 1 (i.e., is elementary); for a term having number $*$, one may expect that it reduces to a value of any length[4].

With numbers in mind, one can easily observe that using R-AppS, the variable x is always replaced (via substitution) with a value having number ! (an elementary value), whereas using R-AppP, x is replaced with a value having number $*$ (a value string). Making this explicit by annotating the variables x of functions $\lambda x\,.\,t$ with either

- !, to express that only elementary values will be substituted for x, or with
- $*$, to express that values of arbitrary length may be substituted for x,

disambiguates between the rules R-AppS and R-AppP in the reduction of function application: $\lambda x!\,.\,t\ v$ calls for R-AppS while $\lambda x*\,.\,t\ v$ calls for R-AppP. Tying the disambiguation to the syntax of functions rather than that of applications acknowledges that in the context of multiple arguments (plurals), we may want to distinguish between distributive and holistic

[4] Note how this is more than expecting nothing: one may still expect that it reduces to a value!

treatment, and that this distinction is tied to the function (or operator; e.g., $\oplus$ vs. sum; see Section 1), not the application. Note that the same distinction could be achieved by introducing a second abstraction symbol (such as Λ, complementing λ) for functions and inferring the number of each variable x from the symbol introducing it; however, my choice of using number annotations (which will be needed anyway, as shown next) lets my extension of the LC appear gentler.

3.2 Mapping Constraints and Number Specifiers

It turns out that simple number annotations on variables are not sufficient to infer the number of the value a term will reduce to: Since functions are values that can be substituted for the parameters (variables) of other functions, in whose bodies their application may determine the result of applying the host function, we need to annotate each function (and the variables for which it may be substituted) with the number of its parameter and that of the term constituting its body. For this, I introduce *number specifiers* π defined by the grammar

$$\pi ::= \eta\langle\mu\rangle \qquad \eta ::= ! \mid * \qquad \mu ::= \square \mid \mu\overset{\eta\,\eta}{\rightarrow}\mu$$

where I refer to μ as a *mapping constraint* (named after the mapping constraints 1:1 and 1:N from relational database theory). Mapping constraints are defined recursively to account for higher-order functions; $\square$ terminates the recursion, thereby playing the role of a base type in the STLC. Like base types, $\square$ demands the existence of values that are not functions, that is, of constants c: the number specifier $*\langle\square\rangle$ is thus to be read as "any number of constants", $!\langle\square\overset{!\,*}{\rightarrow}\square\rangle$ as "one function with singular semantics, mapping one constant to any number of constants", and $*\langle\square\overset{*\,!}{\rightarrow}\square\rangle$ as "any number of functions with plural semantics, each mapping any number of constants to one constant".

Given number specifiers and constants, I extend the syntax of my extended LC further (changes highlighted):

$$t ::= \overline{e} \qquad e ::= x \mid \lambda x\pi\,.\,t \mid t\ t \mid c$$
$$v ::= \overline{u} \qquad u ::= \lambda x\pi\,.\,t \mid c$$

The rules governing reduction of function application are then adapted as follows:

$$[\text{R-AppP}]\ (\Pi_{i=1}^{n}\lambda x{*}\langle\_\rangle\,.\,t_i)\ v \longrightarrow \Pi_{i=1}^{n}[v/x]t_i$$

$$[\text{R-AppS}]\ (\Pi_{i=1}^{n}\lambda x{!}\langle\_\rangle\,.\,t_i)\ (\Pi_{j=1}^{m}u_j) \longrightarrow \Pi_{i=1}^{n}\Pi_{j=1}^{m}[u_j/x]t_i$$

In each rule, the wildcard _ (which is not an element of the object language) in the number specifier $\eta\langle\_\rangle$ of the formal parameter x stands for a mapping constraint that is irrelevant for the (selection of the) rule. The number η on the other hand chooses between singular and plural semantics: if for all functions in a string of functions applied to a value, $\eta = !$, R-AppS applies; if for all functions, $\eta = *$, R-AppP applies. For instance, reduction of $(\lambda x_1{*}\langle\square\rangle\,.\,\lambda x_2{!}\langle\square\rangle\,.\,x_1{\cdot}x_2)\ u_1{\cdot}u_2\ u_3{\cdot}u_4$ must go first through R-AppP, yielding $(\lambda x_2{!}\langle\square\rangle\,.\,u_1{\cdot}u_2{\cdot}x_2)\ u_3{\cdot}u_4$, and then through R-AppS, yielding $u_1{\cdot}u_2{\cdot}u_3{\cdot}u_1{\cdot}u_2{\cdot}u_4$.

3.3 The Numbering Relation

Analogous to the typing relation of the STLC [3], I introduce a *numbering relation* as a ternary relation on *number environments* Υ, terms t, and number specifiers π. I write $\Upsilon \vdash t\,\#\,\pi$ for an element of this relation and $\vdash t\,\#\,\pi$ if Υ is empty. Here (and analogously to

type environments), a number environment Υ is a mapping from variable names x to number specifiers π; I write $\Upsilon, x \mapsto \pi$ for Υ extended with the pair $x \mapsto \pi$.

Membership of $\Upsilon \vdash t \# \pi$ in the numbering relation is derived by the following *numbering rules* (in which $\_ \to \_ ; \_$ is the conditional operator):

$$[\text{N-Str}]\ \frac{n \neq 1 \qquad \left(\Upsilon \vdash e_i \# \_\langle\mu\rangle\right)_{i=1}^{n}}{\Upsilon \vdash e_1 \cdot \ldots \cdot e_n \# *\langle\mu\rangle} \qquad [\text{N-Var}]\ \frac{\Upsilon(x) = \eta\langle\mu\rangle}{\Upsilon \vdash x \# \eta\langle\mu\rangle}$$

$$[\text{N-Fun}]\ \frac{\Upsilon, x \mapsto \eta\langle\mu\rangle \vdash t \# \eta'\langle\mu'\rangle}{\Upsilon \vdash \lambda x \eta\langle\mu\rangle \,.\, t \# !\langle\mu \overset{\eta\,\eta'}{\to} \mu'\rangle} \qquad [\text{N-App}]\ \frac{\begin{array}{c}\Upsilon \vdash t_1 \# \eta_1\langle\mu_1\rangle \qquad \Upsilon \vdash t_2 \# \eta_2\langle\mu_1 \overset{\eta_0\,\eta_3}{\to} \mu_2\rangle \\ \eta = \max((\eta_0 = * \to ! \,;\, \eta_1), \eta_2, \eta_3)\end{array}}{\Upsilon \vdash t_2\ t_1 \# \eta\langle\mu_2\rangle}$$

$$[\text{N-Cns}]\ \Upsilon \vdash c \# !\langle\Box\rangle \qquad [\text{N-Sub}]\ \frac{\Upsilon \vdash t \# !\langle\mu\rangle}{\Upsilon \vdash t \# *\langle\mu\rangle}$$

For instance, for $concat = \lambda x_1 *\langle\Box\rangle \,.\, \lambda x_2 *\langle\Box\rangle \,.\, x_1 \cdot x_2$, we have

$$\vdash concat \# !\langle\Box \overset{*\,!}{\to} \Box \overset{*\,*}{\to} \Box\rangle \qquad \vdash concat\ c_1 \cdot c_2 \# !\langle\Box \overset{*\,*}{\to} \Box\rangle \qquad \vdash concat\ c_1 \cdot c_2\ c_3 \cdot c_4 \# *\langle\Box\rangle$$

and indeed, $concat\ c_1 \cdot c_2\ c_3 \cdot c_4 \longrightarrow (\lambda x_2 *\langle\Box\rangle \,.\, c_1 \cdot c_2 \cdot x_2)\ c_3 \cdot c_4 \longrightarrow c_1 \cdot c_2 \cdot c_3 \cdot c_4$.

If $\vdash t \# \pi$ for some π, I say that t is *well-numbered*. The following is of note:

- By N-Str, ϵ has number $*$; however, its mapping constraint μ remains unspecified (ϵ is polymorphic in a sense).
- In combination with N-Fun, N-Str makes sure that in a string of functions with length greater 1, all formal parameters x have the same number η (either ! or $*$).
- N-App makes the number of the formal parameter x of the applied function(s), η_0, decide (via the choice $\eta_0 = * \to ! \,;\, \eta_1$) whether the number of the argument t_1, η_1, of an application $t_2\ t_1$ affects the number of the result of the application: if $\eta_0 = *$, the reduction must be through R-AppP, which means that the number of the argument is insignificant. For instance, $\vdash (\lambda x *\langle\_\rangle . c)\ v \# !\langle\Box\rangle$, independently of the number of v.
- Nothing in N-App enforces that the number of the argument, η_1, and the number of the formal parameter, η_0, match. For $\eta_0 = !$, this is rendered unnecessary by R-AppS, which substitutes only elementary values for x (hence my choice of call-by-value); for $\eta_0 = *$, $\eta_1 = !$ is actually acceptable: an elementary value may always be substituted for a variable constrained to hold any number (but note how $\eta_1 = !$ does not propagate through a function: e.g., $\vdash (\lambda x *\langle\Box\rangle \,.\, x)\ c \# *\langle\Box\rangle$).
- N-Sub makes the number system polymorphic: all terms having number ! also have number $*$. For number judgements $\Upsilon \vdash t : !\langle\mu \overset{\eta\,!}{\to} \mu'\rangle$ (derived through N-Fun, i.e., for functions), this means that we also have $\Upsilon \vdash t : !\langle\mu \overset{\eta\,*}{\to} \mu'\rangle$ and $\Upsilon \vdash t : *\langle\mu \overset{\eta\,*}{\to} \mu'\rangle$: functions "to one" are subsumed by functions "to any". This gives us well-numbered heterogeneous function strings, i.e., strings whose elementary functions' mapping constraints vary between $\mu \overset{\eta\,!}{\to} \mu'$ and $\mu \overset{\eta\,*}{\to} \mu'$.
- Last but not least, the number system supports the labelling of well-numbered terms $\lambda x !\langle\mu\rangle \,.\, t$ as *total functions* or *relations*: if $\vdash\ \lambda x !\langle\mu\rangle \,.\, t \# !\langle\mu \overset{!\,!}{\to} \_\rangle$, then we may call $\lambda x !\langle\mu\rangle \,.\, t$ a *total function* (because it maps an elementary value to an elementary value); if $\vdash \lambda x !\langle\mu\rangle \,.\, t \# !\langle\mu \overset{!\,*}{\to} \_\rangle$, then we may think of $\lambda x !\langle\mu\rangle \,.\, t$ as a *relation* (because it may map an elementary value to any number of elementary values[5]). Note that by this definition and by the polymorphism introduced through N-Sub, all total functions are also relations.

[5] including the same value more than once, giving us a multi-relation; also, unlike for set-theoretic relations, the values are ordered (but note that both can be had in some relational database systems, and may indeed be desirable in certain domains)

3.4 Number Safety

As for typing, we want to be sure not only that the well-numberedness of a term guarantees that it can be reduced to a value, but also that the term's derived number correctly abstracts from the length of that value. This is expressed by the following theorem:

▶ **Theorem 1** (Number Safety). *If for a term t and some π, $\vdash t \# \pi$ and $t \xrightarrow{*} t'$ and there is no t'' so that $t' \longrightarrow t''$, then t' is a value and $\vdash t' \# \pi$.*

Proof. The proof, which follows a standard layout (see, e.g., [3]), follows immediately from proofs of progress and preservation, which are found in a companion report [7]. ◀

4 Supporting the Relations Language

While the operational semantics of the relations language relies on (flat) bags for representing multiple values (see Section 1 and also [1, 2]), the SNLC builds on strings, which are inherently ordered. This difference, which is owing to the syntactic nature of the LC (and the fact that syntax is not commutative), affects the equality of multiples, which I did not cover[6]. Leaving this fundamental difference aside, the SNLC provides a broad basis for Harkes and Visser's relations language.

4.1 Computing with Multiples

The SNLC's choice of singular and plural semantics of function application supports both distributing operations over and aggregations of multiple operands, as required by addition and summing of the relations language. To see this, assume that elementary integer addition, $+$, is a primitive of the SNLC whose use is numbered by the rule

$$[\text{N-ADD}]\ \frac{\Upsilon \vdash t_1 \# !\langle\Box\rangle \qquad \Upsilon \vdash t_2 \# !\langle\Box\rangle}{\Upsilon \vdash t_1 + t_2 \# !\langle\Box\rangle}\ .$$

Distributive Application. We can then define

$$\oplus \;=\; \lambda x_1 !\langle\Box\rangle\,.\,\lambda x_2 !\langle\Box\rangle\,.\,x_1 + x_2 \qquad (\text{with } \vdash \oplus \# !\langle\Box \xrightarrow{!!} \Box \xrightarrow{!!} \Box\rangle)$$

as the distribution of elementary addition over strings, giving us (in infix notation) $1{\cdot}2 \oplus 3{\cdot}4 \xrightarrow{*} 4{\cdot}5{\cdot}5{\cdot}6$, which corresponds to $\{\!|\, x_1 + x_2 \mid x_1 \in \{\!|\, 1, 2 \,|\!\}, x_2 \in \{\!|\, 3, 4 \,|\!\} \,|\!\}$, the result of the same addition in the the relations language (see Section 1 and [1, 2]). At the same time, above definition of $\oplus$ (in concert with R-APPS on which the reduction of its application relies) gracefully handles the absence of numbers in the style of an `Optional` type: e.g., $1 \oplus \epsilon \xrightarrow{*} \epsilon$ (the strictness of R-APPS on ϵ).

Aggregation. As noted in Section 1, aggregation cannot be defined distributively, but requires holistic treatment of multiples. Rather than offloading this treatment entirely to a semantic domain (as the definition of the relations language did for its aggregation functions), we can implement aggregation – with the help of (explicit) state – using a combination of

[6] But note how this problem parallels that of the equality of lambda terms, which is subject to α-equivalence.

functions with plural and singular semantics, where the singular semantics does the necessary looping. For instance, replacing variable substitution with a mutable variables store and assuming variable assignment, we can define

$$\begin{aligned}\text{sum} \;&= (\lambda x_0!\langle\Box\rangle\,.\,\lambda x_1{*}\langle\Box\rangle\,.\,(\lambda x_2{*}\langle\Box\rangle\,.\,x_0)\;((\lambda x_3!\langle\Box\rangle\,.\,x_0 := x_0 + x_3)\;x_1))\;0\\ &\quad\;(\text{with} \vdash \text{sum}\,\#\,!\langle\Box \xrightarrow{*}{}^{!}\Box\rangle)\end{aligned}$$

where applying $\lambda x_3!\langle\Box\rangle\,.\,x_0 := x_0{+}x_3$ (singular semantics) to x_1 lets x_3 loop over the elements of x_1 (whose number specifier is $*\langle\Box\rangle$) and which gives us, for instance, $\text{sum}\,1{\cdot}2 \xrightarrow{*} 3$. To implement aggregation without resorting to state, we would need to introduce recursion and string deconstruction to the SNLC, which would let strings appear as built-in lists (but see Section 5 for why they are not).

4.2 Relations and their Navigation

The relations language caters for the declaration and navigation of n-ary, non-updatable relations. In the SNLC, I model these relations as extensionally specified (tabular) functions, and their navigation as function application.

Binary Relations. To support the representation and navigation of relations as sets of pairs (rather than computable functions, or λ-abstractions), I extend the SNLC with a new form of terms,

$$\text{case } t \text{ of } \overline{u : u}\ .$$

Here, $\overline{u : u}$ is a string of pairs of elementary values that can be viewed as an extensionally defined binary relation (or a two-column table), and t is a term that selects from the relation a string of values, namely the string of right members of pairs whose left members are matched by t. This behaviour is accomplished by the rules

$$[\text{R-Case1}]\ \frac{t \longrightarrow t'}{\text{case } t \text{ of } \overline{u : u'} \longrightarrow \text{case } t' \text{ of } \overline{u : u'}}$$

$$[\text{R-Case2}]\ \frac{\big(v_i = (u = u_i \to u_i'\,;\epsilon)\big)_{i=1}^{n}}{\text{case } u \text{ of } \Pi_{i=1}^{n}(u_i : u_i') \longrightarrow \Pi_{i=1}^{n} v_i}$$

reducing a case expression to a string $v_1 \cdot \ldots \cdot v_n$, of which each v_i is either u_i' or ϵ, depending on whether the left member u_i of a pair $u_i : u_i'$ equals the selector value u. This reduction behaviour of case expressions is abstracted by the number rule

$$[\text{N-Case}]\ \frac{\Upsilon \vdash t\,\#\,!\langle\mu'\rangle \qquad \big(\Upsilon \vdash u_i\,\#\,!\langle\mu'\rangle\big)_{i=1}^{n} \qquad \big(\Upsilon \vdash u_i'\,\#\,!\langle\mu\rangle\big)_{i=1}^{n}}{\Upsilon \vdash \text{case } t \text{ of } \Pi_{i=1}^{n}(u_i : u_i')\,\#\,{*}\langle\mu\rangle}\ ,$$

expressing that

- the selector term t needs to reduce to precisely one value whose associated mapping constraint μ' must match the mapping constraints of all first places of the pairs, and that
- the case expression reduces to a string of arbitrary length, whose elements all share the same mapping constraint μ (enforced by the third condition of the rule).

Navigation. Given case expressions and their reduction rules, the relations that they capture can be navigated by abstracting from the selector term and applying the resulting abstraction (function) to the source of the navigation. For instance, for

$$\mathit{children} \;=\; \lambda x!\langle\Box\rangle\,.\,\text{case } x \text{ of } (c_1\colon c_2)\cdot(c_3\colon c_4)\cdot(c_3\colon c_5) \qquad (\text{with } \vdash \mathit{children} \,\#\, !\langle\Box \xrightarrow{!*} \Box\rangle)$$

we get

$$\mathit{children}\ c_1 \xrightarrow{*} c_2 \qquad \mathit{children}\ c_2 \xrightarrow{*} \epsilon$$
$$\mathit{children}\ c_3 \xrightarrow{*} c_4\cdot c_5 \qquad \mathit{children}\ c_1\cdot c_2\cdot c_3 \xrightarrow{*} c_2\cdot c_4\cdot c_5\ .$$

Note how this corresponds to a navigation expression `x.children` with `x` substituted accordingly; in fact, if *children* is interpreted as the relation (or mapping) $\{(c_1, c_2), (c_3, c_4), (c_3, c_5)\}$, then its application to, say, $c_1\cdot c_2\cdot c_3$ can be interpreted as the direct image of the set $\{c_1, c_2, c_3\}$ under that relation. Also, relations can be navigated transitively: with

$$\mathit{children} \;=\; \lambda x!\langle\Box\rangle\,.\,\text{case } x \text{ of } (c_1\colon c_2)\cdot(c_2\colon c_3)\ ,$$

we get

$$\mathit{children}\ (\mathit{children}\ c_1) \xrightarrow{*} c_3\ ,$$

corresponding to the navigation expression `c1.children.children`.

Bidirectional Navigation. The encoding of relations using abstractions over case expressions is directed (the relations of the SNLC are mappings). In order to change the direction of navigation of a relation, or to make relations bidirectional as in the relations language, one needs to define relations in pairs, one per direction. Since these pairs are symmetric (one is the permutation of the other), it is straightforward to generate them using suitable preprocessing of SNLC programs.

Attributes. The relations language has not only relations relating objects, but also attributes describing them. In the SNLC, these attributes are modelled as special relations: if constants c are divided into objects, o, and attribute values, a, (integers, booleans, etc.), then abstractions *attr* of the form $\lambda x!\langle\Box\rangle\,.\,\text{case } x \text{ of } \overline{o\colon a}$ associate objects with their attribute values (one abstraction *attr* per attribute). For instance, given a definition of the attribute *age*, we can write

$$\mathit{age}\ (\mathit{children}\ x)\ ,$$

corresponding to the expression `x.children.age` from Section 1.

Higher-Degree Relations. The relations language also caters for ternary and higher-degree relations. In the SNLC, one can model such relations by nesting (abstracted) case expressions: for instance,

$$R \;=\; \lambda x_1!\langle\Box\rangle\,.\,\text{case } x_1 \text{ of } c_1\colon \big(\lambda x_2!\langle\Box\rangle\,.\,\text{case } x_2 \text{ of } (c_2\colon c_3)\cdot(c_2\colon c_4)\big)$$

corresponds to a ternary relation $\{(c_1, c_2, c_3), (c_1, c_2, c_4)\}$, which is navigated by applying R to two values in a row: for instance, $R\ c_1\ c_2 \xrightarrow{*} c_3\cdot c_4$.

Higher-Order Relations. Given that case expressions correspond to relations, nested case expressions like those of R above correspond to nested relations, or relations of relations, which are by definition higher-order.

We can also define higher-order relations computationally. For instance, the definition

$$hop_2 \;=\; \lambda x_1!\langle\square\xrightarrow{!*}\square\rangle\,.\,\lambda x_2!\langle\square\rangle\,.\,x_1\;(x_1\;x_2)$$

lets us rephrase the above term *children* (*children* c_1) as hop_2 *children* c_1. Assuming a second relation *parents*, we can derive both the grandchildren and the grandparents of c_1 using hop_2 *children*·*parents* c_1 (which reduces to (*children* (*children* c_1))·(*parents* (*parents* c_1))). If we wanted to navigate to siblings (children of parents) and spouses (parents of children) of c_1 as well, we would need to switch hop_2 to plural semantics (so that hop_2 *children*·*parents* c_1 reduced to *children*·*parents* (*children*·*parents* c_1)).

hop_2 composes a relation with itself. More generally, one might want to define relation composition (or relative multiplication) as a higher-order relation

$$compose \;=\; \lambda x_1!\langle\square\xrightarrow{!*}\square\rangle\,.\,\lambda x_2!\langle\square\xrightarrow{!*}\square\rangle\,.\,\lambda x_3!\langle\square\rangle\,.\,x_1\;(x_2\;x_3)\;,$$

but for *compose* to be fully general (i.e., applicable to arbitrarily numbered relations), one would need to adopt parametric number specifiers (in analogy to parametric types).

4.3 More Multiplicities

The relations language features not two, but four different numbers (there called multiplicities): besides ! and $*$, which are also covered by the SNLC as presented here, it offers ? (for "none or one") and + (for "one or more"). While integrating these in the SNLC will require some extra effort (the proofs will require significantly more case analyses), this effort may be well-spent: rather than the number η in the derived number specifier of $\lambda x!\langle\mu\rangle\,.\,t$, $!\langle\mu\xrightarrow{!\eta}\_\rangle$, distinguishing between total functions ($\eta = !$) and relations ($\eta = *$), it qualifies $\lambda x!\langle\mu\rangle\,.\,t$ as a relation with properties given by the table

η	left-total	right-unique	classification
?	no	yes	partial function
!	yes	yes	total function
+	yes	no	relation
$*$	no	no	relation

Furthermore, restricting numbers η to ! and ?, we arrive at a calculus that can distinguish and handle total and partial functions without needing an `Optional` type (and a type system supporting it): for instance, an implementation of subtraction ($\ominus$) may have number specifier $!\langle\square\xrightarrow{!!}\square\xrightarrow{!?}\square\rangle$, indicating that applying it to two arguments evaluates to nothing (ϵ) if the subtrahend is greater than the minuend. The expression $1 \ominus 2 \oplus 3$ would then evaluate to ϵ, without the definition of $\oplus$ needing to take extra measures for this (it is automatically strict on ϵ; see Section 4.1).

5 Discussion

One might hold against the SNLC that it makes strings a primitive language construct, where the plain LC is already expressive enough to cover strings, by giving them the form of lists. However, like sets and unlike strings, lists have a deep structure (one can have lists of lists) and indeed, applying a function to a string is not the same as mapping the function over the corresponding list: for instance $(\lambda x!\langle\square\rangle\,.\,\epsilon)\;c_1{\cdot}c_2$ reduces to ϵ, a string of length 0, and not to a string of length 2, as mapping would do. It appears that at the very least, the flattening

that is implicit in the concatenation of strings would need to be added to function application somehow. With this in place, however, and assuming a typing discipline without coercions, one would need to encode everything as a list, if only because the type of a list differs from that of its elements so that one element cannot stand in for any number, as suggested by N-Sub and realized by substitution in the SNLC. On the other hand, the SNLC's lack of both recursion and deconstruction of strings (analogous to the deconstruction of lists into their heads and tails) lets aggregation require explicit state, unlike the fold on lists that has granted functional programming much of its popularity. A more practical language based on the SNLC will therefore likely feature both, strings and lists (but mind that, being a heir to the LC, all that the SNLC is lacking to accommodate lists is recursion).

In several extensions of first-order languages with numbers (or multiplicities), type and number have been observed to be orthogonal (see, e.g., [1, 2, 5, 6]). By adding higher-order functions, however, it becomes apparent that the number annotations must have a form that is parallel to that of types: they must introduce "function numbers" $\overset{\eta\,\eta'}{\rightharpoonup}$ (here called mapping constraints) as analogues of function types. To terminate the recursion of mapping constraints, a single "unary mapping constraint", or "mapping constraint of a constant" must be introduced (here $\square$). While this is much like the one base type that is minimally required by any STLC to be usable [3], in the SNLC, there is actually no use in having more than one such base. Therefore, the terminal $\square$ can be replaced by syntax for (arbitrarily many) base types, which equips us for defining a "simply *typed and numbered* lambda calculus" as a straightforward merger of the STLC and the SNLC as here presented.

6 Conclusion

By adding strings of terms, I have extended the lambda calculus to a higher-order relational language in which an elementary function may map an elementary value to zero, one, or more elementary values. Rather than typing this language, I have numbered it, and have shown that this gives us guarantees analogous to those of typing, with the edge that the number system can distinguish between total functions and relations (ordered multi-relations, to be precise). It turns out that the SNLC serves existing, more complex query languages; specifically, it serves as a foundation of the relations language of Daco Harkes and the late Eelco Visser.

References

1 Daco Harkes and Eelco Visser. Unifying and generalizing relations in role-based data modeling and navigation. In Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju, editors, *Software Language Engineering - 7th International Conference, SLE 2014, Västerås, Sweden, September 15-16, 2014. Proceedings*, volume 8706 of *Lecture Notes in Computer Science*, pages 241–260. Springer, 2014. doi:10.1007/978-3-319-11245-9_14.

2 Daniël Corstiaan Harkes. *Declarative Specification of Information System Data Models and Business Logic*. PhD thesis, Delft University of Technology, Netherlands, 2019. doi:10.4233/uuid:5e9805ca-95d0-451e-a8f0-55decb26c94a.

3 Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

4 Harald Søndergaard and Peter Sestoft. Non-determinism in functional languages. *Comput. J.*, 35(5):514–523, 1992. doi:10.1093/comjnl/35.5.514.

5 Friedrich Steimann. None, one, many – What's the difference, anyhow? In Thomas Ball, Rastislav Bodík, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett, editors, *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA*, volume 32 of *LIPIcs*, pages 294–308. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015. doi:10.4230/LIPIcs.SNAPL.2015.294.

6 Friedrich Steimann. Containerless plurals: Separating number from type in object-oriented programming. *ACM Trans. Program. Lang. Syst.*, 44(4), September 2022. doi:10.1145/3527635.

7 Friedrich Steimann. A simply numbered lambda calculus with number safety proof. Report, Lehrgebiet Programmiersysteme, Fernuniversität in Hagen, Sept 2022. URL: https://feu.de/ps/pubs/SNLC-TR.pdf.

8 Friedrich Steimann and Marius Freitag. The semantics of plurals. In Bernd Fischer, Lola Burgueño, and Walter Cazzola, editors, *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2022, Auckland, New Zealand, December 6-7, 2022*, pages 36–54. ACM, 2022. doi:10.1145/3567512.3567516.