Logic Operators and Quantifiers in Type-Theory of Algorithms

Roussanka Loukanova(✉)

Institute of Mathematics and Informatics, Bulgarian Academy of Sciences, Acad. G. Bonchev Street, Block 8, 1113 Sofia, Bulgaria
rloukanova@gmail.com

Abstract. In this work, I introduce the Type-Theory of Algorithms (TTA), which is an extension of Moschovakis Type-Theory of Algorithms and its reduction calculus, by adding logic operators and quantifiers. The formal language has two kinds of terms of formulae, for designating state-independent and state-dependent propositions and predications. The logic operators include conjunction, disjunction, conditional implication, and negation. I add state-dependent quantifiers, for enhancing the standard quantifiers of predicate logic. I provide an extended reduction calculus of the Type-Theory of Acyclic Algorithms, for reductions of terms to their canonical forms. The canonical forms of the terms provide the algorithmic semantics for computing the denotations.

Keywords: recursion · type-theory · acyclic algorithms · denotational semantics · algorithmic semantics · reduction calculus · logic operators · quantifiers

1 Introduction

This paper is part of the author's work on development of a new type-theory of the mathematical notion of algorithms, its concepts, and potentials for applications to advanced, computational technologies, with a focus on Computational Semantics and Syntax-Semantics-Semantics Interfaces for formal and natural languages.

For the initiation of this approach to mathematics of algorithms, see the original work on the formal languages of recursion (FLR) by Moschovakis [15–17]. The formal languages of recursion FLR are untyped systems. The typed version of this approach to algorithmic, acyclic computations was introduced, for the first time, by Moschovakis [18], with the type theory $\mathrm{L}^{\lambda}_{\mathrm{ar}}$. Type theory L^{λ}_{r} covers full recursion and is an extension of type theory of acyclic recursion $\mathrm{L}^{\lambda}_{\mathrm{ar}}$.

For more recent developments of the language and theory of acyclic algorithms $\mathrm{L}^{\lambda}_{\mathrm{ar}}$, see, e.g., [6–8]. The work in [11] presents an algorithmic η-rule with the induced η-reduction acting on canonical terms in $\mathrm{L}^{\lambda}_{\mathrm{ar}}$, as a special case of (γ^{*}). The algorithmic expressiveness of $\mathrm{L}^{\lambda}_{\mathrm{ar}}$ has been demonstrated by its applications to computational semantics of natural language. Algorithmic semantics of quantifier scope ambiguities and underspecification is presented in [3]. Computational grammar of natural language that coveres syntax-semantics interfaces

https://doi.org/10.1007/978-3-031-43977-3_11

is presented in [5]. The work in [8] is on fundamental notions of algorithmic binding of argument slots of relations and functions, across assignments in recursion terms. It models functional capacities of neural receptors for neuroscience of language. A generalised restrictor operator is introduced in $\mathrm{L}_{\mathrm{ar}}^{\lambda}$ for restricted, parametric algorithms, e.g., in semantics of definite descriptors, by [9], which is extended in a forthcoming publication. Currying order and limited, restricted algorithmic β-conversion in $\mathrm{L}_{\mathrm{ar}}^{\lambda}$ are presented by [10].

In this paper, I extend the formal language, reduction calculus, and semantics of $\mathrm{L}_{\mathrm{ar}}^{\lambda}$ and L_{r}^{λ}, by adding logic operators and logic quantifiers, with two versions of truth values: pure truth values and state-dependent ones. I introduce the logic operators of conjunction, disjunction, implication, and negation in the formal languages of $\mathrm{L}_{\mathrm{ar}}^{\lambda}$ and L_{r}^{λ} by categorematic, logic constants, which have the benefits of sharing various properties and reduction rules with non-logic constants, while maintaining their logical characteristics.

In Sect. 2, I introduce the extended type-theory $\mathrm{L}_{\mathrm{ar}}^{\lambda}$ and L_{r}^{λ} of acyclic algorithms, by its syntax and denotational semantics. The focus of the rest of the paper is on the acyclic type-theory $\mathrm{L}_{\mathrm{ar}}^{\lambda}$. In Sect. 3, I present the extended system of reduction rules and the induced γ^*-reduction calculus of $\mathrm{L}_{\mathrm{ar}}^{\lambda}$. The additional reduction rule (γ^*) greatly reduces the complexity of the terms, without affecting the denotational and algorithmic semantics of $\mathrm{L}_{\mathrm{ar}}^{\lambda}$, in any significant way. I provide the full, formal definition of the congruence relation between terms, which is part of the reduction system of both $\mathrm{L}_{\mathrm{ar}}^{\lambda}$ and L_{r}^{λ}. The reduction calculus of $\mathrm{L}_{\mathrm{ar}}^{\lambda}$ reduces each $\mathrm{L}_{\mathrm{ar}}^{\lambda}$ term to its canonical form. For every term A, its canonical form is unique modulo congruence. The canonical form of every proper $\mathrm{L}_{\mathrm{ar}}^{\lambda}$ term determines the algorithm for computing its denotation and saving the component values, including functions, in memory slots for reuse. Section 4 is on the algorithmic expressiveness of $\mathrm{L}_{\mathrm{ar}}^{\lambda}$. Theorem 2 proves that $\mathrm{L}_{\mathrm{ar}}^{\lambda}$ is a proper extension of Gallin TY_2, see Gallin [1]. There are $\mathrm{L}_{\mathrm{ar}}^{\lambda}$ recursion terms that are not algorithmically equivalent to any explicit, λ-calculus, i.e., TY_2 terms. In addition, such $\mathrm{L}_{\mathrm{ar}}^{\lambda}$ recursion terms, provide subtle semantic distinctions for expressions of natural language. The focus of Sect. 5 is on the semantic and algorithmic distinctions between coordinated predication and sentential conjunction. In Sect. 6, I overview some relations between let-expressions for λ-calculus and $\mathrm{L}_{\mathrm{ar}}^{\lambda}$ recursion terms. I give an explanation why the $\mathrm{L}_{\mathrm{ar}}^{\lambda}$ recursion terms are not algorithmically equivalent to λ-terms in $\mathrm{L}_{\mathrm{ar}}^{\lambda}$ representing let-expressions. I demonstrate the extended reduction calculus with reductions of terms to their canonical forms, which offer distinctive, algorithmic semantics of natural language expressions.

2 Introduction to Type-Theory of Acyclic Algorithms

Type-theory of algorithms (TTA), in each of its variants of full and acyclic recursion, L_{r}^{λ} and $\mathrm{L}_{\mathrm{ar}}^{\lambda}$, respectively, is a computational system, which extends the standart, simply-typed λ-calculus in its syntax and semantics.

The basis for the formal languages of L_{r}^{λ} and $\mathrm{L}_{\mathrm{ar}}^{\lambda}$, and their denotational and algorithmic semantics is a tuple $\mathrm{B}_{\mathrm{r}}^{\lambda} = \langle \mathsf{TypeR}, \mathsf{K}, \mathsf{Vars}, \mathsf{TermR}, \mathsf{RedR} \rangle$, where:

(1) TypeR is the set of the rules that defines the set Types
(2) $\mathsf{K} = \mathsf{Consts}$ is a set of constants (2a)
(3) Vars is a set of variables (2f) of two kinds, pure and recursion (2d)–(2e)
(4) TermR is the set of the rules for the terms of L_r^λ and $\mathrm{L}_{\mathrm{ar}}^\lambda$, given in Definition 1
(5) RedR is the set of the reduction rules given in Sect. 3.2

The focus of this work is on the type-theory $\mathrm{L}_{\mathrm{ar}}^\lambda$ of acyclic algorithms (TTAA).

Notation 1. *We shall use the following meta-symbols (1)–(2):*

(1) "$\equiv$" is used for notational abbreviations and definitions, i.e., for literal, syntactic identities between expressions. The equality sign "$=$" is for the identity relation between objects of L_{ar}^λ (L_r^λ)
(2) "$:\equiv$" is for the replacement, i.e., substitution operation, in syntactic constructions, and sometimes for definitional constructions

2.1 Syntax

The set Types of $\mathrm{L}_{\mathrm{ar}}^\lambda$ is defined recursively, e.g., in Backus-Naur Form (BNF):

$$\tau ::= \mathsf{e} \mid \mathsf{t} \mid \mathsf{s} \mid (\tau \to \tau) \qquad (\mathsf{Types})$$

The type e is for basic entities and $\mathrm{L}_{\mathrm{ar}}^\lambda$ terms denoting such entities, e.g., for animals, people, etc., animate or inanimate objects. The type s is for states that carry context information, e.g., possible worlds, time and space locations, speakers, listeners, etc. The denotations of some expressions of natural language, e.g., proper names and other noun phrases (NPs), can be rendered (translated) to $\mathrm{L}_{\mathrm{ar}}^\lambda$ terms of type $(\mathsf{s} \to \mathsf{e})$. The type t is for truth values. For any $\tau_1, \tau_2 \in \mathsf{Types}$, the type $(\tau_1 \to \tau_2)$ is for functions from objects of type τ_1 to objects of type τ_2, and for $\mathrm{L}_{\mathrm{ar}}^\lambda$ terms denoting such functions. We shall use the following abbreviations:

$$\widetilde{\sigma} \equiv (\mathsf{s} \to \sigma), \quad \text{for state-dependent objects of type } \widetilde{\sigma} \qquad (1a)$$

$$\widetilde{\mathsf{e}} \equiv (\mathsf{s} \to \mathsf{e}), \quad \text{for state-dependent entities} \qquad (1b)$$

$$\widetilde{\mathsf{t}} \equiv (\mathsf{s} \to \mathsf{t}), \quad \text{for state-dependent truth values} \qquad (1c)$$

$$(\overrightarrow{\tau} \to \sigma) \equiv (\tau_1 \to \cdots \to (\tau_n \to \sigma)) \in \mathsf{Types}\,(n \geq 1)$$
$$\text{currying coding, for } \sigma, \tau_i \in \mathsf{Types}, \; i = 1, \ldots, n \qquad (1d)$$

Typed Vocabulary of $\mathrm{L}_{\mathrm{ar}}^\lambda$: For every $\sigma \in \mathsf{Types}$, $\mathrm{L}_{\mathrm{ar}}^\lambda$ has denumerable sets of *constants*, and two kinds of infinite, denumerable sets of pure and recursion variables, all in pairwise different sets:

$$\mathsf{K}_\sigma = \mathsf{Consts}_\sigma = \{\mathsf{c}_0^\sigma, \mathsf{c}_1^\sigma, \dots\}; \qquad \mathsf{K} = \mathsf{Consts} = \textstyle\bigcup_{\tau \in \mathsf{Types}} K_\tau \tag{2a}$$

$$\wedge, \vee, \rightarrow \ \in \mathsf{Consts}_{(\tau \to (\tau \to \tau))}, \ \tau \in \{\, \mathsf{t}, \widetilde{\mathsf{t}} \,\} \qquad \text{(logical constants)} \tag{2b}$$

$$\neg \in \mathsf{Consts}_{(\tau \to \tau)}, \ \tau \in \{\, \mathsf{t}, \widetilde{\mathsf{t}} \,\} \qquad \text{(logical constant for negation)} \tag{2c}$$

$$\mathsf{PureV}_\sigma = \{v_0^\sigma, v_1^\sigma, \dots\}; \qquad \mathsf{PureV} = \textstyle\bigcup_{\tau \in \mathsf{Types}} \mathsf{PureV}_\tau \tag{2d}$$

$$\mathsf{RecV}_\sigma = \mathsf{MemoryV}_\sigma = \{p_0^\sigma, p_1^\sigma, \dots\}; \qquad \mathsf{RecV} = \textstyle\bigcup_{\tau \in \mathsf{Types}} \mathsf{RecV}_\tau \tag{2e}$$

$$\mathsf{PureV}_\sigma \cap \mathsf{RecV}_\sigma = \varnothing; \quad \mathsf{Vars}_\sigma = \mathsf{PureV}_\sigma \cup \mathsf{RecV}_\sigma; \quad \mathsf{Vars} = \textstyle\bigcup_{\tau \in \mathsf{Types}} \mathsf{Vars}_\sigma \tag{2f}$$

Pure variables PureV are used for λ-abstraction and quantification. On the other hand, the recursion variables, which are called also *memory variables*, *memory locations (slots, cells)*, or *location variables*, play a special role in algorithmic computations, for saving information. Values, which can be obtained either directly by immediate, variable valuations, or by algorithmic computations, via recursion or iteration, can be saved, i.e., memorised, in typed memory locations, i.e., in memory variables, of the set RecV, by assignments. Sets of assignments can determine mutually recursive or iterative computations.

I shall use mixed notations for type assignments, $A : \tau$ and A^τ, to express that a term A or an object A is of type τ.

In Definition 1, I introduce the logical constants as categorematic constants for conjunction, disjunction, implication, $\wedge, \vee, \rightarrow \ \in \mathsf{Consts}_{(\tau \to (\tau \to \tau))}$, and negation, $\neg \in \mathsf{Consts}_{(\tau \to \tau)}$, in two variants of truth values $\tau \in \{\, \mathsf{t}, \widetilde{\mathsf{t}} \,\}$.

Definition 1. $\mathsf{Terms} = \mathsf{Terms}(L_{ar}^\lambda) = \bigcup_{\tau \in \mathsf{Types}} \mathsf{Terms}_\tau$ *is the set of the terms of L_{ar}^λ, where, for each $\tau \in$ Types, Terms_τ is the set of the terms of type τ, which are defined recursively by the rules* TermR *in* (3a)–(3g)*, in a typed style of Backus-Naur Form (TBNF):*

$$A :\equiv \mathsf{c}^\tau : \tau \mid x^\tau : \tau \qquad (\textit{constants and variables}) \tag{3a}$$

$$\mid B^{(\sigma \to \tau)}(C^\sigma) : \tau \qquad (\textit{application terms}) \tag{3b}$$

$$\mid \lambda(v^\sigma)(B^\tau) : (\sigma \to \tau) \qquad (\lambda\textit{-abstraction terms}) \tag{3c}$$

$$\mid A_0^{\sigma_0} \ \mathsf{where} \ \{\, p_1^{\sigma_1} := A_1^{\sigma_1}, \dots, p_n^{\sigma_n} := A_n^{\sigma_n} \,\} : \sigma_0 \qquad (\textit{recursion terms}) \tag{3d}$$

$$\begin{aligned} &\mid \wedge(A_2^\tau)(A_1^\tau) : \tau \ \mid \ \vee(A_2^\tau)(A_1^\tau) : \tau \ \mid \ \rightarrow (A_2^\tau)(A_1^\tau) : \tau \\ &(\textit{conjunction} \ / \ \textit{disjunction} \ / \ \textit{implication terms}) \end{aligned} \tag{3e}$$

$$\mid \neg(B^\tau) : \tau \qquad (\textit{negation terms}) \tag{3f}$$

$$\mid \forall(v^\sigma)(B^\tau) : \tau \mid \exists(v^\sigma)(B^\tau) : \tau \qquad (\textit{pure, logic quantifier terms}) \tag{3g}$$

given that

(1) $\mathsf{c} \in K_\tau = \mathsf{Consts}_\tau$
(2) $x^\tau \in \mathsf{PureV}_\tau \cup \mathsf{RecV}_\tau$ *is a pure or memory (recursion) variable,*
$v^\sigma \in \mathsf{PureV}_\sigma$ *is a pure variable*
(3) $A_1^\tau, A_2^\tau, B, A_i^{\sigma_i} \in \mathsf{Terms}$ $(i = 0, \dots, n)$ *are terms of the respective types*
(4) In (3d)*, for* $i = 1, \dots, n$, $p_i \in \mathsf{RecV}_{\sigma_i}$ *are pairwise different recursion (memory) variables;* $A_i^{\sigma_i} \in \mathsf{Terms}_{\sigma_i}$ *assigned to* p_i *is of the same corresponding*

type; and the sequence of assignments $\{ p_1^{\sigma_1} := A_1^{\sigma_1}, \ldots, p_n^{\sigma_n} := A_n^{\sigma_n} \}$ *is acyclic, by satisfying the Acyclicity Constraint* (AC) *in Definition 2.*

(5) In (3e)–(3g), $\tau \in \{ \mathsf{t}, \widetilde{\mathsf{t}} \}$ *are for state-independent and state-dependent truth values, respectively*

Definition 2 (Acyclicity Constraint (AC)). *For any* $A_i \in \mathsf{Terms}_{\sigma_i}$ *and pairwise different memory (recursion) variables* $p_i \in \mathsf{RecV}_{\sigma_i}$, $i \in \{ 1, \ldots, n \}$, *the sequence* (4)*:*

$$\{ p_1^{\sigma_1} := A_1^{\sigma_1}, \ldots, p_n^{\sigma_n} := A_n^{\sigma_n} \} \quad (n \geq 0) \tag{4}$$

is an acyclic system of assignments *iff there is a function* rank

$$\begin{aligned} &\mathsf{rank} : \{p_1, \ldots, p_n\} \to \mathbb{N}, \text{ such that, for all } p_i, p_j \in \{p_1, \ldots, p_n\}, \\ &\text{if } p_j \text{ occurs freely in } A_i, \text{ then } \mathsf{rank}(p_j) < \mathsf{rank}(p_i) \end{aligned} \tag{AC}$$

Free and Bound Variables. The sets $\mathsf{FreeVars}(A)$ and $\mathsf{BoundVars}(A)$ of the free and bound variables of every term A are defined by structural induction on A, in the usual way, with the exception of the recursion terms. For the full definition, see [8]. For any given recursion term A of the form (3d), the constant where designates a binding operator, which binds all occurrences of $p_1, \ldots, p_n$ in A:

$$\text{For } A \equiv A_0 \text{ where } \{p_1 := A_1, \ldots, p_n := A_n\} \in \mathsf{Terms} \tag{5a}$$
$$\mathsf{FreeV}(A) = \cup_{i=0}^{n}(\mathsf{FreeV}(A_i)) - \{ p_1, \ldots, p_n \} \tag{5b}$$
$$\mathsf{BoundV}(A) = \cup_{i=0}^{n}(\mathsf{BoundV}(A_i)) \cup \{ p_1, \ldots, p_n \} \tag{5c}$$

The formal language of full recursion L_r^λ is by Definition 1 without the Acyclicity Constraint (AC),

(A) The terms A of the form (3d) are called *recursion terms.* The constant where designates a binding operator, which binds the recursion variables $p_1, \ldots, p_n$ in A. Its entire scope is A called where-scope or its local recursion scope. The sub-terms A_i, $i = 0, \ldots, n$, are the parts of A and A_0 is its *head* part
(B) We say that a term A is *explicit* iff the constant where does not occur in it
(C) A is a λ-*calculus term*, i.e., a term of Gallin TY_2, iff it is explicit and no recursion variable occurs in it

Definition 3 (Free Occurrences and Replacement Operation). *Assume that* $A, C \in \mathsf{Terms}$, $X \in \mathsf{PureV} \cup \mathsf{RecV}$ *are such that, for some type* $\tau \in \mathsf{Types}$, $X, C : \tau$.

(1) An occurrence of X *in* A *is* free (in A) *if and only if it is not in the scope of any binding operator (e.g.,* $\xi \in \{ \lambda, \exists, \forall \}$ *and* where*) that binds* X
(2) The result of the simultaneous replacement *of all free (unless otherwise stated) occurrences of* X *with* C *in* A *is denoted by* $A\{ X :\equiv C \}$

(3) The replacement $A\{ X :\equiv C \}$ of X with C in A is free if and only if no free occurrence of X in A is in the scope of any operator that binds some variable having free occurrences in C: i.e., no variable that is free in C becomes bound in $A\{ X :\equiv C \}$. We also say that C is free for (replacing) X in A.

Notation 2. *Often, we do not write the type assignments in the term expressions.*

Sometimes, we shall use different kinds of or extra parentheses, or omit such. Application is associative to the left, λ-abstraction and quantifiers to the right.

In addition, we shall use abbreviations for sequences, e.g. ($n \geq 0$):

$$\overrightarrow{p} := \overrightarrow{A} \equiv p_1 := A_1, \ \ldots, \ p_n := A_n \quad (n \geq 0) \tag{6a}$$

$$H(\overrightarrow{A}) \equiv H(A_1)\ldots(A_n) \equiv (\ldots H(A_1)\ldots)(A_n) \qquad \textit{(left-association)} \tag{6b}$$

$$\begin{aligned} &\xi(\overrightarrow{v})(A) \equiv \xi(v_1)\ldots\xi(v_n)(A) \equiv \xi(v_1)\big[\ldots\big[\xi(v_n)\big(A\big)\big]\big] \ \textit{(right-association)} \\ &\qquad \xi \in \{\lambda, \exists, \forall\} \quad (n \geq 0) \end{aligned} \tag{6c}$$

$$\begin{aligned} &\overrightarrow{\xi(v)}(A) \equiv \xi_1(v_1)\ldots\xi_n(v_n)(A) \equiv \xi_1(v_1)\big[\ldots\big[\xi_n(v_n)\big(A\big)\big]\big], \\ &\qquad \xi_i \in \{\lambda, \exists, \forall\}, \ i \in \{1, \ldots, n\} \quad (n \geq 0) \end{aligned} \tag{6d}$$

$$\mathsf{lgh}(\overrightarrow{X}) = \mathsf{lgh}((X_1)\ldots(X_n)) = n, \quad \mathsf{lgh}(\xi(\overrightarrow{v})) = n, \quad \mathsf{lgh}(\overrightarrow{\xi(v)}) = n \tag{6e}$$

2.2 Overview of Algorithmic Semantics in $\mathbf{L}^{\boldsymbol{\lambda}}_{\mathbf{ar}}$ ($\mathbf{L}^{\boldsymbol{\lambda}}_{\boldsymbol{r}}$)

The syntax-semantics interface in $\mathrm{L}^{\lambda}_{\mathrm{ar}}$ (L^{λ}_{r}) provides the interrelations between denotational and algorithmic semantics.

Definition 4 (Immediate and Pure Terms). *The set of the* immediate terms *consists of all terms of the form* (7)*, for $p \in$ RecVars, $u_i, v_j, \in$ PureVars ($i = 1, \ldots, n$, $j = 1, \ldots, m$, $m, n \geq 0$), $V \in$ Vars:*

$$T :\equiv V \mid p(v_1)\ldots(v_m) \mid \lambda(u_1)\ldots\lambda(u_n)p(v_1)\ldots(v_m), \quad \textit{for } m, n \geq 0 \tag{7}$$

Every term A that is not immediate is proper.

The immediate terms $T \equiv \lambda(\overrightarrow{u})p(\overrightarrow{v})$ have no algorithmic meanings. Their denotational value $\mathsf{den}(T)(g)$ is given immediately, by the valuation functions g for $g(v_i)$, and abstracting away from the values u_j, for λ-bound pure variables $\lambda(\overrightarrow{u})p(\overrightarrow{v})$.

For every proper, i.e., non-immediate, term A, there is an algorithm $\mathsf{alg}(A)$ for computing $\mathsf{den}(A)(g)$. The canonical form $\mathsf{cf}_{\gamma^*}(A)$ of a proper term A determines the algorithm for computing its denotational value $\mathsf{den}(A)(g) = \mathsf{den}(\mathsf{cf}_{\gamma^*}(A))(g)$ from the components $\mathsf{den}(A_i)(g)$ of $\mathsf{cf}_{\gamma^*}(A)$. See γ^*-Canonical Form Theorem 1, and [6–8,18].

– The type theories $\mathrm{L}^{\lambda}_{\mathrm{ar}}$ have *effective reduction calculi*, see Sect. 3:
  For every $A \in$ Terms, there is a unique, up to congruence, canonical form $\mathsf{cf}_{\gamma^*}(A)$, which can be obtained from A, by a finite number of reductions:

$$A \Rightarrow^*_{\gamma^*} \mathsf{cf}_{\gamma^*}(A) \tag{8}$$

– For a given, fixed semantic structure $\mathfrak{A}$ and valuations G, for every *algorithmically meaningful*, i.e., proper, $A \in \mathsf{Terms}_\sigma$, the algorithm $\mathsf{alg}(A)$ for computing $\mathsf{den}(A)$ is determined by $\mathsf{cf}(A)$, so that:

$$\mathsf{den}(A)(g) = \mathsf{den}(\mathsf{cf}(A))(g), \text{ for } g \in G \tag{9}$$

Figure 1 depicts of the syntax-semantics relations between the syntax of Natural Language, their rendering to the terms $\mathrm{L}^{\lambda}_{\mathrm{ar}}$ and the corresponding algorithmic and denotational semantics.

$$\underbrace{\text{Syntax: NL / } \mathrm{L}^{\lambda}_{\mathrm{ar}} \Longrightarrow \underbrace{\text{Algorithms for Iterative Computations: } \mathsf{cf}_{\gamma^*}(A)}_{\text{Canonical Computations}} \Longleftrightarrow \text{Denotations}}_{\text{Computational Syn-Sem: Algorithmic and Denotational Semantics of NL via NL } \xrightarrow{\text{render}} \mathrm{L}^{\lambda}_{\mathrm{ar}}}$$

Fig. 1. Computational Syntax-Semantics Interface for Algorithmic Semantics of Natural Language via Compositional Rendering to $\mathrm{L}^{\lambda}_{\mathrm{ar}}$.

2.3 Denotational Semantics of $\mathrm{L}^{\lambda}_{\mathrm{ar}}$

Definition 5. *A* standard semantic structure *of the formal language $L^{\lambda}_{ar}(K)$ is a tuple $\mathfrak{A}(K) = \langle \mathbb{T}, \mathcal{I}(K) \rangle$, where $\mathbb{T}$ is a* frame *of sets (or classes) $\mathbb{T} = \{\, \mathbb{T}_\sigma \mid \sigma \in \mathsf{Types} \,\}$, and the following conditions (S1)–(S3) are satisfied:*

(S1) sets of basic, typed semantic objects:
- $\mathbb{T}_e \neq \varnothing$ *is a nonempty set (class) of entities called* individuals
- $\mathbb{T}_t = \{\, 0, 1, er \,\} \subseteq \mathbb{T}_e$, $\mathbb{T}_t$ *is called the set of the* truth values
- $\mathbb{T}_s \neq \varnothing$ *is a nonempty set of objects called* states

(S2) $\mathbb{T}_{(\tau_1 \to \tau_2)} = \{\, f \mid f : \mathbb{T}_{\tau_1} \to \mathbb{T}_{\tau_2} \,\}$

(S3) The interpretation function $\mathcal{I}$, $\mathcal{I} : K \to \bigcup \mathbb{T}$, *is such that for every constant* $\mathsf{c} \in K_\tau$, $\mathcal{I}(\mathsf{c}) = c$, *for some* $c \in \mathbb{T}_\tau$

Definition 6. *Assume a given semantic structure $\mathfrak{A}$. The set $G^{\mathfrak{A}}$ of all variable valuations (assignments) in $\mathfrak{A}$ is* (10a)–(10b)*:*

$$G^{\mathfrak{A}} = \{\, g \mid g : (\mathsf{PureV} \cup \mathsf{RecV}) \to \cup\mathbb{T}, \tag{10a}$$

$$\textit{and } g(x) \in \mathbb{T}_\tau, \textit{ for all } \tau \in \textit{Type} \textit{ and } x \in \mathsf{PureV}_\tau \cup \mathsf{RecV}_\tau \,\} \tag{10b}$$

Definition 7 (Denotation Function). *A* denotation function $\mathsf{den}^{\mathfrak{A}}$ *of the semantic structure $L^{\lambda}_{ar}(K)$, $\mathsf{den}^{\mathfrak{A}} : \mathsf{Terms} \to (G \to \bigcup \mathbb{T})$, is defined by structural recursion, for all $g \in G$:*

(D1) Variables and constants:

$$\mathsf{den}^{\mathfrak{A}}(x)(g) = g(x), \; \textit{for } x \in \mathsf{Vars}; \quad \mathsf{den}^{\mathfrak{A}}(\mathsf{c})(g) = \mathcal{I}(\mathsf{c}), \; \textit{for } \mathsf{c} \in K \tag{11}$$

(D2) Application:

$$\mathsf{den}^{\mathfrak{A}}(A(B))(g) = \mathsf{den}^{\mathfrak{A}}(A)(g)(\mathsf{den}^{\mathfrak{A}}(B)(g)) \tag{12}$$

(D3) λ-abstraction: *for all* $x : \tau$, $B : \sigma$, $\mathsf{den}^{\mathfrak{A}}(\lambda(x)(B))(g) : \mathbb{T}_\tau \to \mathbb{T}_\sigma$ *is the function such that, for every* $t \in \mathbb{T}_\tau$,

$$[\mathsf{den}^{\mathfrak{A}}(\lambda(x)(B))(g)]\big(t\big) = \mathsf{den}^{\mathfrak{A}}(B)(g\{x := t\}) \tag{13}$$

(D4) Recursion:

$$\mathsf{den}^{\mathfrak{A}}(A_0 \text{ where } \{\, \overrightarrow{p} := \overrightarrow{A} \,\})(g) = \mathsf{den}^{\mathfrak{A}}(A_0)(g\{\, \overrightarrow{p_i} := \overrightarrow{\overline{p}_i} \,\}) \tag{14}$$

where $\overline{p}_i \in \mathbb{T}_{\tau_i}$ *are computed by recursion on* $\mathsf{rank}(p_i)$, *i.e., by* (15)*:*

$$\begin{aligned} \overline{p}_i &= \mathsf{den}^{\mathfrak{A}}(A_i)(g\{\, p_{i,1} := \overline{p}_{i,1}, \dots, p_{i,k_i} := \overline{p}_{i,k_i} \,\}) \\ &\quad \textit{for all } p_{i,1}, \dots, p_{i,k_i}, \ \textit{such that } \mathsf{rank}(p_{i,k}) < \mathsf{rank}(p_i) \end{aligned} \tag{15}$$

The denotation $\mathsf{den}(A_i)(g)$ *may depend essentially on the values stored in* p_j, *for* $\mathsf{rank}(p_j) < \mathsf{rank}(p_i)$.

(D5) Here, for the denotations of the constants of the logic operators, we shall present the state dependent cases, including the erroneous truth values. The state-independent cases are simpler and straightforwardly similar.

(D5a) $\mathsf{den}^{\mathfrak{A}}\big(A_1 \wedge A_2\big)(g) : \mathbb{T}_{\mathsf{s}} \to \mathbb{T}_{\mathsf{t}}$ *is the function such that, for every state* $s \in \mathbb{T}_{\mathsf{s}}$*:*

$$[\mathsf{den}^{\mathfrak{A}}(A_1 \wedge A_2)(g)](s) = V \in \mathbb{T}_{\mathsf{t}}, \ \textit{where } V \textit{ is as in } (17a) - (17c) \tag{16}$$

$$V = \begin{cases} 1, & \textit{if } [\mathsf{den}^{\mathfrak{A}}(A_i)(g)](s) = 1, \textit{ for } i = 1, 2 \quad (17a) \\ 0, & \textit{if } [\mathsf{den}^{\mathfrak{A}}(A_i)(g)](s) = 0, \textit{ for at least one } i = 1, 2 \quad (17b) \\ & \textit{and } [\mathsf{den}^{\mathfrak{A}}(A_i)(g)](s) \neq er, \textit{ for } i = 1, 2 \\ er, & \textit{otherwise, i.e.,} \quad (17c) \\ & \textit{if } [\mathsf{den}^{\mathfrak{A}}(A_i)(g)](s) = er, \textit{ for at least one } i = 1, 2 \end{cases}$$

(D5b) $\mathsf{den}^{\mathfrak{A}}\big(A_1 \vee A_2\big)(g) : \mathbb{T}_{\mathsf{s}} \to \mathbb{T}_{\mathsf{t}}$ *is the function such that, for every state* $s \in \mathbb{T}_{\mathsf{s}}$*:*

$$[\mathsf{den}^{\mathfrak{A}}(A_1 \vee A_2)(g)](s) = V \in \mathbb{T}_{\mathsf{t}}, \ \textit{where } V \textit{ is as in } (19a) - (19c) \tag{18}$$

$$V = \begin{cases} 1, & \textit{if } [\mathsf{den}^{\mathfrak{A}}(A_i)(g)](s) = 1, \textit{ for at least one } i = 1, 2 \quad (19a) \\ & \textit{and } [\mathsf{den}^{\mathfrak{A}}(A_i)(g)](s) \neq er, \textit{ for } i = 1, 2 \\ 0, & \textit{if } [\mathsf{den}^{\mathfrak{A}}(A_i)(g)](s) = 0, \textit{ for } i = 1, 2 \quad (19b) \\ er, & \textit{otherwise, i.e.,} \quad (19c) \\ & \textit{if } [\mathsf{den}^{\mathfrak{A}}(A_i)(g)](s) = er, \textit{ for at least one } i = 1, 2 \end{cases}$$

The definition of $\mathsf{den}^{\mathfrak{A}}(A_1 \rightarrow A_2)(g)$ *is in a similar mode.*

(D6) $\mathsf{den}^{\mathfrak{A}}\big(\neg(A)\big)(g) : \mathbb{T}_{\mathsf{s}} \rightarrow \mathbb{T}_{\mathsf{t}}$ *is such that, for every state* $s \in \mathbb{T}_{\mathsf{s}}$*:*

$$[\mathsf{den}^{\mathfrak{A}}(\neg(A))(g)](s) = \begin{cases} 1, & \text{if } [\mathsf{den}^{\mathfrak{A}}(A)(g)](s) = 0 \qquad (20a) \\ 0, & \text{if } [\mathsf{den}^{\mathfrak{A}}(A)(g)](s) = 1 \qquad (20b) \\ er, & \text{otherwise, i.e., if } [\mathsf{den}^{\mathfrak{A}}(A)(g)](s) = er \qquad (20c) \end{cases}$$

(D7) Pure Universal Quantifier $\forall$*:*[1]

(D7a) For the state-independent quantifier $\forall$ $(\tau = \mathsf{t})$*, the definition is similar to the state dependent one, and we do not present its details*

(D7b) For the state-dependent quantifier $\forall$ $(\tau = \widetilde{\mathsf{t}})$*, for every state* $s \in \mathbb{T}_{\mathsf{s}}$*:*
$\big[\mathsf{den}^{\mathfrak{A}}\big(\forall(v^{\sigma})(B^{\tau})\big)(g)\big](s) = V$*, where:*

$$V = \begin{cases} 1, & \text{if } \big[\mathsf{den}^{\mathfrak{A}}(B^{\tau})(g\{\, v := a\,\})\big](s) = 1, \text{ for all } a \in \mathbb{T}_{\sigma} \qquad (21a) \\ 0, & \text{if } \big[\mathsf{den}^{\mathfrak{A}}(B^{\tau})(g\{\, v := a\,\})\big](s) = 0, \text{ for some } a \in \mathbb{T}_{\sigma} \qquad (21b) \\ & \quad \text{and } \big[\mathsf{den}^{\mathfrak{A}}(B^{\tau})(g\{\, v := b\,\})\big](s) \neq er, \text{ for all } b \in \mathbb{T}_{\sigma} \\ er, & \text{otherwise} \qquad (21c) \end{cases}$$

(D8) Pure Existential Quantifier $\exists$*:*

(D8a) For the state-independent quantifier $\exists$*, with* $\tau = \mathsf{t}$*, the definition is similar to the state dependent one, and we do not present it here*

(D8b) For the state-dependent quantifier $\exists$*,* $(\tau = \widetilde{\mathsf{t}})$*, for every state* $s \in \mathbb{T}_{\mathsf{s}}$*:*
$\big[\mathsf{den}^{\mathfrak{A}}\big(\exists(v^{\sigma})(B^{\tau})\big)(g)\big](s) = V$*, where:*

$$V = \begin{cases} 1, & \text{if } \big[\mathsf{den}^{\mathfrak{A}}(B^{\tau})(g\{\, v := a\,\})\big](s) = 1, \text{ for some } a \in \mathbb{T}_{\sigma} \qquad (22a) \\ & \quad \text{and } \big[\mathsf{den}^{\mathfrak{A}}(B^{\tau})(g\{\, v := b\,\})\big](s) \neq er, \text{ for all } b \in \mathbb{T}_{\sigma} \\ 0, & \text{if } \big[\mathsf{den}^{\mathfrak{A}}(B^{\tau})(g\{\, v := a\,\})\big](s) = 0, \text{ for all } a \in \mathbb{T}_{\sigma} \qquad (22b) \\ er, & \text{otherwise} \qquad (22c) \end{cases}$$

Often, we shall skip the superscript in $G^{\mathfrak{A}}$ and $\mathsf{den}^{\mathfrak{A}}$, by writing G and den.

3 Gamma-Star Reduction Calculus of $\mathrm{L}^{\lambda}_{\mathrm{ar}}$

I designate the logic operators as a set of specialised, logic constants. In this way, I classify the reduction rules for the terms formed by (3e)–(3f) as special cases of the reduction rule for application terms.

In this section, I extend the set of the $\mathrm{L}^{\lambda}_{\mathrm{ar}}$-reduction rules introduced in [18], by adding:

(1) the reduction rules (ξ) for the quantifier terms (3g) together with the λ-abstract terms, $\xi \in \{\, \lambda, \exists, \forall \,\}$
(2) an additional reduction rule, the (γ^*) rule, (30a)–(30b), which extends the corresponding rule in [7]

[1] There are other possibilities for the truth values of the erroneous truth value *er* for the quantifiers, which we do not consider in this paper.

3.1 Congruence Relation Between Terms

Definition 8. *The* congruence *relation is the smallest equivalence relation (i.e., reflexive, symmetric, transitive) between terms* $\equiv_{\mathsf{c}} \subseteq \mathsf{Terms} \times \mathsf{Terms}$*, that is closed under:*

(1) operators of term-formation:
- *application, which includes logic constants because we introduced them as categorematic constants*
- *λ-abstraction and pure, logic quantifiers*
- *acyclic recursion*

$$\textit{If } A \equiv_{\mathsf{c}} A' \textit{ and } B \equiv_{\mathsf{c}} B', \textit{ then } A(B) \equiv_{\mathsf{c}} A'(B') \tag{ap-congr}$$

$$\textit{If } A \equiv_{\mathsf{c}} B, \textit{ and } \xi \in \{\lambda, \exists, \forall\}, \textit{ then } \xi(u)(A) \equiv_{\mathsf{c}} \xi(u)(B) \tag{lq-congr}$$

$$\begin{aligned} &\textit{If } A_i \equiv_{\mathsf{c}} B_i, \textit{ for } i = 0, \ldots, n, \textit{ then:} \\ &\quad A_0 \;\mathsf{where}\; \{ p_1 := A_1, \ldots, p_n := A_n \} \\ &\equiv_{\mathsf{c}} B_0 \;\mathsf{where}\; \{ p_1 := B_1, \ldots, p_n := B_n \} \end{aligned} \tag{rec-congr}$$

(2) renaming bound pure and recursion variables without variable collisions, by free replacements, see Definition 3

(a) renaming pure variables bound by λ-abstraction and pure, logic quantifiers

$$\begin{aligned} &\xi(x)(A) \equiv_{\mathsf{c}} \xi(y)(A\{x :\equiv y\}), \quad \textit{for } x, y \in \mathsf{PureV}_\tau, \xi \in \{\lambda, \exists, \forall\} \\ &\textit{assuming } y \in \mathsf{FreeV}(A) \textit{ and } y \textit{ is free for (replacing) } x \textit{ in } A \end{aligned} \tag{24a}$$

(b) renaming memory location (variables) bound by the recursion operator where*, in assignments*

$$\begin{aligned} A &\equiv A_0 \;\mathsf{where}\; \{ p_1 := A_1, \ldots, p_n := A_n \} \\ &\equiv_{\mathsf{c}} A'_0 \;\mathsf{where}\; \{ p'_1 := A'_1, \ldots, p'_n := A'_n \} \end{aligned} \tag{25a}$$

$$\begin{aligned} &\textit{assuming } p'_i \in \mathsf{FreeV}(A) \textit{ and } p'_i \textit{ is free for (replacing) } p_i \textit{ in } A_j \\ &A'_j \equiv A_j\{p_1 :\equiv p'_1, \ldots, p_n :\equiv p'_n\} \equiv A_j\{\overrightarrow{p} :\equiv \overrightarrow{p'}\}, \\ &i \in \{1, \ldots, n\}, \; j \in \{0, \ldots, n\} \end{aligned} \tag{25b}$$

(3) re-ordering of the assignments within the recursion terms

$$\begin{aligned} &\textit{for every permutation } \pi : \{1, \ldots, n\} \xrightarrow[\mathsf{onto}]{\mathsf{1-to-1}} \{1, \ldots, n\} \\ &A_0 \;\mathsf{where}\; \{ p_1 := A_1, \ldots, p_n := A_n \} \\ &\equiv_{\mathsf{c}} A_0 \;\mathsf{where}\; \{ p_{\pi(1)} := A_{\pi(1)}, \ldots, p_{\pi(n)} := A_{\pi(n)} \} \end{aligned} \tag{26}$$

3.2 Reduction Rules of Extended $\mathbf{L}^{\lambda}_{\mathbf{ar}}$

In this section, we define the set RedR of the reduction rules of TTA, which are the same for its variants of full and acyclic recursion L^{λ}_{r} and $\mathrm{L}^{\lambda}_{\mathrm{ar}}$, respectively.

Congruence If $A \equiv_{\mathsf{c}} B$, then $A \Rightarrow B$ (cong)

Transitivity If $A \Rightarrow B$ and $B \Rightarrow C$, then $A \Rightarrow C$ (trans)

Compositionality Replacement of sub-terms with correspondingly reduced ones respects the term structure by the definition of the term syntax:

$$\text{If } A \Rightarrow A' \text{ and } B \Rightarrow B', \text{ then } A(B) \Rightarrow A'(B') \tag{ap-comp}$$

$$\text{If } A \Rightarrow B, \text{ and } \xi \in \{\, \lambda, \exists, \forall \,\}, \text{ then } \xi(u)(A) \Rightarrow \xi(u)(B) \tag{lq-comp}$$

If $A_i \Rightarrow B_i$, for $i = 0, \ldots, n$, then

$$\begin{aligned} &A_0 \ \mathsf{where}\ \{\, p_1 := A_1, \ldots, p_n := A_n \,\} \\ \Rightarrow\ &B_0 \ \mathsf{where}\ \{\, p_1 := B_1, \ldots, p_n := B_n \,\} \end{aligned} \tag{rec-comp}$$

Head Rule Given that, for all $i = 1, \ldots, n$, $j = 1, \ldots, m$, $p_i \neq q_j$ and p_i does not occur freely in B_j:

$$\begin{aligned} &\big(A_0 \ \mathsf{where}\ \{\, \overrightarrow{p} := \overrightarrow{A} \,\}\big) \ \mathsf{where}\ \{\, \overrightarrow{q} := \overrightarrow{B} \,\} \\ \Rightarrow\ &A_0 \ \mathsf{where}\ \{\, \overrightarrow{p} := \overrightarrow{A},\ \overrightarrow{q} := \overrightarrow{B} \,\} \end{aligned} \tag{head}$$

Bekič-Scott Rule Given that, for all $i = 1, \ldots, n$, $j = 1, \ldots, m$, $p_i \neq q_j$ and q_j does not occur freely in A_i

$$\begin{aligned} &A_0 \ \mathsf{where}\ \{\, p := \big(B_0 \ \mathsf{where}\ \{\, \overrightarrow{q} := \overrightarrow{B} \,\}\big), \overrightarrow{p} := \overrightarrow{A} \,\} \\ \Rightarrow\ &A_0 \ \mathsf{where}\ \{\, p := B_0, \overrightarrow{q} := \overrightarrow{B},\ \overrightarrow{p} := \overrightarrow{A} \,\} \end{aligned} \tag{B-S}$$

Recursion-Application Rule Given that, for all $i = 1, \ldots, n$, p_i does not occur freely in B

$$\big(A_0 \ \mathsf{where}\ \{\, \overrightarrow{p} := \overrightarrow{A} \,\}\big)(B) \Rightarrow A_0(B) \ \mathsf{where}\ \{\, \overrightarrow{p} := \overrightarrow{A} \,\} \tag{recap}$$

Application Rule Given that $B \in \mathsf{Terms}$ is proper and $b \in \mathsf{RecV}$ is fresh, i.e., $b \in \big[\mathsf{RecV} - \big(\, \mathsf{FreeV}\, \big(A(B)\big) \cup \mathsf{BoundV}\, \big(A(B)\big)\big)\big]$,

$$A(B) \Rightarrow\ A(b) \ \mathsf{where}\ \{\, b := B \,\} \tag{ab}$$

λ and Quantifier Rules Let $\xi \in \{\, \lambda, \exists, \forall \,\}$

$$\begin{aligned} &\xi(u)\, (A_0 \ \mathsf{where}\ \{\, p_1 := A_1, \ldots, p_n := A_n \,\}) \\ \Rightarrow\ &\xi(u)\, A'_0 \ \mathsf{where}\ \{\, p'_1 := \lambda(u)\, A'_1, \ldots, p'_n := \lambda(u)\, A'_n \,\} \end{aligned} \tag{ξ}$$

given that, for every $i = 1, \ldots, n$ $(n \geq 0)$, $p'_i \in \mathsf{RecV}$ is a fresh recursion (memory) variable, and A'_i $(0 \leq i \leq n)$ is the result of the replacement of all the free occurrences of $p_1, \ldots, p_n$ in A_i with $p'_1(u), \ldots, p'_n(u)$, respectively, i.e.:

$$A'_i \equiv A_i\{p_1 :\equiv p'_1(u), \ldots, p_n :\equiv p'_n(u)\} \equiv A_i\{\overrightarrow{p} :\equiv \overrightarrow{p'(u)}\}\ (0 \leq i \leq n) \tag{29}$$

γ^*-Rule

$$A \equiv_{\mathsf{c}} A_0 \text{ where } \{\, \overrightarrow{a} := \overrightarrow{A},\, p := \lambda(\overrightarrow{u})\lambda(v)P,\ \overrightarrow{b} := \overrightarrow{B} \,\} \tag{30a}$$

$$\Rightarrow_{\gamma^*} A'_0 \text{ where } \{\, \overrightarrow{a} := \overrightarrow{A}',\, p' := \lambda(\overrightarrow{u})P,\ \overrightarrow{b} := \overrightarrow{B}' \,\} \tag{γ^*}$$

$$\begin{aligned} \equiv A_0\{\, p(\overrightarrow{u})(v) :\equiv p'(\overrightarrow{u}) \,\} \text{ where } \{ & \\ & \overrightarrow{a} := \overrightarrow{A}\{\, p(\overrightarrow{u})(v) :\equiv p'(\overrightarrow{u}) \,\}, \\ & p' := \lambda(\overrightarrow{u})P, \\ & \overrightarrow{b} := \overrightarrow{B}\{\, p(\overrightarrow{u})(v) :\equiv p'(\overrightarrow{u}) \,\} \,\} \end{aligned} \tag{30b}$$

given that:

- the term $A \in \mathsf{Terms}$ satisfies the γ^*-condition (given in Definition 9) for the assignment $p := \lambda(\overrightarrow{u})\lambda(v)P : (\overrightarrow{\vartheta} \to (\vartheta \to \tau))$
- $p' \in \mathsf{RecV}_{(\overrightarrow{\vartheta} \to \tau)}$ is a fresh recursion variable
- for each part X_i of $\overrightarrow{X}$ in (γ^*) and (30b) (i.e., for each $X_i \equiv A_i$ in $\overrightarrow{X} \equiv \overrightarrow{A}$, and each $X_i \equiv B_i$ in $\overrightarrow{X} \equiv \overrightarrow{B}$), X'_i is the result of the free replacements $X'_i \equiv X_i\{\, p(\overrightarrow{u})(v) :\equiv p'(\overrightarrow{u}) \,\}$ of all occurrences of $p(\overrightarrow{u})(v)$ by $p'(\overrightarrow{u})$ (in the free occurrences of p), modulo renaming the variables $\overrightarrow{u}, v$, for $i \in \{0, \ldots, n_X\}$, i.e.:

$$\overrightarrow{X'} \equiv \overrightarrow{X}\{\, p(\overrightarrow{u})(v) :\equiv p'(\overrightarrow{u}) \,\} \tag{31}$$

Definition 9 (γ^*-Condition). *Assume that $i = 1, \ldots, n$ ($n \geq 0$), $\tau, \vartheta, \vartheta_i \in$ Types, $u, u_i \in \mathsf{PureV}$, $p \in \mathsf{RecV}$, $P \in \mathsf{Terms}$, are such that $u : \vartheta$, $u_i : \vartheta_i$, $p : (\overrightarrow{\vartheta} \to (\vartheta \to \tau))$, $P : \tau$, and thus, $\lambda(\overrightarrow{u}^{\overrightarrow{\vartheta}})\lambda(v^{\vartheta})(P^{\tau}) : (\overrightarrow{\vartheta} \to (\vartheta \to \tau))$.*

A recursion term $A \in \mathsf{Terms}$ satisfies the γ^*-condition for an assignment $p := \lambda(\overrightarrow{u}^{\overrightarrow{\vartheta}})\lambda(v^{\vartheta})(P^{\tau}) : (\overrightarrow{\vartheta} \to (\vartheta \to \tau))$, with respect to $\lambda(v)$, if and only if A is of the form (32)

$$A \equiv A_0 \text{ where } \{\, \overrightarrow{a} := \overrightarrow{A},\ p := \lambda(\overrightarrow{u})\lambda(v)P,\ \overrightarrow{b} := \overrightarrow{B} \,\} \tag{32}$$

with the sub-terms of appropriate types, such that the following holds:

(1) $P \in \mathsf{Terms}_{\tau}$ does not have any free occurrences of v, i.e., $v \notin \mathsf{FreeVars}(P)$
(2) All occurrences of p in A_0, $\overrightarrow{A}$, and $\overrightarrow{B}$ are free with respect to p (by renaming bound occurrences of recursion variables) and are occurrences in sub-terms $p(\overrightarrow{u})(v)$, which are in binding scope of $\xi_1(u_1), \ldots, \xi_n(u_n), \xi(v)$, for $\xi_i, \xi \in \{\, \lambda, \exists, \forall \,\}$, modulo renaming the bound variables $\overrightarrow{u}, v$, $i = 1, \ldots, n$ ($n \geq 0$)

Note: If we take away the second part of (2), which requires $p(\overrightarrow{u})(v)$ to be within the binding scopes of $\overrightarrow{\xi(u)}$, $\xi(v)$, the (γ^*) rule may remove free occurrences of pure variables, e.g., v in $p(\overrightarrow{u})(v)$, from some of the parts of the terms. This (strong) form of the γ^*-condition is introduced in [7].

When a recursion term A of the form (32) satisfies the γ^*-condition, given in Definition 9, we also say that *the assignment* $p := \lambda(\overrightarrow{u})\lambda(v)P$ *satisfies the* γ^**-condition, for any term* A' *such that* $A' \equiv_{\mathsf{c}} A$*, i.e., modulo congruence.*

Definition 10 (γ^*-Rules). *We shall call the set* RedR *of the reduction rules* (cong)–(ξ), (γ^*)*,* γ^*-reduction rules *and also, simply* L^{λ}_{ar}-reduction rules*.*

3.3 Reduction Relation

The extended set of reduction rules of $\mathrm{L}^{\lambda}_{\mathrm{ar}}$, (cong)–($\xi$), ($\gamma^*$), given in Sect. 3.2, defines the extended reduction relation $\Rightarrow^*_{\gamma^*}$ between $\mathrm{L}^{\lambda}_{\mathrm{ar}}$-terms, $A \Rightarrow^*_{\gamma^*} B$, by the alternatively expressed, equivalent Definition 11 and Definition 12.

Definition 11. *The* γ^*-reduction *relation* $\Rightarrow^*_{\gamma^*}$ *between terms is the smallest relation* $\Rightarrow^*_{\gamma^*} \subseteq \mathsf{Terms} \times \mathsf{Terms}$*, which is the reflexive and transitive closure of the* immediate reductions *by any of the reduction rules* (cong)–(ξ)*,* (γ^*)*.*

Definition 12 (γ^*-Reduction). *For all* $A, B \in \mathsf{Terms}$*,* $A \Rightarrow^*_{\gamma^*} B$ *iff there is a sequence of consecutive, immediate reductions by* (cong)–(γ^*)*, i.e.:*

$$\begin{aligned} A \Rightarrow^*_{\gamma^*} B \iff & \textit{there exist } A_i \in \mathsf{Terms}, 0 \leq i < n, \textit{ such that:} \\ & A \equiv A_0, \;\; A_n \equiv B, \;\; \textit{and} \qquad (33) \\ & A_i \Rightarrow A_{i+1}, \;\textit{for some of the rules } (\text{cong}) - (\gamma^*) \\ \iff & \;\textit{(abbreviated)}\; A \equiv A_0 \Rightarrow \ldots \Rightarrow A_n \equiv B \;\; (n \geq 0) \qquad (34) \end{aligned}$$

Often, we shall write $A \Rightarrow B$ instead of $A \Rightarrow^*_{\gamma^*} B$, including when applying none or more than one rule.

Lemma 1 (γ^*-Reducing Multiple, Innessential λ-Abstractions in an Assignment). *Assume that* $A \in \mathsf{Terms}$ *is of the form* (35a)–(35b)*:*

$$A \equiv A_0 \text{ where } \{ \, \overrightarrow{a} := \overrightarrow{A}, \; b := \lambda(\overrightarrow{u_1})\lambda(v_1)\ldots\lambda(\overrightarrow{u_k})\lambda(v_k)\lambda(\overrightarrow{u_{k+1}})B, \qquad (35a)$$

$$\overrightarrow{c} := \overrightarrow{C} \, \} \qquad (35b)$$

such that A *satisfies the* γ^**-condition in Definition* 9 *for the assignment for* b *in* (35a)*, with respect to all* λ*-abstractions* $\lambda(v_j)$*, for* $1 \leq j \leq k$*,* $k \in \mathbb{N}$*,* $k \geq 1$*.*

Then, the following reductions (36a)– (36b) *can be done:*

$$A \Rightarrow^*_{\gamma^*} A^k_0 \text{ where } \{ \, \overrightarrow{a} := \overrightarrow{A^k}, \; b^k := \lambda(\overrightarrow{u_1})\ldots\lambda(\overrightarrow{u_k})\lambda(\overrightarrow{u_{k+1}})B, \qquad (36a)$$

$$\overrightarrow{c} := \overrightarrow{C^k} \, \} \qquad (36b)$$

where for each part X_i *of* $\overrightarrow{X}$ *in* (35a)–(35b) *(i.e., for* $X_i \equiv A_i$ *in* $\overrightarrow{X} \equiv \overrightarrow{A}$ *or* $X_i \equiv C_i$ *in* $\overrightarrow{X} \equiv \overrightarrow{C}$*)* X^k_i *in* $\overrightarrow{X^k}$ *is the result of the replacements* (37a)–(37b)*, modulo renaming the bound variables* $\overrightarrow{u_l}, v_j$*, for* $i \in \{0, \ldots, n_X\}$*:*

$$\begin{aligned} X^k_i \equiv X_i\{ \, & b(\overrightarrow{u_1})(v_1)\ldots(\overrightarrow{u_k})(v_k)(\overrightarrow{u_{k+1}}) :\equiv b^k(\overrightarrow{u_1})\ldots(\overrightarrow{u_k})(\overrightarrow{u_{k+1}}) \, \} \\ & \textit{for } i \in \{0, \ldots, n_X\} \end{aligned} \qquad (37a)$$

$$\overrightarrow{X^k} \equiv \overrightarrow{X^k}\{ \, b(\overrightarrow{u_1})(v_1)\ldots(\overrightarrow{u_k})(v_k)(\overrightarrow{u_{k+1}}) :\equiv b^k(\overrightarrow{u_1})\ldots(\overrightarrow{u_k})(\overrightarrow{u_{k+1}}) \, \} \qquad (37b)$$

Proof. The proof is by induction on $k \in \mathbb{N}$, for $\mathrm{L}^{\lambda}_{\mathrm{ar}}$ extended by (ξ) and (γ^*) rules, for $\xi \in \{ \lambda, \exists, \forall \}$. We do not provide it here, because it is long. For such a lemma about the $\mathrm{L}^{\lambda}_{\mathrm{ar}}$, without logic operators and pure quantifiers, see [4,7]. □

Lemma 2 (γ^*-Reduction of the Assignments of a Recursion Term). *For every recursion term $P \equiv P_0$ where $\{ \overrightarrow{p} := \overrightarrow{P} \}$, (38a), there is a term Q of the form in (38b), such that Q does not satisfy the γ^*-condition in Definition 9, for any of its assignments $q_i := Q_i$ $(i = 1, \ldots, n)$ in (38b), and $P \Rightarrow^*_{\gamma^*} Q$, abbreviated by $P \Rightarrow Q$.*

$$P \equiv P_0 \;\mathsf{where}\; \{ p_1 := P_1, \ldots, p_n := P_n \} \equiv P_0 \;\mathsf{where}\; \{ \overrightarrow{p} := \overrightarrow{P} \} \tag{38a}$$

$$\Rightarrow^*_{\gamma^*} Q \equiv Q_0 \;\mathsf{where}\; \{ q_1 := Q_1, \ldots, q_n := Q_n \} \equiv Q_0 \;\mathsf{where}\; \{ \overrightarrow{q} := \overrightarrow{Q} \} \tag{38b}$$

Proof. See [4] extended by (ξ) and (γ^*) rules, for $\xi \in \{ \lambda, \exists, \forall \}$. □

Definition 13 (γ^*-Irreducible Terms). *We say that a term $A \in \mathsf{Terms}$ is* γ^*-irreducible *if and only if* (39) *holds:*

$$\textit{for all } B \in \mathsf{Terms}, \quad A \Rightarrow^*_{\gamma^*} B \implies A \equiv_{\mathsf{c}} B \tag{39}$$

3.4 Canonical Forms and γ^*-Reduction

Theorem 1 (γ^*-Canonical Form: Existence and Uniqueness of Canonical Forms). *See [6–8, 18]. For every term $A \in \mathsf{Terms}$, the following hold:*

(1) (Existence of a γ^*-canonical form of A) There exist explicit, γ^*-irreducible $A_0, \ldots, A_n \in \mathsf{Terms}$ $(n \geq 0)$, such that the term $\mathsf{cf}_{\gamma^*}(A)$ that is of the form (40) is γ^*-irreducible, i.e., irreducible and does not satisfy the γ-condition:

$$\mathsf{cf}_{\gamma^*}(A) \equiv A_0 \;\mathsf{where}\; \{ p_1 := A_1, \ldots, p_n := A_n \} \tag{40}$$

Thus, $\mathsf{cf}_{\gamma^*}(A)$ is γ^*-irreducible.

(2) A and $\mathsf{cf}_{\gamma^*}(A)$ have the same constants and free variables:

$$\mathsf{Consts}(A) = \mathsf{Consts}(\mathsf{cf}_{\gamma^*}(A)) \tag{41a}$$

$$\mathsf{FreeV}(A) = \mathsf{FreeV}(\mathsf{cf}_{\gamma^*}(A)) \tag{41b}$$

(3) $A \Rightarrow^*_{\gamma^*} \mathsf{cf}_{\gamma^*}(A)$

(4) If A is γ^*-irreducible, then $A \equiv_{\mathsf{c}} \mathsf{cf}_{\gamma^*}(A)$

(5) If $A \Rightarrow^*_{\gamma^*} B$, then $\mathsf{cf}_{\gamma^*}(A) \equiv_{\mathsf{c}} \mathsf{cf}_{\gamma^*}(B)$

(6) (Uniqueness of $\mathsf{cf}_{\gamma^*}(A)$ with respect to congruence) For every $B \in \mathsf{Terms}$, such that $A \Rightarrow^*_{\gamma^*} B$ and B is γ^*-irreducible, it holds that $B \equiv_{\mathsf{c}} \mathsf{cf}_{\gamma^*}(A)$, i.e., $\mathsf{cf}_{\gamma^*}(A)$ is unique, up to congruence. We write:

$$A \Rightarrow_{\mathsf{cf}_{\gamma^*}} B \iff B \equiv_{\mathsf{c}} \mathsf{cf}_{\gamma}(A) \tag{42}$$

Proof. The proof is by induction on term structure of A, in Definition 1, i.e., (3a)–(3g), using reduction rules, and properties of the extended γ^*-reduction relation.

Note: the reduction rules don't remove or add any constants and free variables. □

Algorithmic Semantics. The *algorithmic meaning* of a proper $A \in \mathsf{Terms}$, i.e., a non-immediate, algorithmically meaningful term, is designated by $\mathsf{alg}(A)$ and is determined by its canonical form $\mathsf{cf}(A)$.

Informally, for each proper $A \in \mathsf{Terms}$, the *algorithm* $\mathsf{alg}(A)$ for computing its denotation $\mathsf{den}(A)$ consists of computations provided by the basic parts A_i of its canonical form $\mathsf{cf}(A) \equiv A_0 \text{ where } \{p_1 := A_1, \ldots, p_n := A_n\}$, according to their structural rank, by recursive iteration.

For every $A \in \mathsf{Terms}$, $\mathsf{cf}(A)$, i.e., $\mathsf{cf}_{\gamma^*}(A)$, is obtained from A by the reduction calculus of $\mathrm{L}^{\lambda}_{\mathrm{ar}}$, introduced in Sect. 3.2.

Definition 14 (Algorithmic Equivalence). *Assume a given semantic structure $\mathfrak{A}$. For all $A, B \in \mathsf{Terms}$, A and B are γ^*-*algorithmically equivalent (i.e., synonymous) *in $\mathfrak{A}$, $A \approx_{\gamma^*} B$ iff*

- *A and B are both immediate, or*
- *A and B are both proper*

and, in each of these cases, there are explicit, γ^*-irreducible terms (of appropriate types), $A_0, \ldots, A_n, B_0, \ldots, B_n$, $n \geq 0$, such that:

(1) $A \Rightarrow^*_{\gamma^*} A_0 \text{ where } \{\, p_1 := A_1, \ldots, p_n := A_n \,\} \equiv \mathsf{cf}_{\gamma^*}(A)$
(2) $B \Rightarrow^*_{\gamma^*} B_0 \text{ where } \{\, q_1 := B_1, \ldots, q_n := B_n \,\} \equiv \mathsf{cf}_{\gamma^*}(B)$
(3) for all $i \in \{\, 0, \ldots, n \,\}$:

$$\mathsf{den}^{\mathfrak{A}}(A_i)(g) = \mathsf{den}^{\mathfrak{A}}(B_i)(g), \; \textit{for every variable valuation } g \in G \tag{43a}$$

$$\mathsf{den}^{\mathfrak{A}}(A_i) = \mathsf{den}^{\mathfrak{A}}(B_i) \tag{43b}$$

When $A \approx_{\gamma^*} B$, we say that A and B are algorithmically γ^*-equivalent, alternatively, that A and B are γ^*-synonymous. Sometimes, we skip the label γ^*.

4 Algorithmic Expressiveness of $\mathbf{L}^{\lambda}_{\mathbf{ar}}$

Moschovakis [18], via Theorem §3.24, proves that $\mathrm{L}^{\lambda}_{\mathrm{ar}}$ is a proper extension of Gallin TY_2, see Gallin [1]. Gallin [1], via his Theorem 8.2, can provide an interpretation of Montague IL [14] into TY_2. Suitable interpretation can be given in $\mathrm{L}^{\lambda}_{\mathrm{ar}}$ (L^{λ}_{r}), too. That is not our purpose in this paper.

Theorem 2, has the same formulation as Theorem §3.24 in [18]. The difference is that Theorem 2 covers the extended $\mathrm{L}^{\lambda}_{\mathrm{ar}}$ and its $\Rightarrow^*_{\gamma^*}$ reduction.

Theorem 2 (Conditions for Explicit and Non-Explicit Terms). *See Theorem §3.24, Moschovakis [18].*

(1) Necessary Condition for Explicit Terms: *For any explicit $A \in \mathsf{Terms}$, there is no memory (recursion) location that occurs in more than one part A_i $(0 \leq i \leq n)$ of* $\mathsf{cf}_{\gamma^*}(A)$

(2) Sufficient Condition for Non-Explicit Terms: *Assume that* $A \in \mathsf{Terms}$ *is such that a location* $p \in \mathsf{RecV}$ *occurs in (at least) two parts of* $\mathsf{cf}(A)$*, and respectively, of* $\mathsf{cf}_{\gamma^*}(A)$*, and the denotations of those parts depend essentially on p:*

$$A \Rightarrow^*_{\gamma^*} \mathsf{cf}_{\gamma^*}(A) \equiv A_0 \text{ where } \{ p_1 := A_1, \ldots, p_n := A_n \} \tag{44a}$$

$$p \in \mathsf{FreeV}(A_k), \quad p \in \mathsf{FreeV}(A_l) \ (k \neq l) \tag{44b}$$

$$\mathsf{den}(A_k)(g\{p :\equiv r\}) \neq \mathsf{den}(A_k)(g\{p :\equiv r'\}), \ \textit{for some } r, r' \in \mathbb{T}_\sigma \tag{44c}$$

$$\mathsf{den}(A_l)(g\{p :\equiv r\}) \neq \mathsf{den}(A_l)(g\{p :\equiv r'\}), \ \textit{for some } r, r' \in \mathbb{T}_\sigma \tag{44d}$$

Then, there is no explicit term to which A is algorithmically equivalent.

The proof of Theorem §3.24, Moschovakis [18] is extended for the logic operators, pure quantifiers and the γ^*-reduction. □

The extended, algorithmic expressiveness of $\mathrm{L}^{\lambda}_{\mathrm{ar}}$ is demonstrated by the terms in the following examples, which provide specific instantiations of algorithmic patterns of large classes and subtle semantic distinctions.

Logic Quantifiers and Reductions with Quantifier Rules: Assume that $\mathrm{L}^{\lambda}_{\mathrm{ar}}$ has $\mathit{cube}, \mathit{large}_0 \in \mathsf{Consts}_{(\widetilde{\mathsf{e}} \to \widetilde{\mathsf{t}})}$, and $\mathit{large} \in \mathsf{Consts}_{((\widetilde{\mathsf{e}} \to \widetilde{\mathsf{t}}) \to (\widetilde{\mathsf{e}} \to \widetilde{\mathsf{t}}))}$ as a modifier.

$$\text{Some cube is large} \xrightarrow{\mathsf{render}} B \equiv \exists x(\mathit{cube}(x) \land \mathit{large}_0(x)) \tag{45a}$$

$$B \Rightarrow \exists x((c \land l) \text{ where } \{ c := \mathit{cube}(x),\ l := \mathit{large}_0(x) \}) \tag{45b}$$

2x(ab) to $\land$; (lq-comp)

$$\Rightarrow \underbrace{\exists x(c'(x) \land l'(x))}_{B_0 \text{ algorithmic pattern}} \text{ where } \{ \tag{45c}$$

$$\underbrace{c' := \lambda(x)(\mathit{cube}(x)),\ l' := \lambda(x)(\mathit{large}_0(x))}_{\text{instantiations of memory slots } c', l'} \} \tag{45d}$$

$$\equiv \mathsf{cf}(B) \quad \text{from (45b), by } (\xi) \text{ to } \exists$$

$$\approx \underbrace{\exists x(c'(x) \land l'(x))}_{B_0 \text{ algorithmic pattern}} \text{ where } \{ \underbrace{c' := \mathit{cube},\ l' := \mathit{large}_0}_{\text{instantiations of memory slots } c', l'} \} \equiv B' \tag{45e}$$

$$\text{by Definition 14 from (45c)} - \text{(45d)}, \mathsf{den}(\lambda(x)(\mathit{cube}(x))) = \mathsf{den}(\mathit{cube}), \quad \mathsf{den}(\lambda(x)(\mathit{large}_0(x))) = \mathsf{den}(\mathit{large}_0) \tag{45f}$$

Repeated Calculations:

$$\text{Some cube is large} \xrightarrow{\text{render}} T, \quad \mathit{large} \in \mathsf{Consts}_{((\widetilde{\mathsf{e}}\to\widetilde{\mathsf{t}})\to(\widetilde{\mathsf{e}}\to\widetilde{\mathsf{t}}))} \tag{46a}$$

$$T \equiv \exists x\big[\mathit{cube}(x) \land \underbrace{\mathit{large}(\mathit{cube})(x)}_{\text{by predicate modification}}\big] \Rightarrow \ldots \tag{46b}$$

$$\Rightarrow \exists x\big[(c_1 \land l) \text{ where } \{ c_1 := \mathit{cube}(x),\, l := \mathit{large}(c_2)(x),\, c_2 := \mathit{cube} \}\big] \tag{46c}$$

(ab) to $\land$; (lq-comp), (B-S)

$$\Rightarrow \exists x\, \underbrace{(c_1'(x) \land l'(x))}_{T_0} \text{ where } \{ c_1' := \lambda(x)(\mathit{cube}(x)), \tag{46d}$$

$$l' := \lambda(x)(\mathit{large}(c_2'(x))(x)),\, c_2' := \lambda(x)\mathit{cube} \} \tag{46e}$$

(46d) – (46e) is by (ξ) on (46c) for $\exists$

$$\Rightarrow_{\gamma^*} \exists x(c_1'(x) \land l'(x)) \text{ where } \{ c_1' := \lambda(x)(\mathit{cube}(x)), \tag{46f}$$

$$l' := \lambda(x)(\mathit{large}(c_2)(x)),\, c_2 := \mathit{cube} \} \tag{46g}$$

$$\equiv \mathsf{cf}_{\gamma^*}(T) \qquad \text{by } (\gamma^*) \text{ rule to } c_2'(x) \text{ for } c_2' := \lambda(x)\mathit{cube}$$

$$\approx \exists x(c_1'(x) \land l'(x)) \text{ where } \{ c_1' := \mathit{cube}, \tag{46h}$$

$$l' := \lambda(x)(\mathit{large}(c_2)(x)),\, c_2 := \mathit{cube} \} \tag{46i}$$

Proposition 1. *The L_{ar}^{λ}-terms $C \approx \mathsf{cf}(C)$ in (47a)–(47e), similarly to many other L_{ar}^{λ}-terms, are not algorithmically equivalent to any explicit term.*

Therefore, L_{ar}^{λ} (L_{r}^{λ}) is a strict, proper extension of Gallin TY$_2$.

Proof. It follows from (47a)–(47e), by Theorem 2, (2), since c' occurs in two parts of $\mathsf{cf}(C)$ in (47e):

$$\text{Some cube is large} \xrightarrow{\text{render}} C \tag{47a}$$

$$C \equiv \underbrace{\exists x\big[c'(x) \land \mathit{large}(c')(x)\big]}_{E_0} \text{ where } \{ c' := \mathit{cube} \} \tag{47b}$$

$$\Rightarrow \underbrace{\exists x\big[\big(c'(x) \land l\big) \text{ where } \{ l := \mathit{large}(c')(x) \}\big]}_{E_1} \text{ where } \{ c' := \mathit{cube} \} \tag{47c}$$

from (47b), by (ab) to $\land$ of E_0; (lq-comp) of $\exists$; (rec-comp)

$$\Rightarrow \big[\underbrace{\exists x\big(c'(x) \land l'(x)\big) \text{ where } \{ l' := \lambda(x)\big(\mathit{large}(c')(x)\big) \}}_{E_2}\big] \text{ where } \{ \tag{47d}$$

$$c' := \mathit{cube} \} \qquad \text{from (47c), by } (\xi) \text{to} \exists$$

$$\Rightarrow \underbrace{\exists x\big(c'(x) \land l'(x)\big)}_{C_0\text{: an algorithmic pattern}} \text{ where } \{ \underbrace{c' := \mathit{cube},\; l' := \lambda(x)\big(\mathit{large}(c')(x)\big)}_{\text{instantiations of memory } c', l'} \} \tag{47e}$$

$$\equiv \mathsf{cf}(C) \quad \text{from (47d), by (head); (cong) of reordering assignments}$$

5 Expressiveness of L_{ar}^{λ} for Coordination in Natural Language Phrases

5.1 Coordinated Predication Versus Sentential Conjunction

In this paper, we have extended the algorithmic expressiveness of L_{ar}^{λ}.

We demonstrate it by comparing natural language sentences and their renderings into L_{ar}^{λ} recursion terms, which express their algorithmic meanings, e.g., (49c)–(49d) and (50j)–(50k). The canonical forms $\mathsf{cf}(A)$ in (49c)–(49d) and (50j)–(50k) are denotationally and algorithmically equivalent to the λ-calculus term A in (49b) and (50a).

In addition, there are L_{ar}^{λ} recursion terms that are not algorithmically equivalent to any λ-calculi terms, see (A)–(C), Proposition 2, and also Sect. 6.

Coordinated Predication: a class of sentences with coordinated VPs

$$[\Phi_j]_{\text{NP}}\ \big[[\Theta_L \text{ and } \Psi_H]\ [W_w]_{\text{NP}}\big]_{\text{VP}} \xrightarrow{\text{render}} A_0 \tag{48a}$$

$$A_0 \equiv \underbrace{\lambda x_j \big[\lambda y_w \big(L(x_j)(y_w) \land H(x_j)(y_w)\big)(w)\big](j)}_{\text{algorithmic pattern with memory parameters } L, H, w, j} \tag{48b}$$

Specific Instantiations of Parametric Algorithms, e.g., (48a)–(48b) and (49c), by (49d):

$$[\text{John}]_j \text{ loves and honors } [\text{his}]_j \text{ wife.} \xrightarrow{\text{render}} A \tag{49a}$$

$$A \equiv \lambda x_j \big[\lambda y_w \big(\mathit{loves}(y_w)(x_j) \land \mathit{honors}(y_w)(x_j)\big)(\mathit{wife}(x_j))\big](\mathit{john}) \tag{49b}$$

$$\Rightarrow \ldots \Rightarrow \mathsf{cf}(A) \equiv \underbrace{\lambda x_j \big[\lambda y_w \big(L''(x_j)(y_w) \land H''(x_j)(y_w)\big)(w'(x_j))\big](j)}_{\text{algorithmic pattern with memory parameters } L'', H'', w', j} \tag{49c}$$

$$\begin{aligned}\text{where } \{ & L'' := \lambda x_j \lambda y_w\, \mathit{loves}(y_w)(x_j),\\ & H'' := \lambda x_j \lambda y_w\, \mathit{honors}(y_w)(x_j),\\ & \underbrace{w' := \lambda x_j\, \mathit{wife}(x_j),\ j := \mathit{john}}_{\text{instantiations of memory } L'', H'', w', j} \}\end{aligned} \tag{49d}$$

The predication by the sentence (49a) is expressed denotationally by the rendering term A in (49b). The algorithm for computing its denotation $\mathsf{den}(A)$ in L_{ar}^{λ}, is determined by its canonical form $\mathsf{cf}(A)$ (49c)–(49d).

Reduction of Coordinated Relation to Canonical Form. A reduction of the predication term A in (49b) to its canonical form $\mathsf{cf}(A)$ (49c)–(49d) is provided by (50a)–(50j):

$$A \equiv \lambda x_j \Big[\lambda y_w \big[love(y_w)(x_j) \land honors(y_w)(x_j)\big](wife(x_j))\Big](john) \tag{50a}$$

$$\begin{aligned} \Rightarrow \lambda x_j \Big[\lambda y_w \big[(L \land H) \text{ where } \{ L := love(y_w)(x_j), \\ H := honors(y_w)(x_j) \}\big](wife(x_j))\Big](john) \end{aligned} \tag{50b}$$

(50b) is by: 2x(ab) to $\land$, 2x(lq-comp), (ap-comp), from (50a)

$$\begin{aligned} \Rightarrow \lambda x_j \Big[\big[\lambda y_w\, (L'(y_w) \land H'(y_w)) \text{ where } \{ L' := \lambda y_w\, love(y_w)(x_j), \\ H' := \lambda y_w\, honors(y_w)(x_j) \}\big](wife(x_j))\Big](john) \end{aligned} \tag{50c}$$

(50c) is by (ξ) for λy_w, (ap-comp), (lq-comp), (ap-comp), from (50b)

$$\begin{aligned} \Rightarrow \lambda x_j \Big[\big[\lambda y_w\, \big(L'(y_w) \land H'(y_w)\big)(wife(x_j))\big] \\ \text{where } \{ L' := \lambda y_w\, love(y_w)(x_j), \\ H' := \lambda y_w\, honors(y_w)(x_j) \}\Big](john) \end{aligned} \tag{50d}$$

(50d) is by (recap), (lq-comp), (ap-comp), from (50c)

$$\begin{aligned} \Rightarrow \lambda x_j \Big[\big[\lambda y_w\, \big(L'(y_w) \land H'(y_w)\big)(w) \text{ where } \{ w := wife(x_j) \}\big] \\ \text{where } \{ L' := \lambda y_w\, love(y_w)(x_j), \\ H' := \lambda y_w\, honors(y_w)(x_j) \}\Big](john) \end{aligned} \tag{50e}$$

(50e) is by (ab), (rec-comp), (lq-comp), (ap-comp), from (50d)

$$\begin{aligned} \Rightarrow \lambda x_j \big[[\lambda y_w\, \big(L'(y_w) \land H'(y_w)\big)(w)] \\ \text{where } \{ L' := \lambda y_w\, love(y_w)(x_j), \\ H' := \lambda y_w\, honors(y_w)(x_j), \\ w := wife(x_j) \}\big](john) \end{aligned} \tag{50f}$$

(50f) is by (head), (cong), (lq-comp), (ap-comp), from (50e)

$$\begin{aligned} \Rightarrow \Big[\lambda x_j [\lambda y_w\, \big(L''(x_j)(y_w) \land H''(x_j)(y_w)\big)(w'(x_j))] \\ \text{where } \{ L'' := \lambda x_j \lambda y_w\, love(y_w)(x_j), \\ H'' := \lambda x_j \lambda y_w\, honors(y_w)(x_j), \\ w' := \lambda x_j\, wife(x_j) \}\Big](john) \end{aligned} \tag{50g}$$

(50g) is by (ξ) to λx_j, (ap-comp) from (50f)

$$\begin{aligned} \Rightarrow \Big[\big[\lambda x_j [\lambda y_w\, \big(L''(x_j)(y_w) \land H''(x_j)(y_w)\big)(w'(x_j))]\big](john) \\ \text{where } \{ L'' := \lambda x_j \lambda y_w\, love(y_w)(x_j), \\ H'' := \lambda x_j \lambda y_w\, honors(y_w)(x_j), \\ w' := \lambda x_j\, wife(x_j) \}\Big] \end{aligned} \tag{50h}$$

(50h) is by (recap), from (50g)

$$\Rightarrow \Big[\big[[\lambda x_j [\lambda y_w \, \big(L''(x_j)(y_w) \land H''(x_j)(y_w)\big)(w'(x_j))]](j) \text{ where } \{ j := john \}\big] \text{ where } \{ L'' := \lambda x_j \lambda y_w \, love(y_w)(x_j), \; H'' := \lambda x_j \lambda y_w \, honors(y_w)(x_j), \; w' := \lambda x_j \, wife(x_j) \}\Big] \tag{50i}$$

(50i) is by (ab), (rec-comp), from (50h)

$$\Rightarrow \big[\lambda x_j [\lambda y_w \, \big(L''(x_j)(y_w) \land H''(x_j)(y_w)\big)(w'(x_j))](j) \text{ where } \{ L'' := \lambda x_j \lambda y_w \, love(y_w)(x_j), \; H'' := \lambda x_j \lambda y_w \, honors(y_w)(x_j), \; w' := \lambda x_j \, wife(x_j), \; j := john \}\big] \tag{50j}$$

(50j) is by (head), (cong), from (50i)

$$\approx \big[\lambda x_j [\lambda y_w \, \big(L''(x_j)(y_w) \land H''(x_j)(y_w)\big)(w'(x_j))](j) \text{ where } \{ L'' := \lambda x_j \lambda y_w \, love(y_w)(x_j), \; H'' := \lambda x_j \lambda y_w \, honors(y_w)(x_j), \; w' := wife, \; j := john \}\big] \tag{50k}$$

(50k) is by Definition 14 and $\mathsf{den}(\lambda x_j \, wife(x_j)) = \mathsf{den}(wife)$, from (50j)

In contrast to (49a)–(49b), the propositional content of the sentence in (51a), which is a predicative conjunction, can be represented by the following recursion terms (51b)–(51c) of $\mathrm{L}^{\lambda}_{\mathrm{ar}}$. The terms in (51b)–(51c) are algorithmically equivalent (synonymous), by the reduction calculus of $\mathrm{L}^{\lambda}_{\mathrm{ar}}$, and their head parts are conjunction propositions, which is expressed by the sentence (51a) too:

$$[\text{John}]_j \text{ loves } [[\text{his}]_j \text{ wife}]_w \text{ and } [\text{he}]_j \text{ honors } [\text{her}]_w \tag{51a}$$

$$\xrightarrow{\text{render}}_{\text{co-index}_{ar}} \big[love(w)(j) \land honors(w)(j)\big] \text{ where } \{ j := john, \; w := wife(j) \} \tag{51b}$$

$$\Rightarrow_{\mathsf{cf}_{\gamma^*}} \big[L \land H\big] \text{ where } \{L := love(w)(j), \; H := honors(w)(j), \; j := john, \; w := wife(j) \} \tag{51c}$$

Proposition 2. *(1) The terms in the reduction sequence (50a)–(50j) are all algorithmically equivalent with each other and with (50k)*

(2) The terms in (50a)–(50j), (50k) are not algorithmically equivalent with the ones in (51b)–(51c)

(3) The terms (51b)–(51c) are not algorithmically equivalent to any explicit L^{λ}_{ar}, which are λ-calculus, i.e., Gallin TY_2 *terms (see (A)–(C) on page 5)*

(4) The terms (51b)–(51c) are not algorithmically equivalent to any λ-calculus terms that are interpreted IL terms into TY_2

Proof. (1)–(2) follow directly from Definition 14 and (43a)–(43b). (3)–(4) follow from Theorem 2, and also from Theorem §3.24 in [18]. This is because there is a recursion variable (i.e., two, j and w) occurring in more than one part of the γ^*-canonical form (51c). □

6 Some Relations Between Let-Expressions and Recursion Terms

Scott [21] introduced the let-expressions by the LCF language of λ-calculus, which has been implemented by the functional programming languages, e.g., ML, see e.g., Milner [13], Scheme,[2], Haskell,[3] e.g., see Marlow [12], OCaml, etc. Classic imperative languages, e.g., ALGOL and Pascal, implement let-expressions for the scope of functions in their definitions.

A lambda calculus with a formal language that includes terms of let-binding is presented by Nishizaki [19]. A constant where is used in the formation of terms in Landin [2], which are similar to let-expressions.

The formal language of full recursion L_r^λ, see Definition 1, (3a)–(3d), without the acyclicity (AC), is an extension of the language LCF introduced by Plotkin [20]. The λ-calculus of LCF has been having a grounding significance in Computer Science, for the distinctions between denotational and operational semantics.

Details of possible similarities and differences between let-expressions in λ-calculus, and the recursion $\mathrm{L}_{\mathrm{ar}}^\lambda$ terms of the form (3d) $(n \geq 1)$, in Definition. 1, need carefull representation, which is not in the scope of the work.

In this section, we show that, in general, the recursion $\mathrm{L}_{\mathrm{ar}}^\lambda$ terms diverge from the standard let-expressions, in the sense that the reduction calculi of $\mathrm{L}_{\mathrm{ar}}^\lambda$ provide algorithmic meanings of the $\mathrm{L}_{\mathrm{ar}}^\lambda$ terms via their canonical forms $\mathsf{cf}(A)$ and $\mathsf{cf}_{\gamma^*}(A)$, and the γ^*-Canonical Form Theorem 1.

The algorithmic semantics by $\mathrm{L}_{\mathrm{ar}}^\lambda$ and L_r^λ is provided by the reduction system, which includes, very importantly, division of the variables into two kinds, proper and recursion, and also of terms as either immediate or proper. Recursion variables $p \in \mathsf{RecV}$ are for assignments in the scope of the where operator. They can not be used for λ-abstraction, which uses pure variables. To have a correspondence of a recursion term A, e.g. as in (53a), with a let-expression via a sequence of characteristic λ-abstractions, as in (52a), we can use one-to-one, bijective replacements with fresh pure variables, as in (54a).

The λ-terms of the form in (52a) are characteristic for the values of the corresponding let-expressions, and can be used as a defining representation of let-expressions:

$$\mathsf{let}\ x_1 = D_1, \ldots, x_n = D_n\ \mathsf{in}\ D_0 \equiv \lambda(x_1)\big(\ldots[\lambda(x_n)(D_0)](D_n)\ldots\big)(D_1) \tag{52a}$$

$$\text{if } x_j \in \mathsf{FreeV}(D_i),\ \text{then } j < i,\ \text{ i.e., } \mathsf{den}(D_i) \text{ may depend on } \mathsf{den}(x_j) \tag{52b}$$

Assume that $A \in \mathsf{Terms}$ is a $\mathrm{L}_{\mathrm{ar}}^\lambda$ term of the form (53a), for some $A_j \in \mathsf{Terms}$, $j \in \{0, \ldots, n\}$, such that:

(1) A_j has no occurrences of recursion (memory) variables that are different from $p_i \in \mathsf{RecV}$, $i \in \{1, \ldots, n\}$

[2] https://www.scheme.com/tspl4/start.html#./start:h4,.
[3] https://www.haskell.org.

(2) rank is such that (53b) holds

$$A \equiv \mathsf{cf}_{\gamma^*}(A) \equiv A_0 \text{ where } \{p_1 := A_1, \ldots, p_n := A_n\} \tag{53a}$$

$$\mathsf{rank}(p_i) = i, \text{ for } i \in \{1, \ldots, n\} \tag{53b}$$

Note: It can be proved that, for each L_{ar}^{λ} term (3d), Definition 1, there is at least one such rank, see [6]. For any i, j, such that $j < i$, it is not required that $p_j \in \mathsf{FreeV}(A_i)$, but this is possible, even for more than one i, see Theorem 2, sentences like (51a) and terms (51b)–(51c).

For the purpose of the demonstration in this section, we introduce specific let-expressions, by the abbreviations (54a)–(54b). We focus on the special case of $n = 1$, (56), in the rest of this section.

$$\mathsf{let}\ x_1 = D_1, \ldots, x_n = D_n \ \mathsf{in}\ D_0 \tag{54a}$$

$$\equiv \lambda(x_1)\big(\ldots[\lambda(x_n)(D_0)](D_n)\ldots\big)(D_1) \tag{54b}$$

$$x_i \in \mathsf{PureV}_{\tau_i}, x_i \notin \mathsf{Vars}(A), n \geq 1, \text{ for } i \in \{1, \ldots, n\}$$

$$D_j \equiv A_j\{p_1 :\equiv x_1, \ldots, p_n :\equiv x_n\}, \text{ for } j \in \{0, \ldots, n\}$$

In the special case of $n = 1$, with just one assignment::

$$A \equiv \mathsf{cf}_{\gamma^*}(A) \equiv A_0 \text{ where } \{p_1 := A_1\}, \quad p_1 \notin \mathsf{Vars}(A_1) \text{ for acyclicity} \tag{55}$$

$$\mathsf{let}\ x_1 = A_1 \ \mathsf{in}\ A_0\{p_1 :\equiv x_1\} \equiv \lambda(x_1)\big(A_0\{p_1 :\equiv x_1\}\big)(A_1) \tag{56}$$

When replacing a memory variable $p \in \mathsf{RecV}$ with a pure variable $x \in \mathsf{PureV}$, in an explicit, irreducible term A, the result can be reducible term, by (ab). When an immediate term of the form $\lambda(\overrightarrow{u})p(\overrightarrow{v})$, for $(\mathsf{lgh}(\overrightarrow{u}) + \mathsf{lgh}(\overrightarrow{v})) \geq 1$, e.g., $p(u)$, $\lambda(v)p$, $\lambda(v)p(u)$, etc., occurs in an argument position of A. After replacement, $\lambda(\overrightarrow{u})x(\overrightarrow{v})$ is not an immediate term, by Definition 4.

Lemma 3. *Assume that $C \in \mathsf{Terms}$ is explicit, irreducible, such that (1)–(2):*

(1) $p_1 \in \mathsf{RecV}_{\tau_1}$, $\overrightarrow{u}, \overrightarrow{v}, z \in \mathsf{PureV}$, such that $(\mathsf{lgh}(\overrightarrow{u}) + \mathsf{lgh}(\overrightarrow{v})) \geq 1$
(2) $p_1 \notin \mathsf{FreeV}(C)$

Let $A_0 \equiv \lambda(z)\big[C\big(\lambda(\overrightarrow{u})p_1(\overrightarrow{v})\big)\big]$. Let $x_1 \in \mathsf{PureV}_{\tau_1}$ and x_1 be fresh for A_0, i.e., $x_1 \notin \mathsf{Vars}(A_0)$. Then:

$$C(\lambda(\overrightarrow{u})p_1(\overrightarrow{v})) \text{ and } A_0 \equiv \lambda(z)\big[C\big(\lambda(\overrightarrow{u})p_1(\overrightarrow{v})\big)\big] \text{ are explicit, irreducible} \tag{57a}$$

$$[C(\lambda(\overrightarrow{u})p_1(\overrightarrow{v}))]\{p_1 :\equiv x_1\} \text{ and } A_0 \text{ are reducible} \tag{57b}$$

Proof. By (2), $C\{p_1 :\equiv x_1\} \equiv C$. The following reductions can be done:

$$[C(\lambda(\overrightarrow{u})p_1(\overrightarrow{v}))]\{p_1 :\equiv x_1\} \equiv C(\lambda(\overrightarrow{u})x_1(\overrightarrow{v})) \tag{58a}$$

$$\Rightarrow C(r_1) \text{ where } \{\, r_1 := \lambda(\overrightarrow{u})x_1(\overrightarrow{v})\,\} \qquad \text{by (ab) from (58a)} \tag{58b}$$

Then:

$$A_0\{p_1 :\equiv x_1\} \equiv \lambda(z)\big[C\big(\lambda(\overrightarrow{u})p_1(\overrightarrow{v})\big)\big]\{p_1 :\equiv x_1\} \\ \equiv \lambda(z)\big[C\big(\lambda(\overrightarrow{u})x_1(\overrightarrow{v})\big)\big] \tag{59a}$$

$$\Rightarrow \lambda(z)\big[C(r_1) \text{ where } \{\, r_1 := \lambda(\overrightarrow{u})x_1(\overrightarrow{v}) \,\}\big] \qquad \text{by (ab), (lq-comp)} \tag{59b}$$

$$\Rightarrow \lambda(z)\big[C(r_1'(z))\big] \text{ where } \{\, r_1' := \lambda(z)\lambda(\overrightarrow{u})x_1(\overrightarrow{v}) \,\} \equiv A_0' \quad \text{by } (\xi) \text{ for } \lambda(z) \tag{59c}$$

There are two cases:
Case 1 $z \in \mathsf{FreeV}(\lambda(\overrightarrow{u})p_1(\overrightarrow{v}))$. Then $A_0\{p_1 :\equiv x_1\} \Rightarrow A_0' \equiv \mathsf{cf}_{\gamma^*}(A_0')$.
Case 2 $z \notin \mathsf{FreeV}(\lambda(\overrightarrow{u})x_1(\overrightarrow{v}))$. Then:

$$A_0' \Rightarrow_{(\gamma^*)} \lambda(z)\big[C(r_1)\big] \text{ where } \{\, r_1 := \lambda(\overrightarrow{u})x_1(\overrightarrow{v}) \,\} \equiv \mathsf{cf}_{\gamma^*}(A_0') \quad \text{by } (\gamma^*) \tag{60a}$$

$$A_0\{p_1 :\equiv x_1\} \Rightarrow A_0' \Rightarrow_{\gamma^*} \mathsf{cf}_{\gamma^*}(A_0') \tag{60b}$$

□

Lemma 4. *Assume that* $A \in \mathsf{Terms}$ *is as in* (61)*, with the variables as in Lemma 3,* Case 2*, i.e.,* $p_1 \in \mathsf{RecV}_{\tau_1}$, $\overrightarrow{u}, \overrightarrow{v}, z \in \mathsf{PureV}$*, for explicit, irreducible* $C, A_1 \in \mathsf{Terms}$*, such that* A_1 *is proper, and* $p_1 \notin \mathsf{FreeV}(C)$ *(*$p_1 \notin \mathsf{FreeV}(A_1)$ *by acyclicity),* $x_1 \in \mathsf{PureV}_{\tau_1}$*,* $x_1 \notin \mathsf{Vars}(A)$*, and* $z \notin \mathsf{FreeV}(\lambda(\overrightarrow{u})x_1(\overrightarrow{v}))$*:*

$$A \equiv \mathsf{cf}_{\gamma^*}(A) \equiv \underbrace{\lambda(z)\big[C\big(\lambda(\overrightarrow{u})p_1(\overrightarrow{v})\big)\big]}_{A_0} \text{ where } \{\, p_1 := A_1 \,\} \tag{61}$$

Then, the conversion of the assignment in A *into a* λ*-abstract over* A_0*, applied to* A_1*, results in a term, which is not algorithmically equivalent to* A *(similarly, for* Case 1*):*

$$A \not\approx_{\gamma^*} A' \equiv \big[\lambda(x_1)\big(A_0\{p_1 :\equiv x_1\}\big)\big](A_1) \tag{62}$$

Proof.

$$A' \equiv \big[\lambda(x_1)\big(A_0\{p_1 :\equiv x_1\}\big)\big](A_1) \tag{63a}$$

$$\equiv \lambda(x_1)\Big[\underbrace{\big[\lambda(z)\big[C\big(\lambda(\overrightarrow{u})p_1(\overrightarrow{v})\big)\big]\big]}_{A_0}\{p_1 :\equiv x_1\}\Big](A_1) \tag{63b}$$

$$\Rightarrow \lambda(x_1)\Big[\lambda(z)\big[C(r_1)\big] \text{ where } \{\, r_1 := \lambda(\overrightarrow{u})x_1(\overrightarrow{v}) \,\}\Big](A_1) \tag{63c}$$

by (60a), (lq-comp), (ap-comp)

$$\Rightarrow \Big[\lambda(x_1)\big[\lambda(z)\big[C(r_1^1(x_1))\big]\big] \text{where } \{\, r_1^1 := \lambda(x_1)\lambda(\overrightarrow{u})x_1(\overrightarrow{v}) \,\}\Big](A_1) \tag{63d}$$

by (ξ) for $\lambda(x_1)$, (ap-comp)

$$\Rightarrow \lambda(x_1)\big[\lambda(z)\big[C(r_1^1(x_1))\big]\big](A_1) \text{ where } \{\, r_1^1 := \lambda(x_1)\lambda(\overrightarrow{u})x_1(\overrightarrow{v}) \,\} \tag{63e}$$

by (recap)

$$\Rightarrow \Big[\lambda(x_1)\big[\lambda(z)\big[C(r_1^1(x_1))\big]\big](p_1) \text{ where } \{p_1 := A_1\}\Big] \\ \text{where } \{\, r_1^1 := \lambda(x_1)\lambda(\overrightarrow{u})x_1(\overrightarrow{v}) \,\} \tag{63f}$$

by (ab), (rec-comp)

$$\Rightarrow \lambda(x_1)\big[\lambda(z)\big[C(r^1_1(x_1))\big]\big](p_1) \text{ where } \{p_1 := A_1,\ r^1_1 := \lambda(x_1)\lambda(\overrightarrow{u})x_1(\overrightarrow{v})\} \equiv \mathsf{cf}_{\gamma^*}(A') \quad \text{by (head)} \tag{63g}$$

Thus, (62) holds: A and A' are not algorithmically equivalent, $A \not\approx_{\gamma^*} A'$, which follows, by Definition 14, from (61) and (63g). (Similarly, for *Case 1*.) □

Proposition 3. *In general, the algorithmic equivalence does not hold between the L^{λ}_{ar} recursion terms of the form* (53a) *and the λ-calculus terms* (54a)*–*(54b)*, which are characteristic for the corresponding let-expressions in λ-calculus.*

Proof. By Lemma 4, the special set of terms in it provide counterexamples to alleged algorithmic equality between all terms in (53a) and (54a)–(54b). □

The let-expressions, represented by the specific, characteristic λ-terms (54a)–(54b) in $\mathrm{L}^{\lambda}_{\mathrm{ar}}$, are only denotationally equivalent to the corresponding recursion terms, but not algorithmically in the most significant cases. The full proofs are the subject of forthcoming papers.

7 Conclusion and Outlook for Future Work

In this paper, I have presented some of the major characteristics of $\mathrm{L}^{\lambda}_{\mathrm{ar}}$, by also developing it for enhancing its mathematical capacities for logic, theoretically, by targeting applications.

Algorithmic Semantics: The essential theoretic features of $\mathrm{L}^{\lambda}_{\mathrm{ar}}$ provide algorithmic semantics of formal and natural languages. Computational semantics by $\mathrm{L}^{\lambda}_{\mathrm{ar}}$ has the fundamental distinction between algorithmic and denotational semantics. The algorithms determined by terms in canonical forms compute their denotations, see Fig. 1.

While the theory has already been quite well developed, with eyes towards versatile applications, it is an open subject with many open and ongoing tasks and perspectives. The greater semantic distinctions of the formal language and calculi of $\mathrm{L}^{\lambda}_{\mathrm{ar}}$ enhance type-theoretic semantics by traditional λ-calculi. I have demonstrated that, by being a strict extension of Gallin TY_2 [1], $\mathrm{L}^{\lambda}_{\mathrm{ar}}$ exceeds also the facilities of Montague [14] IL, e.g., see Sect. 4, Propositions 1–2.

Algorithmic Patterns for Computational Semantics: Memory locations, i.e., recursion variables in $\mathrm{L}^{\lambda}_{\mathrm{ar}}$ terms represent parameters that can be instantiated by corresponding canonical forms, depending on context, the specific areas of applications, and domain specific texts, e.g., as in (45c) and (47d); and (48a)–(48b), as in (50k).

Logical Constants and Quantifiers in $\mathrm{L}^{\lambda}_{\mathrm{ar}}$: Canonical forms can be used for reasoning and inferences of semantic information by automatic provers and proof assistants. This is a subject of future work.

References

1. Gallin, D.: Intensional and Higher-Order Modal Logic: With Applications to Montague Semantics. North-Holland Publishing Company, Amsterdam and Oxford, and American Elsevier Publishing Company (1975). https://doi.org/10.2307/2271880
2. Landin, P.J.: The mechanical evaluation of expressions. Comput. J. **6**(4), 308–320 (1964). https://doi.org/10.1093/comjnl/6.4.308
3. Loukanova, R.: Relationships between specified and underspecified quantification by the theory of acyclic recursion. ADCAIJ Adv. Distrib. Comput. Artif. Intell. J. **5**(4), 19–42 (2016). https://doi.org/10.14201/ADCAIJ2016541942
4. Loukanova, R.: Gamma-star reduction in the type-theory of acyclic algorithms. In: Rocha, A.P., van den Herik, J. (eds.) Proceedings of the 10th International Conference on Agents and Artificial Intelligence (ICAART 2018). vol. 2, pp. 231–242. INSTICC, SciTePress – Science and Technology Publications, Lda. (2018). https://doi.org/10.5220/0006662802310242
5. Loukanova, R.: Computational syntax-semantics interface with type-theory of acyclic recursion for underspecified semantics. In: Osswald, R., Retoré, C., Sutton, P. (eds.) IWCS 2019 Workshop on Computing Semantics with Types, Frames and Related Structures. Proceedings of the Workshop, Gothenburg, Sweden, pp. 37–48. The Association for Computational Linguistics (ACL) (2019). https://www.aclweb.org/anthology/W19-1005
6. Loukanova, R.: Gamma-reduction in type theory of acyclic recursion. Fund. Inform. **170**(4), 367–411 (2019). https://doi.org/10.3233/FI-2019-1867
7. Loukanova, R.: Gamma-star canonical forms in the type-theory of acyclic algorithms. In: van den Herik, J., Rocha, A.P. (eds.) ICAART 2018. LNCS (LNAI), vol. 11352, pp. 383–407. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-05453-3_18
8. Loukanova, R.: Type-theory of acyclic algorithms for models of consecutive binding of functional neuro-receptors. In: Grabowski, A., Loukanova, R., Schwarzweller, C. (eds.) AI Aspects in Reasoning, Languages, and Computation. SCI, vol. 889, pp. 1–48. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-41425-2_1
9. Loukanova, R.: Type-theory of parametric algorithms with restricted computations. In: Dong, Y., Herrera-Viedma, E., Matsui, K., Omatsu, S., González Briones, A., Rodríguez González, S. (eds.) DCAI 2020. AISC, vol. 1237, pp. 321–331. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-53036-5_35
10. Loukanova, R.: Currying order and restricted algorithmic beta-conversion in type theory of acyclic recursion. In: Materna, P., Jespersen, B. (eds.) Logically Speaking. A Festschrift for Marie Duží. Book Tribute, vol. 49, pp. 285–310. College Publications (2022). https://doi.org/10.13140/RG.2.2.34553.75365
11. Loukanova, R.: Eta-reduction in type-theory of acyclic recursion. ADCAIJ: Adv. Distrib. Comput. Artif. Intell. J. **12**(1), 1–22 (2023). Ediciones Universidad de Salamanca. https://doi.org/10.14201/adcaij.29199
12. Marlow, S.: Haskell 2010, language report. Technical report (2010). https://www.haskell.org, https://www.haskell.org/onlinereport/haskell2010/
13. Milner, R.: A theory of type polymorphism in programming. J. Comput. Syst. Sci. **17**(3), 348–375 (1978). https://doi.org/10.1016/0022-0000(78)90014-4
14. Montague, R.: The proper treatment of quantification in ordinary English. In: Hintikka, J., Moravcsik, J., Suppes, P. (eds.) Approaches to Natural Language, vol. 49, pp. 221–242. Synthese Library. Springer, Dordrecht (1973), https://doi.org/10.1007/978-94-010-2506-5_10

15. Moschovakis, Y.N.: The formal language of recursion. J. Symb. Log. **54**(4), 1216–1252 (1989). https://doi.org/10.1017/S0022481200041086
16. Moschovakis, Y.N.: Sense and denotation as algorithm and value. In: Oikkonen, J., Väänänen, J. (eds.) Logic Colloquium 1990: ASL Summer Meeting in Helsinki, Lecture Notes in Logic, vol. 2, pp. 210–249. Springer-Verlag, Berlin (1993). https://projecteuclid.org/euclid.lnl/1235423715
17. Moschovakis, Y.N.: The logic of functional recursion. In: Dalla Chiara, M.L., Doets, K., Mundici, D., van Benthem, J. (eds.) Logic and Scientific Methods, vol. 259, pp. 179–207. Springer, Dordrecht (1997). https://doi.org/10.1007/978-94-017-0487-8_10
18. Moschovakis, Y.N.: A logical calculus of meaning and synonymy. Linguist. Philos. **29**(1), 27–89 (2006). https://doi.org/10.1007/s10988-005-6920-7
19. Nishizaki, S.: Let-binding in a linear lambda calculus with first-class continuations. IOP Conf. Ser. Earth Environ. Sci. **252**(4), 042011 (2019). https://doi.org/10.1088/1755-1315/252/4/042011
20. Plotkin, G.D.: LCF considered as a programming language. Theoret. Comput. Sci. **5**(3), 223–255 (1977). https://doi.org/10.1016/0304-3975(77)90044-5
21. Scott, D.S.: A type-theoretical alternative to ISWIM, CUCH, OWHY. Theoret. Comput. Sci. **121**(1), 411–440 (1993). https://doi.org/10.1016/0304-3975(93)90095-B