ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal
Regular Issue, Vol. 12 N. 1 (2023), e29199
eISSN: 2255-2863
DOI: https://doi.org/10.14201/adcaij.29199

Eta-Reduction in Type-Theory of Acyclic Recursion

Roussanka Loukanova

Institute of Mathematics and Informatics, Bulgarian Academy of Sciences, Sofia, Bulgaria
rloukanova@gmail.com

KEYWORDS

algorithm; denotation; reduction;

ABSTRACT

We investigate the applicability of the classic eta-conversion in the type-theory of acyclic algorithms. While denotationally valid, classic eta-conversion is not algorithmically valid in the type theory of algorithms, with the exception of few limited cases. The paper shows how the restricted, algorithmic eta-rule can recover algorithmic eta-conversion in the reduction calculi of type-theory of algorithms.

1. Background

1.1. Overview of the Algorithmic Eta-Conversion

The work presented in this paper is part of theoretical development of a new approach to the mathematical notion of algorithm that was introduced with the formal languages of recursion (FLR), by Moschovakis in a sequence of papers (Moschovakis, 1989; Moschovakis, 1993; Moschovakis, 1997). The formal languages of recursion FLR are untyped systems. They formalise algorithmic computations of denotations in untyped domains of recursive functions. Their theoretical strength is that they allow full recursion with cyclicity and are equivalent to any of the classic theories of the mathematical notion of algorithm.

Moschovakis introduced the class of the simply-typed formal languages and theories of full and acyclic recursion, and designated them by L^{λ}_{r} and $\mathrm{L}^{\lambda}_{ar}$, respectively, see (Moschovakis, 2006). The theory L^{λ}_{r} is a proper, strict extension of (Gallin, 1975) logic TY_2, and thus, of Montague Intensional Logic (IL), (Thomason, 1974).

Type theory $\mathrm{L}^{\lambda}_{ar}$ (Moschovakis, 2006) is based on a simply-typed formal language, its syntax, semantics, and reduction calculus. By extending the mathematical notions of FLR, $\mathrm{L}^{\lambda}_{ar}$ is restricted to recursion terms with acyclicity. Thus, $\mathrm{L}^{\lambda}_{ar}$ formalises algorithms that end after finite steps of computations. Typed languages of full recursion L^{λ}_{r} have the syntax of $\mathrm{L}^{\lambda}_{ar}$, without the acyclicity requirement.

In general, the classes of languages of recursion (FLR, L_r^{λ}, and L_{ar}^{λ}) have two semantic layers: denotational semantics and algorithmic semantics. The recursion terms of L_{ar}^{λ} belong to a distinctive, new kind of formal terms, which are essential for representing algorithmic computations of semantic information, and storing it in memory slots. By this, L_{ar}^{λ} sets a new approach to the mathematical concepts of algorithm.

This paper has in its focus the simply-typed theory of acyclic algorithms L_{ar}^{λ}, from the perspective of its potential applications. The formal languages and calculi of L_{ar}^{λ} provide new approaches to intelligent theoretical foundations, with already existing applications. Here, we extend the original reduction calculus of L_{ar}^{λ} introduced in (Moschovakis, 2006), by adding an additional η-rule operating on recursion terms formed by the specialised recursion operator of L_{ar}^{λ}. We consider that the new η-rule and the induced η-reduction, are important for applications of L_{ar}^{λ}, especially to computational syntax-semantics interfaces of formal and natural languages, as well as in the various areas of Artificial Intelligence (AI).

Among the potential applications of the type theory of recursion L_{ar}^{λ} are intelligent software systems, e.g., in robotics and in AI, that perform algorithmic procedures, which also provide reliable performance. Prominent applications of L_{ar}^{λ} are to computational semantics of formal and natural languages, and, in particular, to algorithmic semantics of programming and other specification languages in Computer Science. There are already established applications of the formal languages of recursion, in functional and relational versions, to Natural Language Processing (NLP) and computational grammars that cover computational semantics.

This paper presents a restricted η-reduction in L_{ar}^{λ}, which simplifies terms in canonical forms. The new η-rule introduced in L_{ar}^{λ} is specialised for operating over recursion terms of L_{ar}^{λ}. This η-rule does not preserve the strictest algorithmic equivalence (i.e., the strictest algorithmic synonymy) of its input and output terms, both of which are required in canonical forms. Importantly, the η-rule closely maintains the algorithmic meaning of the terms, while reducing the computational complexity caused by excessive, superfluous λ-abstractions, which are coupled with corresponding functional applications. The η-rule preserves the types and denotations of the input and reduced terms. The papers (Loukanova, 2019d; Loukanova, 2019c) introduce and investigate the properties of more intricate rules and reduction calculi for removing other redundant λ-abstractions. The η-rule, as presented in this paper, is also interesting, by extending the original reduction calculus of L_{ar}^{λ}, because it is applied directly, only to terms in canonical forms.

We should stress that the η-rule introduced in this paper for the type-theory L_{ar}^{λ} is about terms of acyclic recursion and the algorithms designated by them. The canonical forms of the terms determine the algorithms for computations of their denotations, which provide iteration steps, according to a computational rank of the parts of algorithms.

Note that the standard η-rule, from traditional λ-calculi, when expressed in L_r^{λ} and L_{ar}^{λ}, is only denotationally valid, and, in general, it is not algorithmically valid. The η-reduction in L_{ar}^{λ}, presented in this paper, closely maintains the algorithmic computations determined by recursion L_{ar}^{λ}- terms, by also reducing needless λ-abstractions. While both are related, the η-rule in L_{ar}^{λ} is different from the version of the classic, denotational η-rule. This is the reason for the name of the η-rule and its corresponding reduction presented in this paper. See the (η) rule. Sect. 2 provides an overview of L_{ar}^{λ} with references for a more detailed introduction. Sect. 3 gives information about the reduction calculi of the type theory L_{ar}^{λ}, and references to the papers in which it had been introduced in detail, which is essential for algorithmic semantics, especially in selected, specific, semantic domains of data. In Sect. 3.3, we

provide some key theoretical results of L^{λ}_{ar}. On their basis, Sect. 4 briefly introduces the algorithmic semantics of L^{λ}_{ar} with some intuitions.

The second part of the paper is devoted to the possibility of simplifying vacuous λ-abstractions and corresponding applications in recursion terms, by an extended η-reduction relation in L^{λ}_{ar}. Sect. 6 introduces the algorithmic, restricted (η) rule in L^{λ}_{ar}, while Sect. 7 uses it to provide the extended η-reduction calculus of L^{λ}_{ar}.

The central focus of the paper is on the introduction of the new, additional η-rule for the reduction of L^{λ}_{ar}-terms, which are in canonical forms, to simpler η-canonical forms that formalise more efficient algorithmic computations.

The new η-rule can be seen as a generalisation of the classic η-rule from λ-calculi, while these are essentially different. The new η-rule provides η-conversion for a major portion of the L^{λ}_{ar}-terms, with the exception of the class of λ-immediate terms.

1.2. Related Work

There have been works in procedural semantics, on ideas similar to those of the author, relating to distinguishing the procedural components of meanings from denotations.

A grounding work on procedural semantics, also in harmony with the author's work, including this paper, has been (Plotkin, 1975; Plotkin, 1977). It points to the potential problems caused by β-reductions in semantics of programming languages, by a focus on untyped formal language. By the type theory of algorithms, e.g., presented in this paper, the distinctions between denotational and algorithmic meanings are maintained.

As demonstrated in (Loukanova, 2009), the denotational β-reductions play significant roles in both layers, denotational and algorithmic semantics of L^{λ}_{ar} and L^{λ}_{r}. In general, β-conversion does not preserve the algorithmic semantics of L^{λ}_{ar} terms, with the exception of a restricted version of β-rule for special cases of explicit, irreducible, λ-terms applied to pure variables.

By underlining the effects of partial functions, (Tichý, 1995; Tichý, 1988) grounded a paradigm of an approach, which is in harmony with the author's work, both the present one and others. Tichý procedural semantics has been provided by the extensive work of (Duží, 2019). The approach uses structured propositions as structured procedures. Structured propositions are assigned to expressions as their meanings. Their constituents are sub-procedures. Significantly, in procedural semantics of hyperintensional Transparent Intensional Logic (TIL), β-reduction does not hold, as shown by the work of Marie Duží. The work (Duží and Jespersen, 2012; Duží and Kosterec, 2017) applied the approach to problems of natural language, such as anaphora resolution.

Similarly, in type-theory of algorithms L^{λ}_{ar} and L^{λ}_{r}, β-reduction does not preserve algorithmic semantics, in general. The two approaches, Tichý TIL and Moschovakis type-theory of algorithms are essentially different in their fundamental details, while they target and solve similar problems in semantics. A comparison would be an interesting future work, outside of the topic of this paper.

To the best of the author's knowledge, the study that for the very first time addressed similar problems was performed on relational, dependent-type theory of situated, partial information, by situation semantics (Loukanova, 2001; Loukanova, 2002a; Loukanova, 2002b) in situated, model-theoretic structures, i.e., at purely semantic level. A new development, by a formal language of dependent-type theory of situated information, which incorporates L^{λ}_{ar}, has been initiated (Loukanova, 2019b), and enhanced by two layers of numerical assessments (Loukanova, 2023).

A more detailed introduction to the mathematics of L^{λ}_{ar}, including the denotational and algorithmic semantics of L^{λ}_{ar}, and the relations between them, is provided, e.g., in (Moschovakis, 2006) and (Loukanova, 2019d; Loukanova, 2019c).

2. Introduction to Type Theory of Acyclic Recursion

2.1. Types

The set Types is the smallest set defined recursively as follows, by using a wide-spread notation in computer science:

$$\tau :\equiv \mathrm{e} \mid \mathrm{t} \mid \mathrm{s} \mid (\tau_1 \rightarrow \tau_2) \qquad \text{(Types)}$$

The type e is associated with *entities*, also called *individuals*, of the semantic domain, and for the L^{λ}_{ar}-terms denoting individuals. The type s is for states consisting of various pieces of information, e.g., such as possible worlds, context, time and/or space locations, and some agents, e.g., a speaker; t is for truth values. For an elaboration of possible choices of context information, by specialised objects called *states* in semantic domains D_s of type s, which include speaker agents (Loukanova, 2011a). The type $(s \rightarrow \tau)$ is specialised for context dependent objects, i.e., for any state dependent object f, which maps any given sate s to an object $f(s) = a$ of type τ.

2.2. Syntax of the Language of Acyclic Recursion

Vocabulary It consists of typed, pairwise disjoint, sets of typed constants and variables:

- *Constants:* $K = \cup_{\tau \in \mathrm{Types}} K_\tau$, where, for every $\tau \in$ Types, K_τ is the set of constants of type τ : $K_\tau = \{ \mathrm{c}^\tau_0, \ldots, \mathrm{c}^\tau_{k_\tau} \}$ (a finite set, $k_\tau \in \mathbb{N}$)
- *Pure Variables:* $\mathrm{PureV} = \cup_{\tau \in \mathrm{Types}} \mathrm{PureV}_\tau$, and for every $\tau \in$ Types, $\mathrm{PureV}_\tau = \{\mathrm{v}^\tau_0, \ldots\}$ is a countably infinite set (i.e., an infinite, denumerable set)
- *Recursion Variables* (which we also call *memory slots or memory locations*): $\mathrm{RecV} = \cup_{\tau \in \mathrm{Types}} \mathrm{RecV}_\tau$, and for every $\tau \in$ Types, $\mathrm{RecV}_\tau = \{\mathrm{p}^\tau_0, \ldots\}$ is a countably infinite set (i.e., an infinite, denumerable set)

Notation 1. Sometimes, for clarity, we use a verbose designation Consts for the set of the constants, i.e., by assuming Consts = K, and, for each $\tau \in$ Types, K_τ, we may use Consts_τ for the set of the constants of type τ, i.e., $\mathrm{Consts}_\tau = K_\tau$.

Terms We express the recursive rules for the set Terms(K) of the terms of L^{λ}_{ar} (K) by using a notational style of typed Backus–Naur forms (TBNF), in Def. 1. In it, as typically, we use A, B , C, A_0 , …, A_n as meta-variables for terms; $\mathrm{c}^\tau \in K_\tau$ as a meta-variable for constants of type τ; $x^\tau \in \mathrm{PureV}_\tau \cup \mathrm{RecV}_\tau$, for pure and recursion variables of any type τ ; $v^\sigma \in \mathrm{PureV}_\sigma$ for pure variables of type σ; $p_i \in \mathrm{RecV}_{\sigma_i}$ ($i = 1, \ldots, n$, $n \geq 0$) for recursion variables.

ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal
Regular Issue, Vol. 12 N. 1 (2023), e29199
eISSN: 2255-2863 - https://adcaij.usal.es
Ediciones Universidad de Salamanca - CC BY-NC-ND

Note 1 (Meta Type Assignment to Terms at Meta-Level). The assumed types of the given terms are presented as superscripts, and the types of the resulting terms with colon, but they are not per se parts of the term expressions.

In general, the type assignments in Def. 1 can be part of the terms, in specific cases of a language of L^{λ}_{ar}. Here, we assume that they are at the meta level of the notational style of typed TBNF and can be derived given the typed vocabulary (constants and variables), from Def. 1.

Definition 1 (Terms). The set Terms of L^{λ}_{ar} is defined by the following recursive rules (1)–(3):

$$A :\equiv c^{\tau} : \tau \mid x^{\tau} : \tau \mid B^{(\sigma \to \tau)}\,(C^{\sigma}) : \tau \tag{1}$$

$$\mid \lambda(v^{\sigma})\,(B^{\tau}) : (\sigma \to \tau) \tag{2}$$

$$\mid A_0^{\sigma} \text{ where } \{p_1^{\sigma_1} := A_1^{\sigma_1}, \ldots, p_n^{\sigma_n} := A_n^{\sigma_n}\} : \sigma \tag{3}$$

given that in (3): $A_1 : \sigma_1, \ldots, A_n : \sigma_n$ are terms ($n \geq 0$), i.e., $A_i \in \text{Terms}_{\sigma_i}$; $p_1 : \sigma_1, \ldots, p_n : \sigma_n$ are pairwise different recursion variables (memory slots), i.e., $p_i \in \text{RecV}_{\sigma_i}$; $p_i \neq p_j$, for all i, j, such that $i \neq j$ and $1 \leq i, j \leq n$; and, the sequence of *assignments* $\{ p_1 := A_1, \ldots, p_n := A_n \}$ satisfies the Acyclicity Constraint (AC) given below.

Note 2 (Dependence of Recursion on rank). Intuitively, an acyclic system (sequence) of assignments (3), $\{ p_1 := A_1, \ldots, p_n := A_n \}$, defines algorithmic computations of the values $\text{den}(A_1), \ldots, \text{den}(A_n)$, which are saved in the corresponding memory slots $p_1, \ldots, p_n$, via := for the assignment operator.

An acyclic system $\{ p_1 := A_1, \ldots, p_n := A_n \}$ defines recursive computations of the values to be assigned to the locations $p_1, \ldots, p_n$, which close-off after a finite number of steps.

The denotation of A_i is "saved" in p_i and can be dependent on the values saved in the memory variables $p_j \in \text{FreeV}(A_i)$, i.e., $\text{rank}(p_j) < \text{rank}(p_i)$.

Note 3. Definition 1, without the acyclicity requirement, determines an extended language L^{λ}_{r} that admits *full recursion* that ca be *cyclic*. This topic is not in the subject of this paper.

Notation 2. Throughout the paper, we use the following notational agreements:

(N1) The symbol "≡" is a meta-symbol (i.e., it is not in the language L^{λ}_{ar}), for orthographic, i.e., literal identity between expressions of L^{λ}_{ar}, e.g., used to introduce abbreviations and aliases

(N2) The symbol ":≡" is a meta-symbol that we use in definitions, e.g., of types and terms, and for the replacement operation

(N3) Often, we skip some "understood" parentheses, use different shapes and sizes of parentheses, and some extra parentheses, for clarity

(N4) The type σ of a term A may be depicted either as a superscript, A^{σ}, or by a colon, $A : \sigma$, for clarity and convenience

Notation 3. We use the following abbreviations for "folding" and "unfolding" sequences of assignments:

(Ab1) For any $p_i \in \text{RecV}_{\sigma_i}$, $C, D \in \text{Terms}_{\tau}$ and $A_i \in \text{Terms}_{\sigma_i}$, $i = 1, \ldots, n$ ($n \geq 0$):

$$\vec{p} := \vec{A} \equiv p_1 := A_1, \ldots, p_n := A_n \tag{4a}$$

$$\vec{p} := \vec{A}\{C :\equiv D\} \equiv p_1 := A_1\{C :\equiv D\}, \ldots, p_n := A_n\{C :\equiv D\} \tag{4b}$$

where $A_i\{C :\equiv D\}$ is the result of the replacement of all occurrences of C with D in A_i, usually without causing variable clashes

Denotational Semantics of the Language of Algorithms The language L_{ar}^{λ} has denotational semantics provided by a denotational function $den^{\mathfrak{A}}$, for semantic structures $\mathfrak{A}$ consisting of typed semantic domains and variable assignments g in $\mathfrak{A}$. The definition of the denotational values $den^{\mathfrak{A}}(A)$, for all terms A, in any given semantic structure $\mathfrak{A}$ of data, is by structural induction on the terms. For more details, see (Moschovakis, 2006) and (Loukanova, 2019d; Loukanova, 2019c).

3. The Original Reduction Calculus of Acyclic Recursion

The algorithmic semantics in the theories of acyclic L_{ar}^{λ} and full L_{r}^{λ} recursion are determined by the concept of canonical forms of terms. The canonical form $cf(A)$ of each term A is effectively obtained by the reduction calculi of L_{ar}^{λ} and full L_{r}^{λ}.

3.1. Proper versus Immediate Terms

An important distinction of the formal language and theory of L_{ar}^{λ} is the division of the L_{ar}^{λ}-terms between proper and immediate terms. Intuitively, the canonical form $cf(A)$ of each proper term A designates the algorithm for computing its denotation $den(cf(A)) = den(A)$.

On the other hand, the immediate terms are very simple terms that denote their values immediately via any given valuation function of variables.

Definition 2 (Immediate and Proper Terms). The *immediate terms* are all the terms that have one of the forms (5a)–(5b):

$$u, \quad \text{for } u \in \text{PureV} \tag{5a}$$

$$\lambda(u_1)\,[\ldots\,\lambda(u_n)\,((p(v_1)\ldots)(v_m))], \quad \text{for } p \in \text{RecV}, u_1, \ldots, u_n, v_1, \ldots, v_m \in \text{PureV}\ (n, m \geq 0) \tag{5b}$$

The terms that are not immediate are *proper*.

Intuitively, the denotations of the immediate terms are obtained immediately, by being available from any given valuation function $g \in G$ in the given semantic structure $\mathfrak{A}$. For a detailed discussion on a distinction between immediate terms and constants, see (Moschovakis, 2006). On the other hand, the denotations of the proper terms are computed by algorithmic steps, even if they may be very simple, by evoking the interpretation function for some simple constants.

3.2. Original Reduction Rules of the Theory of Acyclic Recursion

The set of the L_{ar}^{λ}-reduction rules is as follows:

Congruence

$$\text{If } A \equiv_c B, \text{ then } A \Rightarrow B \tag{cong}$$

Transitivity

$$\text{If } A \Rightarrow B \text{ and } B \Rightarrow C, \text{ then } A \Rightarrow C \qquad \text{(trans)}$$

Compositionality

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

$$\text{If } A \Rightarrow B, \text{ then } \lambda(u)\,(A) \Rightarrow \lambda(u)\,(B) \qquad (\lambda\text{-comp})$$

$$\begin{array}{l}\text{If } A_i \Rightarrow B_i, \text{ for } i = 0, \ldots, n, \text{ then} \\ A_0 \text{ where } \{p_1 := A_1, \ldots, p_n := A_n\} \\ \Rightarrow B_0 \text{ where } \{p_1 := B_1, \ldots, p_n := B_n\}\end{array} \qquad \text{(wh-comp)}$$

The Head Rule

$$\begin{array}{l}\left(A_0 \text{where} \left\{\vec{p} := \vec{A}\right\}\right) \text{where} \left\{\vec{q} := \vec{B}\right\} \\ \Rightarrow A_0 \text{where} \left\{\vec{p} := \vec{A}, \vec{q} := \vec{B}\right\}\end{array} \qquad \text{(head)}$$

given that no p_i occurs freely in any B_j, for $i = 1, \ldots, n, j = 1, \ldots, m$

The Bekič-Scott Rule

$$\begin{array}{l}A_0 \text{where} \left\{p := \left(B_0 \text{where} \left\{\vec{q} := \vec{B}\right\}\right), \vec{p} := \vec{A}\right\} \\ \Rightarrow A_0 \text{where} \left\{p := B_0, \vec{q} := \vec{B}, \vec{p} := \vec{A}\right\}\end{array} \qquad \text{(B-S)}$$

given that no q_j occurs freely in any A_i, for $i = 0, \ldots, n, j = 1, \ldots, m$

The Recursion-Application Rule

$$\begin{array}{l}\left(A_0 \text{where} \left\{\vec{p} := \vec{A}\right\}\right)(B) \\ \Rightarrow A_0\,(B) \text{ where} \left\{\vec{p} := \vec{A}\right\}\end{array} \qquad \text{(recap)}$$

given that no p_i occurs freely in B, for $i = 1, \ldots, n$

The Application Rule

$$A(B) \Rightarrow A(p) \text{ where } \{p := B\} \qquad \text{(ap)}$$

given that B is a proper term and p is a fresh (recursion) memory variable

The λ-Rule

$$\begin{array}{l}\lambda(u)\,(A_0 \text{ where } \{p_1 := A_1, \ldots, p_n := A_n\}) \\ \Rightarrow \lambda(u)\,A_0' \text{ where } \{p_1' := \lambda(u)\,A_1', \ldots, p_n' := \lambda(u)\,A_n'\}) \\ \equiv \lambda(u)\,A_0 \left\{\vec{p} := \overrightarrow{p'(u)}\right\} \text{where} \left\{\overrightarrow{p'} := \overrightarrow{\lambda(u)\,A\left\{\vec{p} := \overrightarrow{p'(u)}\right\}}\right\}\end{array} \qquad (\lambda)$$

ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal
Regular Issue, Vol. 12 N. 1 (2023), e29199
eISSN: 2255-2863 - https://adcaij.usal.es
Ediciones Universidad de Salamanca - CC BY-NC-ND

where:

(1) $u \in \text{PureV}_\sigma$

(2) for all $i = 1, \ldots, n$, $p_i \in \text{RecV}_{\sigma_i}$,

$p'_i \in \text{RecV}_{(\sigma \to \sigma_i)}$ is a fresh recursion variable

(3) for all $i = 0, \ldots, n$, $A_i \in \text{Terms}_{\sigma_i}$,

A'_i is the result of the replacement of the free occurrences of $p_1, \ldots, p_n$ in A_i with $p'_1(u), \ldots, p'_n(u)$, respectively, i.e.:

$$A'_i :\equiv A_i\{p_1 :\equiv p'_1(u), \ldots, p_n :\equiv p'_n(u)\} \tag{7a}$$

$$A'_i :\equiv A_i\left\{\vec{p} :\equiv \overrightarrow{p'(u)}\right\} \tag{7b}$$

3.3. Some Major Properties of Type Theory of Acyclic Recursion

In this section, we shall present some of the properties of the type theory L^{λ}_{ar}, which are important for the algorithmic semantics of L^{λ}_{ar} and the L^{λ}_{ar}-terms.

Definition 3 (Irreducible Terms). We say that $A \in \text{Terms}$ is *irreducible* if and only if

$$\text{for all } B \in \text{Terms}, A \Rightarrow B \Longrightarrow A \equiv_c B \tag{8}$$

When $A \in \text{Terms}$ is explicit and irreducible, we use an abbreviation, e.g., by writing "A is e.i."

Theorem 1 (Criteria for Irreducibility). *(See Moschovakis (Moschovakis, 2006), § 3.12.)*

(1) If $A \in \text{Consts} \cup \text{Vars}$, then A is irreducible

(2) An application term $A(B)$ is irreducible if and only if A is explicit and irreducible and B is immediate

(3) A λ-term $\lambda(x)\,(A)$ is irreducible if and only if A is explicit and irreducible

(4) A recursion term A_0 where $\{p_1 := A_1, \ldots, p_n := A_n\}$ $(n \geq 0)$ is irreducible if and only if all of its parts $A_0, \ldots, A_n$ are explicit and irreducible

Proof. The proof is by structural induction on Def. 1 of the L^{λ}_{ar}-terms, by using the rules of the reduction calculus, defined in Sect. 3. See Theorem §3.12 in (Moschovakis, 2006). □

Theorem 2 (Canonical Form Theorem). *For more details, see (Moschovakis, 2006) and (Loukanova, 2019d; Loukanova, 2019c).*

For each term A, there is a unique, up to congruence, irreducible term $C \equiv_c \text{cf}(A)$ called the canonical form of A, such that:

(1) $\text{cf}(A) \equiv A_0$ where $\{p_1 := A_1, \ldots, p_n := A_n\}$ *for some explicit, irreducible terms $A_1, \ldots, A_n$ $(n \geq 0)$*

(2) $A \Rightarrow \text{cf}(A)$

(3) *(Uniqueness of the Canonical Form) If $A \Rightarrow B$ and B is irreducible, then $B \equiv_c \text{cf}(A)$, i.e., $\text{cf}(A)$ is the unique, up to congruence, irreducible term to which A can be reduced*

(4) $\text{FreeV}(A) = \text{FreeV}(\text{cf}(A))$

(5) *For every constant $c \in K$, c occurs in A iff c occurs in* $\text{cf}(A)$

Theorem 3. *(See (Moschovakis, 2006), § 3.11.) For any given* $\mathrm{L}_{ar}^{\lambda}$*-terms* $A, B \in$ Terms*:*

$$\text{if } A \Rightarrow B, \text{ then } \mathrm{den}(A) = \mathrm{den}(B) \tag{9a}$$

$$\mathrm{den}(A) = \mathrm{den}(\mathrm{cf}\,(A)) \tag{9b}$$

Proof. is long, by induction on the reduction relation induced by the reduction rules, see Sect. 3.2 and the denotation function. □

Note 4. The recursion operator designated by the operator constant where is one of the important binding operators, which has been used in different ways in Mathematics and Computer Science, taken for granted, by default, without formalisation.

We underline that the symbol where in the Moschovakis formal languages of recursion, including in the simply typed theory of acyclic recursion $\mathrm{L}_{ar}^{\lambda}$ is a specialised constant symbol, but different from the typed constants of the formal languages of $\mathrm{L}_{ar}^{\lambda}$ and L_{r}^{λ}. The constant symbol where has a specialised role for designating recursion operator. Similarly, the usual logic constants designate logic or computation operators, including the symbol λ designating binding, abstraction operator, and the binding operators of quantification.

The canonical forms have a distinguished feature that is part of their computational (algorithmic) role: they provide algorithmic patterns of semantic computations. The more general terms provide algorithmic patterns that consist of sub-terms with components that are recursion variables; the most basic assignments of recursion variables (of lowest ranks) provide the specific basic data that feeds-up the general computational patterns. The more general terms and sub-terms classify language expressions with respect to their semantics and determine the algorithms for computing the denotations of the expressions.

4. On the Algorithmic Semantics in the Theory of Acyclic Recursion

The notion of algorithmic semantics, i.e., algorithmic intension, in the languages of recursion, (Moschovakis, 2006), covers the essential, computational aspect of the concept of meaning. The *referential intension*, $\mathrm{int}(A)$, of a meaningful term A is the tuple of functions (a recursor) that is defined by the denotations $\mathrm{den}(A_i)$ ($i \in \{ 0, \ldots n \}$) of the parts (i.e., the head sub-term A_0 and of the terms A_1, …, A_n in the system of assignments) of its canonical form:

$$\mathrm{cf}(A) \equiv A_0 \text{ where } \{p_1 := A_1, \ldots, p_n := A_n\}$$

Intuitively, for each meaningful term A, the intension of A, $\mathrm{int}(A)$, is the *algorithm* for computing its denotation $\mathrm{den}(A)$. Two meaningful expressions are synonymous if their referential intensions are naturally isomorphic, i.e., they are the same algorithm. Thus, the algorithmic meaning of a meaningful term (i.e., its sense) is the information about how to compute its denotation step-by-step: a meaningful term carries instructions within its structure, which are revealed by its canonical form, for acquiring what they denote in a model. The canonical form $\mathrm{cf}(A)$ of a meaningful term A encodes its intension, i.e., the algorithm for computing its denotation, via: (1) the basic instructions (facts), which consist of $\{p_1 := A_1, \ldots, p_n := A_n\}$ and the head term A_0, which are needed for computing the denotation $\mathrm{den}(A)$, and (2) a terminating rank order of the recursive steps that compute each $\mathrm{den}(A_i)$, for $i \in \{ 0, \ldots, n \}$, for incremental computation of the denotation $\mathrm{den}(A) = \mathrm{den}(A_0)$.

Definition 4 (Algorithmic Equivalence in $\mathfrak{A}$). (In (Moschovakis, 2006), this is Theorem § 3.4, in an equivalent way, without discussion of the detailes here, because this paper does not depend strictly on them.) Two terms A, B, are algorithmically equivalent, $A \approx B$, with respect to a given semantic structure $\mathfrak{A}$, i.e., referentially synonymous in $\mathfrak{A}$, iff

(A) A and B are both immediate, or

(B) A and B are both proper

and there are explicit, irreducible terms (of appropriate types), $A_0, \ldots, A_n, B_0, \ldots, B_n$, $n \geq 0$, such that:

(1) $A \Rightarrow_{\text{cf}} A_0$ where $\{p_1 := A_1, \ldots, p_n := A_n\}$
(2) $B \Rightarrow_{\text{cf}} B_0$ where $\{p_2 := B_1, \ldots, p_n := B_n\}$
(3) (a) for every $x \in \text{PureV} \cup \text{RecV}$,

$$x \in \text{FreeV}(A_i) \quad \text{iff} \quad x \in \text{FreeV}(B_i), \quad \text{for } i \in \{0, \ldots, n\} \tag{10}$$

(b) for all $g \in G$:

$$\text{den}(A_i)(g) = \text{den}(B_i)(g), \quad \text{for } i \in \{0, \ldots, n\} \tag{11}$$

Informally, $A \approx B$ iff one of the cases (A) or (B) holds:

(A) when A and B do not have algorithmic senses, i.e., when A and B are both immediate, A and B have the same denotations (obtained immediately via variable valuations, since both have no algorithmic sense)

(B) otherwise, i.e., when A and B have algorithmic senses, i.e., A and B are both proper, their denotations (which are equal) are computed by the same algorithm represented by their canonical forms

In L_{ar}^{λ}, the β-conversion is valid only denotationally, as it is, normally applicable, in traditional λ-calculus. Detailed arguments against a general β-conversion for the algorithmic semantics, and special cases when it is valid are provided by (Loukanova, 2011b; Loukanova, 2019d; Loukanova, 2019c).

5. Classic Eta-Conversion in the Original Reduction Calculus

In what follows, when we refer to the denotational function, we assume that is a given, while arbitrary semantic structure, without any specific features.

The classic η-conversion *is not algorithmically valid*, in general in L_{ar}^{λ} (and in L_{r}^{λ}), according to the original reduction calculus. It *is valid only denotationally*, which is determined by the denotational function, i.e., (12a)–(12c) hold.

Theorem 4 (Denotational η-Conversion). *For all* $A \in$ Terms*, such that* (12a)*, the denotational* η*-conversions* (12b) *and* (12c) *are valid:*

$$x \notin \text{FreeV}(A) \tag{12a}$$

$$\mathrm{den}(\lambda(x)\ (A(x)))\ (g) = \mathrm{den}(A)\ (g), \textit{ for all variable valuations } g \in G \textit{ in } \mathfrak{A} \tag{12b}$$

$$\mathrm{den}(\lambda(x)\ (A(x))) = \mathrm{den}(A) \tag{12c}$$

Proof. The denotational equivalence (12b) is proved by structural induction on the formation rules of A, see Def. 1. The denotational equality (12c) follows from (12b), for all valuations $g \in G$ in $\mathfrak{A}$. □

6. The Restricted Eta-Rule in Type-Theory of Recursion

In general, the algorithmic η-conversion for recursion terms is not valid. In this session, we shall use a counterexample to demonstrate it.

6.1. The Motivation of a New, Algorithmic Eta-Rule

In Sect. 6.2, we shall introduce an η-rule. Then, in the rest of the paper, we present the corresponding extended reduction calculus and its properties.

Motivation from Computational Semantics of Natural Language We present a typical example from human language, which while simple, represents a pattern for a large class of expressions with varieties of similar phenomena. We consider this example as a strong motivation for the usefulness of additional reduction rules, similar to the η-rule introduced in this paper.

Example 6.1. The detailed steps of rendering a sentence, such as (13) into a term A, and a reduction of A to its canonical form $\mathrm{cf}(A)$ in $\mathrm{L}^{\lambda}_{ar}$, as in (14a)–(14d), are given in (Loukanova, 2019c). Canonical forms similar to $\mathrm{cf}(A)$ in (14a)–(14d), having assignments as in (14d), provide a motivation for the γ-rule and the induced by it γ-reduction. The γ-rule and its γ-reductions are more general and complex than the η-rule and the corresponding η-reduction. The same example is handled similarly by the simpler η-rule, which applies only to terms in canonical forms.

$$\text{Kim hugs some dog} \xrightarrow{\text{render}} A \tag{13}$$

By applying the rules of the $\mathrm{L}^{\lambda}_{ar}$-reduction calculus, the term A reduces to its canonical form $\mathrm{cf}(A)$ in (14a)–(14d):

$$A \Rightarrow \mathrm{cf}(A) \equiv_{\mathrm{c}} [\lambda y_k\ (\mathit{some}(d(y_k))\,(h(y_k)))]\ (k) \text{ where} \tag{14a}$$

$$\{\ k := \mathit{kim}, \tag{14b}$$

$$h := \lambda y_k \lambda x_d\ \mathit{hugs}(x_d)(y_k), \tag{14c}$$

$$d := \lambda y_k\ \mathit{dog}\ \} \tag{14d}$$

Then, by the Algorithmic Equivalence Definition 4, it follows that $\mathrm{cf}(A)$ is not algorithmically equivalent to the term B in (15b)–(15e). Therefore, $\mathrm{cf}(A)$ is not equivalent to any term B' that is congruent to the term B:

$$\mathrm{cf(A)} \neq B \equiv_{\mathrm{c}} B' \tag{15a}$$

$$B \equiv [\lambda y_k\ \mathit{some}(d')(h(y_k))](k) \text{ where} \tag{15b}$$

$$\{\ k := \mathit{kim}, \tag{15c}$$

$$h := \lambda y_k \lambda x_d \; hugs(x_d)(y_k), \tag{15d}$$

$$d' := dog \; \} \tag{15e}$$

The term B in (15b)–(15e) is in a canonical form, but it is not algorithmically equivalent (synonymous) to the term (14a)–(14d), by the L^{λ}_{ar}-reduction calculus and the Algorithmic Equivalence Definition 4. This is because the term parts in (14d) and (15e) are not denotationally equivalent, i.e.:

$$\text{den}\,(\lambda y_k \; dog) \neq \text{den}\,(dog) \tag{16}$$

That is, the two terms (14a)–(14d) and (15b)–(15e) are not algorithmically equivalent with respect to the strictest notion of algorithm introduced in (Moschovakis, 2006).

6.2. A Restricted Eta-Rule in Type Theory of Acyclic Recursion

Definition 5 (η-condition). Any given recursion term $A \in$ Terms *satisfies the* η*-condition with respect to an assignment* $p := \lambda(v)P$ in its where-scope, if and only if the clauses (C1)–(C3) hold:[1]

(C1) A is of the form (17), i.e., A is a recursion term in a canonical form:

$$A \equiv A_0 \text{where} \left\{ \vec{a} := \vec{A},\; p := \lambda\left(v\right)P,\; \vec{b} := \vec{B} \right\} \equiv_c \text{cf}\left(A\right) \tag{17}$$

(C2) The term $P \in \text{Terms}_{\tau}$, in $p := \lambda(v)P$, does not have any free occurrences of v in it, i.e., $v \notin \text{FreeV(P)}$

(C3) Each of the free occurrences of p in any of the term parts A_0, A_i, B_j, i.e., in A_0, $\vec{A}$ and $\vec{B}$, is an occurrence in a subterm $p(v)$ that is in the scope of $\lambda(v)$ (modulo congruence with respect to renaming the variable v)

for:

(P1) $v \in \text{PureV}_{\sigma}$, $p \in \text{RecV}_{(\sigma \to \tau)}$

(P2) $P \in \text{Terms}_{\tau}$, and thus, $\lambda(v)\, P \in \text{Terms}_{(\sigma \to \tau)}$

(P3) $A_0, \vec{A} \equiv A_1, \ldots, A_n \in$ Terms, $\vec{a} \equiv a_1, \ldots, a_n \in$ RecV $(n \geq 0)$, of correspondingly matching types

(P4) $\vec{B} \equiv B_1, \ldots, B_k \in$ Terms, $\vec{b} \equiv b_1, \ldots, b_k \in$ RecV$(k \geq 0)$, of correspondingly matching types

We say that the assignment $p := \lambda(v)P$ and the memory variable p satisfy the η-condition for A, given in Def. 5, and that A *satisfies the* η*-condition* for p in its where-scope.

The (η) **Rule** For any $A \in$ Terms (18a), i.e., (18b) ($n \geq 0$, $k \geq 0$), which satisfies the η-condition in Def. 5, with respect to the assignment $p := \lambda v P$ in its where-scope

[1]One may also add the restriction: The recursion variable p occurs in at least one part of A, i.e., in at least one of the terms A_0, $\vec{A}$, $\vec{B}$. We shall refrain from such a version of the η-condition and its corresponding η-rule in this paper.

$$A \equiv \mathrm{cf}(A) \equiv A_0 \text{ where } \{\, p_1 := A_1, \ldots, p_n := A_n, \quad p := \lambda v P, \quad q_1 := B_1, \ldots, q_k := B_k \,\} \tag{18a}$$

$$\equiv A_0 \text{where} \left\{ \vec{p} := \vec{A},\ p := \lambda v P,\ \vec{q} := \vec{B} \right\} \tag{18b}$$

the (η) rule is the following one-step reduction applied to A:

$$\begin{aligned} A \equiv \mathrm{cf}(A) \equiv A_0 \text{where} \left\{ \vec{p} := \vec{A},\ p := \lambda v P,\ \vec{q} := \vec{B} \right\} \\ \Rightarrow_\eta A_0 \left\{ p(v) :\equiv p' \right\} \text{where} \left\{ \vec{p} := \vec{A} \left\{ p(v) :\equiv p' \right\}, \right. \\ p' := P, \\ \left. \left. \vec{q} := \vec{B} \left\{ p(v) :\equiv p' \right\} \right\} \right. \end{aligned} \tag{η}$$

given that:

(R1) $p' \in \mathrm{RecV}_\tau$ is a fresh recursion variable for A, i.e.:

$$p' \notin \mathrm{Vars}(A) \tag{19}$$

(R2) $\vec{X}\{p(v) :\equiv p'\}$ is the sequence of the terms that are the result of the replacements $X_i\{\, p(v) :\equiv p' \,\}$, of all occurrences of $p(v)$ with p', in all terms X_i, for $X_i \in \{\, A_i \mid i = 0, \ldots, n \,\} \cup \{\, B_i \mid i = 1, \ldots, k \}$, modulo congruence with respect to renaming the bound occurrences of the scope variable v:

$$X_i\{\, p(v) :\equiv p' \,\} \text{ is replacement modulo renaming bound } v \tag{20}$$

Note 5. In the η-rule and its applications in reductions, for all $i \in \{0, \ldots, n\}$ and $j \in \{\, 0, \ldots, k\}$, the replacements $A_i\{p(v) :\equiv p'\}$ and $B_j\{p(v) :\equiv p'\}$, are such that the occurrences of the term $p(v)$ in A_i and B_j are within the scope of the abstraction λv, modulo appropriate renaming of v. By this condition (C3), the (η) rule maintains the pure, free variables of the reduced terms:

$$\text{if } A \Rightarrow_\eta B, \text{ then } \mathrm{FreeV}(A) = \mathrm{FreeV}(B) \tag{21}$$

Example 6.2. Assume that $p, p' \in \mathrm{RecV}$, $x \in \mathrm{PureV}$, p' is fresh for A, $C \in \mathrm{Terms}$, $x \notin \mathrm{FreeV}(C)$, C is explicit, irreducible. We assume that these variables and terms are of suitable types. For example, $book \in \mathrm{Consts}_{(\tilde{e} \to \tilde{\iota})}$. For instance, $C \equiv book$. Then, $\lambda(x)(C) \equiv \lambda(x)(book)$, We may need to reduce such vacuous λ-abstractions, and corresponding applications: $[\lambda(x)(C)](y) \equiv [\lambda(x)(book)](y)$:

$$[\lambda(x)(C)](y) \neq C, \qquad \text{for } x, y \notin \mathrm{FreeV}(C),\ y \in \mathrm{FreeV}([\lambda(x)(C)](y)) \tag{22a}$$

$$[\lambda(x)(C)](x) \neq C, \qquad \text{for } x \notin \mathrm{FreeV}(C),\ x \in \mathrm{FreeV}([\lambda(x)(C)](x)) \tag{22b}$$

$$[\lambda(x)(book)](y) \neq book, \qquad y \in \mathrm{FreeV}([\lambda(x)(book)](y)),\ y \notin \mathrm{FreeV}(book) \tag{22c}$$

$$\lambda(x)(book(x)) \approx book, \qquad x \notin \mathrm{FreeV}(\lambda(x)(book)(x))),\ x \notin \mathrm{FreeV}(book)$$
$$\mathrm{den}(\lambda(x)\,(book(x))) = \mathrm{den}(book) \tag{22d}$$

by Algorithmic Equivalence Definition 4, (10)

Example 6.3. Assume that $p, p' \in$ RecV, $x \in$ PureV, p' is fresh for $A, C \in$ Terms, $x \notin$ FreeV(C), and C is explicit, irreducible, which are of suitable types. For example, $C \in$ Consts.

$$\lambda(x)\,[p_2 \text{ where } \{p_2 := P\,(p_1(x)),\, p_1 := C\}] \tag{23a}$$
$$\Rightarrow \lambda(x)(p_2'(x)) \text{ where } \{\, p_2' := \lambda(x)[P\,(p_1'(x)(x))],\, p_1' := \lambda(x)(C)\,\} \quad \text{by } (\lambda) \tag{23b}$$
$$\Rightarrow_\eta \lambda(x)(p_2'(x)) \text{ where } \{\, p_2' := \lambda(x)[P\,(p_1''(x))],\, p_1'' := C\,\} \quad \text{by } (\eta) \tag{23c}$$
$$\Rightarrow \lambda(x)(p_2'(x)) \text{ where } \{\, p_2' := \lambda(x)[P\,(p_1(x))],\, p_1 := C\,\} \quad \text{by (cong)} \tag{23d}$$
$$\approx p_2' \text{ where } \{\, p_2' := \lambda(x)[P\,(p_1\,(x))],\, p_1 := C\,\} \text{ by Algorithmic Equivalence Definition 4} \tag{23e}$$

Example 6.4. Assume that $book \in$ Consts, $C \in$ Terms, $x \notin$ FreeV(C), and C is explicit, irreducible. Then (η) rule is not applicable to A, B, D, E, since by Def. 5, the condition (C3) is not satisfied:

$$A \equiv p \text{ where } \{\, p := \lambda(x)book\,\} \tag{24a}$$
$$B \equiv p \text{ where } \{\, p := \lambda(x)C\,\} \tag{24b}$$
$$D \equiv p(v) \text{ where } \{\, p := \lambda(x)book\,\}, \qquad v \in \text{FreeV}(D) \tag{24c}$$
$$E \equiv p(v) \text{ where } \{\, p := \lambda(x)C\,\}, \qquad v \in \text{FreeV}(E) \tag{24d}$$

Without requiring $p(v)$ to be in the scope of binding $\lambda(v)$, by the condition (C3), in Def. 5, a tentative application of the (η), would remove v from the free variables of the reduced term:

$$E \Rightarrow_\eta E' \equiv p' \text{ where } \{\, p' := C\,\},\, v \in \text{FreeV}(E),\, v \notin \text{FreeV}(E') \tag{25}$$

Depending on applications of L_{ar}^{λ}, this conditions can be adjusted, by releasing the binding requirement.

Example 6.5. Assume that $p, q \in$ PureV, $p \neq q$, $book \in$ Consts, $C \in$ Terms, $x \notin$ FreeV(C), and C is explicit, irreducible. Then, by Def. 5, the conditions, including (C3), are satisfied for the (η) rule:

$$A \equiv q \text{ where } \{\, p := \lambda(x)\, book\,\} \Rightarrow_\eta q \text{ where } \{p := book\} \tag{26a}$$
$$B \equiv q \text{ where } \{\, p := \lambda(x)C\,\} \Rightarrow_\eta q \text{ where } \{p := C\} \tag{26b}$$

6.3. Denotational Equivalence of Canonical Forms Reduced by Eta-Rule

In this section, we shall consider terms in canonical forms.

Theorem 5 (Denotational Equivalence by η-rule). *Let $A \in$ Terms be a term in a canonical form:*

$$A \equiv \text{cf}\left(A\right) \equiv A_0 \text{ where } \left\{\vec{p} := \vec{A},\, p_{n+1} := \lambda\left(v\right)A_{n+1},\, \vec{q} := \vec{B}\right\} \tag{27}$$

($n \geq 0$), such that:

(P1) $v : \sigma$ is a pure variable and $p_{n+1} : (\sigma \to \tau)$ is a recursion variable.

(P2) The explicit, irreducible term $A_{n+1} : \tau$ does not have any (free) occurrences of v (and p_{n+1})

(P3) All the occurrences of p_{n+1}, in A_0, $\vec{A}$, and $\vec{B}$, are occurrences of the term $p_{n+1}(v)$, which are in the scope of $\lambda(v)$ (modulo appropriate renaming of v)

Let $p'_{n+1} : \tau$ *be a fresh recursion variable, and* A' *be the term as in* (28d)–(28g) *(by the* η*-rule):*

$$A \equiv \Big[A_0 \text{where} \Big\{ \vec{p} := \vec{A} \tag{28a}$$

$$p_{n+1} := \lambda\big(v\big) A_{n+1}, \tag{28b}$$

$$\vec{q} := \vec{B} \Big\} \Big] \tag{28c}$$

$$\Rightarrow_\eta A' \equiv \Big[A_0 \Big\{ p_{n+1}\big(v\big) :\equiv p'_{n+1} \Big\} \text{where} \Big\{ \tag{28d}$$

$$\vec{p} := \vec{A} \Big\{ p_{n+1}\big(v\big) :\equiv p'_{n+1} \Big\}, \tag{28e}$$

$$p'_{n+1} := A_{n+1}, \tag{28f}$$

$$\vec{q} := \vec{B} \Big\{ p_{n+1}\big(v\big) :\equiv p'_{n+1} \Big\} \Big\} \Big] \tag{28g}$$

Then,

$$A \neq A' \tag{29a}$$

$$\text{cf}(A') \equiv_c A' \tag{29b}$$

and, for all $g \in G$, $i \in \{0, \ldots, n\}$, *and* $j \in \{1, \ldots, k\}$, *the following denotational equalities hold:*

$$\text{den}(A)(g) = \text{den}(A')(g) \tag{30}$$

Proof. It is very long and we do not include it in this paper. It will be provided in a separate paper on the mathematical properties of η-reduction. □

Note 6. While the equality (30) is about denotations, its proof is not based on the traditional β-conversion over application terms without recursion assignments.

Notice that there are no other syntactical manipulations over the terms A and A', except for the replacements used according to the (η) rule in $A \Rightarrow_\eta A'$. The term A' designates an algorithm that is very close to the algorithm by A, simplified by reducing the needless λ-abstractions and corresponding functional applications.

7. Eta-Reduction in the Theory of Acyclic Recursion

In this section, the reduction calculus of L_{ar}^{λ} was extended by adding the (η) rule to the set of the rules given in Sect. 3.

Definition 6 (η-reduction). The *η-reduction* relation in L_{ar}^{λ} is the smallest relation $\Rightarrow_\eta^*$ between L_{ar}^{λ}-terms, such that:

(1) For any L_{ar}^{λ}-terms $A, B \in$ Terms:

$$\text{if } \underbrace{\text{cf}\big(A\big) \Rightarrow_\eta B}_{\text{by}(\eta)\text{rule}}, \text{ then } A \Rightarrow_\eta^* B \tag{η-red}$$

(2) The relation $\Rightarrow^*_\eta$, between L^λ_{ar}-terms is closed under the reduction rules of L^λ_{ar}, including the original ones, see Sect. 3.2, and (η), i.e., $\Rightarrow^*_\eta$ is closed under transitivity, congruence, compositionality with respect to term formation rules, (head), (B-S), (recap), (ap), (λ), and (η).

Often, we write $A \Rightarrow_\eta B$ instead of $A \Rightarrow^*_\eta B$, when the (η) rule has been applied at least once.

Theorem 6. *For any* $A, B \in$ Terms*:*

$$\textit{if } A \Rightarrow_{\text{cf}} \underbrace{\text{cf}\left(A\right) \Rightarrow_\eta B}_{\text{by}(\eta)\text{rule}}, \textit{ then } \text{den}(A) = \text{den}(\text{cf}\,(A)) = \text{den}(B) \tag{31}$$

Proof. The denotational equality den(A) = den(cf (A)) is by Theorem 3. The equality den(cf (A)) = den(B) is by Theorem 5. □

Definition 7 (η-Irreducibility). $A \in$ Terms is *η-irreducible* iff it is irreducible, see Def 3, and does not satisfies the η-condition Def. 5.

Definition 8 (η-Equivalence Relation $\approx_\eta$). For all terms $A, B \in$ Terms

$$A \approx_\eta B \Longleftrightarrow \text{for some } C, \quad A \Rightarrow^*_\eta C \text{ and } C \approx B \tag{32}$$

Corollary 6.1. *For any* L^λ_{ar}*-terms A and B,*

(1) if $A \Rightarrow^*_\eta B$*, then* den(A) = den(B)

(2) if $A \approx_\eta B$*, then* den(A) = den(B)

Proof. (1) is proven by induction on the number of the applications of the (η) rule and Theorem 6. Then, (2) follows by Def. 8 of η-equivalence relation $\approx_\eta$. □

Definition 9 (Syntactic Equivalence $\approx_s$). For any L^λ_{ar}-terms A and B

$$\text{A} \approx_s \text{B} \iff \text{cf}(A) \equiv_c \text{cf}(B) \tag{33}$$

Some details about syntactic equivalence (synonymy) are given in (Moschovakis, 2006) and (Loukanova, 2019d; Loukanova, 2019c). The difference is that the syntactic synonymy does not qualify for algorithmic equivalence of denotationally equal constants and syntactic constructs of λ-abstraction. For instance, assuming that *dog* is a constant, then den(*dog*) = den($\lambda(x)$*dog*(x)) (by the denotation function); *dog* $\approx \lambda(x)$*dog*(x) (by the Algorithmic Equivalence Definition 4, because both terms are in canonical forms); *dog* $\not\approx_s \lambda(x)$*dog*(x).

Corollary 6.2. *For any two* L^λ_{ar}*-terms* $A, B \in$ Terms*:*

$$A \Rightarrow B \Longrightarrow A \approx_s B \Longrightarrow \text{cf}(A) \equiv_c \text{cf}(B) \Longrightarrow A \approx_\eta B \Longrightarrow \text{den}(A) = \text{den}(B) \tag{34}$$

Proof. Follows from the definitions. □

Corollary 6.3. *There exist (many)* $\mathrm{L}^{\lambda}_{ar}$*-terms* $A, B, C, E \in$ Terms*, such that:*

$$A \Rightarrow B \Rightarrow^{*}_{\eta} C \implies A \approx_{\eta} B \approx_{\eta} C \tag{35a}$$

$$\text{while } C \neq B \text{ and } C \neq A \tag{35b}$$

$$B \approx E, \text{ while } C \neq B \text{ and } C \neq E \tag{35c}$$

8. Algorithmic Eta-Conversion

Theorem 7 (Algorithmic η-Conversion of Explicit, Irreducible Terms). *For every* $C \in$ Terms*, the algorithmic equivalences* (36a)–(36b) *are valid, given that the restrictions* (36c), (36d), (36e) *hold:*

$$\lambda(x)\ (C(x)) \approx C \qquad \textit{(Algorithmic Equivalence)} \tag{36a}$$

$$\lambda(x)\ C(x) \approx_{\eta} C \qquad \textit{(Restricted Algorithmic Eta-Conversion)} \tag{36b}$$

for $C \in$ Terms*, such that:*

$$C \text{ is explicit, irreducible} \tag{36c}$$

$$x \notin \mathrm{FreeV}(C) \tag{36d}$$

$$\lambda(x)(C\ (x)) \text{ and } C \text{ are both immediate or both proper} \tag{36e}$$

Proof. Note that $\mathrm{FreeV}(\lambda(x)(C\ (x))) = \mathrm{FreeV}(C)$ follows by the definitions of free and bound variables. The denotational equality (37), which is a denotational η-conversion, follows from the definitions of the denotational function.

$$\mathrm{den}\ (\lambda(x)\ (C(x)))\ (g) = \mathrm{den}(C)\ (g), \quad \text{for every variable valuation } g \in G \tag{37}$$

These definitions are provided, e.g., in (Moschovakis, 2006) and (Loukanova, 2019c).

By the Criteria for Irreducibility, Theorem 1, since $C \in$ Terms is explicit, irreducible, the term $\lambda(x)\ C(x)$ is also explicit, irreducible. Thus, by the Canonical Form Theorem 2, both terms C and $\lambda(x)\ (C(x))$ are in canonical forms, i.e., (38a)–(38b) hold:

$$\mathrm{cf}\ (C) \equiv C \tag{38a}$$

$$\mathrm{cf}\ (\lambda(x)\ (C(x))) \equiv \lambda(x)\ (C(x)) \tag{38b}$$

The algorithmic equivalence (36a) follows from (38a)–(38b), by Definition 4. The restricted algorithmic η- conversion (36b) follows from Def. 8 of the η-equivalence relation $\approx_{\eta}$. □

Note 7 (Generalised, algorithmic (η) rule of $\mathrm{L}^{\lambda}_{ar}$). The ($\eta$) rule presented in this paper is a generalisation of the denotational version of the standard η-rule of traditional λ-calculi. Here, in $\mathrm{L}^{\lambda}_{ar}$, the algorithmic ($\eta$) rule is applicable across recursion assignments, in recursion terms in canonical forms. In simple cases, the (η) rule can be applied as in Lemma 1.

Lemma 1. *Assume that* $D_0 \in$ Terms *is in canonical form, such that* (39) *and (1) hold:*

$$D_0 \equiv \mathrm{cf}(D_0) \equiv \lambda(v)(p(v)) \text{ where } \{\ p := \lambda(v)A\ \} \tag{39}$$

(1) $A \in$ Terms *does not have any free occurrences of the pure variable* v *(and of* p*, by the acyclicity), i.e.,* $v \notin \text{FreeV}(A)$

Then, the (η) *rule can be applied as in* (40a)–(40b)*:*

$$D_0 \equiv \lambda(v)(p(v)) \text{ where } \{\ p := \lambda(v)A\ \} \tag{40a}$$

$$\Rightarrow_\eta D_0' \equiv \lambda(v)(p') \text{ where } \{\ p' := A\ \} \tag{40b}$$

Proof. By (39) and (1), the term D_0 satisfies the condition for applicability of the η-rule, according to Def. 5. The (η) rule can be applied as in (40a)–(40b). □

9. Conclusions and Future Work

In this paper, we have introduced the new η-rule in L^{λ}_{ar} to simplify recursion terms in canonical forms, by removing occurrences of vacuous λ-abstractions and reducing the corresponding functional applications. The η-rule closely preserves all other structural components of the canonical terms, and by that, the algorithmic steps in the computations of denotations. The denotations of the reduced terms are strictly preserved.

We have demonstrated that a restricted version of the algorithmic η-conversion, applicable to canonical forms, is valid in L^{λ}_{ar} theory, by using the introduced in this paper algorithmic η-rule in the theory of acyclic recursion L^{λ}_{ar}.

This paper is part of an extended work on the developments of type theory of parametric algorithms and applications.

Theoretical Development In a separate work, we shall present details of various mathematical properties of the extended η-reduction and proofs of the properties presented here, e.g., of Lemma 1.

Applications The η-rule maximally preserves the algorithmic computations of the term to which it applies, while reducing computationally needless λ-abstractions $p := \lambda(v)A$ and subsequent applications $\lambda(v)(p(v))$.

Theorem 5, which is extended to the $\approx_\eta$ equivalence by Corollary 6.1, shows that the $\approx_\eta$ equivalence is one of the many equivalence relations between terms, which is stronger than denotational equality and weaker than the strict algorithmic synonymy in L^{λ}_{ar} focussed on specific semantic structures $\mathfrak{A}$.

Applications to Computational Grammar of Natural Language The presented η-rule has applications to computational semantics of human languages and to semantics of programs and compilers.

In the analyses of certain classes of human language expressions, which we have been covering in recent work, the η-rule provides simplification of canonical terms that are otherwise irreducible by the L^{λ}_{ar}-reduction calculus. In particular, the η-reduction is a simplified expression of the innermost γ-reduction introduced in (Loukanova, 2019c). Thus, for applications of the L^{λ}_{ar}, the η-reduction is easier to apply, directly to the canonical forms of the sub-terms, from inside-out, and combine them, by the compositionality rules, (ap-comp), (λ-comp), (wh-comp).

The reduction calculus of L^{λ}_{ar} extended to η-reduction is useful in applications, especially for reducing algorithmic complexity, by the algorithmic η-conversion across recursion assignments.

In computational grammar, by covering syntax-semantics interfaces, the η-canonical forms of subterms can be combined together compositionally, from the η-canonical forms of the renderings of subexpressions, e.g., as in (Loukanova, 2019a).

The extended η-reduction is useful for various tasks, e.g., in translations between natural language expressions and generation of natural language from semantic representations.

10. Acknowledgements

The work in this paper is an essential extension of (Loukanova, 2020), which introduces the η-rule, for the purpose of applying it to semantic representations of human language, without looking into its properties and details of reductions. In this extended paper, we reformulate the (η) rule. Then we present some of the theoretical properties of the η-rule, and the induced η-reduction calculus, with respect to existing and potential applications of $\mathrm{L}^{\lambda}_{ar}$ to the areas of Artificial Intelligence (AI), especially AI technologies evolving syntax-semantics interfaces in formal and natural languages.

11. Vitae

Roussanka Loukanova has a PhD degree in mathematics from Moscow State University. She has been teaching at Sofia University (Bulgaria), Indiana University Bloomington (US), University of Minnesota (US), Illinois Wesleyan University (US), Uppsala and Stockholm Universities (Sweden). Her research is in the areas of Typed Theory of Situated Information, Type Theory of Algorithms, Computational Syntax, Computational Semantics, and Computational Syntax-Semantics Interface. Currently, she is senior assistant professor in mathematics at the Department of Algebra and Logic, Institute of Mathematics and Informatics (IMI), Bulgarian Academy of Sciences (BAS), Sofia, Bulgaria.

12. References

Duží, M., 2019. If structured propositions are logical procedures then how are procedures individuated? *Synthese*, 196(4):1249–1283.

Duží, M. and Kosterec, M., 2017. A Valid Rule of β-conversion for the Logic of Partial Functions. *Organon F*, 24(1):10–36.

Duží, M. and Jespersen, B., 2012. Procedural isomorphism, analytic information and β-conversion by value. *Logic Journal of the IGPL*, 21(2):291–308. ISSN 1367-0751. doi:10.1093/jigpal/jzs044.

Gallin, D., 1975. *Intensional and Higher-Order Modal Logic: With Applications to Montague Semantics*. North-Holland Publishing Company, Amsterdam and Oxford, and American Elsevier Publishing Company. ISBN 0 7204 0360 X.

Loukanova, R., 2001. Russellian and Strawsonian Definite Descriptions in Situation Semantics. In Gelbukh, A., editor, *Computational Linguistics and Intelligent Text Processing. CICLing 2001*, volume 2004 of *Lecture Notes in Computer Science*, pages 69–79. Springer, Berlin, Heidelberg. ISBN 978-3-540-44686-6.

Loukanova, R., 2002a. Generalized Quantification in Situation Semantics. In Gelbukh, A., editor, *Computational Linguistics and Intelligent Text Processing*, volume 2276 of *Lecture Notes in Computer Science*, pages 46–57. Springer, Berlin, Heidelberg. ISBN 978-3-540-45715-2.

Loukanova, R., 2002b. Quantification and Intensionality in Situation Semantics. In Gelbukh, A., editor, *Computational Linguistics and Intelligent Text Processing. CICLing 2002*, volume 2276 of *Lecture Notes in Computer Science*, pages 32–45. Springer, Berlin, Heidelberg. ISBN 978-3-540-45715-2.

Loukanova, R., 2009. β-Reduction and Antecedent-Anaphora Relations in the Language of Acyclic Recursion. In Cabestany, J., Sandoval, F., Prieto, A., and Rodríguez, J. M. C., editors, *Bio-Inspired Systems: Computational and Ambient Intelligence. IWANN 2009*, volume 5517 of *Lecture Notes in Computer Science*, pages 496–503. Springer, Berlin, Heidelberg, Salamanca, Spain. doi:10.1007/978-3-642-02478-8.

Loukanova, R., 2011a. Modeling Context Information for Computational Semantics with the Language of Acyclic Recursion. In Pérez, J. B., Corchado, J. M., Moreno, M., Julián, V., Mathieu, P., Canada-Bago, J., Ortega, A., and Fernández-Caballero, A., editors, *Highlights in Practical Applications of Agents and Multiagent Systems*, volume 89 of *Advances in Intelligent and Soft Computing*, pages 265–274. Springer Berlin Heidelberg.

Loukanova, R., 2011b. Reference, Co-reference and Antecedent-anaphora in the Type Theory of Acyclic Recursion. In Bel-Enguix, G. and Jiménez-López, M. D., editors, *Bio-Inspired Models for Natural and Formal Languages*, pages 81–102. Cambridge Scholars Publishing.

Loukanova, R., 2019a. Computational Syntax-Semantics Interface with Type-Theory of Acyclic Recursion for Underspecified Semantics. In Osswald, R., Retoré, C., and Sutton, P., editors, *IWCS 2019 Workshop on Computing Semantics with Types, Frames and Related Structures. Proceedings of the Workshop*, pages 37–48. The Association for Computational Linguistics (ACL), Gothenburg, Sweden.

Loukanova, R., 2019b. Formalisation of Situated Dependent-Type Theory with Underspecified Assessments. In Bucciarelli, E., Chen, S.-H., and Corchado, J. M., editors, *Decision Economics. Designs, Models, and Techniques for Boundedly Rational Decisions. DCAI 2018*, volume 805 of *Advances in Intelligent Systems and Computing*, pages 49–56. Springer International Publishing, Cham. ISBN 978-3-319-99698-1.

Loukanova, R., 2019c. Gamma-Reduction in Type Theory of Acyclic Recursion. *Fundamenta Informaticae*, 170(4):367–411. ISSN 0169-2968 (P) 1875-8681 (E).

Loukanova, R., 2019d. Gamma-Star Canonical Forms in the Type-Theory of Acyclic Algorithms. In van den Herik, J. and Rocha, A. P., editors, *Agents and Artificial Intelligence*, pages 383–407. Springer International Publishing, Cham.

Loukanova, R., 2020. Algorithmic Eta-Reduction in Type-Theory of Acyclic Recursion. In Rocha, A., Steels, L., and van den Herik, J., editors, *Proceedings of the 12th International Conference on Agents and Artificial Intelligence (ICAART 2020)*, volume 2, pages 1003–1010. INSTICC, CITEPRESS – Science and Technology Publications, Lda. ISBN 978-989-758-395-7.

Loukanova, R., 2023. Algorithmic Dependent-Type Theory of Situated Information and Context Assessments. In Omatu, S., Mehmood, R., Sitek, P., Cicerone, S., and Rodríguez, S., editors, *Distributed Computing and Artificial Intelligence, 19th International Conference*, volume 583, pages 31–41. Springer International Publishing, Cham. ISBN 978-3-031-20859-1. doi:10.1007/978-3-031-20859-1_4.

Moschovakis, Y. N., 1989. The formal language of recursion. *Journal of Symbolic Logic*, 54(4):1216–1252.

Moschovakis, Y. N., 1993. Sense and denotation as algorithm and value. In Oikkonen, J. and Väänänen, J., editors, *Logic Colloquium '90: ASL Summer Meeting in Helsinki*, volume Volume 2 of *Lecture Notes in Logic*, pages 210–249. Springer-Verlag, Berlin.

Moschovakis, Y. N., 1997. The logic of functional recursion. In Dalla Chiara, M. L., Doets, K., Mundici, D., and van Benthem, J., editors, *Logic and Scientific Methods*, volume 259, pages 179–207. Springer, Dordrecht.

Moschovakis, Y. N., 2006. A Logical Calculus of Meaning and Synonymy. *Linguistics and Philosophy*, 29(1):27–89. ISSN 1573-0549.

Plotkin, G., 1975. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(2):125–159. ISSN 0304-3975. doi:https://doi.org/10.1016/0304-3975(75)90017-1.

Plotkin, G. D., 1977. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–255.

Thomason, R. H., editor, 1974. *Formal Philosophy: Selected Papers of Richard Montague*. Yale University Press, New Haven, Connecticut.

Tichý, P., 1988. The foundations of Frege's logic. In *The Foundations of Frege's Logic*. Berlin: De Gruyter.

Tichý, P., 1995. Constructions as the Subject Matter of Mathematics. In Depauli-Schimanovich, W., Köhler, E., and Stadler, F., editors, *The Foundational Debate: Complexity and Constructivity in Mathematics and Physics*, pages 175–185. Springer Netherlands, Dordrecht. ISBN 978-94-017-3327-4. doi:10.1007/ 978-94-017-3327-4_13.